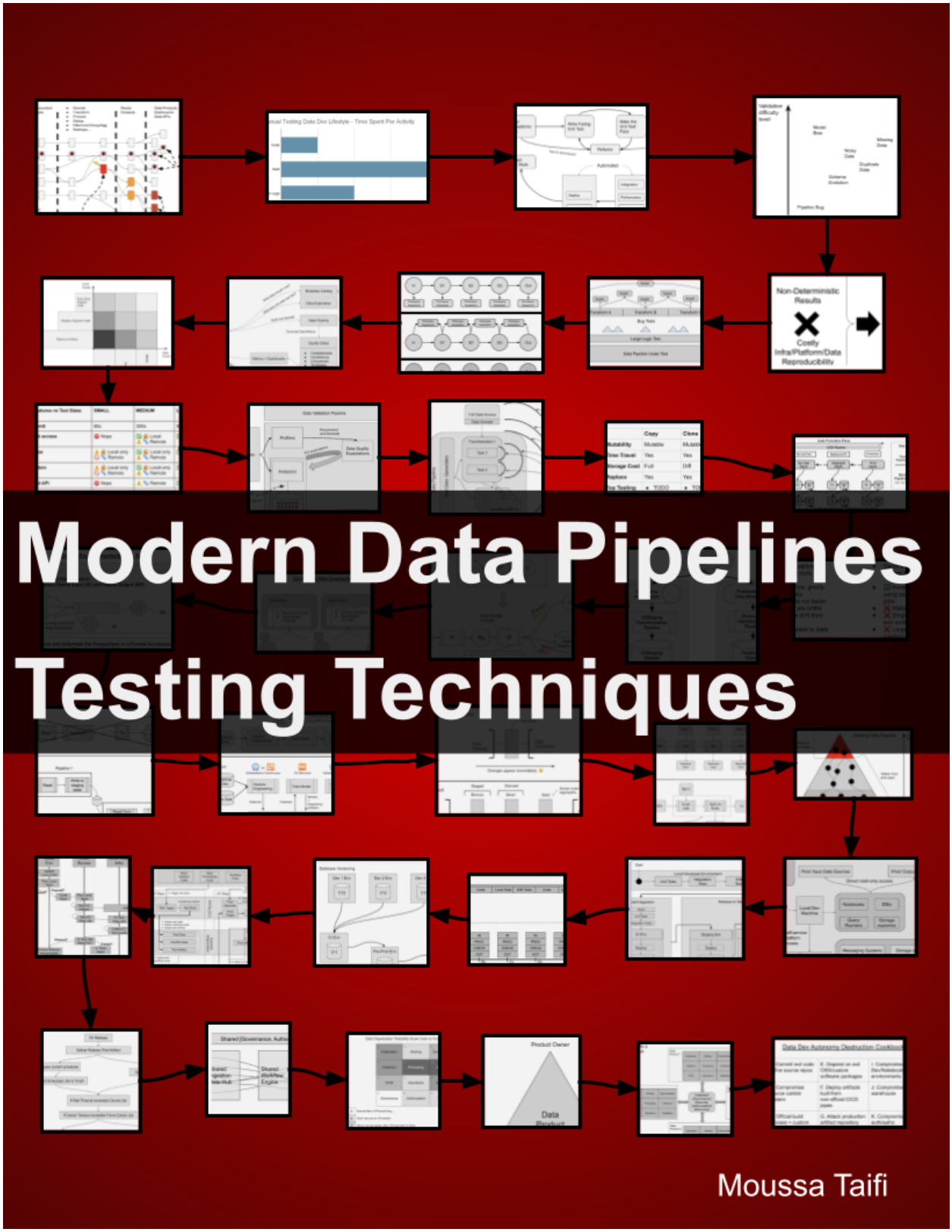


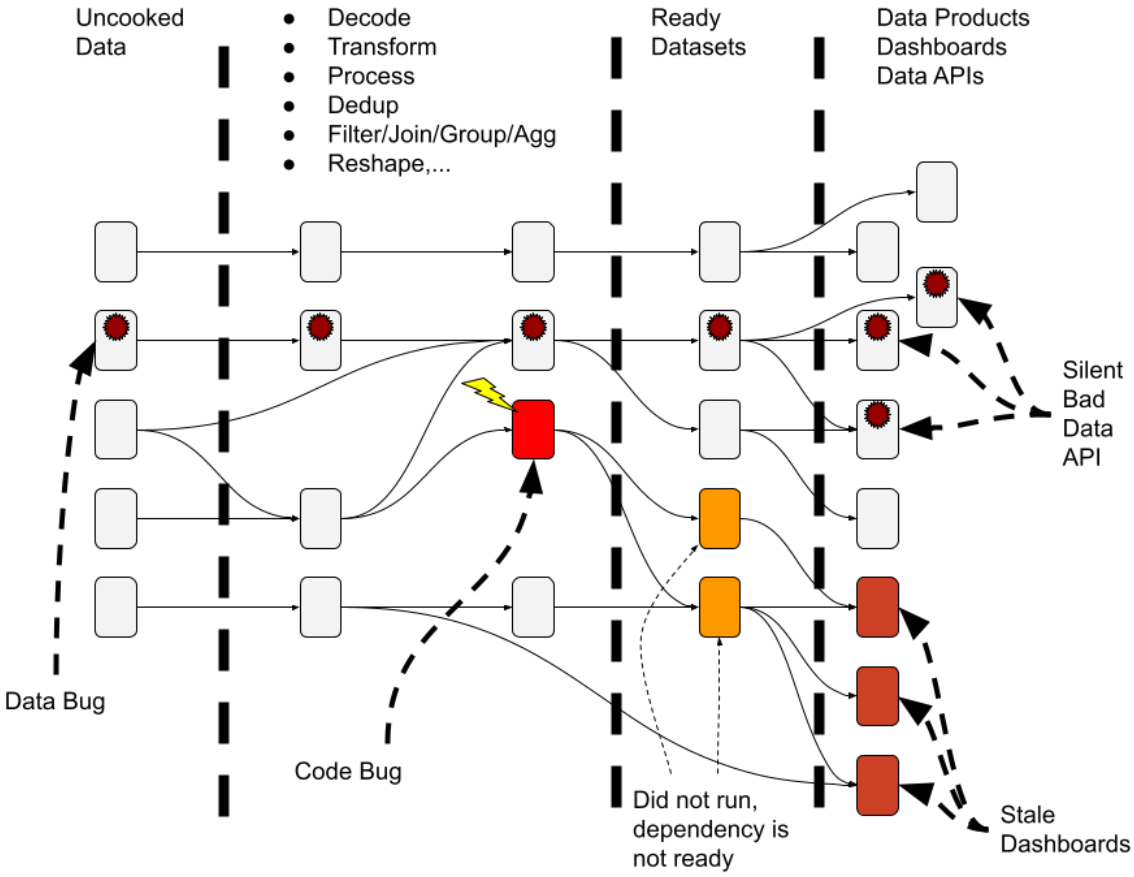
Modern Data Pipelines Testing Techniques

Moussa Taifi

Thanks to Christina, Theo, and Maria Itto for giving me time to work on this book.

Chapter 1
Testing Your Patience

Data Pipeline Transitive Failure Modes: The Reality Check



Data Pipeline Transitive Failure Modes

Problem:

New data devs do not realize that data pipelines have transitive failure modes. These failure modes are not clear to the untrained eye. After the first few months of fighting debugging battles against things like:

- *"Is this number correct?"*
- *"Why is the dashboard not updated for yesterday's data?"*
- *"How come we didn't catch this bug upstream?"*

They finally realize that there are many types of failures that were not covered in the 6-week bootcamp to get this job. Besides, new data devs

find that data producers and consumers are continuously dealing with business change. Unfortunately, these new data devs are ill-equipped to deal with change, like we all were when we started our careers in this data thing.

Complications:

The first thing new devs usually miss is a global view of the "data pipeline." They are assigned to this easy-enough task to add a new aggregate value to a mid-level table. Unfortunately, that task requires them to rename a field to reflect the nature of the new aggregation. Renaming that field will break the data consumers. This transitive failure is the first of many silent new bugs they will need to painfully revert and backfill.

Guidance:

Removing this lack of visibility is quite a challenge in large organizations. Yet, realizing the types of transitive data failures should help. The top learned-on-the-field types are:

- Silent Data Bug
- Code Bug
- Late Data Runs
- Stale Dashboards and Data APIs

The **Data Pipeline Transitive Failure Modes** diagram, adapted from [1], shows a data bug in the source system. This data bug trickles down to all the related downstream datasets. without firing a single alarm. Unfortunately, undetected data bugs are one of the hardest to catch without solid data quality gates.

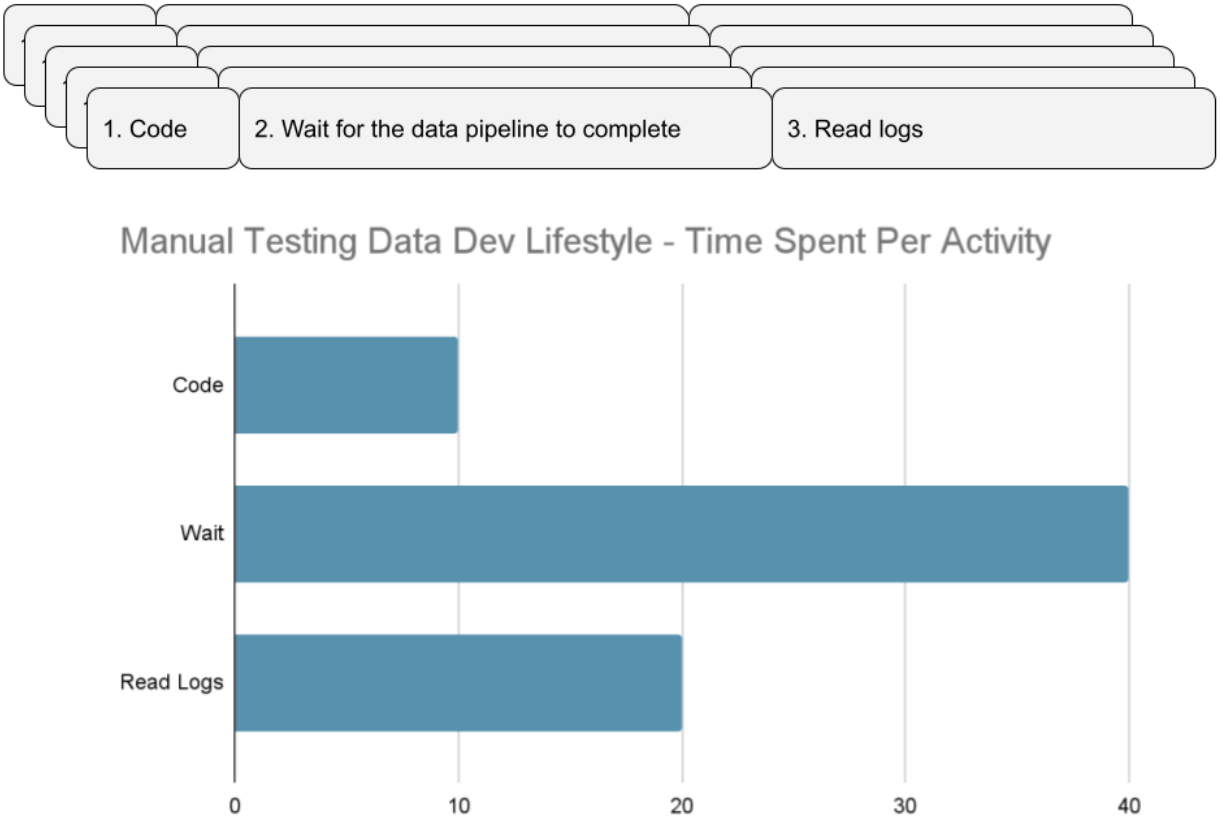
Also, we see a code bug in one of the intermediate steps that made a whole job fail. This job is a dependency of two other jobs downstream. The job workflow scheduler detected this missing dependency and did not

schedule the two downstream jobs. With adequate monitoring, this type of error will alert the data dev.

Sadly, the alert usually comes from the stakeholder directly! *"Why is the dashboard not updated for yesterday's data?"* is the usual complaint. As a result, the data dev loses some credibility with the stakeholder. Then our data dev spends many hours fixing the bug and re-running all the jobs needed to catch up to the latest hours of available data.

Having a visual of the data pipeline's transitive failure modes helps place the current task in the grand scheme. Furthermore, being aware of types of data pipeline failures will motivate new data devs to inject quality in their work. The data devs will hopefully see the value of instrumentation and testing to avoid these issues in the future.

Bad Data Devs Lifestyle



Bad Data Devs Lifestyle

Problem:

"Time for some Email/Instagram/Twitter/Youtube/Reddit" is the typical reaction to waiting for a long data pipeline to run to completion. *"Why wait and babysit this pipeline when I could be doing something else,"* said every new data dev ever.

Sadly, this is the state of affairs for any new data dev. This sad state of affairs stems from their need for more modern, humane, and productive techniques for developing data pipelines.

The Bad Data Devs Lifestyle diagram shows how your average data dev spends time working on a data pipeline. This includes coding, waiting for feedback, and deciphering the feedback.

What we want is to maximize fast feedback. Unfortunately, this is not the case. When running an entire end-to-end pipeline, data pipelines usually run for *"longer-than-average-human-attention-span"* durations. This makes it hard for the data dev to use their time productively, leading to inefficient use of their time and lost productivity for the business.

In addition, you can bet there will be redundant resource usage to rerun the same parts of the pipeline that we already know work fine. Burning shared resources to check that the new code the data dev added to the end of the pipeline didn't break anything seems excessive.

Complications:

Breaking focus is a productivity killer. The data dev spends a bunch of time loading up all the computation graphs in their head. It takes time to build a mental model of the data pipeline in their short-term memory. But then, once they launch the pipeline, they have to wait, just sitting there watching a spinner or inexpressive logs while the whole job runs. Naturally, after waiting for 3 seconds, their mind starts to wander. What's for lunch? What

are the following meetings for the rest of the day? And they ponder on their rich and complex life outside work. So our data devs depart from the mental model of the data pipeline to check something. *"It will be quick,"* they say to themselves. When they come back 30 mins later, they face crazy stack traces. Full of cryptic error messages about a syntax error in a templated SQL query they didn't even touch. Oof! and that happened 10 seconds into the pipeline. That's 29 mins and 50s wasted.

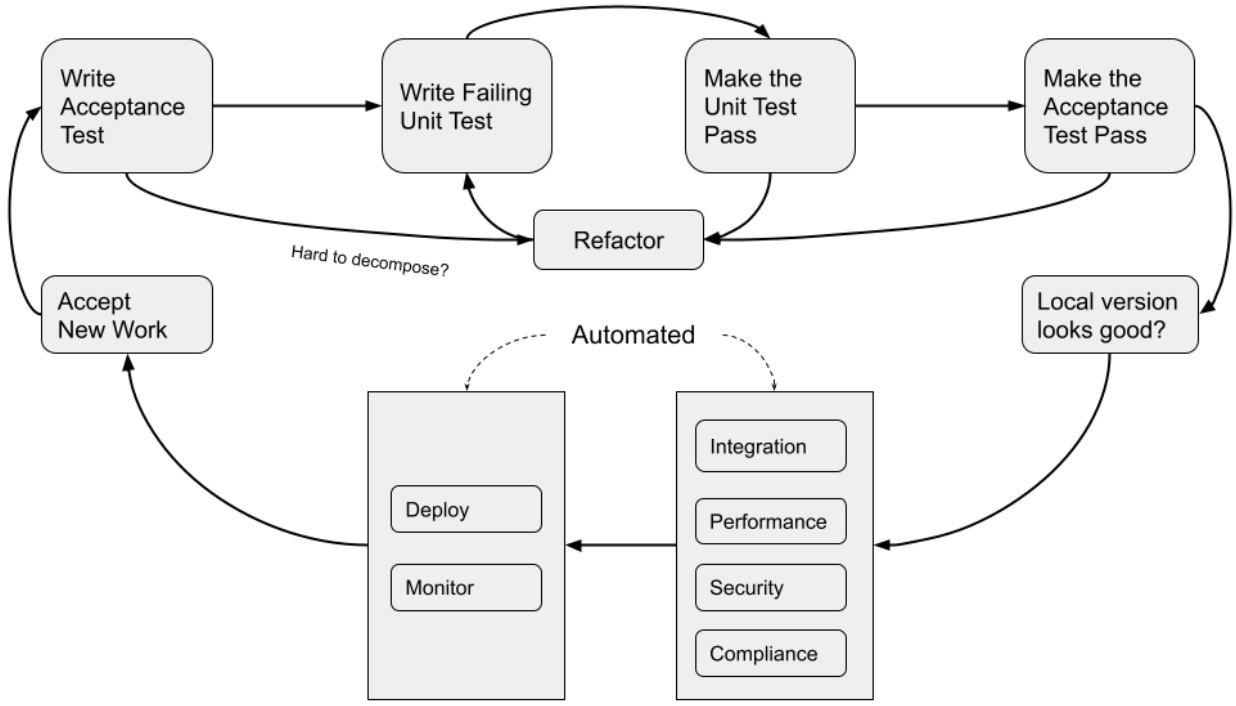
Even worse, a bad data dev lifestyle leads to working at unhealthy times of the day. Early in their new job, data devs realize that resources are constrained. That means data warehouses are busy during the day and under-utilized during the night. So they start working after business hours, at nighttime, to get shorter runtimes and faster feedback. After a few weeks, they realize how unsustainable this is because burnout is real. There must be an easier way.

Guidance:

I remember seeing data similar to the diagram **Bad Data Devs Lifestyle**, in [2]. I realized that many data devs share the same painful experience. The hope is that once new data devs understand that running a costly end-to-end pipeline to check that a one-line change works, is not reasonable.

Instead, they should look for ways to shorten the feedback loop between adding a single line in a function and checking if it works. This is what we will cover in the rest of this work. Initially, the breadth of techniques available can be confusing and overwhelming. Still, we will follow a structured method to organize the modern testing techniques for data pipelines. The problems are not new and your worldwide colleagues around the world have developed many techniques over the recent years. Wisdom of the crowd for the win.

TDD + CICD to the rescue?



Simplified TDD+CICD Loop

Problem:

New data devs usually fall into two camps. The first camp never heard of Test-driven Development, TDD. The second camp tried it and concluded that TDD would get them fired or at least get them late on every project. The competition is fierce. In fact, the other data devs that use ad hoc spot checks are doing just fine. They get rewarded for the quick turnaround on their projects. As a result, applying TDD feels like fighting an uphill battle. They conclude that using TDD in an organization that only partially embraces and rewards software quality is a fool's errand.

Software attracts complexity, and data pipelines are not immune to that. Nevertheless, the original ideas of TDD by Kent Beck [6] have transformed

how software is built worldwide. After a decade of working with data, I still believe that TDD is the best design technique for software that humans have come up with so far. However, it requires such a wide array of skills and techniques to be applied successfully that many data devs give up on using it in their project.

Complications:

Then comes Continuous Delivery [37] on top of that. New devs need the mental tools to write and maintain good tests. But on top of that, Continuous Delivery is imposed on them as the best thing since sliced bread. Yes, it is a fantastic method for developing and delivering software. However, its core value proposition comes from being a repeatable disqualification mechanism. CD tries to automate discarding any software bits that do not pass a set of requirements. But how do we encode requirements in software? Yes, you guessed it, we encode the requirements in tests. Without the skills and discipline to write good tests, devs still get some benefits of CD, where they automate the deployment of their code. But they miss the code value of Continuous Delivery altogether.

As the Continuous Delivery definition puts it: *"Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes, and experiments—into production, or into the hands of users, safely and quickly in a sustainable way."* A new data dev might get the "quickly" bit working. But without automated testing, preferably through TDD, they won't get the "safely" and "sustainably" bits.

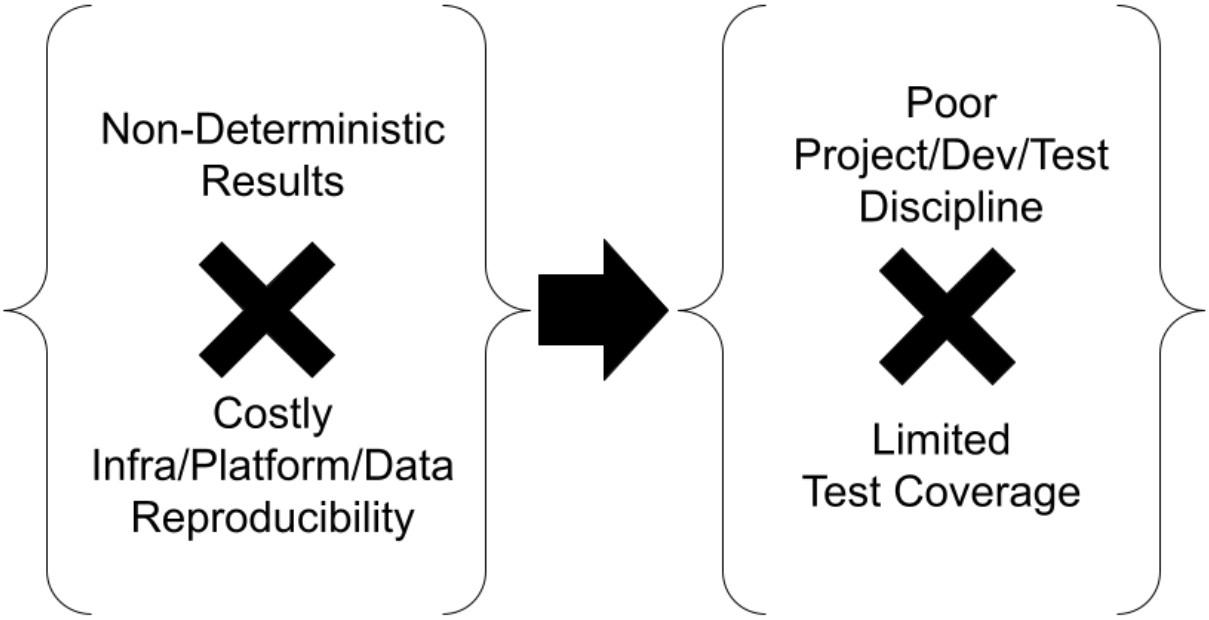
Guidance:

The first step towards improving your software is to be aware of the TDD + CICD loop. In the diagram **Simplified TDD+CICD Loop**, we start by writing the most basic acceptance test of what the data pipeline is supposed to produce. Then we decompose the pipeline into essential parts. We then dive into the individual components of the pipeline and write tests for them before writing the code. Not writing the tests before the code will make

writing the tests feel like a chore. Then we make the tests pass and repeat the process until the acceptance test works.

Initially, we want a "Walking Data Skeleton" or a "Primitive Data Whole." Valuable enough to be deployable. Then we set up the CICD pipeline to be able to deploy the code we have so far. The deployment of a data pipeline usually requires a scheduling mechanism, so we put that in place and include it in the CD pipeline. Once we have a scheduled job that generates the most basic of outputs and a basic deployment process, we can move to the next level of complexity. We write the next acceptance tests and go through the TDD+CICD loop again. Small iterative steps are key here. We will cover this loop in detail in the rest of this book.

Objections to TDD for Data Work



Objections to TDD for Data Work

Problem:

When a new data dev catches the TDD bug and apply TDD for their data pipelines, they report the following objections:

- Data pipelines give non-deterministic results over time
- Creating test data is costly and discouraging
- Recreating the production environment is an expensive nightmare

These objections lead the data devs to cut corners. They lower the test coverage until it reaches a workable tradeoff. On one side of the tradeoff, the effort needed to set test environments. On the other side, the confidence level that their business use case requires.

The core issue is that the business wants the data to be reliable, fresh, and actionable but inexpensive to generate and maintain. This tradeoff is

central to the objections you will tell yourself next time you drift away from TDD for your next data project. Unfortunately, new data devs do not have suitable mental models to explore this tradeoff. What kind of tests do I need? What are the testing components that I can skip, given the business importance of the data? How to set up the bare minimum testing infrastructure required to reach the data reliability the business needs?

Complications:

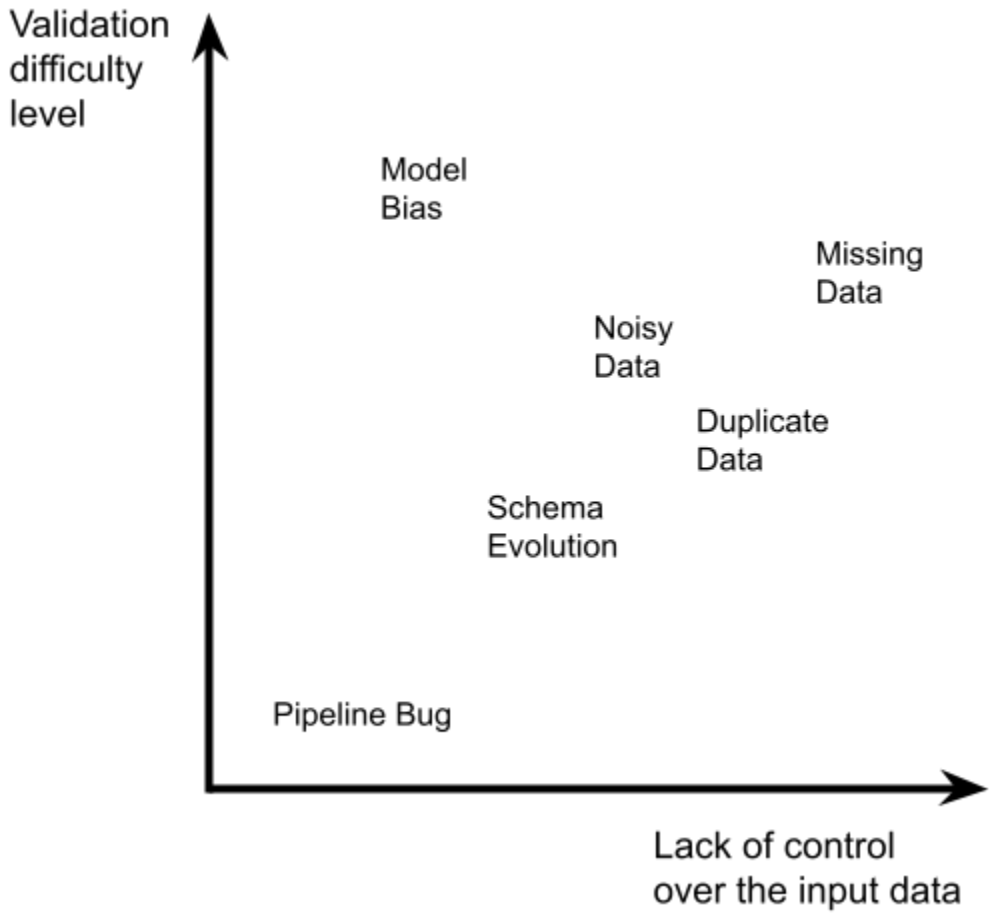
On top of the classic cost vs. reliability tradeoff, technical complexity gets layered on top of that. When adopting TDD+CICD, Data devs face too many decisions at once. How do I get started? What tools, infrastructure, and platform do I need to learn to do this right? How long will it delay my initial release?

Guidance:

As usual, the first step is raising awareness. Having someone tell you that it is normal. All new data devs feel unsure about applying TDD to data pipelines.

The advice here is to learn the skills required for TDD, outside the critical path of your current business process. Take a less important, low-intensity data pipeline and experiment with the concepts in this book. Rewriting an existing data pipeline with TDD+CICD in a parallel data environment is a great starting point. It enables the data dev to learn about the technical practices in a known business domain.

Sources of Data Validation Complexity



Sources of Data Validation Complexity

Problem:

I would bet that many data devs share this view: The inherent lack of control over the input data is the most annoying part of any data pipeline. The fact that a perfectly fine data transformation works on day D1 and breaks on D2, only to work again on D3 can be infuriating. Data pipelines routinely see this happening when a single day of data is bad because of upstream issues. But the rest of the days are fine because the upstream team fixed the issue, and things are well again. Sadly, new devs will not protect the data pipeline from this ahead of time. They never saw it happen, so why bother?

Complications:

The diagram above was adapted from [2].

In addition to the lack of control over the input, the validation complexity is highly variable. It is one thing to fix a bug with known good data on a codebase we are familiar with. But it is another thing to deal with missing data. Missing data is especially weird when you have missing data, but you don't know that you have missing data. As a result, detecting data completeness relies on heuristics that are disheartening for new data devs. These data devs find it to be the perfect excuse for *"nobody told me that the distinct values of this field will change."*

But the source data is one of many culprits. We could also have misunderstood the expected output of our own data transforms. Unexpected outputs are especially common for modeling apps that are trained on biased datasets. So they expose the bias in the output predictions.

Guidance:

The guidance here is to build as good of an understanding of the source data as possible. This is done by talking to the producers and collaborating on data quality contracts. By the same token, the data pipeline under test also has consumers. Indeed, building data quality contracts with consumers is also recommended. This proactive behavior will positively surprise the data consumers. This surprise is because so few data dev teams invest in these data contracts due to the inherent constraints they include. But again, longer-term, you will find that building agreement between your producers and your teams and between your team and your consumers will benefit everybody.

The Data Product Promise No One Can Keep

Problem:
"After almost getting fired because of the previous incident with the exec revenue dashboarding pipeline, I will only work on testable data pipelines from now on," has been said and will be said many times. When new data devs encounter their first data bug in production, they decide to push back on any new data product requirements. No more work until they get full assurance about the data sources and the required data outputs. *"No more dealing with flaky data sources that we have no control over." "No more assuming that we understand what the customer wants."* These are early signs that the data devs are fed up with dealing with so much ambiguity.

Complications:

The reality is that commercial interests will have higher priority than code quality. Besides, platform changes will always happen, and technical

deficiencies will always exist. There is no escape from constant change in the software world. As a result, optimizing our work for learning about the business domain is key. Building delivery mechanisms with fast feedback is critical. Using repeatable engineering practices is the best we can do at this point (the year is 2023). Advances in general artificial intelligence and quantum computing might change that in the future. But for the time being, data engineering is still an engineering profession. Engineering always deals with trade-offs.

Guidance:

The best practices known to humans today for dealing with "not-fully-testable" environments are summarized in the DORA research about effective software teams [3]. Testing is at the core of much of the automation prescribed by the DORA research. We can release millions of times per day, but how do we know if that code works? How do we reduce the lead time between requirements and delivery if the system is a tangled mess of untestable code? How do we quickly recover from data downtime if we cannot test our fixes quickly and reliably? How do we reduce our change failure rate if one fix in one part of the pipeline leads to another failure in an unrelated part of the data pipeline?

The most efficient way to build data pipelines we know of is through disciplined automated testing. This is quite a tall order. Testing data pipelines is at the edge of human knowledge. Thus, only a few new data devs have all the skills necessary to tackle this challenge.

Therefore, the majority ends up giving up on automated testing. Don't join their ranks. Starting small and growing your testing skills iteratively, in small steps, and over time is the best current guidance I can give to work in a "not-fully-testable" data environment. Giving up on testing and reverting to manual testing will only make you bitter. Bitter toward your data sources, pipeline code, and data outputs. Manually validating the quality of your data pipelines will make you mentally burnt out. Having to explain and apologize to your stakeholders about the more recent data downtime gets old quick.

Fighting Against The Manual Auto-Pilot

The Ad Hoc Spot Check

Problem:

The ad hoc spot check is a bad habit. It starts during the exploration phase when you don't know which tables should be used for this report. A couple of queries here and there. A couple of manual size checks. How many rows did I get this time? Did the data land in the correct folder? I guess I can ignore these pandas warnings?

If nobody depends on your data pipeline and you are the only short term user, the adhoc spot check is fine with me. The problem with this is that commercial data pipelines start as single user apps and some of them explode in popularity. Suddenly, the business finds the results of an analysis useful, and asks you to rerun the analysis for a different geo. Now you start scrambling for which notebook to use for the spot checks, and how to change the geo in the relevant places in the analysis. I am not saying that this happens everytime. But even during your exploration activities you might want to invest a little bit of time on automating the testing of your data pipeline.

Complications:

In addition to the business interest, your time is valuable. When developing a data pipeline, the ad hoc spot check of the outputs will need you to rerun the full pipeline every time. Running the entire pipeline will require longer running time because you are ingesting the whole data set. Running all the transformations will be slow. Outputting the resulting full dataset will devour your time. This process is arduous, and without testing automation, you might need to rerun it many times. Checking that the full pipeline completes after a small code change in step 4 of 6 should not be the default. Splitting

the pipeline into meaningful chunks that separate the outside world from the domain logic would be nice. And wouldn't it be nice to test out the effect of a slight change of logic in that step 4 without running all six steps at once?

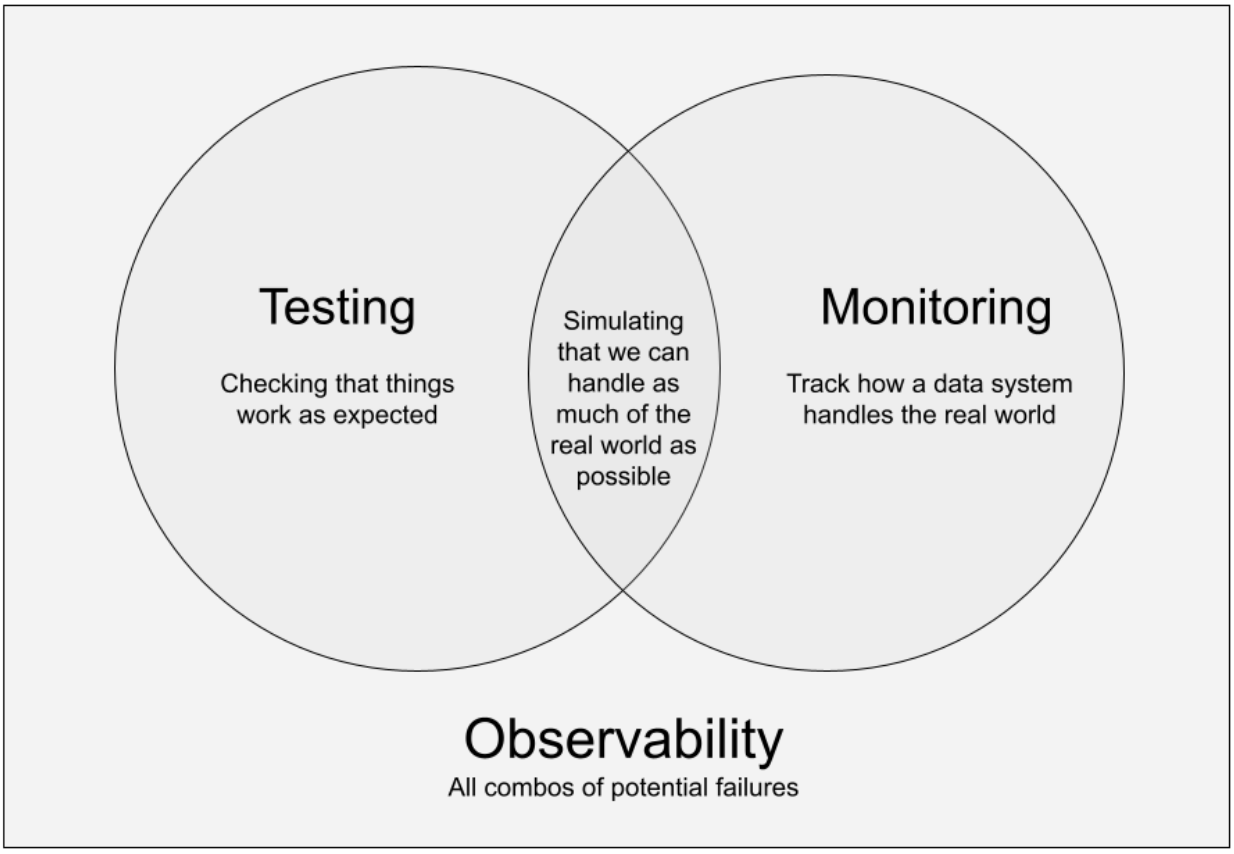
Guidance:

When to move away from ad hoc spot checks? As soon as you think you are working on data that will influence a business decision. Easy.

This book is full of visual techniques to help you decompose your data pipelines into testable parts. In addition, this work will guide the reader on how to deal with the most common issues new data devs were never told about in their initial training.

We will explore core types of data pipeline tests, the supporting components you will need, and how to deal with legacy data pipelines. Then we talk about how to design your pipeline for testability. We show how to organize your data-oriented development environment for maximum autonomy. We then deal with how to optimize your deployment strategy. Then we list ways to influence your data organization to support a healthy testing culture. As a result, having these techniques embedded in your toolbox will help. These techniques will help you decide when ad hoc spot checks are reasonable, because nobody cares. But they will also help to know if you should use some or all the data pipeline testing techniques to meet your business requirements.

Observability vs Testing vs Monitoring



Observability vs Testing vs Monitoring

Problem:

I wish I could tell you that practicing testing will solve all your data problems. Unfortunately, *"no plan survives contact with the enemy"* [4]. This quote is especially true for data products where the lack of control over the inputs is inherent in the data dev world. Data pipelines ingest data sources from external domains that deal with their own set of problems and constraints. There is no way testing the local data pipeline will catch everything, but robust testing will ensure that things work as expected. Sadly, that is just the start of the story.

Complications:

In addition to checking that things work as expected, there is the challenge of checking how the real world impacts our data pipeline. This challenge is where monitoring comes into play. Monitoring is where we observe the boundary of our knowledge about the failure modes of the pipeline. But that is just the beginning. The figure above, **Observability vs Testing vs Monitoring,** adapted from [5], shows how having observability over the data pipeline is even a broader concern than the first two. The infinite combinations of known and unknown behaviors of deployed data systems are much broader than testing and monitoring combined.

Guidance:

"Wider than both testing and monitoring? That's absurd!" Indeed it is, and it goes back to the engineering discipline. Do your best with what you know so far. There is no limit to the number of corner cases that the data pipeline will need to address. Some cases will be business-critical, and some won't be.

What is certain is that the best tool to combat constant change is to optimize the pipeline environment for learning and fast feedback. Those are the only ways to get good at adaptive improvement. Incremental small changes, with solid testing, are vital to dealing with the limitless unknown number of failure scenarios. *"The cluster ran out of disk space? Sure, I'll redeploy the fix that has the clean-up scripts." "The data schema changed? Sure, I'll redeploy the fix that handles the new column name." "We have new categories we have never seen before? Sure, I'll simulate that in our test environment to see the impact and redeploy the new version."*

The key to adapting to change is how fast we can reliably deploy changes to the data pipeline. Having testable data systems is a core rule to support this way of working.

Test-Driven Theater vs Continuous Delivery Theater

The Flamin' Shitnãdo

Problem:

A new data dev joins the team. The team handles a data API that consolidates a set of business metrics that are enriched with external data. The data dev gets onboarded. They are assured that the previous data consultants that worked on this pipeline have put quality front and center. They put in place comprehensive data validation. They have a solid test suite that has a 90% test coverage. There is a set of integration tests that run nightly and a system test scheduled to run daily. The continuous integration server is working fine. Each commit is evaluated on a set of corner cases in addition to the main happy path of the ingested data. The deployment process uses a staging and prod environment. Automated releases and manual approval with quality gates protect the deployment of the pipeline. Looks nice.

Complications:

The new data dev is tasked with adding a unique user count on a dataframe to take into account a GDPR concern that the execs have around the data we collect about users. The data dev gets onboarded on the code base, and things look fine. The dev read this book and has experience with testing. As the initial tests get implemented, the dev notices that most of the testing is really just of the form *"must not crash."* The existing tests run the data transformations on a stale data snapshot and check if the code runs end-to-end.

The dev starts digging deeper and notices that all the assertions are commented out to make the tests pass. The manager checks in, and the

pressure increases to deliver the first iteration of the change. So the dev decides to go ahead without criticizing the current test suite. The CICD pipeline appears to be working, after all. The dev adds the test to check if the unique user count code works, writes the code to make it pass, and moves to release the pipeline since things look OK. The dev gets to release the prod pipeline with the CD tooling, only to find that the actual data pipeline is not deployable with the fake CD tooling. Again the dev finds that the CD pipeline really does not do anything apart from cloning the repo, and linting the code. In an obscure readme, the dev finds a set of manual steps that need to be run on the workflow scheduler manually.

The steps start with *"Hello, fellow cellmate! Sorry to put you through this, but we had to show compliance that we had a full test suite and an automated continuous delivery pipeline for this. We were hired to slap tdd+cicd on this pipeline, and we only had four weeks to do it. Trust nothing the tests and cicd pipeline says. The manual steps here will work fine. Good luck! Don't snitch on us, please. Find a different job if you can."* The dev is stupefied but goes on to do those manual steps, and magically, the set of steps in the readme actually releases the new docker image to prod.

Welcome to TDD+CICD theater. I first heard about TDD theater here [7], but someone had to come up with CICD theater.

Guidance:

Software is a special kind of beast. Only the developers closest to the code know how the system works. So much design is in the hands of developers and so much trust needs to be given to the devs to do the right thing. Culture is almost the only way to get honest estimates and honest work from your developers. Data products are even harder to police. It is one thing to police a login screen, but quite another thing to police a 10-step data pipeline split over three teams.

Giving autonomy to the data devs and equipping them with the best tools the business can afford is a good starting point. Autonomy will encourage

quality data pipelines, and will help escape the evil grip of TDD+CICD theater.

END OF SAMPLE