dbt Labs

The Analytics Development Lifecycle (ADLC)

A letter from our Founder & CEO

In 2016, I wrote a blog post entitled “Building a Mature Analytics Workflow.” In it, I compared analytics to software engineering and stated my position that we should be bringing software engineering best practices into our work as data practitioners (version control, CI/CD, testing, documentation, etc.). Before that, this point of view was counterintuitive and not widely accepted.

Fast forward to today, and that post helped launch a community and a product, and many of the assertions it made have been accepted as best practice in the data industry. However, nearly a decade later, it is clear to me that the original post is in need of an update.

Why? First, we now have the collective experience of tens of thousands of companies applying these ideas. We can observe from dbt product instrumentation data that a large majority of companies that transition to the cloud adopt at least some elements of a mature analytics workflow—particularly related to data transformations. But what about the other layers of the analytics stack?

Here is what I mean:

Can data consumers request support and declare incidents directly from within the analytical systems they interact with? Do you have on-call rotations? Do you have a well-defined incident management process?

Do your ingestion pipelines have clear versioning? Do they have processes to roll back schema changes? Do they support multiple environments?

At your company, do you believe that notebooks and dashboards are well-tested and have provable SLAs?

The answer to these questions, for almost every company out there, is “no.”

The fact is that we—the entire data community—have not rolled out these ideas to all layers of the analytics stack, and this leads to bad outcomes: impaired trust in data, slow decision-making velocity, low quality decisions. We have pushed back this tide within the narrow domain of data transformation; it is time to apply these lessons more broadly across the entire analytics workflow.

We need to collectively acknowledge that we are not done, that there is further to go on this journey.

The goal of this paper is to outline the workflow principles that I believe are the solution to this problem. These principles apply to all analytical jobs-to-be-done; data maturity is an end-to-end effort.

We have achieved so much together over the past decade. I look forward to another decade of progress.

— **Tristan Handy**, September 2024

Table of Contents

Intro: A mature analytics workflow

Analytics is the practice of analyzing data to make truth claims.

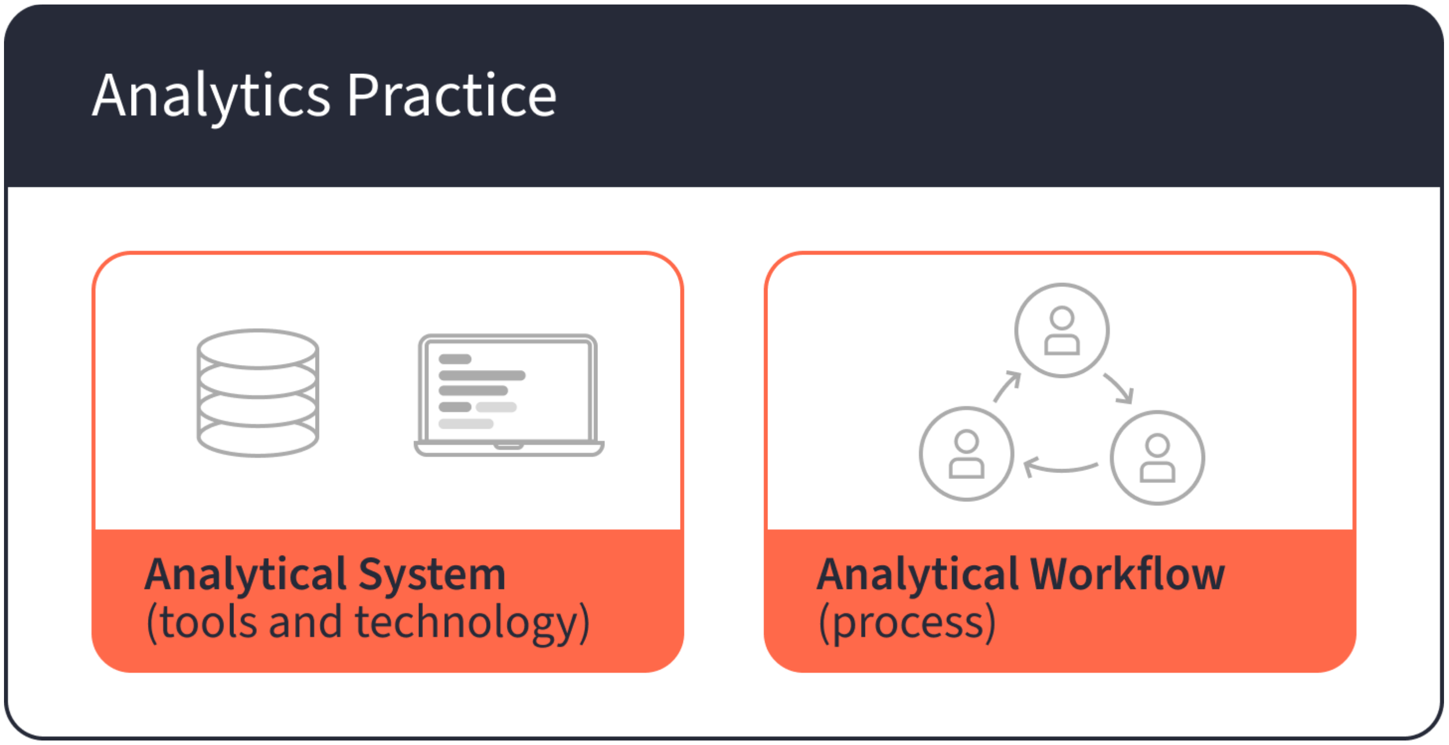
These truth claims can be:

- descriptive ("We had 200 orders yesterday").
- causal ("Revenue is down because our ad inventory is low during the summer").
- predictive ("We estimate that revenue next quarter will come in ahead of plan").
- prescriptive ("Use the following ad copy to this segment to maximize click through rate").

If you are using data to make truth claims, you are practicing analytics. To practice analytics well, you need two things:

- An analytical system (tools and technology)
- An analytical workflow (process)

These two things work together to create your analytics practice.



Intro: A mature analytics workflow (Continued)

Some analytical practices are better than others. And some are simply different: they make different tradeoffs.

For example: velocity and governance are commonly traded off against one another. Spreadsheet-based ad-hoc analysis can be a fast way to get to an answer, but its governance characteristics are typically low.

We believe that teams are capable of practicing analytics in a way that hits a far higher mark on these dimensions simultaneously. This requires both better analytical systems and more mature analytical workflows. Because the reality is that despite technological and process advancements in the past decade, it is still quite uncommon to see mature analytics principles applied to all areas of analytics—from data ingestion to orchestration, observability, discovery, and analysis. As a result, analytics in practice still suffers from many of the same problems it did a decade ago: low velocity, inaccurate results, impaired trust…all without proper cost control. Progress has been made, but there is more we can do.

This paper focuses on the workflows part of the equation. In it, we propose a specific workflow designed to accelerate data velocity while improving data maturity: the Analytics Development Lifecycle (ADLC). We believe that implementing the ADLC is the best path to building a mature analytics practice within an organization of any size.

Requirements of a mature analytics workflow

A mature analytics workflow has the following characteristics.

Data scale

How much data can be processed? A mature workflow requires analytical systems that can scale up and down elastically, abstracting away the complexity involved in processing data sets of any size.

Collaboration scale

How many users can effectively collaborate together? A mature workflow is suitable for a single user and scales to arbitrarily many. As additional users are added to the process, design considerations may change, but the fundamental workflow does not.

Accessibility

How many types of users are capable of using this system? A mature workflow brings different personas together to collaborate as peers.

Velocity

How quickly can a user conduct a given unit of analysis? A mature workflow does require participants to undertake some overhead relative to simple ad-hoc work, but both minimizes this overhead and injects velocity as requirements scale beyond the basic.

Correctness

What is the likelihood that a given output produced is correct? A mature workflow not only produces correct results, it also contains mechanisms to automatically validate correctness.

Auditability

What changes have occurred to produce a given result? A mature workflow produces artifacts with changes tracked and outputs reproducible at any point in time.

Governance

Can we assert that the right people are using data in accordance with all applicable rules and regulations? A mature workflow integrates governance directly and from the outset.

Criticality

Can the business rely on the results of this workflow? A mature workflow produces artifacts that can seamlessly scale from experimental to mission-critical without needing to be re-built.

Reliability

What is the likelihood that the system will operate without failure for a specified time period? A mature workflow requires systems that are resilient to failure and provide uptime SLAs that allow the business to depend on them.

Resilience

Do errors result in massive or minimal business impact? Errors are inevitable in all complex systems, and a mature workflow must anticipate them, minimize their impact, and have mechanisms to quickly remediate them.

Stakeholders of the ADLC

There are three primary personas of individuals that participate in the ADLC:

The Engineer

The engineer creates reusable data assets: pipelines, models, metrics, etc. The engineer is primarily focused on creating data assets that others will use to create business value.

The Analyst

The analyst performs analysis that drives decision-making. The analyst does not make decisions; their role is quantitative investigation, and they present analysis and/or recommendations to the decision maker.

The Decision-Maker

The decision-maker is responsible for taking quantitative outputs and translating them into action for the business. This does not imply seniority: decision-makers include everyone from a campaign manager optimizing segmentation to a CEO directing the resources of an entire company.

Taken together, these three personas cover every individual at an organization who interacts with data for the purpose of analytics.

These personas are not job titles. The ADLC does not require any particular mapping between personas and job titles; different organizations can decide on appropriate job descriptions based on their own unique organizational contexts. For example, a single individual could act as each of these personas at a small startup, while at large companies there may be many job titles that fall along the above continuum.

What is important to the ADLC is how these personas work together. Specifically, the ADLC requires two things.

01 These personas must all collaborate together using a common workflow: the ADLC. The ADLC is not just for engineers and not just for analysts. It is a multi-persona workflow wherein every knowledge worker in an organization collaborates together.

02 These personas must all collaborate together within tooling that empowers each of them to perform their assigned roles according to the ADLC. This might seem obvious given the above point, but in practice this is a major gap in the tooling ecosystem today. While tooling for the engineer has made significant progress over the past decade along its workflow maturity, tooling for the analyst and the decision-maker has largely not kept pace. We see this as one of the biggest barriers to full adoption of the ADLC today.

Hats, not badges

One of the failure modes of an analytical practice is excessive segregation by persona. Because the ADLC is fundamentally an integrated, iterative process, segregating tasks too strongly causes friction and therefore slowness.

For instance, if an analyst has to go to an engineer to source new data, that engineer will likely put a ticket in their queue, then prioritize it, then eventually get to it. This can inject weeks into a process that could otherwise take minutes.

While the ADLC recognizes these three personas—the engineer, the analyst, and the decision-maker—it also encourages us not to see these personas as static.

For example: Analysts get pulled into engineering work to unblock themselves and move faster.

Decision-makers get pulled into analytical work to drill into any analysis provided and ask follow-up questions.

Many jobs are analyst / decision-maker hybrids, where the same person is both analyzing and decisioning on data.

The above three personas are like hats data practitioners put on and take off, not badges we wear all day every day. We have our primary hat, the one we like wearing best. But over the course of the day, as we get pulled into solving real problems, we need the flexibility to put on different hats.

The most effective data practitioners can wear all three hats. And the best data tooling enables as many people as possible to wear all three hats. Even with great tooling, you will still have a hat you prefer. But the ability to wear all of them as the situation demands allows you to complete a single end-to-end task yourself, without getting stuck behind someone else's queue.

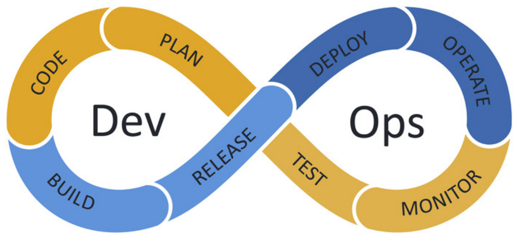
The best organizations encourage talented people to flex between these different personas. They allow them to take an idea and get curious about it, to explore it without needing to file a ticket or wait for anyone else.

This, in turn, requires tooling that prioritizes both *accessibility* and *workflow maturity* at the same time. Rather than saying "stay in your lane," we should be saying "here are tools that allow you to get your job done *in a mature way.*"

The ADLC Model

We propose a simple, straightforward model for the Analytics Development Lifecycle (ADLC). This model governs changes to, maintenance of, and use of any analytical system.

The ADLC is heavily informed by a single guiding principle: analytical systems are software systems. Therefore, in developing large-scale, mission-critical data systems, many of the best lessons that can be learned come directly from software engineering. As such, the ADLC borrows very intentionally from the software development lifecycle (SDLC). There is no one single canonical version of the SDLC, but here is a common visual depiction.

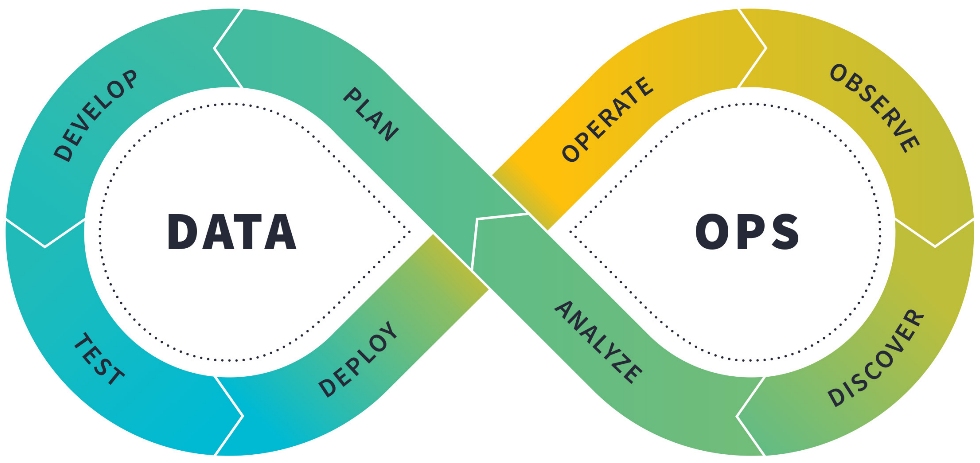


Since its inception, the SDLC has been broadly adopted globally and across industries—it is widely understood and battle-tested. As such, it is a good framework from which to draw upon. In the following sections, we will propose a model for the ADLC that borrows heavily from the SDLC.

The ADLC is not relevant only to a single part of an analytical system: it is relevant to the entire system, from ingesting data to transforming it to analyzing it to building applications on top of it. This paper takes care to describe the work being done as creating analytical 'assets' or 'artifacts'—these can be pipelines, models, dashboards, notebooks, or any other object that creates, moves, processes, or analyzes data.

In this model, there are eight discrete stages:
Plan, Develop, Test, Deploy, Operate, Observe, Discover, Analyze

As in the SDLC, the relationship between these stages is a loop. Here is how the stages logically relate to one another:



Next, we will examine each of these phases. Before diving in, it is important to set realistic expectations. There have been many books written about each one of these stages in the SDLC. Entire books just about writing code, just about operating production systems, etc. This paper will not even come close to a partial treatment on any of these topics; our goal here is to outline the framework as a jumping off point for further work, writing, and community contribution.

Plan

Analytical systems collect data from and build models of the real world, and the real world changes constantly. As a result, changes to analytical systems are constant. The Plan phase is the beginning of the process of making changes to an analytical system.

There is no one-size-fits-all approach to the Plan phase. Changes to an analytical system come from many different places. New business requirements. Issues identified in production. Refactoring. Some changes may be tiny and some may be huge.

The larger the change, the more critical it is to run through a thorough planning process.

Best practices include:

Create and validate the business case
All changes to an analytical system should be based on a solid business case, and often clarifying, documenting, and aligning on that business case is both the most important step in the process and yet the most often skipped. Have a clear process by which changes above a certain threshold must go through before getting worked on.

Create your implementation plan
Determine what shared functionality can be referenced or extended, whether from the community or from others in your org. Extending existing assets can require more up-front effort (because of the testing and coordination required) but over the long term, maintaining multiple similar copies of assets creates a huge drag on a system. Keep your code DRY.

Get stakeholder feedback
Check back in with the stakeholders identified when building the business case to get their comments and buy-in for your proposed approach. Missing this feedback cycle can lead to a lack of alignment and wasted time and effort.

Create a test plan
Before writing a line of code, answer how you will assert that the code you wrote is functioning as intended. What use cases does it need to handle? What problems does it need to be robust to?

Anticipate downstream impacts
If changing an existing asset, identify the downstream consumers of that asset (other assets and other humans). Develop a plan for how to make changes in a non-disruptive way, including testing downstream assets. Create a deprecation plan for older functionality being replaced that will introduce breaking changes.

Plan for maintenance
Historically, most analytical assets were temporary, throwaway. A single spreadsheet attached to an email, ready to be replaced by a new copy next week, disconnected from any larger system. This is not how mature analytical systems are built today. Today's analytical systems are interconnected and long-lived, just like software systems.

And as in software systems, most of the work involved is in the maintenance phase, not in the initial development phase. So, prior to starting the work, make sure to plan for maintenance.

Who should be maintaining the changes you are making over the long term? Are you changing an asset that you built originally? That someone else built? That is owned by your team or another team? Etc.

Determine access levels
What teams and individuals should have access to the work? Is there personal identifiable information (PII) or sensitive personal information (SPI) in the data that you are working with and how does it need to be handled?

Implement larger changes in small pieces
Divide up larger work into smaller units that can be taken through the development process and merged independently. The ADLC is iterative: faster, smaller iteration cycles tend to produce healthier analytical practices.

Develop

The Develop phase is what gets most of the attention in the ADLC, but it is actually overrepresented relative to the amount of time spent in this phase. In practice, with a high-quality Plan phase, the Develop phase should move fairly quickly for an experienced practitioner. It is focused on translating the knowledge you gathered in the Plan phase into code.

Best practices include:

Code first
Whatever type of analytical asset is being built, and whatever persona is building it, the tool you use should read and write human-readable code. Code doesn't have to be the user interface, but it does have to be the underlying representation of all business logic in order to live up to the ADLC. This is for the following reasons:

- Code can be edited by multiple tools, used by multiple personas.
- Code can be checked into source control systems that enable mature collaboration across thousands of co-contributors.
- Code can go through the CI/CD process.
- Code, as language, is composable and therefore maximally expressive.

Choose & customize your own development workflow
There is no 'correct' development workflow or toolset. You can use emacs or vim, CLI or IDE, a graphical user interface or an AI copilot, and you can switch between these different modalities as the situation demands. You are the best judge of how you are maximally effective. What is critical is that your analytical system supports contributing code in multiple modalities as suitable for multiple personas.

Highly productive developers not only choose the tools that best suit them, they customize them, sometimes extensively. This includes everything from hotkeys to color schemes to macros and plugins. The difference between developer productivity within a 'vanilla' development environment and a development environment that has been tuned to your specific workflow can be dramatic.

Adhere to a style guide
Many choices made when writing code are stylistic: capitalization, indentation, etc. While there often is no 'correct' answer on these topics, it is important that they are done consistently across a code base. Create a style guide to define this consistency and then invest the time to follow it—doing so will improve the productivity of every single developer that interacts with your code base.

Prioritize functionality over performance
Your initial job is to write code that meets your functional requirements. Once you do that, you can prioritize performance as much as is appropriate.

Invest in code quality
Most of the time spent in any given analytical system is on its maintenance, not on its original development. Set up yourself and others for success: write good code. This does not just mean following the style guide. It means:

- writing code that is idiomatic to the language you are using
- applying common design patterns
- using descriptive names
- prioritizing readability
- writing in-line comments and supporting documentation
- keeping code DRY
- designing for reusability
- …and many other best practices

Get code reviewed
No code should get merged into production without a second set of eyes on it. Invest in code review in your organization and make sure the process is rigorous and consistent. Make sure peers are incentivized and organized in a way that enables them to review others' work.

Use standards to avoid lock-in
Writing code in proprietary languages risks vendor lock-in. Using open languages (such as SQL and Python) and frameworks (such as Apache Spark and dbt) that are open is highly preferable. Code bases live for a long time—often far longer than the lifetime of any one single product or vendor.

Test

The Test phase is an absolutely essential part of the ADLC. This is one of the areas in which immature analytics workflows often fall furthest from the mark.

We believe that no production analytical artifact should exist without tests. In practice, the data pipeline space has made significant progress in testing over the past decade, but it is still quite uncommon to see well-tested notebooks and dashboards. This represents a significant opportunity for increased maturity in the current ecosystem.

The Test phase falls immediately after the Develop phase and refers to testing changes before they are merged into production. Data is also tested on an ongoing basis in the production environment, but the ADLC considers this the Observe phase. Of course, implementing good tests when writing code typically forms the backbone of effective observability.

The Test phase spans two distinct workflows:

01 Iteratively, interspersed with development.

02 Automatically, as a part of the pull request process. This is known as continuous integration (CI). Changes should never get merged into production without CI.

There are three types of tests that should be implemented:

Unit tests test only the logic being implemented, not the underlying data, and not the system as a whole ("Will this model do what I expect it to do?")

Data tests test the logic being implemented plus the underlying data ("Does the data conform to my expectations?")

Integration tests test the system as a whole ("Do my changes break any other parts of the system?")

As in software engineering, writing tests is not hard (with appropriate tooling), but it can be painstaking. But a well-tested codebase significantly reduces the maintenance burden of an analytical system as errors can be quickly traced to their source. Testing is as much about culture, shared expectations, and accountability as it is about tooling or technique. The desire to skip writing good tests and move on to the next task is always present and must be balanced via accountability mechanisms like code reviews, linting, and test coverage metrics.

In software engineering, there are advocates for different testing methodologies, such as test-driven development. The ADLC does not specify a particular testing methodology, only that no production analytical asset should exist without tests.

It is common and appropriate for developers to focus on testing their own code during development and then running the entire test suite during CI.

Deploy

The Deploy phase is where code is migrated from development to production. This can be a fairly straightforward single hop, or it can be more complicated depending on the needs of the system. All deployment processes should have a set of common characteristics:

Deployment is triggered based on a merge in source control.
As source control stores the state of the repository, branches represent the state of environments. Before deploying code to any environment, that code must be merged to the appropriate branch. The deployment then is made directly from that branch.

Deployment is automated.
There are no human steps required, beyond the act of merging code to a new branch, to deploy changes to a new environment. Sometimes this is straightforward; sometimes this requires work to create migration tooling to enable.

Deployment does not cause user-facing downtime.
Mature analytical systems are constructed in a way to not impact users during deployments.

Rollbacks are automated.
Rollbacks are not only possible, but are automated. Deployments will inevitably surface errors despite robust testing, and mature analytical systems need to plan for this outcome by having a rollback strategy. The best rollback strategy includes smoke tests and automated rollbacks.

Developers choose the size of the change.
Developers can choose to deploy a one-line change or a massive refactor to the entire system. The analytical system must not constrain the way in which they structure their patches.

Operate & Observe

Every analytical system has a production environment, and every organization has certain requirements for its production analytical environment: uptime, latency, throughput, correctness, etc. In the Operate and Observe phase we are not making changes to the system; having deployed and validated changes in the prior phase, we are now operating it in steady state and observing its characteristics to validate that it is conforming to expectations.

Best practices include:

Always-on
In the past, analytical systems were frequently unavailable for significant chunks of the day as new data was loaded or jobs were processed. This is no longer acceptable—analytical systems' production environments should be assumed to be available 24x7x365, with modest windows for planned maintenance.

Tolerate and recover from failure
Your analytical system, with its thousands of models and dashboards and notebooks and sources, each containing thousands to billions of rows, will never be without errors. The goal is to be robust to these errors, not prevent them entirely. Build analytical assets that can recover from failure quickly and with minimal manual intervention.

Catch errors before customers do
Given that any mature analytical system will always contain errors, two of the most important metrics to measure are time to detect an error and time to resolve an error. The goal of error detection and remediation is to identify and resolve errors before your customers see them.

This bar is very rarely hit inside of organizations today. Doing so requires mature processes around incident identification, triaging, and resolution, high quality instrumentation, clear component ownership, and around-the-clock on-call rotations.

Test in production
From increment.com: "Once you deploy, you aren't testing code anymore, you're testing systems—complex systems made up of users, code, environment, infrastructure, and a point in time. These systems have unpredictable interactions, lack any sane ordering, and develop emergent properties which perpetually and eternally defy your ability to deterministically test."

Said another way, most of the bugs you find are not "errors," they are a mismatch between your business logic and the real world. You can never fully anticipate the real world in your test environment, so you inevitably need to test in prod. This requires both excellent instrumentation and tooling that allows you to explore this instrumentation data in real-time.

Choose your own metrics, and then measure them religiously
There are many metrics to choose from in the literature on observability: availability, uptime, latency, throughput, etc. These metrics are all in tension with one another, and there is no universal answer to how these tradeoffs should be managed. Every organization needs to understand these metrics, set their relative priorities, and set goals around the ones that matter. Missing those goals should generate action.

Don't overshoot
Every additional 9 on your SLAs costs an order of magnitude of additional effort/resources to deliver. Assess the real business value of your system's characteristics and aim to deliver what is actually required.

Discover & Analyze

The Discover and Analyze phase includes two distinct but intertwined user flows: discovery of existing data artifacts (data sets, dashboards, metrics, etc.) and using those data assets to answer questions. This phase is where business value is ultimately created. Reports and dashboards are created and viewed. Exploratory data analysis is conducted. Hypotheses are tested. Causal relationships are investigated. Predictions are made.

We find that this phase of the ADLC is, in practice, often relatively immature today.

Every analytical question starts as research. An analyst sets out with a question about the business and looks for data to bring to bear on it. The analyst finds some data, hacks together some code or scripts or whatever to do some sanity checks, and eventually starts to be convinced that there's signal in the data.

At that point, the analyst gradually starts to flip from a mindset of "I need to convince myself" to "I need to convince others." At this point, the analyst anticipates a bunch of follow-on questions that might disconfirm their earlier conclusions and then proactively answers those.

Assuming their initial conclusion stands up to this effort at disconfirmation, they eventually switch from "I need to convince others" to "I need to memorialize this insight." At this point, they consolidate all the analytical artifacts that they have built to get them to this point, clean them, document them, and ship them. At that point, those become long-lived artifacts of the analytical system of an organization.

This, then, is the core tension in the Discover and Analyze phase:

the same set of tools that promotes experimentation and exploration must then also support maturity and productionization.

In practice, most tooling in the presentation layer does not enable this, and as a result this layer of the analytical system often completely skips the ADLC. This results in final products (reports, dashboards, notebooks, etc.) that are low-maturity, even if they are built on top of mature datasets.

Errors can get introduced at any layer of the analytical system, and the presentation layer is no exception. Presentation layer artifacts must go through the full ADLC before they become load-bearing in an organization. The process of productionizing a dashboard should be thought of as no less critical than, and fundamentally no different than productionizing a data pipeline.

This is why the ADLC is a loop. As the analyst moves from exploratory data analysis to memorializing an insight for a wider audience, they shift from the Discover and Analyze phase to the Plan and Develop phase, and thus through another iteration of the entire process.

Requirements of the Discover & Analyze Phase

Beyond the above, the ADLC does not believe that there is a 'right' or a 'wrong' way to conduct exploratory data analysis. Rather, it specifies a set of requirements that all users should have of their analytical systems in the Discover and Analyze phase:

Users should be able to **discover** the artifacts from a mature analytical system directly, through a single search bar, without having to go through any intermediary gatekeepers.

Users should always be able to **operate on data where they find it**, without passing it from person to person in informal networks or ever downloading it locally.

Users should be able to **leave feedback** on any element of a mature analytical system. This feedback should both lead to better discovery as well as fed back into the Plan phase.

Users should be able to straightforwardly **request the access that they need** from a mature analytical system to get their jobs done.

Users should be able to **delegate their own access** to a mature analytical system to their chosen tools and agents.

Users should be able to straightforwardly **validate the correctness and timeliness** of data from a mature analytical system.

Users should be able to straightforwardly **investigate the provenance** of any data element in a mature analytical system.

Users should be able to **view a history** of all state changes to a mature analytical system.

Users should be able to **choose the environment** of a mature analytical system they interact with: dev, staging, prod, etc.

Finally, users should be able to **ignore the implementation details** of a mature analytical system. The system should just work without these users needing to know all of the underlying technical details of how it works.

Conclusion

Our goal in publishing this paper is to create a consistent, shared framework for a mature analytics workflow: the ADLC.

Many parts of this framework can be implemented today. Some require better tooling to effectively implement them. As such, the effort of building towards a mature analytics practice requires the entire industry—data practitioners and technology vendors—working together towards a shared vision of the future.

This process will not play out overnight. Software engineering has only reached its current state of relative maturity after many decades of progress. It will take just as long in data.

As an industry, we are immature in so many ways, and this paper only attempts to lay out, in the broadest of strokes, the path towards a solution. What is required now is to collectively roll up our sleeves and push the conversation forward in every single arena.

We look forward to doing that work right alongside you.

ADLC is the new playbook for data.
Learn more about analytics engineering best practices at getdbt.com/blog

dbt Labs™