The Python Data Stack Migration

Moving From Legacy Pandas Pipelines to a High-Performance Modern Stack

Published by Code-B | 2026 *For data engineering leads, senior data scientists, and engineering managers*

Executive Summary

Most data teams built their Python stack between 2018 and 2022. Pandas was the obvious choice; it was mature, well-documented, and good enough. And for a while, it was.

But datasets have not stayed the same size. Pipelines have not stayed the same in complexity. And the engineering cost of keeping a Pandas-heavy stack running at scale, the memory errors, the slow jobs, and the workarounds piled on top of workarounds have quietly become one of the biggest drags on data team productivity.

This document is not about whether Polars is faster than Pandas. That debate is settled. It is about the harder question: what does it actually take to move a real production codebase with real pipelines, real dependencies, and a real team, to a modern stack? What do you migrate to first? What do you leave alone? What breaks silently that you do not find until two weeks after cutover?

We have structured this around the decisions and failure modes that matter in practice, not the benchmarks that look good in blog posts. By the end, you will have a clear picture of what a modern Python data stack looks like, how to assess your own situation honestly, and how to run a migration that does not blow up in your face.

Section 1: What Changed and Why It Matters Now

There is a version of this conversation that starts with performance numbers. This is not that version. Before the benchmarks, it is worth understanding why the same tool that served teams well for years is now creating friction, since the answer is not that Pandas has become worse. It is that the world around it has changed.

The data volume problem

When Pandas was designed, a "large" analytical dataset might be a few hundred thousand rows. Today, teams working in e-commerce, fintech, healthcare analytics, and product intelligence regularly process datasets of 5 million, 50 million, or even 500 million rows as part of daily pipeline runs.

Pandas was architected for a different reality. It processes data row-by-row in many operations, it loads everything into memory before doing anything with it, and it runs on a single CPU core by default. None of those design decisions was mistakes in 2010. There are constraints in 2026.

The GIL problem

Python's Global Interpreter Lock means that in a standard Python process, only one thread can execute Python code at a time. For Pandas, this means even if you are running on a 16-core

machine, Pandas is using one of those cores for most operations. Your hardware is sitting idle while your pipeline crawls.

Teams have been working around this for years – chunking reads, writing custom multiprocessing wrappers, and spinning up Dask clusters for jobs that should not need a cluster. These are symptoms of the same underlying constraint.

The hidden cost nobody puts in the spreadsheet

Engineering time spent optimising Pandas code is engineering time not spent building things. The senior data engineer who spends a day figuring out the right `chunk size` parameter to stop a job from running out of memory, or who writes a custom chunked-join function because Pandas cannot hold both DataFrames in RAM at once, that is a real cost. It shows up as slow feature delivery and team frustration, not as a line item in your infrastructure bill.

The trigger signals

Not every team is at this point. But if you recognise three or more of these in your current setup, the migration conversation is overdue:

Your daily ETL jobs run for more than 20 minutes, and the main reason is Pandas groupby or join operations on large datasets. Your engineers have added `chunksize` parameters, custom memory profilers, or explicit `gc.collect()` calls to the production pipeline code.

You have had at least one production incident caused by a `MemoryError` on a dataset that should fit in memory. Your cloud compute costs for data processing have increased faster than your data volume, which usually means instances are being sized up to accommodate Pandas' memory requirements rather than the actual computational work.

Your team spends more time debugging Pandas dtype issues, silent coercions, unexpected NaN propagation, and SettingWithCopyWarning than building new pipelines.

If only one or two of those are true, this document is a useful context for the future. If three or more are true, it is a decision you are already too late to make.

Section 2: What the Modern Stack Actually Looks Like

Before talking about migration, it helps to have a clear picture of what you are migrating toward. The modern Python data stack in 2026 is not a single library replacement. It is a layered architecture where each layer has tools designed for what that layer actually needs to do.

The four layers:

1. The ingestion layer is where data enters your pipeline, reading from CSVs, Parquet files, databases, APIs, or object storage. The right tool here reads lazily, meaning it does not load the entire file into memory before beginning to process it. This is where `pl.scan_csv()` and `pl.scan_parquet()` in Polars, or DuckDB's direct file querying, are dramatically better than `pd.read_csv()` for large files.
2. The processing layer is where transformation happens, filtering, joining, aggregating, and reshaping. This is where most performance bottlenecks live and where the choice between Polars, Dask, and PySpark has the most direct impact.
3. The compute layer is the underlying memory and execution model. Apache Arrow has become the connective tissue of the modern data stack because it defines a standard in-memory columnar format that different tools can share without copying data. When Polars, DuckDB, and Pandas 2.x all speak Arrow natively, you can pass data between them without the serialisation overhead that used to make multi-tool pipelines slow.
4. The serving layer is where data leaves your pipeline, into a model training run, a dashboard, an API response, or a database write. This is often where Pandas still makes sense, not because it is faster, but because the tools downstream (Scikit-learn, Matplotlib, and Statsmodels) still expect Pandas DataFrames.

Why Arrow matters more than most teams realise

Apache Arrow is not a library you use directly in most workflows. It is a specification, a definition of how columnar data should be stored in memory. But its implications are significant. A Polars DataFrame and a Pandas 2.x DataFrame backed by ArrowDtype can share the same memory region without copying. DuckDB can query a Polars LazyFrame directly.

A Spark worker can receive Arrow batches without deserialization overhead. This means the modern data stack is not about picking one tool and using it for everything; it is about using the right tool at each stage of a pipeline and moving data between them efficiently because they all share the same memory format.

The hybrid pattern is winning in 2026

The stack that makes most sense for the majority of data engineering teams right now is not pure Polars, and it is not Pandas plus workarounds. It is a thoughtful combination:

DuckDB or Polars lazy scan for ingestion and heavy filtering on large files; Polars for transformation and aggregation; a clean `.to_pandas()` conversion exactly once at the boundary where ML libraries need it, and Parquet as the storage format between pipeline stages rather than CSV.

This is not a theoretical pattern. It is what teams at scale are actually running.

Section 3: Polars - What Makes It Architecturally Different

Every article about Polars says it is fast because it is written in Rust. That is technically true but practically useless. The more important question is what specific architectural decisions make it fast, because understanding those decisions tells you when Polars helps and when it does not.

Lazy evaluation is the biggest difference

When you write a Pandas pipeline, each operation executes immediately. `df.filter(...)` runs, returns a result, and only then does the next operation run. Polars, when you use it in lazy mode, does something different. It builds a query plan, a representation of everything you want to do, and then optimises that plan before executing any of it.

This means if you filter a dataset and then aggregate it, Polars can figure out which columns it actually needs to read from disc, which rows it can discard before loading them into memory, and which operations can run in parallel.

Pandas cannot do any of that because by the time you add a new operation, the previous one has already run.

The real-world impact of this is not small. One team described migrating a 50 million row pipeline from Pandas to Polars' lazy mode. The Pandas version processed the data in 8 minutes. The Polar's version took 19 seconds. The migration itself took two hours. That ratio, a 25x speedup, 2 hours of work, is roughly representative of what teams report on ETL-heavy workloads.

Parallelism without configuration

Polars uses all available CPU cores by default. There is no configuration parameter, no `n_jobs=-1`, no multiprocessing wrapper. The library parallelises at the expression level, meaning different parts of the same computation run on different cores automatically. On a 16-core machine, Polars will use all 16 cores for a groupby aggregation without you doing anything. Pandas will use one.

The expression API

In Pandas, complex transformations often end up as `.apply()` calls with lambda functions. These are essentially Python loops; they bypass vectorisation, they break parallelism, and they are slow.

Polars' expression system (`pl.col()`, `pl.when()`, `pl.lit()`) compiles transformations into optimised execution plans rather than Python loops. If your Pandas code is full of `.apply()` calls, moving to Polars expressions is where you will see the largest improvement. If your Pandas code is already well-vectorised, the gain from this specific change is smaller.

The strict type system

This is a double-edged characteristic of Polars that migration guides often undersell. Polars has a strict, explicit type system. Every column has a defined type, nulls are handled through a proper null type rather than NaN floating-point values, and Polars will raise an error rather than silently coerce a type mismatch.

In Pandas, a column of integers with one missing value becomes a column of floats (because NaN is a float). This happens silently, and it has caused more production data quality bugs than most teams want to admit.

Polars does not do this. A column is either an integer column with nulls or it is a float column, and you have to be explicit about which one you want.

The migration consequence of this: when you first run your pipeline through Polars, the type errors it surfaces are almost certainly real bugs that Pandas was silently propagating. This is a feature, not a problem, but it means your first Polars run on a real dataset will likely fail, and that failure is informative.

Where Polars is not the answer

The performance gap between Polars and Pandas is real, but it is scale-dependent. On datasets under about 1 GB, the difference is often negligible in absolute terms.

A 10x speedup on a 3-second operation is 0.3 seconds. If that pipeline runs once a day, the gain is irrelevant. Polars' advantage compounds with scale; on a 10 GB dataset with complex joins and aggregations, the difference is hours vs minutes, and that is a different conversation.

Polars is also slower than Pandas for heavy string operations that involve complex regular expressions. This is not a corner case; text preprocessing pipelines can see Pandas outperform Polars significantly on this specific workload. Know what your pipeline actually spends its time doing before assuming Polars will be faster.

Section 4: The Migration Decision - How to Think About It Honestly

The most common mistake teams make when considering a migration is turning it into a philosophical question: "Should we be a Polars team or a Pandas team?" This is the wrong frame. The right question is much more specific: which of our pipelines are slow or expensive because of Pandas' architecture, and is the cost of migrating those pipelines less than the ongoing cost of not migrating them?

That calculation is different for every team, and it depends on three things.

First, what your pipelines actually do

Profile before you decide. The actual distribution of time in most data pipelines is uneven; typically, two or three operations (usually a large join, a complex groupby, or reading a large file) account for the majority of runtime.

If those operations are in Pandas, Polars will help dramatically. If your bottleneck is actually a database query, an API call, or a write operation to a slow storage layer, migrating to Polars will not change anything meaningful.

Profiling a Pandas pipeline to find the actual bottleneck takes about 30 minutes with `cProfile` or the `py-spy` sampling profiler. Do this before making any migration decision. Teams that skip this step often find, after migration, that they spent two weeks rewriting a pipeline and the runtime barely changed because the real bottleneck was not Pandas.

Second, your dependency landscape

Polars is the right default for pure data engineering work. But data science pipelines are rarely pure. If your pipeline ends with a Scikit-learn model training run, a Statsmodels regression, a Matplotlib visualisation, or a custom library that was built to accept Pandas DataFrames, you have a compatibility boundary to manage. That is not a reason to avoid Polars, but it is a reason to plan the conversion point carefully.

The conversion `polars_df.to_pandas()` is a real operation with real cost. If you call it once at the end of a large pipeline, the overhead is small. If you call it inside a loop or multiple times within the same pipeline, you are undermining most of the performance you gained.

Third, your team's current capacity

Polars is not a drop-in replacement. The expression API is different enough from Pandas that engineers who are fluent in Pandas will need time, realistically two to four weeks of regular use, before they write idiomatic Polars naturally.

The syntax is not dramatically different, but the mental model is. Writing Pandas code is largely imperative: do this, then do that, then do this other thing. Writing good Polars code is more declarative: express what you want the result to look like, and let the query optimiser figure out how to get there efficiently.

This shift takes time to internalise, and if your team is under delivery pressure, timing the migration poorly creates the worst of both worlds: a half-migrated codebase that nobody is fully comfortable with.

A practical way to make the decision: score your existing pipelines on three dimensions. How slow or expensive is it currently? How self-contained is it (how many Pandas-dependent downstream steps does it feed)? How frequently does it run and how often does its behaviour

change? A high score on all three means migrate it soon. A low score on all three means leave it alone.

Section 5: Three Migration Patterns - What Actually Works

These are not theoretical approaches. They are documented patterns from teams that have run migrations in production, with the tradeoffs each involves stated clearly.

Pattern one: Greenfield first

This is the lowest-risk approach and the right default for most teams. The principle is simple: do not touch existing pipelines. Write every new pipeline in Polars. Over time, as old pipelines age out, get rebuilt, or are replaced by new work, the codebase naturally shifts.

What makes this work is that it requires no migration risk at all for existing production systems. The team builds Polar's fluency gradually on new work, which is the right context to learn a new tool, when the stakes of mistakes are lower. The downside is that it is slow. If your problem is that existing pipelines are costing you money or reliability today, greenfield first does not solve that problem on any useful timeline.

Teams that benefit most from this pattern are teams where pipeline performance is not a current emergency, teams that are actively building new pipelines and can establish Polars as the standard before legacy code becomes the majority, and teams with limited capacity for migration risk.

Pattern two: Bottleneck targeting

Profile your existing pipeline portfolio. Identify the three to five jobs that are slowest, most expensive, or most frequently cause operational problems. Migrate only those. Leave everything else on Pandas.

This pattern delivers 80% of the practical performance gains for roughly 20% of the migration effort because performance in data pipelines follows the same Pareto distribution as most engineering systems, and a small number of operations account for most of the cost.

The GitHub engineering team that migrated a nightly 400 GB ETL from Pandas to Polars is a good example of this pattern in practice. The job went from 90 minutes on a 128 GB machine to 11 minutes on a 32 GB machine. Cloud cost dropped by roughly 75%. The rest of their pipeline portfolio stayed on Pandas.

The practical execution: run profiling on your top ten most resource-intensive pipelines. Pick the top three by some combination of runtime, memory consumption, and operational incident

frequency. Migrate those. Measure the result. Decide whether to continue based on actual data, not estimates.

One important discipline for this pattern: run both implementations in parallel for at least one full pipeline cycle before decommissioning the Pandas version. Validate that the outputs match. This is not optional.

Polars handles certain edge cases differently from Pandas, and those differences will not show up in unit tests on toy data; they show up in production data with messy edge cases that you did not think to write tests for.

Pattern three: Full migration

This is appropriate when infrastructure cost or pipeline SLAs have become a hard constraint when a pipeline is simply too slow for its business purpose, regardless of optimisation, or when cloud costs have grown to the point where the migration pays for itself in months. It is also appropriate for new products where there is no legacy codebase to preserve.

Full migration requires a more careful upfront audit. Before writing a line of Polars code, map every `.apply()` call (these need to be rewritten as expressions, not just translated), every downstream consumer of the pipeline's output (anything that receives a Pandas DataFrame needs to be checked), and every place where Pandas' implicit type coercion might be hiding a data quality assumption you rely on.

The Strangler pattern from software engineering is the right execution model for full migration: replace one segment of the pipeline at a time, validate each replacement, and cut over progressively. Never do a big-bang rewrite of an entire pipeline in one step if that pipeline has been in production for more than six months. The surface area of edge cases is too large.

Section 6: The Things That Break Silently

This section exists because most migration guides skip it, and it is where teams get hurt.

Polars and Pandas handle several edge cases differently. Some of these differences will cause immediate errors that are easy to find and fix. Others will produce slightly wrong results silently, as the pipeline runs, produces output, and nobody notices until a downstream report is wrong. These are the dangerous ones.

Null handling

In Pandas, missing values are represented as `NaN`, which is technically a floating-point value. This creates a specific behaviour: a column of integers that contains a missing value becomes a column of floats, because `NaN` is not an integer.

Calculations involving `NaN` in Pandas propagate the NaN; adding any number to NaN gives NaN. In Polars, nulls are a proper data type separate from any numeric representation. Polars skips nulls in aggregations by default, rather than propagating them.

This sounds like a minor difference, but in practice, a `sum()` or `mean()` on a column with missing values will give you different results in Polars and Pandas unless you explicitly handle nulls the same way in both.

If your pipeline has downstream logic that depends on how missing values aggregate, this is a silent correctness bug waiting to happen.

Join column ordering

In Pandas, after a merge operation, the resulting DataFrame preserves the column order you would intuitively expect based on the order of the input DataFrames. In Polars, column order after a join is not guaranteed to be preserved in the same way.

If any downstream code accesses columns by position rather than by name, `df.iloc[:, 3]` rather than `df["revenue"]`, it will silently read the wrong column after a Polars migration. This is a bad practice in Pandas, too, but it is common in older code.

The `.apply()` trap

If you translate a Pandas `.apply(lambda row: ...)` directly to a Polars `.apply()` or `.map_elements()`, you will get correct results but none of the performance benefits. Row-wise Python functions are still Python loops in Polars; they bypass the Rust execution engine entirely. The correct migration is to rewrite the transformation as a native Polars expression.

This often requires thinking about the operation differently, not just translating syntax, and it is where the mental model shift from imperative to declarative becomes necessary. If you have a complex `.apply()` that you cannot figure out how to express natively in Polars, that is a signal to spend time on it, not to leave it as `.map_elements()` and move on.

Type strictness surprises

Polars will raise a SchemaError if you try to assign a value of the wrong type to a column or if you try to operate on incompatible types.

Pandas would often silently coerce. This means that a column your pipeline assumed was always Int64 but occasionally contained float values, something Pandas handled by silently converting the whole column to Float64, will cause Polars to fail on those rows.

Again, this is surfacing a real bug. But it will surprise you on the first run with production data if you only tested with clean sample data.

The `.to_pandas()` overhead trap

When you have a pipeline that processes large data with Polars and then converts to Pandas for a downstream Scikit-learn step, the conversion is a memory copy. For a 5 GB processed DataFrame, `.to_pandas()` creates a second 5 GB object in memory momentarily.

On a machine with 16 GB RAM, this can cause an OOM error on an operation that should complete fine. The solution is to do the conversion as late as possible and to ensure you are only converting the columns and rows that the downstream step actually needs, not the entire DataFrame.

Section 7: Managing the Pandas Boundary in a Hybrid Pipeline

Most production Python data stacks in 2026 are not pure Polars. They are hybrid: Polars for the heavy lifting and Pandas where compatibility requires it. Getting this boundary right is the practical engineering challenge that determines whether a hybrid pipeline is faster and cleaner than a pure Pandas pipeline or whether it ends up being slower and more complicated than either.

Where the boundary belongs

The correct architecture is doing all filtering, joining, aggregating, and reshaping in Polars. Convert to Pandas once, as close to the downstream consumer as possible, with only the data that the consumer actually needs.

This sounds obvious, but the failure pattern is common; engineers convert early, often right after loading, because they are more comfortable with the Pandas API. This negates the performance benefit of using Polars at all.

A useful discipline: treat `.to_pandas()` like a database write. It is a boundary operation with cost. Mark every call in your codebase explicitly with a comment saying "compatibility boundary: converting for scikit-learn" serves both documentation and code review purposes. When the call is visible and labelled, unnecessary calls get caught in review.

Pandas 2.x with ArrowDtype

Pandas 2.x introduced support for Arrow-backed columns using `pd.ArrowDtype`. When you convert from a Polars DataFrame to a Pandas DataFrame backed by ArrowDtype, the conversion is significantly cheaper because both libraries use the Apache Arrow memory format in some cases, the memory can be shared without copying at all.

This is worth knowing because it means the conversion cost at the boundary, which has historically been a reason to minimise Polars adoption in mixed stacks, is lower in 2026 than it was in 2023.

API compatibility layers

Tools like Narwhals provide a compatibility layer that lets you write DataFrame code that works on both Polars and Pandas inputs. These are useful for library authors and for code that genuinely needs to support both, but they come with a tradeoff: they add complexity, they can hide performance pitfalls if you are not careful about which operations trigger expensive conversions, and they are not a substitute for understanding both APIs. Do not use a compatibility layer to avoid learning Polars. Use it where you have a genuine reason to support both.

The Scikit-learn integration update

As of February 2026, Polars has direct integrations with Scikit-learn and XGBoost, meaning these libraries can now accept certain Polars DataFrame inputs directly without explicit conversion.

This does not cover all use cases yet, but it is narrowing the boundary problem. Check the current integration status for your specific ML workflow before assuming you need a full conversion.

Section 8: What the Team Transition Actually Requires

The technical migration is often the easier part. The harder part is that you are also asking a team of engineers who are fluent in a tool they have used for years to learn a different way of thinking about data transformations. This takes longer than most technical leads budget for, and rushing it produces the worst outcome: code that is syntactically Polars but architecturally still Pandas, which is slower than either.

The expression API is the mental model shift

Engineers coming from Pandas tend to write Polars code imperatively at first, build an intermediate result, assign it to a variable, operate on it, and assign it to another variable. This works, but it bypasses lazy evaluation and loses most of the performance benefit.

The shift to writing expressions, chains of `pl.col()` operations that describe the desired transformation as a single expression rather than a series of mutations, is the core of thinking in Polars. The best way to accelerate this shift is deliberate practice on real data, with code review focused on expression style rather than just correctness.

Two to four weeks is the honest timeline

Engineers who are experienced in Pandas and comfortable with SQL typically take two to four weeks of regular Polars use before their code is naturally idiomatic. Engineers who are less experienced with either will take longer. Factor this into migration planning.

A team of five data engineers migrating a complex pipeline portfolio will be less productive for roughly a month while the mental model shift happens. That is a real cost, and it should be in the migration plan.

Code review is where standards get established

The first six to eight weeks of Polars adoption are when anti-patterns get normalised. If `.map_elements()` with Python lambdas passes code review ten times, it becomes the accepted pattern. If `.to_pandas()` conversions mid-pipeline are not caught and questioned, they accumulate.

The team lead or most experienced Polars user needs to be actively reviewing early Polars code, not just for correctness but for idiomatic style. This investment pays dividends for the next two years of pipeline quality.

Testing Polars migrations

The standard for validating a migrated pipeline is to run both the old Pandas version and the new Polars version on the same production data and compare outputs. The comparison needs to be column-by-column, with explicit handling of the type differences and null handling differences described in Section 6.

A mismatch is not automatically a bug in the Polars version sometimes it surfaces a bug that was always in the Pandas version but went undetected. Either way, investigating mismatches is how you find them before downstream systems do.

Section 9: The Scaling Ladder Beyond Polars

Polars solves the single-machine performance problem. It does not solve the problem of a dataset that is larger than the available RAM of any single machine. Understanding where Polars reaches its limit and what comes next prevents teams from either over-engineering small problems with distributed systems or under-scaling large ones with single-node tools.

Polars' streaming mode

Before reaching for Dask or Spark, it is worth knowing that Polars has a streaming execution mode that processes data in chunks without loading the full dataset into memory. For datasets

that are larger than RAM but still within a single machine's reach, streaming mode often solves the problem without adding the operational complexity of a distributed framework.

Streaming mode is accessed by calling `.collect(streaming=True)` instead of `.collect()` on a lazy query. Not all Polars operations support streaming mode yet, but the common ones — groupby, join, filter, aggregation — do.

Polars Cloud

Polars Inc. launched a commercial cloud offering in 2025 that takes local Polars queries and runs them at scale in a serverless, managed environment. The same Polars code you run locally can be submitted to Polars Cloud for execution on distributed infrastructure without rewriting for Spark or Dask.

For teams that need more than single-machine scale but do not want to manage distributed infrastructure, this is a meaningful option in 2026 that did not exist two years ago.

Dask for the middle ground

For datasets in the range of 10 GB to a few hundred GB that genuinely do not fit in memory on a single machine, Dask remains the most practical step up from Polars. The key advantage is API familiarity; Dask's DataFrame API closely mirrors Pandas, which means engineers fluent in Pandas can adapt quickly. Dask parallelises across cores and can scale across a small cluster if needed.

The limitation is that Dask is still Python-bound and GIL-constrained in ways that Polars is not, so its performance ceiling per node is lower. Use Dask when the dataset size requires distributed computation, but the team's familiarity with Pandas-style code is a practical constraint.

PySpark for genuine enterprise scale

PySpark makes sense when data volumes regularly exceed what a single machine or small cluster can handle, roughly 100 GB and above in practice, though the threshold depends on the complexity of the transformations. PySpark brings significant operational overhead: cluster setup, job configuration, and debugging across distributed logs. It also carries a significant learning curve for engineers coming from Pandas.

The tradeoff is appropriate when the data genuinely requires distributed processing, when integration with existing Databricks or Hadoop infrastructure is a requirement, or when streaming data from Kafka is part of the pipeline. For most analytics-scale problems, jumping to PySpark before exhausting Polars and Dask options is over-engineering.

DuckDB as a complementary tool

DuckDB deserves specific mention because it occupies a useful role that is distinct from any of the above. It is an in-process analytical database; you can run SQL queries directly inside your Python process on Parquet files, CSV files, or DataFrames without loading the full file into memory first.

For pipelines that involve heavy filtering or aggregation on large flat files before the data even enters a DataFrame, DuckDB can handle that initial heavy SQL step faster and more memory-efficiently than any DataFrame library.

The pattern that works well: DuckDB for initial ingestion and SQL-based filtering on raw files, Polars for DataFrame-style transformations on the filtered result, Pandas at the ML boundary.

Section 10: A Practical Migration Roadmap

Everything in this document points toward the same practical outcome: a clear sequence of steps that a data engineering team can follow to move from a legacy Pandas stack to a modern, high-performance stack without breaking production systems in the process.

Before you start: the pipeline audit

Spend one to two weeks profiling your existing pipeline portfolio before making any migration decisions. For each pipeline, record: total runtime, peak memory consumption, frequency of execution, number of downstream consumers, and whether it has caused operational incidents in the last six months. This gives you an objective basis for prioritisation rather than migrating based on which pipeline someone is most excited about.

Months one and two: Establish the new standard

Set Polars as the required tool for all new pipeline development. Write an internal style guide covering the three things that matter most: always use lazy mode with `scan_` functions for file reads, write transformations as expressions rather than `.apply()` calls, and mark every `.to_pandas()` conversion explicitly with a comment. Get the most experienced engineer on the team fluent in Polars first. This person becomes the code review anchor for the rest of the team's transition.

Months three and four: migrate the bottlenecks

Using the pipeline audit data, select the three to five pipelines with the highest combination of runtime cost, memory cost, and operational risk. Migrate these using the parallel-run validation approach: run both versions on production data, compare outputs column by column, and resolve all discrepancies before decommissioning the Pandas version. Document every edge case you find during this process; these become the test cases that protect future migrations.

Month five and six: convert intermediate storage. If your pipelines are passing data between steps as CSV files, convert intermediate storage to Parquet. Parquet is columnar and compressed and reads significantly faster than CSV for both Polars and Pandas. This change is lower risk than a library migration and delivers meaningful performance improvement for any pipeline that reads intermediate results frequently. It also prepares the data layer for future steps up the scaling ladder. Dask and Spark work better with Parquet than with CSV by a large margin.

Month six onwards: gradual replacement

With the team fluent in Polars, the highest-impact migrations complete, and the storage layer modernised, the remaining Pandas pipelines can be replaced progressively as they reach natural rebuild points, performance issues, feature additions, and scheduled maintenance. There is no need to force migration on pipelines that are working well and not causing cost or reliability problems. The goal is a modern stack, not a pure-Polars codebase for its own sake.

The one thing that should never slip

Throughout the entire migration, parallel-run validation before cutover is non-negotiable. Every Polars implementation should be validated against the Pandas implementation it replaces on real production data before the Pandas version is decommissioned. Silent correctness differences are the migration failure mode that teams remember for years. The parallel-run step is what prevents them.

Closing Thoughts

The Python data stack is in a genuine period of change. Not the kind driven by hype, the kind driven by the practical reality that the tools teams built five years ago were designed for a world where datasets were smaller, machines had fewer cores, and the cost of slow pipelines was lower.

The migration conversation is not about being on the right side of a technology trend. It is about whether your team is spending engineering time fighting your tools or building things. The teams that have moved to a modern stack describe the change the same way: not that their pipelines are faster, though they are, but that their engineers spend less time on workarounds and more time on actual work.

The path is not complicated, but it is specific. Profile first. Migrate the bottlenecks. Validate every output before cutting over. Build the team's fluency on the new work before asking them to rewrite old work. Get the storage layer right. Be honest about the compatibility boundaries.

None of that is glamorous. But it is what a migration that actually holds up in production looks like.

This whitepaper was produced by Code-B, a Python development and data engineering practice working with teams across fintech, e-commerce, and enterprise analytics. If you are assessing your current pipeline infrastructure or planning a migration, we are happy to talk through what we have seen work.

Reach us at code-b.dev/contact-us