VERTEX FRONTIER

EDITION 01 · 2026

COMPANION FIELD KIT · FREE DOWNLOAD

The Python Data Stack Field Kit 2026

A practical companion to the Field Guide — decision tools, a migration playbook, hidden gotchas, and benchmarks the article didn't have room to print.

Decision Tree | Comparison Matrix | Migration Playbook | Hidden Gotchas | Cost Worksheet

Vertex Frontier · Knowledge Series

Direct download — no email required.
For data engineers, analysts & technical leads.

DOWNLOAD & USE

PART 01 / 03

Pick Your Library in 60 Seconds

Most teams lose a week debating libraries before checking the one variable that actually decides the answer: data size. This decision tree collapses the trade-offs into a single scan. Read your row, follow the arrow, ship the recommendation.

STEP 1 — SIZE YOUR WORKLOAD

? How many rows, and what fits in RAM?

SCALE

< 1M rows

↓

Pandas 3.0

The ecosystem depth (matplotlib, scikit-learn, seaborn) outweighs any speed gap. Pandas 3.0's copy-on-write + Arrow strings closed most of the old pain. Don't pay migration tax you won't earn back.

SCALE

1M – 20M rows

↓

Polars

Multi-core Rust engine starts pulling 5–10× over Pandas at this scale. Lazy API drops redundant scans. Syntax is close enough to Pandas to learn in an afternoon.

SCALE

20M+ rows or out-of-RAM

↓

DuckDB or Dask

DuckDB if the job is fundamentally SQL (joins/aggs on files). **Dask** if you must keep the Pandas/scikit-learn API and need to scale across machines. Mix the two only when one job legitimately needs both.

SIGNAL 01 — RUNTIME CREEP

A job that took seconds now takes minutes.

Trigger: 5× slowdown over 6 months on the same code.

SIGNAL 02 — RAM CRASHES

Script dies on a file that isn't unusually large.

Trigger: Out-of-memory on < RAM × 0.5 file size.

SIGNAL 03 — NOTEBOOK PAIN

Kernel restarts mid-groupby or mid-join.

Trigger: 2+ restarts per session on the same workload.

SIGNAL 04 — AVOIDANCE

Team starts skirting the dataset, sampling when they shouldn't.

Trigger: Ad-hoc questions go unanswered because the data is "too slow".

Rule of thumb: hit any *two* signals at once → it's time to benchmark a different engine on the real workload, not in the abstract.

PART 02 / 03

The Comparison Matrix

A side-by-side scan of the six engines that matter in 2026 — what each one is actually built for, where it breaks, and how much pain a migration will cost you. Friction is rated 1 (drop-in) to 5 (full rewrite of mental model).

LIBRARY	SWEET SPOT	ENGINE MODEL	MEMORY BEHAVIOR	FRICTION	ECOSYSTEM
Pandas 3.0 Jan 2026 · Arrow-backed	< 1 GB, notebook-scale	Single-threaded, in-memory, copy-on-write	Silent duplication removed by CoW	1 / 5	Deepest
Polars Rust · Lazy + eager	1 GB – 50 GB, CPU-bound transforms	Multi-core, query optimizer, lazy by default via scan_	Streaming engine (1.37+) spills, still RAM-hungry vs DuckDB	3 / 5	Maturing fast
DuckDB Embedded OLAP DB	SQL jobs on files/DFs, > RAM datasets	Columnar, vectorized, serverless	Purpose-built out-of-core buffer manager	3 / 5	SQL-first
Dask Pandas-compatible distributed	> RAM, multi-machine, Pandas API required	Task-graph scheduler wrapping Pandas partitions	Partitions spill across cluster nodes	2 / 5	Pandas-native
PyArrow Columnar memory layer	Interop bridge, not a primary tool	Zero-copy exchange format	N/A (it's the format, not the engine)	1 / 5	Underlies all of the above
Ibis Universal DataFrame API	Teams expecting backend changes	Compiles to 20+ backends (DuckDB, Polars, Snowflake…)	Inherits the backend's behavior	4 / 5	Portable by design

GITHUB ETL · 400 GB NIGHTLY

8× faster

90 min → 11 min. Cut from 128 GB to 32 GB instance. ~75% cloud bill reduction. Documented by GitHub engineers.

FINQORE · REPORTING PIPELINE

60× faster

8 hours → 8 minutes. Postgres → DuckDB. Columnar scan vs row-oriented transactional engine.

140 GB PARQUET / 32 GB RAM

13× less RAM

DuckDB peak memory vs Polars. Out-of-core buffer manager built from day one.

AWS R5 PRICING, NIGHTLY JOB

95%+ cost cut

$3.02 \rightarrow 0.13$ per run. $\sim 1,100/yr \rightarrow$ 46/yr for the same logic, smaller instance + shorter runtime.

COLD-START OVERHEAD NEW

~200× faster boot

DuckDB ~150 ms to first query vs Spark ~30 s cluster warmup. Decisive for ad-hoc + low-latency jobs.

IBIS BACKEND SWAP NEW

~0 lines changed

Same expression, point at DuckDB locally then Snowflake in prod. The trade-off: a new abstraction layer to maintain.

PART 03 / 03

Migration Playbook & Hidden Gotchas

A migration is not a port — it's a bet on a new default. These three phases, kill switches included, fail fast if the engine you picked turns out to be the wrong one. **Skip any phase and you'll find out about the wrong bet in production.**

PHASE 01

Audit

1. **Measure first.** Profile the current job on real data — wall time, peak RSS, the 95th-percentile run. Write these numbers down. You'll need them to prove the migration worked.
2. **Identify the real bottleneck.** CPU-bound (transforms)? I/O-bound (loading)? RAM-bound (out-of-memory)? Each maps to a different engine.
3. **Inventory Pandas-API dependencies.** List every `.loc`, `.reindex`, datetime-indexed resample, and scikit-learn hand-off. These are the lines that won't port cleanly to Polars or DuckDB.

> KILL SWITCH
> If the audit shows the bottleneck is I/O (CSV parsing, network fetch), no DataFrame engine will fix it. Stop here — fix the I/O first.

PHASE 02

Pilot

1. **Port one job end-to-end.** Not a toy dataset — the real one. Compare output row counts and totals against the Pandas version, not just speed.
2. **Benchmark on production-equivalent hardware.** A laptop benchmark misleads. Polars on 8 cores looks great until you discover prod runs on 2 cores.
3. **Measure engineer-hours honestly.** If the pilot took 3 days instead of the estimated 1, multiply the full migration estimate by 3 — before committing.

> KILL SWITCH
> If the pilot doesn't deliver > 3× speedup OR > 50% RAM reduction, the migration isn't worth the maintenance cost. Roll back.

PHASE 03

Cut Over

1. **Run both in parallel for two weeks.** Same inputs, same schedule. Alert on output diff > 0.01%. This is where silent schema drift gets caught.
2. **Downsize the instance deliberately.** Most teams forget this step and lose the cloud savings even after a successful migration. Re-measure, then shrink.
3. **Document the new default.** Write a one-page README: when to use the new engine, when to fall back to Pandas, and which gotchas below apply.

> KILL SWITCH
> If two weeks of parallel runs surface more than 1 material output diff per week, the engines are not equivalent for this workload. Investigate before flipping traffic.

Cost-Savings Worksheet `paste your own numbers`

```
Monthly_Savings = (old_hours × old_rate × runs_per_month)
                - (new_hours × new_rate × runs_per_month)
```

Example: 1.5 hr × $2.016/hr \times 30 runs$ =90.72/mo old. 0.25 hr × $0.504/hr \times 30 runs$ =3.78/mo new. Savings: $86.94/mo (96%).

Don't forget: subtract the engineer-hours cost of the migration itself, amortized over expected lifetime. A 2-week senior-engineer rewrite costs ~$5k$–8k — payback must clear that bar inside 12 months.

01 **Polars lazy is opt-in, not default.**

`read_csv()` loads eagerly. `scan_csv()` builds a lazy plan that only executes on `.collect()`. If you reach for `read_` out of Pandas habit, you lose the query optimizer and most of the speedup. The fix is one rename, but you have to know to make it.

02 **DuckDB memory_limit uses SET, not PRAGMA.**

In the Python API, the correct incantation is `con.execute("SET memory_limit='50GB'")` and `con.execute("SET temp_directory='/path/to/ssd'")`. `PRAGMA` is for SQLite-style metadata only — silently no-ops here. Setting it wrong means your out-of-core job runs out of memory anyway.

03 **Dask has real coordination overhead on small data.**

A job that fits in memory will often run *slower* under Dask than under plain Pandas, because the task-graph build + scheduler overhead exceeds the parallelization gain. Rule: don't reach for Dask until single-machine RAM is genuinely the ceiling.

04 **Polars string cache must be enabled for joins.**

Categorical joins on Polars require `pl.enable_string_cache()` at the top of the script. Without it, two DataFrames built independently with the same categories won't join correctly — they have separate enum hierarchies. The error message is cryptic. Set it globally once per session.

05 **Arrow-backed Pandas breaks some scikit-learn calls.**

Pandas 3.0's Arrow string dtype is faster, but a handful of older scikit-learn transformers (`OneHotEncoder` pre-1.4, some `ColumnTransformer` paths) still expect NumPy object arrays. If you hit a `TypeError` on a pipeline that used to work, the quick fix is `df.astype(str)` at the boundary — not a downgrade of Pandas.

Field Kit Recap `the 7 things to remember`

01. Size, not preference, decides the library.
02. Pandas 3.0 closes the gap below ~1 GB — don't migrate prematurely.
03. Polars wins on CPU-bound transforms above 1 GB, with the lazy API.
04. DuckDB wins on SQL jobs and out-of-RAM datasets.
05. Dask wins when Pandas-API compatibility is non-negotiable.
06. Dollar savings usually outpace speed savings — smaller instance + shorter run.
07. Always benchmark on production-equivalent hardware before committing.

VERTEX FRONTIER

Read the full **Field Guide** on Vertex Frontier — benchmarks, case studies, the cloud-cost math, and the honest trade-offs behind every recommendation in this kit.