My 10 Most-Read Posts for Databricks Engineers + 2 Hands-On Labs (1M+ Views)

Jakub Lasak - Databricks Data Engineer (ex-Uber, 5+ years)

These are the posts that resonated most with over 1 million data engineers.

What's inside:

🎯 **Interview Prep** (4 posts) Ace senior-level Databricks interviews

⚙️ **Technical Playbooks** (2 posts) Production-grade solutions you'll reference

📈 **Career & Growth** (2 posts) Navigate from $100k to $400k+

😅 **Community & Humor** (2 posts) You're not alone in this

🛠️ **Hands-On Labs** (2 projects) Build real pipelines on Databricks Free Edition

---

Get more: dataengineerwiki.substack.com

Table of Contents

🎯 Section 1: Interview Prep

Post 1: How Delta Lake Achieves ACID Without a Database

Interview for a $200k Databricks Data Engineer role: "Delta Lake has no database, just files. How does it guarantee ACID transactions?" 85% answer "transaction logs" and get rejected.

What's the REAL answer?

This question separates engineers who've just used Delta Lake from those who understand its internals. Most candidates know Delta has ACID properties but can't explain HOW file-based storage achieves database-level guarantees without a centralized coordinator.

Here's what separates rejection from an offer:

⚠️ The Junior (Rejection) Answer

"Delta Lake uses transaction logs to track changes to the table."

This shows you've read the docs but don't understand the mechanism.

✅ The Senior (Offer-Winning) Answer

"Delta Lake uses optimistic concurrency control with atomic file operations in the _delta_log directory. Each commit attempts to write a numbered JSON file - if two jobs race, only one succeeds atomically. Linearizable reads from the log guarantee consistent snapshots without locks."

⚙️ Transaction Log Mechanism

Junior thinking: "Changes are written to JSON files in _delta_log"

Senior understanding: Each transaction gets a monotonically increasing version number (000000.json, 000001.json, etc.). The log is the source of truth - Delta reconstructs table

state by replaying these immutable entries. Cloud storage's atomic PUT operation ensures only one writer succeeds when committing the same version number.

This is why Delta can guarantee consistency without a database - the file system itself provides atomicity.

🔍 Optimistic Concurrency Control

Junior thinking: "Multiple jobs can write simultaneously"

Senior understanding: Delta doesn't lock - it assumes writes succeed. Each job reads the current version, performs changes, then attempts to commit the next version atomically. If another job already wrote that version, the commit fails with ConcurrentAppendException. The losing job must retry with conflict resolution.

Real-world impact: Enables high-concurrency writes (10+ simultaneous jobs) without centralized coordination overhead.

💡 Conflict Detection & Resolution

Junior thinking: "Delta handles conflicts automatically"

Senior understanding: Delta ensures serializable isolation across the table by detecting conflicts at the file level. A commit fails if files it read were modified by a concurrent transaction, forcing a retry.

Appending distinct files to the same partition can succeed, but a conflict occurs if one job modifies a file another has read. Partitioning improves concurrency by physically separating files, which reduces the probability of these conflicts.

Post 2: Debug OOM Errors Like a Senior Engineer

Final round at FAANG for a Senior Databricks Data Engineer role. The interviewer asks: "Your main job just hit an OOM error. How do you debug it?'"

(Hint: The offer-winning answer is in the Spark UI).

This question separates engineers who guess from those who diagnose.

A junior asks for a bigger cluster. A senior opens the Spark UI.

Here's the offer-winning answer:

❌ **The Junior (Rejection) Answer**

"I'd increase executor memory or get a bigger cluster."

This is a costly guess. It treats the symptom, not the cause, and signals that you solve problems by spending money instead of thinking.

✅ **The Senior (Offer-Winning) Answer**

"My first step is always diagnosis, not action. I go straight to the Spark UI for the failed job to find the evidence. The goal is to determine if the root cause is true memory pressure or, more commonly, data skew."

--

⚙️ **Step 1: Analyze the Executors Tab**

"This page is my ground truth. I'd check the aggregated metrics for three red flags:"

Spill (Memory/Disk): Any spill > 0 GB is a major warning sign. It means Spark used disk when it ran out of RAM, a classic sign of an impending OOM.

GC Time: If GC time is >10% of total task time, the executor is fighting for memory instead of working. This signals high memory pressure.

Task Time (Min/Median/Max): The clearest indicator of data skew. If max task time is >5-10x the median, a few tasks are doing all the work.

--

💭 **Step 2: Formulate a Hypothesis**

"Based on those metrics, I can distinguish between the two main scenarios:"

Scenario A: It's Data Skew.
Evidence: One or two executors show massive Spill/GC Time while others are fine. Task Time max is huge compared to the median.

Fix: "This is a data distribution problem, not a memory problem. I'd fix it by salting keys or ensuring Adaptive Query Execution (AQE) is enabled to handle the skew."

Scenario B: It's True Memory Pressure.
Evidence: High Spill and GC time are distributed evenly across all executors.

Fix: "Before adding memory, I'd first audit the code for inefficiencies like a .collect() call, a bad UDF, or oversized data partitions. I'd only recommend a larger cluster after code optimization."

📝 **TL;DR**

- Never guess. Diagnose first in the Spark UI.

- Check the Executors tab for Spill, GC Time, and skewed Task Times.

- Differentiate between data skew and true memory pressure.

- Propose code-level fixes before recommending expensive hardware.

What's the most deceptive OOM error you've ever debugged?

Post 3: Why Your Query Takes 30s Sometimes, 5min Others

Senior Databricks Data Engineer interview. 80% of candidates fail this question: 'A 500GB query takes 30s sometimes, 5min others. Why?'

Most candidates guess "network issues" or "cluster was busy" without diagnosing actual job behavior. This separates engineers who systematically use Spark UI from those who guess. Here's the offer-winning approach:

⚠️ The Junior (Rejection) Answer

"Maybe the cluster was busy with other jobs, or network was slow, or autoscaling was happening."

This fails because you're guessing at external factors without diagnosing what the job actually did.

✅ The Senior (Offer-Winning) Answer

"Inconsistent performance with same query/data suggests environmental factors. I'd check three things in Spark UI: cache state, shuffle spill variance, and AQE plan differences between runs."

⚙️ Cause 1: Cold vs Warm Cache

Junior thinking: "Databricks caches automatically"

Senior understanding: First run reads from S3/remote storage (slower), subsequent runs use local SSD disk cache (significantly faster). The performance difference can be substantial for large datasets - a query reading 500GB from remote storage can take several times longer than the same query reading from local disk cache. Check Spark UI Storage tab - if "Cached Partitions" = 0 on slow runs, it's cold cache.

🔍 **Cause 2: Shuffle Spill Variance**

Junior thinking: "Same query = same shuffle"

Senior understanding: Concurrent workloads steal executor memory, causing shuffle spills on some runs. Check Spark UI Stages tab for "Spill (Memory)" and "Spill (Disk)".

Evidence: Fast run = 0 GB spilled. Slow run = 50 GB spilled because another query consumed 200 GB of cluster memory.

💡 **Cause 3: AQE Plan Changes**

Junior thinking: "Same query = same plan"

Senior understanding: AQE changes plans mid-execution. Sometimes converts to broadcast join (30s), sometimes keeps sort-merge join (5min).

Check: Compare plans in Spark UI SQL tab. Fast run shows broadcast join (no shuffle), slow run shows sort-merge (expensive shuffle).

📝 **TL;DR:**

• Cold vs warm cache causes substantial performance variance
• Concurrent queries cause shuffle spill - check Spark UI Stages
• AQE changes plans based on runtime stats
• Systematic Spark UI diagnosis beats guessing

What's the most frustrating performance inconsistency you've debugged? 👇

Post 4: Design a Medallion Architecture (System Design)

Senior Databricks Engineer Interview: "Design a Medallion architecture for 1TB/day of data with a 1hr SLA". How would you answer to get the job?

(Hint: True streaming is the trap).

This isn't about one "right" answer. It's a test of architectural thinking - balancing cost, performance, and complexity.

A junior engineer lists tools.

A senior discusses trade-offs.

Here's the breakdown that gets you the offer.

⚙️ **Trade-off 1: Ingestion - True Streaming vs. Micro-batch**

The 1-hour SLA is the key constraint. Your first decision is how to ingest the data to meet it.

The Trap: Choosing true streaming by default. A continuous, always-on streaming job is complex to manage and can be expensive. For a 1-hour SLA, it's often over-engineering.

The Winner 🏆: A robust micro-batch approach using *Auto Loader* with `*Trigger.AvailableNow*`, scheduled to run every 15 minutes. This provides near-real-time data well within the SLA, but with lower costs and simpler operations than a 24/7 stream.

Senior Insight: You show cost-awareness by choosing the simplest solution that meets the business need.

🏗️ **Trade-off 2: Orchestration - Manual Jobs vs. Delta Live Tables (DLT)**

Once data is in Bronze, how do you manage the flow to Silver and Gold?

The Trap: Using notebooks in Workflows. This forces you to manually manage dependencies and error handling, making the system brittle at scale.

The Winner 🏆: *Delta Live Tables* (*DLT*). It's a declarative framework where you define the transformations and DLT manages the orchestration, dependency graph, and infrastructure for you. Built-in `Expectations` handle data quality automatically.

Senior Insight: You prioritize reliability and low operational overhead, building systems that are easy to maintain.

💡 **Trade-off 3: Gold Layer - One Big Table vs. Aggregates**

How do you structure the Gold layer to be both flexible for ad-hoc queries and fast for critical BI dashboards?

The Trap: Believing it's an either/or choice. Relying only on a single, massive 1TB+ Gold table will eventually fail a query SLA. Building only aggregates limits exploration for power users.

The Winner 🏆: A hybrid approach.
1. Create a comprehensive, well-optimized Gold table using *Liquid Clustering* to serve as the single source of truth.
2. Build a small set of pre-aggregated tables from Gold that are purpose-built to power the top 5-10 most critical (and slowest) BI dashboards.

Senior Insight: You serve both analysts and executives with a pragmatic, multi-layered solution.

What's the #1 mistake you see engineers make when designing Medallion architectures?

⚙️ Section 2: Technical Playbooks

Post 5: Cluster Sizing for 500GB Workloads

Say we're processing a dataset of 500 GB in Databricks. How would you configure the cluster to achieve optimal performance?

If you've ever waited hours for a Spark job to finish or wasted credits on an oversized cluster, you know how critical proper sizing is.

Here's a practical guide to configuring your Databricks cluster for a 500 GB workload, based on best practices and real-world experience.

📊 **Partitioning: Focus on Shuffle Partitions, Not Table Partitions**
For datasets under 1 TB, avoid manual table partitioning. Instead, leverage Delta Lake's Liquid Clustering and auto-compaction features to optimize file sizes and layout without the complexity of static partitions.

Shuffle partitions are where Spark parallelizes your work during joins, aggregations, and shuffles.

Rule of thumb: Aim for 128–256 MB per shuffle partition.

Calculation:
500 GB × 1024 MB/GB ÷ 200 MB/partition ≈ 2560 partitions

Set this with spark.sql.shuffle.partitions = 2560. Adaptive Query Execution (AQE) will fine-tune this at runtime.

⚙️ **Cluster Sizing: Executors, Cores, and Memory**

Cores per executor: Keep it between 2–5 cores. I recommend 4 cores per executor for balanced CPU utilization and manageable JVM overhead.

Total cores: You don't need one core per partition. Instead, size your cluster to process many partitions in parallel without overwhelming resources.

Example: 16 worker nodes × 8 cores each = 128 total cores. This means 128 partitions processed concurrently, with the rest queued.
-
Memory per core: Allocate 4–8 GB RAM per core depending on workload complexity.

Example: 128 cores × 6 GB = 768 GB total executor memory. Spread across 16 nodes ⟶ ~48 GB RAM per node for executors (leaving headroom for OS and overhead).

Node type: Use memory-optimized instances for shuffle-heavy workloads. For cost savings, consider spot instances with autoscaling enabled.

⚖️ **Handling Data Skew**
Detect skew:
Use the Spark UI to spot tasks that take significantly longer than others.

Mitigate skew:
Enable Adaptive Query Execution (AQE) (default in modern Databricks runtimes). AQE dynamically optimizes skewed joins and shuffle partitions.

If AQE isn't enough, apply salting: add a random prefix to skewed keys before shuffle joins, then remove it after.

💡 **Final Tips**:
- Don't trust defaults. Always check and tune spark.sql.shuffle.partitions based on your data size
- Monitor jobs. Look for garbage collection pauses, data spills, and skewed task durations
- Iterate. Start with a conservative cluster size and scale based on observed metrics

📝 **TL;DR**
- Shuffle partitions: ~2560 (200 MB each)
- Executors: 4 cores per executor
- Total cores: ~128 (e.g., 16 nodes × 8 cores)
- Memory per core: 4–8 GB (start with 6 GB)
- Node type: Memory-optimized, spot instances
- Skew handling: Enable AQE, use salting if needed

TLDR:

databricks

Cluster Sizing Cheat Sheet

Configuration Aspect	Rule of Thumb
Partitioning	For tables < 1TB, use Liquid Clustering & auto-compaction, not manual partitioning.
Shuffle Partitions	Target 128–256 MB per shuffle partition using spark.sql.shuffle.partitions.
Executor Cores	Assign 2–5 cores per executor; start with 4 for a balanced load.
Total Cores	Ensure enough total cores for high parallelism (e.g., ~128 for a 500 GB job).
Memory per Core	Allocate 4–8 GB RAM per core; use 6 GB as a baseline.
Node Type	Use memory-optimized nodes for shuffles; enable spot instances & autoscaling to save costs.

Post 6: Choose the Right Ingestion Tool

Imagine your CFO asks to add a new data source to Databricks. How do you pick the tool that won't require a painful refactor in 6 months when the data volume 10x's?

This isn't just a coding question. It's a career question.

A junior engineer picks the tool that's easiest today (*spark.read*). A senior engineer picks the tool that can handle tomorrow's 10x scale.

That forward-thinking choice is what gets you recognized.

To make the right call every time, I use the simple decision guide in the photo. Save it for your next data ingestion project.

Here's how each choice holds up against the "10x scale" problem:

🏆 **Auto Loader (The Scalable Default)**
This is your future-proof choice. It's built for the 10x data volume problem, handling massive scale and schema changes automatically. This is the tool that lets you build it once and sleep at night.

Use when: You need a production-grade, fault-tolerant solution for files arriving from cloud storage.

🤖 **SQL COPY INTO (The Simple Bulk Load)**
A straightforward, idempotent SQL command. It's reliable for predictable batch jobs today, but it lacks the advanced features of Auto Loader for handling evolving, high-volume sources.

Use when: You have a simple, one-time load where future scale isn't the primary concern.

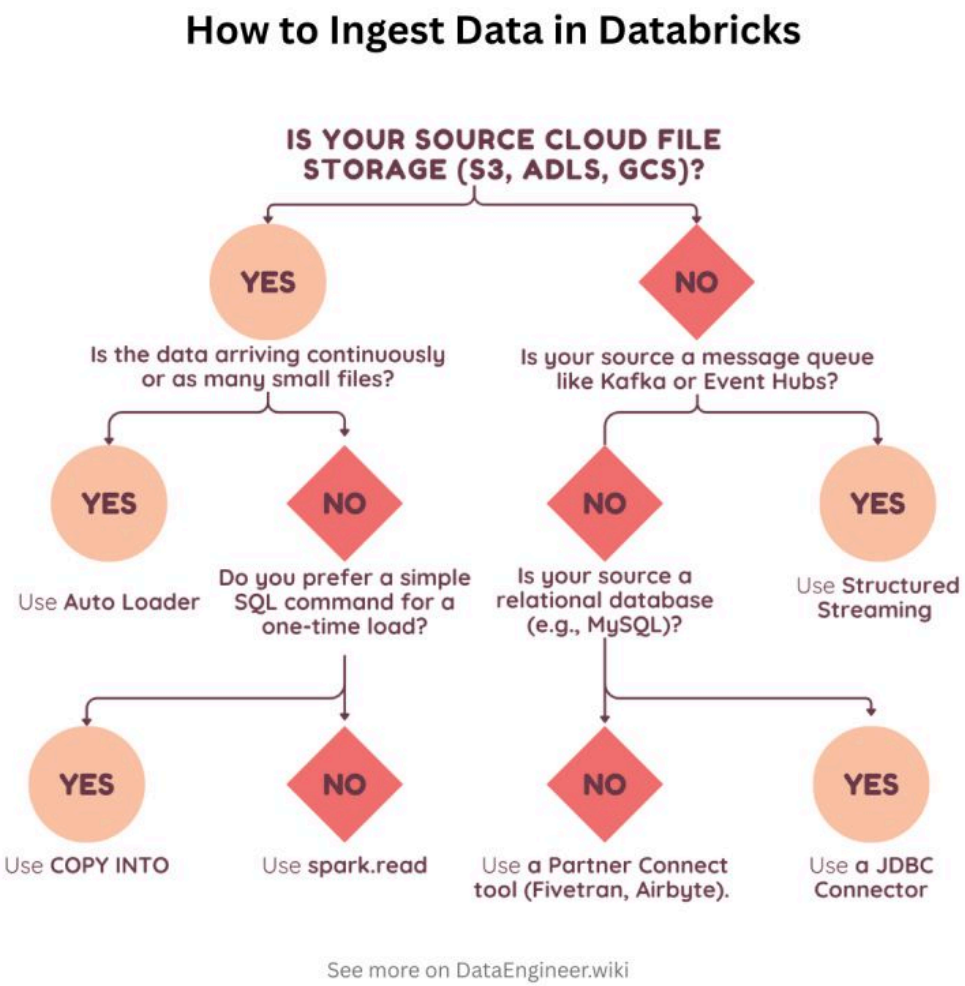
⚠️ **spark.read (The Scalability Trap)**
This is the ultimate trap. It works perfectly for your initial 10GB dataset but will cripple your system and burn cash when it scales to 100GB. It lacks the features needed for reliable, large-scale ingestion.

Use when: Only for interactive analysis in a notebook. Never for a production pipeline that needs to scale.

🚀 **Structured Streaming (For Message Queues)**
The right tool for true streaming sources. It's built for low-latency, high-volume message data from systems like Kafka, which is a different kind of scaling problem.

Use when: Your source is a message bus, not files in cloud storage.

TLDR:



📈 Section 3: Career & Growth

Post 7: Career Progression: $100k to $400k+

What's the difference between a $100k, $200k, and $400k+ Databricks Data Engineer? Asked this in a community with engineers at all levels.

Junior Databricks Data Engineer (**$80k-$120k**, **0-2 years**)
• First "I shipped to prod" moment feels incredible
• Googling Spark optimization daily
• Every ticket is learning (and terrifying)
• Still figuring out why .collect() broke everything
• Imposter syndrome is real, but you're learning fast

Mid-Level (**$120k-$180k**, **2-4 years**)
• You stop panicking when pipelines fail at 3 AM
• Can debug most issues without help
• Built enough pipelines to recognize patterns
• Explaining Delta Lake to analysts confidently
• Spark UI finally makes sense (mostly)

Senior (**$180k-$250k**, **4-7 years**)
• You design architectures others build on
• Data skew? AQE? Partitioning? Handled it dozens of times
• Reviewing code and mentoring juniors
• Stakeholders ask YOU when issues arise
• Biggest shift: Solving business problems, not just technical ones

Lead (**$250k-$400k+**, **7+ years**)
• You set technical direction for the data platform
• Decisions impact company-wide strategy
• Juggling architecture, people management, stakeholder alignment
• More meetings than code (and that's okay)
• Your GitHub drops, but team velocity increases

Biggest trap at each level:
• Junior: Trying to know everything before shipping
• Mid: Getting comfortable and stopping learning
• Senior: Overengineering because you CAN, not because you SHOULD
• Lead: Losing touch with code

What drives growth:
• Juniors: Ship code. Break things. Ask questions.
• Mids: Own end-to-end projects. Learn adjacent skills.
• Seniors: Mentor others. Design for scale. Think business impact.
• Leads: Build systems that work without you. Stay technical enough to earn respect.

What gets old fast at every level:
• Junior: Feeling like you're bothering people with questions (you're not)
• Mid: Being stuck on maintenance work when you want to build new things
• Senior: Endless context-switching between firefighting and architecture reviews
• Lead: Meetings about meetings. Politics. Losing time to actually build.

What brings satisfaction:
• Junior: First pipeline running without errors
• Mid: Building something users rely on daily
• Senior: Watching someone you mentored surpass you
• Lead: Platform you designed scaling to millions of events per second

Post 8: The Databricks Data Engineer's Paradox

The Databricks Data Engineer's Paradox: The better you are at your job, the less people realize they need you.

When your pipeline is struggling:

- Dashboards show data from two days ago
- Jobs crash with OOM errors at 3 AM
- Cloud costs spike to $10k/month

Everyone knows they need a Senior Data Engineer to fix it.

But when you're excelling:

- 5TB ingestion lands by 7:00 AM sharp
- Autoscaling keeps costs under $500
- Queries on 1B+ rows return in seconds

People think: "Databricks just runs itself, right? Why do we need a Senior Data Engineer?"

The Truth:

- Your invisible work handled the data skew in that massive join
- Your logic optimized the Z-Ordering for those dashboards
- Your governance prevented a PII leak that could have cost millions

To all the Databricks Engineers: Your impact might be invisible, but it's indispensable.

😅 Section 4: Community & Humor

Post 9: If Databricks Error Messages Were Honest

If Databricks Error Messages Were Honest:

Error: java.lang.OutOfMemoryError: Java heap space
Honest Version: "You tried to .collect() a 2TB DataFrame, didn't you? You knew this was wrong, but you did it anyway. Now think about what you've done."

Error: AnalysisException: Path does not exist.
Honest Version: "I've looked in the 3 places you might have meant by that S3 path typo. I'm not looking anymore. It's your problem now."

Error: SparkException: Job aborted due to stage failure.
Honest Version: "One of my 1,000 children (tasks) has thrown a tantrum and I'm shutting this whole thing down. I'm not telling you which one. Good luck."

Error: DeltaConcurrentModificationException
Honest Version: "You and three other jobs tried to write to this table at the same time. This isn't Google Docs, folks. Form a line."

Error: SparkException: Failed to merge fields 'user_id'. Incompatible types StringType and IntegerType.
Honest Version: "The upstream team decided 'user_id' is now a string. Your entire Medallion architecture, which assumed it was an integer, respectfully disagrees. This is now a knife fight.

Post 10: How to Irritate a Databricks Data Engineer

How to irritate a Databricks Data Engineer? 9 quick ways that always work:

1️⃣ Call their perfectly architected Medallion Lakehouse a "data swamp," then ask for admin access to "just run a few things."

2️⃣ Look them dead in the eye and ask them to .collect() a 2TB DataFrame because you "want to see all the data in one place."

3️⃣ Tell them "Just use the biggest cluster, we need it to be fast." Then, forward them the $5,000 cloud bill with a "?"

4️⃣ Wait until their DLT pipeline is finally stable, then ask them to "just quickly add" a new data source that changes its schema twice a day.

5️⃣ Request a CSV export of a 50-billion-row Delta table so you can "do a quick VLOOKUP in Excel."

6️⃣ Complain the dashboard is slow after running SELECT * with no filters on a 5TB Delta table.

7️⃣ Ask why the pipeline failed, when the _rescued_data column clearly shows the source team sent corrupted JSON. Again.

8️⃣ After they spend all week debugging a memory spill, helpfully ask, "Did you try using a bigger cluster?"

9️⃣ Name your code Untitled_Notebook_2025_09_24_final_v7_prod(1).ipynb and ask them to deploy it.

🛠️ SECTION 5: HANDS-ON LABS

Two free projects to practice what you've learned. Both run on Databricks Free Edition.

Lab 1: Build a Production-Grade DLT Pipeline

Build a Medallion pipeline for an apparel company with streaming data.

What you'll build:

- Bronze ⟶ Silver ⟶ Gold pipeline with DLT
- Streaming data ingestion with Auto Loader
- Data quality enforcement with DLT Expectations
- SCD Type 2 for slowly changing dimensions
- BI-ready aggregate tables

Prerequisites:

- Basic PySpark/SQL knowledge
- Databricks workspace (Free Edition works)

Duration: 1-3 hours

Get started: github.com/jrlasak/databricks_apparel_streaming

Lab 2: Data Engineer Associate Certification Prep

End-to-end pipeline covering all 5 exam sections.

What you'll build:

- Complete Medallion architecture
- Auto Loader & COPY INTO ingestion
- Unity Catalog governance setup
- Multi-task job orchestration
- Role-based access control

Prerequisites:

- Basic Databricks familiarity
- Databricks workspace (Free Edition works)

Duration: 2-4 hours

Get started: github.com/jrlasak/databricks_optimization_techniques

---

Want More?

Subscribe to my newsletter for:

- Weekly senior-level interview strategies
- Production-grade technical solutions
- Career insights from a Databricks engineer (ex-Uber)

Join 10,000+ Databricks engineers: → dataengineerwiki.substack.com

---