Engineering

SHARE

FEBRUARY 12, 2025

Cutting over: Our journey from AWS Aurora MySQL to TiDB

Zander Hill

Zander is a software engineer and former engineering lead at Plaid who founded the Online Storage team, and has demonstrated expertise in large-scale database systems. Zander has a track record of reducing downtime, cutting costs, and leading teams through complex migrations in timeframes that vendors tell him is too ambitious and a bit crazy.

Switching database platforms is one of the most daunting challenges in modern infrastructure. Database platform replacements demand rigorous planning to maintain data consistency, ensure uptime, and preserve performance and feature compatibility. But in January of 2023, we kicked off a “Future of SQL” project to lay the online relational database foundation for Plaid’s growth over the next 5 to 10 years. We’ve now transitioned the majority of our services from AWS Aurora MySQL to TiDB with minimal service disruption and are seeing the benefits from our investment.

In this post, we share our process for approaching and delivering a project that’s at the core of Plaid’s reliability and engineering velocity, with the goal of providing a roadmap for others facing similar challenges in infrastructure and company-wide replatforming.

Below, we’ll explore our motivation for moving to TiDB, how we transitioned each service, and how we’ve refined and accelerated our process over time. We hope our journey can serve as a blueprint for other organizations looking to modernize their data infrastructure.

Motivation

As the founder of the Storage Team at Plaid, I saw the investment we were putting into Aurora MySQL and the limitations we faced compared to other systems we self-host. Plaid’s Storage Team provides a scalable and reliable platform for storing online data at Plaid and focuses on investments in relational, NoSQL, and caching storage systems.

I devised the structure for assessing our alternatives along with Joy Zheng, our Architecture Lead, and our deep technical expert on databases Mingjian Liu, and then set an ambitious timeline for the team that we would do a quarter of research, a quarter of prototyping, and aim to complete our service transitions to a new platform before the Amazon’s MySQL 5.7 deprecation! To ensure the project delivered high value to the business, we had the design constraint that this project couldn’t last more than two years company-wide.

Technical Landscape

- **Total online storage stack:** ~800 servers, ~500k QPS, ~650 TB of data, and an internal SLA of 99.99%+ availability.
- **SQL platform:** ~446 servers, ~140k QPS, ~40 TB of data, 99.99+% SLA.
- **TiDB portion (current):** ~160 servers, ~170k QPS, ~70 TB of data, ~700 vCPUs across clusters.
- **6 Software Engineers** in Storage Team.

Why We’re Moving Off Aurora MySQL

Several converging reasons pushed us to find a new solution for Plaid’s primary relational data store:

1. Reliability Challenges

- Plaid’s public SLAs and internal SLOs at application layer translates to 99.995% for the database tier (applying Google SRE best practices of one extra “9” of reliability).
- Aurora’s single-writer architecture becomes a single point of failure and introduces downtime during configuration changes, scaling, restarts, and failovers.
- Requiring each team to build database expertise for reliability didn’t scale well, and we saw an opportunity to centralize that ownership and the reliability guarantees.

2. Slow Developer Velocity

- As workloads grew on Aurora, we encountered write throughput bottlenecks (e.g., ~12k writes/sec on large tables with high-row contention patterns).
- Online schema changes often require heavy-handed workarounds. Teams added new tables instead of modifying existing ones to avoid service interruptions or extended maintenance.
- We were spending two engineering years each year architecting around Aurora limitations, especially on big tables (2–10+ TB).

3. High Maintenance Burden

- We spent two engineering quarters in 2023 upgrading Aurora from MySQL 5.6 to 5.7. Each of these upgrades or routine tasks (enabling binlog replication, resizing, etc.) required minutes of downtime.

4. Sharding and Scalability Requirements

- Aurora scales very well for writes up to a point and extensively for read traffic.
- TiDB provides automatic sharding, enabling consistent performance even for write traffic.

5. MySQL 5.7 Deprecation

- MySQL 5.7 community support ended, and Aurora's EOL loomed.
- We knew MySQL 5.7 → 8.0 upgrades would be another huge effort based on Facebook's 8.0 upgrade.
- We decided to repurpose the upgrade effort into shifting platforms.

Our Timeline at a Glance

- **Q1 2023:** Evaluated options to replace Aurora MySQL 5.7.
- **Q2 2023:** Completed initial proof-of-concept and validated TiDB's functionality for Plaid.
- **Q3 2023:** Finalized RFC, secured financial and leadership buy-in, and built the initial TiDB infrastructure.
- **Q4 2023 → Q1 2024:** Adopted first service on TiDB. Progressed to ~17 services within a few quarters.
- **Jan 2025:** We've transitioned 41 services by the end of 2024.
- **Future:** Full transition by mid-2025.

Service Transition

Life of a Single Service

Moving a database to a new platform can be broken into many micro-steps. Our internal runbook has about 200 distinct tasks and ~20 sub-runbooks. At a high level, each service transition follows four phases: **remove incompatibilities**, **replicate data**, **validate**, and **switchover**.

1. Remove Incompatibilities

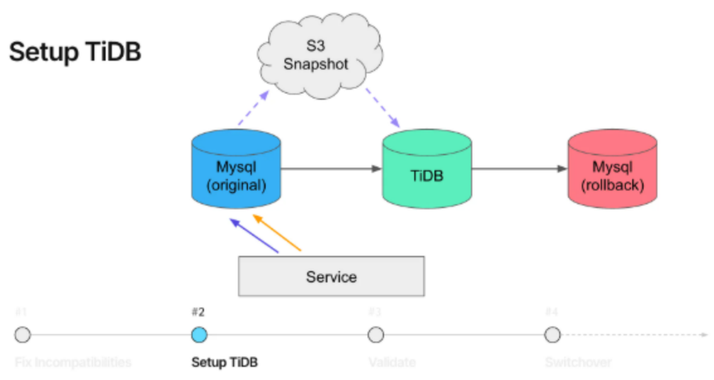
- **Enforce Primary Keys:** TiCDC (TiDB's change data capture mechanism) requires a primary key for consistent replication. If a table lacks one, we ask the service team to add it.
- **Remove Foreign Keys:** Foreign keys aren't supported in some of TiDB's ecosystem tools and can degrade performance in distributed databases. We drop them at the database layer and enforce them at the application level if needed.
- **Audit Transaction Isolation:** TiDB doesn't offer SERIALIZABLE isolation. Teams using it in MySQL must adjust application logic to handle SNAPSHOT ISOLATION (aka REPEATABLE READ) or alternative patterns.
- **Audit Auto-Incrementing Fields:** TiDB auto-increment IDs aren't globally monotonic (due to multiple nodes allocating ranges). If a service relies on strictly increasing IDs, we recommend ordering by timestamps or re-architecting ID generation logic.
- **Prove Compatibility:** Switch service over to use TiDB in automated unit and integration tests and pause schema migrations for this service until production cutover.

2. Replicate Data

Once tables and queries are compatible:

1. **Snapshot** the original MySQL database and import it into TiDB using **TiDB Lightning**.
2. **Clone** the original MySQL database to a rollback cluster using the same snapshot, giving us a fallback path in case of issues.
3. **Set up replication** among the original MySQL, TiDB, and rollback MySQL cluster.

Integrate with our "mysql/tidb/rollback mysql" cutover client, a feature-flagged approach built into a shared database driver that allows stopping traffic and pointing to a different database endpoint.



3. Validate

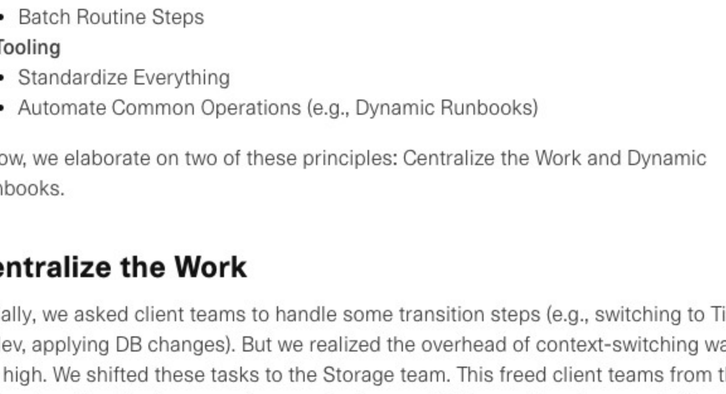
Validation is repeated at every step, and we prioritize **correctness over velocity**:

- **Data Consistency Checks:** After importing or starting replication, we run a series of different tools (e.g., TiDB's "sync-diff-inspector") to ensure data matches exactly. We've uncovered edge cases around binary primary keys and timestamp columns that required patches from PingCAP.
- **Live Query Validation:** We do dual writes and reads from Aurora to our validation cluster to compare correctness and performance. This has highlighted query plan differences where sometimes TiDB uses a suboptimal index, causing additional database load. We fix those queries by rewriting them or using index hints.
- **Performance Benchmarks:** These measurements provide higher behavioral correctness guarantees for latency, which is critical for high-traffic Tier 0 and 1 services.

4. Switchover

Perform an **atomic switch** using feature flags in the "mysql/tidb/rollback mysql" framework:

1. **Suspend writes** to Aurora, allowing replication to catch up fully.
2. **Flip and re-enable reads** on TiDB.
3. **Perform** safety and correctness validations.
4. **Flip and re-enable writes** on TiDB.



This approach avoids the reliability and consistency issues of maintaining simultaneous reads and writes on two live databases. If something fails post-cutover, we can **fail forward** onto the dedicated rollback MySQL cluster. The entire cutover can be done in ~60 seconds of write downtime, with reads remaining available and consistent throughout.

Accelerating

With dozens of services still in the queue and a lean Storage team, we've refined our strategy:

- **Principles**
  - Tackle the Hardest Services First
  - Always Have a Rollback Plan
- **Planning**
  - Centralize the Work
  - Batch Routine Steps
- **Tooling**
  - Standardize Everything
  - Automate Common Operations (e.g., Dynamic Runbooks)

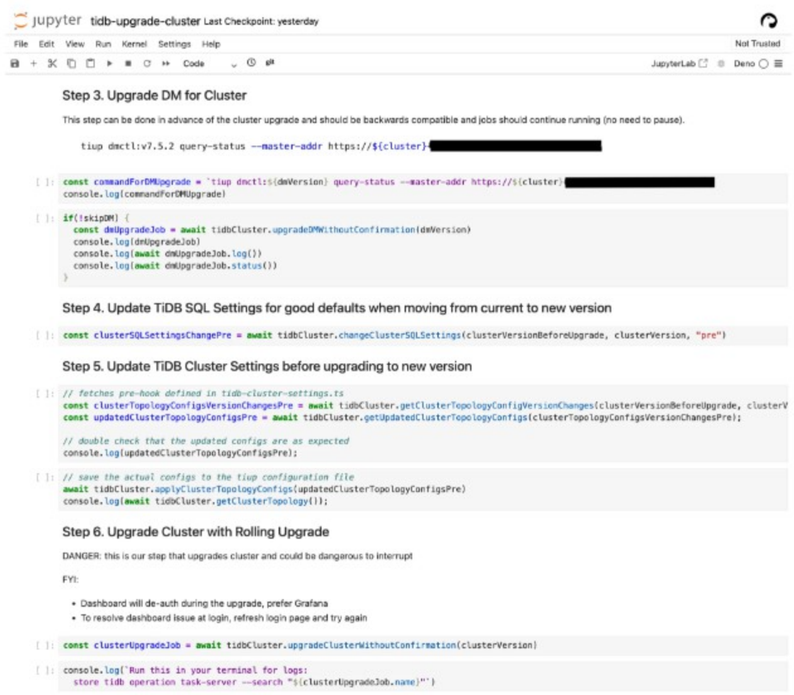
Below, we elaborate on two of these principles: Centralize the Work and Dynamic Runbooks.

Centralize the Work

Initially, we asked client teams to handle some transition steps (e.g., switching to TiDB in dev, applying DB changes). But we realized the overhead of context-switching was too high. We shifted these tasks to the Storage team. This freed client teams from the intricacies of replication or environment setup, and it let us automate repeated tasks in a uniform way. We found it improved overall organizational efficiency and reduced client team frustration.

Dynamic Runbooks

Dynamic runbooks have transformed our operations by merging the clarity of written documentation with the power of executable scripts. Instead of juggling static Google Docs and manually tracking each step, our team can now interleave Markdown instructions with TypeScript code directly in Jupyter notebooks. This approach provides explicit, self-updating process documentation while enabling engineers to run commands in context, dramatically reducing human effort and errors.



Our adoption of these runbooks was born from necessity. We used to clone a separate Google Doc for each new operation, painstakingly revising commands each time.

As the team lead, I researched the alternatives and learned of the concept of dynamic runbooks from a colleague in a larger company. While waiting for long-running tasks to complete, I prototyped a few solutions to the problem and settled on an elegantly simple solution and built a CLI to support the design pattern, then brought in teammates to help build our internal SDK that abstracts concepts like an Aurora cluster or TiDB cluster and simplified interfaces for executing complex cluster commands.

We chose Deno as our core language for the Jupyter integration because it had the following advantages over a more traditional python notebook:

1. Strong proficiency in the team/company with typescript
2. Dependency management via inline declarations
3. No virtualenv complexity
4. Built-in type safety for our 6k+ lines of runbook supporting code
5. A robust and minimal shell execution wrapper: dax
6. Wide compatibility with Deno and NPM libraries

With dynamic runbooks, we can create a template runbook that outlines the steps, sets up parameters, and references our internal SDK for execution steps. Centralizing everything in one place enables each engineer to repeatably follow and establish best practices. Operations are smoother, faster, and error-free because every step is both documented and push-button executable on the spot.

Our CLI tool for runbooks helps orchestrate parameter inputs and our SDK logs each cell's execution in Slack for auditing. This eliminates version drift in processes and gives the team real-time visibility into ongoing work. We've seen a direct uptick in velocity and correctness since implementing dynamic runbooks.

Our approach speeds up execution by **5x for our cutover phase** and by **3-4x for a service transition to TiDB (200 steps)**. We notice the greatest velocity improvement on our very standard services or ~80% of services.

We strongly encourage other organizations looking to modernize their operational workflows to adopt this pattern.

Learnings

What went well:

- **Performance:** Our benchmarking ahead of moving to TiDB was very indicative of our actual performance behavior on TiDB.
- **Rolling Upgrades and Scale-out:** We upgraded six production clusters in one week with **zero downtime**, a stark contrast to Aurora's six engineering months and 10s of minutes of downtime. Our second TiDB upgrade only took two days instead of one week and again zero downtime.
- **Online Schema Changes:** We've already made schema modifications that would have required maintenance windows in MySQL. There's no downtime needed in TiDB even on tables with 5+TB of data and tens to hundreds of billions of rows.
- **Vendor Support:** PingCAP's engineers (special shout-out to Michael Zhang) provided timely support, including new patches and recommendations. We wish all vendors achieved this level of operational excellence and customer support.
- **Communication:** The success of Plaid's transition from Aurora MySQL to TiDB was driven by clear communication and aligning the project's goals with partner team needs. By emphasizing the benefits (improved reliability, reduced KTLO, and significant cost efficiencies) and addressing key pain points like scalability and maintenance burden, we gained strong adoption and prioritization across teams. Early wins during the proof-of-concept phase and proactive stakeholder engagement further ensured alignment and support for the operation.

What could have gone better:

- **Ecosystem Tooling Polish:** Replication (DM, TiCDC) and import (TiDB Lightning) each had edge-case bugs that required patching. PingCAP has been a great partner, but each bug ate up engineering time in triage and testing.
- **Offline Export Pipeline:** Building an offline export pipeline to replace certain ETL jobs took longer than expected.
- **Query Plan Differences:** TiDB occasionally chooses a different execution plan than Aurora. Some queries ended up doing full table scans, which we discovered through our dual read procedure. We mitigated this with query hints.
- **Resource Isolation:** Resource controls in TiDB are good, but not perfect, and we've found edge cases where they insufficiently protected our systems for workload isolation.
- **Configuration Complexity:** TiDB has great tuning flexibility, but it's also difficult to reason about the interactions between multiple configuration values. We needed to read source code to better understand and recommend better simplifications and best practices guidance.

Conclusion

Thanks to the creation of dynamic runbooks, meticulous planning, and our team's operational excellence, our cutovers now average around **one week per service** (down from 3-4 weeks) and **under 60 seconds of write downtime** (down from ~5 minutes). This has allowed us to transition services at a much faster clip without sacrificing reliability. From talking to peers in the industry, we're achieving this at an industry-leading rate of adoption.

We still have more to do as we scale to the final set of MySQL workloads. But the success of our transitions has validated the core advantages we sought from TiDB: horizontally scaling without disruption, safe DDLs, better observability, and simpler operations. Most importantly, our engineering teams are sleeping better at night with fewer pages and feel safe making routine DDL changes.

No database platform replacement is easy, but with careful planning, automation, and iteration, we've shown it doesn't have to take years. If you're considering TiDB or any other modern distributed SQL database, our advice boils down to:

1. **Plan thoroughly and test early:** include validation for data correctness and performance.
2. **Automate repeated tasks:** standardize your approach, reduce manual steps, and keep a rollback path.
3. **Seek out the tough edge cases first:** it'll pay dividends in the long run.
4. **Bring your A-game to communications:** with client teams, leadership, and the vendor.

At Plaid, we're excited about TiDB's promise for the next 5 to 10 years of our storage needs. We know running databases at scale is challenging, but we are optimistic we can avoid most or all need for maintenance windows, have the tools to mitigate or avoid production incidents, and more flexibility for teams to move fast on building the products our customers love are the ultimate wins in any engineering effort.

In September, the Storage team at Plaid (led by **Zander Hill** and **Andrew Chen**) shared our experience with TiDB at PingCAP's 2024 HTAP Summit. We were honored to give a keynote on our speedy transition and practical, tactical advice for moving to TiDB. The positive response from conference attendees reaffirmed how unique our approach was, especially in terms of pace and reliability.

Thanks for reading, and stay tuned for more insights on how we're bringing predictability and flexibility to the storage layer at Plaid. If you're curious about the tactical details, feel free to reach out or tell us about your own platform transition experience.

Our journey to a new SQL platform has been a major project for the Plaid Storage team, but we couldn't have done it without strong partnerships:

- **Storage Team:** Zander Hill (Lead), Mingjian Liu (TL), Andrew Chen (EM), Lauren McCarty, Brian Xie, Seyoung Kim, Catherine Shen, and Lohit Verma (TPM).
- **Architectural Guidance:** Our Platform Architecture Lead, Joy Zheng, provided technical critique and pushed us to rigorously define our operating principles.
- **Leadership Support:** Strong advocacy based on the design principles we committed to from our Head of Platform Jipeng Han and our engineering sponsor Pranav Kantawala.
- **Our partners at Pingcap:** Michael Zhang, Logan O'Brien, Lijia Xu, Max Liu, Ed Huang.

Interested in these problems and approaches we take in the Storage team at Plaid? We're hiring.

ENGINEERING