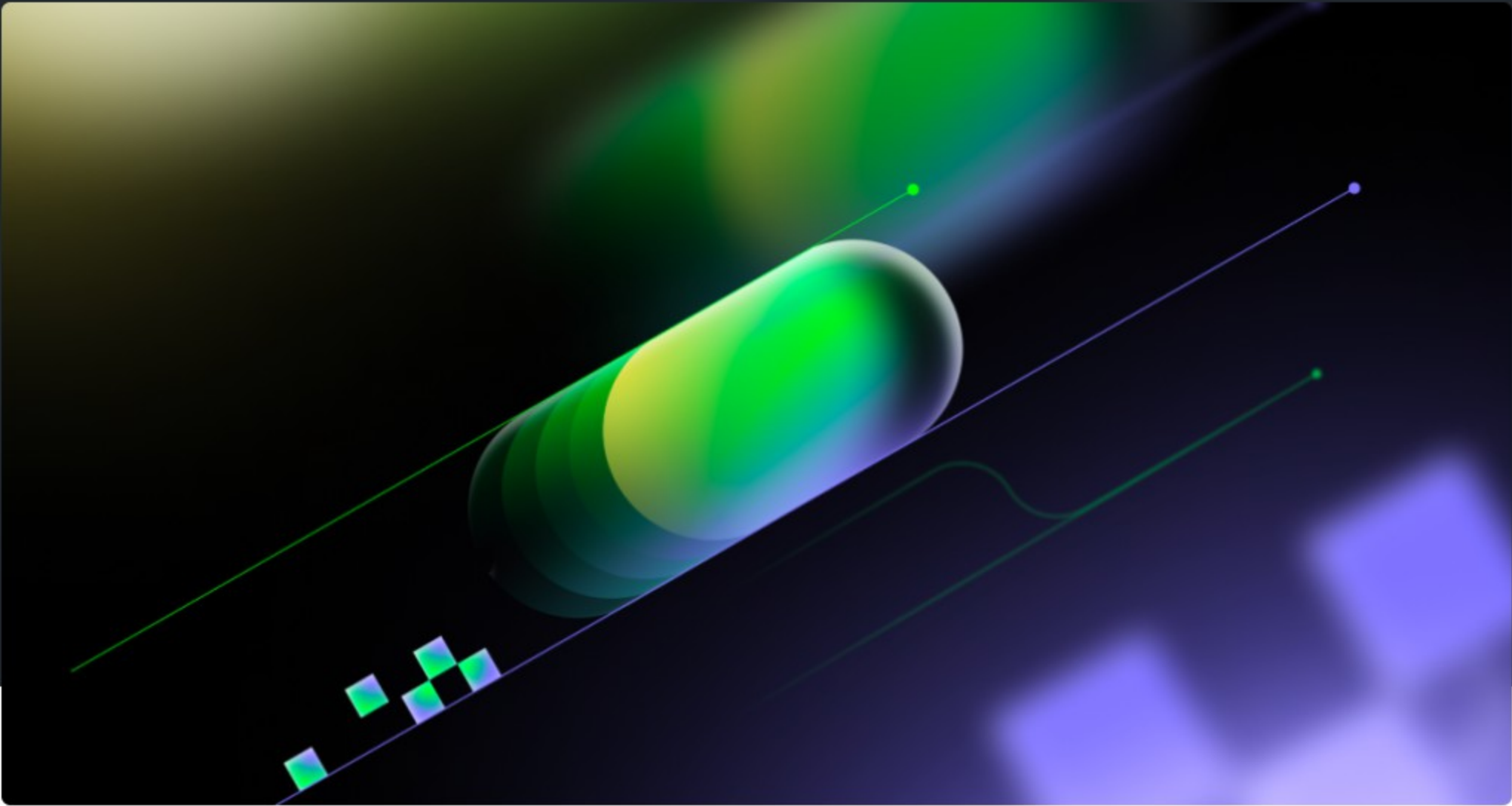


Home / Engineering / Infrastructure

Upgrading GitHub.com to MySQL 8.0

GitHub uses MySQL to store vast amounts of relational data. This is the story of how we seamlessly upgraded our production fleet to MySQL 8.0.



Jiaqi Liu, Daniel Rogart & Xin Wu

December 7, 2023 | Updated May 21, 2024 | ⌚ 13 minutes

Share:   

Over 15 years ago, GitHub started as a Ruby on Rails application with a single MySQL database. Since then, GitHub has evolved its MySQL architecture to meet the scaling and resiliency needs of the platform—including [building for high availability](#), [implementing testing automation](#), and [partitioning the data](#). Today, MySQL remains a core part of GitHub’s infrastructure and our relational database of choice.

This is the story of how we upgraded our fleet of 1200+ MySQL hosts to 8.0. Upgrading the fleet with no impact to our Service Level Objectives (SLO) was no small feat—planning, testing and the upgrade itself took over a year and collaboration across multiple teams within GitHub.

Motivation for upgrading

Why upgrade to MySQL 8.0? With [MySQL 5.7 nearing end of life](#), we upgraded our fleet to the next major version, MySQL 8.0. We also wanted to be on a version of MySQL that gets the latest security patches, bug fixes, and performance enhancements. There are also new features in 8.0 that we want to test and benefit from, including Instant DDLs, invisible indexes, and compressed bin logs, among others.

- availability
- databases
- Infrastructure
- MySQL

Table of Contents

Motivation for upgrading

GitHub's MySQL infrastructure

Preparing the journey

Upgrade plan

Ability to Rollback

Challenges

Learnings and takeaways

Conclusion

GitHub’s MySQL infrastructure

Before we dive into how we did the upgrade, let’s take a 10,000-foot view of our MySQL infrastructure:

- Our fleet consists of 1200+ hosts. It’s a combination of Azure Virtual Machines and bare metal hosts in our data center.
- We store 300+ TB of data and serve 5.5 million queries per second across 50+ database clusters.
- Each cluster is [configured for high availability](#) with a primary plus replicas cluster setup.
- Our data is partitioned. We leverage both horizontal and vertical sharding to scale our MySQL clusters. We have MySQL clusters that store data for specific product-domain areas. We also have horizontally sharded [Vitess](#) clusters for large-domain areas that outgrew the single-primary MySQL cluster.
- We have a large ecosystem of tools consisting of Percona Toolkit, [gh-ost](#), [orchestrator](#), [freno](#), and in-house automation used to operate the fleet.

All this sums up to a diverse and complex deployment that needs to be upgraded while maintaining our SLOs.

Preparing the journey

As the primary data store for GitHub, we hold ourselves to a high standard for availability. Due to the size of our fleet and the criticality of MySQL infrastructure, we had a few requirements for the upgrade process:

- We must be able to upgrade each MySQL database while adhering to our Service Level Objectives (SLOs) and Service Level Agreements (SLAs).
- We are unable to account for all failure modes in our testing and validation stages. So, in order to remain within SLO, we needed to be able to roll back to the prior version of MySQL 5.7 without a disruption of service.
- We have a very diverse workload across our MySQL fleet. To reduce risk, we needed to upgrade each database cluster atomically and schedule around other major changes. This meant the upgrade process would be a long one. Therefore, we knew from the start we needed to be able to sustain operating a mixed-version environment.

Preparation for the upgrade started in July 2022 and we had several milestones to reach even before upgrading a single production database.

Prepare infrastructure for upgrade

We needed to determine appropriate default values for MySQL 8.0 and perform some baseline performance benchmarking. Since we needed to operate two versions of MySQL, our tooling and automation needed to be able to handle mixed versions and be aware of new, different, or deprecated syntax between 5.7 and 8.0.

Ensure application compatibility

We added MySQL 8.0 to Continuous Integration (CI) for all applications using MySQL. We ran MySQL 5.7 and 8.0 side-by-side in CI to ensure that there wouldn’t be regressions during the prolonged upgrade process. We detected a variety of bugs and incompatibilities in CI, helping us remove any unsupported configurations or features and escape any new reserved keywords.

To help application developers transition towards MySQL 8.0, we also enabled an option to select a MySQL 8.0 prebuilt container in GitHub Codespaces for debugging and provided MySQL 8.0 development clusters for additional pre-prod testing.

More on availability

How we improved availability through iterative simplification

Solving and staying ahead of problems when scaling up a system of GitHub’s size is a delicate process. Here’s a look at some of the tools in GitHub’s toolbox, and how we’ve used them to solve problems.

Nick Hengeveld

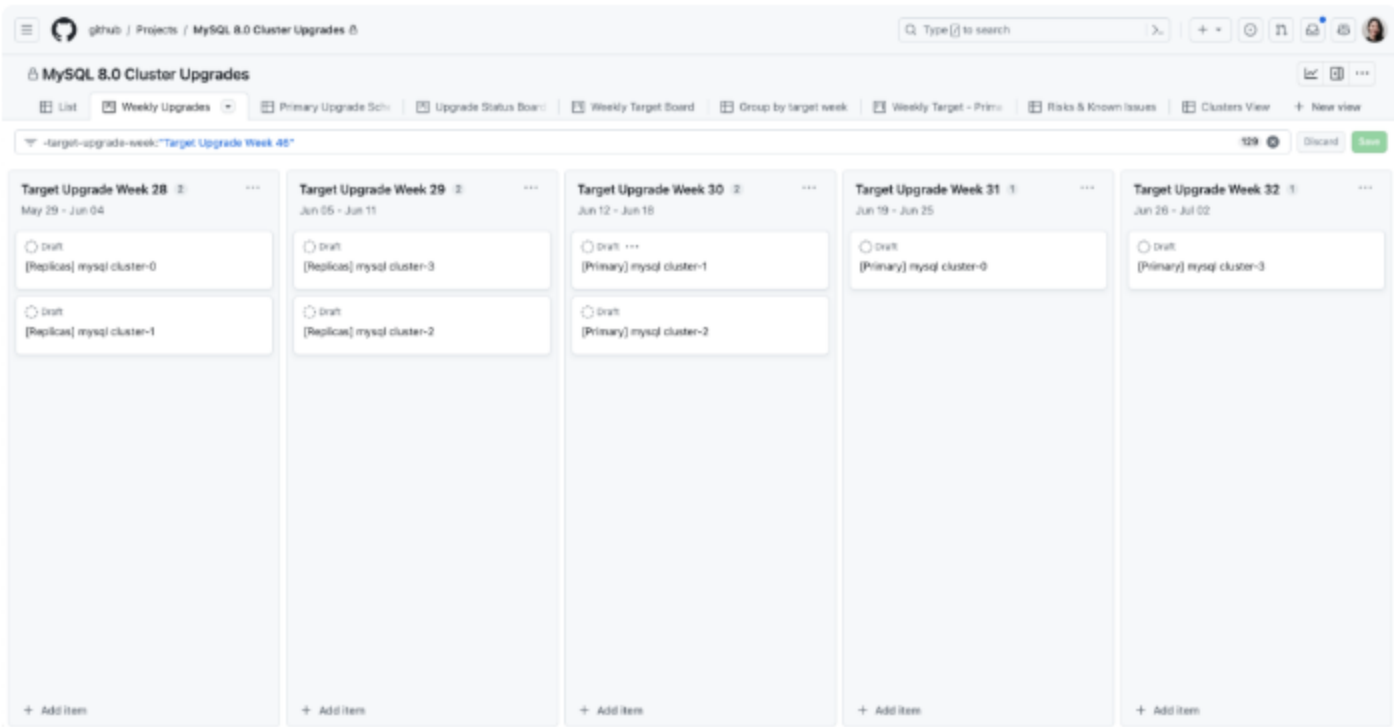
GitHub's Engineering Fundamentals program: How we deliver on availability, security, and accessibility

The Fundamentals program has helped us address tech debt, improve reliability, and enhance observability of our engineering systems.

Deepthi Rao Coppisetty

Communication and transparency

We used GitHub Projects to create a rolling calendar to communicate and track our upgrade schedule internally. We created issue templates that tracked the checklist for both application teams and the database team to coordinate an upgrade.



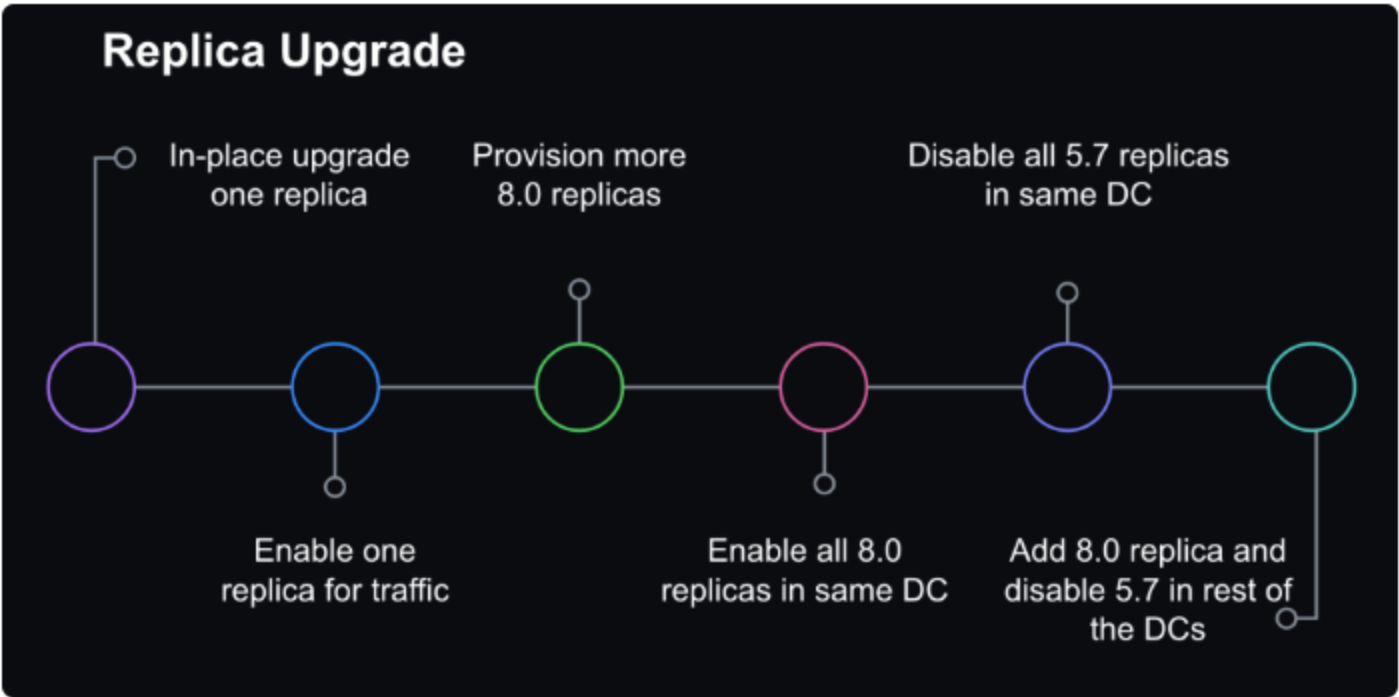
Project Board for tracking the MySQL 8.0 upgrade schedule

Upgrade plan

To meet our availability standards, we had a gradual upgrade strategy that allowed for checkpoints and rollbacks throughout the process.

Step 1: Rolling replica upgrades

We started with upgrading a single replica and monitoring while it was still offline to ensure basic functionality was stable. Then, we enabled production traffic and continued to monitor for query latency, system metrics, and application metrics. We gradually brought 8.0 replicas online until we upgraded an entire data center and then iterated through other data centers. We left enough 5.7 replicas online in order to rollback, but we disabled production traffic to start serving all read traffic through 8.0 servers.



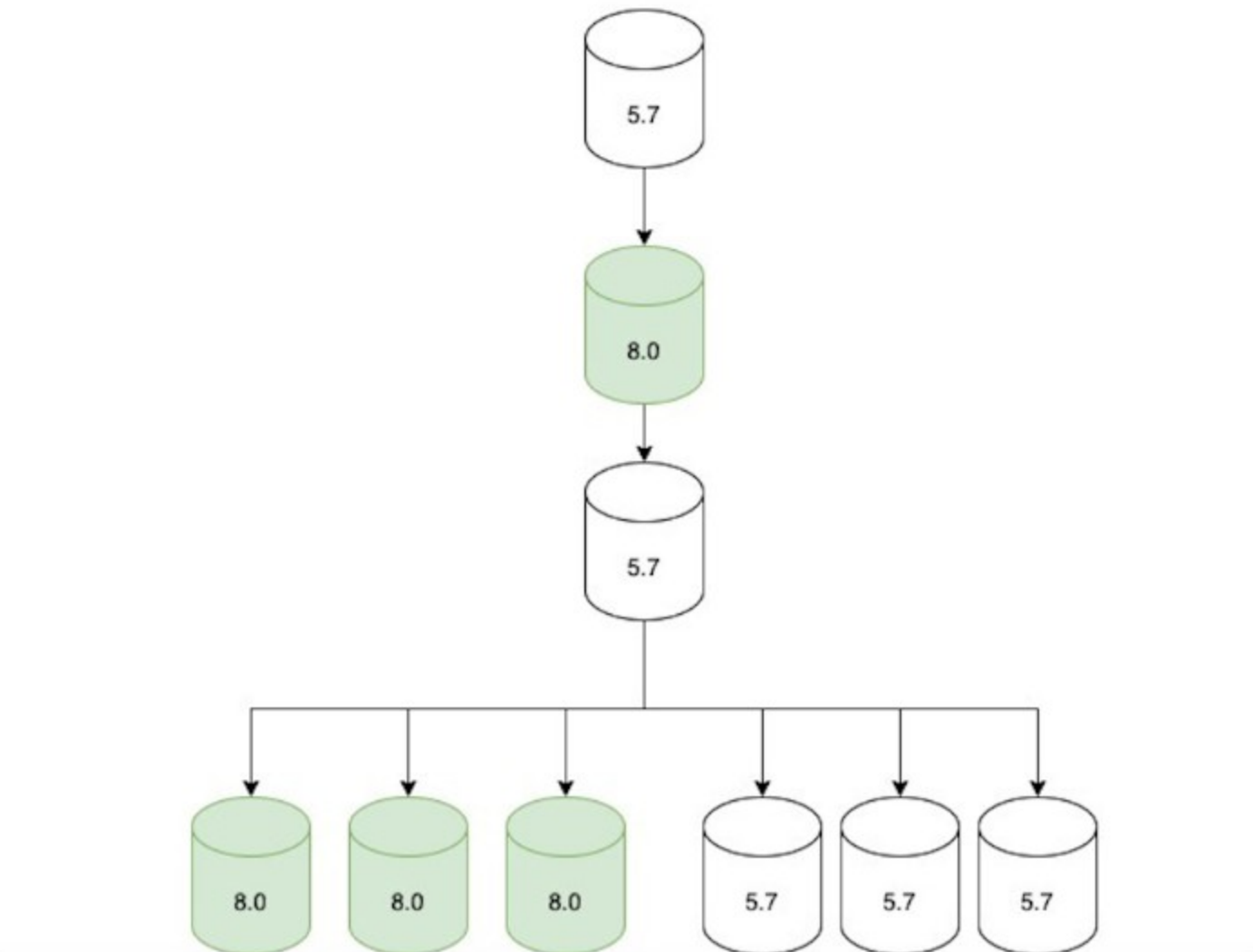
The replica upgrade strategy involved gradual rollouts in each data center (DC).

Step 2: Update replication topology

Once all the read-only traffic was being served via 8.0 replicas, we adjusted the replication topology as follows:

- An 8.0 primary candidate was configured to replicate directly under the current 5.7 primary.
- Two replication chains were created downstream of that 8.0 replica:

- A set of only 5.7 replicas (not serving traffic, but ready in case of rollback).
- A set of only 8.0 replicas (serving traffic).
- The topology was only in this state for a short period of time (hours at most) until we moved to the next step.

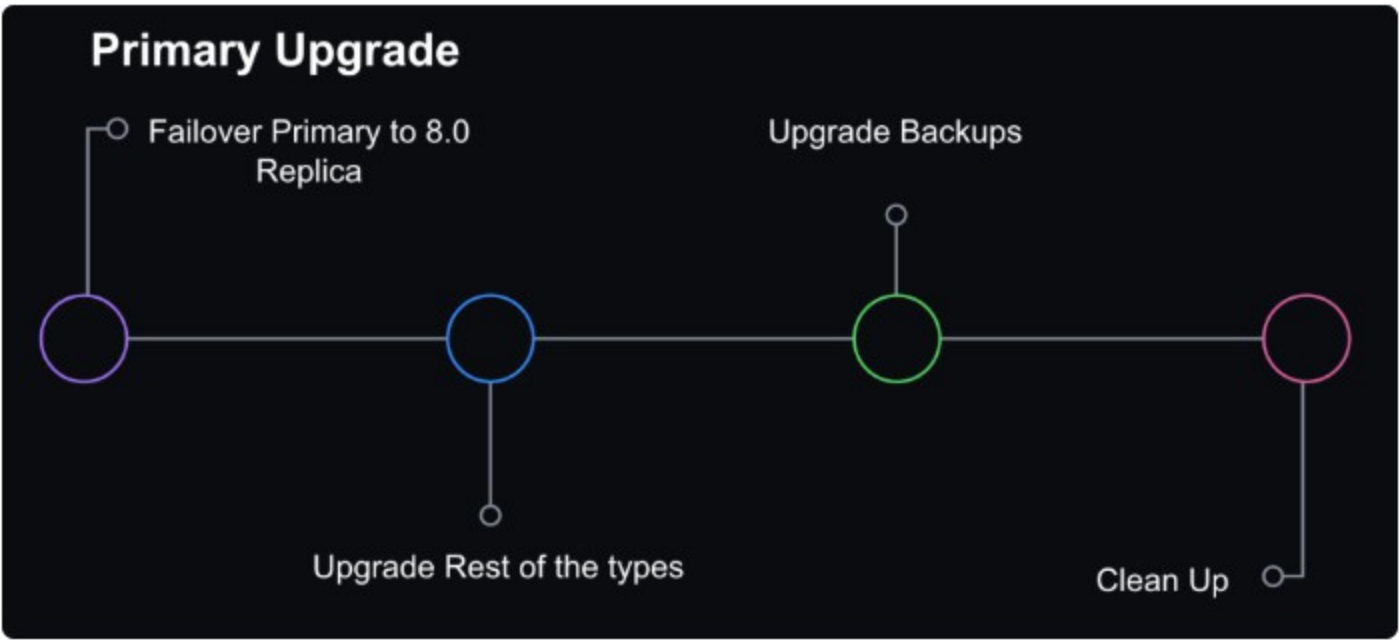


To facilitate the upgrade, the topology was updated to have two replication chains.

Step 3: Promote MySQL 8.0 host to primary

We opted not to do direct upgrades on the primary database host. Instead, we would promote a MySQL 8.0 replica to primary through a graceful failover performed with [Orchestrator](#). At that point, the replication topology consisted of an 8.0 primary with two replication chains attached to it: an offline set of 5.7 replicas in case of rollback and a serving set of 8.0 replicas.

Orchestrator was also configured to blacklist 5.7 hosts as potential failover candidates to prevent an accidental rollback in case of an unplanned failover.



Primary failover and additional steps to finalize MySQL 8.0 upgrade for a database

Step 4: Internal facing instance types upgraded

We also have ancillary servers for backups or non-production workloads. Those were subsequently upgraded for consistency.

Step 5: Cleanup

Once we confirmed that the cluster didn't need to rollback and was successfully upgraded to 8.0, we removed the 5.7 servers. Validation consisted of at least one complete 24 hour traffic cycle to ensure there were no issues during peak traffic.

Ability to Rollback

A core part of keeping our upgrade strategy safe was maintaining the ability to rollback to the prior version of MySQL 5.7. For read-replicas, we ensured enough 5.7 replicas remained online to serve production traffic load, and rollback was initiated by disabling the 8.0 replicas if they weren't performing well. For the primary, in order to roll back without data loss or service disruption, we needed to be able to maintain backwards data replication between 8.0 and 5.7.

MySQL supports replication from one release to the next higher release but does not explicitly support the reverse ([MySQL Replication compatibility](#)). When we tested promoting an 8.0 host to primary on our staging cluster, we saw replication break on all 5.7 replicas. There were a couple of problems we needed to overcome:

1. In MySQL 8.0, `utf8mb4` is the default character set and uses a more modern `utf8mb4_0900_ai_ci` collation as the default. The prior version of MySQL 5.7 supported the `utf8mb4_unicode_520_ci` collation but not the latest version of Unicode `utf8mb4_0900_ai_ci`.
2. MySQL 8.0 [introduces roles](#) for managing privileges but this feature did not exist in MySQL 5.7. When an 8.0 instance was promoted to be a primary in a cluster, we encountered problems. Our configuration management was expanding certain permission sets to include role statements and executing them, which broke downstream replication in 5.7 replicas. We solved this problem by temporarily adjusting defined permissions for affected users during the upgrade window.

To address the character collation incompatibility, we had to set the default character encoding to `utf8` and collation to `utf8_unicode_ci`.

For the GitHub.com monolith, our Rails configuration ensured that character collation was consistent and made it easier to standardize client configurations to the database. As a result, we had high confidence that we could maintain backward replication for our most critical applications.

Challenges

Throughout our testing, preparation and upgrades, we encountered some technical challenges.

What about Vitess?

We use Vitess for horizontally sharding relational data. For the most part, upgrading our Vitess clusters was not too different from upgrading the MySQL clusters. We were already running Vitess in CI, so we were able to validate query compatibility. In our upgrade strategy for sharded clusters, we upgraded one shard at a time. VTgate, the Vitess proxy layer, advertises the version of MySQL and some client behavior depends on this version information. For example, one application used a Java client that disabled the query cache for 5.7 servers—since the query cache was removed in 8.0, it generated blocking errors for them. So, once a single MySQL host was upgraded for a given keyspace, we had to make sure we also updated the VTgate setting to advertise 8.0.

Replication delay

We use read-replicas to scale our read availability. GitHub.com requires low replication delay in order to serve up-to-date data.

Earlier on in our testing, we encountered a replication bug in MySQL that was [patched on 8.0.28](#):

Replication: If a replica server with the system variable `replica_preserve_commit_order` = 1 set was used under intensive load for a long period, the instance could run out of commit order sequence tickets. Incorrect behavior after the maximum value was exceeded caused the applier to hang and the applier worker threads to wait indefinitely on the commit order queue. The commit order sequence ticket generator now wraps around correctly. Thanks to Zhai Weixiang for the contribution. (Bug #32891221, Bug #103636)

We happen to meet all the criteria for hitting this bug.

- We use `replica_preserve_commit_order` because we use GTID based replication.
- We have intensive load for long periods of time on many of our clusters and certainly for all of our most critical ones. Most of our clusters are very write-heavy.

Since this bug was already patched upstream, we just needed to ensure we are deploying a version of MySQL higher than 8.0.28.

We also observed that the heavy writes that drove replication delay were exacerbated in MySQL 8.0. This made it even more important that we avoid heavy bursts in writes. At GitHub, we use [freno](#) to throttle write workloads based on replication lag.

Queries would pass CI but fail on production

We knew we would inevitably see problems for the first time in production environments—hence our gradual rollout strategy with upgrading replicas. We encountered queries that passed CI but would fail on production when encountering real-world workloads. Most notably, we encountered a problem where queries with large `WHERE IN` clauses would crash MySQL. We had large `WHERE IN` queries containing over tens of thousands of values. In those cases, we needed to rewrite the queries prior to continuing the upgrade process. Query sampling helped to track and detect these problems. At GitHub, we use [Solarwinds DPM \(VividCortex\)](#), a SaaS database performance monitor, for query observability.

Learnings and takeaways

Between testing, performance tuning, and resolving identified issues, the overall upgrade process took over a year and involved engineers from multiple teams at GitHub. We upgraded our entire fleet to MySQL 8.0 – including staging clusters, production clusters in support of GitHub.com, and instances in support of internal tools. This upgrade highlighted the importance of our observability platform, testing plan, and rollback capabilities. The testing and gradual rollout strategy allowed us to identify problems early and reduce the likelihood for encountering new failure modes for the primary upgrade.

While there was a gradual rollout strategy, we still needed the ability to rollback at every step and we needed the observability to identify signals to indicate when a rollback was needed. The most challenging aspect of enabling rollbacks was holding onto the backward replication from the new 8.0 primary to 5.7 replicas. We learned that consistency in the [Trilogy client library](#) gave us more predictability in connection behavior and allowed us to have confidence that connections from the main Rails

monolith would not break backward replication.

However, for some of our MySQL clusters with connections from multiple different clients in different frameworks/languages, we saw backwards replication break in a matter of hours which shortened the window of opportunity for rollback. Luckily, those cases were few and we didn't have an instance where the replication broke before we needed to rollback. But for us this was a lesson that there are benefits to having known and well-understood client-side connection configurations. It emphasized the value of developing guidelines and frameworks to ensure consistency in such configurations.

Prior efforts to [partition our data](#) paid off—it allowed us to have more targeted upgrades for the different data domains. This was important as one failing query would block the upgrade for an entire cluster and having different workloads partitioned allowed us to upgrade piecemeal and reduce the blast radius of unknown risks encountered during the process. The tradeoff here is that this also means that our MySQL fleet has grown.

The last time GitHub upgraded MySQL versions, we had five database clusters and now we have 50+ clusters. In order to successfully upgrade, we had to invest in observability, tooling, and processes for managing the fleet.

Conclusion

A MySQL upgrade is just one type of routine maintenance that we have to perform – it's critical for us to have an upgrade path for any software we run on our fleet. As part of the upgrade project, we developed new processes and operational capabilities to successfully complete the MySQL version upgrade. Yet, we still had too many steps in the upgrade process that required manual intervention and we want to reduce the effort and time it takes to complete future MySQL upgrades.

We anticipate that our fleet will continue to grow as GitHub.com grows and we have goals to partition our data further which will increase our number of MySQL clusters over time. Building in automation for operational tasks and self-healing capabilities can help us scale MySQL operations in the future. We believe that investing in reliable fleet management and automation will allow us to scale github and keep up with required maintenance, providing a more predictable and resilient system.

The lessons from this project provided the foundations for our MySQL automation and will pave the way for future upgrades to be done more efficiently, but still with the same level of care and safety.

—

Tags :

availability

databases

Infrastructure

MySQL

Written by



Jiaqi Liu
[@itsJiaqi](#)

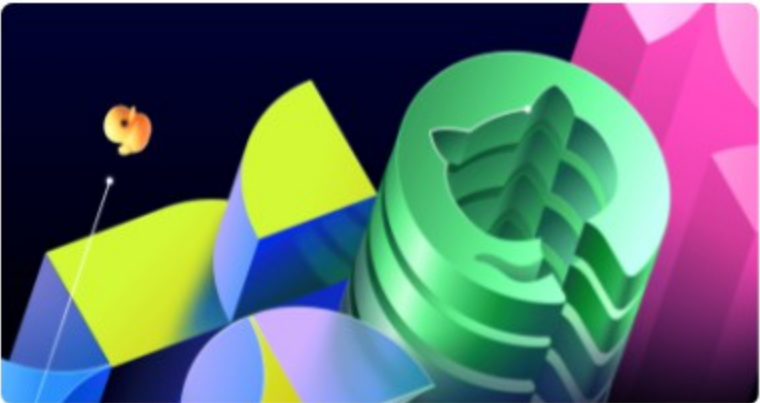


Daniel Rogart
[@drogart](#)



Xin Wu
[@xinwu5](#)

Related posts



Engineering

How GitHub engineers tackle platform problems

Our best practices for quickly identifying, resolving, and preventing issues at scale.

Fabian Aguilar Gomez



Application development

GitHub Issues search now supports nested queries and boolean operators: Here’s how we (re)built it

Plus, considerations in updating one of GitHub’s oldest and most heavily used features.

Deborah Digges




Engineering

Design system annotations, part 2: Advanced methods of annotating components

How to build custom annotations for your design system components or use Figma’s Code Connect to help capture important accessibility details before development.

Jan Maarten


Explore more from GitHub



Docs

Everything you need to master GitHub, all in one place.


[Go to Docs](#)



The ReadME Project

Stories and voices from the developer community.


[Learn more](#)



GitHub Copilot

Don’t fly solo. Try 30 days for free.

[Learn more](#)



Enterprise content

Executive insights, curated just for you

[Get started](#)

We do newsletters, too

Discover tips, technical guides, and best practices in our biweekly newsletter just for devs.

Your email address

*

Your email address

[Subscribe](#)

☐ Yes please, I'd like GitHub and affiliates to use my information for personalized communications, targeted advertising and campaign effectiveness. See the [GitHub Privacy Statement](#) for more details.

