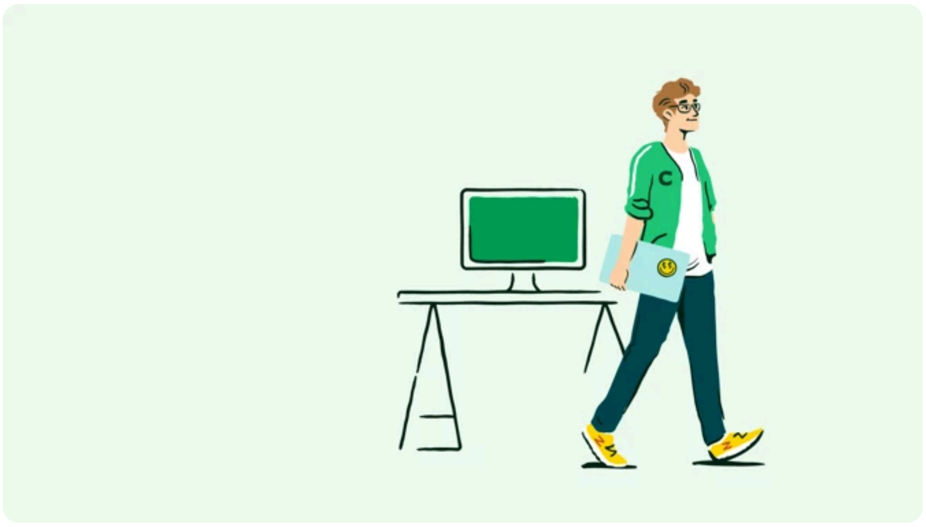
04/17/2025

How we upgraded our core database with just 5 minutes of downtime



By Nicolas Richard, Senior Software Engineer

When your database powers critical services that millions of members rely on daily, upgrading it is like performing heart surgery while the patient is running a marathon. Recently, we completed one of our most significant technical achievements: upgrading CoreDB, our largest and most critical MySQL cluster, from MySQL 5 to MySQL 8—

before Chime was even called Chime. It powers core services that are fundamental to our operations, but we had reached a critical juncture: MySQL 5 was approaching end-of-life support from AWS. While we could have extended support, it would have incurred additional monthly costs.

Over the years CoreDB has grown with Chime and it's now a cluster of 5 DB instances, 2 are AWS's **db.r5.8xlarge** and 3 are **db.r5.24xlarge**, which are the largest memory optimized instances offered by AWS! The writer instance routinely serves more than 10k connections and the entire cluster 30k. The data stored is over 40 Tb.

The timing for this upgrade, while forced by AWS's timeline, was also strategic: we knew that upgrading alongside other companies during AWS's transition period would give us access to maximum support and shared learning from the broader community facing the same challenge.

While our existing system was stable, the MySQL 5 version created several pain points:

First, our engineering team faced database maintenance challenges, the most notable of which was our inability to delete tables—an operation that could lock the entire database for minutes at a time. For a database of this size and importance, even a few minutes could impact our members' experience. As a result, we were living with a cluttered database that was difficult to maintain and optimize.

Second, the cost implications were substantial. Beyond the looming monthly fees for extended support, our accumulated data was driving up our operating costs. By upgrading and being able to clean up our database, we projected a significant cost savings..

features.

Why we couldn't just "update in place"

As our CoreDB database is central to nearly every service in our architecture, any issues during the upgrade would have far-reaching consequences across our entire platform.
One approach to database upgrades is to perform an in-place upgrade—essentially downloading new software and rebooting. However, this approach posed significant risks:

- The downtime would be longer and harder to predict
- Once started, we couldn't easily roll back if something went wrong

The blue/green strategy: Our path forward

After researching similar upgrades at companies like GitHub and Uber, we determined that a blue/green strategy would give us what we needed most: better control over the process and the ability to roll back if anything went wrong. Think of it like getting a new laptop with the latest software while keeping the old one running—we'd only switch over once everything was ready and tested. The Blue/Green strategy, with its BluePrime standby cluster, gave us a reliable fallback to MySQL 5 if needed and offered a more predictable and shorter downtime compared to an in-place upgrade.

While migrating read queries to the Green DB was straightforward, the write cutover required careful handling. We couldn't allow writes to occur true source of truth. This meant some downtime was unavoidable during the write cutover from the old to the new database.

With one of our core values being “Member Obsessed”, we focused on minimizing the disruption to our members’ experience. We designed a tailor-made Blue/Green switchover runbook and leveraged our 'maintenance mode' feature to provide clear communication about temporary service unavailability rather than subjecting members to unpredictable app behavior.

The entire upgrade process, from initial planning to completion, took six months of careful preparation and execution, with much of that time spent testing and refining our approach in non-production environments.

Phase 1: Testing ground

To prepare for the MySQL 8 upgrade, we:

1. Updated test suites and CI systems to support both MySQL 5 and 8
2. Configured cloud development environments to use MySQL 8
3. Implemented automatic comparison of new migration files between versions 5 and 8

This comprehensive approach ensured thorough testing and validation before any changes were made to production, minimizing potential disruptions.

Then we attempted an in-place upgrade of a clone of the production DB. While we knew we weren’t going to use the in-place upgrade for the real production DB, this attempt allowed us to benefit from the AWS built-in pre-upgrade checks, which revealed some issues. AWS runs the script before proceeding with the upgrade and proceed.

For example, we had a number of tables affected by this issue:

```
"id": "zeroDatesCheck", "title": "Zero Date, Datetime,
and Timestamp values", "description": "Warning: By
default zero date/datetime/timestamp values are no
longer allowed in MySQL, as of 5.7.8 NO_ZERO_IN_DATE and
NO_ZERO_DATE are included in SQL_MODE by default. These
modes should be used with strict mode as they will be
merged with strict mode in a future release. If you do
not include these modes in your SQL_MODE setting, you
are able to insert date/datetime/timestamp values that
contain zeros. It is strongly advised to replace zero
values with valid ones, as they may not work correctly
in the future.",
```

The reason for this was that some tables had no default for their **"updated_at"** column. We updated the default on all those tables and fixed the records where needed.

Each issue we discovered in testing was one less surprise waiting for us in production.

Phase 2: Refactoring

We had to refactor the way CoreDB and associated AWS resources are managed in our Terraform codebase for 2 reasons:

1. Between prod and non-prod, the setup was different, which didn't allow for testing or development of a strategy that would work for both.
2. We needed to be able to create Green and BluePrime clusters conveniently.

connect to the DB.

After this work, bringing up a clone of DB for testing was a simple PR to our Terraform repo. This paid off massively for us when we needed to create and migrate clusters repeatedly to refine our runbooks.

Phase 3: The Replication Chain

Having decided to go with the Blue/Green strategy meant that we'd need to set up multiple DB clusters and a chain of replication between them. Blue DB replicates to Green DB, Green DB replicates to BluePrime DB—our fallback DB (in case an upgrade to Green didn't work we would transfer all traffic to BluePrime DB).

Here is an overview of the steps we took to create the new Green DB cluster.

- We created a MySQL8 parameter group. We had to ensure the binary log was ON as this would be necessary for replication from Green to BluePrime. We also reviewed any custom settings we had on our Blue cluster's parameter groups and brought them over to the new MySQL8 parameter group. (Example the parameter "tx_isolation" was renamed to "transaction_isolation".).
- We created a new cluster from a snapshot of the Blue cluster and all other needed resources (such as RDS proxy).
- We configured monitoring.
- We upgraded the Green cluster to MySQL8
  ```
  aws rds modify-db-cluster --db-
  cluster-identifier main-green --
  engine-version
  8.0.mysql_aurora.3.04.3 --db-cluster-
  ```

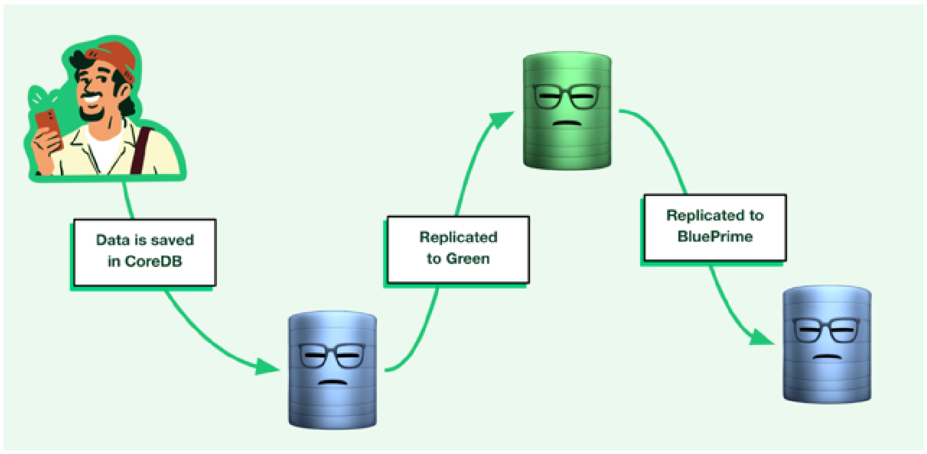
```
aurora-db-mysql8-parameter-group --
allow-major-version-upgrade --apply-
immediately
```

- We set up replication from Blue to Green. Binlog position details were in the AWS RDS web UI—we were able to find them by checking the events of the writer instance of our DB cluster.

```
mysql> CALL
mysql.rds_set_external_source ('main-
blue.cluster-crmksbav3ozo.us-east-
1.rds.amazonaws.com', 3306,-
>repl_user', 'qwerty-1234567890',
'mysql-bin-changelog.000005',
39478089, 0);mysql> CALL
mysql.rds_start_replication;
```

For the BluePrime cluster the setup steps were the same except we didn't upgrade it and it replicates from Green.

While at first we considered letting Blue DB and BluePrime DB share the same parameter group we decided against it. The reason being that during the upgrade we need to control the DB's ability to receive writes (or not) with the "read_only" parameter, which is dynamic (i.e. no reboot of the DB needed) and we didn't want to risk having a DB accepting writes when it shouldn't as this could lead to data integrity issues.

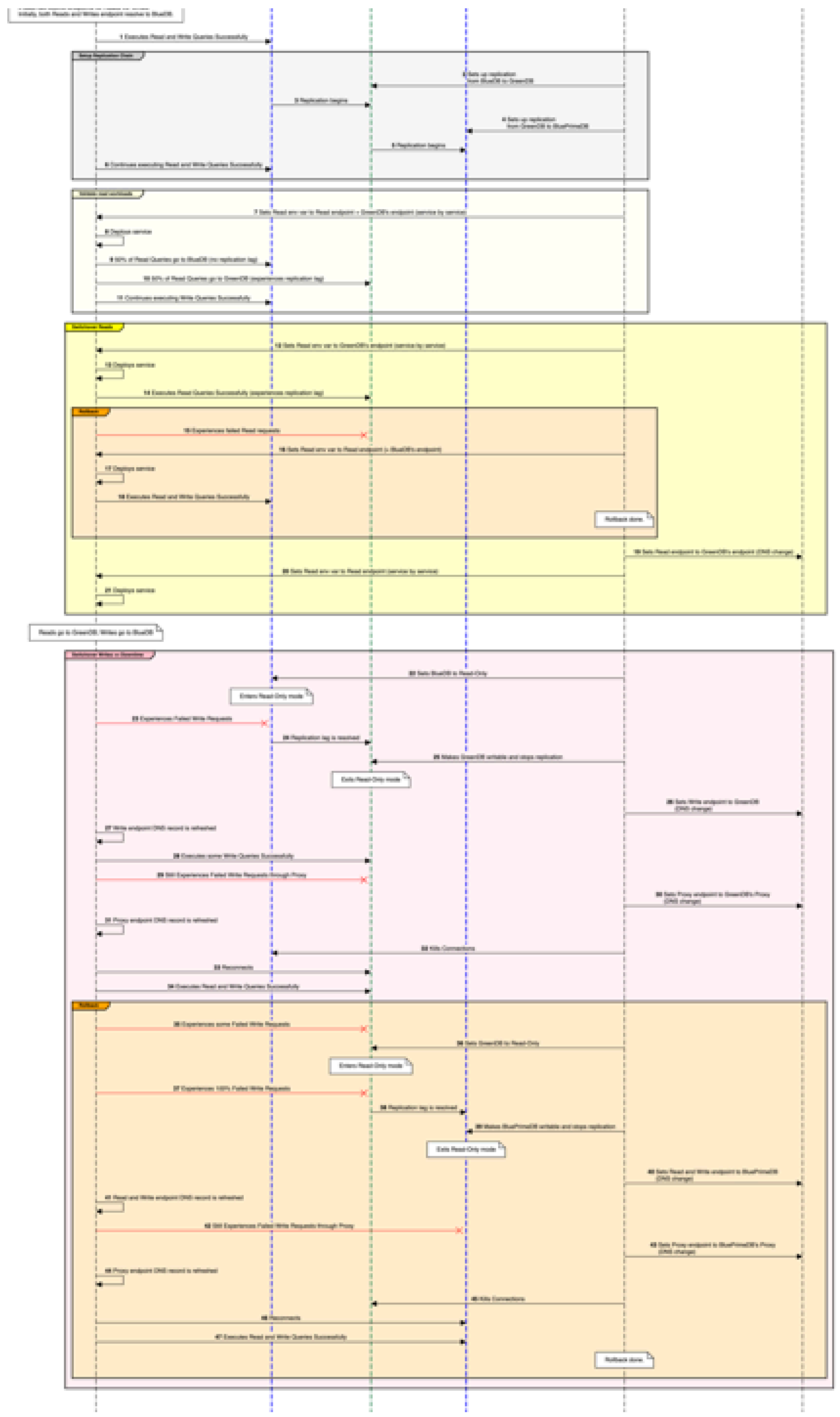


eye on the GreenDB's behavior in case any errors appeared (for example, for use of a feature or keyword deprecated from MySQL5 to 8).

Phase 4: Testing Blue/Green upgrade in non-prod

We started drafting our runbook, detailing the steps we'd need to take and figuring out the best order in which they needed to happen. We developed a very detailed sequence diagram containing all steps necessary to migrate traffic between the DBs. This diagram became our north star in developing the runbooks.

Working in our test environment, we iterated, learning as we went. It took five attempts until we were able to run an upgrade and rollback in nonprod without encountering any unexpected issues in the process or our scripts. We got to the point where we were confident the steps were in the most optimal order and that we could run through them in just a few minutes.



Phase 5: Setting up the prod clusters and replication

When we started the work in prod we had to take 2 preliminary steps:

1. **We turned on binary logging**: This was a crucial feature necessary for our strategy to work, since

for a long time until we had to turn it off to improve performance on that DB. Fortunately, since that decision had been made, some work to reduce the load on the DB had also happened. By turning this on, there was a chance we might impact performance, so we approached it with caution (spoiler alert: it went fine).

2. **We turned off the Query cache feature**: We identified that we could further de-risk the upgrade by turning off the Query cache feature. And since the feature was being deprecated in MySQL 8, we knew we couldn't rely on it and it might also impact performance in the future. Because we wanted to have fewer things to deal with during the actual upgrade, we carefully got rid of it ahead of time in order to find out early on.

Phase 6: Tackling the replication lag challenge

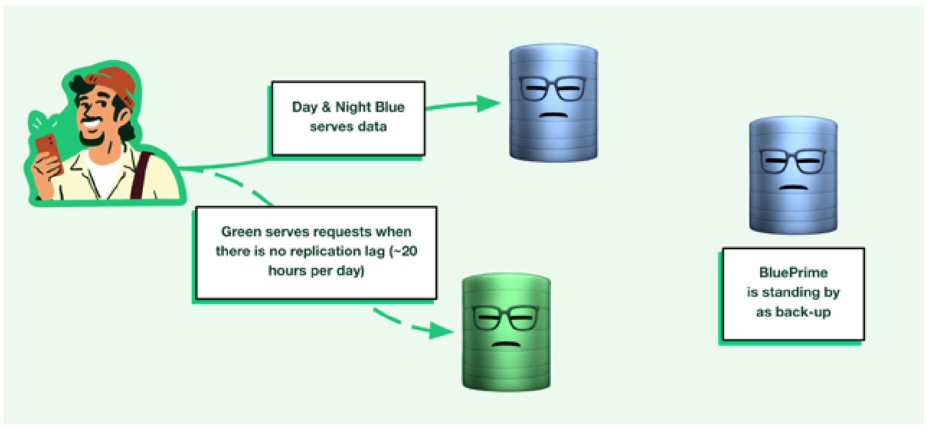
We proceeded to create the Green and BluePrime clusters in prod and began testing. While things looked fine at first, we quickly noticed a major issue to our strategy. During peak traffic, our Green cluster was falling hours behind in processing updates from Blue.

This posed a critical risk, we had been hoping to send read queries to green as a way to confirm that MySQL 8 could handle the load. This was a cornerstone of our de-risking strategy but we couldn't have members seeing stale data.

The solution came through our Chime Atlas library, which manages database connections for all our services. The reality is that BlueDB was a cluster of 5 instances and only 1 of them received the writes—the others were read-only clones that served read traffic to group that can receive read queries.

The way this works: we are running pt-heartbeat to constantly update the BlueDB. It makes it easy to check if a replica is in-sync with its source by comparing the value of the heartbeat and the current time. Atlas does this constantly when routing queries to the DB instances.

So we updated Atlas' configuration so GreenDB's readers were treated just like BlueDB clones—that way, when the lag would pick up, Atlas would realize the Green instance as not suitable and would stop sending queries there until recovery. 50% of read traffic was now going to GreenDB at night, during which GreenDB was receiving a diversity of queries from all services.


Day & Night Blue serves data
Green serves requests when there is no replication lag (~20 hours per day)
BluePrime is standing by as back-up

As queries started hitting the Green DB readers we ran into an issue, we noticed queries were failing and Atlas was not considering Green DB instances to be healthy even though there was no replication lag. In our application logs we found a barrage of messages like **"The table '/rdsdbdata/tmp/#sql5041_8e0484_5d' is full"**. It turned out that the default setting for **temptable_max_mmap** was really low, so we increased it to 50% of the available local storage (found per instance type here).

performance data without risking member experience and gained the confidence we needed to trust MySQL 8.

The big night: 5 minutes of magic

After months of preparation, the actual upgrade took place on November 12th. Here's how we did it, following the steps in the sequence diagram:

At 9pm PDT we switched over 100% of reads to the Green cluster, then we monitored.

At 11pm PDT we switched over of the writes:

1. We lowered DNS TTL and enabled maintenance mode
2. We made Blue DB read-only
3. We waited for replication to catch up (took under a second!)
4. We made Green DB writable
5. We updated DNS endpoints to point to Green DB
6. We forced backend services to re-connect by killing connections on the Blue DB

All in all, the total downtime our members experienced was a mere five minutes—something we're really proud of.

In case things didn't go according to plan, we had a rollback strategy. We would repeat the switch over process, redirecting read and write operations from GreenDB to BluePrimeDB.

While it was unlikely that GreenDB would be unable to perform (this was all happening at night when traffic was low), there was a small chance that the next morning when query volume increased, the team might find out GreenDB wasn't able to handle all the new

After all, everything turned out fine and there was no additional impact to our members beyond the five minutes of downtime.

Why this matters—to our team and our members

This upgrade wasn't just about keeping up with technology—it was about maintaining the reliability and security our members expect while improving our engineering team's ability to maintain and optimize our systems. By completing this upgrade:

- We avoided monthly extended support costs
- We improved our ability to maintain the database
- We demonstrated we could handle major infrastructure changes with minimal member impact
- We set up our systems for better performance and maintainability

The reality we faced was that any incident with our core database could have far-reaching downstream consequences—since nearly every service Chime provides relies on a service that connects to this DB. Instead, because of our rigorous planning, backup plans, and testing, the upgrade was achieved with minimal downtime during which we put our mobile and web apps into maintenance mode for just five minutes.

The success of this project wasn't just in the technical execution—it was in the careful planning, rigorous testing, and building of safety nets that ensured we could handle anything that came our way. It's a testament to how modern engineering teams can tackle complex challenges while maintaining the high standards of reliability that financial services—and our members—require.

A huge thank you to Tim Chepeleff for co-driving and to Eric Holmes, Ethan Erchinger, and David Lyons, for the help figuring it all out!

Thank you to Sorin Neacsu, Rohan Mathure, and Patrick Isaacs for their help and encouragement and to everyone else involved in making this happen.

Share

Tags

Engineering

chime® Careers Home

Chime Financial, Inc.

In the News

Careers

Trust & Safety

Resources

Policies

Chime U.S. Privacy Notice

Help Center

Second Chance Banking

Become an Affiliate

Supporting Those With Disabilities

Contact Us

1-844-244-6363

Download on the App Store

GET IT ON Google Play

Privacy Policy / Sitemap Do Not Sell or Share My Personal Information or Stride Bank pursuant to a license from Visa U.S.A. Inc. and may be used everywhere Visa debit cards are accepted. The Chime Visa® Credit Builder Card is issued by Stride Bank pursuant to a license from Visa U.S.A. Inc. and may be used everywhere Visa credit cards are accepted. Please see back of your Card for its issuing bank.

By clicking on some of the links above, you will leave the Chime Careers website and be directed to Chime or a third-party website. The privacy practices of those third parties may differ from those of Chime. We recommend you review the privacy statements of those third party websites, as Chime is not responsible for those third parties' privacy or security practices.

Chime is a financial technology company, not a bank. Banking services provided by The Bancorp Bank N.A. or Stride Bank N.A.; Members FDIC.