Zero Downtime PostgreSQL Upgrades

GitLab

GitLab

Alexander Sosna

Senior Database Reliability Engineer

Agenda

This talk will discus:

- How we execute PostgreSQL major upgrades at GitLab, with **zero* downtime**.

By answering these questions:

- PostgreSQL Upgrades - How do they work and why are they hard?
- How did GitLab.com perform these upgrades in the past?
- Why did we need to improve?
- How did we improve to minimize impact to our users?

README

- Slides showing the white triangle in the top left corner are not shown during presentations
- They are added to provide more context when reading the slides

What is zero down time?

What is zero down time?

- All services need to stay available to end users!

What is zero down time?

- All services need to stay available to end users!
- What is “available” here ?
  - That’s up for debate :)
  - Acceptable response times need to be defined
  - Service Level Objectives (SLO)

What is GitLab using for performance measuring?

$$\text{Apdex}_t = \frac{\text{SatisfiedCount} + (0.5 \cdot \text{ToleratingCount}) + (0 \cdot \text{FrustratedCount})}{\text{TotalSamples}}$$

- Apdex (Application Performance Index)
  - Open standard developed by an alliance of companies for measuring performance of software applications in computing. [...] It is based on counts of "satisfied", "tolerating", and "frustrated" users, [...]
  - Requires tuned thresholds to classify samples
  - Details: wikipedia.org/wiki/Apdex

Why are PostgreSQL Major Upgrades hard?

- Major releases (can) change the layout of system tables
  - Internal data storage format rarely changes drastically
  - Mostly added or changed metadata fields
- Data files can not be used by newer versions
- Rewriting of system tables and metadata is necessary
- Depending on data size and complexity this can take significant time

Upgrade Methods - pg_dumpall

1. maintenance mode (DB becomes RO)

2. physical to logical (binary to SQL*)

3. logical to physical (SQL* to binary)

4. index (re)creation statistic collection

Upgrade Methods - pg_dumpall

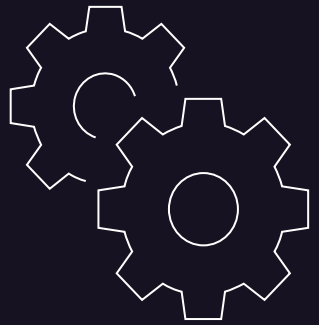
- Data is extracted and brought to a logical representation
  - SQL, or optimized internal format
- Logical data is then imported in the new cluster
- Both operations are resource and time consuming
  - Can be performed in parallel to disk
    OR
  - Piped from old to new cluster
- All data gets validated
- All indexes are freshly created
- No bloat in the new cluster

Upgrade Methods - pg_dumpall

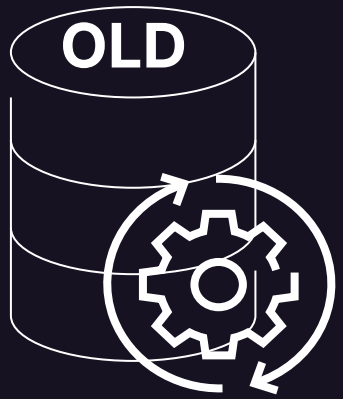
Best use case

- **Safest method** available, but requires **longest downtime**
  - Hard to provide universal numbers: *~20 TiB DB may take ~1 day*
- If this fulfills your needs, it's the safest option! Don't look any further!

Upgrade Methods - pg_upgrade

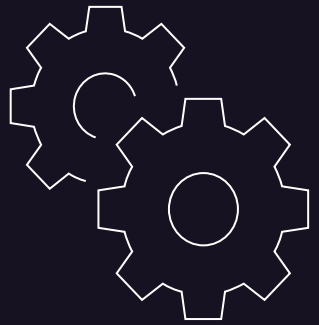


1.
Maintenance mode
(offline / RO with standby)

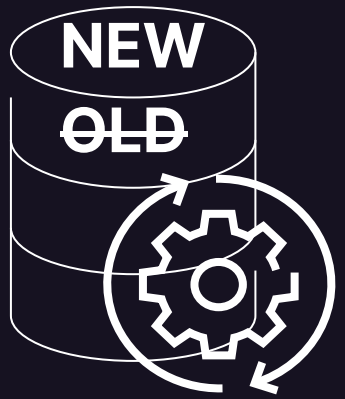


2.
In-place upgrading
binary data

Upgrade Methods - pg_upgrade



1.
Maintenance mode
(offline / RO with standby)



2.
In-place upgrading
binary data

Upgrade Methods - pg_upgrade

- The physical data structures needing adjustment are rewritten either:
  - in a data copy
  - in-place utilizing hard links
- In-place rewrite is quite fast, depending on the data size
- Rewrite requires downtime
- No rollback possible, e.g. should the new version not perform as required
- Additional post upgrade steps might be necessary or advised
  - Eg. recreating indexes to utilize new b-tree optimizations

Upgrade Methods - pg_upgrade

Best use case

- Reasonable fast
- Reasonable safe
- Quite simple
- If validation or benchmarking steps are required, this expands the downtime!
- If this fulfills your needs, it’s a safe and simple option! Don’t look any further!

How did we perform Upgrades in the past?

How did we perform Upgrades in the past?

pg_upgrade, with significant downtime

1. Create second cluster from backup
2. Sync new with main cluster (streaming replication)
3. **Put GitLab.com into maintenance**
4. Used *pg_upgrade* to upgrade primary
5. Re-create all standbys from primary
6. Run full QA tests and benchmark on new cluster (multiple hours)
7. Switch application to use new cluster
8. Bring **GitLab.com** back online

Why did we need to improve?

- Impact was not acceptable
    - Inconvenient for our customers
    - Responsible for most of our downtime
- PostgreSQL upgrades were avoided to minimize the impact
    - We were running PG12 at the beginning of 2023

How to improve, to minimize impact for users?

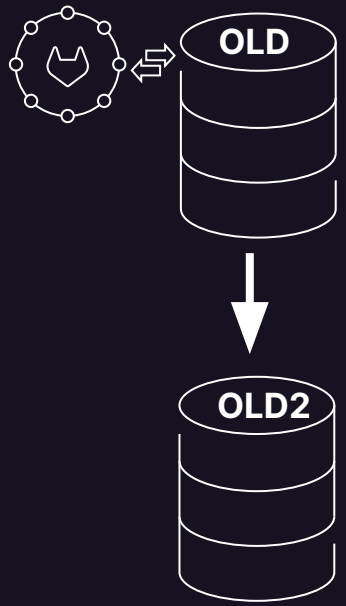
Optimizing the old process

- Most time is “lost” due to benchmarking the new cluster
- Removing tests would bring downtime from hours to ~minutes
- Deemed too dangerous, due to:
  - Possibility of long lasting impact
  - No rollback without data loss, after upgrade

OR

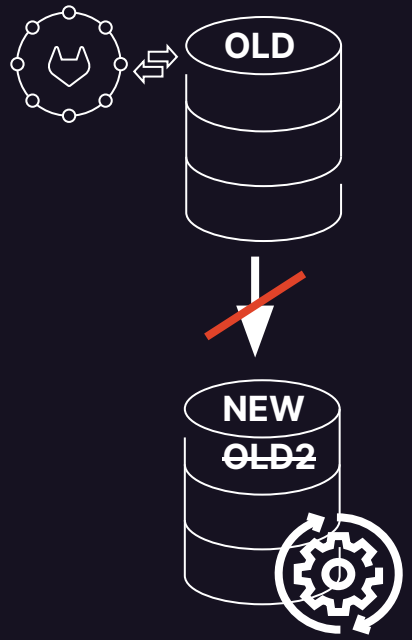
Utilization of logical replication to upgrade asynchronous during normal operation

logical replication + pg_upgrade



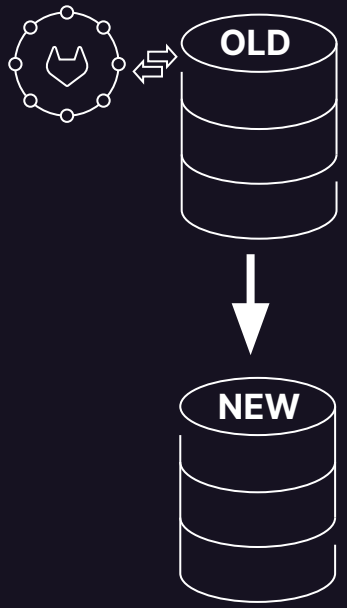
1. Create and sync new instance

logical replication + pg_upgrade



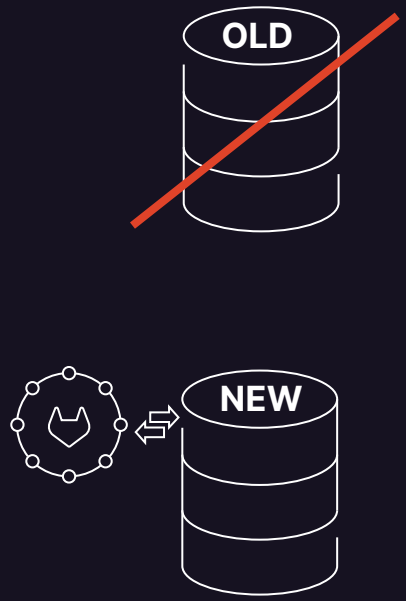
2. Upgrade new cluster
(no sync during upgrade)

logical replication + pg_upgrade



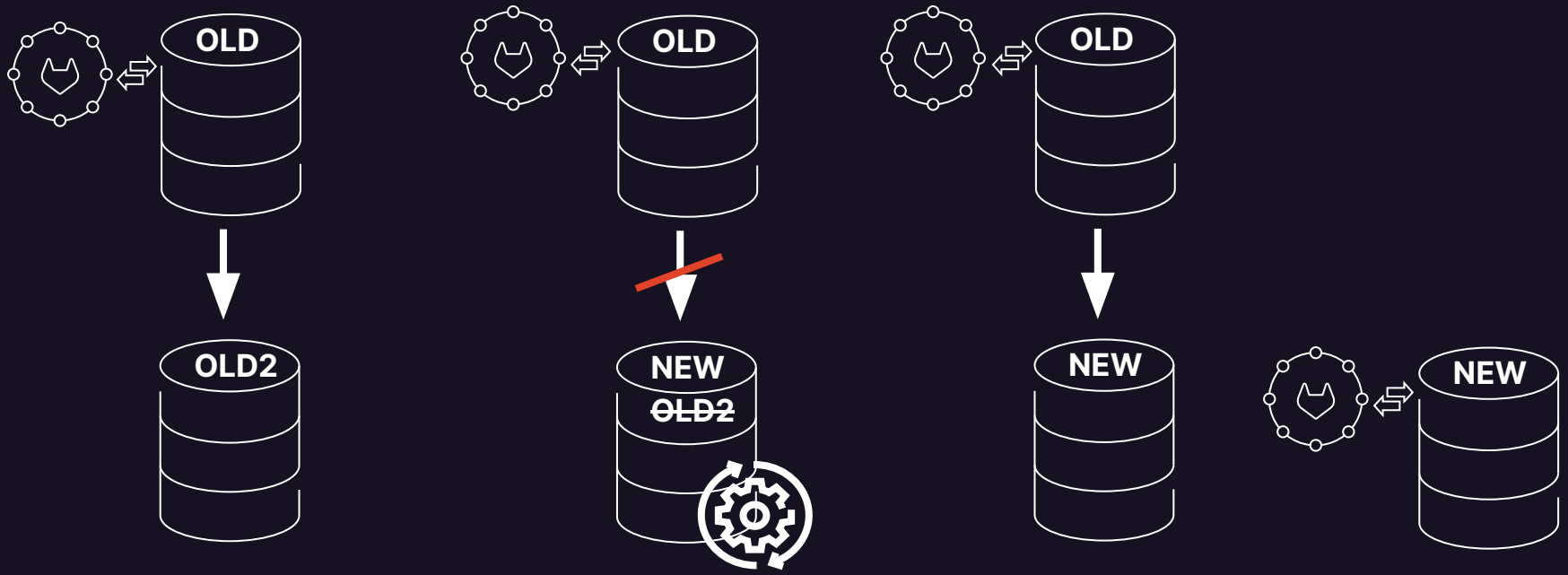
3. Resync with LR

logical replication + pg_upgrade



4.Switchover Application

logical replication + pg_upgrade



1. Sync new instance with

2. Upgrade new cluster, no sync during upgrade

3. Resync with LR

4. Switchover App.

logical replication

- Replicate changes across different:
  - PostgreSQL versions
  - Operation Systems (libc version)
  - CPU Architectures
- Enables us to execute major upgrades asynchronously

logical replication

What is the catch?

1. Database schema and DDL commands are not replicated!
2. Sequences are not replicated, but are needed for auto increment values
3. Each table needs a *REPLICA IDENTITY*, to distribute changes
    - Primary key
    - Other unique key
    - *FULL*, last resort, all changes need to be recorded
4. More complex
    - Prone to human errors
    - Automation and testing is highly advised

logical replication - DDL is not replicated

- Schema changes would break logical replication!
  - No DDL allowed: CREATE, ALTER, DROP
- The initial schema is also not replicated

logical replication - DDL is not replicated

Our solution

- We disabled all background migration and maintenance jobs, which did the trick! You need to check **YOUR** applications DDL usage!
  - Disabling such jobs for reducing load is advised in any case
- Start from the latest backup, not from an empty database
  - No manual schema export required
  - No unnecessary logical transformation of historical data

logical replication - Sequences are not replicated

- Sequences are vital to PostgreSQL
  - Generates unique sequential numbers wherever they are needed
  - Used for SERIAL (AUTO INCREMENT)

logical replication - Sequences are not replicated

Our solution

- Measure the daily growth of all sequences
- Defined a large “sequences buffer value”, eg. *1 million*
- Increase the sequences on the NEW cluster by this value
- Before switchover check that the sequences on OLD, have not grown more than expected (optional)
- Simple solution, only uses up a fraction of the keyspace of 64 bit integer

logical replication - REPLICA IDENTITY

- Each table needs a *REPLICA IDENTITY*, to distribute changes
  - Primary key
  - Other unique key
  - *FULL*, last resort, all changes need to be recorded

logical replication - REPLICA IDENTITY

Our solution

- Nothing to do, we already had primary keys :D

logical replication - Complexity

- More complex
  - Prone to human errors
  - Automation and testing is highly advised

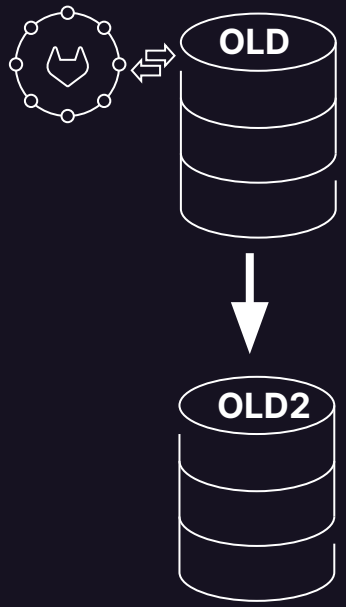
logical replication - Complexity

Our solution

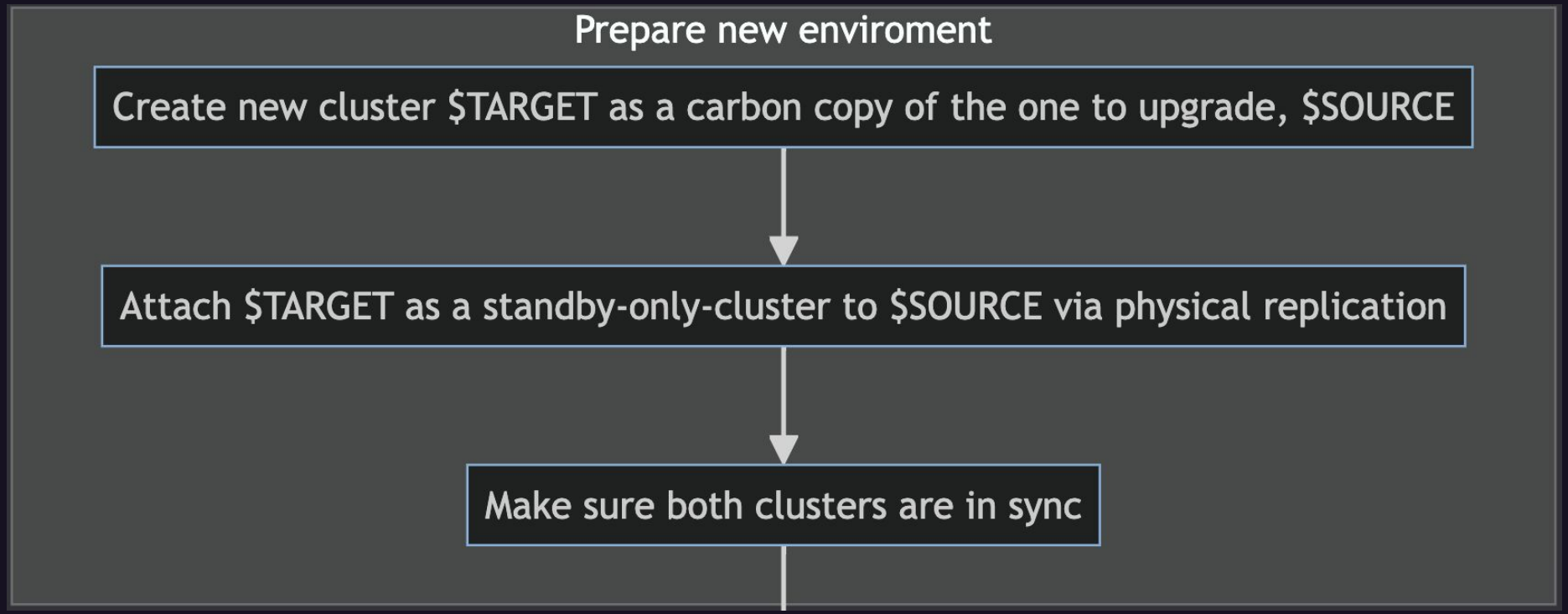
- Complete automation
  - Orchestration via Ansible
  - Process as CR issue which could be executed repetitively
- Excessive testing - “*When it hurts, do it more often*”
  - Intense QA tests before switchover, rollback if not perfect
  - Dry runs in production
    - All steps until switchover
    - Measure timings, performance metrics
    - Iterate on QA test suite and process

Improved process with logical replication

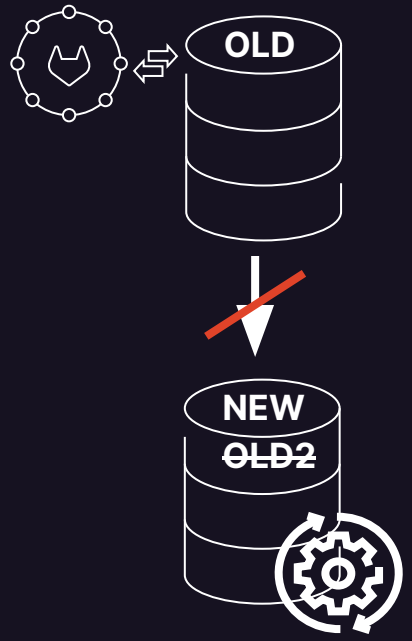
Upgrade - 1. Prepare new Environment



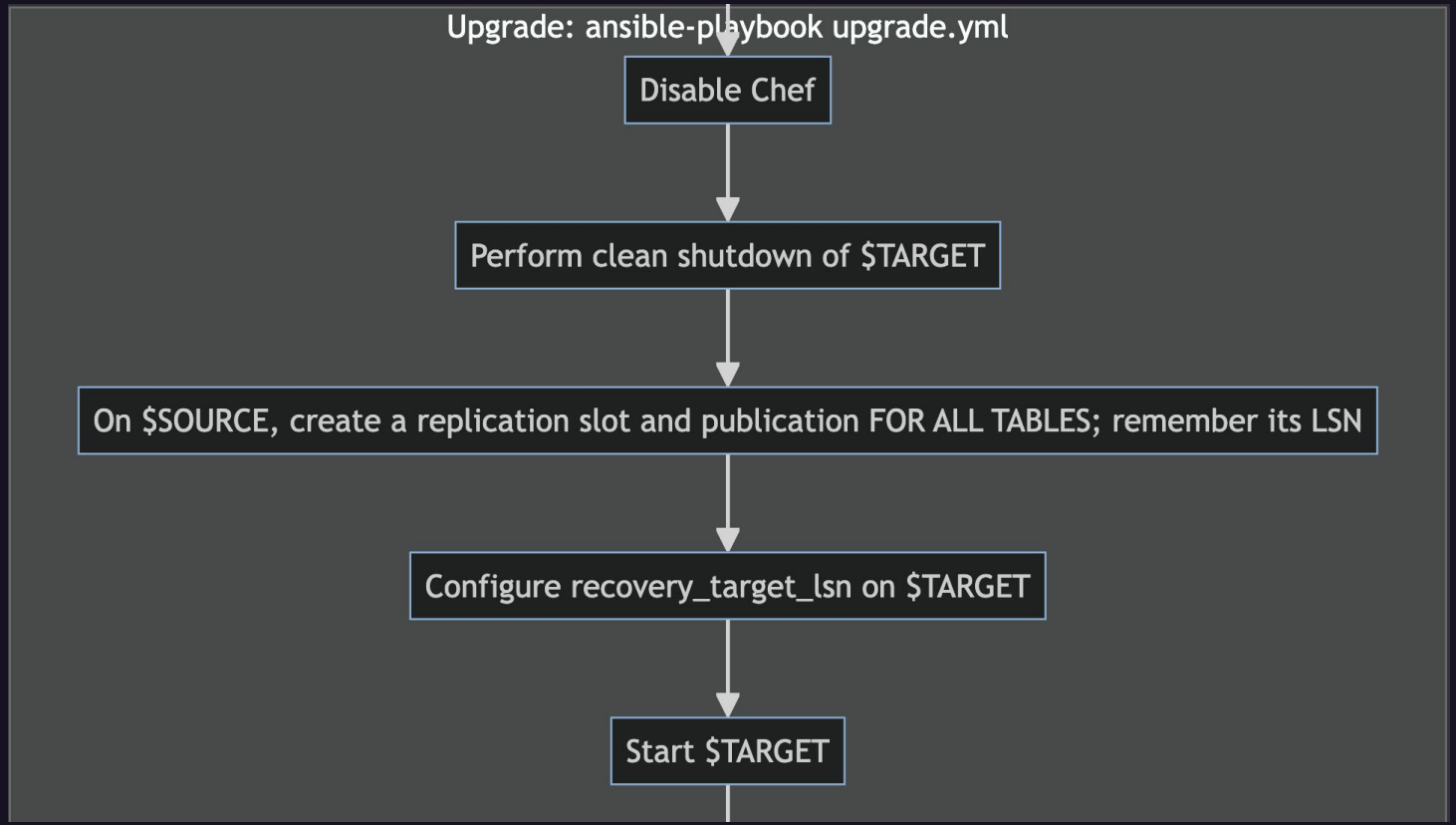
Upgrade - 1. Prepare new Environment



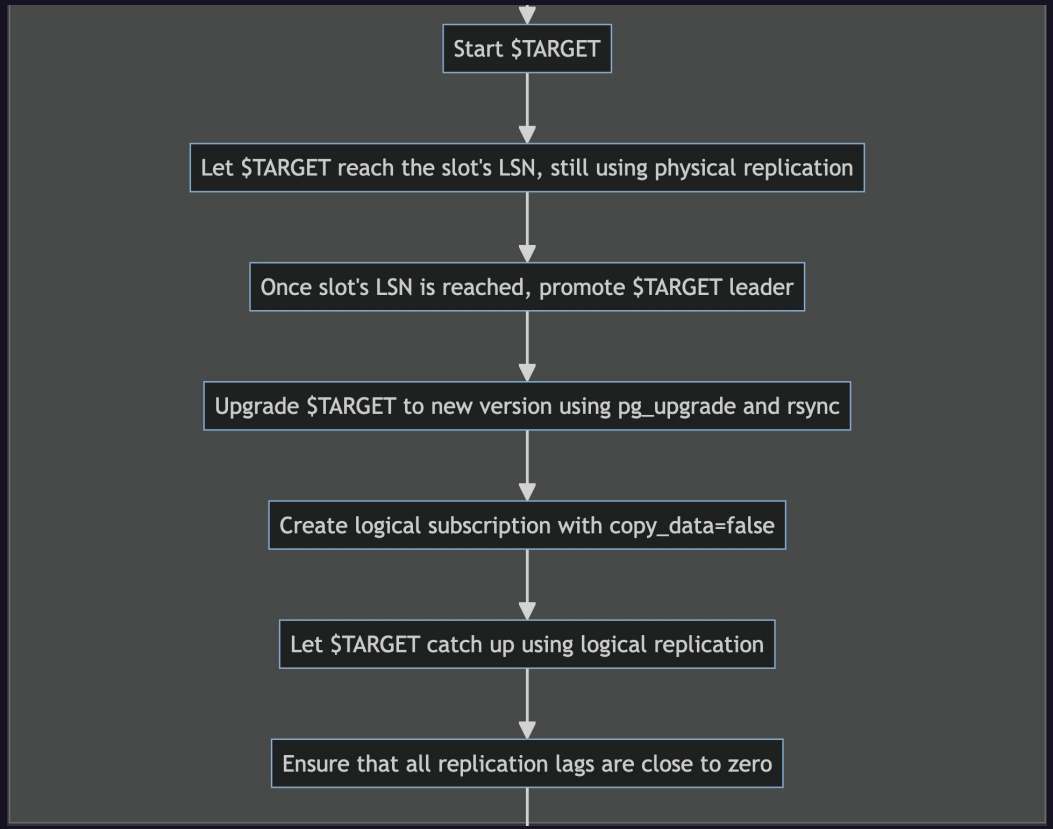
Upgrade - 2 Upgrade new Cluster



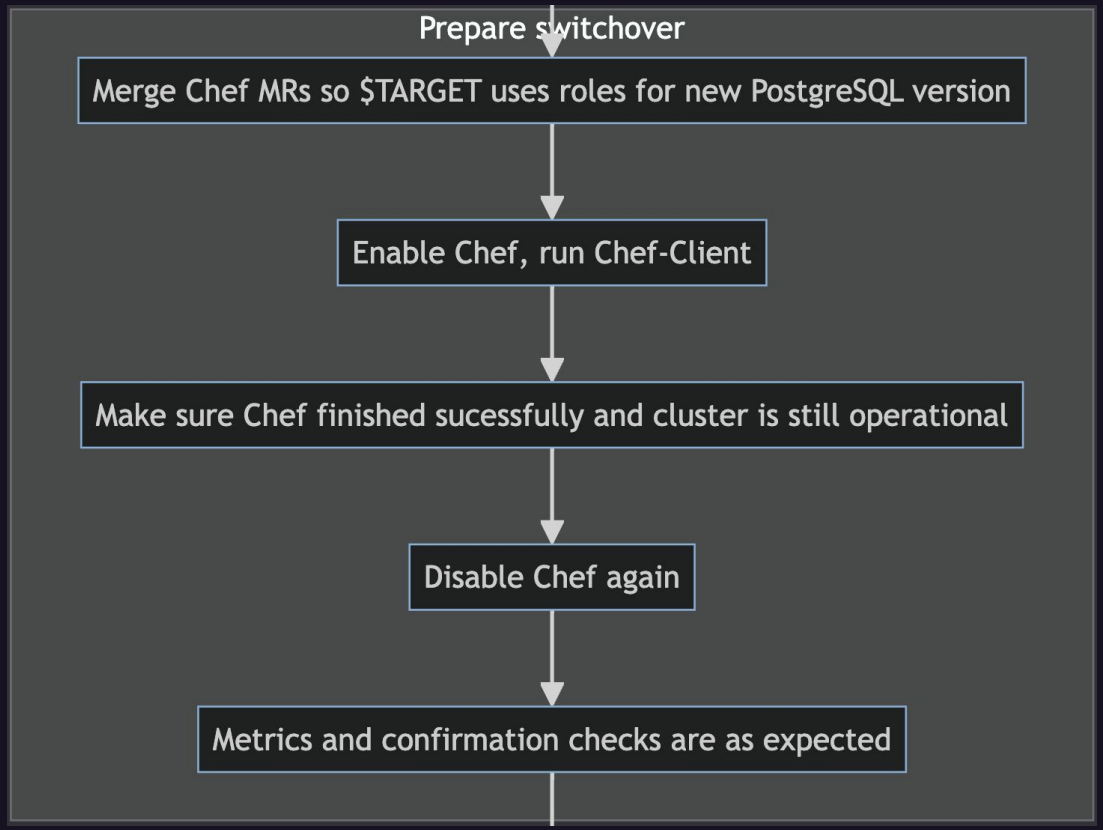
Upgrade - 2.1 Upgrade new Cluster



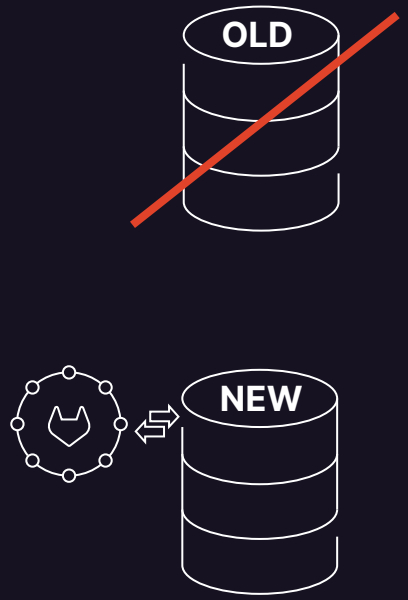
Upgrade - 2.2 Upgrade new Cluster



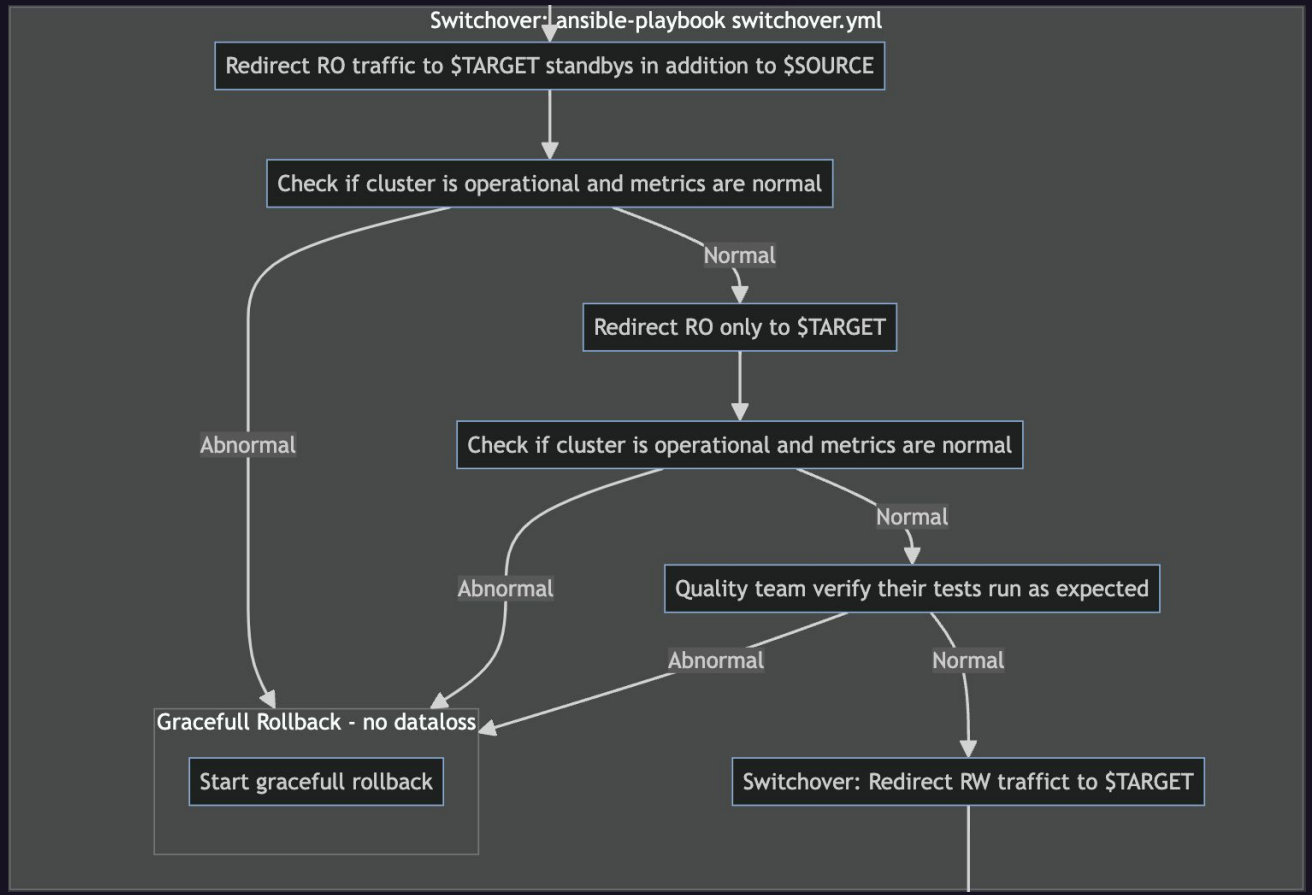
Upgrade - 3. Prepare Switchover



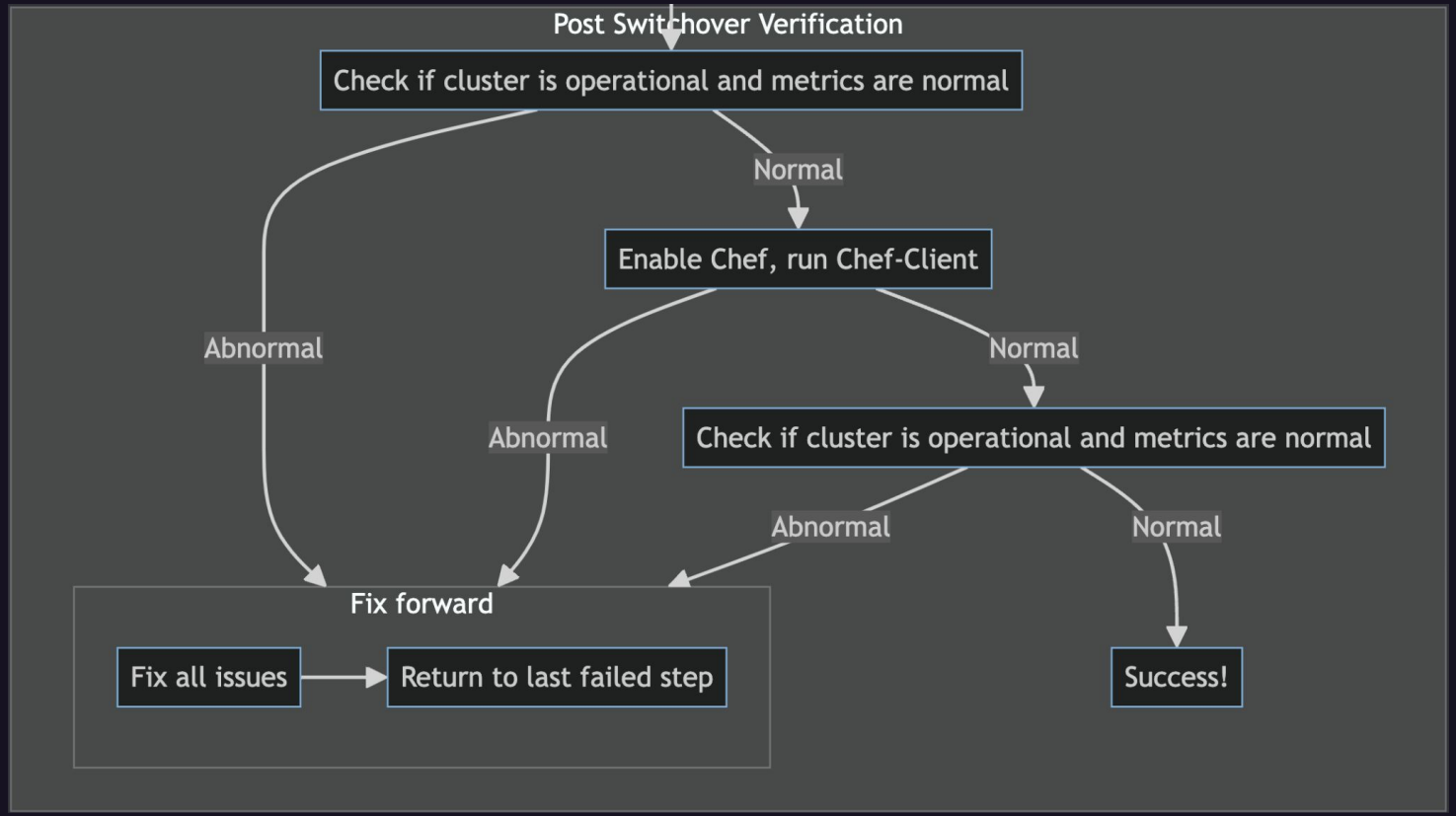
Upgrade - 4. Switchover



Upgrade - 4. Switchover

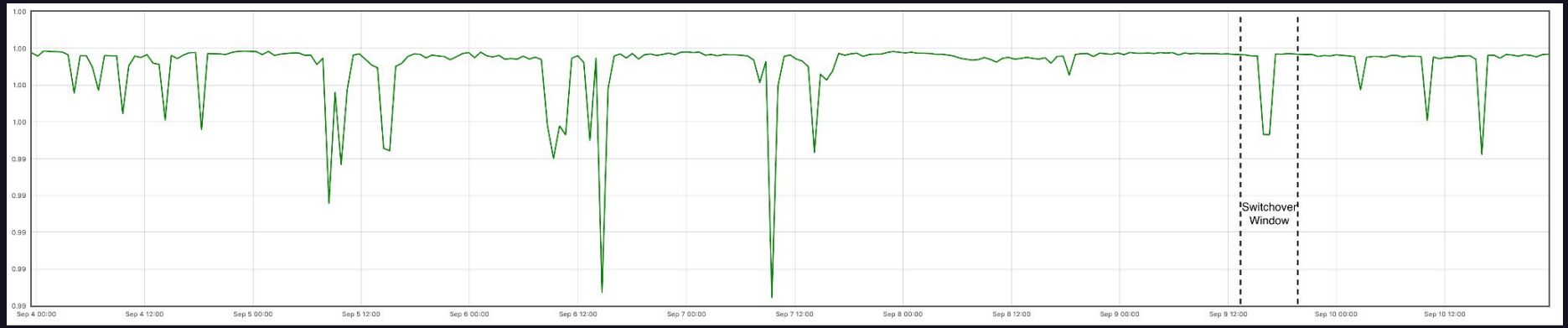


Upgrade - 5. Post Switchover Verification

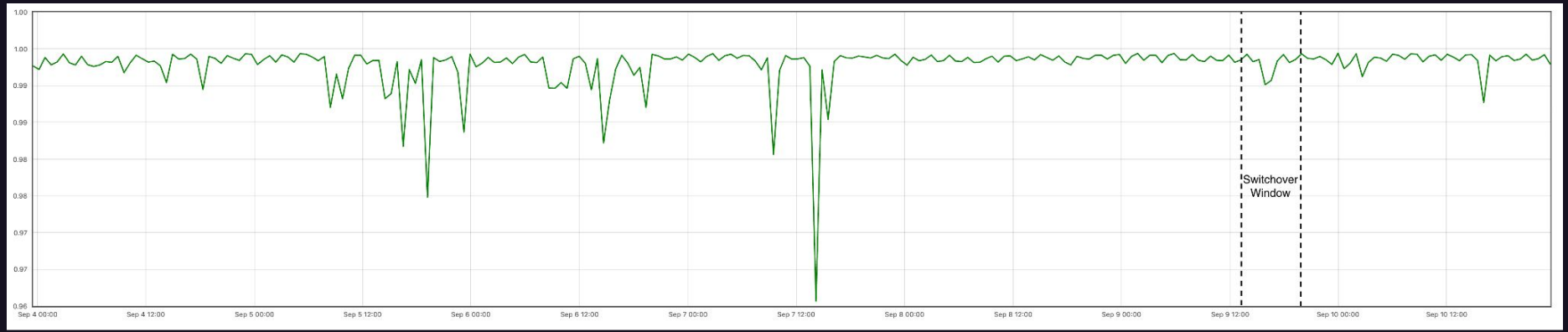


Did we improve?

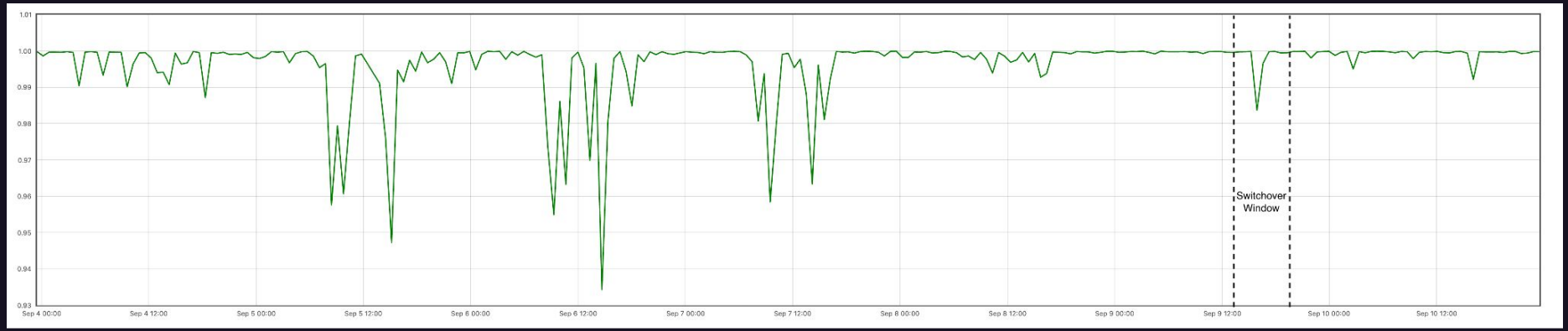
How well did we do? - Web



How well did we do? - API



How well did we do? - Git

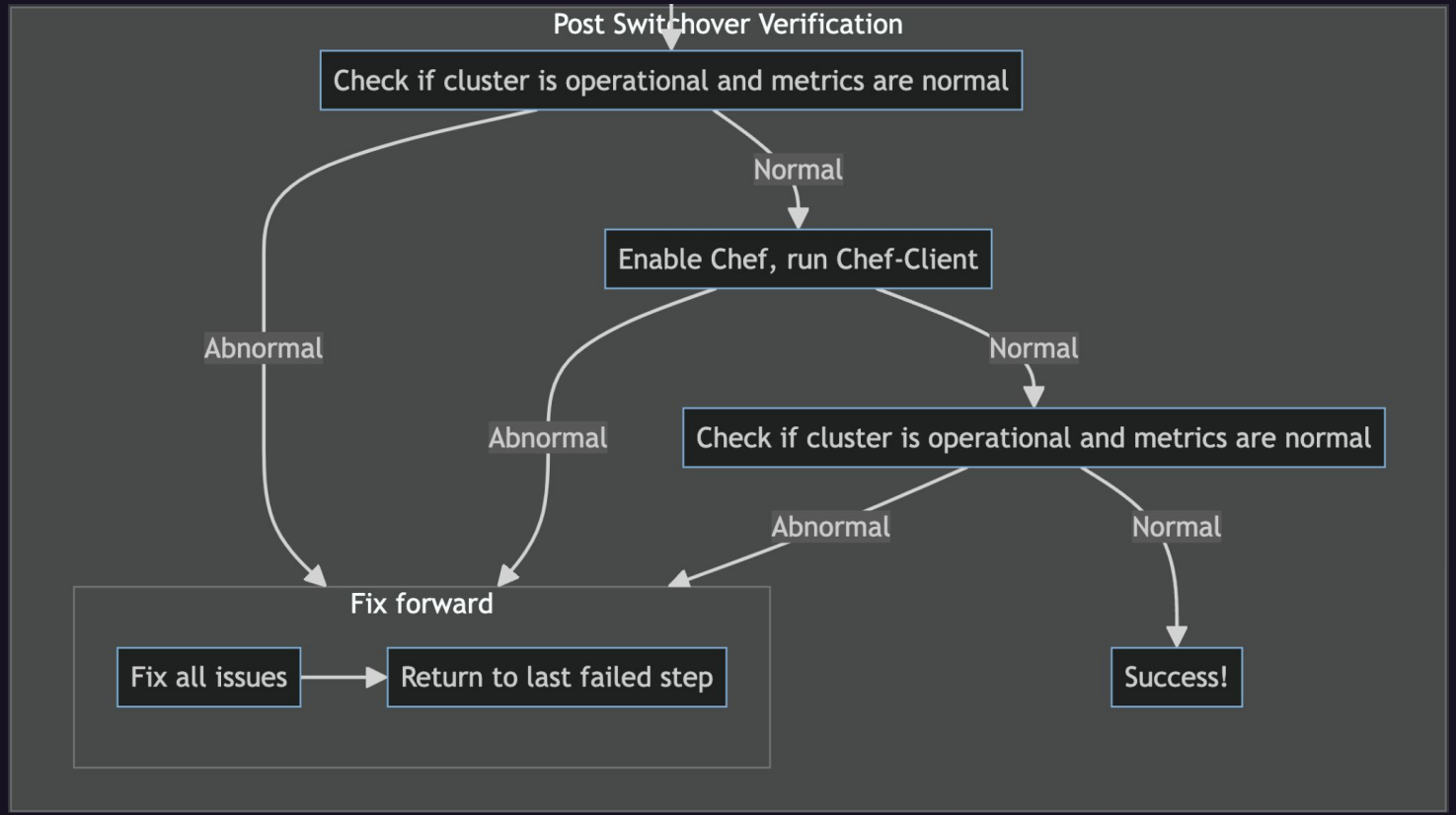


Upgrade - Possible Improvements?

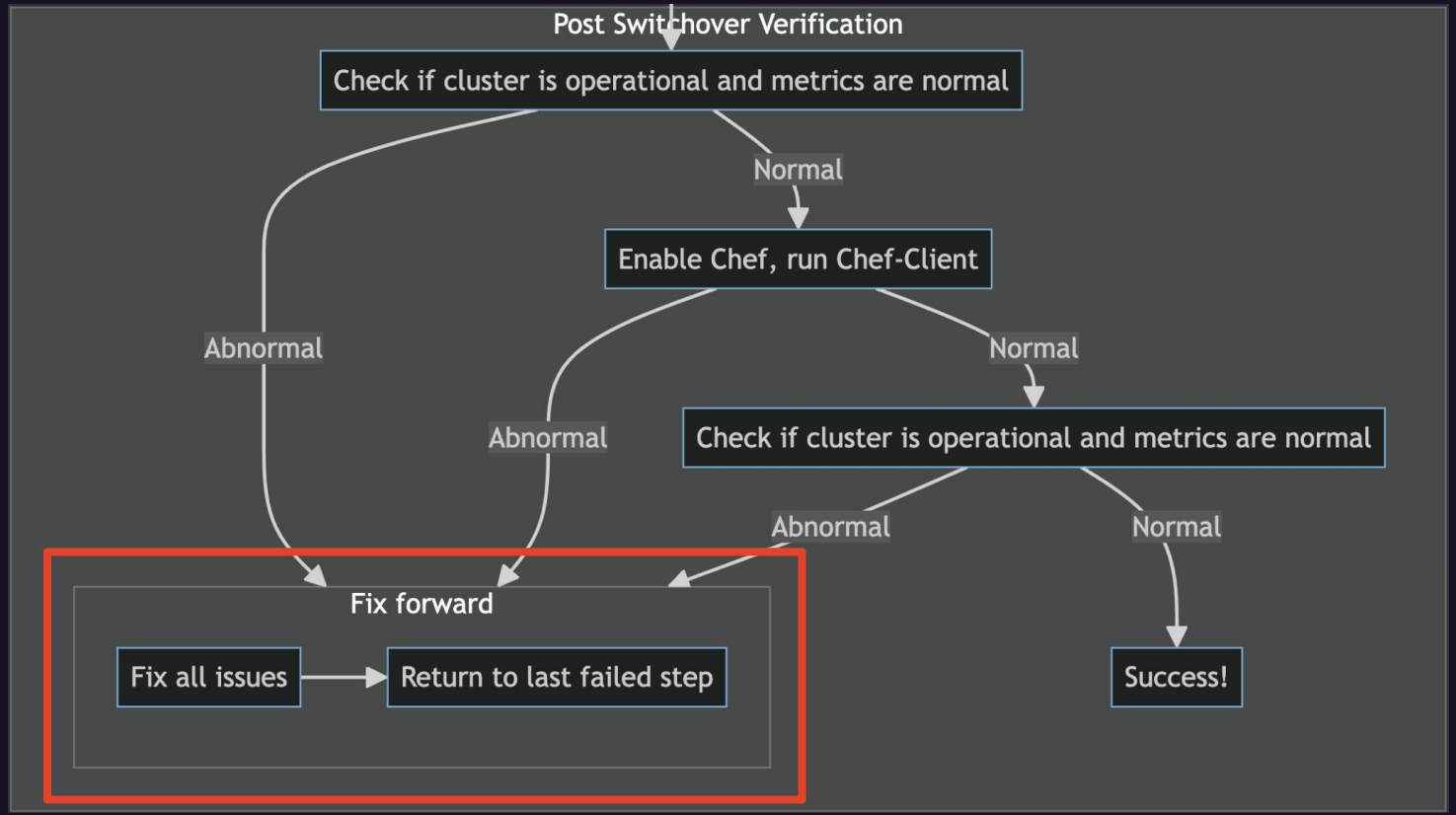
Reverse Replication

- After the Switchover, the old cluster replicates data from the new cluster
- Enables late rollback without data loss

Upgrade - Possible Improvements?



Upgrade - Possible Improvements?



Resources

- GitLab: about.gitlab.com
- The Handbook: about.gitlab.com/handbook
- Our RDBMS: about.gitlab.com/handbook/engineering/infrastructure/database
- Ansible Playbooks: https://gitlab.com/gitlab-com/gl-infra/db-migration
  - Developed with the help of postgres.ai
- Slide deck with addition annotations: 2023.pgconf.eu
  - Please leave Feedback!
- Alexander Sosna
  - about.gitlab.com/company/team/#alexander-sosna
  - sosna.de

Interested in ALL the details?

- Main epic https://gitlab.com/groups/gitlab-com/gl-infra/-/epics/642
  - [PHASE-1][PG13] Investigate migration approaches
  - [PHASE-2][PG14] Benchmark the performance and tuning for PG14
  - [PHASE-3][PG14] Playbook development & benchmark testing
  - [PHASE-4][PG14] Rollout upgrade in [GSTG]
  - [PHASE-5][PG14] Rollout upgrade in [GPRD]
  - [PHASE-6][PG14] Post-Upgrade tasks
- Flow chart source https://gitlab.com/gitlab-com/gl-infra/db-migration/-/blob/master/.gitlab/issue_templates/pg14_upgrade.md?ref_type=heads#high-level-overview

Questions?

- Now!
- During the event!
- Later!
  - sosna.de