

PGCLOUD

The Master Book

Zheng Lin Lei



PGCLOUD

Zheng Lin Lei

November 7, 2026

Version 1.1.0

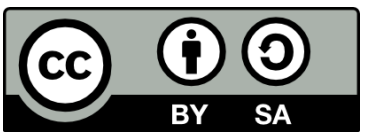
This wiki will begin by explaining in detail the configurations and knowledge required to create Postgres, PgPool and PgBouncer clusters step by step, requiring manual tweaks as if it were a tutorial. Later, the installation and execution manual for PgCloud will be incorporated, in which the system will deploy, write configuration and prepare the cluster of all the layers together with failover configuration and more automatically on the servers indicated for the creation of nodes.

Please note that this wiki will only teach content that is necessary and useful to know how to use PgCloud and its tools. THIS IS NOT AN OFFICIAL WIKI FOR ANY TOOL USED IN PGCLOUD.

Postgres Manual: https://www.postgresql.org/docs/16/
PgPool Manual: https://www.pgpool.net/mediawiki
PgBouncer: https://www.pgbouncer.org/usage.html

All content on the wiki will be updated as updates are released. But we gratefully consider it necessary to validate its veracity with our online wiki () and wiki on Github (). Wiki book written by @Zheng Lin Lei and revision done by @Sankey r3 and @tostadanevera.

This project manual is protected under CC BY-SA 4.0 Licence, any partial or complete appropriation of the work is not allowed.



- PgCloud -

1 - Introduction....5
2 - Diagram....6
2.1 - Two-tier Architecture....6
2.2 - Three-tier Architecture....7
2.3 - Other Architecture....7
3 - Developer Wiki....9
3.1 - Replication Layer....9
3.1.1 - Configuration....11
3.1.1.1 - Enable WAL....11
3.1.1.2 - Primary Role Database....12
3.1.1.3 - Replica Role Database....14
3.1.1.4 - Docker Compose Example....14
3.1.2 - WAL Define....16
3.1.2.1 - WAL Logging Levels....16
3.1.2.2 - WAL Sender....17
3.1.2.3 - WAL Receiver....20
3.1.2.4 - Example Of Each WAL Method....22
3.1.2.4.1 - File-Based Log Shipping....22
3.1.2.4.2 - Streaming Replication....22
3.1.3 - Synchronisation....24
3.1.3.1 - Replication Method....25
3.1.3.2 - Streaming Replication - How is information sent?....29
3.1.3.3 - File-Based Log Shipping - How is information sent?....30
3.1.3.4 - Synchronisation rule....30
3.1.3.5 - Recovery of Data After a Failover....31
3.1.4 - Backup....34
3.1.4.1 - WAL Archiving and Point-In-Time Recovery (PITR)....35
3.1.4.2 - Pg_Basebackup Tool....36
3.1.5 - Roles....37
3.1.5.1 - Primary node....37
3.1.5.2 - Replica node....37
3.1.5.3 - Publisher and Subscriber....38
3.1.6 - MultiNodes....39
3.1.7 - Replication slot....40
3.2 - Load Balance Layer....43
3.2.1 - Configuration....43
3.2.1.1 - General Configuration....43
3.2.1.2 - Backend Configuration....45
3.2.1.3 - Watchdog Configuration....47
3.2.2 - PgPool PCP....49

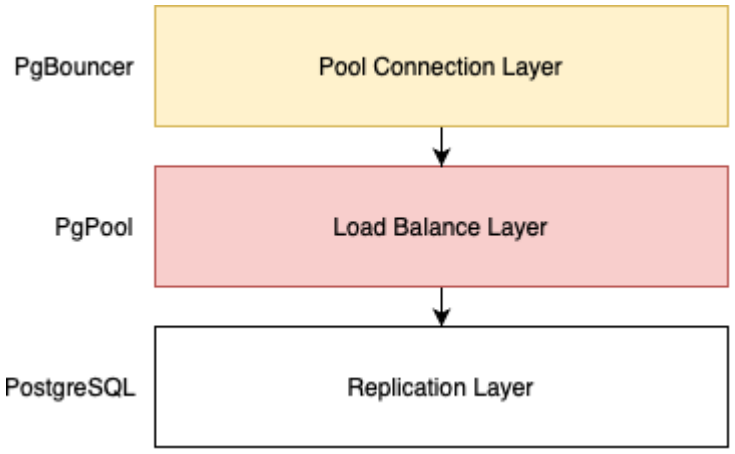
3.2.2.1 - What is PCP Used For? 49
3.2.2.2 - How PCP Works 49
3.2.2.3 - How to Use PCP 50
3.2.2.3.1. Configuration 50
3.2.2.3.2. Using PCP Command Line Tools 50
3.2.3 - Health Check 51
3.2.4 - Failover and Failback 54
3.2.4.1 - Failover 54
3.2.4.2 - Failback 57
3.2.5 - Replication Delay 58
3.2.5.1 How PgPool Detects Replication Delay 58
3.2.5.2 - Configuring PgPool for Replication Delay Management 59
3.2.6 - Watchdog 60
3.2.6.1 - History of PgPool Watchdog 60
3.2.6.2 - Functionality and How Watchdog Works 60
3.2.6.3 - Key Features of PgPool Watchdog 61
3.2.6.4 - Setting Up PgPool-II Watchdog (Alternative method) 62
2.2.6.5 - Virtual IP Configuration 63
2.2.6.6 - Starting the PgPool-II Cluster 63
2.2.6.6.1 - Watchdog Failover Flow 63
2.2.6.6.2 - Additional Considerations 63
2.2.6.7 - Conclusion 64
3.2.7 - Initial Children and Pooling Algorithm 65
3.2.7.1 - Max Pooling Algorithm and Connection Management 66
3.2.7.2 - Formula for Managing Connections 67
3.2.7.3 - Connection Pooling and Reuse 68
3.2.7.4 - Connection Limits and PgPool-II Behavior 69
3.2.7.5 - Best Practices for Configuring PgPool-II and PostgreSQL Connections 69
3.3 - Connection Pooling Layer 71
3.3.1 - Key Features of PgBouncer 71
3.3.2 - PgBouncer Modes 72
3.3.3 - Configuration 72
User Authentication 73
3.3.4 - Extra information 73
3.3.5 - Pooler working 74
3.3.5.1 - Connection Pooling Basics 74
3.3.5.2 - Pooling Modes 74
3.3.5.3 - Pooling Configuration Parameters 75
3.3.5.4 - Connection Lifecycle Management 76
3.3.6 - Switch Over 76
4. The Big Manual 78

4.1 - Before you start 78
4.2 - Getting the images 80
4.3 - Way A: automatic deployment with the PgCloud Launcher 81
4.4 - Way B: single host with Docker Compose 82
4.5 - Way C: manual deployment, server by server 82
4.6 - Configuration reference 85
4.7 - Operating the cluster 86
4.8 - Security checklist 89
4.9 - Troubleshooting 90
5. Tutorial for noobs 92
5.1 - What you need 92
5.2 - Step 1: start the launcher 92
5.3 - Step 2: meet the interface 93
5.4 - Step 3: check the settings 94
5.5 - Step 4: draw your cluster 95
5.6 - Step 5: tell the launcher how to reach each server 96
5.7 - Step 6: deploy 99
5.8 - Step 7: watch it live 101
5.9 - Step 8: break it on purpose 104
5.10 - Everyday operations 106
5.11 - Try it on your laptop 110
6. Tests and Benchmarks 111
6.1 - Test environment 111
6.2 - Functional tests 112
6.3 - Benchmarks 114
6.4 - Failover under load 116
6.5 - Behaviour observed and defects fixed 117
6.6 - Switchover and watchdog (1.1.0) 118
6.7 - Known limitations of 1.1.0 119

1 - Introduction

PgCloud is an automatic system for generating distributed nodes of PostgreSQL, PgPool and PgBouncer without the need to remain on a physical machine, since it uses docker technology. The system is capable of generating Postgres nodes with database replication on the same or different machines, load balancing with PgPool and connection caching with PgBouncer.

There are several architectures offered by PgCloud tested perfectly by our team. There is even the possibility of customising the architecture with some extra configuration. The architecture is separated by layers, specifically three: Replication Layer, Load Balancing Layer and Connection Pool Layer.

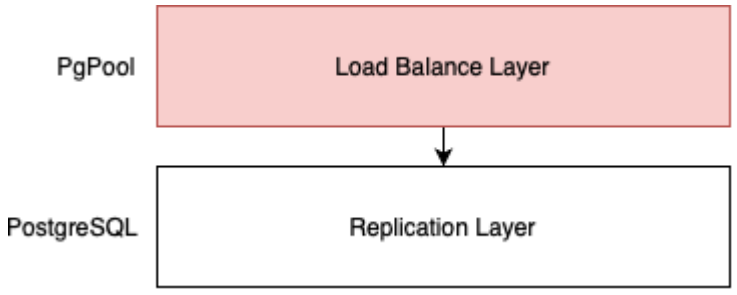


The three-tier architecture is the most used in environments where it requires much more data load and data consistency is prioritised, but its disadvantage is the time cost per request. (Section 2.2)

Please note that there may be a bottleneck in PgBouncer if you do not place the nodes on machines with sufficient resources.

View more: https://en.wikipedia.org/wiki/CAP_theorem

The other two-tier architecture is preferred for systems where time is prioritised, but the amount of load is low. It is also the fastest in preparing the cluster in case of a node failover. Since there is less communication and it is easier to reach a node agreement.

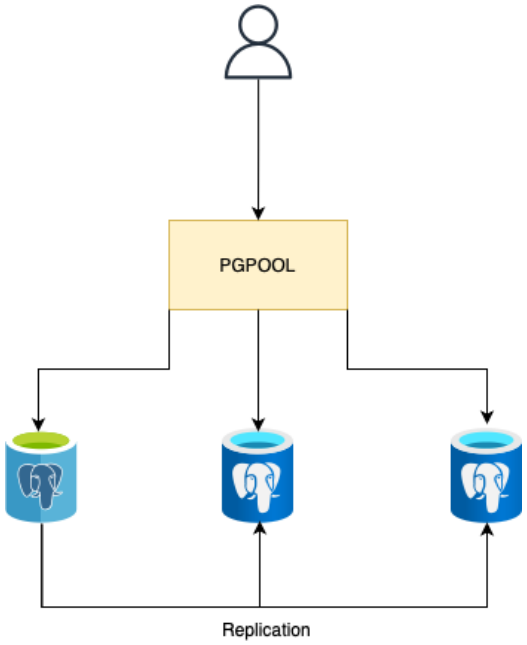


By default the system is enabled to launch this architecture (section 0)

2 - Diagram

Each layer architecture has different conditions of nodes to occupy, below is an example of a complete architecture with connections. Each figure represents a node and not a physical machine, the decision of putting one or two different nodes in the same machine depends on the load and stability that is desired to give to the system.

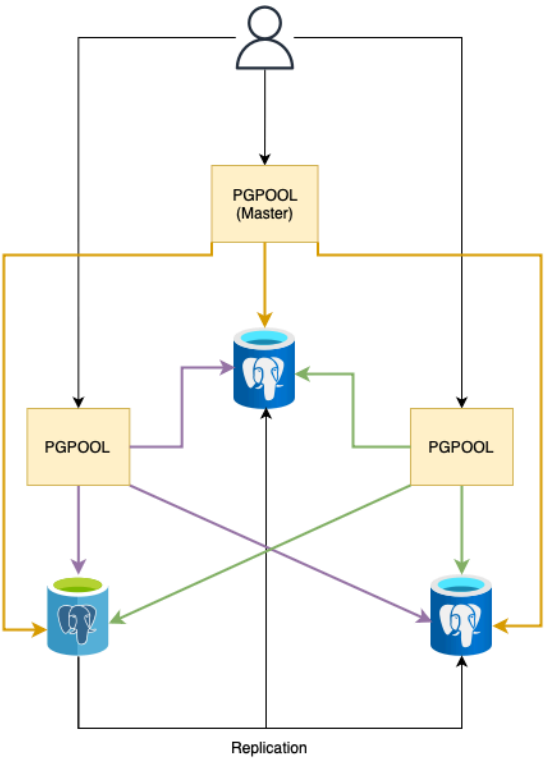
2.1 - Two-tier Architecture



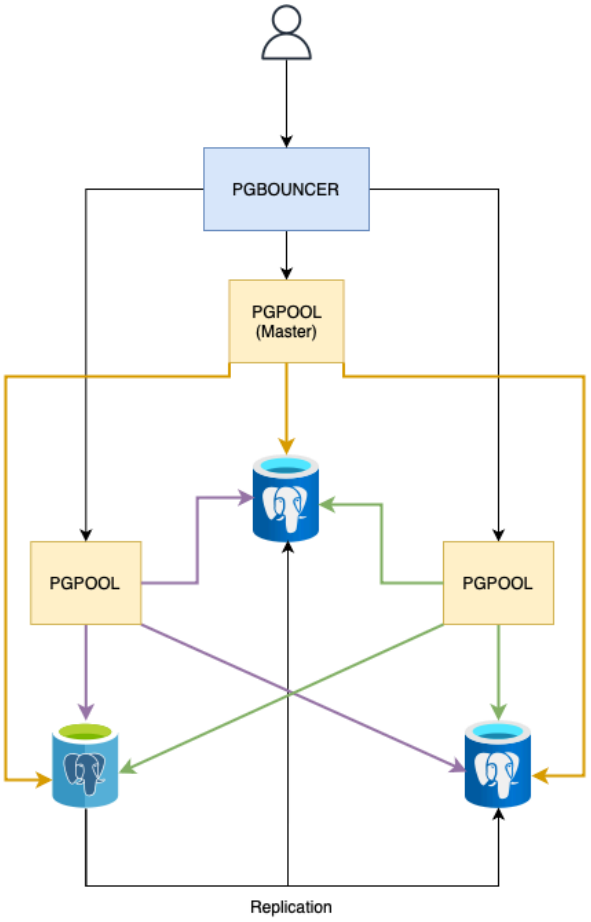
This is the most basic configuration that we can configure to offer load balancing. Its weakness is that it does not have a PgPool backup when a node goes down, the entire system stops receiving requests. Although it is undoubtedly the least expensive, optimal and simple for simple systems where it does not require much availability and load.

To configure this architecture, follow the configuration (Section 0) and disable Watchdog on PgPool

Creating a PgPool cluster alongside the PostgreSQL cluster helps to solve both the failover of a postgres node and a PgPool node. Throughout this documentation we will use primary and replica to map the postgres and master and slave roles for the PgPool cluster.



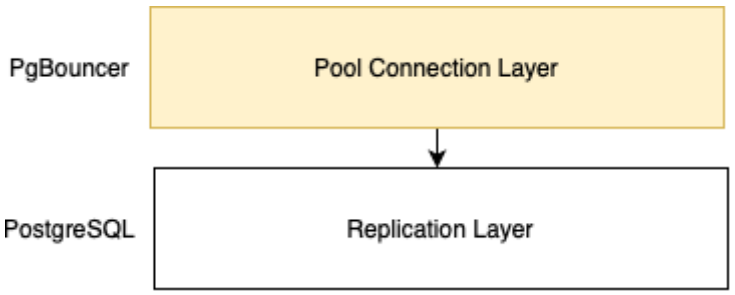
2.2 - Three-tier Architecture



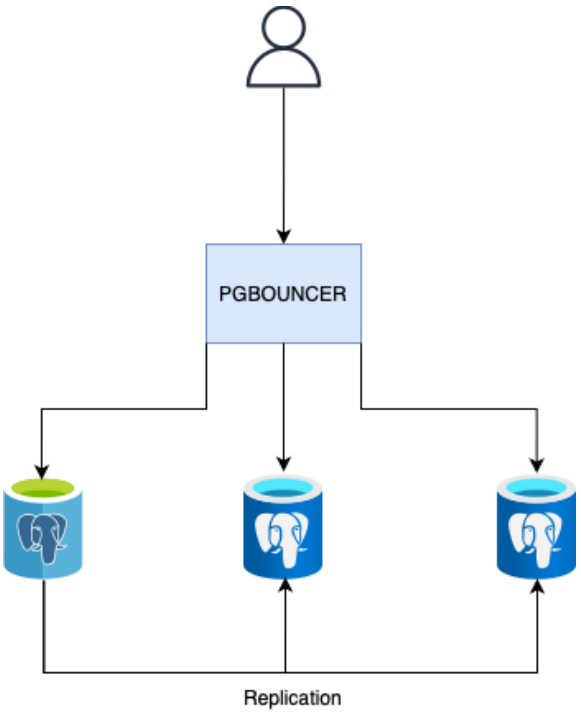
With a three-tier architecture we have better consistency between nodes, this way if PgBouncer were to fail we could replace it with a PgPool gate, losing the capacity offered by PgBouncer but obtaining more availability. With this we are able to create a distributed system that offers load balancing and connection caching for fast management.

2.3 - Other Architecture

You can also use the system where you omit the load balancing layer, that is, use PgBouncer with Postgres. This architecture is not the most common but it does offer quite a few advantages if you only require these two modules.



An example using this layer is:



PgBouncer is an optimal choice for efficiently pooling database connections, especially when your system needs connection caching rather than load balancing. Unlike PgPool, which offers additional features like load balancing, PgBouncer focuses solely on connection pooling. This makes it lightweight, with lower memory consumption and minimal configuration, ideal for handling high connection rates while maintaining a stable database environment.

3 - Developer Wiki

This section explains all the content in a technical way about the technologies, steps and testing used for the development of PgCloud. The content of this wiki is exclusive with parameters that are required in PgCloud. If you want to learn more, you will always be provided with more information in each section so that you can browse the official sources of the tools. THIS CONTENT IS FOR EDUCATIONAL USE ONLY.

If your purpose of reading the wiki is to learn how to use PgCloud, go to Section 4

3.1 - Replication Layer

The replication layer within this system is effectively managed by PostgreSQL, which plays a crucial role in ensuring data consistency and high availability across multiple synchronised servers. This replication mechanism is integral to the system's interconnection architecture, and several key aspects define its implementation and functionality:

- Cluster Configuration

  The PostgreSQL cluster is configured with a primary node, often referred to as the master, and multiple replica nodes, commonly known as slaves. The primary node is responsible for handling all write operations, while the replica nodes continuously synchronise with the primary to maintain up-to-date copies of the data.

  The replica nodes are not just passive copies; they are actively maintained and are capable of taking over the role of the primary node in the event of a failure. This failover capability ensures that service continuity is maintained even if the primary node becomes unavailable. The automatic promotion of a replica node to the primary role minimises downtime, which is critical in environments requiring high availability.

  Additionally, this replication setup enhances data redundancy and disaster recovery. By distributing copies of the data across multiple servers, the system safeguards against

data loss, ensuring that even if one or more servers fail, the integrity and availability of the data are preserved.

- Replication Method

PgPool optimises replication by using the stream replication method. This ensures that updates on the primary server are immediately transmitted to the secondary servers. This guarantees that data is always up-to-date on all replicas.

The replication method of stream replication is part of WAL distribution, also known as Write-Ahead Log (WAL) Shipping, which can be implemented in two different delivery modes: file-based log shipping and streaming replication. The primary server is configured to support both delivery methods (but is recommended to use stream), allowing replicas to use either option. However, PostgreSQL does not recommend using file-based WAL shipping and streaming replication simultaneously.

- File-based log shipping: (Section 3.3.1) In this method, WAL (Write-Ahead Log) segments are continuously written to a file until a certain size or time threshold is reached. These segments are then copied from the primary server to the replica via SSH or rsync. Once the replica receives the file, it applies the updates to its data and performs a cleanup process. This method is asynchronous, meaning there is a potential window for data loss if the primary server fails. It is recommended for backups or non-active replicas.

- Streaming replication: (Section 3.3.2) In this method, changes are transmitted from the primary to the secondary server in real-time. It can be either synchronous or asynchronous. Synchronous replication ensures no data loss but may impact the primary server's performance.

More types of replication:
https://www.postgresql.org/docs/current/different-replication-solutions.html

Cluster Official Documentation:
https://www.postgresql.org/docs/current/high-availability.html

3.1.1 - Configuration

The configuration explanations in this wiki will start by explaining the configurations that are exposed as customised in the configurations of the software used, and the values modified in the conf file (postgresql.conf and pgpool.conf). Read the manual configuration of PgCloud on (Section 6)

All system configurations are found in their respective .env files.

3.1.1.1 - Enable WAL

By default, the system settings enable WAL logging, which is essential for both file-based log shipping and streaming replication. This setting is not configurable because disabling it would result in the loss of replication functionality.

About WAL:
https://www.postgresql.org/docs/current/wal-intro.html

```
# postgresql.conf

# Values accepted minimal, replica y logical
# minimal: Logs only the information necessary for crash recovery.
#          It is not sufficient for replication or point-in-time recovery.
# replica: Includes enough information for real-time replication
#          (streaming replication) and for logical replication logs. This is
#          the default setting.
# logical: In addition to what "replica" includes, it adds the necessary information
#          for logical replication, enabling data publication and subscription.
#
#  Use of replica
wal_level = replica

# If synchronous mode is enabled, transactions are considered accepted only when
# the rules defined in `synchronous_standby_names` are met, specifically with the
# status of `remote_write`. This means that the WAL (Write-Ahead Log) has been
# written to the replica. If you want the transaction to be acknowledged only after
# the data has been written to the replica database itself, rather than just to the
# WAL, you should set it to `remote_apply`.
synchronous_commit = remote_write
# Wal log activated
wal_log_hints = on

# Get last timeline after a failover on recovery stage
recovery_target_timeline = 'latest'
```

You can get more detail information on WAL section (Section 3.2.1)

3.1.1.2 - Primary Role Database

To set up the primary node, you need to fill in the PostgreSQL configuration parameters in the Docker image and create the replication user. Those parameters are from the official Postgres Docker Image repository and are required to create the container.

For more information, visit: https://hub.docker.com/_/postgres#:~:text=POSTGRES_DB

```
# This data is required for the creation of the database with the docker image. The data
# is needed for the primary database
POSTGRES_DB=db_pgcloud
POSTGRES_USER=u_pgcloud
POSTGRES_PASSWORD=p_pgcloud
PASSWORD_AUTHENTICATION=md5
PGDATA=/opt/pg_data
```

Container launching.

```
$ docker run -d \
        --name pgcdb \
        -e POSTGRES_PASSWORD=p_pgcloud \
        -e POSTGRES_USER=u_pgcloud \
        -e PGDATA=/opt/pgdata \
        -e POSTGRES_DB=db_pgcloud \
        -v pgcdata:/opt/pgdata \
        postgres
```

It is important to know that if you modify the value of PGDATA you must modify the volume path in the script that launches the containers. (Section 0)

Now we need to create a replica user with replication permission. To create a replication user in your database, you can utilise the `/docker-entrypoint-initdb.d` directory for additional initialization tasks. If you're working with an image based on this one, simply place any necessary .sql, .sql.gz, or .sh scripts into the /docker-entrypoint-initdb.d directory (create this directory if it doesn't already exist).

During the container startup, after the initdb process creates the default postgres user and database, the entrypoint script will execute any .sql files, run any executable .sh scripts, and source any non-executable .sh scripts located in this directory. This allows you to perform further initialization before the database service begins. (This action will be executed once)

Save any .sql file with this content to the directory and Postgres will execute it at database initialization.

```
-- init.sql

-- Valor del archivo init.sql que se ejecutará por postgres automáticamente
CREATE ROLE pg_replica WITH REPLICATION LOGIN PASSWORD 'p_pgcloud';
```

After all it is time to modify the postgresql.conf configuration file, please note that we have done some previous configurations (Section 3.1.1.1), the configuration required for primary database are:

```
# postgresql.conf

# Enable timeout to raise an error if no replica is receiving data
wal_sender_timeout = 1000ms    # in milliseconds; 0 disables it

# Enable Stream Replication for real-time synchronised updates
# You can see additional configuration in "synchronous_commit"
# The possible values are:
# *                               :   All replicas must receive the copy before
#                                     the transaction is confirmed

# ANY {INT} (*)                   :   At least x nodes must be updated to the latest
#                                     commit. Example: ANY 2 (*)

# ANY {INT} ({NODE NAME LIST})    :   At least x nodes from the list must be updated

# ({NODE NAME LIST})              :   All nodes in the list must be updated

# {EMPTY}                         :   None are synchronous; all nodes will be
#                                     asynchronous
synchronous_standby_names = 'FIRST 1 (*)'
```

More information:
https://www.postgresql.org/docs/current/runtime-config-replication.html

Synchronisation rule:
Section 3.1.3.3

3.1.1.3 - Replica Role Database

There is not much configuration required on the replica, because everything the replica needs can be obtained from the primary. The only requirement is to have an empty pgdata folder.

As the default postgres image creates a folder with database content, the first thing we will do is delete it once the container is launched.

Once deleted, run a command from a postgres tool to synchronise the data:

```
$ pg_basebackup -h ${PRIMARY_HOST} -p ${PRIMARY_PORT} -U ${REPLICA_USER}
                -d "host=${PRIMARY_HOST} port=${PRIMARY_PORT} user=${REPLICA_USER}
                  dbname=${POSTGRES_DB} application_name=${BACKEND_NAME}"
                -X stream -C -S replica$slot -v -R -w -D ${PGDATA}
```

For more details of pg_basebackup (Section 3.1.4.2)

It is important to highlight the `$slot` variable. A replica slot is the identifier where the primary node stores the data that will be shared with the replica. Losing the slot means losing the replication history, which can compromise the integrity of the replica. (Section 3.1.7)

The PostgreSQL file configuration for replica includes the following aspects:

```
# postgresql.conf

primary_slot_name = 'replication_slot'
hot_standby = on     # on means that the replica is read-only
```

It is recommended to have the configuration of the primary and replica databases configured from a pre-configured database. The configuration does not overlap since the role that the database will have will apply its corresponding configuration from the postgres.conf file. In this way we can promote and depromote automatically without having to have human intervention.

3.1.1.4 - Docker Compose Example

Here you have an example of a docker-compose file for launching two containers with custom data. One primary and the other replica:

```
# docker-compose.yaml

x-postgres-common:
 &postgres-common
 image: postgres:16-alpine
 user: postgres
 restart: always
 healthcheck:
   test: 'pg_isready -U user --dbname=db_pgcloud
   interval: 10s
   timeout: 5s
   retries: 5

services:
 postgres_primary:
   <<: *postgres-common
   ports:
     - 5432:5432
   environment:
     POSTGRES_USER: u_pgcloud
     POSTGRES_DB: db_pgcloud
     PGDATA: /opt/pgdata
     POSTGRES_PASSWORD: p_pgloud
     POSTGRES_HOST_AUTH_METHOD: "scram-sha-256\nhost replication all 0.0.0.0/0 md5"
     POSTGRES_INITDB_ARGS: "--auth-host=scram-sha-256"
   command: |
     postgres
     -c wal_level=replica
     -c hot_standby=on
     -c max_wal_senders=10
     -c max_replication_slots=10
     -c hot_standby_feedback=on
   volumes:
     - ./init.sql:/docker-entrypoint-initdb.d/init.sql

 postgres_replica:
   <<: *postgres-common
   ports:
     - 5433:5432
   environment:
     PGUSER: pg_replica
     PGPASSWORD: p_pgcloud
   command: |
     bash -c "
     until pg_basebackup --pgdata=/opt/pgdata -R --slot=slot --host=postgres_primary
--port=5432
     do
     echo 'Waiting for primary to connect...'
     sleep 1s
     done
     echo 'Backup done, starting replica...'
```

```
    chmod 0700 /opt/pgdata
    postgres
    "
  depends_on:
    - postgres_primary
```

3.1.2 - WAL Define

Write-Ahead Logging is a technique used by PostgreSQL to ensure data integrity and durability. It involves recording changes to a log before they are applied to the database itself. This approach guarantees that in the event of a crash or failure, PostgreSQL can recover the database to its most recent consistent state by replaying these logs.

WAL serves to protect against data loss. By writing changes to a log file before applying them to the main database files, PostgreSQL ensures that it can recover committed transactions even if a system crash occurs.

Process:

- Logging Changes: When a transaction modifies the database, the changes are first recorded in the WAL. This log entry includes all the necessary information to redo the changes.
- Applying Changes: After the WAL entries are written, the changes are applied to the actual database files. This sequence ensures that the database can be restored to a consistent state by replaying the WAL entries in the event of a crash.

Recovery:

- During recovery, PostgreSQL uses the WAL to apply all changes recorded in the log since the last checkpoint, ensuring that no committed data is lost.

WAL files are stored in a specific directory within the PostgreSQL data directory. The exact location is configured in the PostgreSQL configuration file (postgresql.conf), but by default, WAL files are located in `$PGDATA/pg_wal`

3.1.2.1 - WAL Logging Levels

There are different levels of replication, and it may be interesting to apply one or the other depending on the purpose. PgCloud applies the replication level (formerly physical) because it is the minimum required for stream replication, while logical is too much data being transmitted where only 30% of the data is really useful for replication.

In PostgreSQL, different levels of Write-Ahead Logging (WAL) provide varying degrees of logging detail, tailored to different needs:

Minimal logging is designed to record only the essential information required for crash recovery. However, it has significant limitations, as it is insufficient for replication and does not support point-in-time recovery (PITR). This level is best suited for environments where minimal logging is adequate, and there is no need for replication or detailed recovery options.

Replica logging goes a step further by providing the necessary information for real-time replication and logical replication logs. It supports streaming replication, which allows continuous and real-time data synchronisation between primary and replica servers. This level is also suitable for logical replication, where specific datasets or tables can be replicated instead of the entire database. Replica logging is the default setting, ensuring the system is ready for replication by default.

Logical logging extends the replica logging level by adding information needed for logical replication. It supports all the features of replica logging, including streaming and logical replication. Additionally, it enables advanced replication features like data publication and subscription, allowing specific datasets to be replicated across different databases or systems. This level is ideal for complex replication scenarios where selective data replication or sharing data across multiple databases is required.

3.1.2.2 - WAL Sender

The WAL (Write-Ahead Log) sender, or WAL sender, is a key component in PostgreSQL's replication and high-availability architecture. It plays a crucial role in streaming replication by transmitting WAL records from the primary server to one or more standby servers. Here's an overview of how the WAL sender works and its importance in PostgreSQL:

- Role and Functionality

The WAL sender is a PostgreSQL background process that operates on the primary server. Its main responsibility is to send WAL records to standby servers in real-time. These WAL records contain a log of all changes made to the database, ensuring that the standby servers can apply these changes and stay synchronised with the primary server.

When a transaction is committed on the primary server, the changes are first recorded in the WAL before being written to the actual data files. The WAL sender picks up these changes from the WAL and streams them to the connected standby servers. This process allows the standby servers to apply the changes almost simultaneously, enabling them to quickly take over if the primary server fails.

- Operation

The WAL sender process is started when a standby server connects to the primary server for replication. The connection is established using the replication protocol, which is different from a standard client connection. Once connected, the WAL sender begins streaming WAL records starting from the last known consistent point in the standby server.

The WAL sender reads the WAL records from the primary server's pg_wal (or pg_xlog in older versions) directory and sends them over to the standby server. If the standby server falls behind, the WAL sender can stream older WAL segments to catch it up. The process continues as long as the replication connection remains active.

- Configuration and Management

Several configuration parameters in postgresql.conf control the behaviour of the WAL sender:

- max_wal_senders: This parameter specifies the maximum number of WAL sender processes that can run simultaneously on the primary server. This limits how many standby servers can connect for streaming replication.

- wal_keep_size: This setting determines how much WAL data is retained on the primary server for standby servers that may fall behind. This is critical in ensuring that the standby servers can catch up without needing to request WAL segments from archive storage.

- synchronous_commit: When set to on or configured with synchronous replication, this parameter ensures that transactions are not considered committed until the WAL records have been sent to and acknowledged by the standby server(s). This enhances data safety but may impact performance due to the added latency.

- wal_sender_timeout: This parameter sets the time that the WAL sender will wait for an acknowledgment from the standby before timing out. If the timeout is reached, the connection to the standby server is closed.

- archive_mode: Set to on if you want to enable File-Based log shipping, although streaming replication will take the sender job.

- archive_command: (Will be used if archive_mode is on) Command to send WAL files to the destination node, can be remote or local.

- WAL Sender States

The WAL sender can be in different states during its operation:

- Streaming: Actively sending WAL records to the standby server.
- Backup: Also known as File-based log shipping. Sending a base backup to the standby (if requested by the standby server, needs to be configured as a backup replica).
- Catchup: Sending WAL records to a standby server that is catching up after falling behind.

- Monitoring

The status of WAL senders can be monitored using the pg_stat_replication view in PostgreSQL. This view provides information such as the state of the WAL sender, the current location of the WAL records being sent, and the lag between the primary and standby servers.

- Importance

The WAL sender is crucial for maintaining the integrity and availability of data in a replicated PostgreSQL environment. By continuously streaming changes to standby servers, it ensures that there is minimal data loss in the event of a primary server failure. Additionally, it allows for load balancing by offloading read operations to the standby servers, enhancing overall system performance.

3.1.2.3 - WAL Receiver

The WAL (Write-Ahead Log) receiver, or WAL receiver, is the counterpart to the WAL sender in PostgreSQL's streaming replication architecture. While the WAL sender operates on the primary server to transmit WAL records, the WAL receiver runs on a standby server and is responsible for receiving these records and applying them to keep the standby in sync with the primary.

- Role and Functionality

The WAL receiver is a background process on a standby server that connects to the primary server's WAL sender. Its main function is to receive the stream of WAL records sent by the WAL sender and write these records to the standby's pg_wal (or pg_xlog in older versions) directory. Once the WAL records are received and written to disk, the standby server applies these changes to its local copy of the database, keeping it up to date with the primary server.

- Operation

When a standby server is started, it connects to the primary server using a replication connection. Once connected, the WAL receiver process is initiated. It listens for incoming WAL records from the primary server's WAL sender. The WAL receiver writes these records to the WAL directory on the standby server in the same order they were created on the primary server.

As the WAL records are written, the standby server applies them to its database files, replaying the changes just as they occurred on the primary server. This process ensures that the standby server is nearly an exact replica of the primary, ready to take over in case of failure.

- Configuration and Management

Several configuration parameters in postgresql.conf on the standby server control the behaviour of the WAL receiver:

- primary_conninfo: This setting specifies the connection string used by the WAL receiver to connect to the primary server. It includes details like the host, port, user, and password needed to establish the replication connection.

- primary_slot_name: If replication slots are used, this parameter specifies the name of the replication slot that the standby server will use. Replication slots ensure that the

primary server retains the necessary WAL segments until they have been successfully received by the standby.

- wal_receiver_timeout: This parameter controls how long the WAL receiver will wait for data from the primary server before timing out. If no data is received within this period, the WAL receiver will assume there is a problem with the connection.

- restore_command: Set this parameter if primary server File-Based log shipping method is enabled.

- wal_receiver_status_interval: This setting determines how often the WAL receiver will send feedback to the WAL sender about its current progress. This feedback helps the primary server manage WAL segment retention and the status of replication.

- WAL Receiver States

The WAL receiver process can be in several states during its operation:

- Streaming: Actively receiving WAL records from the primary server's WAL sender and writing them to the WAL directory.
- Waiting: Waiting for the next batch of WAL records to arrive from the primary server (restore_command).
- Stopping: The process of shutting down the WAL receiver, typically when the standby server is being stopped or the replication connection is being terminated.

- Monitoring

The status and activity of the WAL receiver can be monitored using views such as pg_stat_wal_receiver. This view provides information about the connection to the primary server, including the current WAL file being received, the amount of data received, and any potential lag in replication.

- Importance

The WAL receiver is critical for maintaining the synchronisation between the primary and standby servers in a PostgreSQL replication setup. By continuously receiving and applying WAL records, the WAL receiver ensures that the standby server is up to date and can take over with minimal data loss in the event of a primary server failure. It also plays a role in enabling

read-only queries on the standby server, allowing the load to be distributed between the primary and standby.

3.1.2.4 - Example Of Each WAL Method

Here you have some examples of configuration to set up the WAL shipping method you want to work with.

3.1.2.4.1 - File-Based Log Shipping

Primary:

```
wal_level = replica
# Maximum of 3 replicas
max_wal_senders = 3
# Wal log activated
wal_log_hints = on


# Active File-Based log shipping
archive_mode = on

# Can use cp and external process will send to destination or
# use rsync or scp
archive_command = 'rsync %p user@remote:/tmp/wal/%f'
```

Replica:

```
standby_mode = 'on'
primary_conninfo = 'host=127.0.0.1 port=5432 user=user password=pass'

# All WAL files send by primary will be copied to postgres pg_wal folder to be applied
restore_command = 'cp /tmp/wal/%f %p'
```

3.1.2.4.2 - Streaming Replication

Primary:

```
wal_level = replica
# Maximum of 3 replicas
max_wal_senders = 3
# Wal log activated
wal_log_hints = on
# Get last timeline after a failover on recovery stage
recovery_target_timeline = 'latest'

synchronous_commit = remote_write
# in milliseconds; 0 disables it, If timeout will send error to client
wal_sender_timeout = 1000ms
synchronous_standby_names = 'FIRST 1 (*)'
```

Replica:

```
standby_mode = 'on'
primary_conninfo = 'host=127.0.0.1 port=5432 user=user password=pass'

# In Streaming replication is recommended the use of slot to prevent data loss and
# conflict
primary_slot_name = 'slot'
```

3.1.3 - Synchronisation

When a node fails due to network problems or a process failure, it is possible to perform a failover managed by Pgpool (Section 3.2). However, PostgreSQL alone does not have the ability to recover a downed node automatically. For this reason, it is crucial that nodes are managed and monitored, either by a suitable middleware or through constant human supervision.

Replication slots are used to recover data from the last point in time where the replica left off (Section 3.1.7). This method of data synchronisation is the best and most optimal for a system that is recoverable since this way the primary will retain the information when the replica node is down and will not delete it until the replica receives the lost data. It is possible not to use this method and use automatic deletion of WAL writing, where it is possible to keep a configured amount of old WAL segments by using wal_keep_size, or by storing the segments in a file using archive_command or archive_library. However, these methods often result in retaining more WAL segments than necessary and even deleting necessary data because the limit has been exceeded and the fallen replica has not been recovered, whereas replication slots retain only the number of segments known to be needed. On the other hand, replication slots can retain as many WAL segments as fill the space allocated for pg_wal; max_slot_wal_keep_size limits the size of WAL files retained by replication slots. The best option is using replication slots with an indefinite max_slot_wal_keep_size.

To start synchronisation when doing a pg_basebackup and assigning the slot, create the primary_conninfo configuration in postgresql.auto.conf and create the slot on the primary:

```
$ pg_basebackup -h ${PRIMARY_HOST} -p ${PRIMARY_PORT} -U ${REPLICA_USER}
                -d "host=${PRIMARY_HOST} port=${PRIMARY_PORT} user=${REPLICA_USER}
                  dbname=${POSTGRES_DB} application_name=${BACKEND_NAME}"
                -X stream -C -S replica$slot -v -R -w -D ${PGDATA}
```

For more details of pg_basebackup (Section 3.1.4.2)

Additionally, replication with slots can be created manually in a primary database with a SQL sequence.

```
# Generate slot for one replica
psql -At -U $POSTGRES_USER -d $POSTGRES_DB -p $PRIMARY_HOST -h $PRIMARY_PORT -c "SELECT
* FROM pg_create_physical_replication_slot('slot_of_replica');"
```

3.1.3.1 - Replication Method

Database replication in PostgreSQL can be achieved using several methods, each serving different use cases and offering varying levels of complexity and control.

Shared Disk Failover

Shared disk failover avoids synchronisation overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it were recovering from a database crash. This allows rapid failover with no data loss.

Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behaviour. One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

File System (Block Device) Replication

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system — specifically, writes to the standby must be done in the same order as those on the primary. DRBD is a popular file system replication solution for Linux.

Write-Ahead Log Shipping (Used in PgCloud)

Warm and hot standby servers can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the standby contains almost all of the data of the main server, and can be quickly made the new primary database server. This can be synchronous or asynchronous and can only be done for the entire database server.

A standby server can be implemented using file-based log shipping or streaming replication, or a combination of both.

Logical Replication

Logical replication allows a database server to send a stream of data modifications to another server. PostgreSQL logical replication constructs a stream of logical data modifications from the WAL. Logical replication allows replication of data changes on a per-table basis. In addition, a server that is publishing its own changes can also subscribe to changes from another server, allowing data to flow in multiple directions. For more information on logical replication. Through the logical decoding interface, third-party extensions can also provide similar functionality.

Trigger-Based Primary-Standby Replication

A trigger-based replication setup typically funnels data modification queries to a designated primary server. Operating on a per-table basis, the primary server sends data changes (typically) asynchronously to the standby servers. Standby servers can answer queries while the primary is running, and may allow some local data changes or write activity. This form of replication is often used for offloading large analytical or data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple standby servers. Because it updates the standby server asynchronously (in batches), there is possible data loss during failover.

SQL-Based Replication Middleware

With SQL-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries must be sent to all servers, so that every server receives any changes. But read-only queries can be sent to just one server, allowing the read workload to be distributed among them.

If queries are simply broadcast unmodified, functions like random(), CURRENT_TIMESTAMP, and sequences can have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast rather than actual data changes. If this is unacceptable, either the middleware or the application must determine such values from a single source and then use those values in write queries. Care must also be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (PREPARE TRANSACTION and

COMMIT PREPARED). Pgpool-II and Continuent Tungsten are examples of this type of replication.

Asynchronous Multi Master Replication

For servers that are not regularly connected or have slow communication links, like laptops or remote servers, keeping data consistent among servers is a challenge. Using asynchronous multi master replication, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules. Bucardo is an example of this type of replication.

Synchronous Multimaster Replication

In synchronous multi master replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking and commit delays, leading to poor performance. Read requests can be sent to any server. Some implementations use a shared disk to reduce the communication overhead. Synchronous multi master replication is best for mostly read workloads, though its big advantage is that any server can accept write requests — there is no need to partition workloads between primary and standby servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like random().

PostgreSQL does not offer this type of replication, though PostgreSQL two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED) can be used to implement this in application code or middleware.

As you can see, PgCloud uses the WAL shipping method and it will be the only method taught in this wiki, File-based log shipping and streaming replication. Below is an explanation of the data synchronisation process for both data shipping logics.

The steps that PostgreSQL will follow when a downed replica recovers, whether using file-based log shipping or streaming replication, are similar. It is important to note that in file-based log shipping, the process is asynchronous, so the primary server will start sending the files without waiting for a response from the replica. The recovery steps are as follows:

- Restart and Reconnect: When a downed replica recovers, the first thing it does is restart its recovery process and reestablish the connection with the primary server. The connection process is managed by the walreceiver process on the replica.

- Checking the Replica's State: The replica will ensure that its state is consistent with the primary. To do this, the walreceiver process will communicate with the primary to determine the last WAL log it has received and applied.

- Requesting Missing WALs: If the replica has been down for a period of time, it is likely that it is missing some WALs. The replica requests from the primary server the WAL files that were generated while the replica was down. These files contain the records of all the transactions that occurred on the primary during its downtime.

  The method of transmitting the WALs is different for the two submissions. See the next section. (Section 3.1.3.2)

- Applying the WALs: Once the replica receives the necessary WAL files, it begins applying them to bring its database up to date. This application process may involve reading transaction logs and applying them in the order in which they were produced to ensure consistency with the primary.

- Synchronisation and Commitment: After applying the WALs, the replica will verify that its state is consistent with that of the primary. This includes checking that there are no outstanding transactions or inconsistencies. Once verified, the replica sends a confirmation to the primary indicating that it has recovered and applied the WALs up to the most recent point.

  The following steps do not apply to file-based.

- Resuming WAL Replay: The replica resumes continuous replay of WAL files from the primary to stay in sync with ongoing transactions. This process is performed continuously and automatically as long as the primary continues to generate new WALs.

- Error Monitoring and Recovery: During and after the recovery process, the replica and primary monitor synchronisation for possible errors or delays. If problems are detected, they will be automatically attempted to be resolved or alerts will be generated for the administrator to review.

- Configuration Adjustments and Final Verification: Once the replica has been successfully synchronised, it may be necessary to adjust the configuration or perform a final verification to ensure that the replica is functioning correctly and that performance is optimal. This may include adjustments to the recovery.conf settings or replication parameters.

  These steps allow the replica to become synchronised and operational again, ensuring data consistency and minimising downtime during recovery. The efficiency and speed of the recovery process depends on several factors, including the volume of unsynchronized data and the performance of the network and hardware.

3.1.3.2 - Streaming Replication - How is information sent?

- Walsender process:

On the primary server, a process called walsender is responsible for sending WAL logs to the replicas. This process is started when a replica connects to the primary to start streaming replication.

- Walreceiver process:

On the replica, the walreceiver process is responsible for receiving the WAL logs sent by the primary's walsender. The walreceiver connects to the walsender using a PostgreSQL protocol connection (similar to a client connection).

- Data retrieval:

The replica sends its timeline and the slot that belongs to the primary, and the primary triggers a walsender to start transmitting all the missing data over the PostgreSQL socket until it is up to date. This process is automatic and no extra configuration is required. Its sending efficiency is doubtful compared to WAL log shipping. It can be said that it is almost similar.

- Communication Protocols:
WAL logs are transmitted using the PostgreSQL replication protocol, which is an extension of the standard PostgreSQL client-server communication protocol. This communication is done over TCP/IP sockets, providing a continuous and efficient connection.

- Acknowledgement and Data Flow:
The walreceiver process on the replica sends periodic acknowledgements of received and applied WAL logs back to the walsender process on the primary. This helps the primary manage the retention and archiving of WAL logs, knowing how far the replica is in sync.

3.1.3.3 - File-Based Log Shipping - How is information sent?

For this type of shipping, it is almost identical, but the target and channel for sending the information is made by the programmer's specifications in the configuration file. Its sending is done through an ssh that copies the WAL files once full to the replica machine, and the replica applies them when it detects new files. To see this configuration in detail (Section 3.1.2.3).

In summary, in the file-based log shipping method, WAL files are transferred using file copy tools such as rsync or scp. These files are archived on the primary and recovered on the replica using configured restore commands. This method is more manual and may have more latency compared to streaming replication, but it is useful in situations where real-time replication is not necessary or possible.

3.1.3.4 - Synchronisation rule

To ensure data consistency, it is necessary to implement a synchronous replication system. In the current system, synchronisation is applied to a single node with a FIRST rule. This configuration ensures that transactions are committed only after receiving confirmation from one of the designated standby servers.

In PostgreSQL, synchronous replication is configured using the synchronous_standby_names parameter in the postgresql.conf file. To use the FIRST rule with a single node, you can follow the example below:

```
synchronous_standby_names = 'FIRST 1 (standby1, standby2, standby3)'

# Transaction confirmation level
synchronous_commit = 'remote_write'

# Timeout after a wal is sent
wal_sender_timeout = 1000ms
```

In this example, what it does is wait for an ACK from the standby1 replica to be able to give an OK to the client that the operation (INSERT, UPDATE, DELETE type...) has been carried out correctly. If the standby1 does not send an ACK and the timeout occurs, standby2 is taken into account; if it has received the ACK from 2 while waiting for 1, it will give the OK, otherwise it will start to call the timeout and so on.

A request is failed when no node has responded to the primary.

3.1.3.5 - Recovery of Data After a Failover

In a PostgreSQL cluster, during a failover event, the roles of the primary and replica nodes shift, requiring a coordinated recovery process to maintain data integrity. When the primary node fails, a failover is triggered, and one of the replica nodes is promoted to become the new primary. The old primary, once it is brought back online, cannot immediately resume its previous role due to the risk of data inconsistency. To address this, the old primary must either be reconfigured as a replica or restored as a new primary, but only after ensuring its data is synchronised with the new primary. Meanwhile, the replica that has been promoted to primary will complete the failover process by applying any outstanding Write-Ahead Log (WAL) files, ensuring it is fully up-to-date before switching to its new role as the primary database. The other replicas in the cluster will then recognize this change and begin replicating from the new primary instead of the old one.

A key tool in this recovery process is pg_rewind, which plays a crucial role in synchronising the old primary with the new primary. pg_rewind works by identifying the point of divergence in the WAL logs between the old and new primary nodes. It then rewinds the old primary's state by removing or overwriting any data that diverged after this point, effectively rolling back any transactions that occurred post-divergence. This allows the old primary to catch up with the new primary and safely rejoin the cluster as a replica. For pg_rewind to function, it requires access to both the data directories and the WAL logs of the new primary. Once the

old primary has been rewound, it can be set up as a replica, starting to stream the latest data changes and eventually becoming fully synchronised with the new primary. This process minimises downtime and ensures that the PostgreSQL cluster remains consistent and reliable even after a failover event.

Let's walk through a practical example to illustrate how failover and recovery work in a PostgreSQL cluster, including the use of pg_rewind.

Imagine you have a PostgreSQL cluster with one primary node, Node A, and two replicas, Node B and Node C. Node A is handling all the write operations, while Node B and Node C are continuously replicating the data from Node A.

- Scenario: Primary Node Failure

One day, Node A suddenly crashes due to a hardware failure. As a result, the cluster needs to elect a new primary to ensure continued operation. The failover process begins, and Node B is promoted to become the new primary. This promotion involves applying all the outstanding WAL logs to make sure Node B has all the latest changes up until Node A's failure. Node B then starts accepting write operations as the new primary.

- Recovering the Old Primary (Node A)

Later, Node A is repaired and brought back online. However, Node A's data is now out of sync with the current state of the cluster because during the time it was down, Node B (now the new primary) has processed new transactions that Node A is unaware of.

At this point, Node A cannot simply rejoin the cluster as it is. It needs to be synchronised with Node B. This is where pg_rewind comes into play.

- Using pg_rewind

To synchronise Node A with Node B, you would run pg_rewind on Node A. The process works as follows:

1. Divergence Detection: pg_rewind first checks the WAL logs to find where Node A's and Node B's data began to differ. This divergence likely occurred at the moment Node A crashed.

2. Rewinding Data: Once the divergence point is identified, pg_rewind removes or modifies any data in Node A that was recorded after this point, effectively rolling back Node A's state to match the state of Node B at the time of the crash.

3. Reconfiguration: After pg_rewind completes its work, Node A's data is now consistent with the new primary, Node B. You can then configure Node A to act as a replica of Node B. Node A will start replicating from Node B, catching up with any changes that occurred after the failover.

- Final Cluster State

Now, Node A, which was the old primary, is operating as a replica, while Node B continues as the primary. Node C remains a replica as well. The cluster is back to a fully operational state, with all nodes synchronised and ready to handle any future failures.

In this example, pg_rewind was crucial in quickly bringing the old primary (Node A) back into the cluster without the need for a full backup and restore, saving significant time and minimising disruption.

3.1.4 - Backup

Backup in PostgreSQL is a critical part of maintaining data integrity and ensuring recovery in case of data loss or corruption. PostgreSQL offers several methods for creating backups, each suited to different needs, ranging from simple logical backups to more complex physical backups. File-Based log shipping is also a backup method.

- Logical Backup

Logical backups in PostgreSQL are created using the `pg_dump` and `pg_dumpall` utilities. These tools produce a dump of the database schema and data, which can be stored in a script file or an archive format.

1. pg_dump:
- This tool is used to back up a single PostgreSQL database. It creates a logical copy of the database in a format that can be restored using the `psql` utility.
- The dump file can be in plain text (a SQL script), custom format, directory format, or tar format.
- Logical backups created with pg_dump are useful for migrating databases, upgrading PostgreSQL versions, or simply creating portable backups that can be restored on any PostgreSQL instance.

2. pg_dumpall:
- Unlike pg_dump, pg_dumpall is used to back up all databases in a PostgreSQL cluster, including global objects like users and roles.
- It produces a single script file that can be used to recreate the entire cluster.

Logical backups are convenient and portable but can be slower for large databases because they involve exporting the entire database to a file.

- Physical Backup

Physical backups involve copying the actual files that PostgreSQL uses to store database data. This can be done using various methods, including:

1. Base Backup with pg_basebackup:

- Pg_basebackup (Section 3.1.4.2) is the most common way to create a physical backup of a PostgreSQL database cluster.
- It creates a consistent snapshot of the database by copying the data files directly from the file system.
- It can be combined with continuous WAL (Write-Ahead Logging) archiving to enable point-in-time recovery (PITR).
- Base backups are typically stored alongside archived WAL files to allow for restoration to any point in time after the base backup was taken.

2. Custom File System-Level Backup:

- This method involves manually copying the PostgreSQL data directory using tools like rsync or tar.
- To ensure a consistent backup, the database should be stopped, or a backup label file should be created using `pg_start_backup()` and `pg_stop_backup()` functions, which can also be used for backing up a live server without stopping it.

Physical backups are generally faster and more efficient for large databases because they copy the data directly, including indexes and other metadata. However, they are less portable and usually need to be restored on a server with the same configuration as the original.

3.1.4.1 - WAL Archiving and Point-In-Time Recovery (PITR)

To complement physical backups, PostgreSQL supports WAL archiving. WAL files contain a record of every change made to the database, and by archiving these logs, you can replay them after restoring a base backup to bring the database to a specific point in time.

- WAL Archiving: Configuring WAL archiving involves setting the archive_mode and archive_command parameters in postgresql.conf. The archive_command defines how WAL files are copied to an archive location.
- Point-In-Time Recovery (PITR After restoring a base backup, you can use the archived WAL files to roll forward the database to any point in time before an error or failure occurred. This is done by restoring the base backup, configuring the recovery.conf file, and applying the WAL logs.

Summary

Backup in PostgreSQL can be done using logical or physical methods, depending on the requirements. Logical backups (pg_dump and pg_dumpall) are great for portability and ease of restoration but may be slower for large databases. Physical backups, especially using pg_basebackup, are more efficient for large-scale data but require a more consistent environment for restoration. To achieve granular recovery, such as to a specific point in time, PostgreSQL uses WAL archiving in conjunction with physical backups. This comprehensive approach ensures that data can be recovered reliably in various scenarios, from individual data corruption to complete system failures.

3.1.4.2 - Pg_Basebackup Tool

pg_basebackup is a PostgreSQL tool used to create a full backup of a running database. Internally, it performs a copy of all necessary files to restore the database to a consistent state. When pg_basebackup is executed, it connects to the PostgreSQL server using a client connection. The process begins by requesting the server to create a checkpoint, marking the starting point for the backup in the write-ahead log (WAL) Section 3.1.2. This ensures that the backup process is consistent from the outset.

Once the checkpoint is created, pg_basebackup starts copying all the files from the server's or local data directory. This includes configuration files (such as postgresql.conf and pg_hba.conf), data files for the tables, ongoing WAL files, and various system and catalogue files necessary for the database's operation. Additionally, during the copy process, pg_basebackup ensures that all WAL files generated while the backup is being taken are included. This is typically managed using the -X option (e.g., -X stream), which streams the necessary WAL segments to the backup, ensuring that the backup is consistent and can be used for recovery. At the end of the copying process, pg_basebackup asks the server to finalise the backup. A backup_label file is then created in the backup directory. This file contains metadata about the backup, such as the start time, checkpoint ID, and the locations of the first and last WAL files needed for recovery. These details are crucial for restoring the database from the backup. The tool does not delete or modify any files on the source server. Instead, it only copies the necessary files to the specified backup location. All files in the data directory are copied as they are, ensuring that the backup is a complete and accurate snapshot of the database at the time the backup was taken. However, pg_basebackup may exclude temporary or intermediate files that are not required for recovery, such as files in pg_replslot or pg_stat_tmp, depending on the options used.

3.1.5 - Roles

In PostgreSQL, roles play a crucial part in managing the behaviour and responsibilities of each node within a cluster. Depending on the role assigned to a node, it can perform different functions, making it essential to understand the various roles available. Some of the common roles in PostgreSQL include primary, replica, replicator, publisher, and subscriber.

3.1.5.1 - Primary node

The primary node is a well-known role within the streaming replication method in PostgreSQL. This node is the main source of truth for data, responsible for storing the Write-Ahead Logs (WALs). These WALs contain a sequential record of all changes made to the database. The primary node sends these logs to all its replica nodes to ensure data consistency across the cluster. It handles all the write operations, which are then streamed to replica nodes in near real-time, allowing them to replicate the primary's data state.

The primary node ensures high availability and data durability, as it continuously ships the WALs to its replicas. In case of a failure on the primary node, one of the replicas can be promoted to the primary role to maintain service continuity.

3.1.5.2 - Replica node

The replica node is another critical role in PostgreSQL's replication setup. A replica node receives WALs from the primary node and applies them to maintain an up-to-date copy of the primary's data. Replica nodes are often used to distribute read loads, enhance performance, and provide failover capabilities in case the primary node becomes unavailable.

Replica nodes operate in a read-only mode by default, meaning they do not accept direct write operations from clients. Instead, all write requests are directed to the primary node, and the changes are propagated to the replicas via WALs. In a high-availability setup, if the primary node fails, a replica can be promoted to a primary role, ensuring continuous operation of the database system.

The replicator, unlike the replica, is used in File-log based replication where there is no primary per se, but the replicator node obtains the data through rsync or scp to be able to apply it later in its replica database.

3.1.5.3 - Publisher and Subscriber

Publisher and subscriber are roles used in PostgreSQL's logical replication mechanism. Unlike physical replication, which involves streaming WALs to create an exact binary copy of the primary node's data, logical replication allows more granular control over what data is replicated and how it is applied.

- Publisher: This role is assigned to the node that produces and provides the changes to be replicated. A publisher defines a set of tables whose data changes are to be replicated. It sends these changes to the subscriber nodes in the form of logical changesets, rather than raw WALs. This makes it possible to replicate specific parts of a database, rather than the whole database.
- Subscriber: This role is assigned to the node that consumes changes from a publisher. A subscriber connects to the publisher to receive data changes and applies them locally. This allows for flexible replication setups, including selective replication of tables, consolidating data from multiple publishers into one subscriber, or even replicating data across different PostgreSQL versions.

Logical replication using the publisher and subscriber roles is useful for scenarios where different parts of a database need to be synchronised, data needs to be shared between different databases, or when upgrading PostgreSQL versions without downtime.

Each role in PostgreSQL serves a distinct purpose, and understanding these roles is essential for setting up a reliable, high-performance, and scalable database environment.

3.1.6 - MultiNodes

This section explains some relevant information for creating multiple instances of Postgres or PgPool on the same machine without them colliding with each other. It should be noted that the ports cannot coincide, so it is important to modify the corresponding value in the configurations for each container.

For Postgres change the to a usable port number.

```
listen_addresses = '*'
                   # comma-separated list of addresses;
                   # defaults to 'localhost'; use '*' for all
                   # (change requires restart)
#port = 5432                # (change requires restart)
```

Additionally, you can modify the port using environment variables that postgres will use, in this case PGPORT. And remember when launching containers change the volume name for each instance.

```
# Instance 1
$ docker run -d \
       --name pgcdb1 \
       -e POSTGRES_PASSWORD=p_pgcloud \
       -e POSTGRES_USER=u_pgcloud \
       -e PGDATA=/opt/pgdata \
       -e PGPORT=5432 \
       -e POSTGRES_DB=db_pgcloud \
       -v pgcdata1:/opt/pgdata \
       postgres


# Instance 2
$ docker run -d \
       --name pgcdb2 \
       -e POSTGRES_PASSWORD=p_pgcloud \
       -e POSTGRES_USER=u_pgcloud \
       -e PGDATA=/opt/pgdata \
       -e PGPORT=5433 \
       -e POSTGRES_DB=db_pgcloud \
       -v pgcdata2:/opt/pgdata \
       postgres
```

3.1.7 - Replication slot

Replication slots are a crucial mechanism in PostgreSQL for streaming replication that ensure reliable and consistent replication, particularly in environments where replicas may temporarily go offline or fall behind in applying changes. By maintaining a replication slot on the primary server, PostgreSQL can manage WAL (Write-Ahead Log) retention and replication status tracking, helping to prevent data loss and ensuring replicas can catch up when they reconnect.

Key Features of Replication Slots

1. Preserving WAL Files:
   Replication slots help the primary server manage the retention of WAL files, which are essential for replication. In a typical replication scenario, once all replicas have acknowledged that they have applied a WAL file, the primary server deletes it to save disk space. However, if a replica is offline or lagging behind, it might not be able to acknowledge receipt of the WAL files in real-time. Without a replication slot, these WAL files could be prematurely deleted, causing the replica to lose data when it comes back online.

2. How Replication Slots Work:
   - Creation: When a replication slot is created on the primary server, it is given a unique name and is associated with a specific amount of reserved disk space to store WAL files.
   - Retention: The primary server uses this slot to retain WAL files until they are confirmed as applied by the replica associated with the slot. Even if the replica is offline or unable to keep up with the rate of changes, the primary server continues to retain the necessary WAL files in the replication slot.
   - Catch-Up: When the replica reconnects after being offline, it retrieves the WAL files stored in the slot to catch up with the changes made on the primary. This ensures data consistency and prevents any data loss.

3. Monitoring Replica Lag:
   Replication slots also enable the primary server to monitor the lag of each replica. By tracking how many WAL files are retained in a slot and how far behind the replica is, the primary can assess the synchronisation status of its replicas. This monitoring is essential for maintaining data integrity and identifying potential issues with replication performance.

4. Automatic Cleanup:
   PostgreSQL manages the disk space reserved for WAL files in replication slots by automatically retaining only the WAL files that have not yet been applied by the replicas. Once a replica catches up and confirms it has applied certain WAL files, PostgreSQL will automatically remove those files from the replication slot, freeing up space.

5. Safety Against Replication Failures:
   If a replica is down for an extended period and the primary server keeps accumulating WAL files in the replication slot, the disk space might eventually run out. In such cases, administrators must either increase disk space, remove old replication slots that are no longer needed, or resolve the replica's connectivity issues. However, the replication slot mechanism ensures that no WAL files needed for replicas are lost, providing a safeguard against replication failures.

Additional Considerations

- Logical vs. Physical Replication Slots:
  PostgreSQL supports both logical and physical replication slots. Physical replication slots are used for streaming physical changes (binary-level changes) to replicas, while logical replication slots are used for logical replication, which allows selective replication of specific tables or databases and supports replication across different PostgreSQL versions.

- Impact on Performance:
  While replication slots are vital for ensuring data consistency and reliability, they can impact performance if not managed properly. If a replica is offline for a prolonged period, the primary may accumulate a significant amount of WAL files, which could lead to increased disk usage and potentially degrade the performance of the primary server.

- Failover and Slots:
  In a high-availability setup, when a failover occurs and a replica is promoted to become the new primary, replication slots need to be recreated or managed accordingly. This is because replication slots are tied to the original primary node, and a new primary will not automatically have these slots unless explicitly configured.

Replication slots are a powerful feature in PostgreSQL that help manage and ensure the integrity of data replication across nodes. By preserving WAL files and monitoring replication lag, replication slots prevent data loss and facilitate seamless synchronisation of replicas, even when they experience temporary outages or lag behind. Proper management of replication slots is crucial for maintaining a healthy and performant PostgreSQL environment, especially in high-availability or disaster recovery scenarios.

3.2 - Load Balance Layer

The main process of the load balancing layer is managed by PgPool, which facilitates the efficient distribution of connections and statement execution across all available nodes in the system. This layer is crucial for optimising performance and preventing resource saturation, such as network bandwidth, memory, and processing cores. PgPool not only balances the load of incoming requests but also monitors the health of the nodes, ensuring that traffic is routed only to those that are available and functioning properly. This maximises resource utilisation and guarantees a high level of availability and stability in the system.

In addition, PgPool offers extra features such as connection pooling, which reduces the overhead caused by the constant creation and termination of connections, and query replication, allowing a query to be executed simultaneously on multiple nodes to improve read performance. With all these capabilities, PgPool becomes a key component in ensuring scalability and efficient management of distributed databases in high-demand environments.

We recommend you a lot to read the online official PgPool documentation, which it includes many well explained information: https://www.pgpool.net/docs/latest/en/html/

3.2.1 - Configuration

The configuration of PgPool can be splitted into 3 main parts, general part, postgres cluster part and pgpool cluster part.

3.2.1.1 - General Configuration

PgPool requires general configuration to function effectively as a load balancing layer, not only because it operates as a proxy or middleware for PostgreSQL, but also to optimise its performance. Proper configuration ensures efficient resource management, query distribution, and system reliability.

When setting up multiple PgPool nodes to form a PgPool cluster (commonly known as a Watchdog cluster, as detailed in Section 3.2.6), additional configuration is needed. This setup enables high availability, fault tolerance, and continuous service, ensuring that if one PgPool node fails, others can take over seamlessly, preventing downtime and maintaining the integrity of the database system.

The Watchdog component is particularly important because it monitors the health of each PgPool node and facilitates failover and recovery. Therefore, fine-tuning the PgPool configuration not only improves performance but also enhances the robustness and resilience of the entire PostgreSQL infrastructure.

This should be the configuration for every node, all the configuration can be find in /etc/pgpool2/pgpool.conf or /etc/pgpool-II/pgpool.conf:

```
# pgpool.conf

listen_addresses = '*'
port = 5434    # By default it is 9999

# Load balancing, when active it will distribute the load by statements
load_balance_mode = on
# Connection Pooling
# You can find more information in Section 3.2.7
num_init_children = 500
max_pool = 2

health_check_period = 10
health_check_timeout = 20

# You can find more information in Section 3.2.4
failover_command = '/usr/local/bin/failover.sh %d %P %h %H'

# When enabled PgPool will failback and attach the node to the system auto
auto_failback = on
auto_failback_interval = 5

# To check streaming replication of postgres in every node
sr_check_period = 3
sr_check_user = 'sr_pgcloud'
sr_check_password = 'p_pgcloud'
sr_check_database = 'db_pgcloud'

# Write some ip from a non-diedable server, can be private or public
trusted_servers = '10.0.0.1,10.0.0.2'
```

Go to Section 3.2.7 If you want to know more about how to set the values of max_pool and num_init_children. And you can find more in:
https://www.pgpool.net/docs/43/en/html/runtime-config-connection-pooling.html

Go to Section 3.2.4 To find how the failover and failback works.

3.2.1.2 - Backend Configuration

For each PostgreSQL node, you need to configure it in the pgpool.conf file. If one of the nodes is not properly configured in this file, the system will not consider it as part of the replication cluster.

Example of a PostgreSQL Node Configuration in pgpool.conf

In the pgpool.conf file, each backend (PostgreSQL node) must be explicitly defined. The following is an example of how to configure two nodes in the pgpool.conf file:

```
# Backend 0 (Primary node) Zero always is going to be primary
backend_hostname0 = '10.0.0.1'   # IP address or hostname of the primary node
backend_port0 = 5432                # PostgreSQL port number for this node
backend_weight0 = 1                 # Weight for load balancing
backend_flag0 = 'ALLOW_TO_FAILOVER'  # This flag determines if failover is allowed for
this node

# Backend 1 (Replica node)
backend_hostname1 = '10.0.0.2'   # IP address or hostname of the replica node
backend_port1 = 5432                # PostgreSQL port number for this node
backend_weight1 = 1                 # Weight for load balancing
backend_flag1 = 'ALLOW_TO_FAILOVER'  # This flag allows failover for this replica node
```

Explanation of Configuration Parameters:

1. backend_hostnameX: This specifies the IP address or hostname of the PostgreSQL node. Each node (backend) needs a unique identifier (X), starting from 0 for the primary node and incrementing for each additional node (e.g., backend_hostname0, backend_hostname1).
    - Example: backend_hostname0 = '10.0.0.1'
      Here, the primary PostgreSQL node has the IP address 10.0.0.1 from private ip range.

2. backend_portX: This defines the port number on which the PostgreSQL server is listening. The default port for PostgreSQL is 5432, but this can be customised depending on your setup.
    - Example: backend_port0 = 5432
      The primary node in this configuration listens on port 5432.

3. backend_weightX: The weight determines how much traffic or load is directed to this particular node during load balancing. Nodes with higher weights will receive more queries.
    - Example: backend_weight0 = 1, backend_weight1 = 1
      In this setup, both the primary and replica nodes have equal weights, meaning they will share the load equally.

4. backend_flagX: This setting controls specific behaviour for the node. The most common value is 'ALLOW_TO_FAILOVER', which enables automatic failover if the node becomes unavailable. There are other possible values, but this is the standard configuration for high availability setups.
    - Example: backend_flag0 = 'ALLOW_TO_FAILOVER'
      This allows the system to fail over to another node if this one fails.

Key Points to Remember:

1. Node Identification: Each node is identified by a number (e.g., backend_hostname0 for the primary, backend_hostname1 for the first replica). The numbering must be continuous and consistent for all configuration parameters (backend_portX, backend_weightX, etc.).

2. Replication and Failover: If any node is not configured correctly in pgpool.conf, that node will not be considered part of the replication cluster, and the system won't fail over to it in the event of a failure. It's crucial to ensure that all active nodes are included in this configuration file.

3. Weighting and Load Balancing: By adjusting the backend_weightX, you can control how much traffic each node handles. For example, you might want to assign more load to a higher-performance server by giving it a higher weight.

3.2.1.3 - Watchdog Configuration

Watchdog is a feature in Pgpool-II that provides high availability by monitoring Pgpool instances and ensuring that, in the event of a failure of the primary Pgpool node, another node will take over. This feature is essential in preventing single points of failure and maintaining a high-availability system. (Section 3.2.6)

Here's a detailed explanation of how to configure Watchdog in the pgpool.conf file, along with examples and explanations of the parameters involved.

```
# Enable watchdog, do not enabled it if you only use one pgpool
use_watchdog = on
# Delegate and VIP to a pgpool, pgcloud are not going to use it. This system allow
multiple operation between different pgpools
delegate_ip = '192.168.100.50'

# As same as postgres servers
hostname0 = '10.0.0.3'
wd_port0 = 9000
pgpool_port0 = 5434

hostname1 = '10.0.0.4'
wd_port1 = 9000
pgpool_port1 = 5434

# To check the pgpool alive status
wd_lifecheck_method = 'heartbeat'
wd_interval = 3

# As same as postgres and watchdog
heartbeat_hostname0 = '10.0.0.3'
heartbeat_port0 = 9694
heartbeat_device0 = ''
heartbeat_hostname1 = '10.0.0.4'
heartbeat_port1 = 9694
heartbeat_device1 = ''

wd_heartbeat_keepalive = 2
wd_heartbeat_deadtime = 30
```

As you can see the configuration of PgPool cluster is the same as Postgres cluster, but now the problem is PgPool how will it know which watchdog configuration to use.

Pgpool-II 4.1 or earlier, because it is required to specify its own pgpool node information and the destination pgpool nodes information, the settings are different per pgpool node (Section 3.2.6.4). Since Pgpool-II 4.2, all configuration parameters are identical on all hosts. If a watchdog feature is enabled, to distinguish which host is which, a pgpool_node_id file is required. You need to create a pgpool_node_id file and specify the pgpool (watchdog) node number (e.g. 0, 1, 2 ...) to identify the pgpool (watchdog) host.

If you have 3 pgpool nodes with hostname server1, server2 and server3, create the pgpool_node_id file on each host as follows. When installing Pgpool-II using RPM, pgpool.confis installed under /etc/pgpool-II/.

server1

```
[server1]# cat /etc/pgpool-II/pgpool_node_id
0
```

server2

```
[server2]# cat /etc/pgpool-II/pgpool_node_id
1
```

-

server3 and more ...

```
[server3]# cat /etc/pgpool-II/pgpool_node_id
2
```

And also if you configure Virtual IP, you must specify how to up or down the IP link

```
if_up_cmd = '/usr/bin/sudo /sbin/ip addr add $_IP_$/24 dev enp0s8 label enp0s8:0'
if_down_cmd = '/usr/bin/sudo /sbin/ip addr del $_IP_$/24 dev enp0s8'
arping_cmd = '/usr/bin/sudo /usr/sbin/arping -U $_IP_$ -w 1 -I enp0s8'
```

3.2.2 - PgPool PCP

PgPool-II's PCP (PgPool Control Protocol) is a management tool used to interact with PgPool-II, a middleware that sits between PostgreSQL clients and PostgreSQL servers. PgPool-II provides several features like connection pooling, load balancing, and replication. PCP, in particular, is a utility to manage and monitor PgPool-II's internal state and configurations. It allows system administrators to perform certain tasks, such as attaching or detaching PostgreSQL nodes from the cluster, checking the health of nodes, or viewing connection statistics.

3.2.2.1 - What is PCP Used For?

PCP provides a command-line interface (CLI) and a protocol to manage PgPool-II remotely. It is primarily used for tasks such as:

1. Managing Nodes: Add, remove, or list the status of PostgreSQL backend nodes that PgPool-II is managing.
2. Health Checks: Check the health and status of PostgreSQL servers, allowing you to identify failed nodes and take corrective action.
3. Connection Management: View or terminate backend connections or pool sessions.
4. Failover/Recovery: Trigger failover or recovery mechanisms to ensure high availability.
5. System Monitoring: Retrieve statistics on connections, session information, and other monitoring data from the PgPool-II process.

3.2.2.2 - How PCP Works

PCP communicates with the PgPool-II process over a TCP/IP connection using a specified port. It allows administrators to run commands from a different machine or over the network.

There are two primary ways to interact with PCP:

1. PCP Command-Line Utilities: PgPool-II provides a set of utilities that allow you to issue PCP commands from the terminal.
2. PCP Client Library: For more programmatic use, there's a C-based client library that allows you to integrate PCP commands into scripts or custom applications.

3.2.2.3 - How to Use PCP

3.2.2.3.1. Configuration

To use PCP, it needs to be enabled in your PgPool-II configuration (pgpool.conf). You have to configure:

- PCP port (pcp_port): This is the port number PgPool-II will listen on for PCP commands.
- PCP socket directory: Location where the PCP socket will be created.
- PCP users and passwords: Configured in pcp.conf. These are the credentials used for authentication when issuing PCP commands.

```
# pgpool.conf
pcp_port = 9898
```

And configure pcp.conf

```
u_pgcloud:md5_password_hash
```

You can hash your password with the command pg_md5 offered by PgPool. More info: https://www.pgpool.net/docs/latest/en/html/pg-md5.html

3.2.2.3.2. Using PCP Command Line Tools

There are several PCP commands that are useful for managing PgPool-II. Some of the common ones are:

pcp_node_info: Retrieve information about a specific PostgreSQL backend node.

```
pcp_node_info -h localhost -U u_pgcloud -n 0 -p 9898
```

- This command retrieves the status of node 0.

pcp_attach_node: Attach a PostgreSQL backend node that was previously detached.

```
pcp_attach_node -h localhost -U u_pgcloud -n 0 -p 9898
```

- This attaches node 0 to PgPool-II.

pcp_detach_node: Detach a backend node, marking it as inactive.

```
pcp_detach_node -h localhost -U u_pgcloud -n 0 -p 9898
```

- This detaches node 0.

pcp_stop_pgpool: Stop the PgPool-II service gracefully or forcefully.

```
pcp_stop_pgpool -h localhost -U u_pgcloud -m fast -p 9898
```

- This will stop PgPool-II using the "fast" shutdown mode.

3.2.3 - Health Check

PgPool-II's health check is a crucial feature designed to continuously monitor the status and availability of PostgreSQL backend nodes. The health check feature ensures that PgPool-II can detect if a node becomes unresponsive or fails and then take appropriate actions, such as removing the failed node from the pool of available nodes, thus ensuring that traffic is routed only to healthy nodes.

PgPool-II periodically sends a simple query (usually a SELECT 1) to each PostgreSQL backend node. This is a lightweight query used to verify that the database server is online and functioning. If the node responds successfully, it is marked as healthy and remains in the pool of available nodes. If the node fails to respond within a defined timeout period or returns an

error, PgPool-II will mark the node as "down" and take further actions like failover or detaching the node from the cluster.

Health checks are customizable via several parameters in the pgpool.conf file. Some important configuration options include:

- health_check_period: Specifies the time interval (in seconds) between two health check requests to a node. If set to 0, health checking is disabled.

```
health_check_period = 30  # Health check every 30 seconds
```

- health_check_timeout: Sets the time (in seconds) that PgPool-II waits for a response from a node before considering it unresponsive.

```
health_check_timeout = 20  # 20 seconds to wait for response
```

- health_check_user: The PostgreSQL user that PgPool-II uses to perform health checks. This user needs to have minimal read permissions to execute the health check query.

```
health_check_user = 'u_pgcloud'
```

- health_check_password: Password for the health_check_user.

```
health_check_password = 'p_pgcloud'
```

- health_check_database: The database used for the health check query. It could be any small database that exists on all nodes.

```
health_check_database = 'db_pgcloud'
```

- health_check_max_retries: Number of consecutive failed health checks after which PgPool-II will consider a node down and remove it from the pool.

```
health_check_max_retries = 3
```

- health_check_retry_delay: The delay (in seconds) between retry attempts after a failed health check.

```
health_check_retry_delay = 5  # Retry health check every 5 seconds after a failure
```

Health Check Process

1. Health Check Query: At each health check interval (health_check_period), PgPool-II sends a simple query (like SELECT 1) to each PostgreSQL node using the health_check_user credentials.
2. Response Validation: PgPool-II waits for the node to respond within the health_check_timeout period. If the node responds correctly, it is considered healthy.
3. Failure Detection: If a node fails to respond or returns an error (e.g., due to a network failure, system crash, or database service being down), PgPool-II retries the check up to health_check_max_retries times with delays of health_check_retry_delay seconds between each attempt.
4. Node Removal: If the node fails the health check continuously for the number of retries specified, PgPool-II marks the node as down and removes it from the pool of available backend nodes. This ensures that no further queries are sent to the failed node.
5. Failover: If PgPool-II is configured for automatic failover, the failed node's workload can be shifted to a standby node.

What Happens After a Node is Marked as Down?

When a node fails the health check and is marked as down, PgPool-II takes action to ensure the continued availability of the service:

- Connection Redirection: Client connections are routed to the remaining healthy nodes.
- Failover: If failover is enabled, PgPool-II may promote a standby node to become the primary node to replace the failed primary node.
- Recovery: Once the failed node is repaired or becomes available again, it can be reattached to the cluster using PgPool-II's pcp_attach_node command, or PgPool-II may automatically detect that the node is back online (if configured).

3.2.4 - Failover and Failback

Failover and Failback are two ways or mechanisms in PgPool-II to recover a lost node, designed to ensure high availability and disaster recovery in PostgreSQL clusters. These processes manage what happens when a database node fails and how to bring it back online once it's repaired or restored.

3.2.4.1 - Failover

Failover refers to the automatic or manual process by which PgPool-II shifts database operations from a failed primary node to a healthy standby node. This ensures that database services remain available despite node failures.

How Failover Works

1. Node Failure Detection: PgPool-II uses health checks to continuously monitor the status of PostgreSQL nodes. If a primary node (master) fails or becomes unresponsive, PgPool-II detects this via the health check process.
2. Failover Trigger: When PgPool-II determines that the primary node has failed, the failover process is initiated. PgPool-II can be configured to trigger failover automatically, or it can be done manually by an administrator.
3. Promoting a Standby Node: PgPool-II promotes one of the standby nodes (replicas) to become the new primary (master) node. This process usually involves:
    - Executing a promotion command that promotes the standby to primary using PostgreSQL's `pg_ctl promote` or similar mechanisms.
    - Redirecting client read/write queries to the newly promoted primary.
4. Client Connection Management: PgPool-II updates its internal configuration to route client connections to the new primary node. For applications using PgPool-II, this failover is generally transparent, ensuring continued operation with minimal disruption.
5. Notification: PgPool-II can notify administrators about the failover event via email or logs, allowing them to take further actions if necessary.

Example Failover Configuration in pgpool.conf

In pgpool.conf, several settings control failover behaviour:

- failover_command: Defines the command to execute during a failover. This command is responsible for promoting a standby node to the primary role.

```
failover_command = '/usr/local/bin/failover.sh %d %H %P %R'
```

    - %d: The node ID of the failed node.
    - %H: The hostname of the failed node.
    - %P: The path of the PgPool-II configuration file.
    - %R: The ID of the standby node that will be promoted.

- failover_on_backend_error: Determines whether PgPool-II should automatically trigger failover when a backend node (i.e., primary or standby) fails.

```
failover_on_backend_error = on
```

- enable_pool_hba: If you want to control connections with client-side failover, this can be combined with authentication controls like pg_hba.conf.

Failover Use Case

Imagine a scenario where a primary PostgreSQL node (node0) suddenly crashes:

- PgPool-II detects the failure through a health check.
- PgPool-II triggers the failover process, promoting a standby node (node1) to the new primary role.
- Client applications continue to send queries through PgPool-II, which now routes them to node1.
- The database remains available with minimal downtime, thanks to automatic failover.

Example of failover script.

This is an example following `'/usr/local/bin/failover.sh %d %H %P %R'` :

```
#!/bin/bash

# Failover parameters passed from PgPool-II
FAILED_NODE_ID=$1
FAILED_NODE_HOST=$2
NEW_PRIMARY_HOST=$3
NEW_PRIMARY_PORT=$4

echod() {
     echo "$1" >> ./failover.log
}
# Log the failover event
echod "Failover triggered due to failure of node $FAILED_NODE_ID ($FAILED_NODE_HOST)"
# Promote the standby node to primary
echod "Promoting new primary node: $NEW_PRIMARY_HOST:$NEW_PRIMARY_PORT"
ssh ssh_user@$NEW_PRIMARY_HOST "pg_ctl promote -D /var/lib/postgresql/data"

# Log success or failure
if [ $? -eq 0 ]; then
    echod "Node $NEW_PRIMARY_HOST successfully promoted to primary."
else
    echod "Failed to promote node $NEW_PRIMARY_HOST to primary."
    exit 1
fi

exit 0
```

3.2.4.2 - Failback

Failback refers to the process of restoring a previously failed node to the cluster and resuming its original role, typically as a standby node. In some cases, the failed node may need to be brought back as the primary node, depending on the configuration.

How Failback Works

1. Node Restoration: After the failed node is repaired or restarted, it must be re-synchronized with the current primary node. This usually involves catching up with any missing transaction logs (WAL files) and becoming a replica again.
2. Reintegration into the Cluster: Once the node is back online and synchronised, it can be re-added to the pool of available nodes by PgPool-II. This can be done manually using the PCP tool (pcp_attach_node), or automatically if PgPool-II is configured for auto-recovery.
3. Optional Role Reversal: In some scenarios, administrators may prefer to return the original node to its primary role (failback). This requires promoting the previously failed node back to primary status and demoting the current primary to a standby node. However, this is usually a manual process to avoid frequent switching.

3.2.5 - Replication Delay

Replication delay in a PostgreSQL replication setup refers to the time difference between when a transaction is committed on the primary node and when it is applied on the replica node(s). This delay can happen for various reasons such as network latency, the workload on the replica, or hardware limitations.

3.2.5.1 How PgPool Detects Replication Delay

PgPool-II, when used in conjunction with a PostgreSQL replication setup, has mechanisms to monitor replication delay. It does this by using a feature called pg_stat_replication on the primary node, which provides information about the replication lag of each standby. PgPool queries this PostgreSQL system view to track replication lag by comparing the transaction log positions (lsn, or Log Sequence Number) between the primary and each replica.

Key parameters in PgPool related to replication delay detection include:

- health_check_replication_delay: This option checks the replication delay for the backends periodically.
- replication_mode: PgPool-II can operate in replication mode to send read queries to replicas.
- delay_threshold: This setting allows you to specify the maximum allowed replication delay (in bytes or time). If the delay exceeds this threshold, PgPool can mark the node as unsuitable for query routing.

What Happens When a Node Has a High Replication Delay?
If a replica node exhibits a significant replication delay, PgPool has mechanisms to handle it depending on configuration settings:

1. Query Routing Behaviour:

- If a replica's replication delay exceeds the defined threshold (delay_threshold), PgPool can stop sending read queries to that replica. This ensures that stale data is not served to clients.
- In a scenario where the delay threshold is not configured or is set to a very high value, PgPool might continue routing read queries to the delayed node, risking inconsistent query results.

2. Failover Handling:

- If the delay becomes critical or the node becomes unreachable, PgPool may initiate a failover event, promoting another node if necessary (depending on how the failover mechanism is configured).
- If the node is still reachable but severely delayed, PgPool might log warnings and stop sending traffic to that node until its delay drops below the threshold.

3. Load Balancing:

- PgPool-II uses its load balancing feature to distribute read queries across replicas. However, when a node is delayed, PgPool can exclude that node from the load balancing pool. This prevents the load balancer from sending queries to a node that is lagging behind the primary.
- Write queries, which must always go to the primary node, are not affected by replication delay directly but can lead to higher delays on the replicas if the primary is heavily loaded.

3.2.5.2 - Configuring PgPool for Replication Delay Management

To ensure PgPool handles replication delay appropriately, it's important to configure the following settings:

1. delay_threshold: This specifies the maximum replication delay allowed before PgPool stops routing queries to the delayed replica. It can be configured in either bytes or time.

2. check_replication_time_interval: This parameter controls how often PgPool checks the replication delay of backends.

3. failover_on_backend_error: Determines if PgPool should trigger a failover when replication delay or other errors occur on a backend node.

By using these configurations, PgPool can effectively manage replication delay and ensure that queries are routed to up-to-date replicas, maintaining data consistency and reliability.

3.2.6 - Watchdog

The Watchdog feature of PgPool-II is designed to enhance the high availability (HA) of PostgreSQL clusters by managing multiple PgPool-II instances. It monitors the health of PgPool-II nodes and automatically handles failovers, ensuring that the database cluster can recover from node failures without manual intervention.

3.2.6.1 - History of PgPool Watchdog

The Watchdog feature was introduced in PgPool-II to address the limitations of single-node deployments. Earlier versions of PgPool-II only supported failover at the database node level (such as promoting a standby to primary), but this setup had a single point of failure in PgPool itself. To resolve this, Watchdog was introduced to provide failover and HA for PgPool-II itself. This means that if one PgPool-II instance fails, another PgPool-II instance can take over, ensuring continuous operation of the cluster.

3.2.6.2 - Functionality and How Watchdog Works

The Watchdog feature allows multiple PgPool-II instances to work together in a cluster, ensuring failover at both the database and PgPool levels. Here's how Watchdog achieves high availability:

1. PgPool-II Clustering:

Watchdog forms a PgPool-II cluster by coordinating multiple PgPool-II nodes. These nodes communicate with each other and ensure that only one node is active (called the Leader) at any given time, which handles the client requests. The remaining nodes are in a Standby state and are ready to take over if the leader fails.

2. Health Monitoring:

Each PgPool-II node monitors the health of the other PgPool-II nodes and the database backends. If the leader node or a database node becomes unresponsive or fails, the Watchdog triggers a failover event.

3. Leader Election:

Watchdog employs a consensus-based leader election mechanism, ensuring that when the current leader PgPool-II instance fails, a new leader is elected automatically. This leader manages failover decisions, health checks, and query routing.

4. Failover and Failback:

- Failover: When Watchdog detects a failure in a PgPool-II node (e.g., the leader), it automatically promotes a standby node to be the new leader.
- Failback: When the failed node recovers, it rejoins the cluster, but only as a standby unless the new leader fails.

5. Split-Brain Avoidance:

In a multi-node system, there's a risk that a network partition or connectivity issue might lead to a "split-brain" situation, where two PgPool-II nodes incorrectly think they are both leaders. Watchdog uses quorum-based decision making and external entities like a Virtual IP (VIP) and shared communication interfaces to avoid this situation. Only one leader node controls the VIP, preventing split-brain scenarios.

3.2.6.3 - Key Features of PgPool Watchdog

1. Virtual IP (VIP):

Watchdog supports the use of a Virtual IP address. The VIP is always assigned to the leader node, and client applications connect to the PgPool-II cluster via this IP. When a leader node fails, the VIP is moved to the newly promoted leader node, ensuring seamless client connectivity.

2. LifeCheck:

Watchdog uses the LifeCheck feature to monitor the health of PgPool-II nodes. It can perform checks such as pinging other nodes to verify that they are reachable. If a node becomes unresponsive, Watchdog removes it from the cluster.

3. Arbitration and Quorum:

Watchdog uses a quorum mechanism to make decisions regarding failover and leader election. This avoids the problem of nodes independently taking actions that might conflict with the overall cluster's state.

4. Integration with Failover Scripts:

Watchdog can integrate with custom failover scripts, allowing you to define specific actions that should be performed during failover, such as notifying admins or integrating with external systems.

5. Database Health Checks:

Watchdog can check the health of the PostgreSQL backend nodes and trigger failovers when necessary. If the primary node goes down, PgPool-II can promote a standby to primary.

3.2.6.4 - Setting Up PgPool-II Watchdog (Alternative method)

The configuration below can only be used for Pgpool-II version 4.1 or earlier. For later version, please visit Section 3.2.1.3

To set up PgPool-II with Watchdog, you need to configure multiple PgPool-II instances to communicate and coordinate with each other. Here is a high-level overview of how to configure Watchdog:

In each PgPool-II node, you need to enable and configure the Watchdog settings in the pgpool.conf file.

Here are key parameters to set up:

```
# Enable watchdog
use_watchdog = on

# Define the VIP address to be used by the leader
delegate_IP = '192.168.1.100'  # Virtual IP address shared by PgPool-II nodes

# Heartbeat communication settings
heartbeat_destination0 = '192.168.1.101'  # Address of node 1
heartbeat_destination1 = '192.168.1.102'  # Address of node 2

# Watchdog communication settings
wd_hostname = 'pgpool-node1'  # Hostname of this PgPool-II node
wd_port = 9000  # Port for Watchdog communication

# Other Watchdog nodes in the cluster
other_pgpool_hostname0 = 'pgpool-node2'
other_pgpool_port0 = 9000
```

2.2.6.5 - Virtual IP Configuration

For the VIP setup, you can configure your system's network interface settings to allow the leader PgPool-II node to claim the VIP. Watchdog will automatically manage this IP address during failover. In most cases, you'll need to configure this in the pgpool.conf file and ensure that you have root or sudo access for the PgPool process to manage the VIP.

2.2.6.6 - Starting the PgPool-II Cluster

Once all PgPool-II nodes are configured with Watchdog, start the PgPool-II service on each node. The nodes will automatically elect a leader, and the VIP will be assigned to that node.

2.2.6.6.1 - Watchdog Failover Flow

Here's how a typical failover scenario works with Watchdog:

1. Primary PgPool-II Failure: If the leader PgPool-II node fails, Watchdog automatically detects the failure using its LifeCheck mechanism.
2. Leader Election: Watchdog triggers a leader election among the remaining PgPool-II nodes, and the new leader takes control.
3. VIP Assignment: The Virtual IP (VIP) is moved to the newly elected leader node, ensuring that client applications continue to connect seamlessly.
4. Database Failover: If the failure involves the PostgreSQL primary node, PgPool-II can trigger database failover as well, promoting a standby to the primary role.
5. Recovery: Once the failed node recovers, it rejoins the PgPool-II cluster as a standby.

2.2.6.6.2 - Additional Considerations

- Split-Brain: The quorum-based approach in Watchdog is critical to prevent a split-brain situation where multiple nodes mistakenly believe they are the leader.
- Scalability: Watchdog supports scaling PgPool-II nodes horizontally. Multiple PgPool-II instances can be added to increase HA and load distribution.
- Monitoring and Logging: Watchdog logs detailed information about health checks, failovers, and leader elections. Monitoring these logs can be critical for maintaining a healthy cluster.

2.2.6.7 - Conclusion

PgPool-II's Watchdog feature is a robust solution for achieving high availability and failover for both the PgPool-II nodes and PostgreSQL backends. By configuring multiple PgPool-II nodes in a cluster and utilising features like Virtual IPs, quorum-based leader election, and health monitoring, Watchdog ensures continuous operation, automatic failover, and prevention of split-brain scenarios. It's an essential component for any high-availability PostgreSQL architecture.

3.2.7 - Initial Children and Pooling Algorithm

PgPool-II is designed to manage connection pooling for PostgreSQL databases, and it achieves this by maintaining a pool of backend connections. To optimise resource usage, it allows you to control the number of initial backend connections (children processes) and how connections are managed and reused. Understanding how the Initial Children and Max Pooling Algorithm work is essential for effectively managing database connections in high-traffic environments.

Initial Children (num_init_children)

- Definition: The num_init_children parameter defines the number of child processes (worker processes) that PgPool-II spawns when it starts. These child processes are responsible for handling client connections to the database.

- How It Works:
    - When a client connects to PgPool-II, it assigns the connection to one of the pre-existing child processes.

    - Each child process can handle multiple client connections, depending on how the connection pooling is configured (i.e., the max number of connections that can be handled per child process).

    - If the number of active clients exceeds the number of initial children, PgPool-II will dynamically spawn more children (within limits set by the configuration) to handle the load.

    - The idea behind this parameter is to have a number of child processes ready to go when the system starts, which can minimise connection latency for incoming requests.

- Key Points:
    - num_init_children controls the number of pre-forked child processes.

- The default value might be around 32, but this should be adjusted based on your expected workload.

- Having too few initial children can lead to slower handling of client connections, while too many can lead to high resource consumption (memory and CPU).

3.2.7.1 - Max Pooling Algorithm and Connection Management

PgPool-II's connection pooling mechanism allows multiple client connections to share a smaller number of database connections. This can significantly reduce the overhead of repeatedly opening and closing database connections.

The key components related to the max pooling algorithm and connections are:

1. Max Connections in PostgreSQL (`max_connections`)

The PostgreSQL setting max_connections controls the maximum number of database connections that the PostgreSQL server can handle at any given time. Each PgPool-II child process maintains backend connections to PostgreSQL, and if too many child processes create connections, you could exceed the PostgreSQL max_connections limit.

Example:
If PostgreSQL's max_connections is set to 100, and PgPool-II has 32 child processes, each handling up to 4 connections, you could potentially exhaust the max_connections value, leading to connection failures.

2. max_pool Configuration in PgPool-II

PgPool-II uses the `max_pool` parameter to define how many backend database connections each child process can maintain for reuse. This is part of its max pooling algorithm, which determines how client connections are mapped to database connections.

- max_pool defines the number of connection pools per child process for a specific user/database combination.
- For each unique combination of username and database name, PgPool-II maintains a separate connection pool.
- Each pool can manage multiple client connections to reuse database connections efficiently.

Example of Pooling Mechanism:

- If you set max_pool = 4, it means each child process can maintain up to 4 connections to the database backend for every unique (user, database) pair.

- For example:
    - User A connects to Database X → 4 connection pools are created for this combination.

    - User B connects to Database Y → Another 4 connection pools are created.

    - If 5 clients using the same user/database combination connect, the 5th client might reuse one of the 4 connection pools (depending on whether the connections are idle or not).

This pooling mechanism helps avoid exhausting PostgreSQL's max_connections and reduces the overhead associated with repeatedly creating and closing database connections.

3.2.7.2 - Formula for Managing Connections

The total number of backend connections created by PgPool-II is influenced by a combination of several parameters:

- num_init_children: Number of PgPool-II child processes (initially started workers).
- max_pool: Maximum number of connection pools per user/database combination.
- PostgreSQL max_connections: Limit set on the PostgreSQL server for maximum allowed concurrent connections.

The formula to estimate the maximum number of database connections is (Continue reading for specific formula):

```
Max Database Connections = num_init_children * max_pool * (number of unique user-database pair)
```

For example:
- num_init_children = 32
- max_pool = 4
- Assume 2 unique (user, database) pairs.

This would result in:
Max Database Connections = 32 * 4 * 2 = 256

This means PgPool-II could potentially create up to 256 connections to the PostgreSQL backend under these conditions.

But a lot of times we have only one pair of user-database connections. And rare architectures need to satisfy different pairings, if more pairing is needed, different PostgreSQL database creation will satisfy this disadvantage. The more pairing you have the less connection you remain left to establish client connections.

With the condition of one pair connection the final formula will be:

```
Max Database Connections >= num_init_children * max_pool
```

PgCloud has been configured with 665 children and 3 pools in each child (You can change it later, by reading the manual you will know all the parameters). That means that PostgreSQL must be configured with 2000 max_connections, less configured connections will block PgPool and on many occasions PgPool will consider this event as node failure and will do a failover.

3.2.7.3 - Connection Pooling and Reuse

When a client disconnects, PgPool-II does not immediately close the associated backend connection. Instead, it leaves the connection open in the pool for future use. If another client connects and requires the same user/database combination, it can reuse this existing connection, thus saving the overhead of re-establishing the connection to the PostgreSQL server.

- If all backend connections for a particular user/database pair are busy, PgPool-II will wait for one to become available.

- If none become available within the configured timeout period, PgPool-II will either reject new connections or spawn additional child processes (within the limits set by the configuration).

3.2.7.4 - Connection Limits and PgPool-II Behavior

1. Exceeding PostgreSQL max_connections:

If PgPool-II attempts to open more database connections than the PostgreSQL max_connections allows, the PostgreSQL server will reject these connections. It's crucial to set the PgPool-II parameters (`num_init_children` and `max_pool`) in a way that does not exceed the PostgreSQL connection limit.

2. Balancing Connections:

PgPool-II manages connections intelligently, balancing client connections across the available pools and backend connections. It ensures that the connection limits defined in both PgPool-II and PostgreSQL are not exceeded unless explicitly allowed by the configuration.

3.2.7.5 - Best Practices for Configuring PgPool-II and PostgreSQL Connections

1. Set num_init_children Based on Expected Client Load:

Start with a reasonable number of child processes that can handle your typical load and adjust based on monitoring. Too few will result in slower handling of connections, while too many will use unnecessary resources.

2. Maximise Pool Efficiency (`max_pool`):

Set `max_pool` to optimise connection reuse for different user/database combinations. A higher value reduces the need to create new backend connections but increases memory usage.

3. Ensure PostgreSQL `max_connections` is Sufficient:

PostgreSQL's max_connections should be set higher than what PgPool-II will attempt to use, calculated using the formula:

```
max_connections ≥ num_init_children * max_pool
```

4. Use Connection Timeouts:

Set reasonable timeouts for idle clients and backend connections to prevent resource exhaustion.

3.3 - Connection Pooling Layer

This layer is managed by PGBouncer. PgBouncer is a lightweight connection pooler for PostgreSQL. It's designed to reduce the overhead associated with opening new connections to the database. PostgreSQL, by default, is a process-based server, meaning that every client connection is handled by a new process. This can be expensive in terms of memory and CPU usage, especially in environments with a high number of concurrent connections.

It can be used directly in PostgreSQL or can jump the pooling with PgPool, for more information on how to select your architecture, read section 2.

3.3.1 - Key Features of PgBouncer

- Connection Pooling: PgBouncer manages a pool of database connections and reuses them, reducing the overhead of establishing connections repeatedly.
- Lightweight: It has a very low memory footprint, making it ideal for environments where resources are constrained.
- Fast Switching: It can quickly switch between different databases, even within the same pool, without the need to establish a new connection each time.
- User Authentication: PgBouncer supports various authentication methods, including PostgreSQL native, MD5, and others.
- Monitoring: PgBouncer provides several internal tables to monitor its activity and performance.

3.3.2 - PgBouncer Modes

PgBouncer can operate in three different pooling modes:

1. Session Pooling (default): A server connection is assigned to a client for the duration of the client session.
2. Transaction Pooling: A server connection is assigned to a client only for the duration of a transaction.
3. Statement Pooling: A server connection is assigned to a client only for the duration of a single statement. This mode has the highest efficiency.

3.3.3 - Configuration

The main configuration file for PgBouncer is usually located at /etc/pgbouncer/pgbouncer.ini.

Here's an example configuration for a setup with three PostgreSQL nodes:

```
[databases]
; Define the database and connection strings for the three nodes
db_pgcloud= host=10.0.0.1,10.0.0.2,10.0.0.3 port=5432 dbname=db_pgcloud

[pgbouncer]
; Basic settings
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid

; Listen for incoming connections on this address and port
listen_addr = *
listen_port = 6432

; Pooling mode: can be session, transaction, or statement
pool_mode = transaction

; Set the maximum number of client connections
max_client_conn = 100

; Set the minimum number of server connections to maintain in the pool
min_pool_size = 10

; Set the maximum number of server connections in the pool
default_pool_size = 20

; Authentication settings
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
```

```
; SSL settings (optional, but recommended for production)
; If your PostgreSQL nodes are configured to use SSL, configure PgBouncer to do so as
well.
; server_tls_sslmode = require
; server_tls_ca_file = /etc/pgbouncer/ca.pem
; server_tls_cert_file = /etc/pgbouncer/pgbouncer.pem
; server_tls_key_file = /etc/pgbouncer/pgbouncer.key

[users]
; This section is optional. Here you can define specific settings for individual users.

[pgbouncer_stats]
; Set up a database to monitor PgBouncer statistics
stats_users = admin
```

User Authentication

Create a userlist file (/etc/pgbouncer/userlist.txt) to define users who can authenticate against PgBouncer:

```
"u_pgcloud" "md5c4a4e1b77a9e21d5938de0869bcd2c3a"
```

In this example, "myuser" is the username, and "md5c4a4e1b77a9e21d5938de0869bcd2c3a" is the MD5 hash of the password.

3.3.4 - Extra information

For load balancing across the three PostgreSQL nodes, you can use the PgBouncer configuration in your pgbouncer.ini file as shown above. PgBouncer does not provide advanced load balancing or failover mechanisms by itself; it will round-robin requests across the specified hosts.

The connection between back and PgBouncer should be very close or in the same area (CPD), therefore using PgPool with PgBouncer, both including Postgres must be in the same area at least one node of every cluster.

3.3.5 - Pooler working

In PgBouncer, the pooler is a core component that manages and optimises the use of database connections. It essentially acts as a middleman between client applications and PostgreSQL, pooling connections to reduce the overhead associated with establishing and tearing down database connections. Here's how the pooler works in PgBouncer:

3.3.5.1 - Connection Pooling Basics

- Purpose: The primary goal of the PgBouncer pooler is to manage a limited set of database connections efficiently, sharing them among multiple client applications. This helps to minimise the resource consumption on the PostgreSQL server and reduces the time spent on connection establishment and teardown.
- How It Works:
    - When a client application connects to PgBouncer, it doesn't directly connect to PostgreSQL. Instead, it connects to PgBouncer, which manages a pool of connections to PostgreSQL.
    - PgBouncer decides whether to allocate an existing connection from the pool to the client or, if none are available, to create a new one (within configured limits).

3.3.5.2 - Pooling Modes

PgBouncer offers three different pooling modes to manage connections based on the needs of your application:

1. Session Pooling (pool_mode = session):
    - How It Works: In this mode, each client connection is mapped to a distinct database connection for the duration of the client's session. When the session ends, the connection is returned to the pool.
    - Use Case: Best suited for applications where a connection needs to maintain state across multiple queries (e.g., using temporary tables or session-level variables).

2. Transaction Pooling (pool_mode = transaction):
    - How It Works: In this mode, each client transaction gets its own database connection from the pool. Once the transaction is complete, the connection is returned to the pool and can be reused by another transaction.
    - Use Case: Ideal for stateless applications where transactions are independent, as it allows for more efficient reuse of connections.

3. Statement Pooling (pool_mode = statement):
    - How It Works: Each SQL statement is executed on a separate database connection. After the statement is executed, the connection is returned to the pool immediately.
    - Use Case: Suitable for very lightweight and stateless queries but generally less common due to its restrictive nature.

3.3.5.3 - Pooling Configuration Parameters

Several key parameters control the behaviour of the connection pooler:

- default_pool_size: Specifies the maximum number of connections that PgBouncer will maintain per database-user pair. When this limit is reached, additional client requests must wait for a connection to become available.
- min_pool_size: Specifies the minimum number of connections that PgBouncer will keep open at all times per database-user pair. This helps reduce latency for the first few client requests.
- reserve_pool_size: This is an additional number of connections that PgBouncer can allocate beyond default_pool_size in case of high demand. This acts as a buffer to handle sudden spikes in traffic.
- reserve_pool_timeout: Determines how long a client can wait for a connection from the reserve pool before being dropped.

3.3.5.4 - Connection Lifecycle Management

- Connection Reuse: When a client releases a connection, PgBouncer does not immediately close it. Instead, it returns the connection to the pool, where it waits to be reused by another client. This reduces the overhead on PostgreSQL, as opening and closing connections repeatedly can be expensive.
- Connection Limits: PgBouncer enforces limits on the number of connections it maintains to the PostgreSQL server using the max_db_connections and max_user_connections parameters. These parameters ensure that the PostgreSQL server is not overwhelmed by too many simultaneous connections.
- Idle Connection Management: PgBouncer can close idle connections after a certain period (server_idle_timeout) to free up resources when they are no longer needed.

3.3.6 - Switch Over

PgBouncer only works with one database, but you can Round-Robin with many different ip connections. This means that if you want to add different type of databases such as read-only and read-write databases you can do it in a different connection:

```
[databases]
; Define the database and connection strings for the four nodes
; r & w
db_pgcloud= host=10.0.0.1,10.0.0.2 port=5432 dbname=db_pgcloud
; r -only
db_pgcloud_r= host=10.0.0.3,10.0.0.4 port=5432 dbname=db_pgcloud
```

There are some limitations that PgBouncer does not allow, rather as a connection pooler, it can not natively handle load balancing between multiple database instances or replicas. Its primary purpose is to pool connections to a single PostgreSQL instance (or database) and manage the reuse of those connections to optimise performance.

4. The Big Manual

This section explains how to deploy PgCloud 1.1.0 on your servers, how to operate the cluster and what to do when something goes wrong. If your purpose is to learn how PostgreSQL, Pgpool-II and PgBouncer work internally, go to Section 3 first: this manual focuses on doing.

There are three ways to deploy PgCloud. All of them use the same Docker images and the same configuration files, so you can start with one and move to another later:

Way	Best for	What you need
A. Automatic (PgCloud Launcher)	Production and anyone who prefers a visual tool	One machine to run the launcher, SSH access to the servers
B. Single host (Docker Compose)	Development, demos, evaluating PgCloud	One machine with Docker
C. Manual, server by server	Full control, custom environments, learning	The repository (or the images) on each server

Section 5 is a step-by-step tutorial of way A with screenshots. Section 6 documents every test and benchmark that was run before releasing versions 1.0.0 and 1.1.0.

4.1 - Before you start

4.1.1 - What gets deployed

PgCloud is distributed as four Docker images. The three node images run on your database servers; the launcher image runs anywhere that can reach them.

Image	Contains	Base image	Size
`pgcloud/db`	PostgreSQL 18 (primary or streaming replica) + PgCloud scripts	`postgres:18`	561 MB
`pgcloud/pgpool`	Pgpool-II 4.7 (PGDG): load balancing, health checks, failover, watchdog	`ubuntu:24.04`	410 MB
`pgcloud/pgbouncer`	PgBouncer 1.23.1 (compiled from source): connection pooling	Red Hat `ubi9`	858 MB
`pgcloud/launcher`	Web UI + API that designs, deploys and monitors clusters over SSH	`node:22-alpine`	184 MB

On every server each node runs as one container with a fixed name: `pgcloud_db`, `pgcloud_pool` or `pgcloud_bouncer`. Pgpool-II uses those names in its failover commands, so a server can run **at most one node of each type** (for example a PostgreSQL primary, a Pgpool-II and a PgBouncer together).

4.1.2 - Server requirements

- **Operating system**: any Linux distribution able to run containers (x86_64 or arm64). Tested on Alpine 3.21; Debian, Ubuntu, RHEL/Rocky and similar work the same way.
- **Container runtime**: Docker 20.10+ or Podman 4+. The SSH user must be able to run it (member of the `docker` group, root, or passwordless `sudo`, which the launcher can use).
- **SSH server** (only for the launcher and for Pgpool-II automatic failover recovery). Pgpool-II logs in to the database servers with the **password** of each server to repair a failed node.
- **Resources**: 2 vCPU and 2 GB RAM is enough to try it; size PostgreSQL memory with the settings of section 4.6.1. Reserve disk for the data volume (`pgcloud_<node>_data`) and about 2 GB for the images.
- **Internet access** on the servers only if the images are built there (default of the launcher) or pulled from a registry. Offline servers can receive the images as a `tar.gz` archive.

4.1.3 - Network and ports

Containers use host networking, so the ports below are opened directly on the servers. They must be reachable between the servers and from the launcher, and blocked from anywhere else.

Port	Service	Who connects
5432	PostgreSQL (`PGPORT`)	Replicas, Pgpool-II, launcher monitoring, applications (2-tier)
5434	Pgpool-II (`PGPOOL_PORT`)	PgBouncer, applications
9898	Pgpool-II PCP administration	Local tools only (`attach.sh`, `detach.sh`...)
9000 / 9694	Pgpool-II watchdog / heartbeat	Other Pgpool-II nodes (only with 2+ Pgpool-II)
6432	PgBouncer (`PGB_PORT`)	Applications, launcher monitoring
22	SSH	Launcher; Pgpool-II towards the PostgreSQL servers (failover)
8080	PgCloud Launcher web UI	Administrators (keep it private or behind a VPN / TLS proxy)

Authentication is `scram-sha-256` everywhere. PostgreSQL and Pgpool-II only accept remote connections from the networks in `ALLOWED_CIDRS` (by default the private ranges `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`).

4.2 - Getting the images

You only need the images on the servers. There are four ways to get them there; the launcher supports all of them (Settings → PgCloud images).

Source	How	When to use it
build	The launcher copies the PgCloud sources it ships to each server (SFTP) and runs `docker build` there. Images are labelled with a hash of their sources and only rebuilt when the sources change.	Default. No registry needed. First deploy takes 5–10 minutes.
registry	`docker pull <registry>/<name>:<tag>` on each server. Selecting it fills in the official images: `docker.io/zhenglinlei`, `pgcloud-db`, `pgcloud-pgpool`, `pgcloud-pgbouncer`, tag `1.1.0`.	Fastest. Official images on Docker Hub, or your own registry.
tar.gz	Upload an archive made with `docker save`; it is copied once per server and loaded with `docker load`.	Air-gapped servers.
on hosts	Nothing: the images must already exist on every server.	You build or load them yourself.

The repository root has one script per task (bash, macOS or Linux). Every script has `--help`; optional settings such as `PGCLOUD_HUB` (Docker Hub namespace) go in `.env`.

Script	What it does
`./BUILD.sh [launcher db pgpool pgbouncer]`	Builds the images on this machine (all four by default), tagged `zhenglinlei/pgcloud-<name>:<version>` and `:latest`.
`./EXPORT.sh tar`	Saves `dist/pgcloud-nodes-<version>.tar.gz` (the three node images, ready for the launcher tar.gz source) and `dist/pgcloud-launcher-<version>.tar.gz`.
`./EXPORT.sh hub`	Builds for `linux/amd64` and `linux/arm64` and pushes to Docker Hub with the tags `<version>`, `<major.minor>` and `latest` (run `docker login` first).
`./RUN.sh`	Starts the launcher from Docker Hub, or `--local` with the image of `./BUILD.sh`. Also `--logs`, `--stop`, `-p <port>`, `--password <pw>`.
`./RUN.dev.sh`	Playground: three simulated servers and the launcher built from the checkout (section 5.11).
`./CLEAN.sh`	Removes build leftovers; `--dev`, `--images`, `--launcher [--data]`, `--all`, `-n` (dry run).

Publishing a GitHub release runs the workflow `.github/workflows/release-images.yml`, which does the same as `./EXPORT.sh hub` on GitHub. For air-gapped servers, upload the nodes archive in the launcher or copy it yourself and run `gunzip -c pgcloud-nodes-1.1.0.tar.gz | docker load` on each server.

4.3 - Way A: automatic deployment with the PgCloud Launcher

The launcher is a web application packaged as a Docker image. You draw the cluster, give it SSH access to the servers and press Deploy. For every node it detects Docker or Podman, prepares the image, writes the configuration files, (re)creates the container and waits until the node accepts connections, always in the order primary → replicas → Pgpool-II → PgBouncer. Then it monitors the cluster in real time.

```
$ docker run -d --name pgcloud-launcher --restart unless-stopped \
    -p 8080:8080 -v pgcloud-launcher:/data zhenglinlei/pgcloud-launcher:1.1.0
$ docker logs pgcloud-launcher        # prints the generated admin password
$ ./RUN.sh                            # the same from the repository, prints URL and password
```

Open `http://<launcher-host>:8080` and follow Section 5. The launcher stores its state in the `/data` volume: an SQLite database with the topology, settings, events and deployment logs, the key that encrypts the stored secrets, and the uploaded image archives. Back up that volume.

Variable	Default	Description
`PORT` / `HOST`	`8080` / `0.0.0.0`	Listening address of the web UI and API.
`DATA_DIR`	`/data`	Database, encryption key and archives. Mount it as a volume.
`LAUNCHER_ADMIN_PASSWORD`	generated	Fixed admin password. If unset, a random one is generated on first start, printed in the logs and saved in `/data/admin-password.txt`.
`LAUNCHER_SECRET_KEY`	key file	Key used to encrypt SSH and cluster passwords (AES-256-GCM). If unset, `/data/secret.key` is created.
`LAUNCHER_SECURE_COOKIES`	`false`	Set to `true` when the UI is served over HTTPS.

What the launcher creates on each server:

Node	Container	Files in ~/pgcloud/<node>/ (mode 700 / 600)
PostgreSQL	`pgcloud_db` + volume `pgcloud_<node>_data`	`.env`
Pgpool-II	`pgcloud_pool`	`.env`, `conf/postgres/server<N>.env`, `conf/pgpool/backend<N>.env`
PgBouncer	`pgcloud_bouncer`	`.env`

> **Security.** The launcher holds administrator access to every server. Expose it only on a private network or behind a VPN or TLS reverse proxy, and keep its `/data` volume private.

4.4 - Way B: single host with Docker Compose

`compose.dev.yml` at the repository root starts a complete three-tier cluster on one machine: a primary, two streaming replicas, Pgpool-II and PgBouncer. It is the fastest way to evaluate PgCloud or to develop against it.

```
$ cp .env.example .env              # set POSTGRES_PASSWORD, REPLICA_PASS and PCP_PASS
$ docker compose -f compose.dev.yml up -d --build
$ docker compose -f compose.dev.yml ps    # every service becomes "healthy"
$ psql -h localhost -p 6432 -U u_pgcloud db_pgcloud
```

Service	Role	Port on the host
`db0`	PostgreSQL primary	–
`db1`, `db2`	PostgreSQL streaming replicas	–
`pgpool`	Load balancing (reads to the replicas), health checks	5434
`pgbouncer`	Connection pooling in front of Pgpool-II	6432

Add `-f compose.dev.yml` to every command: `logs -f pgpool`, `exec pgpool ./script/status.sh` (Pgpool-II view of the nodes), `exec db0 ./script/cluster.sh` (replication status), `down` (stop, keep data) and `down -v` (delete everything). Use `up -d db0 db1 db2 pgpool` for a two-tier cluster without PgBouncer.

> **Limitation.** Everything runs on one host, so this setup protects you from nothing if the machine fails. Automatic failover recovery over SSH is not wired up in this mode.

4.5 - Way C: manual deployment, server by server

This is exactly what the launcher does, done by hand. Each folder in `docker/` contains a `Dockerfile`, `BUILD.sh`, `RUN.sh` and a documented `.env.example`. The example below uses three servers:

Server	Address	Nodes
A	10.0.0.11	PostgreSQL primary `node0`, Pgpool-II, PgBouncer
B	10.0.0.12	PostgreSQL replica `node1`
C	10.0.0.13	PostgreSQL replica `node2`

`RUN.sh` starts a container with `--rm` and opens a shell in it, which is handy for a first try. For a real cluster start the containers in the background with a restart policy, as shown in each step. Pgpool-II needs the PostgreSQL containers to survive a stop to repair them after a failover.

4.5.1 - PostgreSQL primary (server A)

```
$ cd docker/db && cp .env.example .env

# docker/db/.env (only the values to change)
POSTGRES_PASSWORD=<strong password>
REPLICA_PASS=<strong password>
NODE_ROLE=primary
BACKEND_NAME=node0
ALLOWED_CIDRS=10.0.0.0/24
SHARED_BUFFERS=1GB

$ ./BUILD.sh                          # or load / pull the image
$ docker run -d --name pgcloud_db --restart unless-stopped --network host \
    --env-file .env -v pgcloud_node0_data:/opt/pgdata \
    -v "$PWD/.env":/opt/pgcloud/.env pgcloud_db
```

On the first start the entrypoint runs `initdb` with data checksums, creates the database and the replication role, writes `postgresql.conf` and `pg_hba.conf` and starts PostgreSQL. It refuses to start while a password is empty or still `change_me`.

4.5.2 - PostgreSQL replicas (servers B and C)

Same image and same passwords. Change only the role, the node name and where the primary is:

```
NODE_ROLE=replica
BACKEND_NAME=node1                  # node2 on server C
PRIMARY_HOST=10.0.0.11
PRIMARY_PORT=5432
```

On the first start the replica runs `pg_basebackup` from the primary, creates its own replication slot `pgcloud<id>` there and starts streaming. With the default `SYNCHRONOUS_RULE=FIRST 1` and `SYNCHRONOUS_NAME=*` the first connected replica is synchronous and the others are potential candidates. Leave both empty for asynchronous replication.

4.5.3 - Pgpool-II (server A)

Pgpool-II needs one file per PostgreSQL node and one per Pgpool-II node, numbered from 0. `server0.env` must be the initial primary.

```
$ cd docker/pgpool && cp .env.example .env
$ cp conf/postgres/server.sample.env conf/postgres/server0.env    # + server1.env, server2.env
$ cp conf/pgpool/backend.sample.env  conf/pgpool/backend0.env

# conf/postgres/server1.env
PSQL_HOST=10.0.0.12
PSQL_PORT=5432
PSQL_WEIGHT=1                # 0 = no read traffic
```

```
PSQL_FAIL=ALLOW_TO_FAILOVER
PSQL_PATH=/opt/pgdata
PSQL_NAME=node1               # = BACKEND_NAME of that node
SSH_USER=root                 # used by the automatic failover recovery
SSH_PORT=22
SSH_PASS=<password of server B>

# docker/pgpool/.env (values to change)
POSTGRES_PASSWORD=<same as the database>
REPLICA_USER=pgrepuser
REPLICA_PASS=<same as the database>
PCP_PASS=<strong password>
PGPOOL_HOST=10.0.0.11
PGPOOL_NUM=1                  # 1 = no watchdog; use 3+ for a watchdog cluster
ALLOWED_CIDRS=10.0.0.0/24

$ ./BUILD.sh
$ docker run -d --name pgcloud_pool --restart unless-stopped --network host --env-file .env \
    -v "$PWD/conf/pgpool":/opt/pgcloud/conf/pgpool:ro \
    -v "$PWD/conf/postgres":/opt/pgcloud/conf/postgres:ro \
    -v "$PWD/conf/pgbouncer":/opt/pgcloud/conf/pgbouncer:ro pgcloud_pool
```

The failover behaviour is configured with the `FAILOVER_*` variables of `.env`; the defaults implement the flow of section 4.7.3. In the commands, `%h`/`%p` are the new primary and `%x`/`%r` the old primary.

4.5.4 - PgBouncer (server A)

```
# docker/pgbouncer/.env (values to change)
POSTGRES_PASSWORD=<same as the database>
BACKEND_HOST=10.0.0.11        # Pgpool-II (3-tier) or the primary (2-tier)
BACKEND_PORT=5434
POOL_MODE=transaction
MAX_CONNECTIONS=2000
DEFAULT_POOL_SIZE=100

$ cd docker/pgbouncer && ./BUILD.sh
$ docker run -d --name pgcloud_bouncer --restart unless-stopped --network host --env-file .env pgcloud_bouncer
```

For separate read-write and read-only pools (PgBouncer directly in front of PostgreSQL), copy `conf/backend.sample.env` to `conf/backend.env` and mount `./conf` at `/opt/pgcloud/conf/server`.

4.5.5 - Verify the cluster

```
$ docker exec pgcloud_db ./script/cluster.sh          # on the primary: replicas, sync state, slots
$ docker exec pgcloud_pool ./script/status.sh         # SHOW pool_nodes: every node "up"
$ docker exec pgcloud_bouncer ./script/getpool.sh     # SHOW POOLS
$ psql -h 10.0.0.11 -p 6432 -U u_pgcloud db_pgcloud -c "select 1"
```

4.6 - Configuration reference

Every variable is documented in the `.env.example` of each folder. These are the ones you will touch most often.

4.6.1 - PostgreSQL node (`docker/db/.env`)

Variable	Default	Meaning
`POSTGRES_DB` / `POSTGRES_USER` / `POSTGRES_PASSWORD`	`db_pgcloud` / `u_pgcloud` / –	Database and superuser.
`PGPORT`, `PGDATA`	`5432`, `/opt/pgdata`	Port and data directory.
`ALLOWED_CIDRS`	private ranges	Networks allowed in `pg_hba.conf` (normal and replication).
`MAX_CONNECTIONS`	`1000`	Must exceed what Pgpool-II opens (`num_init_children × max_pool`).
`SHARED_BUFFERS`, `WORK_MEM`, `EFFECTIVE_CACHE_SIZE`, `MAINTENANCE_WORK_MEM`	256MB, 128MB, 512MB, 64MB	25–40 %, ~4 %, 50–75 % and ~1 % of the RAM.
`NODE_ROLE`, `BACKEND_NAME`	`primary`, `node0`	Role on first start and replication `application_name`.
`REPLICA_USER` / `REPLICA_PASS`	`pgrepuser` / –	Replication role (same on every node and in Pgpool-II).
`SYNCHRONOUS_RULE` / `SYNCHRONOUS_NAME`	`FIRST 1` / `*`	Builds `synchronous_standby_names`; empty = asynchronous.
`PRIMARY_HOST` / `PRIMARY_PORT`	–	Primary to clone from (replicas). Updated automatically after a failover.
`PROMOTE_MIN_DELAY`	`0`	Max replay delay (s) accepted to promote this replica; 0 = always.
`WAL_KEEP_SIZE`	`1GB`	WAL kept on every node so `pg_rewind` can resynchronise a former primary.

4.6.2 - Pgpool-II node (`docker/pgpool/.env`)

Variable	Default	Meaning
`PGPOOL_HOST` / `PGPOOL_PORT`	– / `5434`	Address of this node and listening port.
`PGPOOL_BALANCE_MODE`	`on`	Send read-only queries to the replicas.
`NUM_INIT_CHILDREN` / `MAX_POOL`	`250` / `4`	Concurrent client sessions and cached connections per process.
`HEALTH_CHECK_PERIOD` / `_TIMEOUT`	`10` / `20` s	How fast a dead backend is detected.

Variable	Default	Meaning
PCP_USER / PCP_PASS / PCP_PORT	– / – / 9898	Administration interface used by the tools.
PGPOOL_NUM, WD_*	3, …	Number of Pgpool-II nodes (1 disables the watchdog) and watchdog tuning.
FAILOVER_PROMOTE_CMD	… pgcloud_promote	Run over SSH on the new primary.
FAILOVER_REDIRECT_CMD	… pgcloud_redirect	Run on every other replica.
FAILOVER_DEPROMOTE_CMD	stop → pgcloud_rewind → start	Turns the old primary into a replica.
FAILOVER_RECOVER_REPLICA	true	Restart and re-attach failed replicas automatically.

4.6.3 - PgBouncer node (docker/pgbouncer/.env)

Variable	Default	Meaning
BACKEND_HOST / BACKEND_PORT	127.0.0.1 / 5432	Where PgBouncer connects (Pgpool-II or PostgreSQL).
PGB_PORT, PGB_AUTH	6432, scram-sha-256	Listening port and client authentication.
POOL_MODE	transaction	session, transaction or statement.
MAX_CONNECTIONS / DEFAULT_POOL_SIZE	2000 / 100	Client connections accepted / server connections per pool.

4.7 - Operating the cluster

4.7.1 - Connecting applications

Applications connect to the entry point of the topology: PgBouncer (6432) in three tiers, Pgpool-II (5434) in two tiers. Both hide failovers: the address never changes.

```
postgresql://u_pgcloud:<password>@10.0.0.11:6432/db_pgcloud
```

Open connections are closed when the primary dies, exactly as with a single PostgreSQL server: use a driver or pool that reconnects and retries the transaction.

4.7.2 - Everyday commands

Where	Command	What it does
db	./script/connect.sh	psql to the local database.

Where	Command	What it does
db	`./script/getrole.sh`	Prints `true` on a primary and `false` on a replica.
db	`./script/cluster.sh`	Replication, archiver and slots (primary) or WAL receiver (replica).
db	`./tools/add_slot.sh`, `./tools/rm_slot.sh`	Create or drop a replication slot.
pgpool	`./script/status.sh`	`SHOW pool_nodes` through Pgpool-II.
pgpool	`./tools/attach.sh <id>`, `detach.sh <id>`, `recovery.sh <id>`	Re-attach, detach or recycle a backend with PCP.
pgbouncer	`./script/getpool.sh`, `getclient.sh`, `getserver.sh`, `status.sh`	`SHOW POOLS / CLIENTS / SERVERS / STATS`.

Run them with `docker exec <container> <command>`. Logs: `docker logs <container>`; Pgpool-II also writes `/opt/pgcloud/log/failover.log` and `failback.log`.

4.7.3 - What happens when the primary fails

Pgpool-II detects the failure with its health check (or immediately, on a backend error) and runs `failover.sh`. With the PgCloud defaults the cluster repairs itself:

1. **Promote.** Over SSH, `pgcloud_promote` checks the replay delay of the best replica and promotes it. Synchronous replication is then switched off on the new primary, otherwise every commit would wait for a standby that is not there yet.
2. **Redirect.** Every other replica runs `pgcloud_redirect`: it creates its slot on the new primary, changes `primary_conninfo` and reloads.
3. **Rewind.** The old primary is stopped cleanly and `pgcloud_rewind` runs in a one-off container that shares its volumes: `pg_rewind` against the new primary, standby configuration, own slot kept. Then the container starts again as a replica. At no moment are there two primaries.
4. **Re-attach.** About 15 seconds after the failover script returns, the node is attached back to Pgpool-II.

Since 1.1.0 Pgpool-II also runs with `auto_failback = on`: a replica that Pgpool-II detached (for example while its server rebooted) is attached again by itself as soon as it streams from the primary. Pgpool-II never attaches a **primary** by itself, though. If the whole system restarts and the primary comes back after Pgpool-II, the primary stays detached and writes fail. The launcher detects any running node that Pgpool-II keeps detached and shows a warning; select the Pgpool-II node and press **Re-attach** (by hand: `docker exec pgcloud_pool`

`./tools/attach.sh <node_id>`; with the watchdog one Pgpool-II node is enough).

Two safety nets complete the flow. Every node keeps `WAL_KEEP_SIZE` (1 GB) of WAL so `pg_rewind` finds what it needs; if it still fails, `pgcloud_rewind` clones the node again with `pg_basebackup`. And a node configured as a replica that finds the data of a primary (for example an old primary that was down during the failover) rejoins the primary in `PRIMARY_HOST` before starting, so it can never become a second primary. Since 1.1.0 the role is also recorded in the node's mounted `.env` (`NODE_ROLE=primary` after a promotion), the entrypoint reads it from there, and it never rejoins a host that is itself a replica, so restarting a promoted node is safe.

Measured times are in section 6.4. To repair a node by hand (for example after an SSH failure), run on its server:

```
$ docker stop -t 60 pgcloud_db
$ docker run --rm --network host --volumes-from pgcloud_db --entrypoint pgcloud_rewind \
    $(docker inspect -f '{{.Config.Image}}' pgcloud_db) <new_primary_ip> 5432
$ docker start pgcloud_db
$ docker exec pgcloud_pool ./tools/attach.sh <node_id>       # on the Pgpool-II server
```

> After a failover the new primary replicates asynchronously. In the launcher, press **Adopt live roles** and deploy the cluster: the drawing follows the new roles and synchronous replication is enabled again as soon as a replica streams. By hand: `ALTER SYSTEM SET synchronous_standby_names = 'FIRST 1 (*)'` and `SELECT pg_reload_conf()`.

4.7.4 - Planned switchover and watchdog leader

To move the primary role on purpose (server maintenance, hardware change, a test), select a running replica in the launcher and press **Make primary** (section 5.10). It is a planned switchover, not a failover: no committed transaction is lost.

1. All Pgpool-II nodes are stopped at once, so none of them starts a failover on its own.
2. The primary is stopped cleanly: its last WAL record (the shutdown checkpoint) is streamed to the replicas. The launcher reads that position with `pg_controldata` and waits until the chosen replica replayed it. If that does not happen within 60 seconds, the old primary is started again and nothing changes.
3. The replica is promoted (`pgcloud_promote`) and the other replicas follow it (`pgcloud_redirect`).
4. The old primary runs `pgcloud_rewind` (after a clean stop usually "no rewind required") and starts as a replica.
5. The drawing is updated like **Adopt live roles** (PgBouncer nodes linked straight to the old primary move to the new one), the configuration files on the PostgreSQL hosts are

rewritten, Pgpool-II is redeployed with the new backend list and synchronous replication is restored.

Writes stop for about a minute (section 6.6). Clients waiting in PgBouncer continue afterwards; clients connected straight to PostgreSQL must reconnect.

With two or more Pgpool-II nodes, their watchdog elects a leader. The inspector of a Pgpool-II node shows LEADER or STANDBY, and Make watchdog leader restarts the current leader until the chosen node wins the election. By hand: `docker exec pgcloud_pool ./script/getrole.sh` shows the watchdog state and `docker restart pgcloud_pool` on the leader hands the role over. Use three or more Pgpool-II nodes: with two, the node left alone has no quorum. Redeploying Pgpool-II (also during a switchover) starts a new election.

4.7.5 - Planned maintenance

Stopping a PostgreSQL container on purpose looks exactly like a crash to Pgpool-II, which would then promote a replica. The launcher pauses the Pgpool-II containers while it redeploys or restarts PostgreSQL nodes (a short, planned interruption of the entry point) and resumes them afterwards. When working by hand, stop Pgpool-II first (`docker stop pgcloud_pool`), do the maintenance, and start it again.

4.7.6 - Growing the cluster

To add a replica: start a new PostgreSQL node with `NODE_ROLE=replica`, add a `server<N>.env` for it on every Pgpool-II node and recreate the Pgpool-II containers (the configuration is generated once and locked with `/etc/pgpool2/lock.pgcloud`). In the launcher this is one node, one link and one click on Deploy.

Adding a Pgpool-II node changes the watchdog member list of all of them, and the watchdog rejects a member whose list differs ("Add to watchdog cluster request is rejected"). Since 1.1.0 deploying one Pgpool-II node redeploys every deployed Pgpool-II node. The launcher also remembers the configuration it deployed on each node and warns when the topology or the settings changed since ("its configuration changed since it was deployed… Redeploy it").

4.8 - Security checklist

- Strong, different passwords for the superuser, replication and PCP. The images refuse `change_me`; the launcher generates random ones.
- `ALLOWED_CIDRS` restricted to your servers and application network. Never `0.0.0.0/0`.

- Database ports reachable only from the private network; the launcher behind a VPN or a TLS proxy (`LAUNCHER_SECURE_COOKIES=true`).
- `.env` files are git-ignored and written with mode 600. Never commit them.
- Pgpool-II stores the SSH password of each PostgreSQL server in `server<N>.env`: use a dedicated user for it.
- TLS between nodes is not enabled by PgCloud 1.1.0; use a private network or a VPN between the servers.

4.9 - Troubleshooting

Symptom	Cause and fix
Container exits at start with "is empty or still change_me"	Set real passwords in `.env` (or in the launcher Settings).
`no pg_hba.conf entry for host`	The client address is not in `ALLOWED_CIDRS`. Add the network and recreate the node.
Pgpool-II: `failed to authenticate with backend using SCRAM`	Different `POSTGRES_PASSWORD` in Pgpool-II and PostgreSQL. Fix it and delete `lock.pgcloud` (or recreate the container).
Writes hang after a failover	Synchronous replication with no standby. PgCloud disables it on promotion since 1.0.0; with older scripts run the `ALTER SYSTEM` of section 4.7.3.
Pgpool-II: `pool_passwd file does not contain an entry`	Images 1.0.0 with a password that starts with `-`: `pg_enc` read it as an option. Fixed in 1.1.0; redeploy Pgpool-II with the new image.
Old primary does not come back as a replica	Check `failover.log`; run the manual rewind of section 4.7.3. If `pg_rewind` fails, destroy the node and deploy it again as a replica.
Pgpool-II shows a running node as `down` (writes fail if it is the primary)	Pgpool-II detached it, e.g. after a restart of every server. Launcher: select the Pgpool-II node → Re-attach. By hand: `./tools/attach.sh <node_id>`.
Pgpool-II restarts in a loop: `pid file found. is another pgpool running?`	Images 1.0.x killed by `docker stop` left their pid file. Fixed in 1.1.0; redeploy Pgpool-II (a new container starts clean).
Pgpool-II restarts in a loop: `Add to watchdog cluster request is rejected`	The Pgpool-II nodes were deployed with different member lists. Redeploy all of them (1.1.0 does it by itself).
Launcher: "No usable container runtime"	Install Docker/Podman, add the SSH user to the `docker` group or enable sudo for the node.
Launcher: "only one postgres container per host"	Two nodes of the same type on one server. Use another server.
Launcher: monitoring shows DEGRADED	TCP works but SQL does not: wrong password or the launcher network is not in `ALLOWED_CIDRS`. The node inspector (Live tab) shows the exact error.

5. Tutorial for noobs

No previous knowledge needed. In this tutorial you will create a real high-availability PostgreSQL cluster with the **PgCloud Launcher**, a web application that does all the work for you: you draw the cluster, tell it how to reach your servers and press a button. Then you will watch it live and break it on purpose to see it repair itself.

Every screenshot in this chapter was taken during a real deployment on two servers, `srv-a` (172.30.0.11) and `srv-b` (172.30.0.12). Your addresses will be different.

5.1 - What you need

- **Two or more Linux servers** with Docker (or Podman) and SSH access as a user that can run it. Cloud VMs, old PCs or virtual machines are all fine.
- **One computer to run the launcher** (it can be one of the servers) with Docker installed.
- The servers must reach each other, and the launcher must reach them.
- No servers yet? Section 5.11 starts a complete playground on your laptop with one command.

5.2 - Step 1: start the launcher

On the computer that will run the launcher, open a terminal and run:

```
$ docker run -d --name pgcloud-launcher -p 8080:8080 -v pgcloud-launcher:/data \
    zhenglinlei/pgcloud-launcher:1.1.0
$ docker logs pgcloud-launcher | grep password
```

Or, from a copy of the PgCloud repository, simply `./RUN.sh`: it starts the same container and prints the address and the password.

The last command shows the **admin password**, generated the first time. Open `http://localhost:8080` (or the address of that computer) in your browser and enter it.

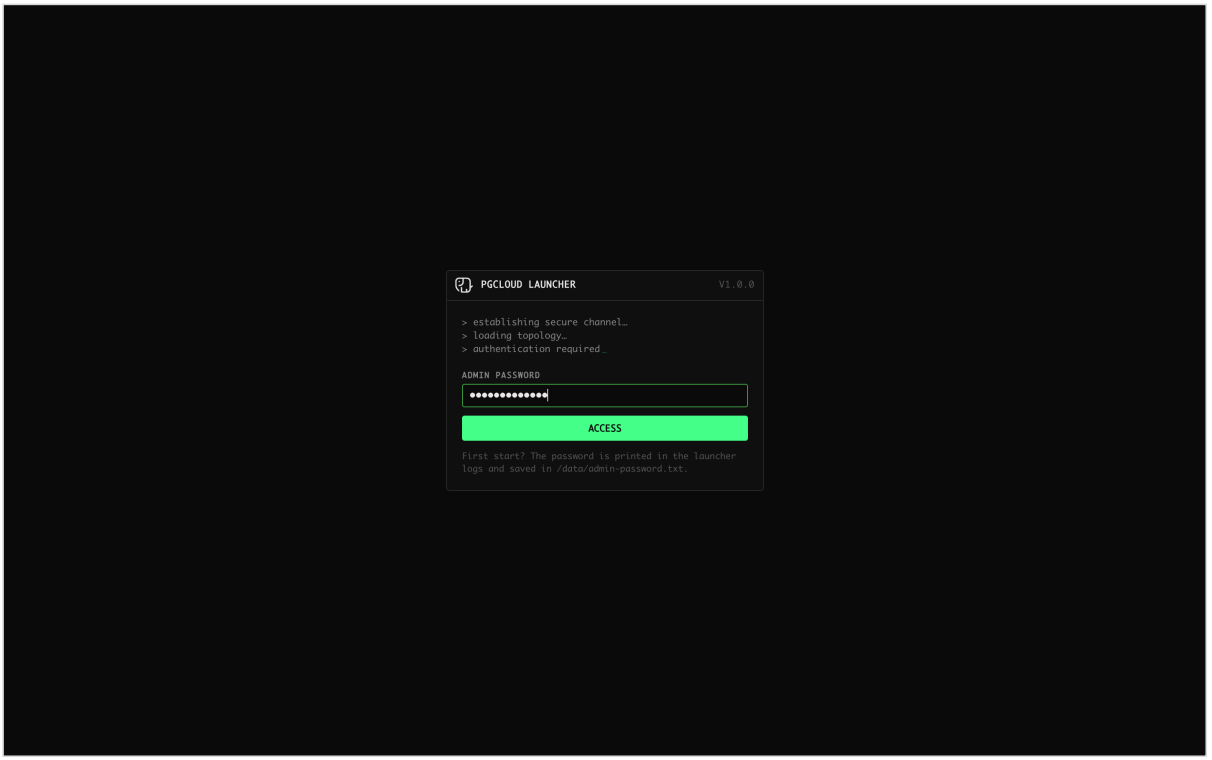


Figure 5.1 - Login screen of the PgCloud Launcher.

5.3 - Step 2: meet the interface

After logging in you see an empty canvas. The top bar shows the logo, a green **LIVE** dot (the page receives updates in real time), how many nodes are up or down, the UTC clock, the light/dark switch and the logout button. Below it there are five tabs:

Tab	What you do there
Topology	Draw the cluster: add nodes, connect them, configure each one. Also shows the live state.
Monitor	Real-time charts and tables for every node.
Deploy	Check that everything is ready, deploy, follow the console, see the history.
Events	Everything that happened: deployments, nodes going down or up, role changes.
Settings	Passwords of the cluster, where the images come from, runtime and monitoring.

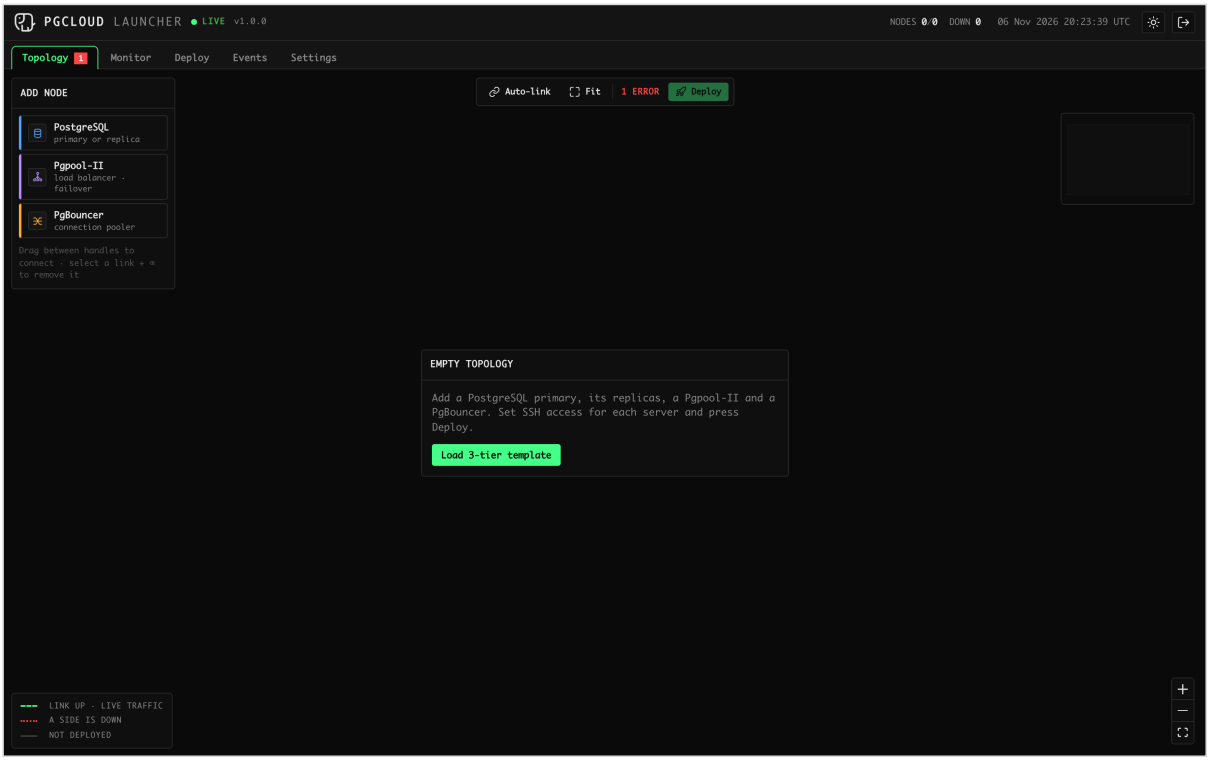


Figure 5.2 - The Topology tab of a new installation.

5.4 - Step 3: check the settings

Open **Settings**. You do not need to change anything to start:

- **Cluster credentials**: strong random passwords were generated for the database superuser, replication and PCP. Click the eye icon to see one (you will need the superuser password to connect your applications).
- **PgCloud images**: **build** means the launcher builds the images on each server from the PgCloud sources it ships. Leave it like that unless you published the images in a registry.
- **Hosts**: the runtime is detected automatically (Docker, otherwise Podman) and the configuration is written to `~/pgcloud` on each server.
- **Allowed networks**: the private networks are allowed by default. If your servers use public addresses, write them here.

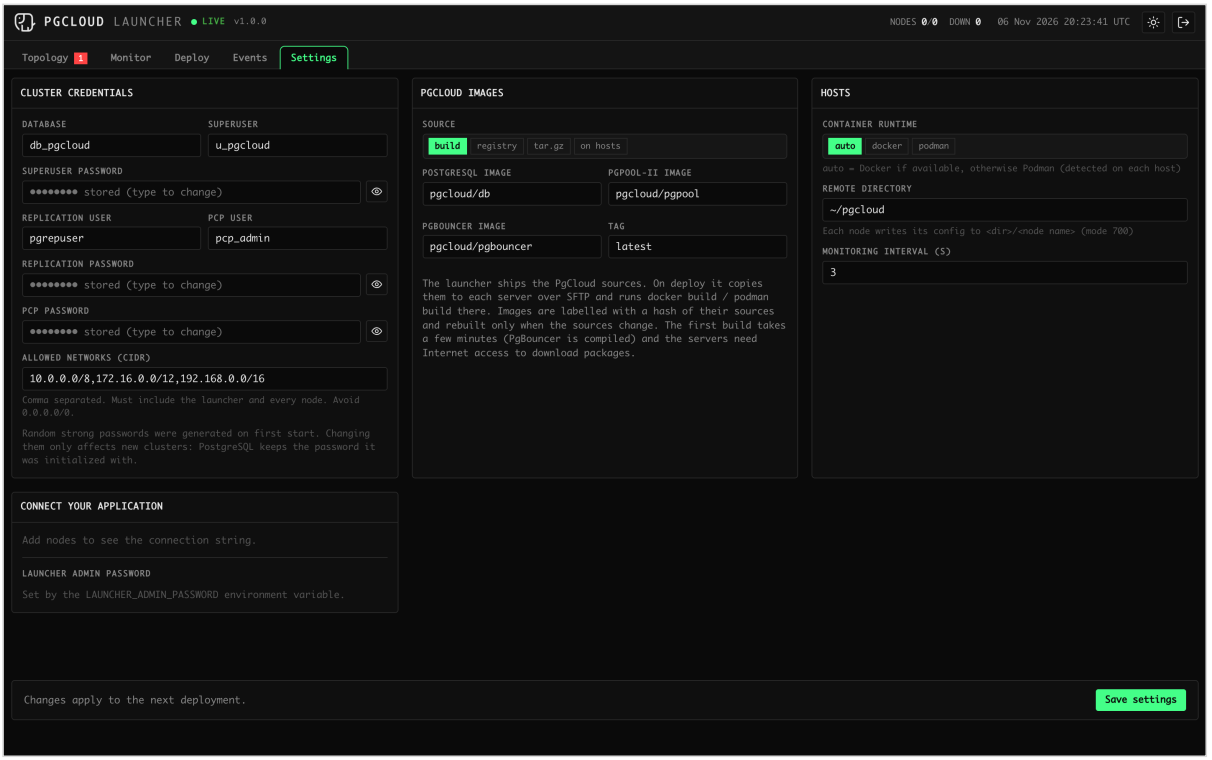


Figure 5.3 - Settings: credentials, image source and hosts.

5.5 - Step 4: draw your cluster

Back in **Topology**, click **Load 3-tier template**. Five nodes appear, already connected: PgBouncer at the top, Pgpool-II in the middle and three PostgreSQL servers at the bottom. You can also drag nodes from the **Add node** panel and connect them yourself by dragging from one small square (handle) to another.

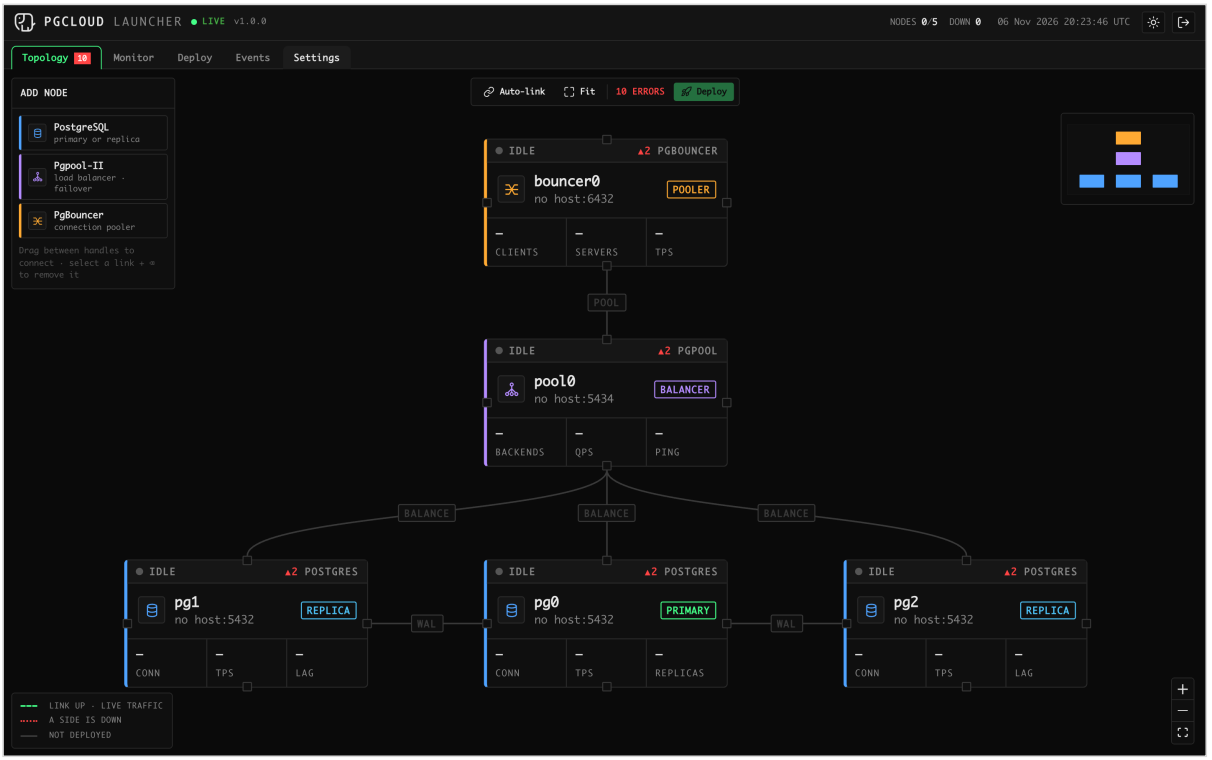


Figure 5.4 - The three-tier template: PgBouncer → Pgpool-II → PostgreSQL primary and replicas.

The links decide the roles, you never type them:

Link	Meaning	Label
PostgreSQL → PostgreSQL	The first one is a **replica** of the second (the **primary**)	WAL / SYNC / ASYNC
Pgpool-II → PostgreSQL	Pgpool-II balances and protects these databases	BALANCE
PgBouncer → Pgpool-II (or PostgreSQL)	PgBouncer pools connections in front of it	POOL

Each server can run only one node of each type. With two servers we keep two PostgreSQL nodes: select the third one and press **Delete node** in its panel. The badge with a number on a node (▲) counts the problems that still block the deployment, for example a missing SSH host.

5.6 - Step 5: tell the launcher how to reach each server

Click a node: its panel opens on the right with four tabs. In **SSH** type the address of the server, the user and the password (or paste a private key), then press **Save & test**. The launcher connects, detects Docker or Podman and shows the result. Do it for every node.

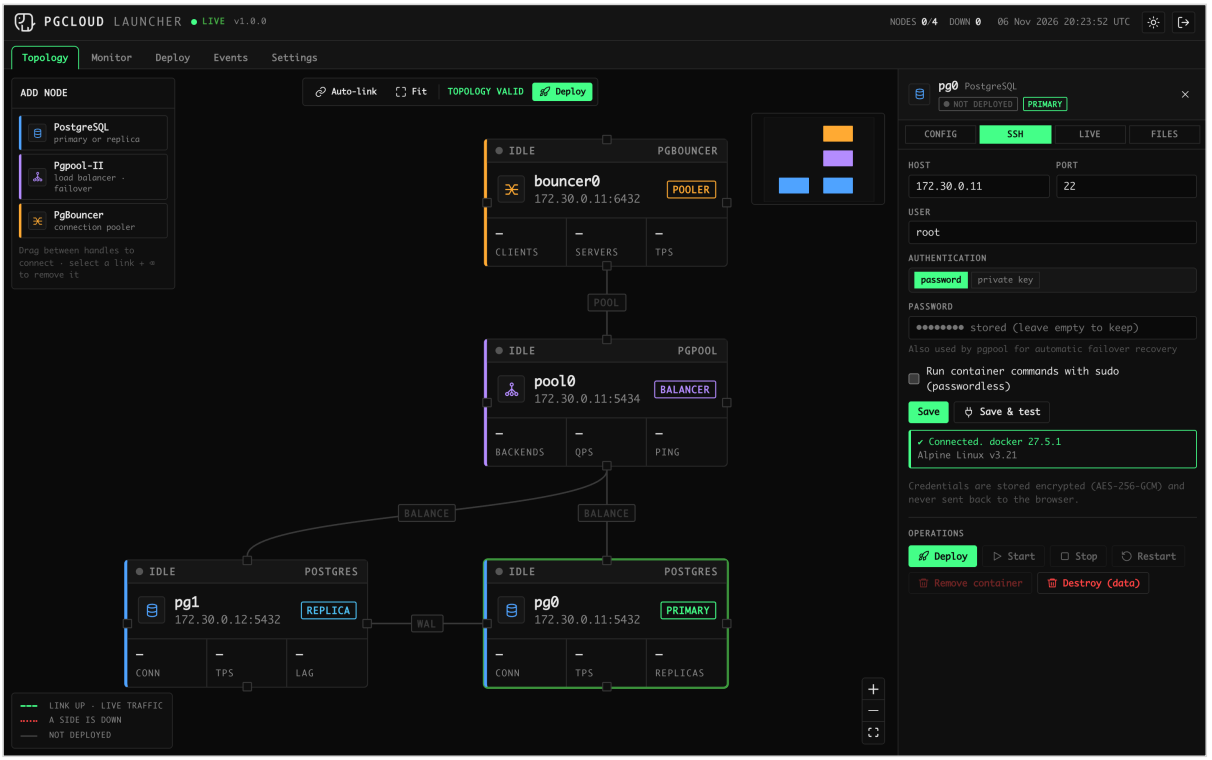


Figure 5.5 - SSH settings of pg0 and a successful connection test.

> **Tip.** Several nodes can use the same server, as long as they are of different types: here the primary, Pgpool-II and PgBouncer share srv-a and the replica uses srv-b. Passwords are stored encrypted and never shown again.

The **Config** tab has the settings of the node (port, memory, pool sizes…); the defaults are fine. The **Files** tab shows exactly what will be written on the server, with the passwords hidden:

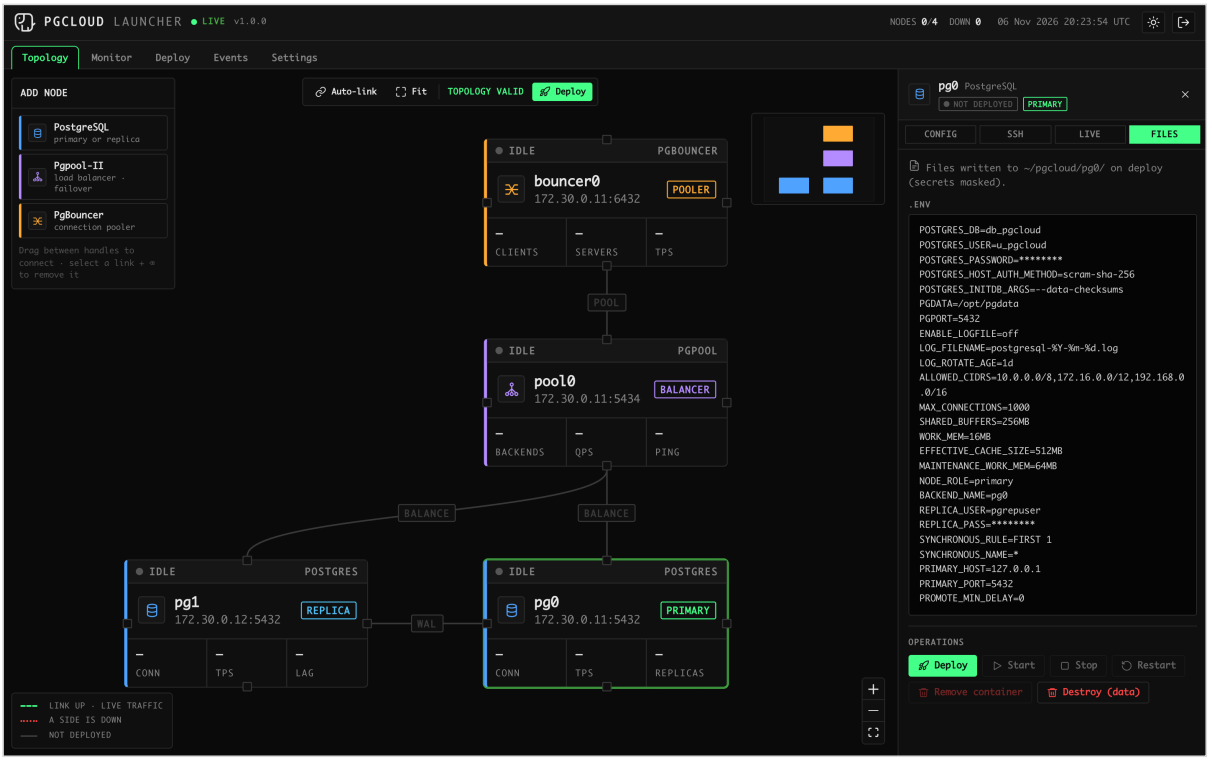


Figure 5.6 - Preview of the configuration files generated for pg0.

When every node is configured, the toolbar says TOPOLOGY VALID:

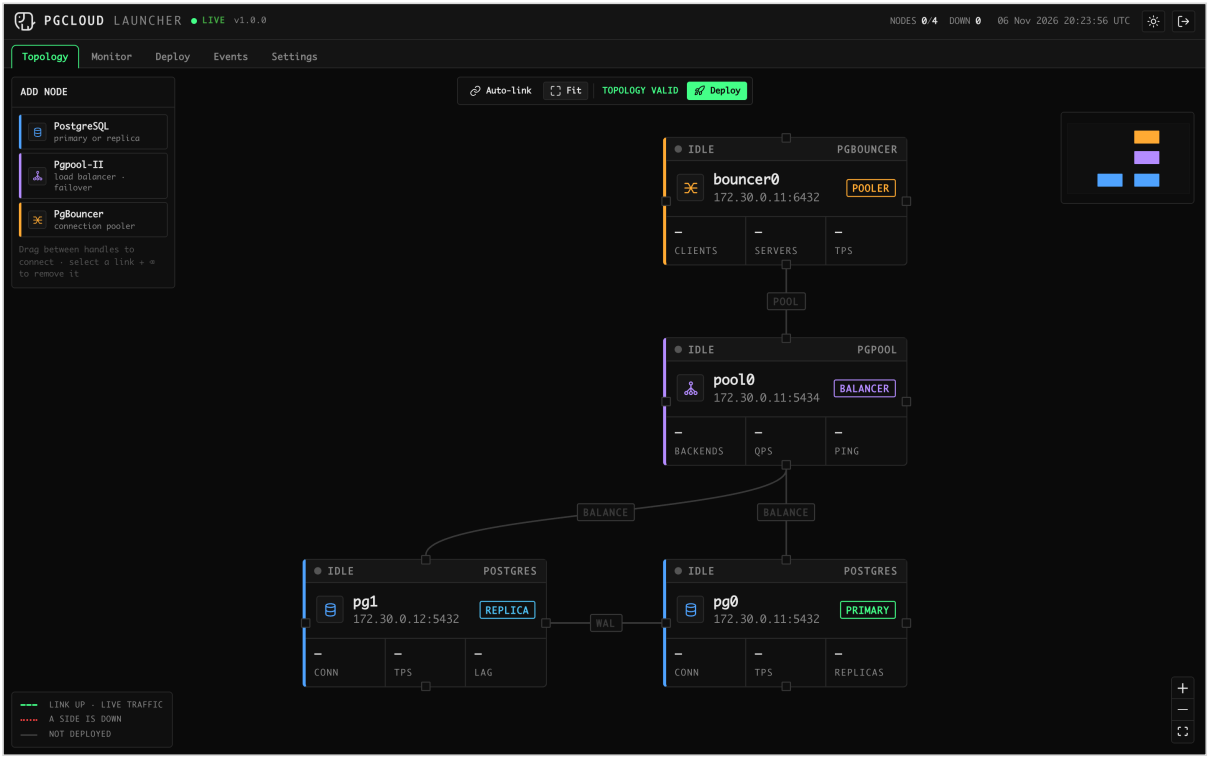


Figure 5.7 - A valid two-server topology, ready to deploy.

5.7 - Step 6: deploy

Open **Deploy**. The **pre-flight checks** must say READY and the **execution plan** lists the order: primary, replicas, Pgpool-II, PgBouncer.

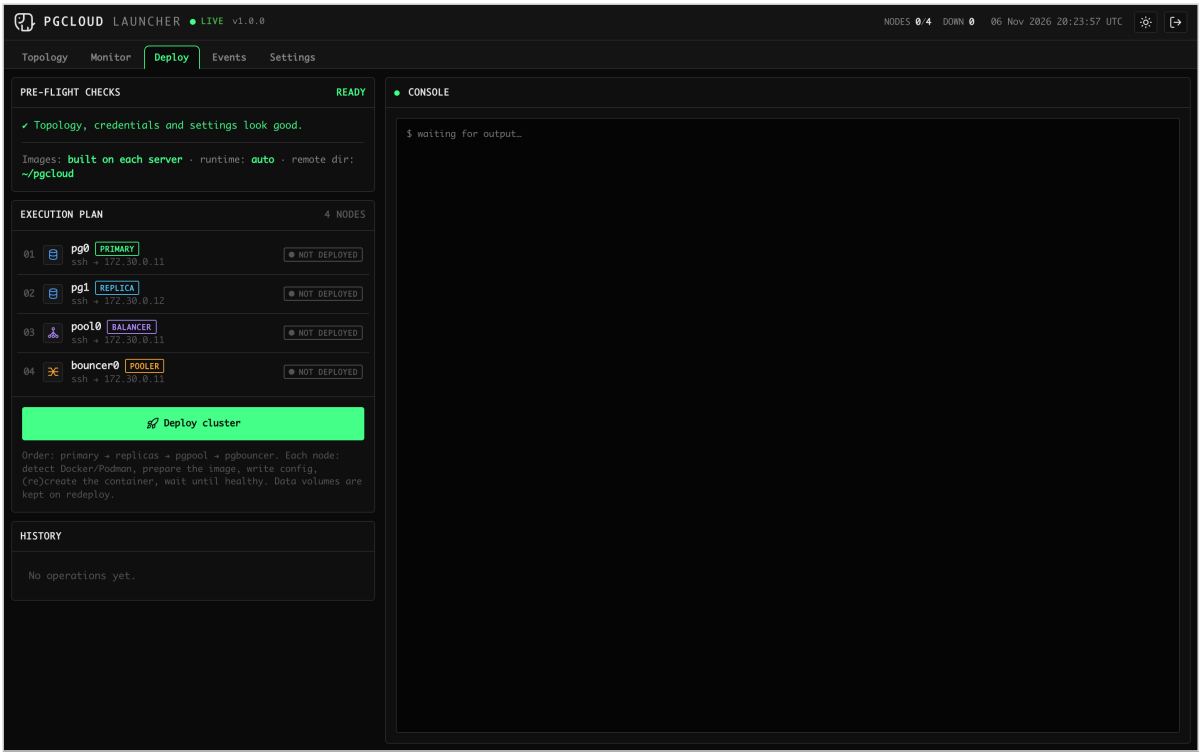


Figure 5.8 - Pre-flight checks and execution plan.

Press **Deploy cluster** and confirm. Existing containers of these nodes would be replaced; data volumes are always kept.

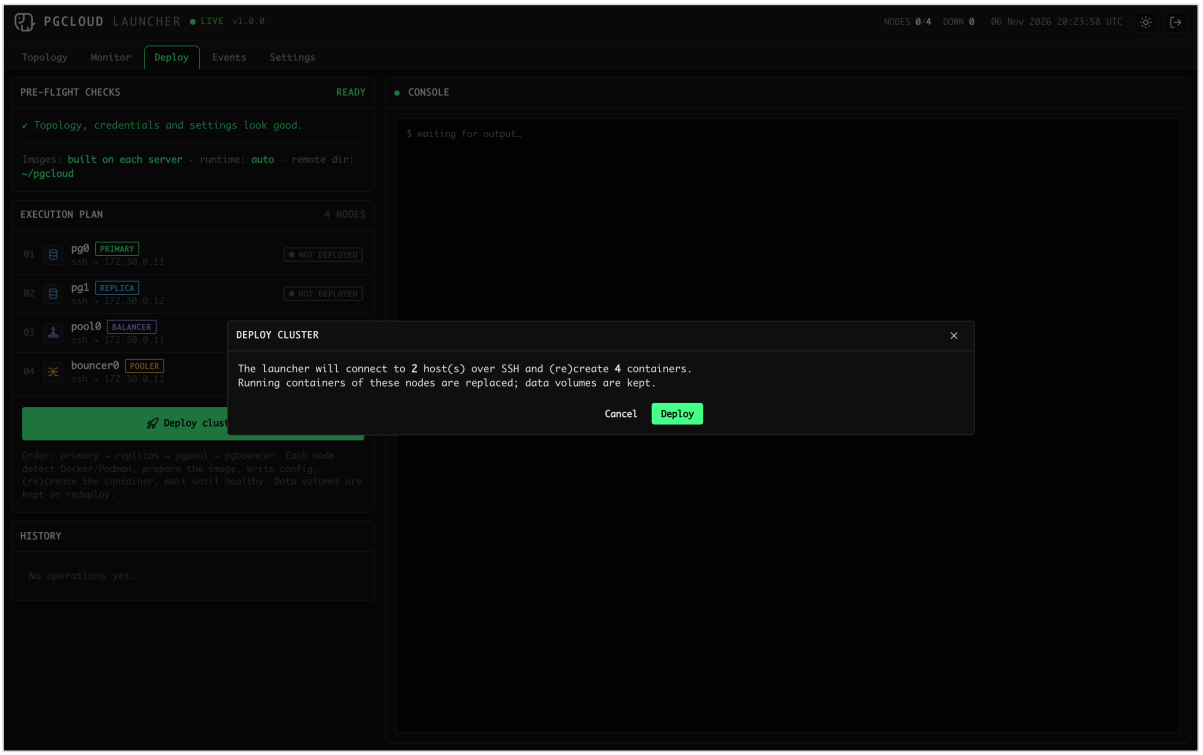


Figure 5.9 - Confirmation before touching the servers.

The console on the right shows everything the launcher does, live: the uploads, every command and the output of `docker build`. The first deployment builds the images on each server, so be patient: it took **6 minutes and 7 seconds** in this tutorial. The next ones take about 20 seconds.

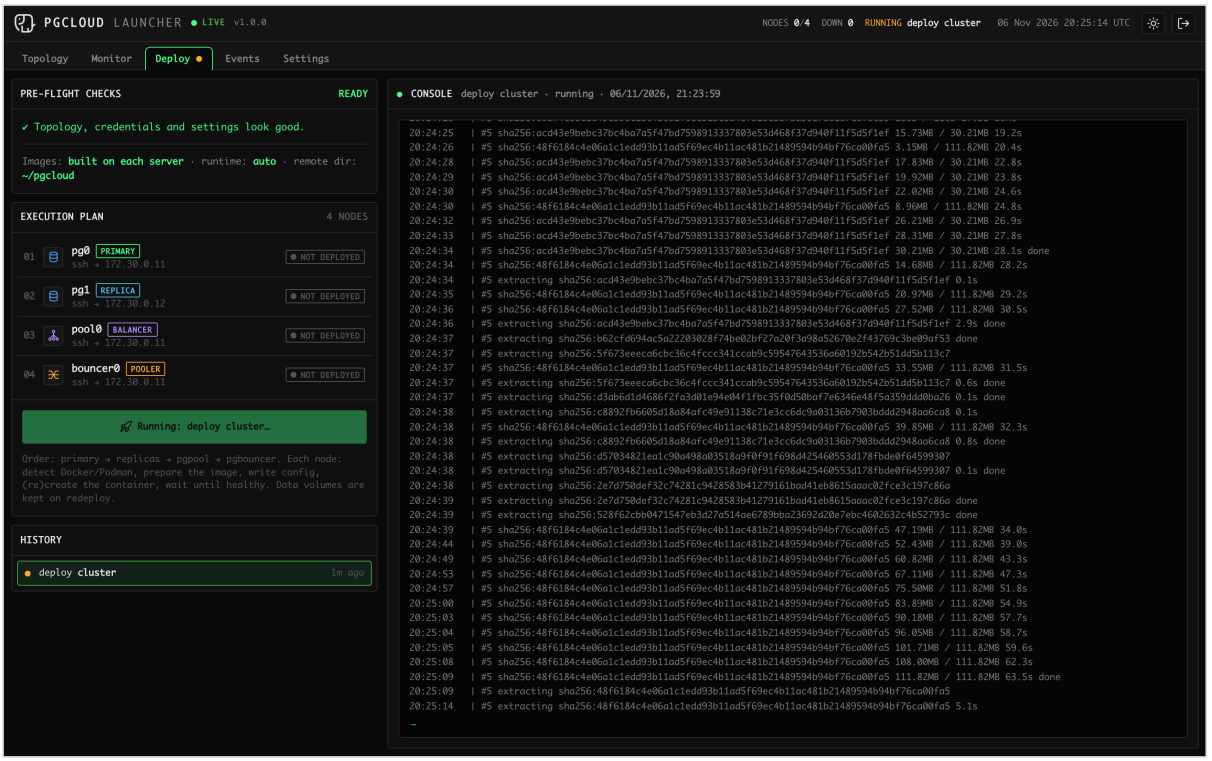


Figure 5.10 - The console while the images are built on the servers.

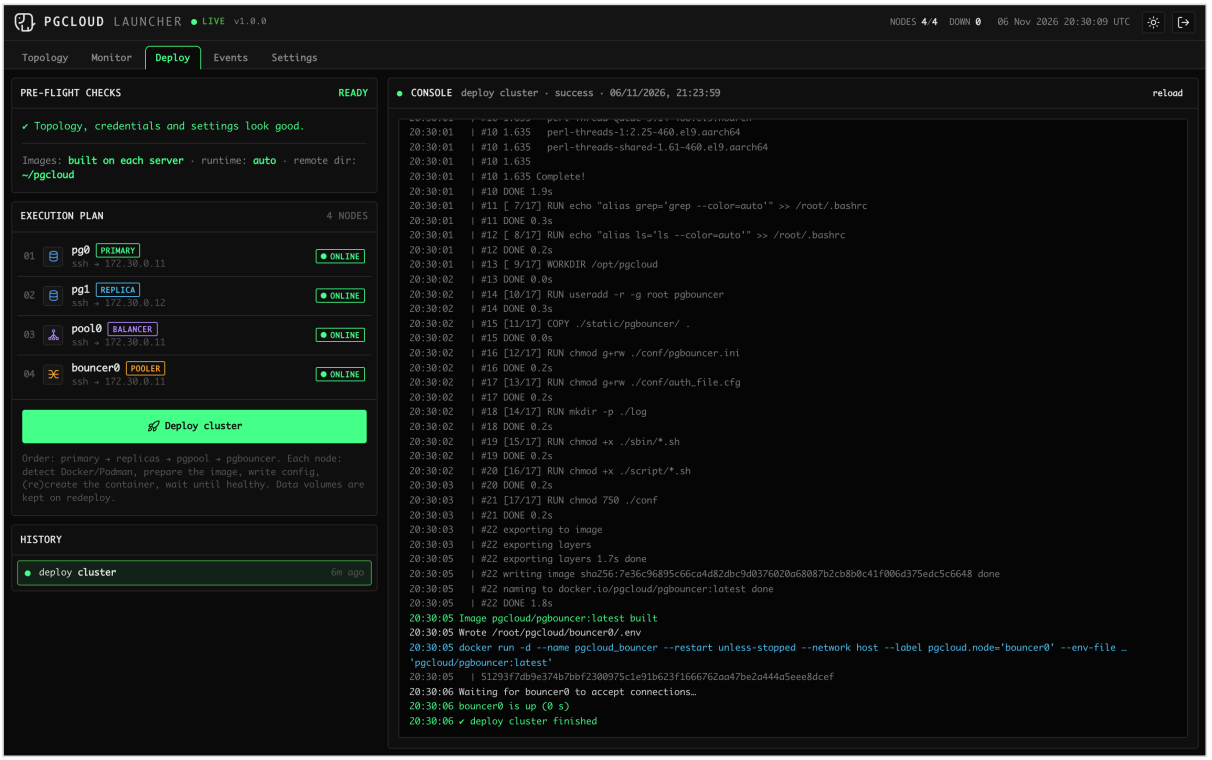


Figure 5.11 - Deployment finished: every node is online.

5.8 - Step 7: watch it live

Your cluster is running. Connect your application to the PgBouncer node (the connection string is in **Settings → Connect your application**). In the screenshots below a benchmark was writing through PgBouncer.

In **Topology** the nodes now show live numbers and the links are animated. A dot travels along each link while both sides are up. The replication link shows **SYNC** because the replica confirms every commit.

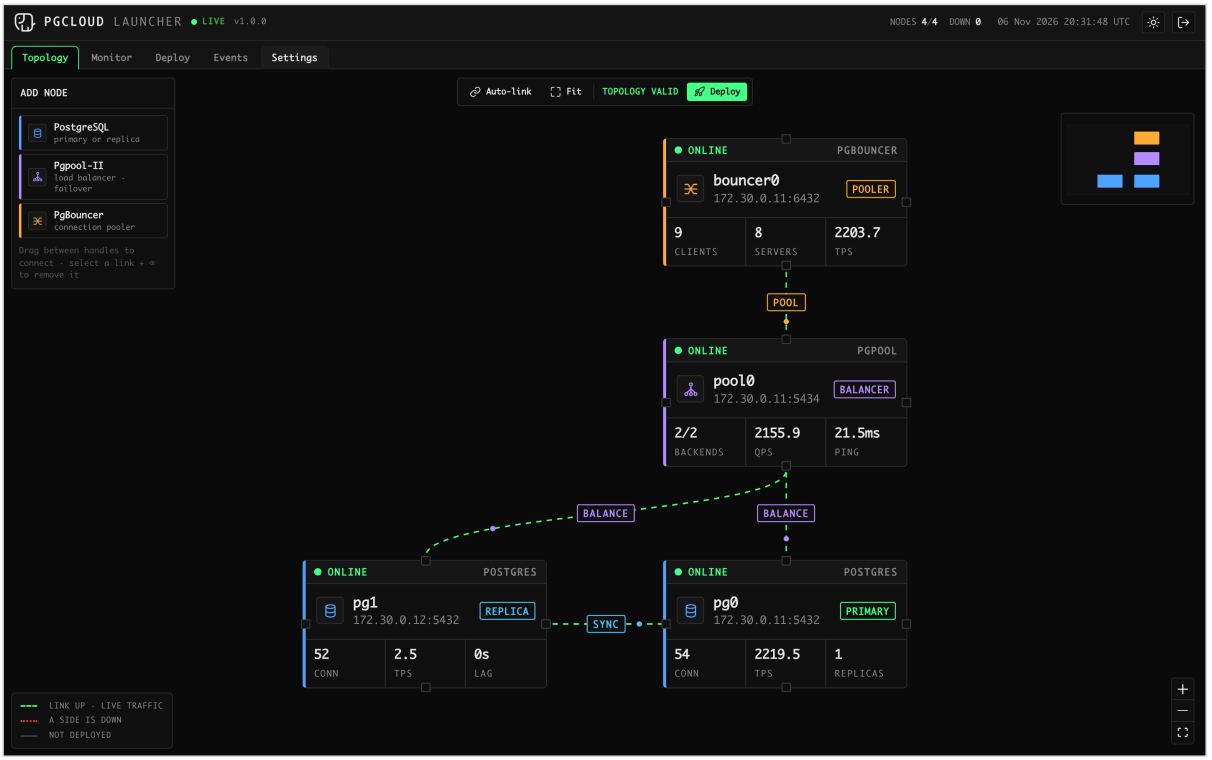


Figure 5.12 - The live topology under load: clients, TPS, backends, replication.

The **Live** tab of a node shows all its metrics and tables. **Container logs** opens the last lines of the container without logging in to the server.

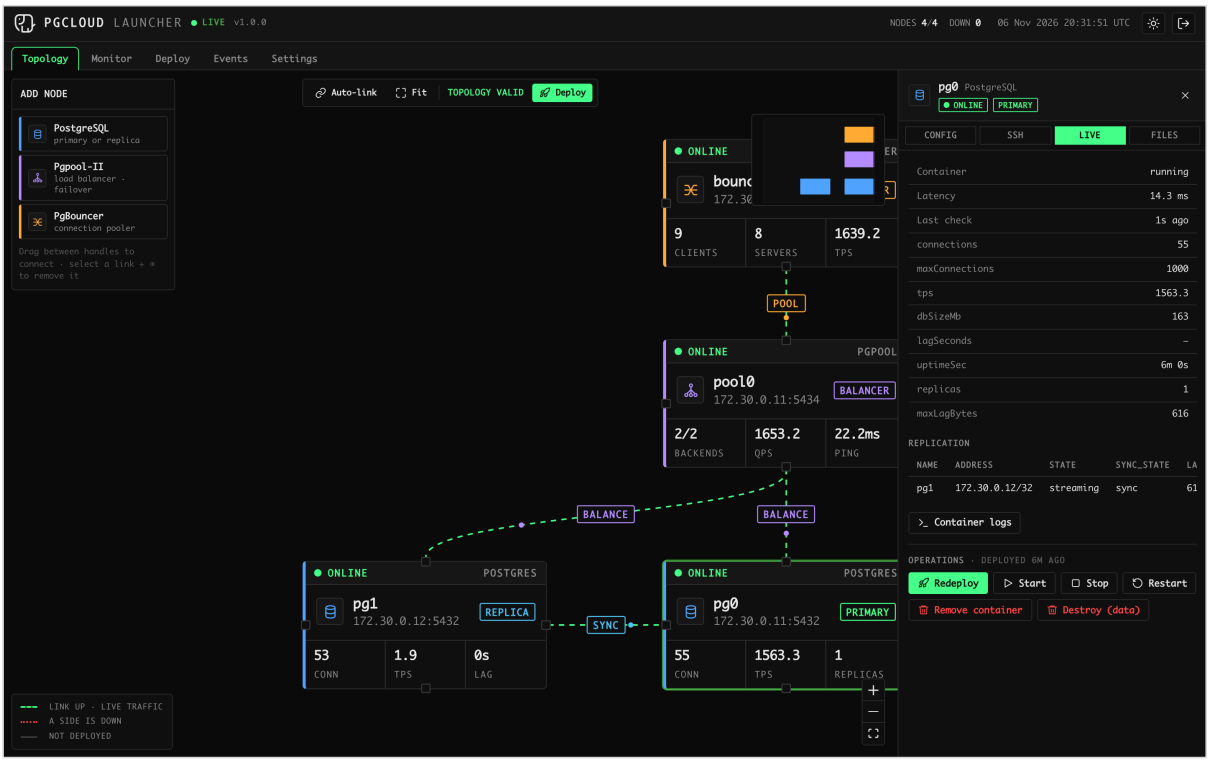


Figure 5.13 - Live metrics of the primary, including its replicas.

Monitor puts everything together: cluster status, a card with charts per node, the replication table of the primary, the view of Pgpool-II and the pools of PgBouncer.

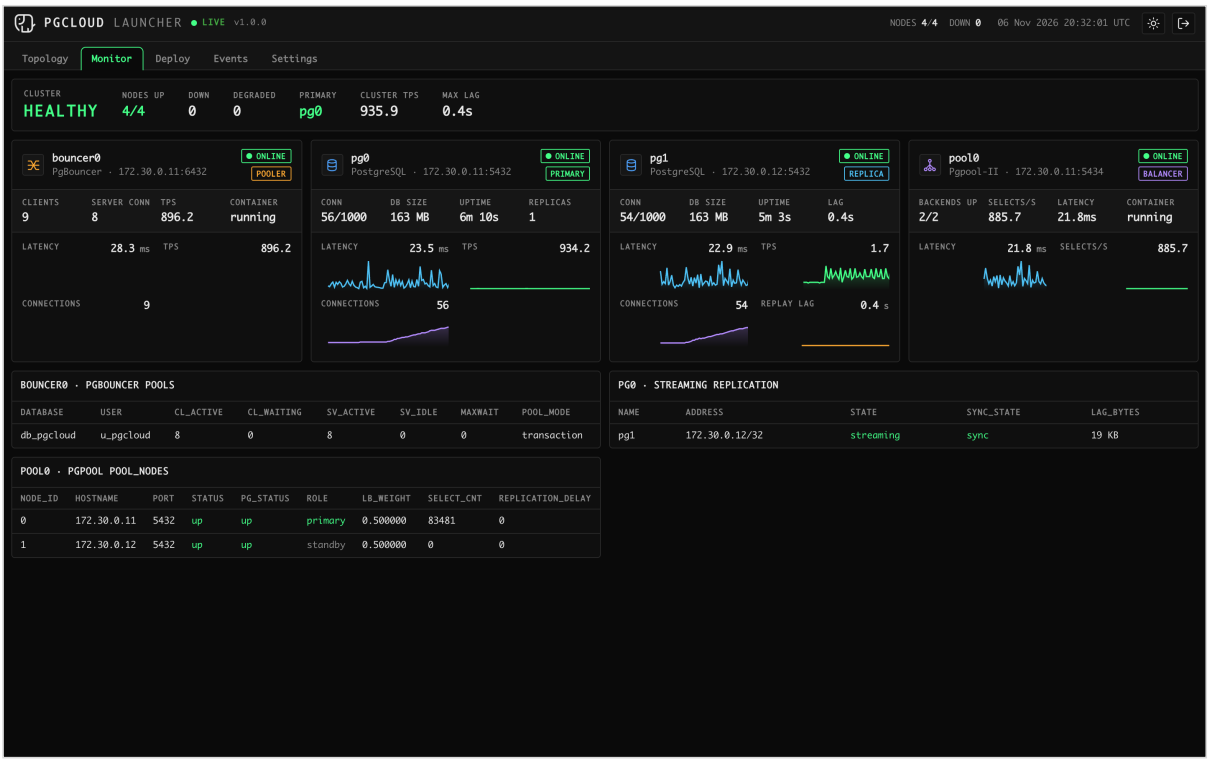


Figure 5.14 - The Monitor tab.

Prefer a light screen? The sun icon in the top bar switches the theme; the choice is remembered.

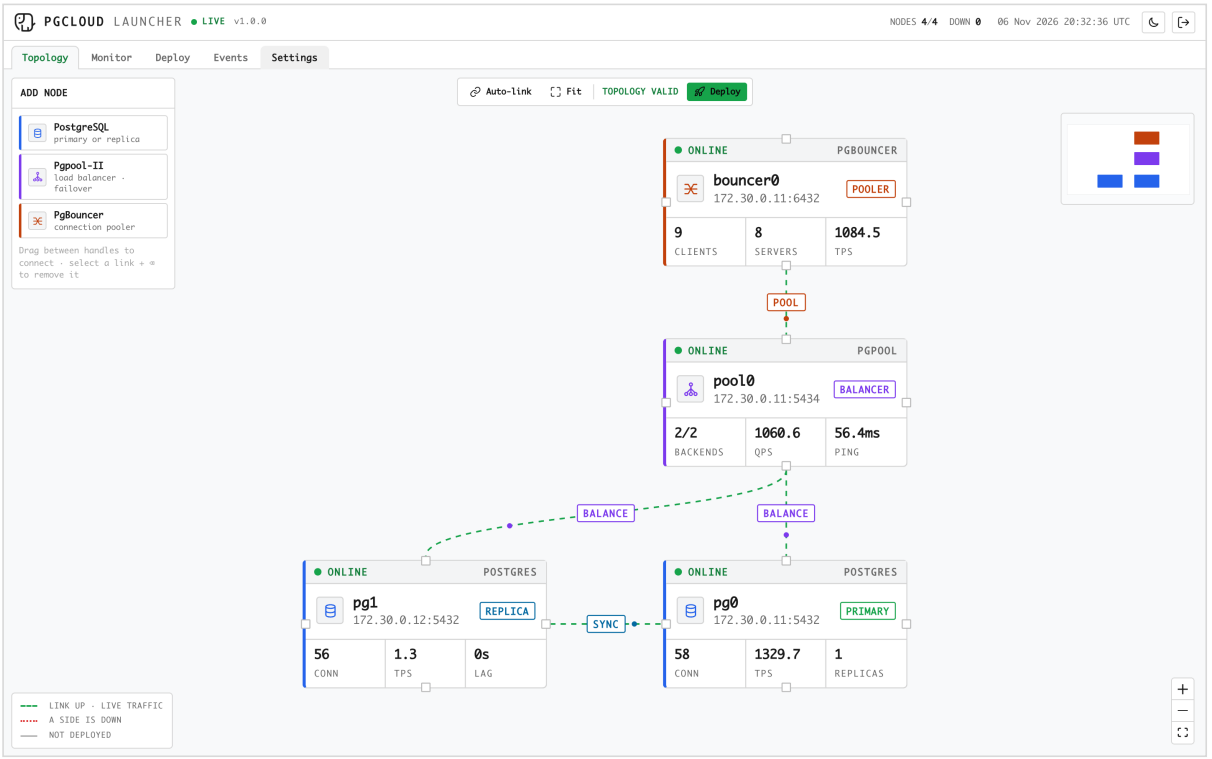


Figure 5.15 - The same cluster in light mode.

5.9 - Step 8: break it on purpose

High availability is only real if it has been tested. Stop the primary, as if its server had crashed:

```
$ docker stop pgcloud_db          # on srv-a, the server of the primary
```

Within seconds the launcher paints the primary red and the links that depend on it turn into dotted red lines. Pgpool-II shows DEGRADED for a moment: it is running the failover.

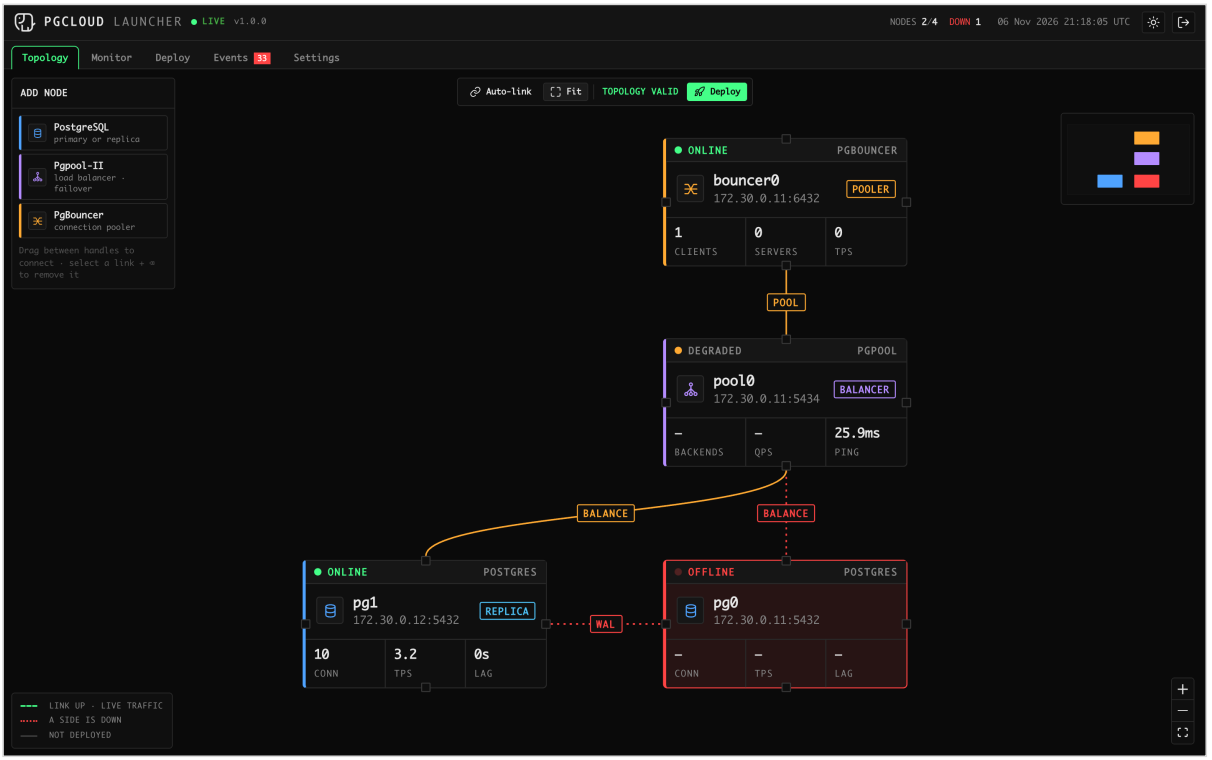


Figure 5.16 - Two seconds after the crash: pg0 offline, Pgpool-II promoting pg1.

You do not have to do anything else. Pgpool-II promotes the replica, rewinds the old primary with pg_rewind and starts it again as a **replica** of the new primary. In this test the whole repair took about three seconds. The application never changed its connection string.

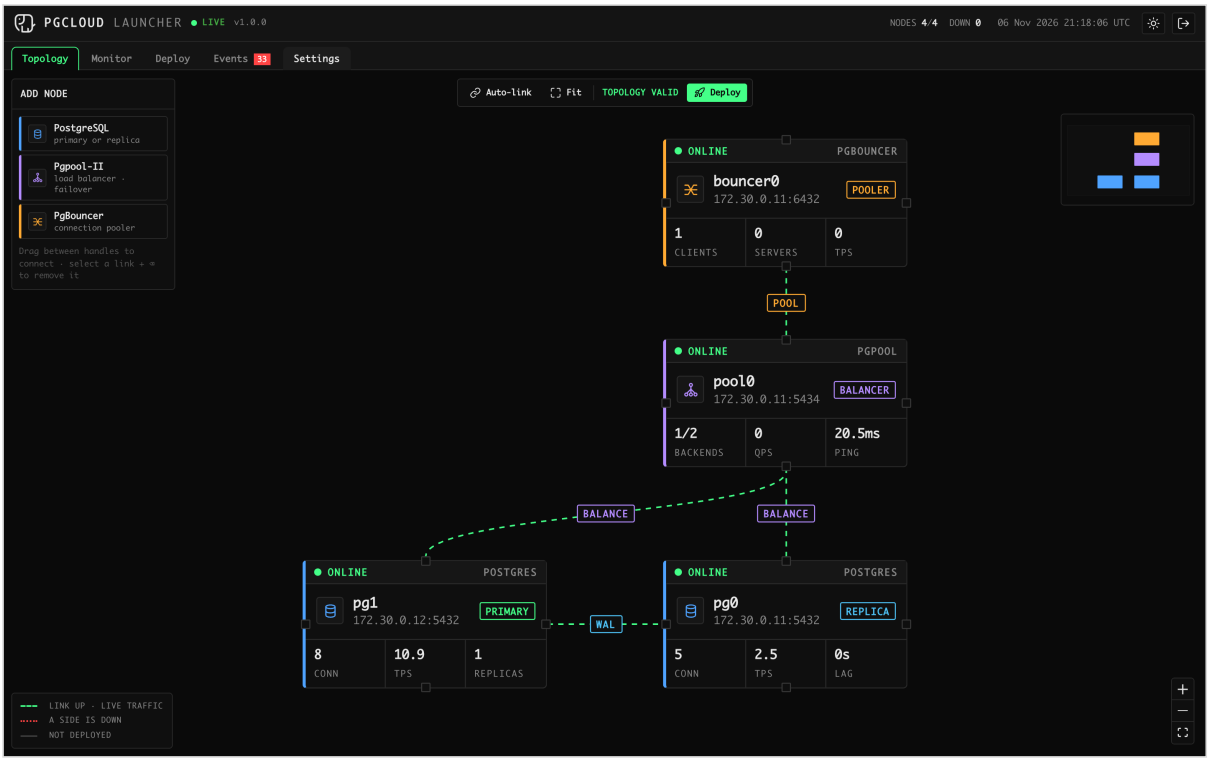


Figure 5.17 - Three seconds later: pg1 is the primary and pg0 is back as its replica.

Events tells the whole story with timestamps: pg0 down, pg1 promoted, pg0 back as a replica, Pgpool-II up again.

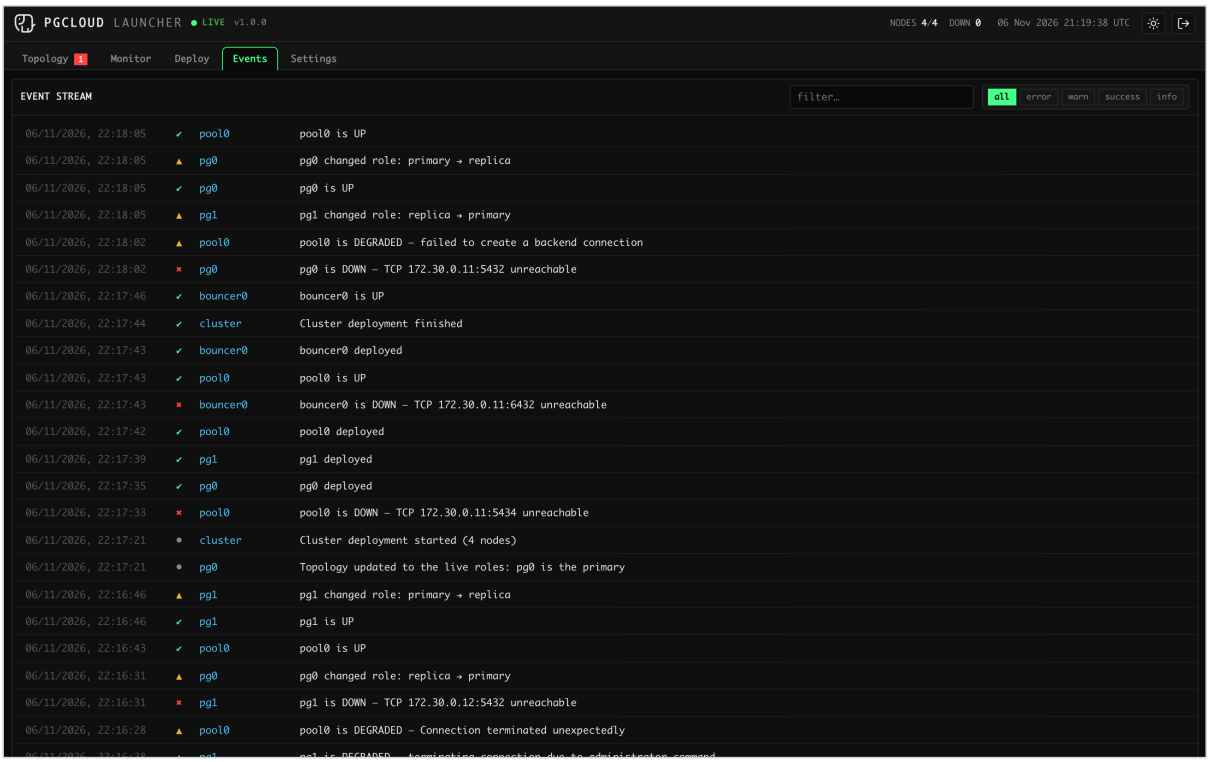


Figure 5.18 - Events: node down, promotion, recovery, in three seconds.

Notice the red 1 on the Topology tab: your drawing still says that pg0 is the primary, but now pg1 is. The launcher refuses to deploy until they match, because deploying the old drawing would turn pg0 into a primary again. Open Topology and press **Adopt live roles**: the replication link is redrawn to the new primary. A deploy afterwards also turns synchronous replication back on.

5.10 - Everyday operations

Everything else is in the panel of each node (Operations):

Button	What it does
Deploy / Redeploy	Writes the configuration again and recreates the container. Data is kept. Pgpool-II is paused meanwhile, so planned work never triggers a failover.
Start / Stop / Restart	Controls the container and waits until the node is healthy again.
Remove container	Deletes the container but keeps its data and configuration.
Destroy (data)	Deletes the container, its data volume and its configuration. You must type the node name.
Delete node	Removes the node from the drawing only; its container keeps running.

Button	What it does
Make primary	On a replica: planned switchover, it becomes the primary and the primary its replica.
Make watchdog leader	On a Pgpool-II node (two or more deployed): it becomes the watchdog leader.
Re-attach	On a Pgpool-II node, shown when Pgpool-II keeps a running PostgreSQL node detached (for example after all the servers restarted): attaches it again.

Move the primary on purpose. Select a running replica: the High availability box offers Make primary. It is a planned switchover: the launcher stops the primary cleanly, waits until the replica has every transaction, promotes it, makes the old primary its replica and updates the drawing and Pgpool-II. No committed transaction is lost; writes stop for about a minute.

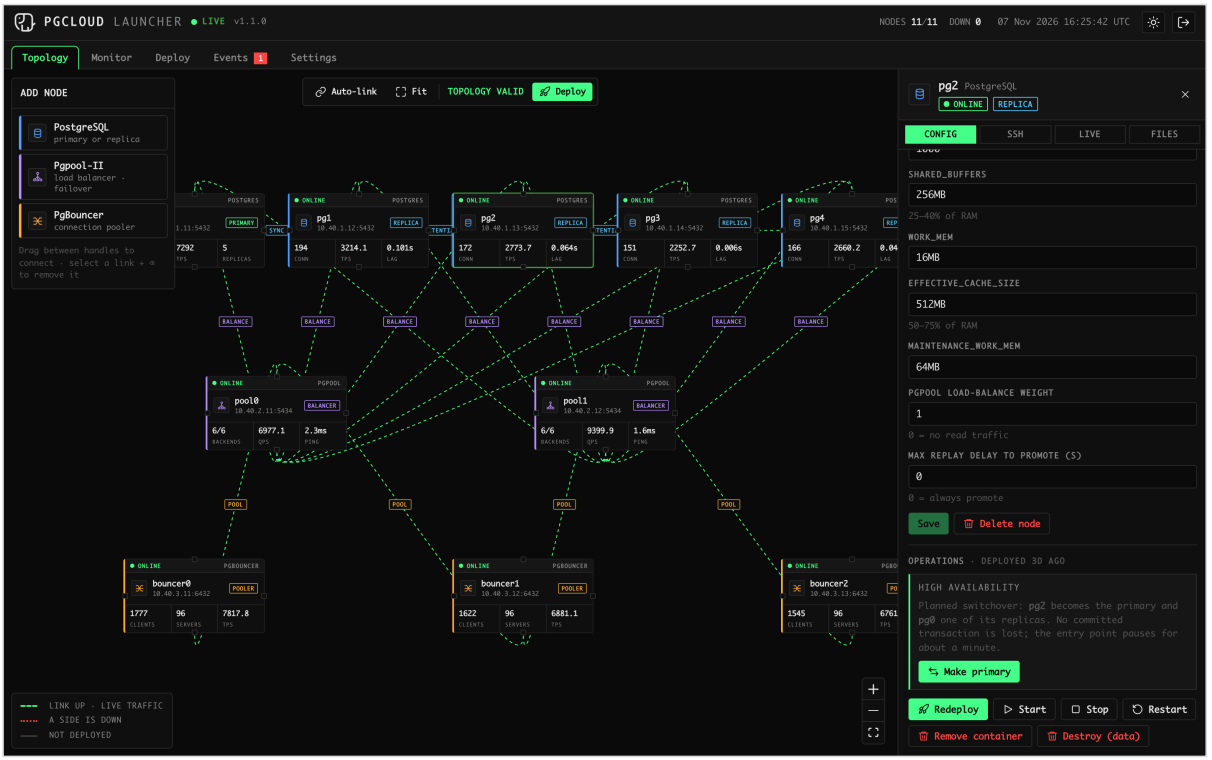


Figure 5.19 - A replica (pg2) selected: the High availability box offers "Make primary".

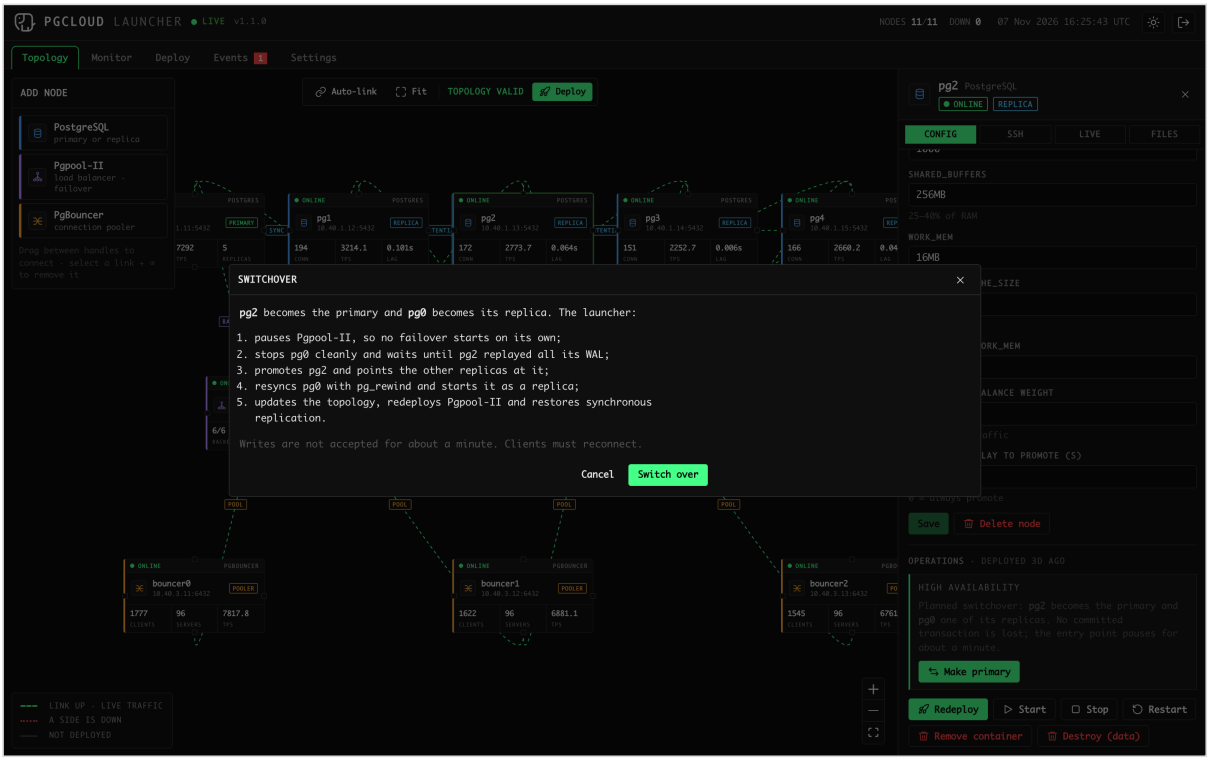


Figure 5.20 - The confirmation lists every step before anything happens.

The progress appears in the Deploy console, like any operation. When it finishes the drawing already shows the new primary, so there is nothing to adopt:

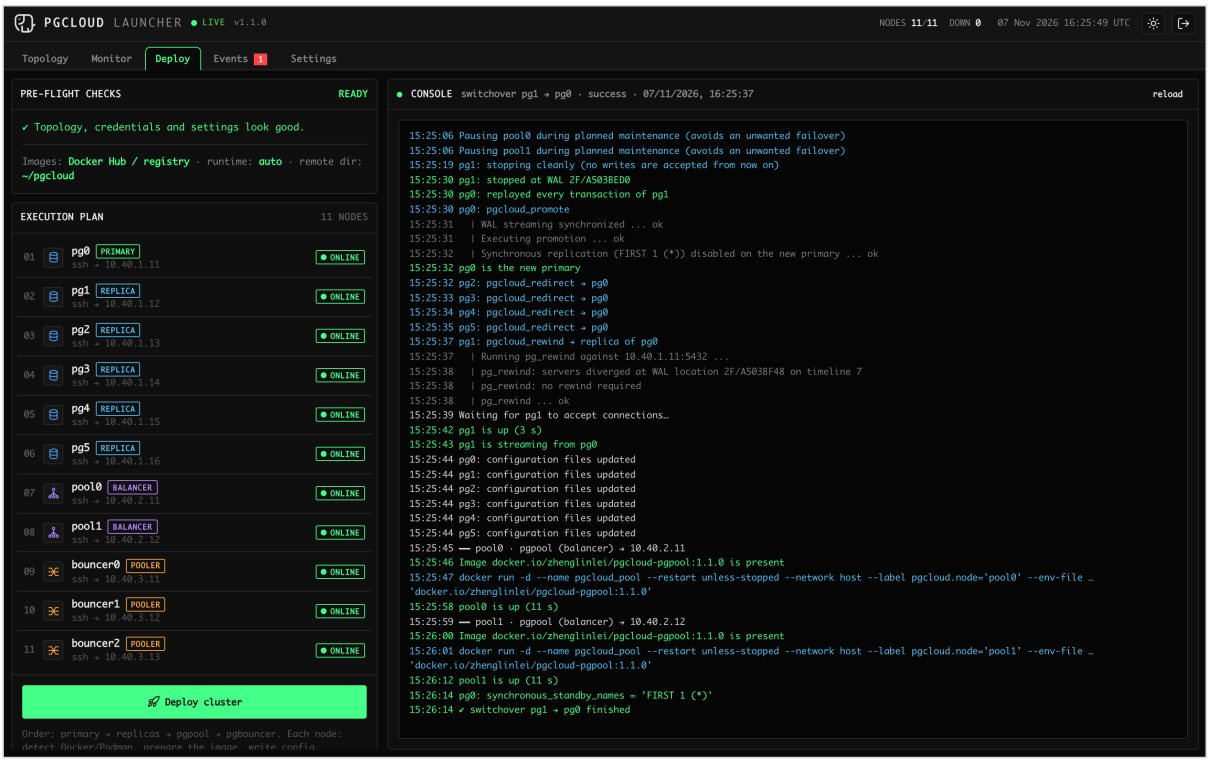


Figure 5.21 - Console of a switchover: clean stop, promotion, redirect, rewind, Pgpool-II redeployed.

Choose the Pgpool-II leader. With two or more Pgpool-II nodes, their watchdog elects a leader. The inspector of a Pgpool-II node shows LEADER or STANDBY; Make watchdog leader hands the role to that node.

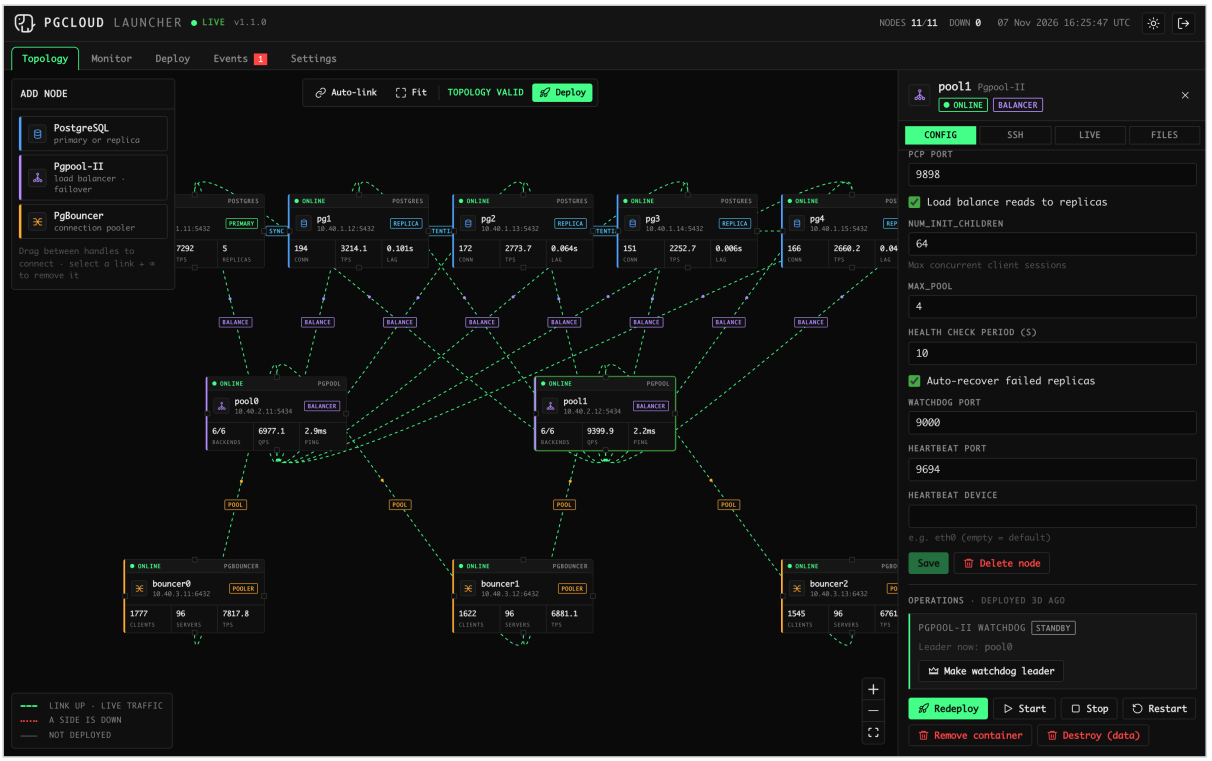


Figure 5.22 - pool1 is STANDBY (leader: pool0): "Make watchdog leader" moves the role.

Every operation is recorded in Deploy → History with its full console log:

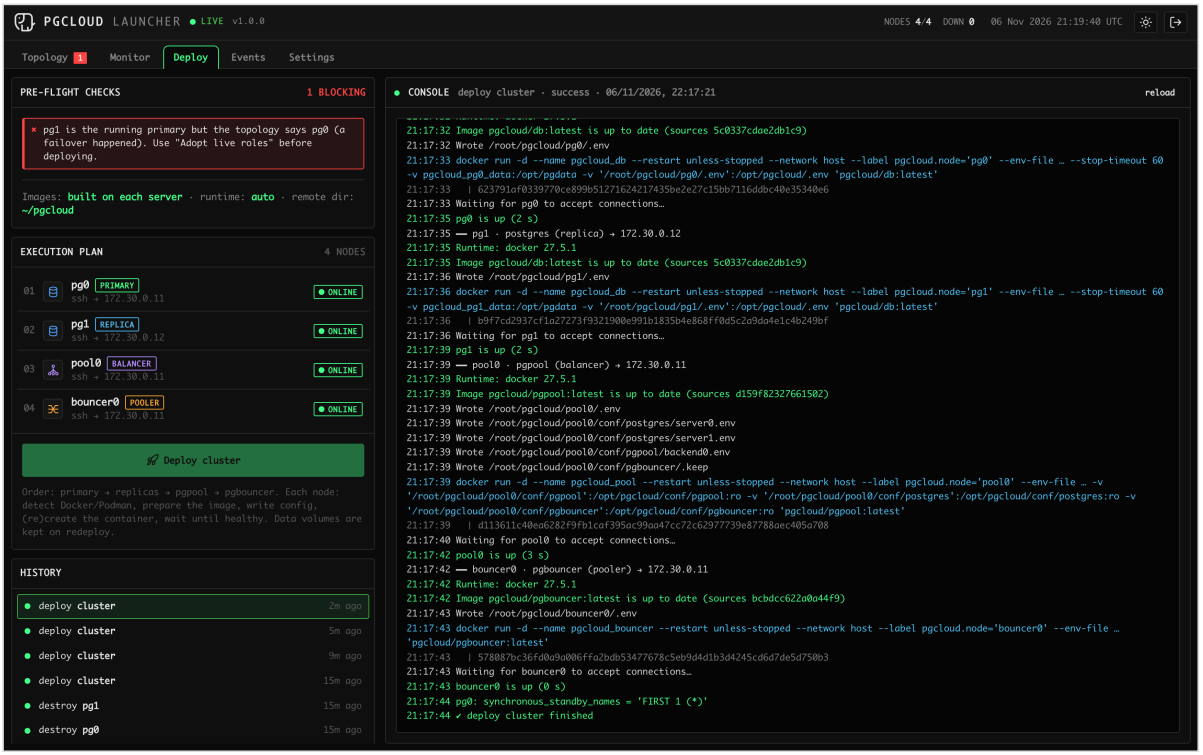


Figure 5.23 - Deployment history with the console of each operation.

5.11 - Try it on your laptop

No servers? The repository includes a playground: the launcher plus three fake servers (Docker containers that run Docker and SSH, so the launcher treats them like real machines). It is the environment used for this tutorial.

```
$ ./RUN.dev.sh
# open http://localhost:8080 (password launcher-test)
# servers: 172.30.0.11, 172.30.0.12 and 172.30.0.13, user root, password pgcloud-test
$ ./RUN.dev.sh --reset          # when you finish: deletes the playground
```

If port 8080 is already in use, run `./RUN.dev.sh -p 8081`. `./RUN.dev.sh --servers` starts only the three servers, for a launcher you already run. All the details are in `tools/pgcloud-launcher/test/README.md`.

> **Congratulations!** You deployed a three-tier PostgreSQL cluster with automatic failover. Section 4 explains every option in depth and section 6 shows how it behaved under tests and benchmarks.

6. Tests and Benchmarks

Before calling PgCloud 1.0.0 production-ready, and again for 1.1.0, every feature described in this book was tested on a real deployment: first a single-host cluster with Docker Compose, then a two-server cluster deployed by the launcher, loaded with pgbench and broken on purpose. This chapter records what was tested, how the system behaved, the numbers measured and the defects found and fixed along the way.

6.1 - Test environment

Item	Value
Host machine	Apple MacBook Pro 13" (Apple M2, 8 cores, 8 GB RAM), macOS 26.6
Container engine	Docker Desktop, Docker Engine 23.0.5 (Linux VM with 8 vCPU and 7.8 GB RAM, shared by every container)
"Servers"	Two Docker-in-Docker containers with OpenSSH (`srv-a` 172.30.0.11, `srv-b` 172.30.0.12), Docker 27.5.1, Alpine Linux 3.21, on a private bridge network
Topology	`srv-a`: PostgreSQL primary `pg0`, Pgpool-II `pool0`, PgBouncer `bouncer0`; `srv-b`: replica `pg1`. Synchronous replication (`FIRST 1 (*)`)
Versions	PostgreSQL 16.15, Pgpool-II 4.3.7, PgBouncer 1.23.1, PgCloud Launcher 1.0.0 (Node.js 22)
Deployment	PgCloud Launcher, image source "build" (images built on each server)
Load generator	pgbench 16 in a container on the same network, scale factor 10 (1,000,000 accounts)

> **Versions.** The benchmarks and the three failover runs below were measured with PostgreSQL 16.15 and Pgpool-II 4.3.7. Releases 1.0.0 and 1.1.0 ship **PostgreSQL 18.6** (supported until November 2030) and **Pgpool-II 4.7.3**. That combination was verified again on three servers (`srv-c` 172.30.0.13 added): the launcher deployed one primary, two replicas, Pgpool-II and PgBouncer in 4 min 59 s; replication was `sync` / `potential`; after killing the primary, `pg1` was promoted, `pg2` followed it, and the old primary came back as a replica with `pg_rewind` (309 MB copied). Writes through PgBouncer never failed and the three nodes ended with the same rows.

> **Read the numbers with care.** All "servers" share one laptop and its disk, and Docker Desktop adds a virtualisation layer. Absolute values are far below real hardware; what matters is the comparison between the three entry points and the behaviour during failures.

6.2 - Functional tests

Each row is a test that was executed and checked by hand or with a script. "Pass" means the observed behaviour matched the expected one in the final code (tests 29 to 39: release 1.1.0).

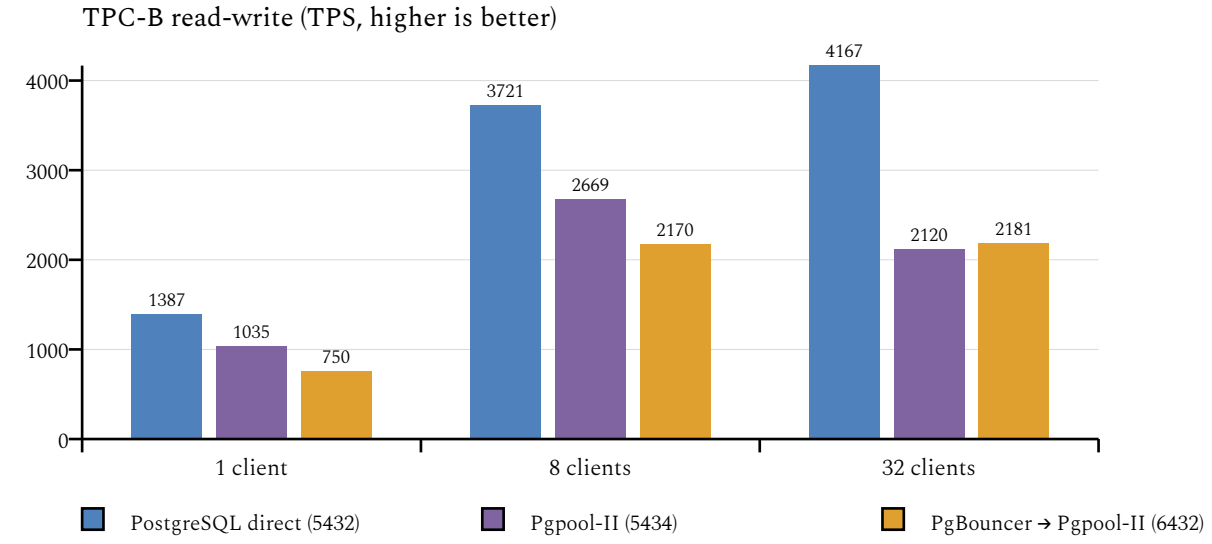
#	Test	Expected behaviour	Result
1	Compose: `docker compose up` (3-tier)	Primary, 2 replicas, Pgpool-II and PgBouncer healthy	Pass
2	Replication slots	Each replica has its own slot `pgcloud<id>` on the primary	Pass
3	Synchronous replication	One replica `sync`, the rest `potential`	Pass
4	Write through PgBouncer → Pgpool-II	Row written on the primary and visible on the replicas	Pass
5	50 consecutive connections through PgBouncer	50 / 50 succeed	Pass
6	Wrong password	Rejected (`SASL authentication failed`)	Pass
7	Placeholder passwords (`change_me`)	Containers refuse to start	Pass
8	`pg_hba.conf` from `ALLOWED_CIDRS`	Only the listed networks; no `0.0.0.0/0`	Pass
9	Pgpool-II restart	Configuration not duplicated (lock file)	Pass
10	PCP tools (`attach.sh`, `pcp_node_info`)	Authenticated, correct node state	Pass
11	Manual `pgcloud_promote` + `pgcloud_redirect`	Replica promoted; other replica streams from it	Pass
12	Launcher: SSH test	Connects, detects Docker 27.5.1 / Alpine 3.21	Pass
13	Launcher: validation	Blocks 2 primaries, missing links/SSH, port clashes, 2 nodes of a type per host	Pass
14	Launcher: tar.gz image source	Archive uploaded over SFTP and loaded (672 MB archive: SFTP upload 2 min 13 s to 2 min 30 s and docker load 3 min 35 s to 4 min 3 s per server)	Pass
15	Launcher: build image source	Images built on each server; reused when sources unchanged (first deploy 6 min 7 s: db 1 min 47 s and 59 s, pgpool 42 s, pgbouncer 2 min 20 s; source change: 6 s)	Pass
16	Launcher: deploy order and health	primary → replica → Pgpool-II → PgBouncer, each waits until healthy	Pass
17	Launcher: redeploy with images present	4 nodes in 18 to 32 s (63 s right after a heavy benchmark, when PostgreSQL needed a long shutdown checkpoint)	Pass
18	Launcher: restart of the launcher	Topology, settings, archives and history persist	Pass
19	Launcher: live monitoring	Status, latency, TPS, lag, pool tables update every 3 s	Pass

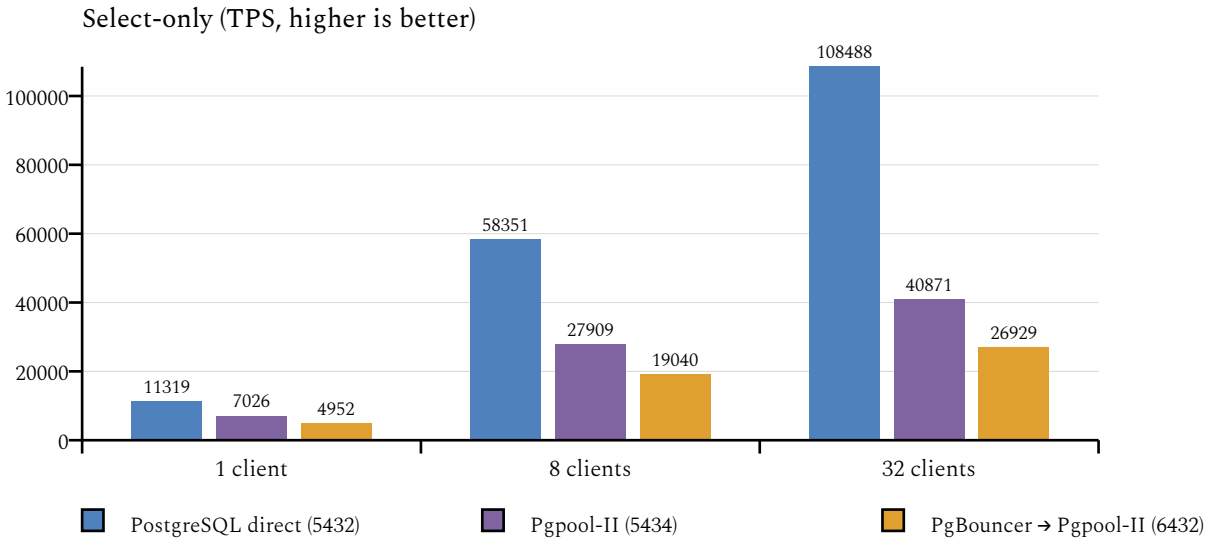
#	Test	Expected behaviour	Result
20	Secrets	Never returned by the API; masked in files preview and logs	Pass
21	Automatic failover (primary killed under load)	Promotion, rewind, re-attach; no split-brain (section 6.4)	Pass
22	Writes after failover	Accepted and replicated to the rejoined node	Pass
23	Entry points during failover	Pgpool-II and PgBouncer keep answering	Pass
24	Planned redeploy of the cluster	Pgpool-II paused, no failover, replica still `sync`	Pass
25	Deploy after a failover	Blocked until "Adopt live roles"; then roles redrawn and deploy OK	Pass
26	Old primary with diverged, recycled WAL	`pg_rewind` fails, node re-cloned with `pg_basebackup`, streams	Pass
27	Synchronous replication after failover	Off after promotion, back to `sync` on the next deploy	Pass
28	Changed PgCloud sources	Only the affected image is rebuilt on each server	Pass
29	Switchover from the launcher (3 runs)	Roles swapped, replicas follow, Pgpool-II and topology updated, `sync` again	Pass
30	Switchover under write load	392 commits through PgBouncer, 0 missing on the new primary	Pass
31	Restart of a promoted node	Starts as primary (role read from its `.env`), no rejoin attempt	Pass
32	Watchdog leader change (twice)	Chosen Pgpool-II becomes LEADER in 21 s and 32 s	Pass
33	Password starting with `-`	`pool_passwd` and PCP hash written (`--` before the password)	Pass
34	Root scripts	`BUILD.sh`, `EXPORT.sh tar` / `hub --local`, `RUN.sh`, `RUN.dev.sh`, `CLEAN.sh -n`	Pass
35	Deploy one Pgpool-II node (3 deployed)	All three redeployed; watchdog of 3 members with quorum	Pass
36	Configuration changed after deploy	Warning on the node; gone when the change is reverted	Pass
37	`docker stop` + `docker start` of Pgpool-II	Stops in 1 s (was > 10 s, killed); starts again, no restart loop	Pass
38	Pgpool-II keeps a running node detached	Warning after 8 s; Re-attach (PCP) brings the primary back	Pass
39	Replica detached by Pgpool-II	Attached again automatically by the failover script (about 20 s)	Pass

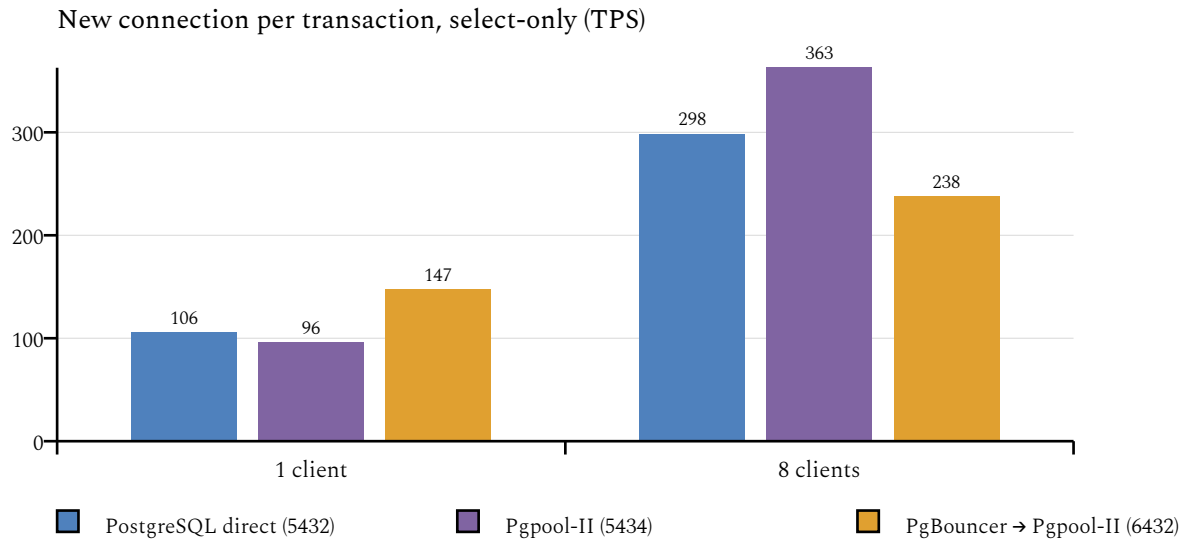
6.3 - Benchmarks

pgbench ran 20 seconds per configuration against the three entry points of the same cluster: PostgreSQL directly, Pgpool-II, and PgBouncer (which forwards to Pgpool-II, as in a three-tier deployment). Three workloads were measured:

- TPC-B (pgbench default): read-write transactions. Every commit waits for the synchronous replica.
- Select-only (`-S`): one primary-key lookup per transaction. Pgpool-II may balance it to the replica.
- New connection per transaction (`-S -C`): the client opens and closes a connection for each transaction, like many web applications without a pool. This is where a pooler should shine.







Workload	Clients	Direct TPS	Pgpool-II TPS	PgBouncer TPS	Latency direct / pool / bouncer (ms)
TPC-B	1	1,387	1,035	750	0.72 / 0.97 / 1.33
TPC-B	8	3,721	2,669	2,170	2.15 / 3.00 / 3.69
TPC-B	32	4,167	2,120	2,181	7.68 / 15.10 / 14.67
Select-only	1	11,319	7,026	4,952	0.09 / 0.14 / 0.20
Select-only	8	58,351	27,909	19,040	0.14 / 0.29 / 0.42
Select-only	32	108,488	40,871	26,929	0.29 / 0.78 / 1.19
New conn. / tx	1	106	96	147	9.45 / 10.41 / 6.79
New conn. / tx	8	298	363	238	26.81 / 22.05 / 33.63

What the numbers say.

- **Every hop costs latency.** With one client, a TPC-B transaction takes 0.72 ms directly, 0.97 ms through Pgpool-II and 1.33 ms through PgBouncer → Pgpool-II. Poolers do not make a query faster: they make the database survive many clients and connections.
- **Pgpool-II is the throughput bottleneck on a single machine.** Directly, TPC-B grows to 4,167 TPS with 32 clients; through Pgpool-II it stays around 2,100–2,700 TPS, and read-only queries reach 108,488 TPS directly against 40,871 through Pgpool-II. Pgpool-II parses every statement to route it, and here it competed for the same 8 CPU cores as both databases.
- **Read balancing needs real servers.** Pgpool-II did send read-only queries to the replica, but on one laptop the replica uses the same CPUs as the primary, so it cannot add capacity. On separate servers each replica adds its own CPUs for reads.

- **Connection pooling pays off when the application connects often.** Opening a new connection per transaction costs about 9.5 ms here (SCRAM authentication). PgBouncer raised that workload from 106 to 147 TPS with one client (+39 %), and Pgpool-II, which keeps backend connections cached, from 298 to 363 TPS with eight clients (+22 %). Applications that keep their own pool get little from an extra pooler.
- **Synchronous replication is included in every write number.** Each commit waits until the replica confirms it, which is the price of losing no committed transaction in a failover.

6.4 - Failover under load

With `pgbench` writing through PgBouncer, the container of the current primary was stopped (`docker stop pgcloud_db`) at t = 0. A probe sampled the cluster about twice per second: the role of both databases, a write through PgBouncer and the state of both backends in Pgpool-II. The test was repeated three times, alternating the primary, after the fixes of section 6.5.

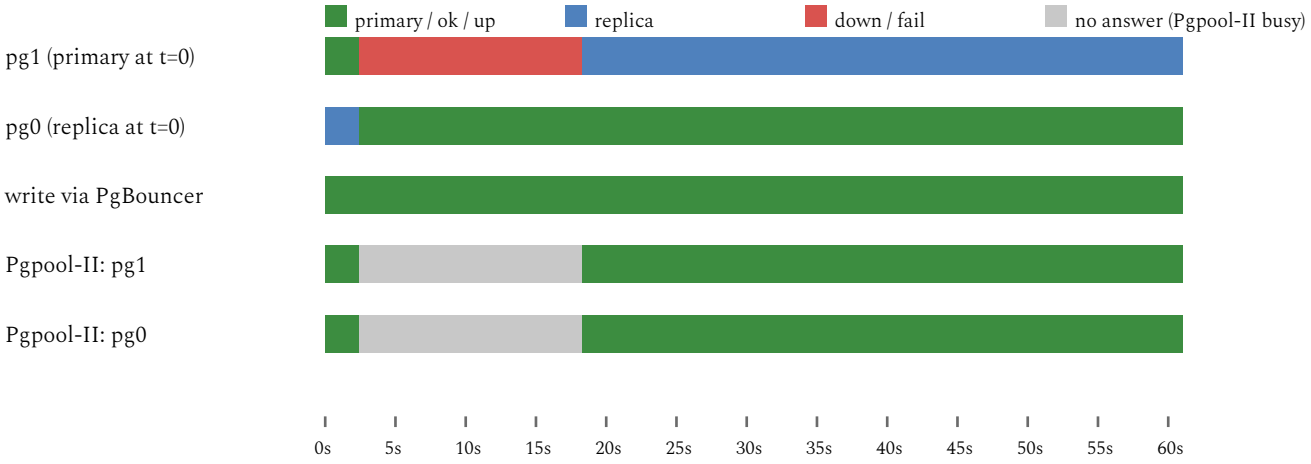


Figure 6.1 - Run 1: pg1 (primary) stopped at t = 0.

Milestone (time after the crash)	Run 1: pg1 → pg0	Run 2: pg0 → pg1	Run 3: pg0 → pg1
New primary accepts writes (≤)	2.4 s	3.4 s	5.0 s
Old primary back as a replica (`pg_rewind`)	18.3 s	22.3 s	35.9 s
Old primary attached again in Pgpool-II	18.3 s	22.3 s	36.8 s
Moments with two primaries	none	none	none
Failed writes through PgBouncer	0 of 69 probes	2 of 62 probes	0 of 17 probes

In the three runs the new primary accepted writes within 2.4 to 5 seconds and the old primary came back as a streaming replica after `pg_rewind` in 18 to 36 seconds (the longer runs had to copy more changed data: 357 MB in run 2). Pgpool-II attached it again right after. At no moment were there two primaries. PgBouncer kept accepting writes: the only failed probes

were two samples of run 2, during the switch and during the re-attach, when Pgpool-II restarts its child processes. Clients connected to the failed primary lose their connection (pgbench aborts its run), exactly as with a single server. A fourth run, recorded with screenshots in section 5.9, completed the whole repair in about three seconds.

6.5 - Behaviour observed and defects fixed

Testing on real deployments found defects that unit tests would not have shown. All of them are fixed (the last six in 1.1.0):

Component	Observed behaviour	Fix
Pgpool-II image	Did not build on Ubuntu 24.04 (`netcat` has no candidate)	Package `netcat-openbsd`
Pgpool-II	Exited at start: missing `/var/run/pgpool`	Directory created in the image
Pgpool-II	Ignored node 0: template without final newline glued `backend_hostname0` to a comment	Newline guaranteed
Pgpool-II	Syntax error with IP addresses (`hostname0 = 172.30.0.11` unquoted)	Values quoted
Pgpool-II	SCRAM authentication to backends failed	`pool_passwd` (AES) + `pool_hba` scram
PgBouncer	Could not log in to Pgpool-II (SCRAM pass-through with a foreign salt)	Password in `auth_file`
Failover	Pgpool-II deadlocked: the script queried Pgpool-II while it waited for the script	Status / attach in background
Failover	~45 s with two primaries: `postmaster.pid` deleted under a running server	Clean stop + `pg_rewind`
Failover	Old primary could not follow the new timeline	New `pgcloud_rewind`
Failover	Writes blocked after promotion (synchronous standby missing)	Sync disabled on promotion
PostgreSQL	Empty `SYNCHRONOUS_RULE` produced an invalid `synchronous_standby_names`	Empty value handled
Launcher	Container failed on hosts without `/etc/localtime`	Mounted only if present
Scripts	Logs expanded `********` as a file glob	Quoted output
Images	`pgcloud_*` command symlinks were relative and broken	Absolute symlinks
Launcher	Redeploying the primary made Pgpool-II fail over: two primaries	Pgpool-II paused during planned work
Launcher	Deploying the drawing after a failover would restart the old primary as primary	"Adopt live roles" + check

Component	Observed behaviour	Fix
PostgreSQL	`pg_rewind` failed when the old primary had recycled its WAL	`wal_keep_size` + re-clone fallback
PostgreSQL	A replica-configured node with primary data started as a primary	Self-healing rejoin at start
Launcher	Synchronous replication stayed off after a failover	Restored at the end of a deploy
Pgpool-II	Empty `pool_passwd` (every login rejected) when the password started with `-`	`--` + entry checked
PostgreSQL	A promoted node kept `NODE_ROLE=replica`: a restart could rejoin the old primary	Role kept in `.env`
Launcher	Pgpool-II nodes were paused one by one (slower; a running one could fail over)	Paused all at once
Launcher	A Pgpool-II node added later was rejected by the watchdog and restarted in a loop	All Pgpool-II redeployed
Pgpool-II	After `docker stop` it restarted in a loop ("pid file found"): the entrypoint did not pass the stop signal	`exec` + SIGINT + cleanup
Pgpool-II	After a restart of every server the primary came back later and stayed detached: writes failed	Re-attach + `auto_failback`

The audit before the tests also fixed configuration bugs in the failover scripts (undefined variables, wrong paths, the slot file not being copied), removed `sudo` without password and `chmod 777` from the images, moved secrets out of the repository and stopped printing passwords in the logs.

6.6 - Switchover and watchdog (1.1.0)

The new launcher operations were tested on a second playground with three servers: PostgreSQL `pg0`, `pg1`, `pg2` (one per server), three Pgpool-II nodes with the watchdog enabled and one PgBouncer, all deployed by the launcher with the `build` image source.

Operation	Result	Total time	Writes stopped
Switchover `pg0` → `pg1`	OK, "no rewind required"; Pgpool-II redeployed	2 min 29 s*	≈ 2 min*
Switchover `pg1` → `pg0`	OK; `pg2` sync, `pg1` potential	1 min 22 s	58 s
Switchover `pg0` → `pg1` under load	392 commits, 0 lost, 1 failed attempt	1 min 22 s	≈ 1 min
Watchdog leader → `pool2`	One restart of `pool0`	21 s	–
Watchdog leader → `pool1`	One restart of `pool2`	32 s	–

* First run, before the Pgpool-II nodes were paused all at once instead of one by one.

During the switchover under load a client inserted one row every 0.2 s through PgBouncer and recorded every id the server confirmed. After the switchover all 392 confirmed ids were found on the new primary. Only one insert failed: the others waited in PgBouncer while Pgpool-II was stopped and continued afterwards. Restarting the promoted node with the launcher confirmed the second fix of 1.1.0: although its container was created as a replica, it started as the primary because the role is now read from its `.env`.

6.7 - Known limitations of 1.1.0

- After an automatic failover the new primary replicates **asynchronously** until the cluster is deployed again (section 4.7.3). A switchover from the launcher restores synchronous replication itself.
- A switchover stops writes for about a minute, and it starts a new watchdog election (Pgpool-II is redeployed).
- Planned work on PostgreSQL nodes pauses Pgpool-II: the entry point is unavailable for those seconds.
- Connections open against a failed primary are closed; clients must reconnect (pgbench aborts its run).
- Pgpool-II never attaches a primary by itself: after a restart of every server, check the launcher and press **Re-attach** if it warns about a detached node.
- One node of each type per server; Pgpool-II automatic recovery needs SSH **password** access to the database servers.
- No TLS between nodes and no scheduled backups or WAL archiving yet: use a private network and your own backup tool.

PGCLOUD

Zheng Lin Lei