PostgreSQL

Overview

PostgreSQL is an open-source relational database management system. It supports SQL language and offers features like transactions, referential integrity, and user-defined types and functions. PostgreSQL can handle complex queries, supports various data types, and is extensible, allowing users to add custom functions and data types. It runs on multiple operating systems, including Windows, Linux, and macOS.

Key Features of PostgreSQL:

- **Extensibility:** Users can define custom data types, operators, and index methods, tailoring the database to specific application needs.
- **Standards Compliance:** PostgreSQL conforms to at least 170 of the 177 mandatory features for SQL:2023 Core conformance, ensuring compatibility with SQL standards.
- **Advanced Data Types:** Supports a wide array of data types, including JSON for unstructured data, arrays, hstore (key-value pairs), and geometric types, enhancing its versatility.
- **Concurrency and Performance:** Utilizes Multi-Version Concurrency Control (MVCC) to handle multiple transactions simultaneously without conflicts, ensuring data integrity and performance.
- **Replication and High Availability:** Offers asynchronous replication, allowing data to be copied to standby servers for load balancing and failover support, enhancing reliability.
- **ACID Compliance:** Ensures transactions are processed reliably through Atomicity, Consistency, Isolation, and Durability properties, which are crucial for applications requiring data integrity.
- **Security Features:** Provides robust security mechanisms, including role-based access control, data encryption, and connection security, safeguarding data against unauthorized access.
- **Cross-Platform Support:** Runs on all major operating systems, including Windows, Linux, macOS, FreeBSD, and OpenBSD, offering flexibility in deployment environments.

These features make PostgreSQL a preferred choice for developers and enterprises seeking a reliable, feature-rich, and scalable database solution.

How to Connect

Connecting with Node.js

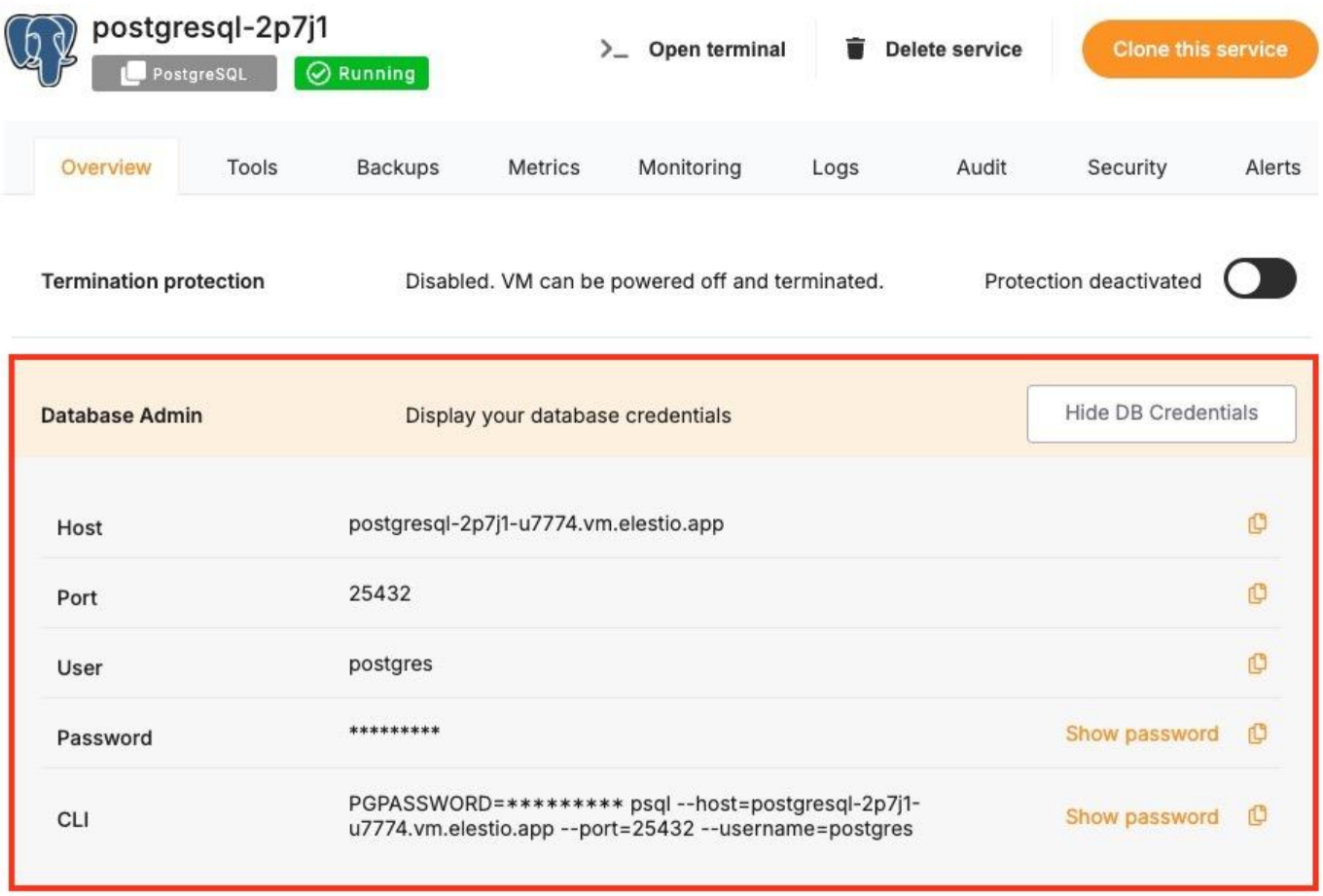
This guide explains how to establish a connection between a Node.js application and a PostgreSQL database using the `pg` package. It walks through the necessary setup, configuration, and execution of a simple SQL query.

Variables

Certain parameters must be provided to establish a successful connection to a PostgreSQL database. Below is a breakdown of each required variable, its purpose, and where to find it. Here's what each variable represents:

Variable	Description	Purpose
`USER`	PostgreSQL username, from the Elestio service overview page	Identifies the database user who has permission to access the PostgreSQL database.
`PASSWORD`	PostgreSQL password, from the Elestio service overview page	The authentication key required for the specified `USER` to access the database
`HOST`	Hostname for PostgreSQL connection, from the Elestio service overview page	The address of the server hosting the PostgreSQL database.
`PORT`	Port for PostgreSQL connection, from the Elestio service overview page	The network port is used to connect to PostgreSQL. The default port is `5432`.
`DATABASE`	Database Name for PostgreSQL connection, from the Elestio service overview page	The name of the database being accessed. A PostgreSQL instance can contain multiple databases.

These values can usually be found in the Elestio service overview details as shown in the image below, make sure to take a copy of these details and add it to the code moving ahead.



Prerequisites

- **Install Node.js and NPM**
  - Check if Node.js is installed by running:
    `node -v`
  - If not installed, download it from nodejs.org and install.
  - Verify npm installation:
    `npm -v`
- **Install the `pg` Package**
  The `pg` package enables Node.js applications to interact with PostgreSQL. Install it using:
  `npm install pg --save`

Code

Once all prerequisites are set up, create a new file named `pg.js` and add the following code:

```
const pg = require("pg");


// Database connection configuration
```

```
const config = {
  user: "USER",
  password: "PASSWORD",
  host: "HOST",
  port: "PORT",
  database: "DATABASE",
};

// Create a new PostgreSQL client
const client = new pg.Client(config);

// Connect to the database
client.connect((err) => {
  if (err) {
    console.error("Connection failed:", err);
    return;
  }
  console.log("Connected to PostgreSQL");

  // Run a test query to check the PostgreSQL version
  client.query("SELECT VERSION()", [], (err, result) => {
    if (err) {
      console.error("Query execution failed:", err);
      client.end();
      return;
    }

    console.log("PostgreSQL Version:", result.rows[0]);

    // Close the database connection
    client.end((err) => {
      if (err) console.error("Error closing connection:", err);
    });
  });
});
```

To execute the script, open the terminal or command prompt and navigate to the directory where `pg.js`. Once in the correct directory, run the script with the command

```
node pg.js
```

If the connection is successful, the terminal will display output similar to:

```
Connected to PostgreSQL
PostgreSQL Version: {
  version: 'PostgreSQL 16.8 (Debian 16.8-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc
(Debian 12.2.0-14) 12.2.0, 64-bit'
}
```

Connecting with Python

This guide explains how to establish a connection between a **Python** application and a **PostgreSQL** database using the `psycopg2-binary` package. It walks through the necessary setup, configuration, and execution of a simple SQL query.

Variables

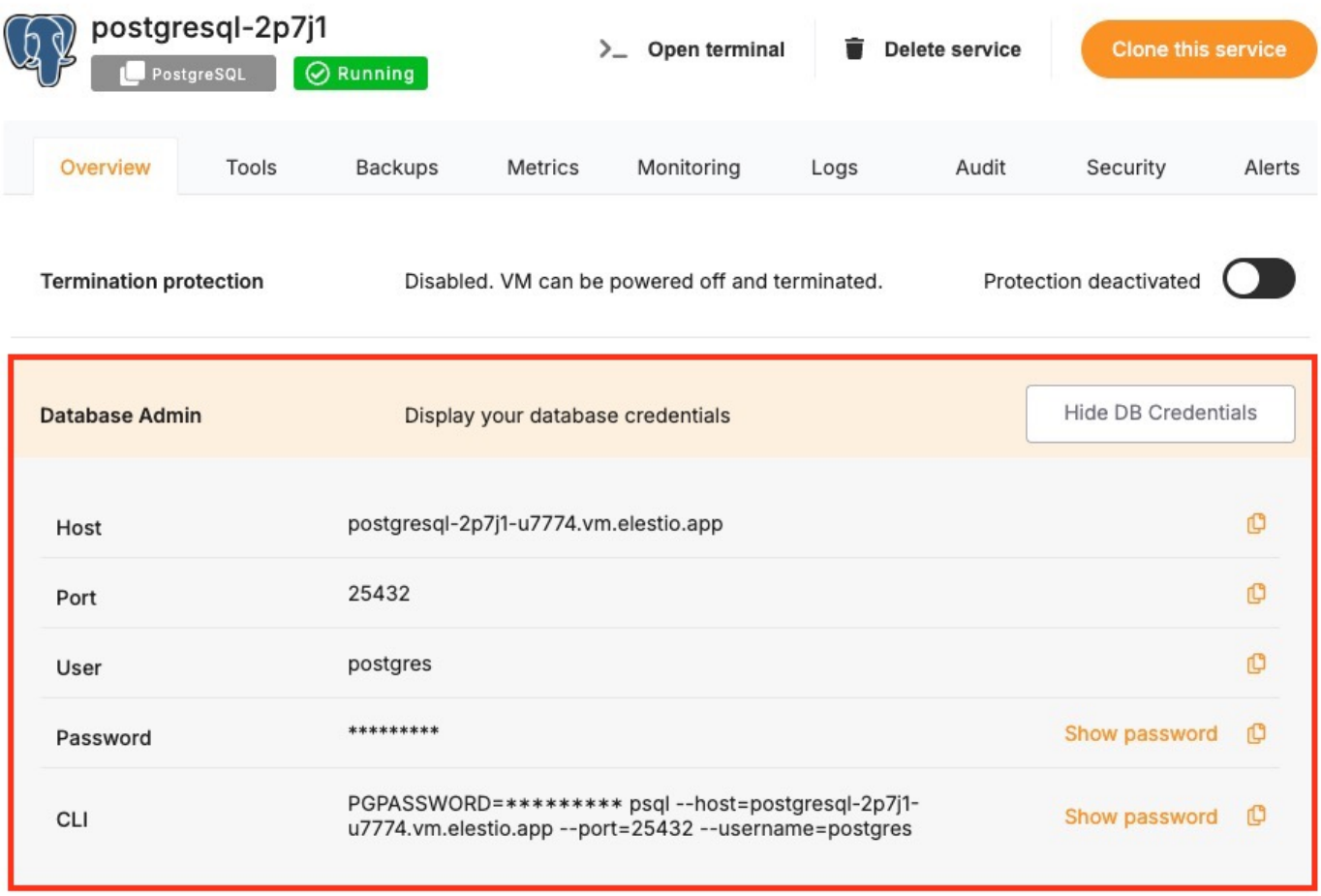
To connect to a PostgreSQL database, you only need one environment variable — the **connection URI**. This URI contains all the necessary information like username, password, host, port, and database name.

Variable	Description	Purpose
POSTGRESQL_URI	Full PostgreSQL connection string (from the Elestio service overview page)	Provides all necessary credentials and endpoint details in a single URI format.

The URI will look like this:

```
postgresql://<USER>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>
```

You can find the details needed in the URI from the **Elestio service overview** details. Copy and replace the variables carefully in the URI example provided above.



Prerequisites

Install Python

Check if Python is installed by running:

```
python --version
```

If not installed, download it from python.org and install it.

Install `psycopg2-binary` Package

The `psycopg2-binary` package enables Python applications to interact with PostgreSQL. Install it using:

```
pip install psycopg2-binary
```

Code

Once all prerequisites are set up, create a new file named `pg.py` and add the following code and replace the `POSTGRESQL_URI` with actual link or in environment setup as you wish:

```
import psycopg2

def get_db_version():
    try:
        db_connection = psycopg2.connect('POSTGRESQL_URI')
        db_cursor = db_connection.cursor()
        db_cursor.execute('SELECT VERSION()')
        db_version = db_cursor.fetchone()[0]
        return db_version

    except Exception as e:
        print(f"Database connection error: {e}")
        return None

    finally:
        if 'db_cursor' in locals():
            db_cursor.close()
        if 'db_connection' in locals():
            db_connection.close()

def display_version():
    version = get_db_version()
    if version:
        print(f"Connected to PostgreSQL: {version}")

if __name__ == "__main__":
    display_version()
```

To execute the script, open the terminal or command prompt and navigate to the directory where `pg.py`. Once in the correct directory, run the script with the command

```
python pg.py
```

If the connection is successful, the terminal will display output similar to:

```
Connected to PostgreSQL: PostgreSQL 16.8 (Debian 16.8-1.pgdg120+1) on x86_64-pc-linux-gnu,
compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
```

Connecting with PHP

This guide explains how to establish a connection between a **PHP** application and a **PostgreSQL** database using the built-in `PDO` extension. It walks through the necessary setup, configuration, and execution of a simple SQL query.

Variables

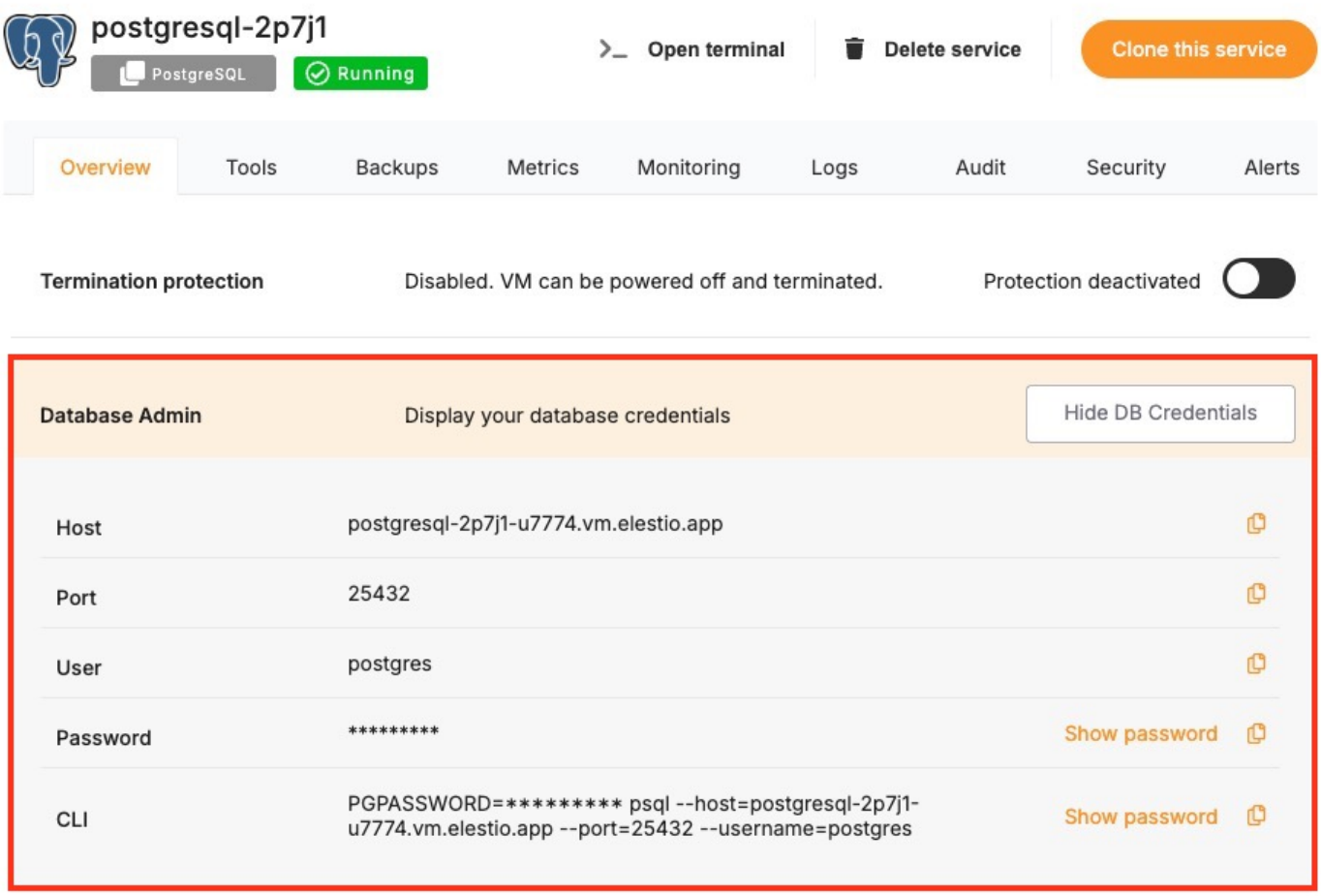
To connect to a PostgreSQL database, you only need one environment variable — the **connection URI**. This URI contains all the necessary information like username, password, host, port, and database name.

Variable	Description	Purpose
POSTGRESQL_URI	Full PostgreSQL connection string (from the Elestio service overview page)	Provides all necessary credentials and endpoint details in a single URI format.

The URI will look like this:

```
postgresql://<USER>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>
```

You can find the details needed in the URI from the **Elestio service overview** details. Copy and replace the variables carefully in the URI example provided above.



Prerequisites

Install PHP

Check if PHP is installed by running:

```
php -v
```

If not installed, download and install it from https://www.php.net/downloads.php.

Code

Once all prerequisites are set up, create a new file named `pg.php` and add the following code and replace the `POSTGRESQL_URI` with actual link or in environment setup as you wish:

```
<?php

$db_url = "POSTGRESQL_URI";//Replace with actual URI
$db_parts = parse_url($db_url);
```

```
$dsn = "pgsql:host={$db_parts['host']};port={$db_parts['port']};dbname=Elestio";//Replace with your DB name
$pdo = new PDO($dsn, $db_parts['user'], $db_parts['pass']);

$version = $pdo->query("SELECT VERSION()")->fetchColumn();
echo $version;
```

To execute the script, open the terminal or command prompt and navigate to the directory where `pg.php`. Once in the correct directory, run the script with the command

```
php pg.php
```

If the connection is successful, the terminal will display output similar to:

```
PostgreSQL 16.8 (Debian 16.8-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
```

Connecting with Go

This guide explains how to establish a connection between a **Go (Golang)** application and a **PostgreSQL** database using the `github.com/lib/pq` driver. It walks through the necessary setup, configuration, and execution of a simple SQL query.

Variables

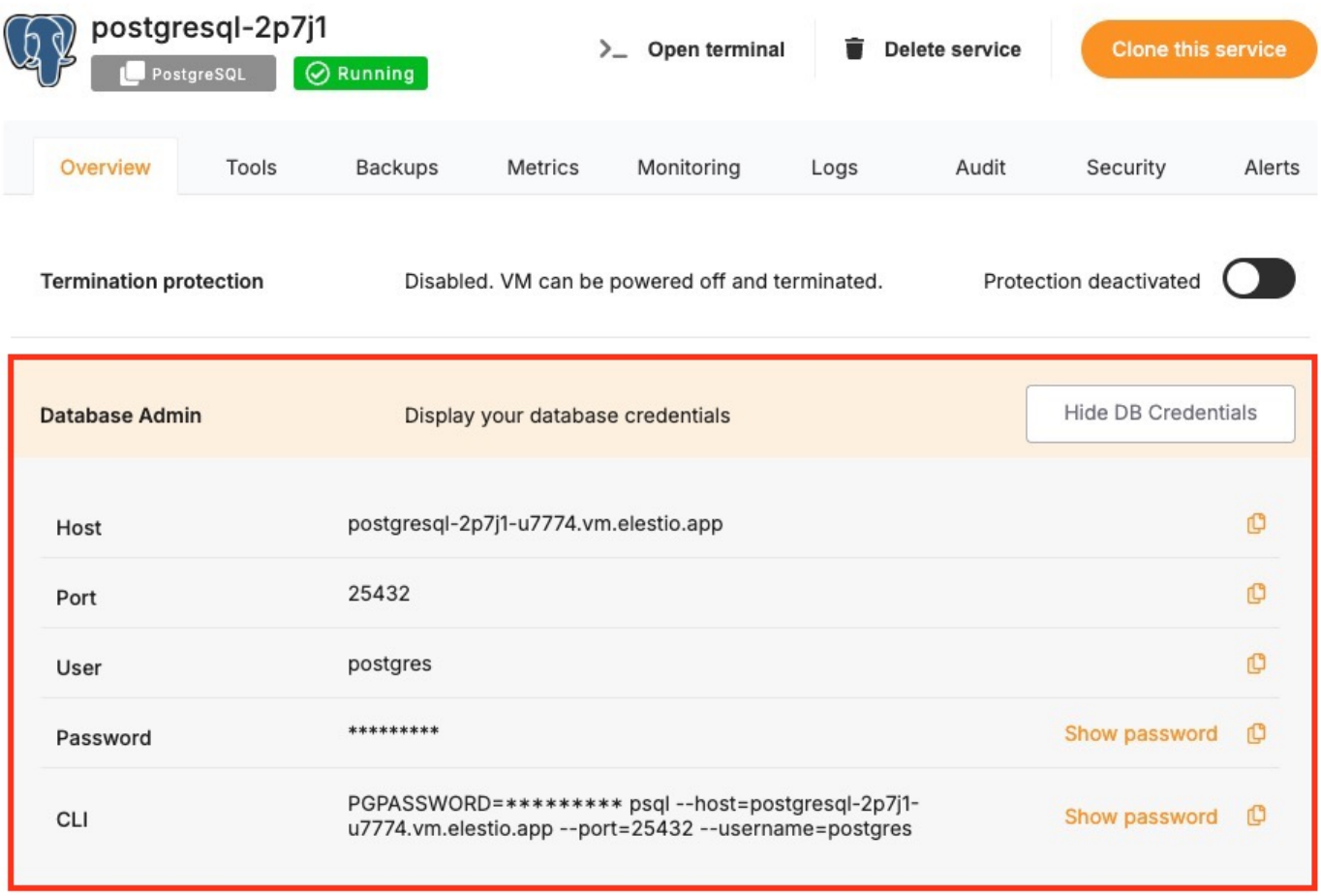
To connect to a PostgreSQL database, you only need one environment variable — the **connection URI**. This URI contains all the necessary information like username, password, host, port, and database name.

Variable	Description	Purpose
POSTGRESQL_URI	Full PostgreSQL connection string (from the Elestio service overview page)	Provides all necessary credentials and endpoint details in a single URI format.

The URI will look like this:

```
postgresql://<USER>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>
```

You can find the details needed in the URI from the **Elestio service overview** details. Copy and replace the variables carefully in the URI example provided above.



Prerequisites

Install Go

Check if Go is installed by running:

```
go version
```

If not installed, download and install it from https://go.dev/dl/.

Install `pq` Package

Install the `pq` driver using:

```
go get github.com/lib/pq
```

Code

Once all prerequisites are set up, create a new file named `main.go` and add the following code, and replace the `POSTGRESQL_URI` with actual link or in environment setup as you wish:

```
package main

import (
	"database/sql"
	"fmt"
	"log"
	"net/url"

	_ "github.com/lib/pq"
)

func getDBConnection(connectionString string) (*sql.DB, error) {
	parsedURL, err := url.Parse(connectionString)
	if err != nil {
		return nil, fmt.Errorf("Failed to parse connection string: %v", err)
	}

	db, err := sql.Open("postgres", parsedURL.String())
	if err != nil {
		return nil, fmt.Errorf("Failed to open database connection: %v", err)
	}

	return db, nil
}

func main() {
	connectionString := "POSTGRESQL_URI"

	db, err := getDBConnection(connectionString)
	if err != nil {
		log.Fatal(err)
	}
	defer db.Close()

	query := "SELECT current_database(), current_user, version()"
	rows, err := db.Query(query)
	if err != nil {
		log.Fatal("Failed to execute query:", err)
	}
	defer rows.Close()
```

```
	for rows.Next() {
		var dbName, user, version string
		if err := rows.Scan(&dbName, &user, &version); err != nil {
			log.Fatal("Failed to scan row:", err)
		}
		fmt.Printf("Database: %s\nUser: %s\nVersion: %s\n", dbName, user, version)
	}
}
```

To execute the script, open the terminal or command prompt and navigate to the directory where `main.go`. Once in the correct directory, run the script with the command

```
go run main.go
```

If the connection is successful, the terminal will display output similar to:

```
Database: Elestio
User: postgres
Version: PostgreSQL 16.8 (Debian 16.8-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
```

Connecting with Java

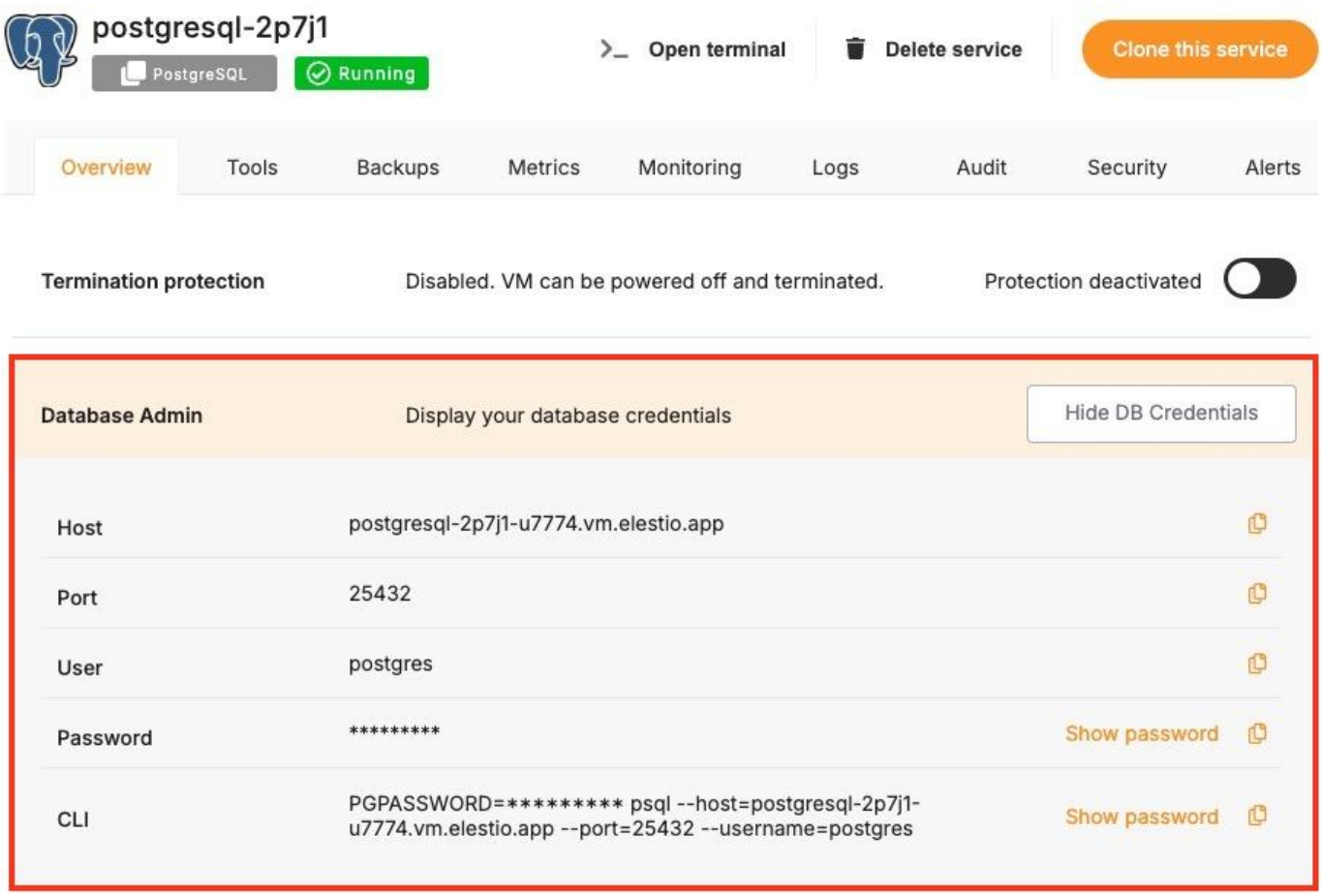
This guide explains how to establish a connection between a **Java** application and a **PostgreSQL** database using the **JDBC driver**. It walks through the necessary setup, configuration, and execution of a simple SQL query.

Variables

Certain parameters must be provided to establish a successful connection to a PostgreSQL database. Below is a breakdown of each required variable, its purpose, and where to find it. Here's what each variable represents:

Variable	Description	Purpose
`USER`	PostgreSQL username, from the Elestio service overview page	Identifies the database user who has permission to access the PostgreSQL database.
`PASSWORD`	PostgreSQL password, from the Elestio service overview page	The authentication key required for the specified `USER` to access the database
`HOST`	Hostname for PostgreSQL connection, from the Elestio service overview page	The address of the server hosting the PostgreSQL database.
`PORT`	Port for PostgreSQL connection, from the Elestio service overview page	The network port is used to connect to PostgreSQL. The default port is `5432`.
`DATABASE`	Database Name for PostgreSQL connection, from the Elestio service overview page	The name of the database being accessed. A PostgreSQL instance can contain multiple databases.

These values can usually be found in the Elestio service overview details, as shown in the image below. Make sure to take a copy of these details and add them to the code moving ahead.



Prerequisites

Install Java & JDBC driver

Check if Java is installed by running:

```
java -version
```

If not installed, install it first and then download and install **JDBC** driver from https://jdbc.postgresql.org/download/ or if you have Maven installed, run the following command with updated version of the driver:

```
mvn org.apache.maven.plugins:maven-dependency-plugin:2.8:get -
Dartifact=org.postgresql:postgresql:42.7.5:jar -Ddest=postgresql-42.7.5.jar
```

Code

Once all prerequisites are set up, create a new file named `Pg.java` and add the following code:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;

public class Pg {
    private static class ConnectionConfig {
        private final String host;
        private final String port;
        private final String database;
        private final String username;
        private final String password;

        public ConnectionConfig(String host, String port, String database, String username,
String password) {
            this.host = host;
            this.port = port;
            this.database = database;
            this.username = username;
            this.password = password;
        }

        public String getConnectionUrl() {
            return String.format("jdbc:postgresql://%s:%s/%s?sslmode=require", host, port,
database);
        }

        public boolean isValid() {
            return host != null && !host.isEmpty() &&
                   port != null && !port.isEmpty() &&
                   database != null && !database.isEmpty();
        }
    }

    private static Map<String, String> parseArguments(String[] args) {
        Map<String, String> config = new HashMap<>();
        for (int i = 0; i < args.length - 1; i++) {
```

```
            String key = args[i].toLowerCase();
            String value = args[++i];
            config.put(key, value);
        }
        return config;
    }

    private static ConnectionConfig createConfig(Map<String, String> args) {
        return new ConnectionConfig(
            args.get("-host"),
            args.get("-port"),
            args.get("-database"),
            args.get("-username"),
            args.get("-password")
        );
    }

    private static void validateConnection(Connection connection) throws SQLException {
        try (Statement stmt = connection.createStatement();
             ResultSet rs = stmt.executeQuery("SELECT version()")) {
            if (rs.next()) {
                System.out.println("Database Version: " + rs.getString("version"));
            }
        }
    }

    public static void main(String[] args) {
        try {
            // Load PostgreSQL driver
            Class.forName("org.postgresql.Driver");

            // Parse and validate configuration
            Map<String, String> parsedArgs = parseArguments(args);
            ConnectionConfig config = createConfig(parsedArgs);

            if (!config.isValid()) {
                System.err.println("Error: Missing required connection parameters (host, port,
database)");
                return;
            }
```

```
            // Establish connection and validate
            try (Connection conn = DriverManager.getConnection(
                    config.getConnectionUrl(),
                    config.username,
                    config.password)) {

                System.out.println("Successfully connected to the database!");
                validateConnection(conn);
            }

        } catch (ClassNotFoundException e) {
            System.err.println("Error: PostgreSQL JDBC Driver not found");
            e.printStackTrace();
        } catch (SQLException e) {
            System.err.println("Database connection error:");
            e.printStackTrace();
        }
    }
}
```

To execute the script, open the terminal or command prompt and navigate to the directory where `Pg.java`. Once in the correct directory, run the script with the command (Update the variables with actual values acquired from previous steps.)

```
javac Pg.java && java -cp postgresql-42.7.5.jar:. Pg -host HOST -port PORT -database DATABASE
-username avnadmin -password PASSWORD
```

If the connection is successful, the terminal will display output similar to:

```
Successfully connected to the database!
Database Version: PostgreSQL 16.8 (Debian 16.8-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
```

Connecting with psql

This guide explains how to connect to a **PostgreSQL** database using the `psql` command-line tool. It walks through the necessary setup, connection process, and execution of a simple SQL query.

Variables

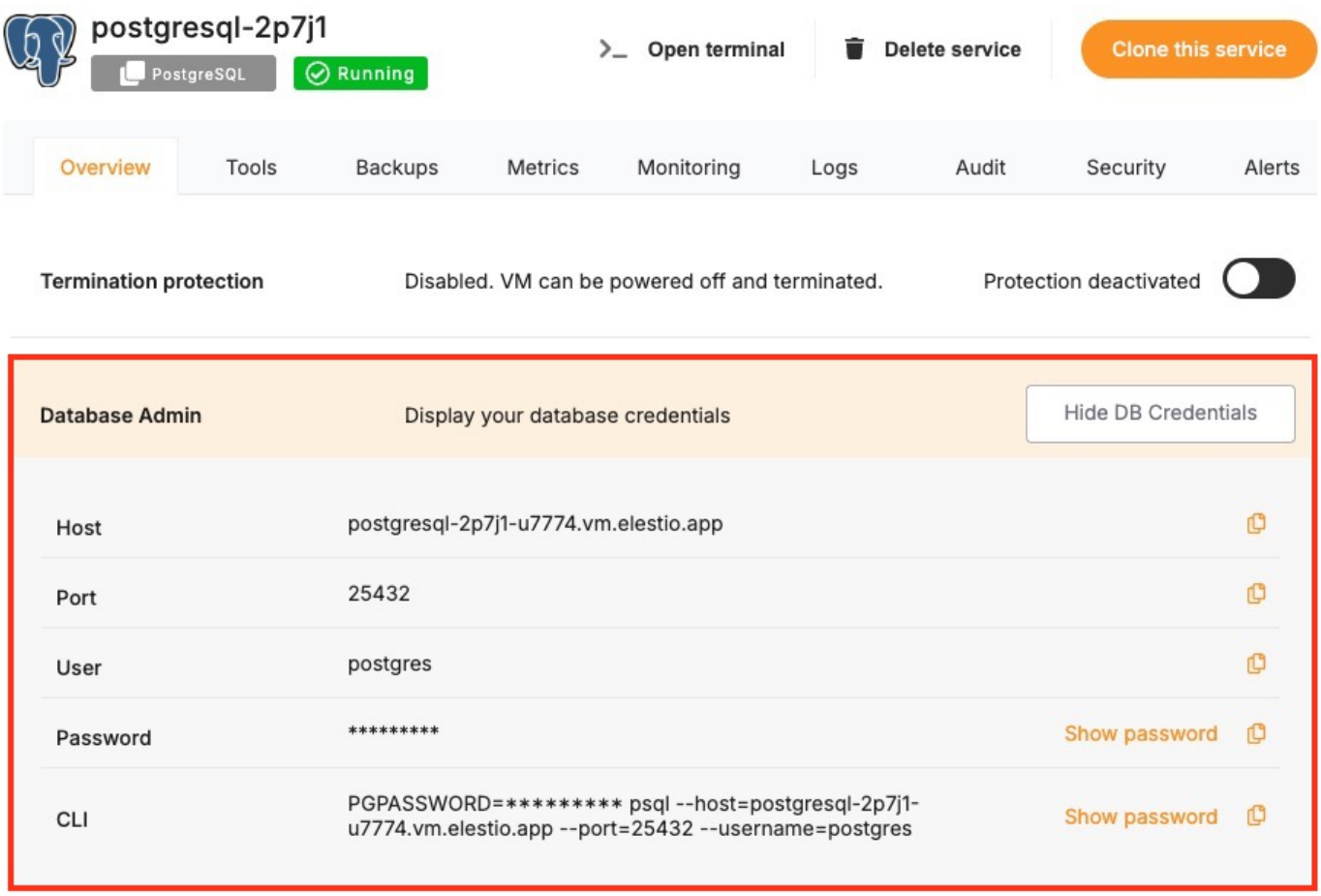
To connect to a PostgreSQL database, you only need one environment variable — the **connection URI**. This URI contains all the necessary information like username, password, host, port, and database name.

Variable	Description	Purpose
POSTGRESQL_URI	Full PostgreSQL connection string (from the Elestio service overview page)	Provides all necessary credentials and endpoint details in a single URI format.

The URI will look like this:

```
postgresql://<USER>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>
```

You can find the details needed in the URI from the **Elestio service overview** details. Copy and replace the variables carefully in the URI example provided above.



Prerequisites

While following this tutorial, you will need to have `psql` already installed; if not head over to https://www.postgresql.org/download/ and download it first.

Connecting to PostgreSQL

Open your terminal and run the following command to connect to your PostgreSQL database using the full connection URI:

```
psql POSTGRESQL_URI
```

If the connection is successful, you'll see output similar to this. Here it will show you the database you tried to connect to, which in this case is Elestio:

```
psql (17.4, server 16.8 (Debian 16.8-1.pgdg120+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off, ALPN: none)
Type "help" for help.
```

```
Elestio=#
```

To ensure you're connected correctly, run this command inside the `psql` prompt:

```
SELECT version();
```

You should receive output like the following:

```
version
------------------------------------------------------------------------------------------------
-----------------------
 PostgreSQL 16.8 (Debian 16.8-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian
12.2.0-14) 12.2.0, 64-bit
(1 row)
```

Connecting with pgAdmin

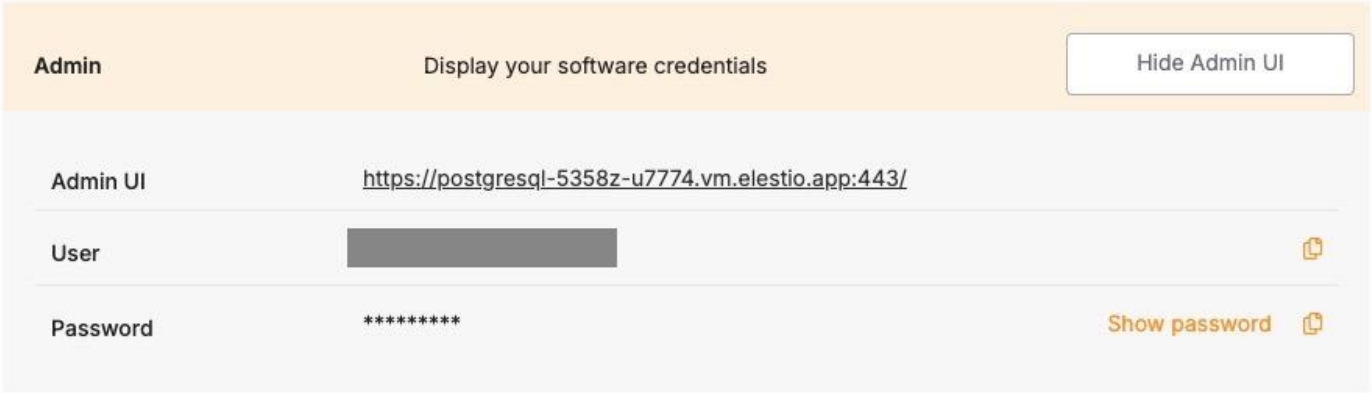
pgAdmin is a widely used graphical interface for PostgreSQL that allows you to manage, connect to, and run queries on your databases with ease.

Variables

To connect using `pgAdmin`, you'll need the following connection parameters. When you deploy a PostgreSQL service on Elestio, you also get a pgAdmin dashboard configured for you to use with these variables. These details are available in the **Elestio service overview page**:

Variable	Description	Purpose
USER	pgAdmin username	Identifies the pgAdmin user with access permission.
PASSWORD	pgAdmin password	Authentication key for the `USER`.

You can find these values in your Elestio project dashboard under **Admin** section.



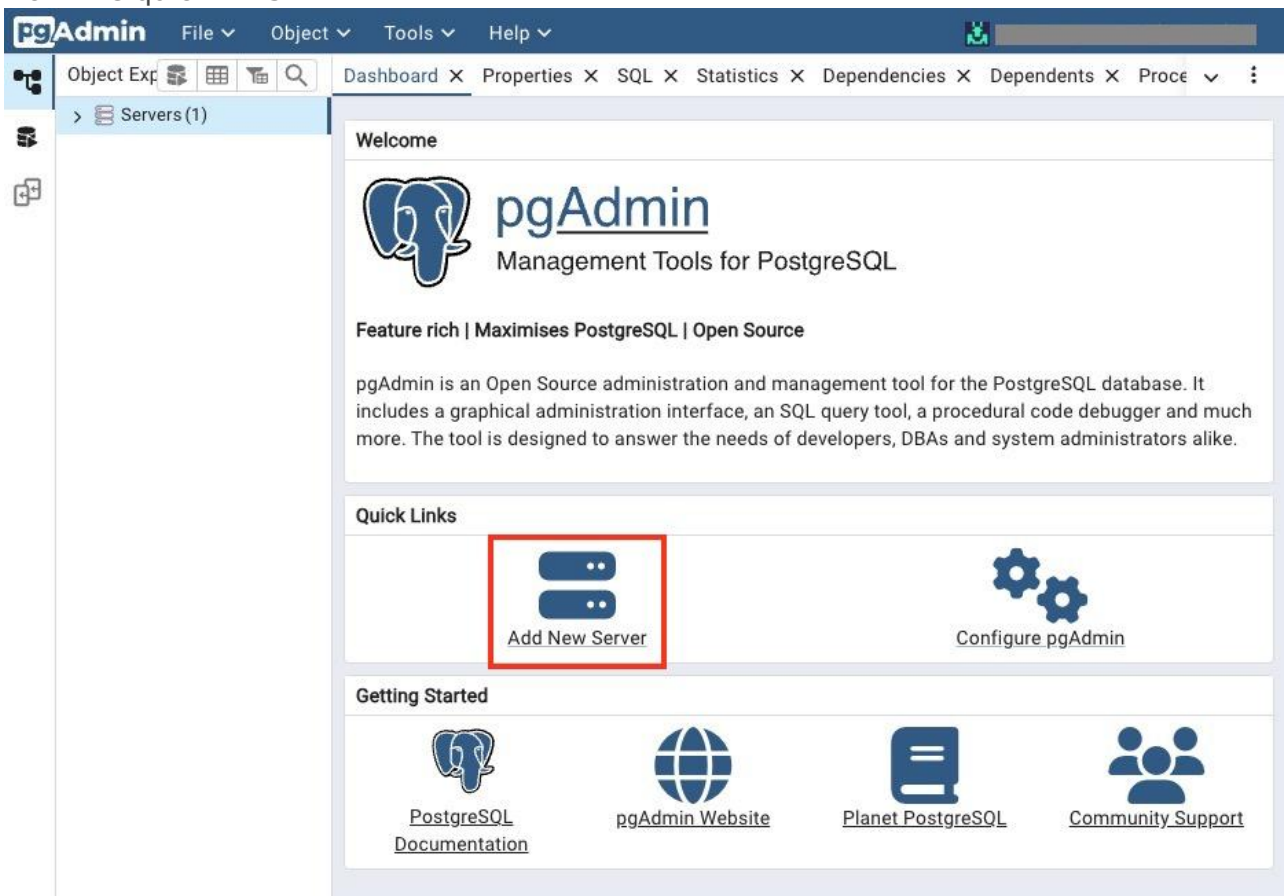
Prerequisites

Make sure the **PostgreSQL** service is correctly deployed on Elestio and you are able to access the Admin section like the one in the image above.

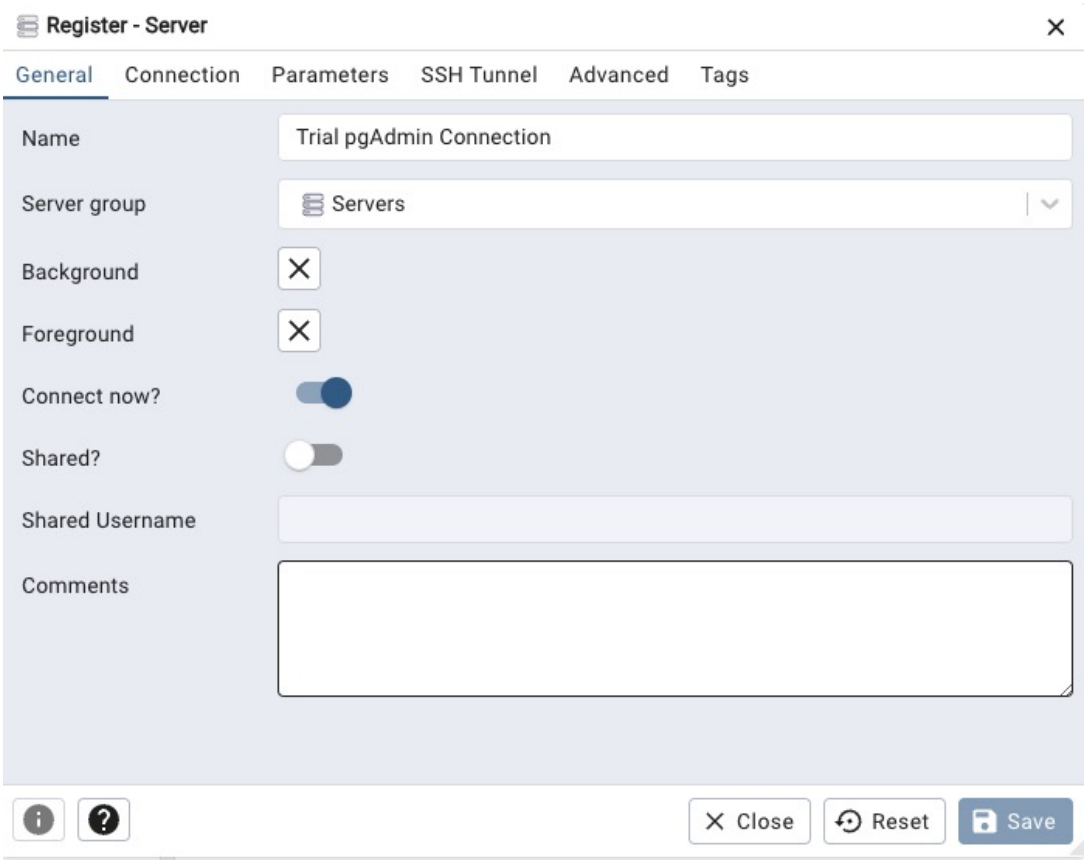
Setting Up the Connection

1. Launch **pgAdmin** from the Admin UI URL and log in with the credentials acquired in the steps before.

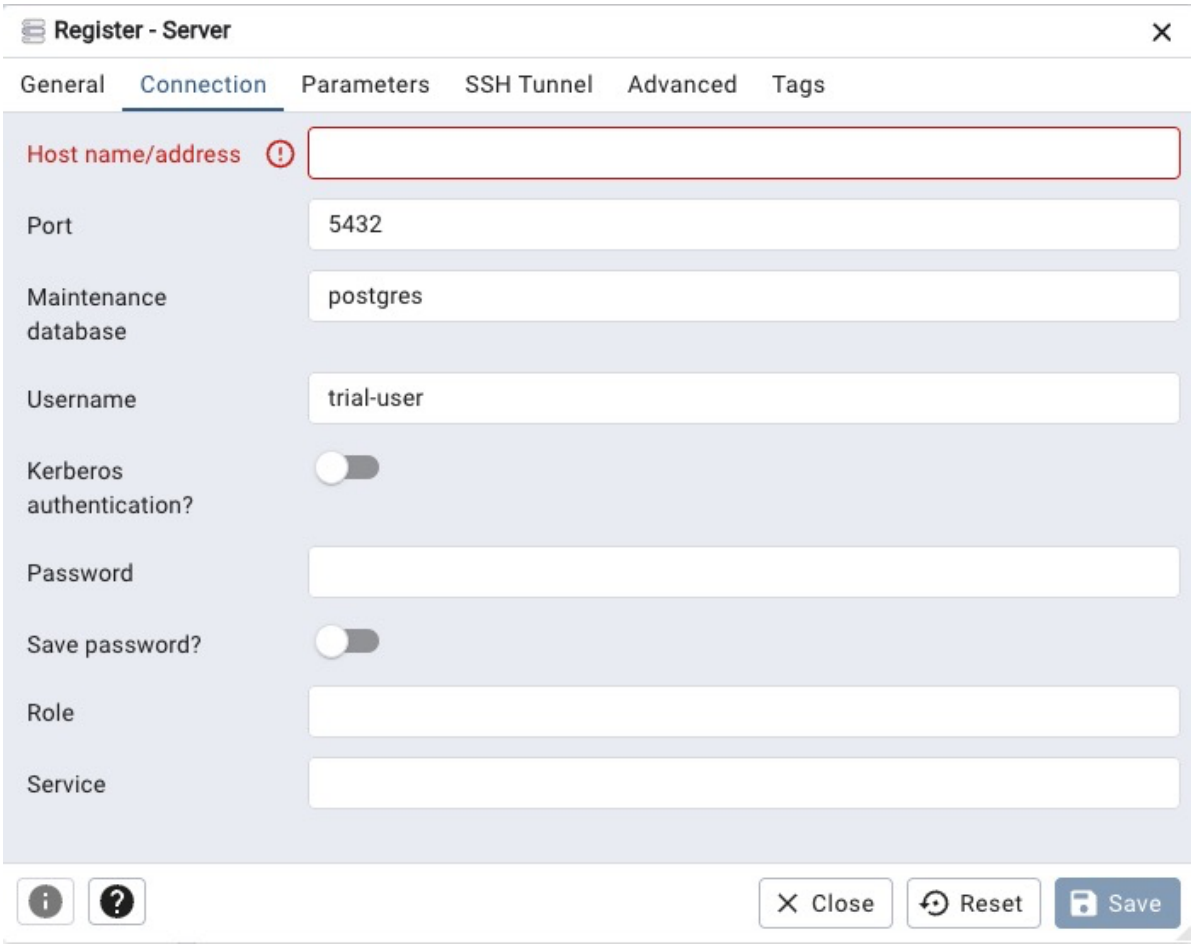
2. Click on **"Create"** and select **"Server..."** from the dropdown, or find **Add New Server** from the quick links



3. In the **General** tab:
   - Enter a name for your connection (e.g., `Trial pgAdmin Connection`).



4. Go to the **Connection** tab and enter the following details:
   - **Host name/address**: `HOSTNAME`
   - **Port**: `PORT`
   - **Maintenance database**: `DATABASE`
   - **Username**: `USERNAME`
   - **Password**: `PASSWORD`


Register - Server
General
Connection
Parameters
SSH Tunnel
Advanced
Tags
Host name/address
Port
5432
Maintenance database
postgres
Username
trial-user
Kerberos authentication?
Password
Save password?
Role
Service
Close
Reset
Save

How-To Guides

Creating a Database

PostgreSQL allows you to create databases using different methods, including the PostgreSQL interactive shell (`psql`), Docker (assuming PostgreSQL is running inside a container), and the command-line interface (`createdb`). This guide explains each method step-by-step, covering required permissions, best practices, and troubleshooting common issues.

Creating Using psql CLI

PostgreSQL is a database system that stores and manages structured data efficiently. The `psql` tool is an interactive command-line interface (CLI) that allows users to execute SQL commands directly on a PostgreSQL database. Follow these steps to create a database:

Connect to PostgreSQL

Open terminal on your local system, and if PostgreSQL is installed locally, connect using the following command. If not installed, install from official website:

```
psql -U postgres
```

For a remote database, use:

```
psql -h HOST -U USER -d DATABASE
```

Replace `HOST` with the database server address, `USER` with the PostgreSQL username, and `DATABASE` with an existing database name.

Create a New Database

Inside the `psql` shell, run:

```
CREATE DATABASE mydatabase;
```

The default settings will apply unless specified otherwise. To customize the encoding and collation, use:

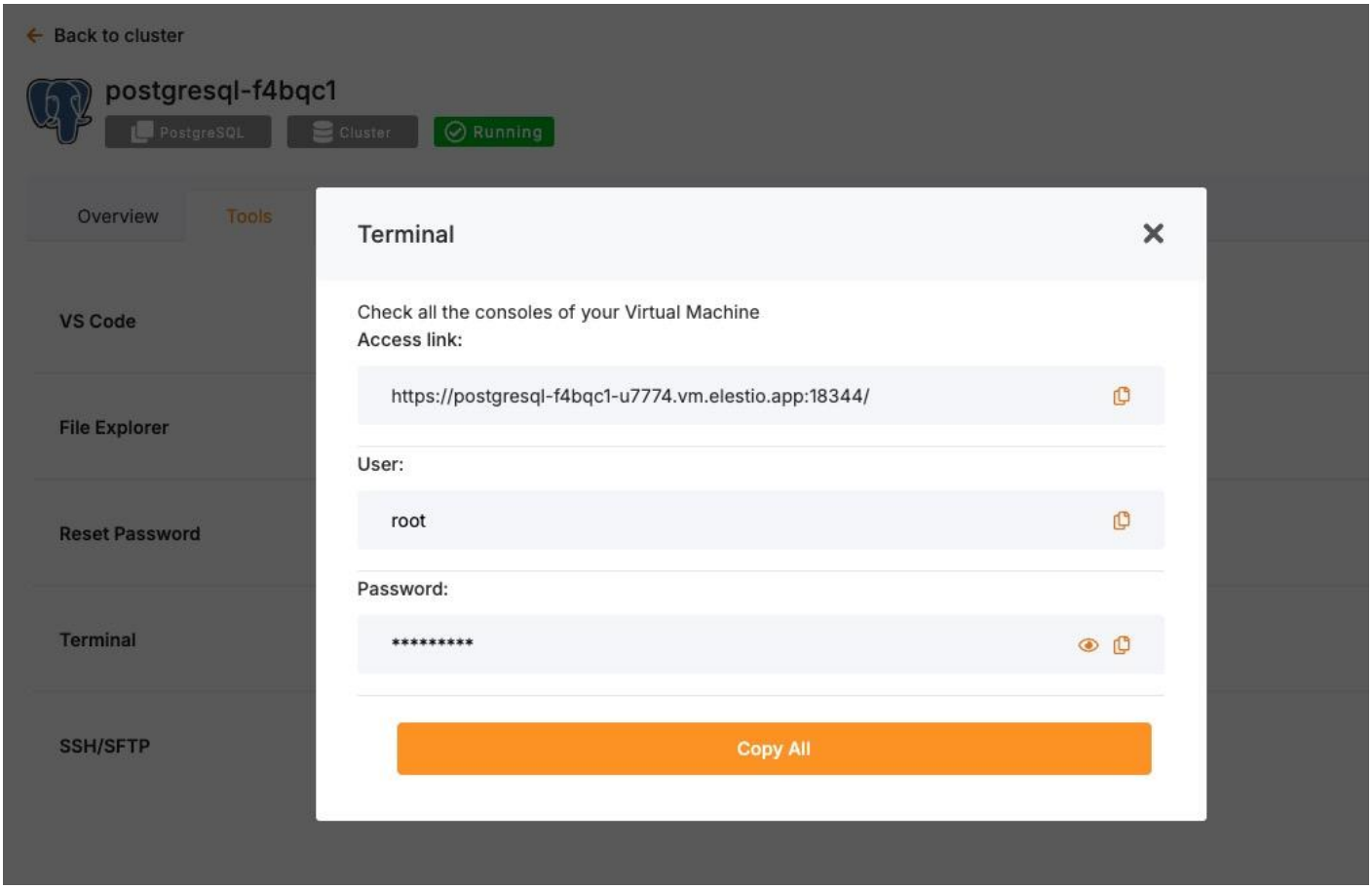
```
CREATE DATABASE mydatabase ENCODING 'UTF8' LC_COLLATE 'en_US.UTF-8' LC_CTYPE 'en_US.UTF-8'
TEMPLATE template0;
```

Creating Database in Docker

Docker is a tool that helps run applications in isolated environments called containers. A PostgreSQL container provides a self-contained database instance that can be quickly deployed and managed. If you are running PostgreSQL inside a Docker container, follow these steps:

Access Elestio Terminal

Head over to your deployed PostgreSQL service dashboard and head over to **Tools > Terminal**. Use the credentials provided there to log in to your terminal.



Once you are in your terminal, run the following command to head over to the correct directory to perform the next steps

```
cd /opt/app/
```

Access the PostgreSQL Container Shell

Instead of pulling an image or running the container manually, use Docker Compose to interact with your running container. As you are using Elestio, it will already be a Docker compose:

```
docker-compose exec postgres bash
```

This opens a shell session inside the running PostgreSQL container.

Use Environment Variables to Connect via psql

Once inside the container shell, if environment variables like `POSTGRES_USER` and `POSTGRES_DB` are already set in the stack, you can use them directly:

```
psql -U "$POSTGRES_USER" -d "$POSTGRES_DB"
```

Or use the default one:

```
psql -U postgres
```

Create Database

Now, to create a database, use the following command. This command tells PostgreSQL to create a new logical database called `mydatabase`. By default, it inherits settings like encoding and collation from the template database (template1), unless specified otherwise.

```
CREATE DATABASE mydatabase;
```

You can quickly list the database you just created using the following command

```
/l
```

Creating Using createdb CLI

The `createdb` command simplifies database creation from the terminal without using `psql`.

Ensure PostgreSQL is Running

Check the PostgreSQL service status, this ensures that the PostgreSQL instance is running on your local instance:

```
sudo systemctl status postgresql
```

If not running, start it:

```
sudo systemctl start postgresql
```

Create a Database

Now, you can create a simple database using the following command:

```
createdb -U postgres mydatabase
```

To specify encoding and collation:

```
createdb -U postgres --encoding=UTF8 --lc-collate=en_US.UTF-8 --lc-ctype=en_US.UTF-8
mydatabase
```

Verify Database Creation

List all databases using the following commands, as it will list all the databases available under your PostgreSQL:

```
psql -U postgres -l
```

Connect to the New Database

Next, you can easily connect with the database using the psql command and start working on it.

```
psql -U postgres -d mydatabase
```

Required Permissions for Database Creation

Creating a database requires the `CREATEDB` privilege. By default, the `postgres` user has this privilege. To grant it to another user:

```
ALTER USER username CREATEDB;
```

For restricted access, assign specific permissions:

```
CREATE ROLE newuser WITH LOGIN PASSWORD 'securepassword';
GRANT CONNECT ON DATABASE mydatabase TO newuser;
GRANT USAGE ON SCHEMA public TO newuser;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO newuser;
```

Best Practices for Creating Databases

- **Use Meaningful Names:** Choosing clear and descriptive names for databases helps in organization and maintenance. Avoid generic names like `testdb` or `database1`, as they do not indicate the database's purpose. Instead, use names that reflect the type of data stored, such as `customer_data` or `sales_records`. Meaningful names make it easier for developers and administrators to understand the database's function without extra documentation.
- **Follow Naming Conventions:** A standardized naming convention ensures consistency across projects and simplifies database management. PostgreSQL is case-sensitive, so using lowercase letters and underscores (e.g., `order_details`) is recommended to avoid unnecessary complexities. Avoid spaces and special characters in names, as they require additional quoting in SQL queries.
- **Restrict User Permissions:** Granting only the necessary permissions improves database security and reduces risks. By default, users should have the least privilege required for their tasks, such as read-only access for reporting tools. Superuser or administrative privileges should be limited to trusted users to prevent accidental or malicious changes. Using roles and groups simplifies permission management and ensures consistent access control.
- **Enable Backups:** Regular backups ensure data recovery in case of accidental deletions, hardware failures, or security breaches. PostgreSQL provides built-in tools like `pg_dump` for single-database backups and `pg_basebackup` for full-instance backups. Automating backups using cron jobs or scheduling them through a database management tool reduces the risk of data loss.
- **Monitor Performance:** Monitoring database performance helps identify bottlenecks, optimize queries, and ensure efficient resource utilization. PostgreSQL provides system views like `pg_stat_activity` and `pg_stat_database` to track query execution and database usage. Analyzing slow queries using `EXPLAIN ANALYZE` helps in indexing and optimization.

```
SELECT datname, numbackends, xact_commit, blks_read FROM pg_stat_database;
```

Common Issues and Troubleshooting

Issue	Possible Cause	Solution
`ERROR: permission denied to create database`	User lacks `CREATEDB` privileges	Grant permission using `ALTER USER username CREATEDB;`
`ERROR: database "mydatabase" already exists`	Database name already taken	Use a different name or drop the existing one with `DROP DATABASE mydatabase;`
`FATAL: database "mydatabase" does not exist`	Attempting to connect to a non-existent database	Verify creation using `\l`

Issue	Possible Cause	Solution
`psql: could not connect to server`	PostgreSQL is not running	Start PostgreSQL with `sudo systemctl start postgresql`
`ERROR: role "username" does not exist`	The specified user does not exist	Create the user with `CREATE ROLE username WITH LOGIN PASSWORD 'password';`

Upgrading to a Major Version

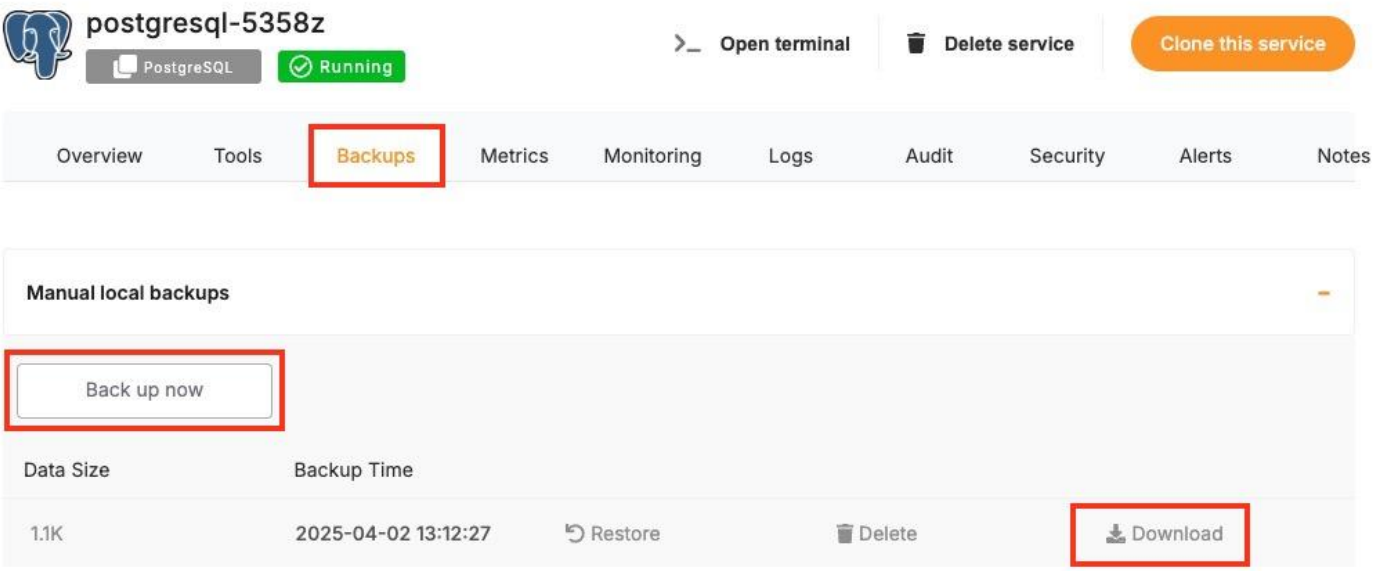
Upgrading a database service on Elestio can be done without creating a new instance or performing a full manual migration. Elestio provides a built-in option to change the database version directly from the dashboard. This is useful for cases where the upgrade does not involve breaking changes or when minimal manual involvement is preferred. The version upgrade process is handled by Elestio internally, including restarting the database service if required. This method reduces the number of steps involved and provides a way to keep services up to date with minimal configuration changes.

Log In and Locate Your Service

To begin the upgrade process, log in to your Elestio dashboard and navigate to the specific database service you want to upgrade. It is important to verify that the correct instance is selected, especially in environments where multiple databases are used for different purposes such as staging, testing, or production. The dashboard interface provides detailed information for each service, including version details, usage metrics, and current configuration. Ensure that you have access rights to perform upgrades on the selected service. Identifying the right instance helps avoid accidental changes to unrelated environments.

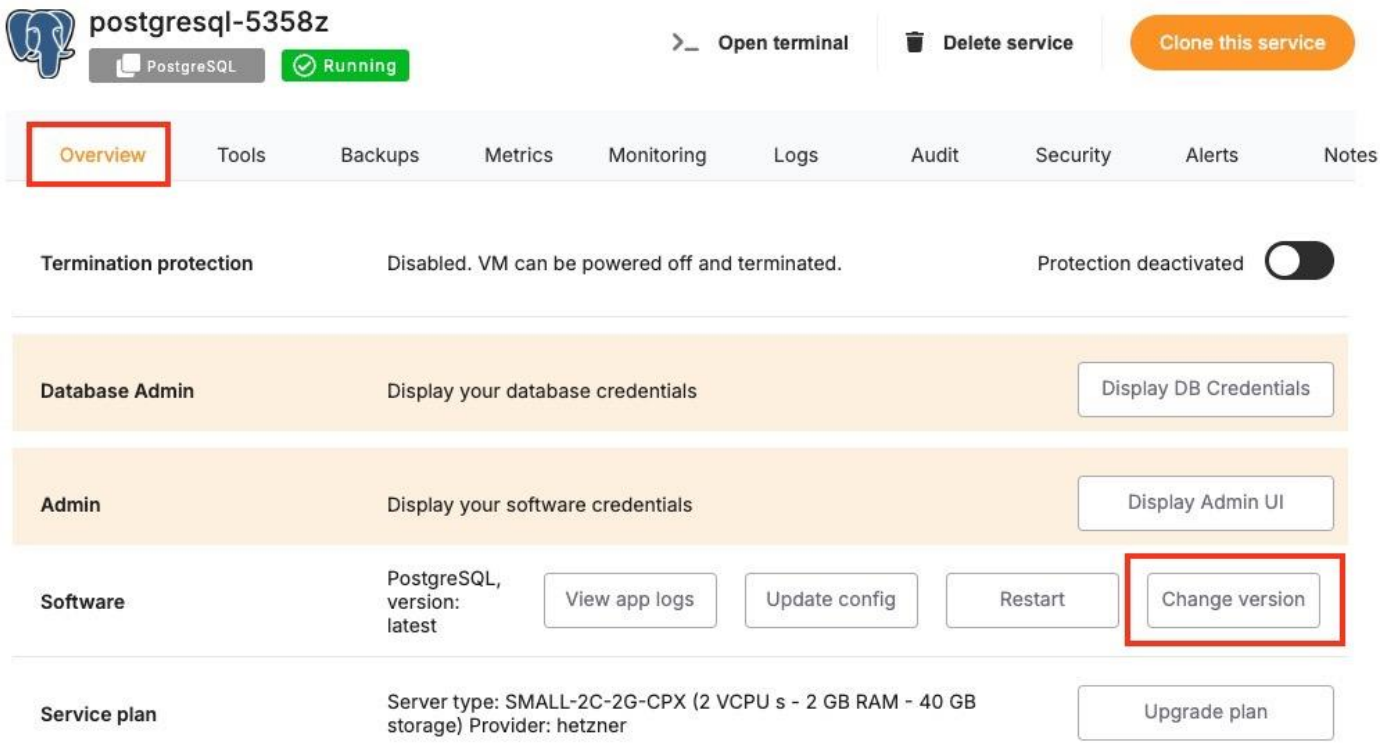
Back Up Your Data

Before starting the upgrade, create a backup of your database. A backup stores the current state of your data, schema, indexes, and configuration, which can be restored if something goes wrong during the upgrade. In Elestio, this can be done through the **Backups** tab by selecting **Back up now** under Manual local backups and **Download** the backup file. Scheduled backups may also be used, but it is recommended to create a manual one just before the upgrade. Keeping a recent backup allows quick recovery in case of errors or rollback needs. This is especially important in production environments where data consistency is critical.

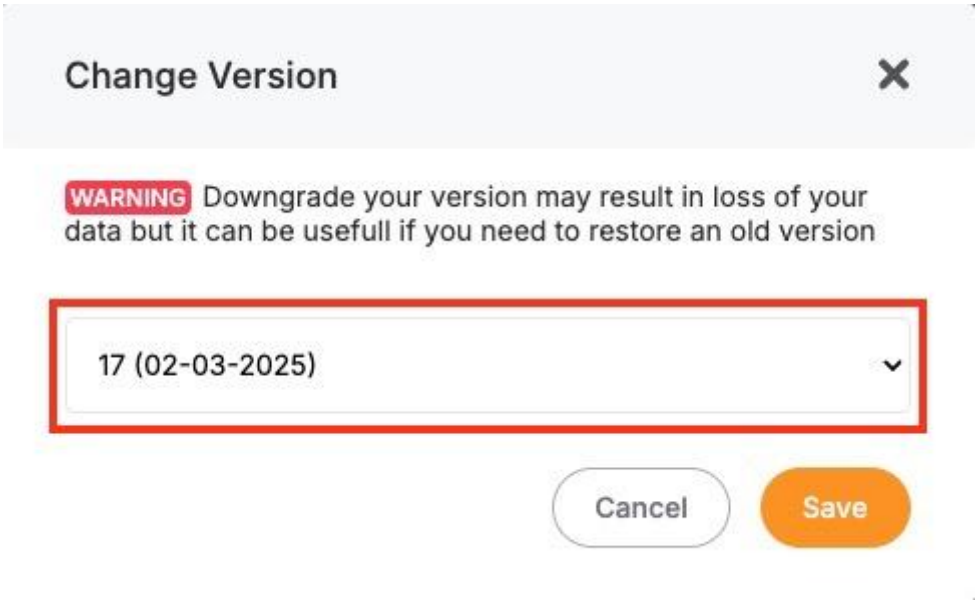


Select the New Version

Once your backup is secure, proceed to the **Overview** and then **Software > Change version** tab within your database service page.



Here, you'll find an option labeled **Change Version**. In the **Change Version** menu, select the desired database version from the available list. After confirming the version, Elestio will begin the upgrade process automatically. During this time, the platform takes care of the version change and restarts the database if needed. No manual commands are required, and the system handles most of the operational aspects in the background.



Monitor the Upgrade Process

The upgrade process may include a short downtime while the database restarts. Once it is completed, it is important to verify that the upgrade was successful and the service is operating as expected. Start by checking the logs available in the Elestio dashboard for any warnings or errors during the process. Then, review performance metrics to ensure the database is running normally and responding to queries. Finally, test the connection from your client applications to confirm that they can interact with the upgraded database without issues.

Installing or Updating an Extension

PostgreSQL supports a wide range of extensions that add extra functionality to the core database system. Extensions like uuid-ossp, pg_trgm, and postgis are often used to provide features for text search, spatial data, UUID generation, and more. If you are running PostgreSQL on Elestio, you can enable many of these extensions directly within your database. This document explains how to enable, manage, and troubleshoot PostgreSQL extensions in an Elestio-hosted environment. It also includes guidance on checking extension compatibility with different PostgreSQL versions.

Installing and Enabling Extensions

PostgreSQL extensions can be installed in each database individually. Most common extensions are included in the PostgreSQL installation on Elestio. To enable an extension, you need to connect to your database using a tool like psql.

Start by connecting to your PostgreSQL database. You can follow the detailed documentation as provided here.

Once connected, you can enable an extension using the CREATE EXTENSION command. For example, to enable the uuid-ossp extension:

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
```

To check which extensions are already installed in your current database, use the `\dx` command within psql. If you want to see all available extensions on the server, use:

```
SELECT * FROM pg_available_extensions ORDER BY name;
```

If the extension you need is not listed in the available extensions, it may not be installed on the server.

Checking Extension Compatibility

Each PostgreSQL extension is built for a specific PostgreSQL version. Not all extensions are compatible across major versions. Before upgrading PostgreSQL or deploying an extension, it is important to check whether the extension is compatible with the version you are using.

To check the installed version of an extension and the default version provided by the system, run:

```
SELECT name, default_version, installed_version
FROM pg_available_extensions
WHERE name = 'pg_trgm';
```

If you are planning to upgrade your PostgreSQL version, it is recommended to deploy a new instance with the target version and run the above query to see if the extension is available and compatible. Some extensions may require specific builds for each version of PostgreSQL. After upgrading your database, you may also need to update your extensions using:

```
ALTER EXTENSION <extension_name> UPDATE;
```

This ensures the extension objects in the database match the new database version.

Troubleshooting Common Extension Issues

There are some common issues users may encounter when working with extensions. These usually relate to missing files, permission problems, or version mismatches.

If you see an error like could not open extension control file, it means the extension is not installed on the server. This usually happens when the extension is not included in the PostgreSQL installation. If the error message says that the extension already exists, it means it has already been installed in the database. You can confirm this with the \dx command or the query:

```
SELECT * FROM pg_extension;
```

If you need to reinstall it, you can drop and recreate it. Be careful, as dropping an extension with CASCADE may remove objects that depend on it:

```
DROP EXTENSION IF EXISTS <extension_name> CASCADE;
CREATE EXTENSION <extension_name>;
```

Another common issue appears after upgrading PostgreSQL, where some functions related to the extension stop working. This is often due to the extension not being updated. Running the following command will usually fix this.

```
ALTER EXTENSION <name> UPDATE;
```

In some cases, you may get a permission denied error when trying to create an extension. This means your database role does not have the required privileges. You will need to connect using a superuser account like postgres, or request that Elestio enable the extension for you.

Creating Manual Backups

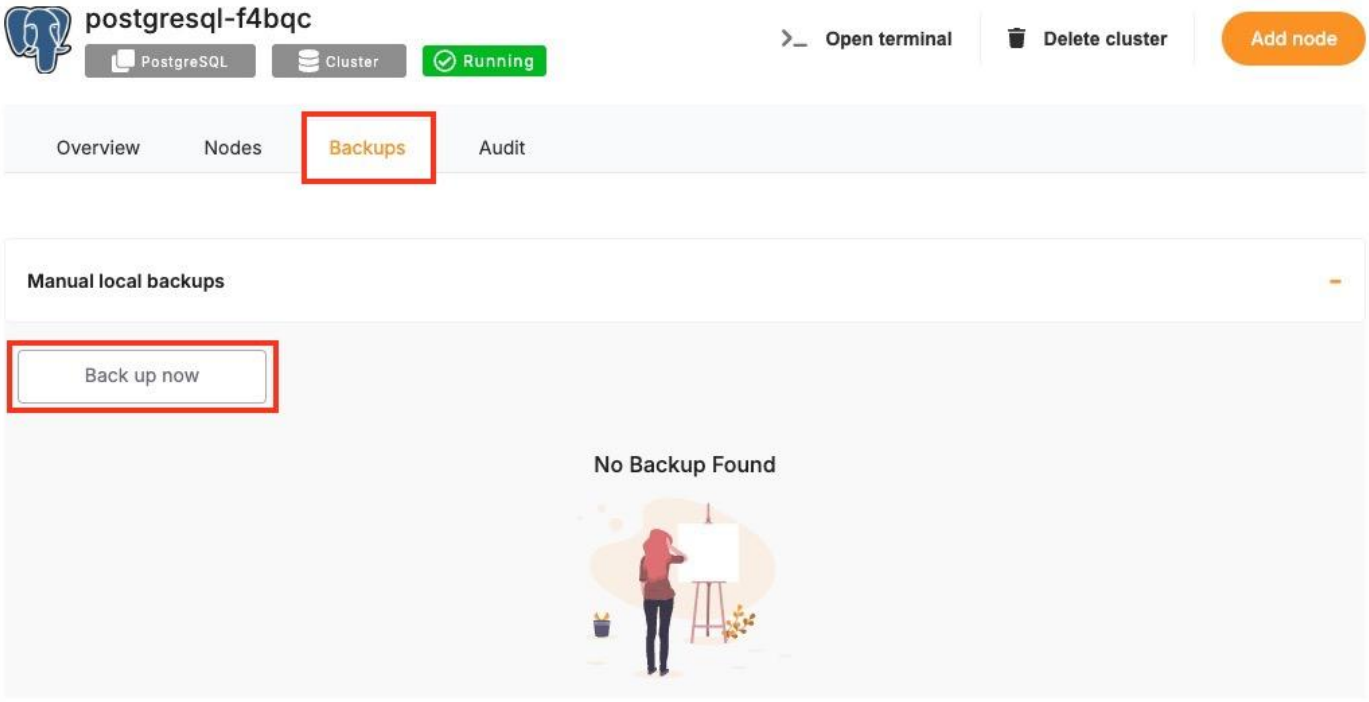
Regular backups are a key part of managing a PostgreSQL deployment. While Elestio provides automated backups by default, you may want to perform manual backups for specific reasons, such as preparing for a major change, keeping a local copy, or testing backup automation. This guide walks through how to create PostgreSQL backups on Elestio using multiple approaches. It covers manual backups through the Elestio dashboard, using PostgreSQL CLI tools, and Docker Compose-based setups. It also includes advice for backup storage, retention policies, and automation using scheduled jobs.

Manual Service Backups on Elestio

If you’re using Elestio’s managed PostgreSQL service, the easiest way to create a manual backup is through the dashboard. This built-in method creates a full snapshot of your current database state and stores it within Elestio’s infrastructure. These backups are tied to your service and can be restored through the same interface. This option is recommended when you need a quick, consistent backup without using any terminal commands.

To trigger a manual backup from the Elestio dashboard:

1. Log in to the Elestio dashboard and navigate to your PostgreSQL service/cluster.
2. Click the **Backups** tab in the service menu.
3. Select **Back up now** to generate a snapshot.

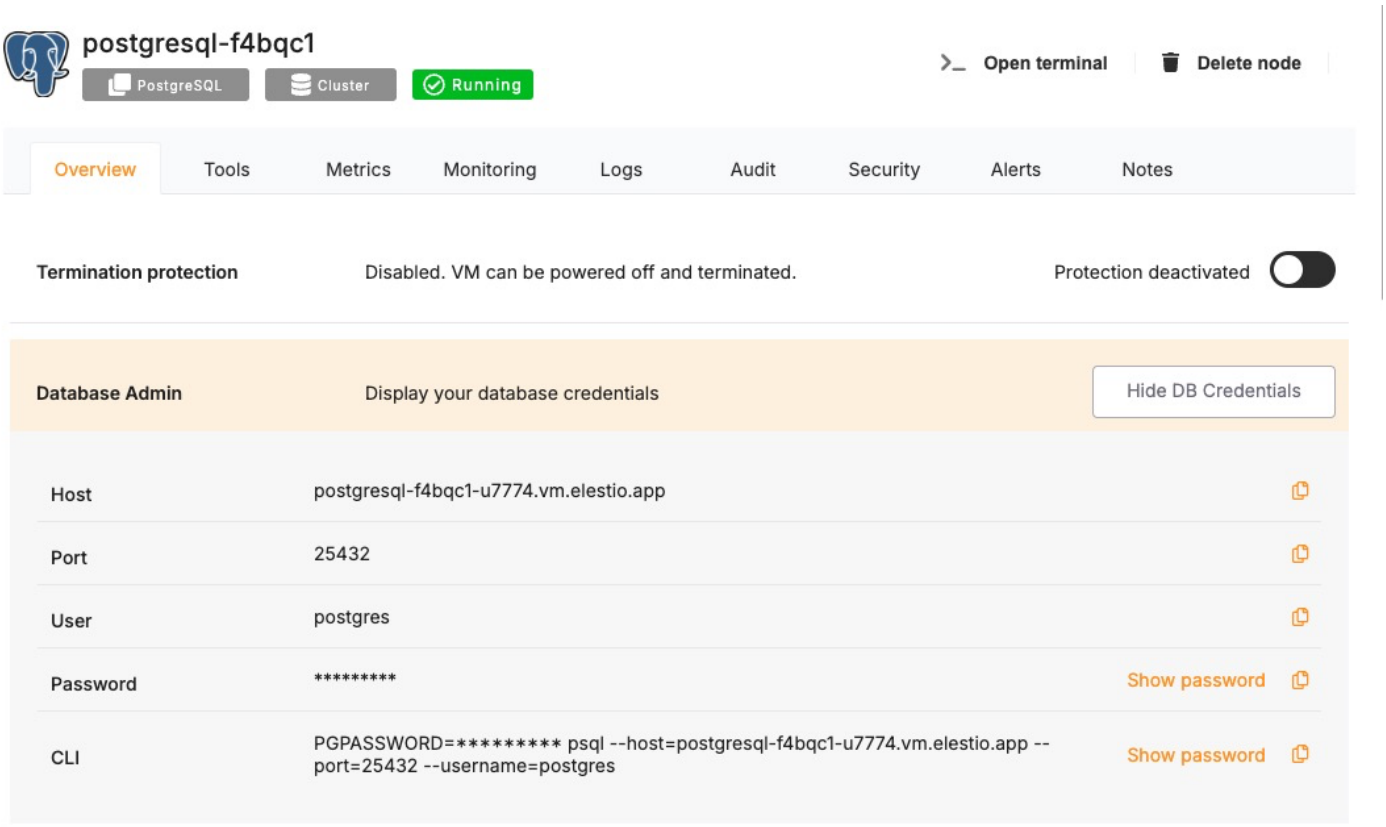


Manual Backups Using PostgreSQL CLI

PostgreSQL provides a set of command-line tools that are useful when you want to create backups from your terminal. These include `pg_dump` exporting the database, psql for basic connectivity and queries, and `pg_restore` restoring backups. This approach is useful when you need to store backups locally or use them with custom automation workflows. The CLI method gives you full control over the backup format and destination.

Collect Database Connection Info

To use the CLI tools, you'll need the database host, port, name, username, and password. These details can be found in the **Overview** section of your PostgreSQL service in the Elestio dashboard.



Back Up with pg_dump

Use `pg_dump` to export the database in a custom format. This format is flexible and preferred for restore operations using `pg_restore`. Replace the values with actual values that you copied from the Elestio overview page.

```
PGPASSWORD='<your-db-password>' pg_dump \
  -U <username> \
  -h <host> \
  -p <port> \
  -Fc -v \
  -f <output_file>.dump \
  <database_name>
```

This command connects to the Elestio database and creates a `.dump` file containing your data. You can use the `-v` flag for verbose output and confirm that the backup completed successfully.

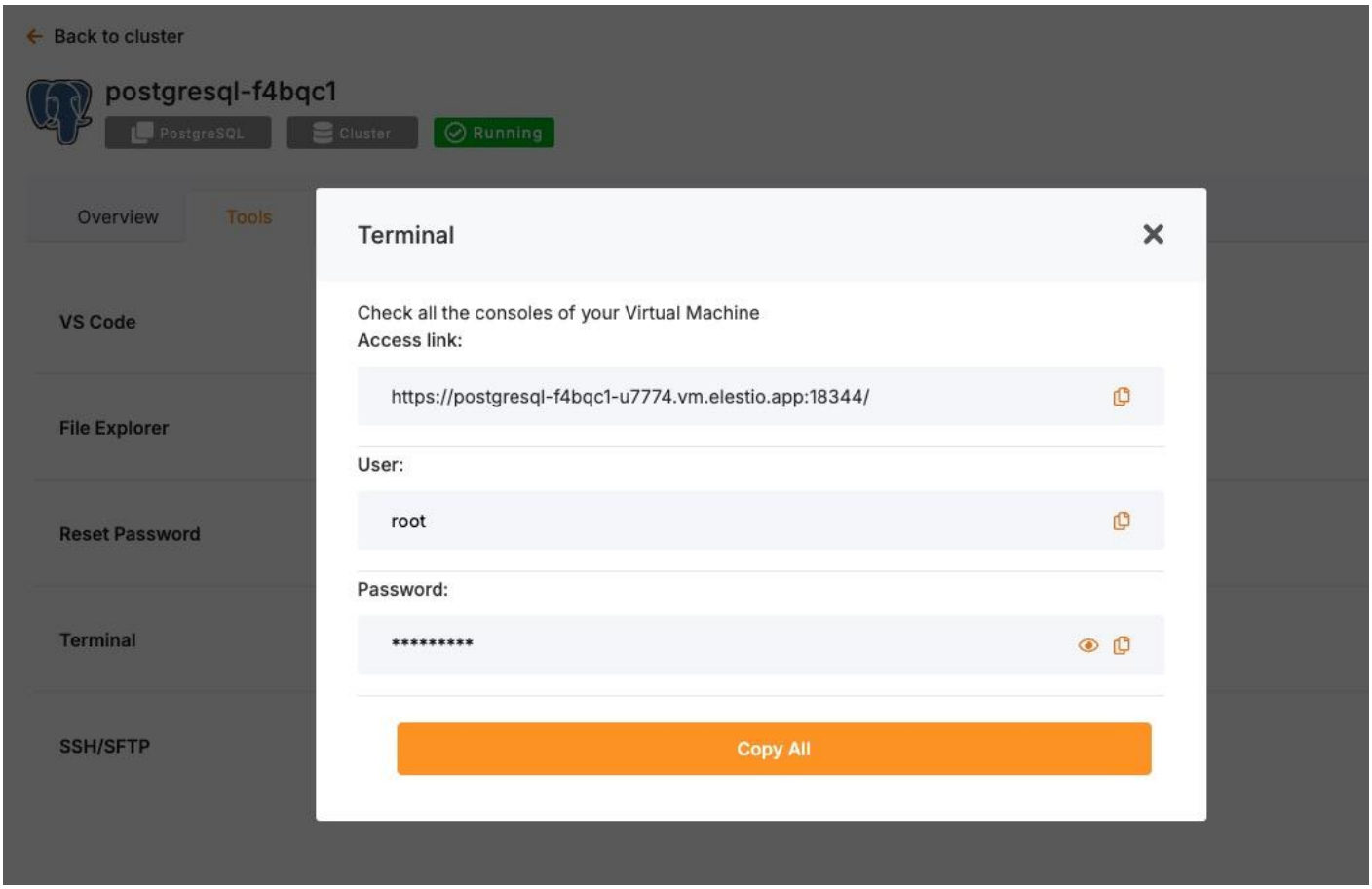
Manual Backups Using Docker Compose

If your PostgreSQL database is deployed through a Docker Compose setup on Elestio, you can run the `pg_dump` command from within the running container. This is useful when the tools are installed

inside the container environment and you want to keep everything self-contained. The backup can be created inside the container and then copied to your host system for long-term storage or transfer.

Access Elestio Terminal

Head over to your deployed PostgreSQL service dashboard and head over to **Tools > Terminal**. Use the credentials provided there to log in to your terminal.



Once you are in your terminal, run the following command to head over to the correct directory to perform the next steps

```
cd /opt/app/
```

Run `pg_dump` Inside the Container

This command runs `pg_dump` from inside the container and saves the backup to a file in `/tmp`. Make sure you have the following things in command in your env, else replace them with actual values and not the env variables.

```
docker-compose exec postgres \
  bash -c "PGPASSWORD='\$POSTGRES_PASSWORD' pg_dump -U \$POSTGRES_USER -Fc -v \$POSTGRES_DB > /tmp/manual_backup.dump"
```

This assumes that environment variables like `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB` are defined in your Compose setup.

Copy Backup to Host

After creating the backup inside the container, use `docker cp` to copy the file to your host machine.

```
docker cp $(docker-compose ps -q postgres):/tmp/manual_backup.dump ./manual_backup.dump
```

This creates a local copy of the backup file, which you can then upload to external storage or keep for versioned snapshots.

Backup Storage & Retention Best Practices

Once backups are created, they should be stored securely and managed with a clear retention policy. Proper naming, encryption, and rotation reduce the risk of data loss and help during recovery. Use timestamped filenames to identify when the backup was created. External storage services such as AWS S3, Backblaze B2, or an encrypted server volume are recommended for long-term storage.

Here are some guidelines to follow:

- Name your backups clearly: `mydb_backup_2024_04_02.dump`.
- Store in secure, off-site storage if possible.
- Retain 7 daily backups, 4 weekly backups, and 3–6 monthly backups.
- Remove old backups automatically to save space.

By combining storage hygiene with regular scheduling, you can maintain a reliable backup history and reduce manual effort.

Automating Manual Backups (cron)

Manual backup commands can be scheduled using tools like cron on Linux-based systems. This allows you to regularly back up your database without needing to run commands manually. Automating the process also reduces the risk of forgetting backups and ensures more consistent retention.

Example: Daily Backup at 2 AM

Open your crontab file for editing:

```
crontab -e
```

Then add a job like the following:

```
0 2 * * * PGPASSWORD='mypassword' pg_dump -U elestio -h db-xyz.elestio.app -p 5432 -Fc -f /backups/backup_$(date +\%F).dump mydatabase
```

Make sure the `/backups/` directory exists and is writable by the user running the job. You can also compress the backup and upload it to a remote destination as part of the same script.

Restoring a Backup

Restoring backups is essential for recovery, environment duplication, or rollback scenarios. Elestio supports restoring backups both through its built-in dashboard and via command-line tools like `pg_restore` psql. You can also restore from inside Docker Compose environments. This guide provides detailed steps for full and partial restores using each method and explains how to address common errors that occur during restoration.

Restoring from a Backup via Terminal

This method is used when you've created a `.dump` file using `pg_dump` in custom format. You can restore it using `pg_restore`, which gives you fine-grained control over what gets restored. This is useful for restoring backups to new environments, during version upgrades, or testing data locally.

Create the target database if it does not exist

If the database you're restoring into doesn't already exist, you must create it first.

```
PGPASSWORD='<your-password>' createdb \
  -U <username> \
  -h <host> \
  -p <port> \
  <database_name>
```

Run pg_restore to import the backup

This command restores the full contents of the .dump file into the specified database.

```
PGPASSWORD='<your-password>' pg_restore \
  -U <username> \
  -h <host> \
  -p <port> \
  -d <database_name> \
  -v <backup_file>.dump
```

You can add `--clean` to drop existing objects before restoring.

Restoring via Docker Compose

If your PostgreSQL service is deployed using Docker Compose, you can restore the database inside the container environment. This is useful when PostgreSQL runs in an isolated Docker setup, and you want to handle all backup and restore processes inside that environment.

Copy the backup into the container

Use docker cp to move the `.dump` file from your host machine to the PostgreSQL container.

```
docker cp ./manual_backup.dump $(docker-compose ps -q postgres):/tmp/manual_backup.dump
```

Run the restore inside the container

Use pg_restore from within the container to restore the file to the database.

```
docker-compose exec postgres \
  bash -c "PGPASSWORD='\$POSTGRES_PASSWORD' pg_restore -U \$POSTGRES_USER -d \$POSTGRES_DB -Fc
-v /tmp/manual_backup.dump"
```

Make sure your environment variables in the Docker Compose file match the values used here.

Partial Restores

PostgreSQL supports partial restores, allowing you to restore only selected tables, schemas, or schema definitions. This can be useful when recovering a specific part of the database or testing part of the data.

Restore a specific table

Use the -t flag to restore only one table from the `.dump` file.

```
PGPASSWORD='<your-password>' pg_restore \
  -U <username> \
  -h <host> \
  -p <port> \
  -d <database_name> \
  -t <table_name> \
  -v <backup_file>.dump
```

Restore schema only (no data)

This command will restore only the table structures, types, functions, and other schema definitions without inserting any data.

```
pg_restore \
  -U <username> \
  -h <host> \
  -p <port> \
  -d <database_name> \
  --schema-only \
  -v <backup_file>.dump
```

Partial restores work best with custom-format .dump files generated by `pg_dump` `-Fc`.

Common Errors & How to Fix Them

Errors during restore are often caused by permission issues, incorrect formats, or missing objects. Understanding the error messages and their causes will help you recover faster and avoid data loss.

1. Could not connect to database

```
pg_restore: [archiver] could not connect to database
```

This usually happens if the database doesn’t exist or the credentials are incorrect. Make sure the database has been created and the connection details are correct.

2. Permission denied for schema

```
ERROR: permission denied for schema public
```

This error indicates that the user account used for restore lacks the privileges needed to write into the schema. Use a superuser account or adjust the schema permissions before restoring.

3. Input file appears to be a text format dump

```
pg_restore: error: input file appears to be a text format dump
```

This means you are trying to use `pg_restore` a plain SQL file. In this case, you should use psql instead:

```
psql -U <username> -h <host> -p <port> -d <database_name> -f backup.sql
```

4. Duplicate key value violates unique constraint

This occurs when the restore process tries to insert rows that already exist in the target database. You can either drop the target database before restoring or use `--clean` it in `pg_restore` to drop existing objects automatically.

Identifying Slow Queries

Slow queries can significantly affect application performance and user experience. PostgreSQL offers built-in tools to analyze and identify these slow operations. On Elestio, whether you're connected via terminal, inside a Docker Compose container, or using PostgreSQL CLI tools, you can use several methods to pinpoint and fix performance issues. This guide walks through various techniques to identify slow queries, interpret execution plans, and apply optimizations.

Analyzing Slow Queries Using Terminal

When connected to your PostgreSQL service via terminal, you can use built-in tools like psql and SQL functions to observe how queries behave. This method is useful for immediate, ad hoc diagnostics in production or staging environments. You can use simple commands to view currently running queries, analyze individual query plans, and measure runtime performance. These steps help determine which queries are taking the most time and why.

Use psql to connect directly to your PostgreSQL instance. This provides access to administrative and diagnostic SQL commands.

```
psql -U <username> -h <host> -d <database>
```

Now use the following command to show the query plan the database will use. It highlights whether PostgreSQL will perform a sequential scan, index scan, or other operation.

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 42;
```

Another type of command that executes the query and returns actual runtime and row counts. Comparing planned and actual rows helps determine if the planner is misestimating costs.

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE customer_id = 42;
```

Lastly, monitor queries in real time using the following command. This view lists all active queries, sorted by duration. It helps you identify queries that are taking too long and might need optimization.

```
SELECT pid, now() - query_start AS duration, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY duration DESC;
```

Analyzing Slow Queries in Docker Compose Environments

If your PostgreSQL is deployed using Docker Compose on Elestio, you can inspect and troubleshoot slow queries from within the container. This method is useful when the PostgreSQL instance is isolated inside a container and not accessible directly from the host. Logs and query data can be collected from inside the service container using PostgreSQL tools or by checking configuration files.

```
docker-compose exec postgres bash
```

This command opens a shell inside the running PostgreSQL container. From here, you can run commands like psql or view logs. Use the same psql interface from inside the container to interact with the database and execute analysis commands.

```
psql -U $POSTGRES_USER -d $POSTGRES_DB
```

Next, edit postgresql.conf inside the container to enable slow query logging:

```
log_min_duration_statement = 500
log_statement = 'none'
```

This setting logs all queries that take longer than 500 milliseconds. You may need to restart the container for these settings to take effect.

Using CLI Tools to Analyze Query Performance

PostgreSQL offers CLI-based tools and extensions like pg_stat_statements for long-term query performance analysis. These tools provide aggregated metrics over time, helping you spot frequently executed but inefficient queries. This section shows how to use PostgreSQL extensions and views to collect detailed statistics.

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
```

This extension logs each executed query along with performance metrics such as execution time and row count. The next command shows the queries that have consumed the most total execution time. These are strong candidates for indexing or rewriting.

```
SELECT query, calls, total_time, mean_time, rows
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

Understanding Execution Plans and Metrics

PostgreSQL’s query planner produces execution plans that describe how a query will be executed. Reading these plans can help identify operations that slow down performance, such as full table scans or repeated joins. Comparing estimated and actual rows processed can also reveal outdated statistics or inefficient filters. Understanding these elements is key to choosing the right optimization strategy.

Key elements to understand:

- **Seq Scan**: A full table scan; slow on large tables unless indexed.
- **Index Scan**: Uses an index for fast lookup; typically faster than a sequential scan.
- **Cost**: Estimated cost of the query, used by the planner to decide the best execution path.
- **Rows**: Estimated vs. actual rows; large mismatches indicate bad planning or outdated stats.
- **Execution Time**: Total time it took to run the query; from `EXPLAIN ANALYZE`.

Use these metrics to compare how the query was expected to run versus how it actually performed.

Optimizing Queries for Better Performance

Once you’ve identified slow queries, the next step is to optimize them. Optimizations may involve adding indexes, rewriting SQL statements, or updating statistics. The goal is to reduce scan times, avoid redundant operations, and guide the planner to more efficient execution paths. Performance tuning is iterative—test after each change.

Common optimization steps:

- **Add indexes** to columns used in WHERE, JOIN, and ORDER BY clauses.
- **Use EXPLAIN ANALYZE** before and after changes to measure impact.
- **Avoid SELECT *** to reduce data transfer and memory use.
- **Use LIMIT** to restrict row output when only a subset is needed.
- **Run ANALYZE** to update PostgreSQL's internal statistics and improve planner accuracy:

```
ANALYZE;
```

By focusing on frequent and long-running queries, you can make improvements that significantly reduce overall load on the database.

Detect and terminate long-running queries

Long-running queries can significantly impact database performance by consuming CPU, memory, and I/O resources over extended periods. In production environments like Elestio, it’s important to monitor for these queries and take timely action to terminate them when necessary. PostgreSQL provides built-in monitoring tools and system views to help detect problematic queries and respond accordingly. This guide covers how to identify and cancel long-running queries using PostgreSQL’s terminal tools, Docker Compose environments, and logging features, along with preventive practices.

Identifying Long-Running Queries via Terminal

When connected to your PostgreSQL service through the terminal using psql, you can check which queries are running and how long they have been active. This can help identify queries that are stuck, inefficient, or blocked.

To list all active queries sorted by duration, you can use:

```
SELECT pid, now() - query_start AS duration, state, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY duration DESC;
```

This query reveals which operations have been running the longest and their current state. If you want to isolate queries that have exceeded a specific duration (e.g., 1 minute), add a time filter:

```
SELECT pid, now() - query_start AS runtime, query
FROM pg_stat_activity
WHERE state = 'active' AND now() - query_start > interval '1 minute';
```

These queries help you locate potential performance bottlenecks in real time.

Terminating Long-Running Queries Safely

Once a problematic query is identified, PostgreSQL allows you to cancel it using the pid (process ID). If you want to cancel the query without affecting the client session, use:

```
SELECT pg_cancel_backend(<pid>);
```

This tells PostgreSQL to stop the running query, but keep the session connected. If the query is unresponsive or the client is idle for too long, you can fully terminate the session using:

```
SELECT pg_terminate_backend(<pid>);
```

This forcibly closes the session and stops the query. Termination should be used cautiously, especially in shared application environments.

Working Within Docker Compose Environments

If PostgreSQL is deployed using Docker Compose on Elestio, you can detect and manage queries from inside the container. Start by entering the container:

```
docker-compose exec postgres bash
```

Inside the container, connect to the database with:

```
psql -U $POSTGRES_USER -d $POSTGRES_DB
```

From here, you can use the same commands as above to monitor and cancel long-running queries. The logic remains the same; you’re simply operating inside the container’s shell environment.

Using Logs and Monitoring Tools

PostgreSQL supports logging queries that exceed a certain duration threshold, which is useful for long-term monitoring and post-incident review. To enable this, modify your postgresql.conf file and set:

```
log_min_duration_statement = 500
```

This setting logs every query that takes longer than 500 milliseconds. The logs are written to PostgreSQL's log files, which you can access through the Elestio dashboard (if supported) or inside the container under the PostgreSQL data directory.

For cumulative insights, enable the pg_stat_statements extension to track long-running queries over time:

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
```

Then query the collected data:

```
SELECT query, total_time, mean_time, calls
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

This shows which queries are consistently expensive, not just slow once.

Best Practices to Prevent Long-Running Queries

Preventing long-running queries is more effective than terminating them after the fact. Start by indexing columns used in WHERE, JOIN, and ORDER BY clauses. Use query analysis tools like EXPLAIN ANALYZE to find out how queries are executed and where performance issues may occur.

Also, consider setting timeouts for queries. At the session level, you can use:

```
SET statement_timeout = '2s';
```

This automatically cancels any query that runs longer than 2 seconds. For applications, set timeout configurations in the client or ORM layer to ensure they don't wait indefinitely on slow queries. Monitoring tools and alerts can help you detect abnormal query behavior early. If you're managing your own monitoring stack, connect it to PostgreSQL logs or pg_stat_activity to trigger alerts for long-running operations.

Preventing Full Disk Issues

Running out of disk space in a database environment can lead to failed writes, service downtime, and even data corruption. PostgreSQL systems require available space not only for storing data but also for managing temporary files, WAL logs, indexes, and routine background tasks. On Elestio, while infrastructure is managed, you are still responsible for monitoring growth and preventing overuse. This guide outlines how to monitor disk usage, configure alerts, automate cleanup, and follow best practices to avoid full disk conditions in PostgreSQL.

Monitoring Disk Usage

Proactively monitoring disk usage helps you detect unusual growth in time to act. Whether you're accessing your database directly via the terminal or through a Docker Compose environment, several built-in tools can provide usage stats and trends. Combining filesystem-level monitoring with PostgreSQL-specific checks gives a complete view of space utilization

To check the overall disk usage of the system from a terminal or container:

```
df -h
```

This command shows available space for each mounted volume. Focus on the mount point where your PostgreSQL data directory is stored, usually `/var/lib/postgresql`.

For detailed PostgreSQL-specific usage, connect to your database and run:

```
SELECT pg_size_pretty(pg_database_size(current_database()));
```

This shows the total size used by the active database. You can also analyze individual tables and indexes using:

```
SELECT relname AS object, pg_size_pretty(pg_total_relation_size(relid)) AS size
FROM pg_catalog.pg_statio_user_tables
ORDER BY pg_total_relation_size(relid) DESC
LIMIT 10;
```

This query highlights the largest tables by size, helping you identify which parts of your schema consume the most space.

Configuring Alerts and Cleanup

Even with monitoring in place, automatic alerts and cleanup scripts ensure you act before hitting disk limits. You can set up external monitoring agents or run container-level scripts to track disk usage and notify you.

If you're using Docker Compose, you can monitor container-level storage stats using:

```
docker system df
```

This command provides an overview of Docker volumes, images, and container usage. To monitor and clean unused volumes and logs manually:

```
docker volume ls
docker volume rm <volume-name>
```

Make sure you're not deleting active database volumes. Always verify that backups exist and are up-to-date before running cleanup commands.

To configure PostgreSQL-specific cleanup, enable auto-vacuum and monitor its effectiveness. PostgreSQL removes dead tuples and reclaims space using this process. Check the vacuum activity with:

```
SELECT relname, n_dead_tup, last_vacuum, last_autovacuum
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;
```

If dead tuples accumulate, increase autovacuum frequency or run a manual vacuum:

```
VACUUM ANALYZE;
```

Autovacuum settings can also be tuned in postgresql.conf to trigger more aggressively based on table activity.

Best Practices for Disk Space Management

- Beyond immediate cleanup, long-term strategies help keep disk usage under control. These include data retention policies, partitioning, compression, and regular maintenance. It's also important to have a growth plan based on usage trends.

- Avoid storing large binary objects like images or PDFs directly in the database. Use object storage for large files and reference them by URL. If historical data is no longer needed for queries, archive it into a separate cold-storage database or export to files.
- Partition large tables by time or ID ranges to manage growth and make pruning easier. Use tools like `pg_partman` native PostgreSQL table partitioning to automatically offload older data
- Regularly rotate and clean up PostgreSQL logs and WAL files. If using archive mode, ensure archived WALs are uploaded and removed from disk after successful backup.
- To keep your setup safe, also monitor backup file sizes and locations. Backups stored on the same volume as the database may consume critical space. If possible, push backups to remote object storage or another disk volume.

Checking Database Size and Related Issues

As your PostgreSQL database grows over time, it’s important to monitor its size and identify what parts of the database consume the most space. Unmanaged growth can lead to performance issues, disk exhaustion, and backup delays. On Elestio, where PostgreSQL is hosted in a managed environment, you can use SQL and command-line tools to measure database usage, analyze large objects, and troubleshoot storage problems. This guide explains how to check database size, detect bloated tables and indexes, and optimize storage usage efficiently.

Checking Database and Table Sizes

PostgreSQL provides built-in functions to report the size of the current database, its individual schemas, tables, and indexes. These functions are useful for understanding where most of your storage is being used and planning cleanup or archiving strategies.

To check the total size of the active database:

```
SELECT pg_size_pretty(pg_database_size(current_database()));
```

This returns a human-readable value like “2 GB”, indicating how much space the entire database consumes on disk.

To list the largest tables in your schema:

```
SELECT relname AS table, pg_size_pretty(pg_total_relation_size(relid)) AS total_size
FROM pg_catalog.pg_statio_user_tables
ORDER BY pg_total_relation_size(relid) DESC
LIMIT 10;
```

This helps you identify which tables take up the most space, including indexes and TOAST (large field) data.

To break down table vs index size separately:

```
SELECT relname AS object,
       pg_size_pretty(pg_relation_size(relid)) AS table_size,
```

```
       pg_size_pretty(pg_indexes_size(relid)) AS index_size
FROM pg_catalog.pg_statio_user_tables
ORDER BY pg_relation_size(relid) DESC
LIMIT 10;
```

This distinction allows you to assess whether most space is used by raw table data or indexes, which can inform optimization decisions.

Identifying Bloat and Inefficiencies

Database bloat occurs when PostgreSQL retains outdated or deleted rows due to its MVCC model. This is common in high-write tables and can lead to wasted space and degraded performance. Bloated indexes and tables are often invisible unless explicitly checked. To estimate bloat at a table level, you can use a community query like this:

```
SELECT schemaname, relname, round(100 * (pg_total_relation_size(relid) -
pg_relation_size(relid)) / pg_total_relation_size(relid), 2) AS bloat_pct
FROM pg_catalog.pg_statio_user_tables
ORDER BY bloat_pct DESC
LIMIT 10;
```

This query calculates how much of a table’s total size is not accounted for by its base data—higher percentages suggest unused or dead space. You can also check dead tuples directly:

```
SELECT relname, n_dead_tup
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 10;
```

A high count of dead tuples suggests that autovacuum might not be keeping up and that a manual VACUUM could help.

Optimizing and Reducing Database Size

Once you’ve identified large or bloated objects, the next step is to optimize them. PostgreSQL offers tools like VACUUM, REINDEX, and CLUSTER to reclaim space and improve storage efficiency. These commands must be run with care to avoid locking critical tables during active hours. To reclaim dead tuples and update statistics:

```
VACUUM ANALYZE;
```

This command removes dead rows and refreshes query planning statistics, which helps performance and frees up storage. To shrink large indexes that aren’t cleaned automatically, use:

```
REINDEX TABLE <table_name>;
```

This recreates the table’s indexes from scratch and can free up disk space if indexes are fragmented or bloated. If a table is heavily bloated and full table rewrites are acceptable during maintenance, use:

```
CLUSTER <table_name>;
```

This rewrites the entire table based on an index order and reclaims space similar to VACUUM FULL, but with more control.

Additionally, removing or archiving old data from large time-based tables can reduce total size. Consider partitioning large tables to manage this process more efficiently.

Database Migration

Database Migration Service for PostgreSQL

Elestio provides a structured approach for migrating PostgreSQL databases from various environments, such as on-premises systems or other cloud platforms, to its managed services. This process ensures data integrity and minimizes downtime, facilitating a smooth transition to a managed environment.

Key Steps in Migrating to Elestio

Pre-Migration Preparation

Before initiating the migration process, it's essential to undertake thorough preparation to ensure a smooth transition:

- **Create an Elestio Account:** Register on the Elestio platform to access their suite of managed services. This account will serve as the central hub for managing your PostgreSQL instance and related resources.
- **Deploy the Target PostgreSQL Service:** Set up a new PostgreSQL instance on Elestio to serve as the destination for your data. It's crucial to match the software version of your current PostgreSQL database to avoid compatibility issues during data transfer. Detailed prerequisites and guidance can be found in Elestio's migration documentation.

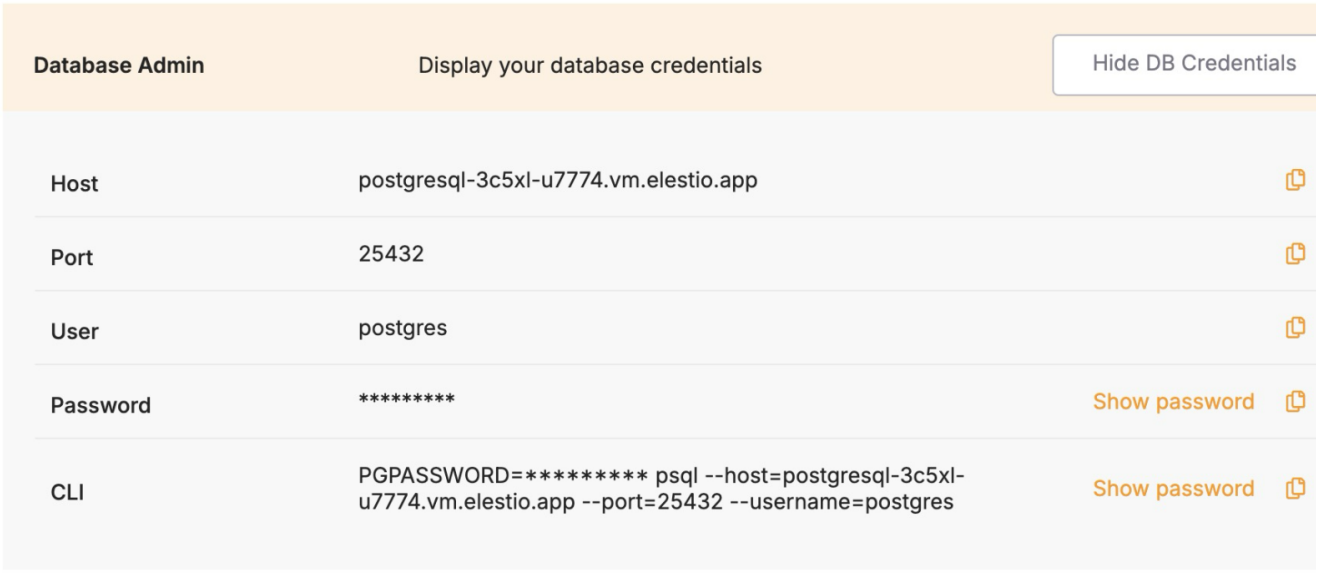
Initiating the Migration Process

With the preparatory steps completed, you can proceed to migrate your PostgreSQL database to Elestio:

1. **Access the Migration Tool:** Navigate to the overview of your PostgreSQL service on the Elestio dashboard. Click on the "Migrate Database" button to initiate the migration process. This tool is designed to facilitate a smooth transition by guiding you through each step.
2. **Configure Migration Settings:** A modal window will open, prompting you to ensure that your target service has sufficient disk space to accommodate your database. Adequate storage is vital to prevent interruptions during data transfer. Once confirmed, click on the "Get started" button to proceed.
3. **Validate Source Database Connection:** Provide the connection details for your existing PostgreSQL database, including:

- **Hostname:** The address of your current database server.
- **Port:** The port number on which your PostgreSQL service is running (default is 5432).
- **Database Name:** The name of the database you intend to migrate.
- **Username:** The username with access privileges to the database.
- **Password:** The corresponding password for the user.

After entering these details, click on "Run Check" to validate the connection. This step ensures that Elestio can securely and accurately access your existing data. You can find these details under Database admin section under your deployed PostgreSQL service.



4. **Execute the Migration:** If all checks pass without errors, initiate the migration by selecting "Start migration." Monitor the progress through the real-time migration logs displayed on the dashboard. This transparency allows for immediate detection and resolution of any issues, ensuring data integrity throughout the process.

Post-Migration Validation and Optimization

After completing the migration, it's crucial to perform validation and optimization tasks to ensure the integrity and performance of your database in the new environment:

- **Verify Data Integrity:** Conduct thorough checks to ensure all data has been accurately transferred. This includes comparing row counts, checksums, and sample data between the source and target databases. Such verification maintains the reliability of your database and ensures that no data was lost or altered during migration.
- **Test Application Functionality:** Ensure that applications interacting with the database function correctly in the new environment. Update connection strings and configurations as necessary to reflect the new database location. This step prevents potential disruptions and ensures seamless operation of dependent systems.
- **Optimize Performance:** Utilize Elestio's managed service features to fine-tune database performance. Set up automated backups to safeguard your data, monitor resource utilization to identify and address bottlenecks, and configure scaling options to accommodate future growth. These actions contribute to improved application

responsiveness and overall system efficiency.
- **Implement Security Measures:** Review and configure security settings to protect your data within the Elestio environment. Set up firewalls to control access, manage user access controls to ensure only authorized personnel can interact with the database, and enable encryption where applicable to protect data at rest and in transit. Implementing these security measures safeguards your data against unauthorized access and potential threats.

Benefits of Using Elestio for PostgreSQL

Migrating your PostgreSQL database to Elestio offers several advantages:

- **Simplified Management:** Elestio automates database maintenance tasks, including software updates, backups, and system monitoring, reducing manual work. The platform provides a dashboard with real-time insights into database performance and resource usage. It allows for adjusting service plans, scaling CPU and RAM as needed. Users can modify environment variables and access software information to manage configurations.

- **Security:** Elestio keeps PostgreSQL instances updated with security patches to protect against vulnerabilities. The platform automates backups to ensure data integrity and availability. It provides secure access mechanisms, including randomly generated passwords for database instances, which can be managed through the dashboard.

- **Performance:** Elestio configures PostgreSQL instances for performance based on workload requirements. The platform supports the latest PostgreSQL versions, incorporating updates that improve database operations. Its infrastructure handles different workloads and maintains performance during high usage periods.

- **Scalability:** Elestio's PostgreSQL service allows for scaling database resources to handle growth and changing workloads without major downtime. Users can upgrade or downgrade service plans, adjusting CPU and RAM as needed. The platform supports adding network volumes to increase storage capacity.

Cloning a Service to Another Provider or Region

Migrating or cloning services across cloud providers or geographic regions is a critical part of modern infrastructure management. Whether you're optimizing for latency, preparing for disaster recovery, meeting regulatory requirements, or simply switching providers, a well-planned migration ensures continuity, performance, and data integrity. This guide outlines a structured methodology for service migration, applicable to most cloud-native environments.

Pre-Migration Preparation

Before initiating a migration, thorough planning and preparation are essential. This helps avoid unplanned downtime, data loss, or misconfiguration during the move:

- **Evaluate the Current Setup**: Begin by documenting the existing service's configuration. This includes runtime environments (container images, platform versions), persistent data (databases, object storage), network rules (ports, firewalls), and application dependencies (APIs, credentials, linked services).
- **Define the Migration Target**: Choose the new cloud provider or region you plan to migrate to. Confirm service compatibility, resource limits, and geographic latency requirements. If you're replicating an existing environment, make sure the target region supports the same compute/storage features and versions.
- **Provision the Target Environment**: Set up the target infrastructure where the service will be cloned. This could involve creating new Kubernetes clusters, VM groups, container registries, databases, or file storage volumes—depending on your stack.
- **Backup the Current Service**: Always create a full backup or snapshot of the current service and its associated data before proceeding. This acts as a rollback point in case of migration issues and ensures recovery in the event of failure.

Cloning Execution

The first step in executing a clone is to replicate the configuration of the original service in the target environment. This involves deploying the same container image or service binary using the same runtime settings. If you're using Kubernetes or container orchestrators, this can be done via Helm charts or declarative manifests. Pay close attention to environment variables, secrets, mounted paths, storage class definitions, and health check configurations to ensure a consistent runtime environment.

Next, you'll need to migrate any persistent data tied to the service. For PostgreSQL databases, this might involve using pg_dump to export the schema and data, followed by psql or pg_restore to import it into the new instance. In more complex cases, tools like pgBackRest, wal-g, or logical replication can be used to minimize downtime during the switchover. For file-based storage, tools like rsync or rclone are effective for copying volume contents over SSH or cloud storage backends. It's crucial to verify compatibility across disk formats, database versions, and encoding standards to avoid corruption or mismatched behavior.

After replicating the environment and data, it's important to validate the new service in isolation. This means confirming that all application endpoints respond as expected, background tasks or cron jobs are functioning, and third-party integrations (e.g., payment gateways, S3 buckets) are accessible. You should test authentication flows, data read/write operations, and retry logic to ensure the new service is functionally identical. Use observability tools to monitor resource consumption and application logs during this stage.

Once validation is complete, configure DNS and route traffic to the new environment. This might involve updating DNS A or CNAME records, changing cloud load balancer configurations, or applying new firewall rules. For high-availability setups, consider using health-based routing or weighted DNS to gradually transition traffic from the old instance to the new one.

Post-Migration Validation and Optimization

Once the new environment is live and receiving traffic, focus on optimizing and securing the setup:

- **Validate Application Functionality**: Test all integrations, user workflows, and background jobs to confirm proper behavior. Review logs for silent errors or timeouts. Ensure all applications pointing to the service are updated with the new URL or connection string.
- **Monitor Performance**: Analyze load, CPU, memory, and storage utilization. Scale resources as needed, or optimize runtime settings for the new provider/region. Enable autoscaling where applicable.
- **Secure the Environment**: Implement firewall rules, IP restrictions, and access controls. Rotate secrets and validate that no hardcoded credentials or endpoints point to the old service.
- **Cleanup and Documentation**: Once validated, decommission the old setup safely. Update internal documentation with new deployment details, endpoint addresses, and any configuration changes.

Benefits of Cloning

Cloning a database service, particularly for engines like PostgreSQL offers several operational and strategic advantages. It allows teams to test schema migrations, version upgrades, or major application features in an isolated environment without affecting production. By maintaining a cloned copy, developers and QA teams can work against realistic data without introducing risk.

Cloning also simplifies cross-region redundancy setups. A replica in another region can be promoted quickly if the primary region experiences an outage. For compliance or analytics purposes, cloned databases allow for read-only access to production datasets, enabling safe reporting or data processing without interrupting live traffic.

Additionally, rather than building a new environment from scratch, you can clone the database into another provider, validate it, and cut over with minimal disruption. This helps maintain operational continuity and reduces the effort needed for complex migrations.

Manual Migration Using pg_dump and pg_restore

Manual Migrations using PostgreSQL's built-in tools `pg_dump` and `pg_restore` are ideal for users who prefer full control over data export and import, particularly during provider transitions, database version upgrades, or when importing an existing self-managed PostgreSQL dataset into Elestio's managed environment. This guide walks through the process of performing a manual migration to and from Elestio PostgreSQL services using command-line tools, ensuring that your data remains portable, auditable, and consistent.

When to Use Manual Migration

Manual migration using `pg_dump` and `pg_restore` is well-suited for scenarios where full control over the data export and import process is required. This method is particularly useful when migrating from an existing PostgreSQL setup, whether self-hosted, on-premises, or on another cloud provider, into Elestio's managed PostgreSQL service. It allows for one-time imports without requiring continuous connectivity between source and target systems.

This approach is also ideal when dealing with version upgrades, as PostgreSQL's logical backups can be restored into newer versions without compatibility issues. In situations where Elestio's built-in snapshot or replication tools aren't applicable such as migrations from isolated environments or selective schema transfers, manual migration becomes the most practical option. Additionally, this method enables users to retain portable, versioned backups outside of Elestio's infrastructure, which can be archived, validated offline, or re-imported into future instances.

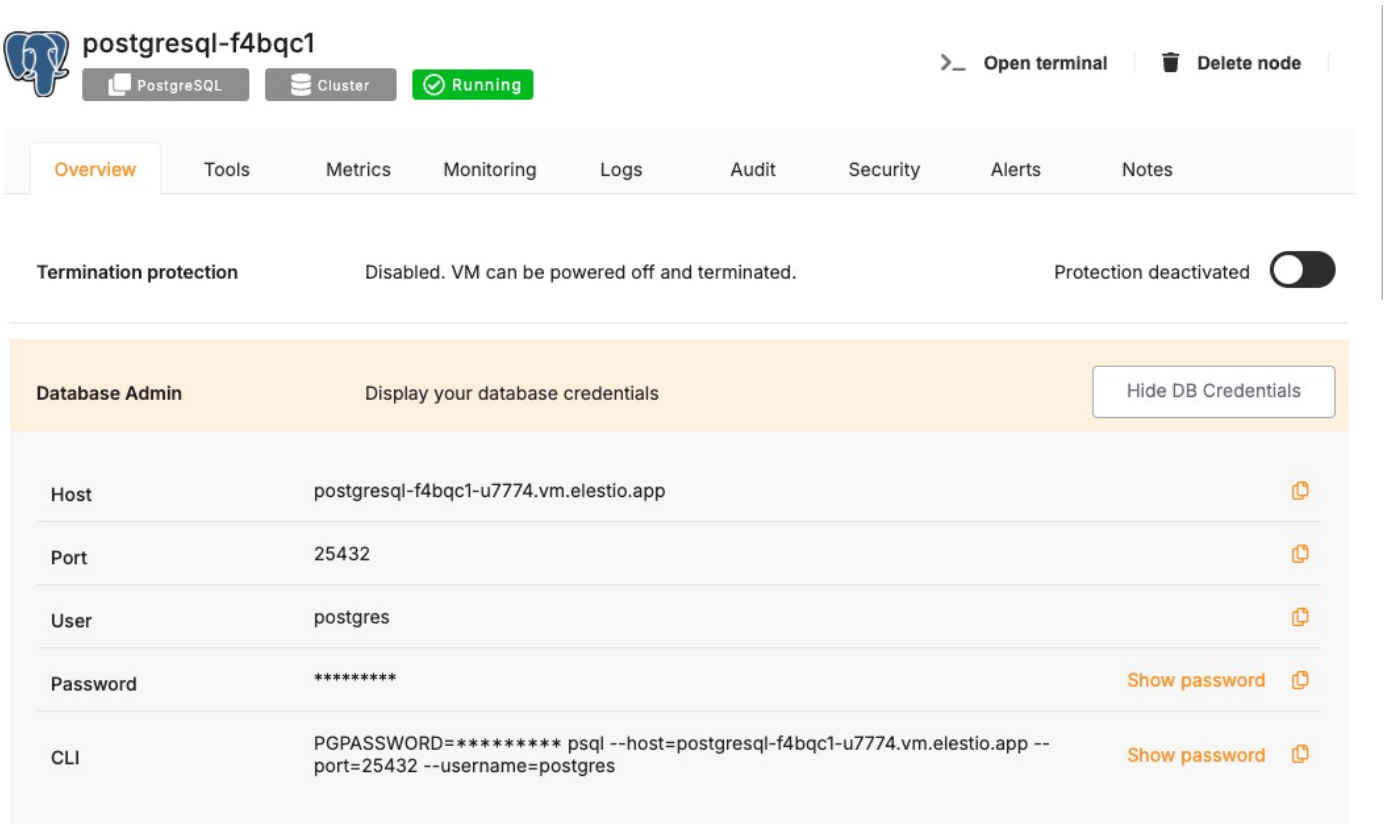
Performing the Migration

Prepare the Environments

Before initiating a migration, verify that PostgreSQL is properly installed and configured on both the source system and your Elestio service. On the source, you need an active PostgreSQL instance with a user account that has sufficient privileges to read schemas, tables, sequences, and any installed extensions. The user must also be allowed to connect over TCP if the server is remote.

On the Elestio side, provision a PostgreSQL service from the dashboard. Once deployed, retrieve the connection information from the Database admin tab. This includes the hostname, port, database name, username, and password. You'll use these credentials to connect during the

restore step. Ensure that your IP is allowed to connect under the Cluster Overview > Security > Limit access per IP section; otherwise, the PostgreSQL port will be unreachable during the migration.



Create a Dump Using pg_dump

In this step, you generate a logical backup of the source database using `pg_dump`. This utility connects to the PostgreSQL server and extracts the structure and contents of the specified database. It serializes tables, indexes, constraints, triggers, views, and functions into a consistent snapshot. The custom format (-Fc) is used because it produces a compressed binary dump that can be restored selectively using pg_restore.

```
pg_dump -U <source_user> -h <source_host> -p <source_host> -Fc <source_database> > backup.dump
```

This command connects to the source server (-h), authenticates with the user (-U), targets the database (source_database), and exports the entire schema and data into `backup.dump`. The resulting file is portable and version-aware. You can also add `--no-owner` and `--no-acl` If you're migrating between environments that use different database roles or access models. This prevents restore-time errors related to ownership mismatches.

Transfer the Dump File to the Target

If your source and target environments are on different hosts, the dump file must be transferred securely. This step ensures the logical backup is available on the system from which you'll perform the restore. You can use secure copy (scp), rsync, or any remote file transfer method.

```
scp backup.dump your_user@your_workstation:/path/to/local/
```

If restoring from your local machine to Elestio, ensure the dump file is stored in a location readable by your current shell user. Elestio does not require the file to be uploaded to its servers; the restore is performed by connecting over the network using standard PostgreSQL protocols. At this point, your backup is isolated from the source environment and ready for import.

Create the Target Database

By default, Elestio provisions a single database instance. However, if you wish to restore into a separate database name or if your dump references a different name, you must create the new database manually. Use the psql client to connect to your Elestio service using the credentials from the dashboard.

```
psql -U <elestio_user> -h <elestio_host> -p <elestio_host> -d postgres
```

Within the psql session, create the database:

```
CREATE DATABASE target_database WITH ENCODING='UTF8' LC_COLLATE='en_US.UTF-8'
LC_CTYPE='en_US.UTF-8' TEMPLATE=template0;
```

This ensures that the new database has consistent encoding and locale settings, which are critical for text comparison, sorting, and indexing. Using template0 avoids inheriting default extensions or templates that might conflict with your dump file. At this stage, you can also create any roles, schemas, or extensions that were used in the original database if they are not included in the dump.

Restore Using pg_restore

With the target database created and the dump file in place, initiate the restoration using `pg_restore`. This tool reads the custom-format archive and reconstructs all schema and data objects in the new environment.

```
pg_restore -U elestio_user -h elestio_host -p 5432 -d target_database -Fc /path/to/backup.dump
--verbose
```

This command establishes a network connection to the Elestio PostgreSQL service and begins issuing `CREATE`, `INSERT`, and `ALTER` statements to rebuild the database. The `--verbose` flag provides real-time feedback about the objects being restored. You can also use `--jobs=N` to run the restore in parallel, improving performance for large datasets, provided the dump was created with `pg_dump --jobs=N`.

It’s important to ensure that all referenced extensions, collations, and roles exist on the target instance to avoid partial restores. If errors occur, the logs will point to the missing components or permission issues that need to be resolved.

Validate the Migration

Once the restore completes, you must validate the accuracy and completeness of the migration. Connect to the Elestio database using psql or a PostgreSQL GUI (such as pgAdmin or TablePlus), and run checks across critical tables.

Begin by inspecting the table existence and row counts:

```
\dt
SELECT COUNT(*) FROM your_important_table;s
```

Validate views, functions, and indexes, especially if they were used in reporting or application queries. Run application-specific health checks, reinitialize ORM migrations if applicable, and confirm that the application can read and write to the new database without errors.

If you made any changes to connection strings or credentials, update your environment variables or secret managers accordingly. Elestio also supports automated backups, which you should enable post-migration to protect the restored dataset.

Benefits of Manual Migration

Manual PostgreSQL migration using `pg_dump` and `pg_restore` on Elestio provides several key advantages:

- **Compatibility and Portability**: Logical dumps allow you to migrate from any PostgreSQL-compatible source into Elestio, including on-premises systems, Docker containers, or other clouds.
- **Version-Safe Upgrades**: The tools support migrating across PostgreSQL versions, which is ideal during controlled upgrades.
- **Offline Archiving**: Manual dumps serve as portable archives for cold storage, disaster recovery, or historical snapshots.
- **Platform Independence**: You retain full access to PostgreSQL’s native tools without being locked into Elestio-specific formats or interfaces.

This method complements Elestio’s automated backup and migration features by enabling custom workflows and one-off imports with full visibility into each stage.

Cluster Management

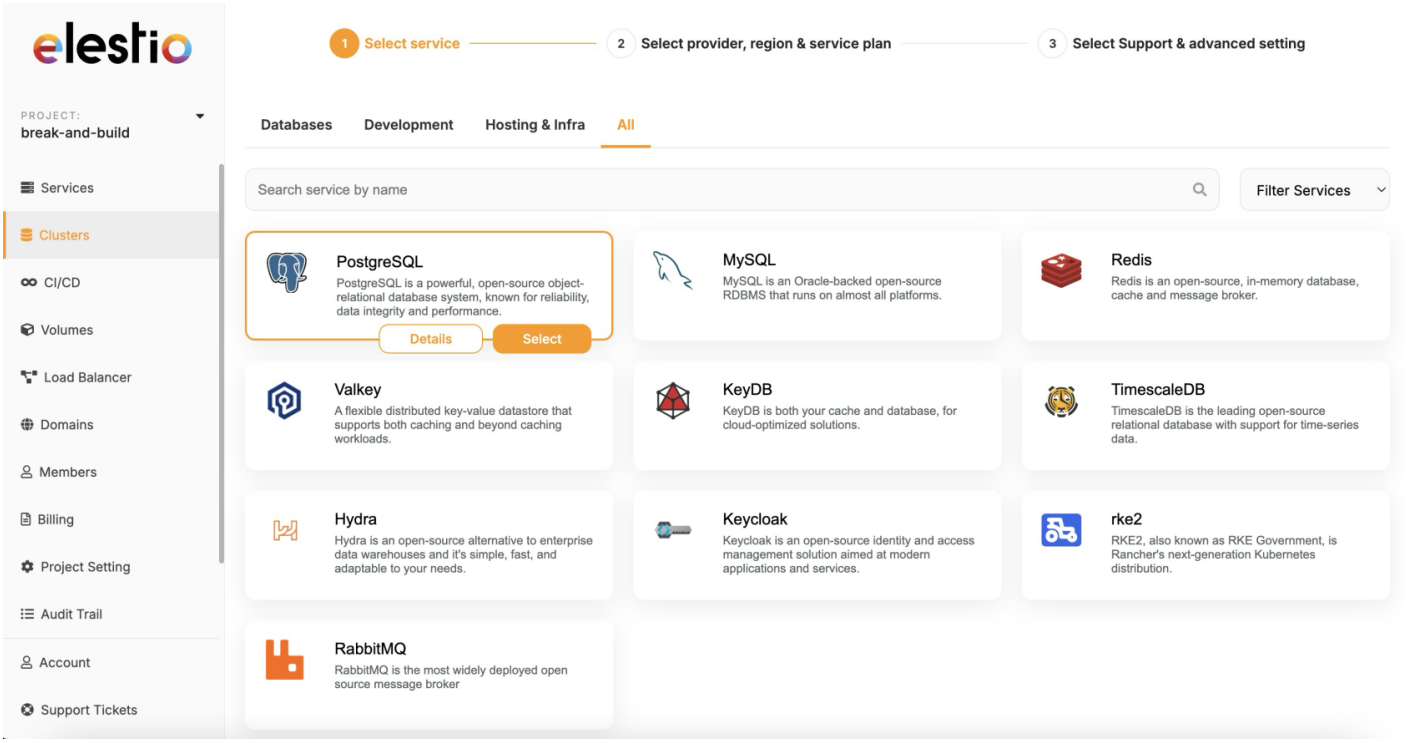
Overview

Elestio provides a complete solution for setting up and managing software clusters. This helps users deploy, scale, and maintain applications more reliably. Clustering improves performance and ensures that services remain available, even if one part of the system fails. Elestio supports different cluster setups to handle various technical needs like load balancing, failover, and data replication.

Supported Software for Clustering:

Elestio supports clustering for a wide range of open-source software. Each is designed to support different use cases like databases, caching, and analytics:

- **MySQL**:
  Supports Single Node, Primary/Replica, and Multi-Master cluster types. These allow users to create simple setups or more advanced ones where reads and writes are distributed across nodes. In a Primary/Replica setup, replicas are updated continuously through replication. These configurations are useful for high-traffic applications that need fast and reliable access to data.
- **PostgreSQL**:
  PostgreSQL clusters can be configured for read scalability and failover protection. Replication ensures that data written to the primary node is copied to replicas. Clustering PostgreSQL also improves query throughput by offloading read queries to replicas. Elestio handles replication setup and node failover automatically.
- **Redis/KeyDB/Valkey**:
  These in-memory data stores support clustering to improve speed and fault tolerance. Clustering divides data across multiple nodes (sharding), allowing horizontal scaling. These tools are commonly used for caching and real-time applications, so fast failover and data availability are critical.
- **Hydra and TimescaleDB**:
  These support distributed and time-series workloads, respectively. Clustering helps manage large datasets spread across many nodes. TimescaleDB, built on PostgreSQL, benefits from clustering by distributing time-based data for fast querying. Hydra uses clustering to process identity and access management workloads more efficiently in high-load environments.



> **Note**: Elestio is frequently adding support for more clustered software like OpenSearch, Kafka, and ClickHouse. Always check the Elestio catalogue for the latest supported services.

Cluster Configurations:

Elestio offers several clustering modes, each designed for a different balance between simplicity, speed, and reliability:

- **Single Node**:
  This setup has only one node and is easy to manage. It acts as a standalone Primary node. It's good for testing, development, or low-traffic applications. Later, you can scale to more nodes without rebuilding the entire setup. Elestio lets you expand this node into a full cluster with just a few clicks.
- **Primary/Replica**:
  One node (Primary) handles all write operations, and one or more Replicas handle read queries. Replication is usually asynchronous and ensures data is copied to all replicas. This improves read performance and provides redundancy if the primary node fails. Elestio manages automatic data syncing and failover setup.

Cluster Management Features:

Elestio's cluster dashboard includes tools for managing, monitoring, and securing your clusters. These help ensure stability and ease of use:

- **Node Management**:
  You can scale your cluster by adding or removing nodes as your app grows. Adding a node increases capacity; removing one helps reduce costs. Elestio handles provisioning and

configuring nodes automatically, including replication setup. This makes it easier to scale horizontally without downtime.

- **Backups and Restores**:
  Elestio provides scheduled and on-demand backups for all nodes. Backups are stored securely and can be restored if something goes wrong. You can also create a snapshot before major changes to your system. This helps protect against data loss due to failures, bugs, or human error.
- **Access Control**:
  You can limit access to your cluster using IP allowlists, ensuring only trusted sources can connect. Role-based access control (RBAC) can be applied for managing different user permissions. SSH and database passwords are generated securely and can be rotated easily from the dashboard. These access tools help reduce the risk of unauthorized access.
- **Monitoring and Alerts**:
  Real-time metrics like CPU, memory, disk usage, and network traffic are available through the dashboard. You can also check logs for troubleshooting and set alerts for high resource usage or failure events. Elestio uses built-in observability tools to monitor the health of your cluster and notify you if something needs attention. This allows you to catch problems early and take action.

Deploying a New Cluster

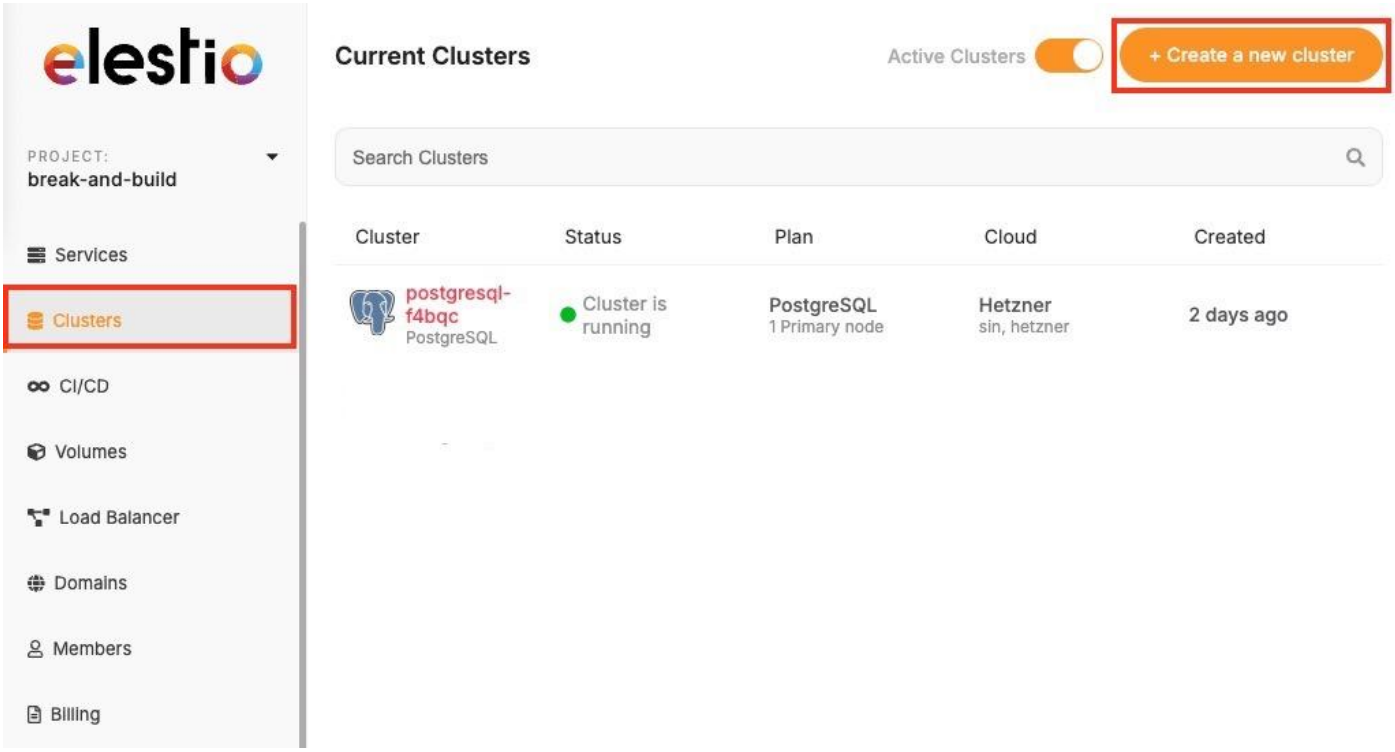
Creating a cluster is a foundational step when deploying services in Elestio. Clusters provide isolated environments where you can run containerized workloads, databases, and applications. Elestio’s web dashboard helps the process, allowing you to configure compute resources, choose cloud providers, and define deployment regions without writing infrastructure code. This guide walks through the steps required to create a new cluster using the Elestio dashboard.

Prerequisites

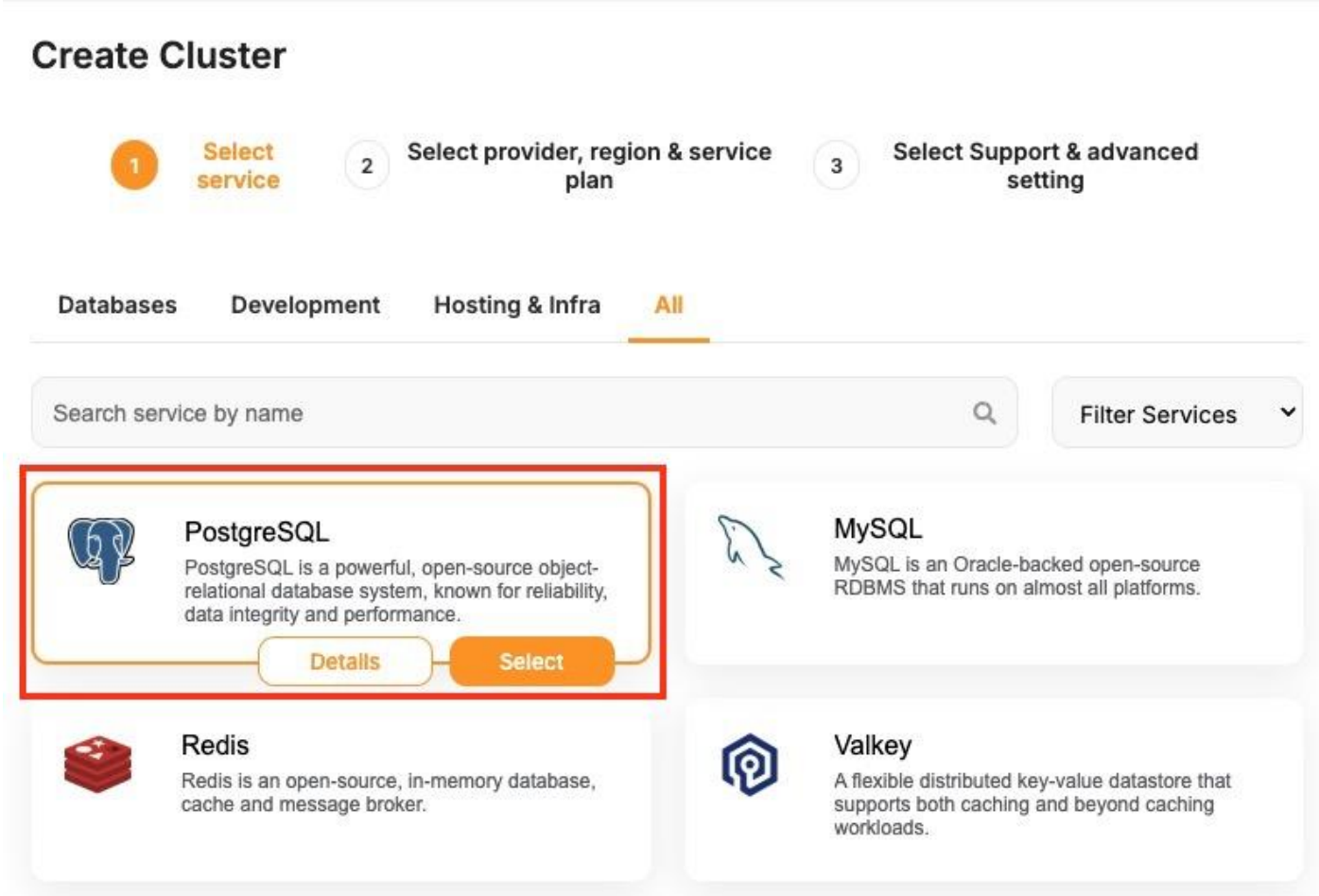
To get started, you’ll need an active Elestio account. If you’re planning to use your own infrastructure, make sure you have valid credentials for your preferred cloud provider (like AWS, GCP, Azure, etc.). Alternatively, you can choose to deploy clusters using Elestio-managed infrastructure, which requires no external configuration.

Creating a Cluster

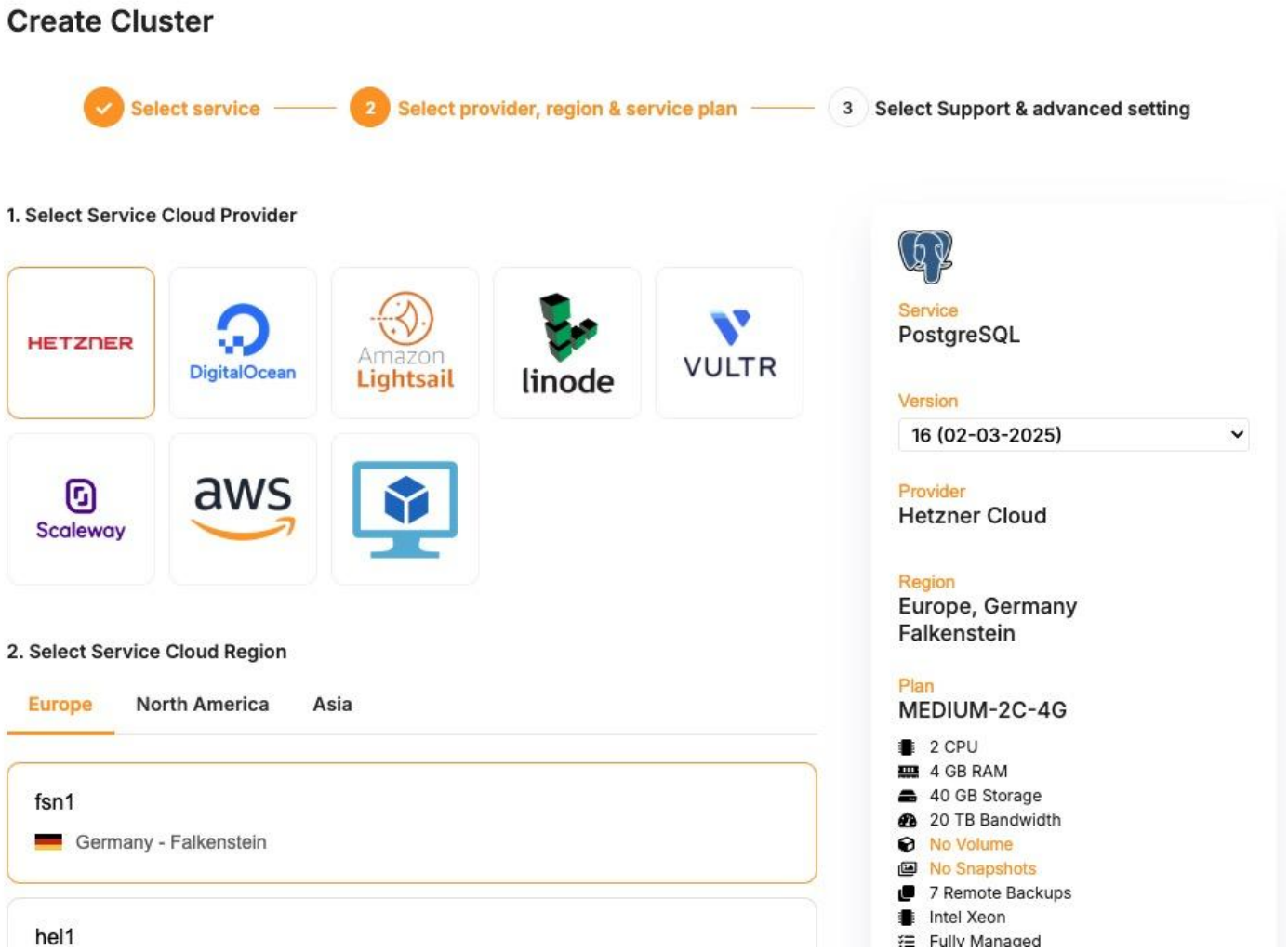
Once you’re logged into the Elestio dashboard, navigate to the **Clusters** section from the sidebar. You’ll see an option to **Create a new cluster**—clicking this will start the configuration process. The cluster creation flow is flexible but simple for defining essential details like provider, region, and resources in one place.



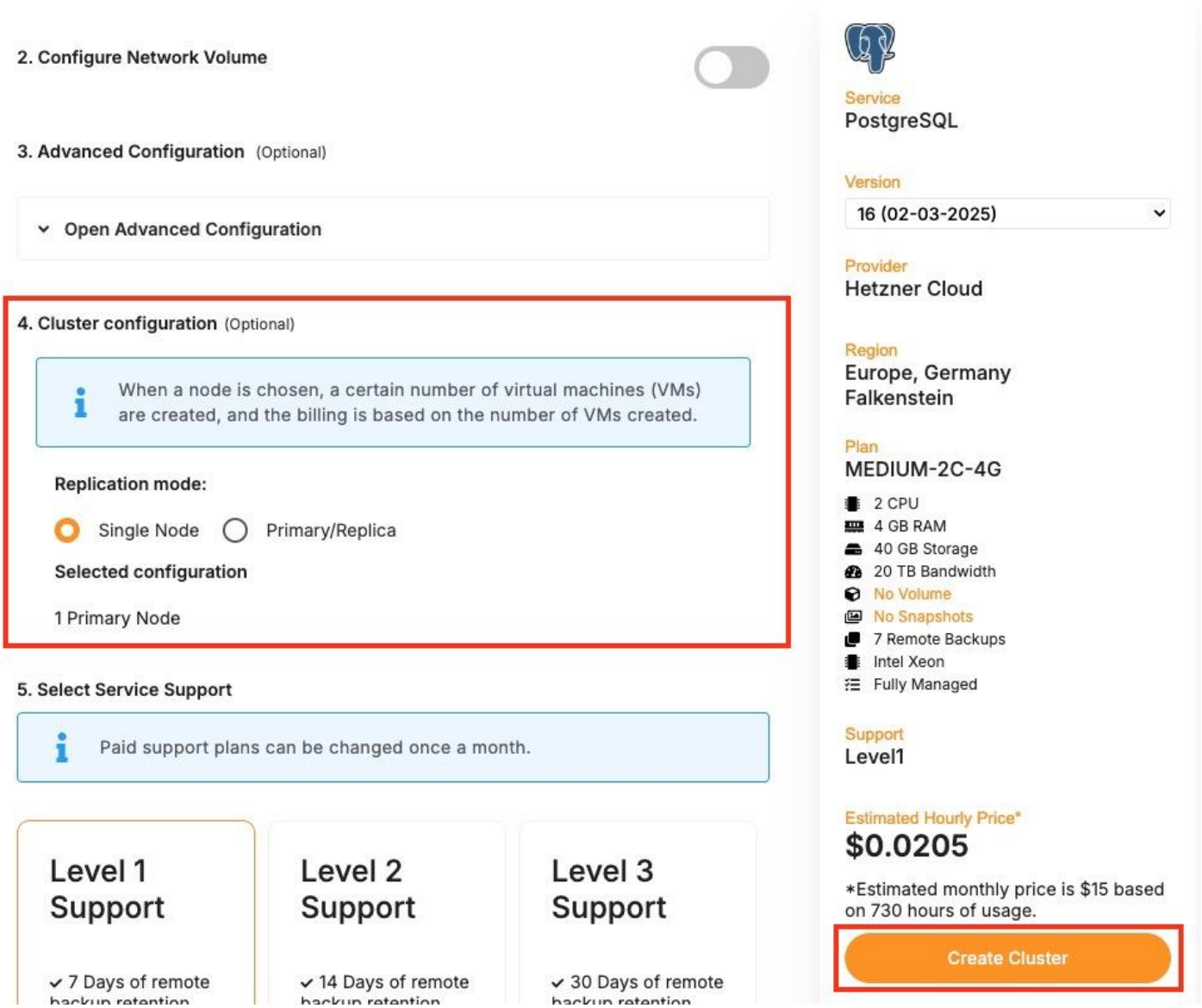
Now, select the database service of your choice that you need to create in a cluster environment. Click on **Select** button as you choose one.



During setup, you'll be asked to choose a hosting provider. Elestio supports both managed and BYOC (Bring Your Own Cloud) deployments, including AWS, DigitalOcean, Hetzner, and custom configurations. You can then select a region based on latency or compliance needs, and specify the number of nodes along with CPU, RAM, and disk sizes per node.



If you're setting up a high-availability cluster, the dashboard also allows you to configure cluster-related details under **Cluster configuration,** where you get to select things like replication modes, number of replicas, etc. After you've configured the cluster, review the summary to ensure all settings are correct. Click the **Create Cluster** button to begin provisioning.



Elestio will start the deployment process, and within a few minutes, the cluster will appear in your dashboard. Once your cluster is live, it can be used to deploy new nodes and additional configurations. Each cluster supports real-time monitoring, log access, and scaling operations through the dashboard. You can also set up automated backups and access control through built-in features available in the cluster settings.

Node Management

Node management plays a critical role in operating reliable and scalable infrastructure on Elestio. Whether you're deploying stateless applications or stateful services like databases, managing the underlying compute units nodes is essential for maintaining stability and performance.

Understanding Nodes

In Elestio, a **node** is a virtual machine that contributes compute, memory, and storage resources to a cluster. Clusters can be composed of a single node or span multiple nodes, depending on workload demands and availability requirements. Each node runs essential services and containers as defined by your deployed applications or databases.

Nodes in Elestio are provider-agnostic, meaning the same concepts apply whether you're using Elestio-managed infrastructure or connecting your own cloud provider (AWS, Azure, GCP, etc.). Each node is isolated at the VM level but participates fully in the cluster's orchestration and networking. This abstraction allows you to manage infrastructure without diving into the complexity of underlying platforms.

Node Operations

The Elestio dashboard allows you to manage the lifecycle of nodes through clearly defined operations. These include:

- **Creating a node**, which adds capacity to your cluster and helps with horizontal scaling of services. This is commonly used when load increases or when preparing a high-availability deployment.
- **Deleting a node**, which removes underutilized or problematic nodes. Safe deletion includes draining workloads to ensure service continuity.
- **Promoting a node**, which changes the role of a node within the cluster—typically used in clusters with redundancy, where certain nodes may need to take on primary or leader responsibilities.

Each of these operations is designed to be safely executed through the dashboard and is validated against the current cluster state to avoid unintended service disruption. These actions are supported by Elestio's backend orchestration, which handles tasks like container rescheduling and load balancing when topology changes.

Monitoring and Maintenance

Monitoring is a key part of effective node management. Elestio provides per-node visibility through the dashboard, allowing you to inspect **CPU**, **memory**, and **disk utilization** in real time. Each node also exposes **logs**, **status indicators**, and **health checks** to help detect anomalies or degradation early.

In addition to passive monitoring, the dashboard supports active maintenance tasks. You can **reboot a node** when applying system-level changes or troubleshooting, or **drain a node** to safely migrate workloads away from it before performing disruptive actions. Draining ensures that running containers are rescheduled on other nodes in the cluster, minimizing service impact.

For production setups, combining resource monitoring with automation like scheduled reboots, log collection, and alerting can help catch issues before they affect users. While Elestio handles many aspects of orchestration automatically, having visibility at the node level helps teams make informed decisions about scaling, updates, and incident response.

Cluster-wide resource graphs and node-level metrics are also useful for capacity planning. Identifying trends such as memory saturation or disk pressure allows you to preemptively scale or rebalance workloads, reducing the risk of downtime.

Cluster Management

Adding a Node

As your application usage grows or your infrastructure requirements change, scaling your cluster becomes essential. In Elestio, you can scale horizontally by adding new nodes to an existing cluster. This operation allows you to expand your compute capacity, improve availability, and distribute workloads more effectively.

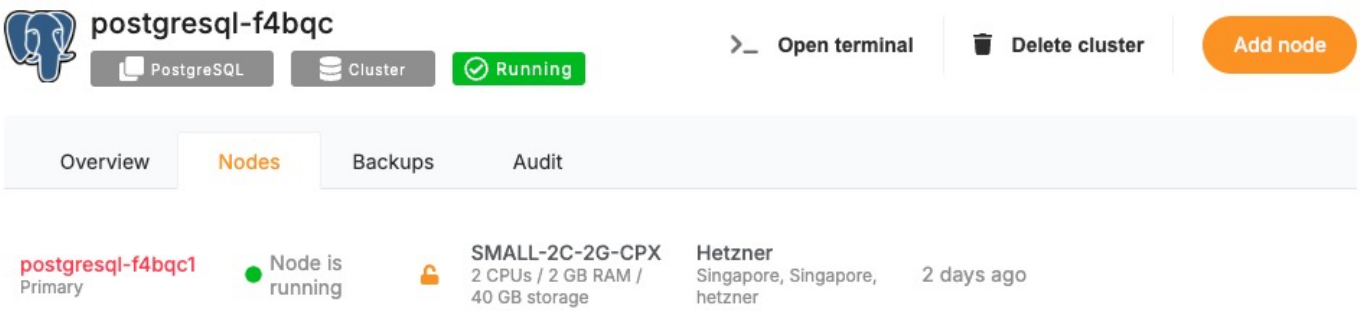
Need to Add a Node

There are several scenarios where adding a node becomes necessary. One of the most common cases is **resource saturation** when existing nodes are fully utilized in terms of CPU, memory, or disk. Adding another node helps distribute the workload and maintain performance under load.

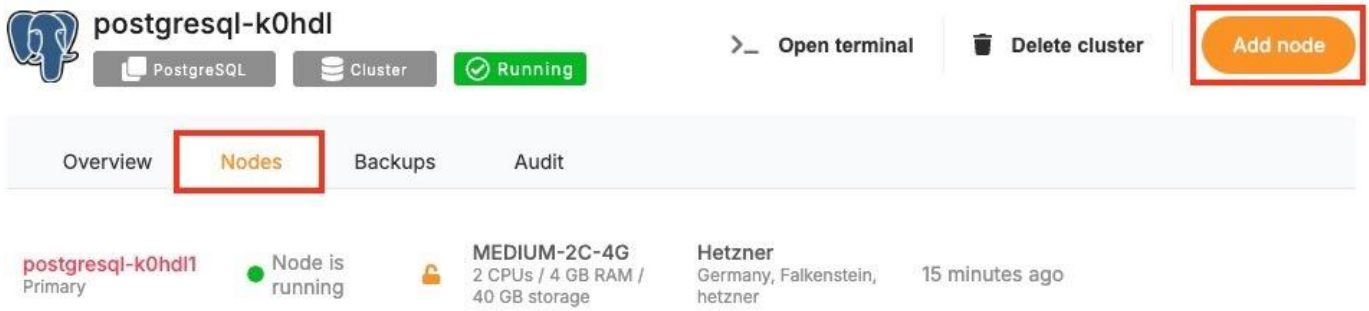
In clusters that run **stateful services** or require **high availability**, having additional nodes ensures that workloads can fail over without downtime. Even in development environments, nodes can be added to isolate environments or test services under production-like load conditions. Scaling out also gives you flexibility when deploying services with different resource profiles or placement requirements.

Add a Node to Cluster

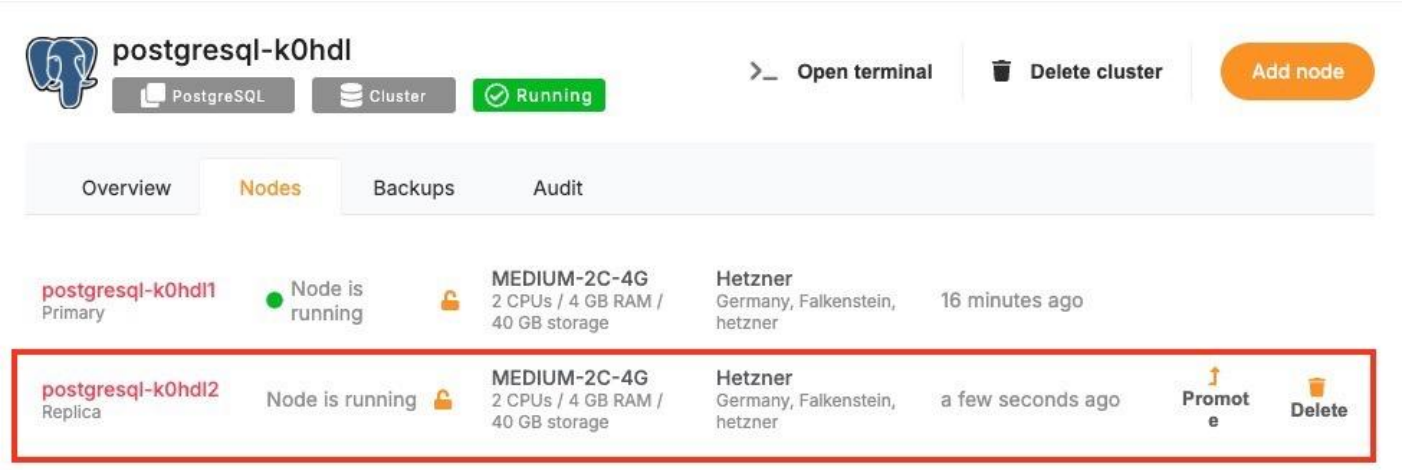
To begin, log in to the Elestio dashboard and navigate to the **Clusters** section from the sidebar. Select the cluster you want to scale. Once inside the cluster view, switch to the **Nodes** tab. This section provides an overview of all current nodes along with their health status and real-time resource usage.



To add a new node, click the **“Add Node”** button. This opens a configuration panel where you can define the specifications for the new node. You’ll be asked to specify the amount of **CPU**, **memory**, and **disk** you want to allocate. If you’re using a bring-your-own-cloud setup, you may also need to confirm or choose the cloud provider and deployment region.



After configuring the node, review the settings to ensure they meet your performance and cost requirements. Click **“Create”** to initiate provisioning. Elestio will begin setting up the new node, and once it’s ready, it will automatically join your cluster.



Once provisioned, the new node will appear in the node list with its own metrics and status indicators. You can monitor its activity, verify that workloads are being scheduled to it, and access its logs directly from the dashboard. From this point onward, the node behaves like any other in the cluster and can be managed using the same lifecycle actions such as rebooting or draining.

Post-Provisioning Considerations

After the node has been added, it becomes part of the active cluster and is available for scheduling workloads. Elestio’s orchestration layer will begin using it automatically, but you can further customize service placement through resource constraints or affinity rules if needed.

For performance monitoring, the dashboard provides per-node metrics, including CPU load, memory usage, and disk I/O. This visibility helps you confirm that the new node is functioning

correctly and contributing to workload distribution as expected.

Maintenance actions such as draining or rebooting the node are also available from the same interface, making it easy to manage the node lifecycle after provisioning.

Promoting a Node

Clusters can be designed for high availability or role-based workloads, where certain nodes may take on leadership or coordination responsibilities. In these scenarios, promoting a node is a key administrative task. It allows you to change the role of a node. While not always needed in basic setups, node promotion becomes essential in distributed systems, replicated databases, or services requiring failover control.

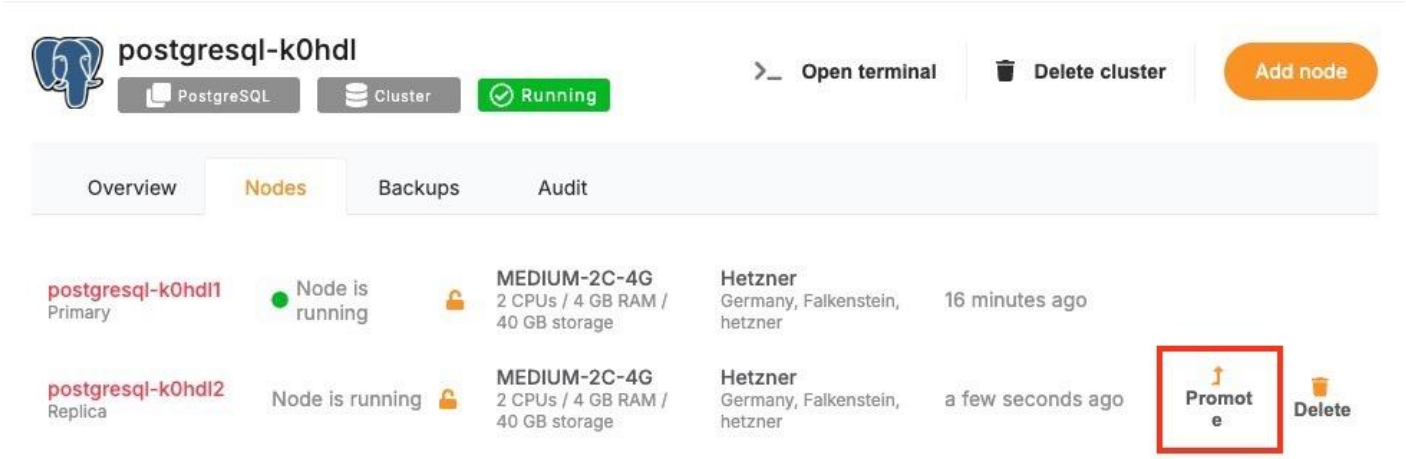
When to Promote a Node?

Promoting a node is typically performed in clusters where role-based architecture is used. In high-availability setups, some nodes may act as leaders while others serve as followers or replicas. If a leader node becomes unavailable or needs to be replaced, you can promote another node to take over its responsibilities and maintain continuity of service.

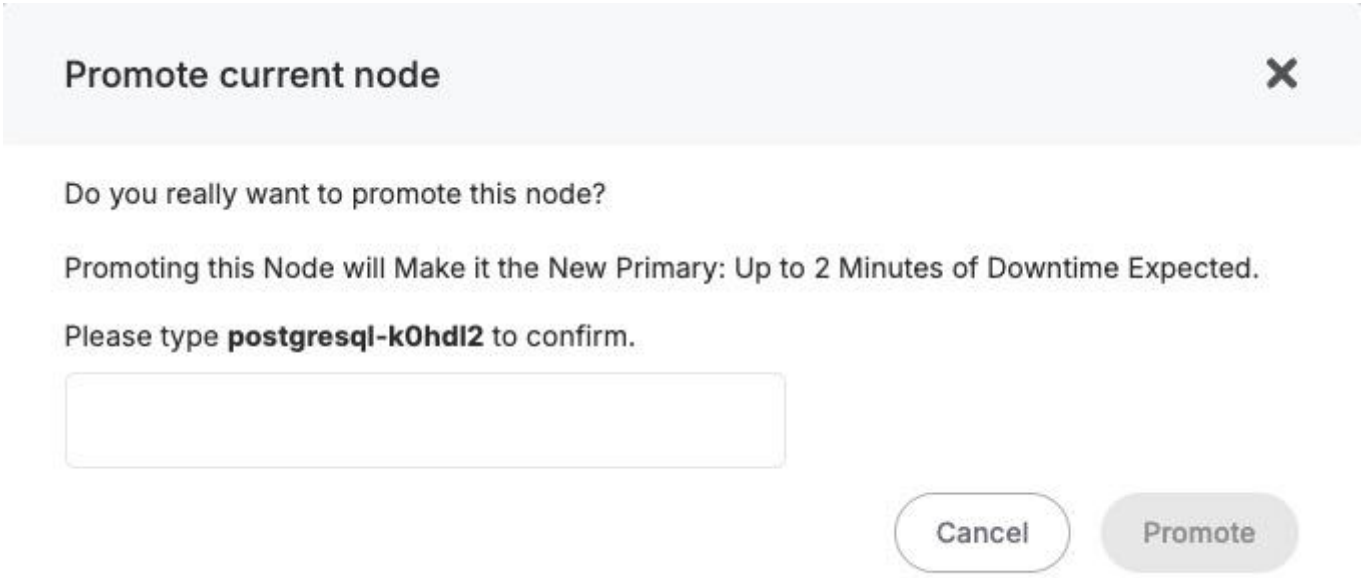
Node promotion is also useful when scaling out and rebalancing responsibilities across a larger cluster. For example, promoting a node to handle scheduling, state tracking, or replication leadership can reduce bottlenecks and improve responsiveness. In cases involving database clusters or consensus-driven systems, promotion ensures a clear and controlled transition of leadership without relying solely on automatic failover mechanisms.

Promote a Node in Elestio

To promote a node, start by accessing the **Clusters** section in the Elestio dashboard. Choose the cluster containing the node you want to promote. Inside the cluster view, navigate to the **Nodes** tab to see the full list of nodes, including their current roles, health status, and resource usage. Locate the node that you want to promote and open its action menu. From here, select the **“Promote Node”** option.



You may be prompted to confirm the action, depending on the configuration and current role of the node. This confirmation helps prevent unintended role changes that could affect cluster behavior.



Once confirmed, Elestio will initiate the promotion process. This involves reconfiguring the cluster's internal coordination state to acknowledge the new role of the promoted node. Depending on the service architecture and the software running on the cluster, this may involve reassigning leadership, updating replication targets, or shifting service orchestration responsibilities.

After promotion is complete, the node's updated role will be reflected in the dashboard. At this point, it will begin operating with the responsibilities assigned to its new status. You can monitor its activity, inspect logs, and validate that workloads are being handled as expected.

Considerations for Promotion

Before promoting a node, ensure that it meets the necessary resource requirements and is in a stable, healthy state. Promoting a node that is under high load or experiencing performance issues can lead to service degradation. It's also important to consider replication and data

synchronization, especially in clusters where stateful components like databases are in use.

Promotion is a safe and reversible operation, but it should be done with awareness of your workload architecture. If your system relies on specific leader election mechanisms, promoting a node should follow the design patterns supported by those systems.

Removing a Node

Over time, infrastructure needs change. You may scale down a cluster after peak load, decommission outdated resources, or remove a node that is no longer needed for cost, isolation, or maintenance reasons. Removing a node from a cluster is a safe and structured process designed to avoid disruption. The dashboard provides an accessible interface for performing this task while preserving workload stability.

Why Remove a Node?

Node removal is typically part of resource optimization or cluster reconfiguration. You might remove a node when reducing costs in a staging environment, when redistributing workloads across fewer or more efficient machines, or when phasing out a node for maintenance or retirement.

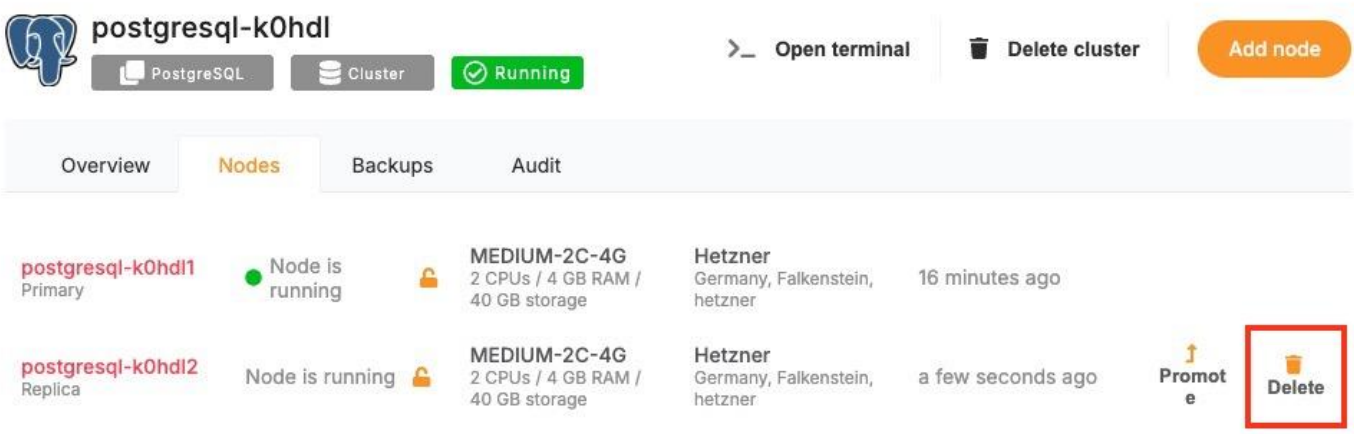
Another common scenario is infrastructure rebalancing, where workloads are shifted to newer nodes with better specs or different regions. Removing an idle or underutilized node can simplify management and reduce noise in your monitoring stack. It also improves scheduling efficiency by removing unneeded targets from the orchestration engine.

In high-availability clusters, node removal may be preceded by data migration or role reassignment (such as promoting a replica). Proper planning helps maintain system health while reducing reliance on unnecessary compute resources.

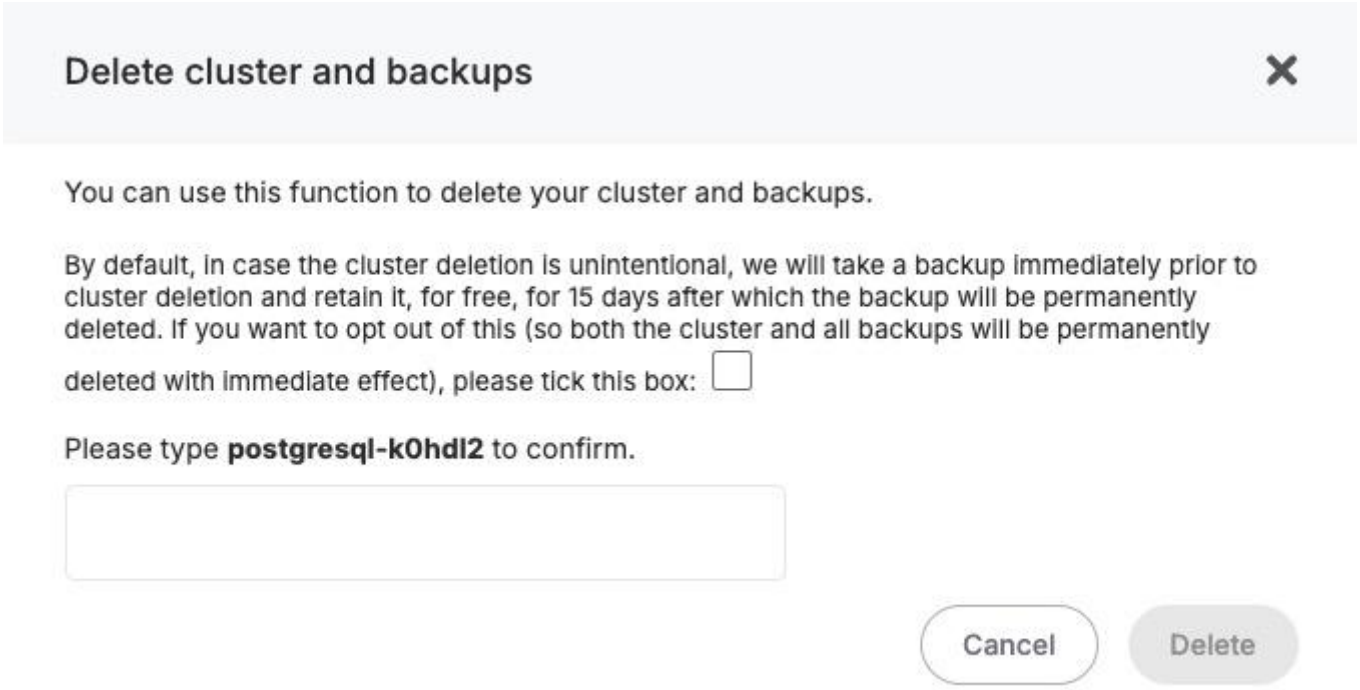
Remove a Node

To begin the removal process, open the Elestio dashboard and navigate to the **Clusters** section. Select the cluster that contains the node you want to remove. From within the cluster view, open the **Nodes** tab to access the list of active nodes and their statuses.

Find the node you want to delete from the list. If the node is currently running services, ensure that those workloads can be safely rescheduled to other nodes or are no longer needed. Since Elestio does not have a built-in drain option, any workload redistribution needs to be handled manually, either by adjusting deployments or verifying that redundant nodes are available. Once the node is drained and idle, open the action menu for that node and select **“Delete Node”**.



The dashboard may prompt you to confirm the operation. After confirmation, Elestio will begin the decommissioning process. This includes detaching the node from the cluster, cleaning up any residual state, and terminating the associated virtual machine.



Once the operation completes, the node will no longer appear in the cluster's node list, and its resources will be released.

Considerations for Safe Node Removal

Before removing a node in Elestio, it’s important to review the services and workloads currently running on that node. Since Elestio does not automatically redistribute or migrate workloads during node removal, you should ensure that critical services are either no longer in use or can be manually rescheduled to other nodes in the cluster. This is particularly important in multi-node environments running stateful applications, databases, or services with specific affinity rules.

You should also verify that your cluster will have sufficient capacity after the node is removed. If the deleted node was handling a significant portion of traffic or compute load, removing it without replacement may lead to performance degradation or service interruption. In high-availability clusters, ensure that quorum-based components or replicas are not depending on the node targeted for deletion. Additionally, confirm that the node is not playing a special role such as holding primary data or acting as a manually promoted leader before removal. If necessary, reconfigure or promote another node prior to deletion to maintain cluster integrity.

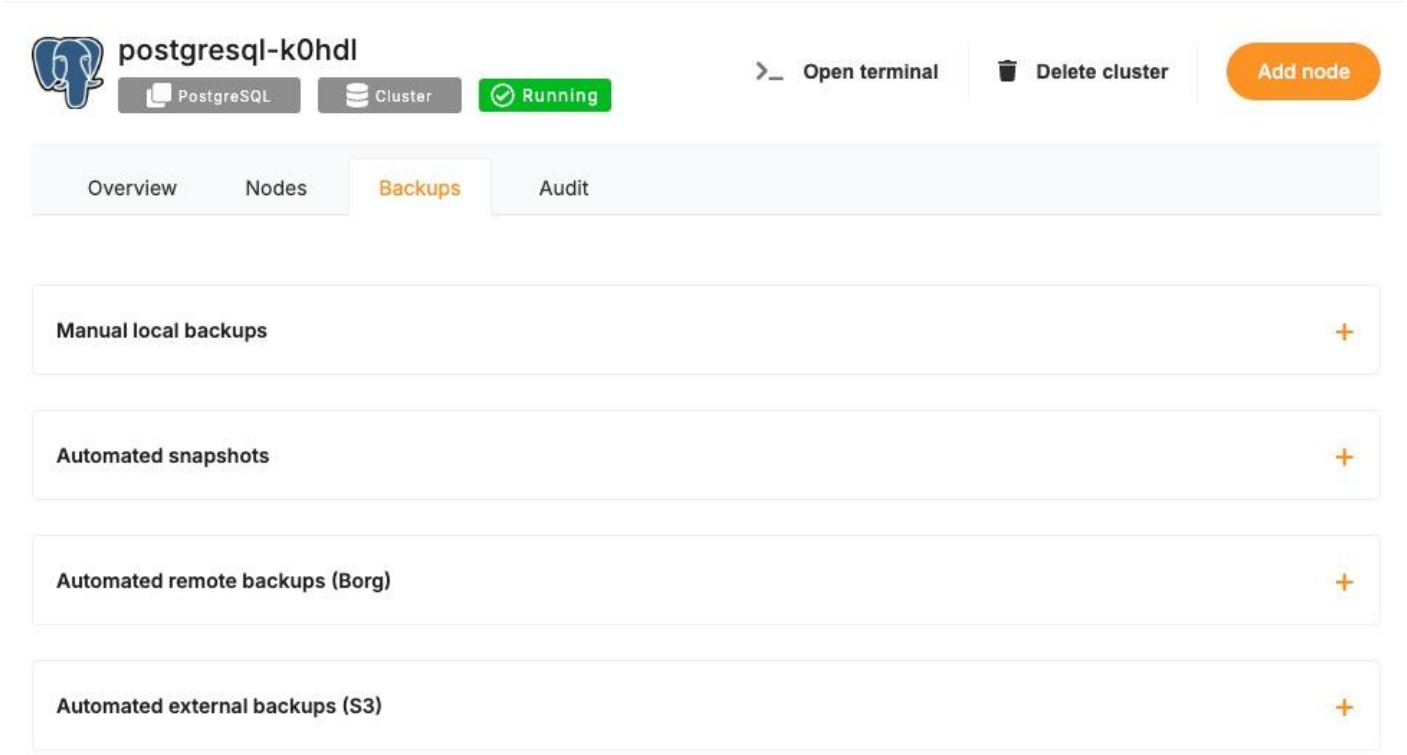
Backups and Restores

Reliable backups are essential for data resilience, recovery, and business continuity. Elestio provides built-in support for managing backups across all supported services, ensuring that your data is protected against accidental loss, corruption, or infrastructure failure. The platform includes an automated backup system with configurable retention policies and a straightforward restore process, all accessible from the dashboard. Whether you're operating a production database or a test environment, understanding how backups and restores work in Elestio is critical for maintaining service reliability.

Cluster Backups

Elestio provides multiple backup mechanisms designed to support various recovery and compliance needs. Backups are created automatically for most supported services, with consistent intervals and secure storage in managed infrastructure. These backups are performed in the background to ensure minimal performance impact and no downtime during the snapshot process. Each backup is timestamped, versioned, and stored securely with encryption. You can access your full backup history for any given service through the dashboard and select any version for restoration.

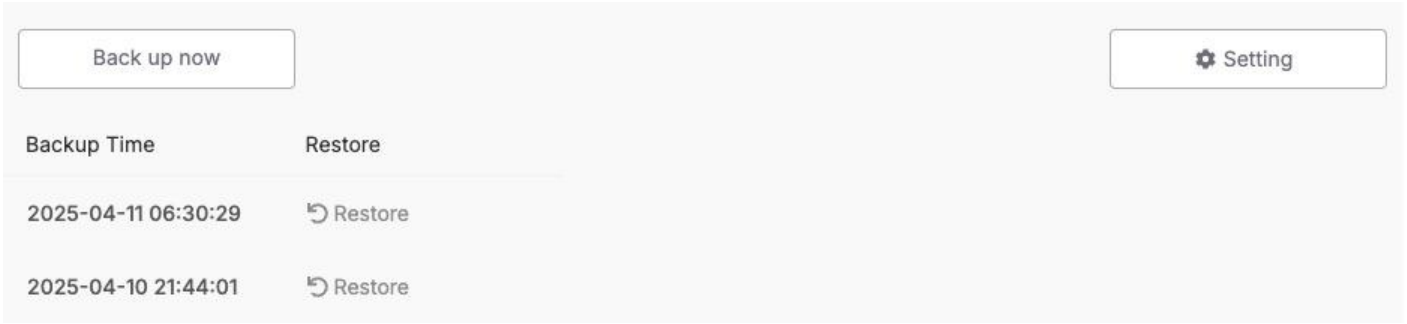
You can utilize different backup options depending on your preferences and operational requirements. Elestio supports **manual local backups** for on-demand recovery points, **automated snapshots** that capture the state of the service at fixed intervals, and **automated remote backups using Borg**, which securely stores backups on external storage volumes managed by Elestio. In addition, you can configure **automated external backups to S3-compatible storage**, allowing you to maintain full control over long-term retention and geographic storage preferences.



Restoring from a Backup

Restoring a backup in Elestio is a user-initiated operation, available directly from the service dashboard. Once you're in the dashboard, select the service you'd like to restore. Navigate to the **Backups** section, where you'll find a list of all available backups along with their creation timestamps.

To initiate a restore, choose the desired backup version and click on the **"Restore"** option. You will be prompted to confirm the operation. Depending on the type of service, the restore can either overwrite the current state or recreate the service as a new instance from the selected backup.



The restore process takes a few minutes, depending on the size of the backup and the service type. Once completed, the restored service is immediately accessible. In the case of databases, you can validate the restore by connecting to the database and inspecting the restored data.

Considerations for Backup & Restore

- Before restoring a backup, it’s important to understand the impact on your current data. Restores may **overwrite existing service state**, so if you need to preserve the current environment, consider creating a manual backup before initiating the restore. In critical environments, restoring to a new instance and validating the data before replacing the original is a safer approach.
- Keep in mind that restore operations are not instantaneous and may temporarily affect service availability. It’s best to plan restores during maintenance windows or periods of low traffic, especially in production environments.
- For services with high-frequency data changes, be aware of the backup schedule and retention policy. Elestio’s default intervals may not capture every change, so for high-volume databases, consider exporting incremental backups manually or using continuous replication where supported.

Monitoring Backup Health

Elestio provides visibility into your backup history directly through the dashboard. You can monitor the **status**, **timestamps**, and **success/failure** of backup jobs. In case of errors or failed backups, the dashboard will display alerts, allowing you to take corrective actions or contact support if necessary.

It’s good practice to periodically verify that backups are being generated and that restore points are recent and complete. This ensures you’re prepared for unexpected failures and that recovery options remain reliable.

Restricting Access by IP

Securing access to services is a fundamental part of managing cloud infrastructure. One of the most effective ways to reduce unauthorized access is by restricting connectivity to a defined set of IP addresses. Elestio supports IP-based access control through its dashboard, allowing you to explicitly define which IPs or IP ranges are allowed to interact with your services. This is particularly useful when exposing databases, APIs, or web services over public endpoints.

Need to Restrict Access by IP

Restricting access by IP provides a first layer of network-level protection. Instead of relying solely on application-layer authentication, you can control who is allowed to even initiate a connection to your service. This approach reduces the surface area for attacks such as brute-force login attempts, automated scanning, or unauthorized probing.

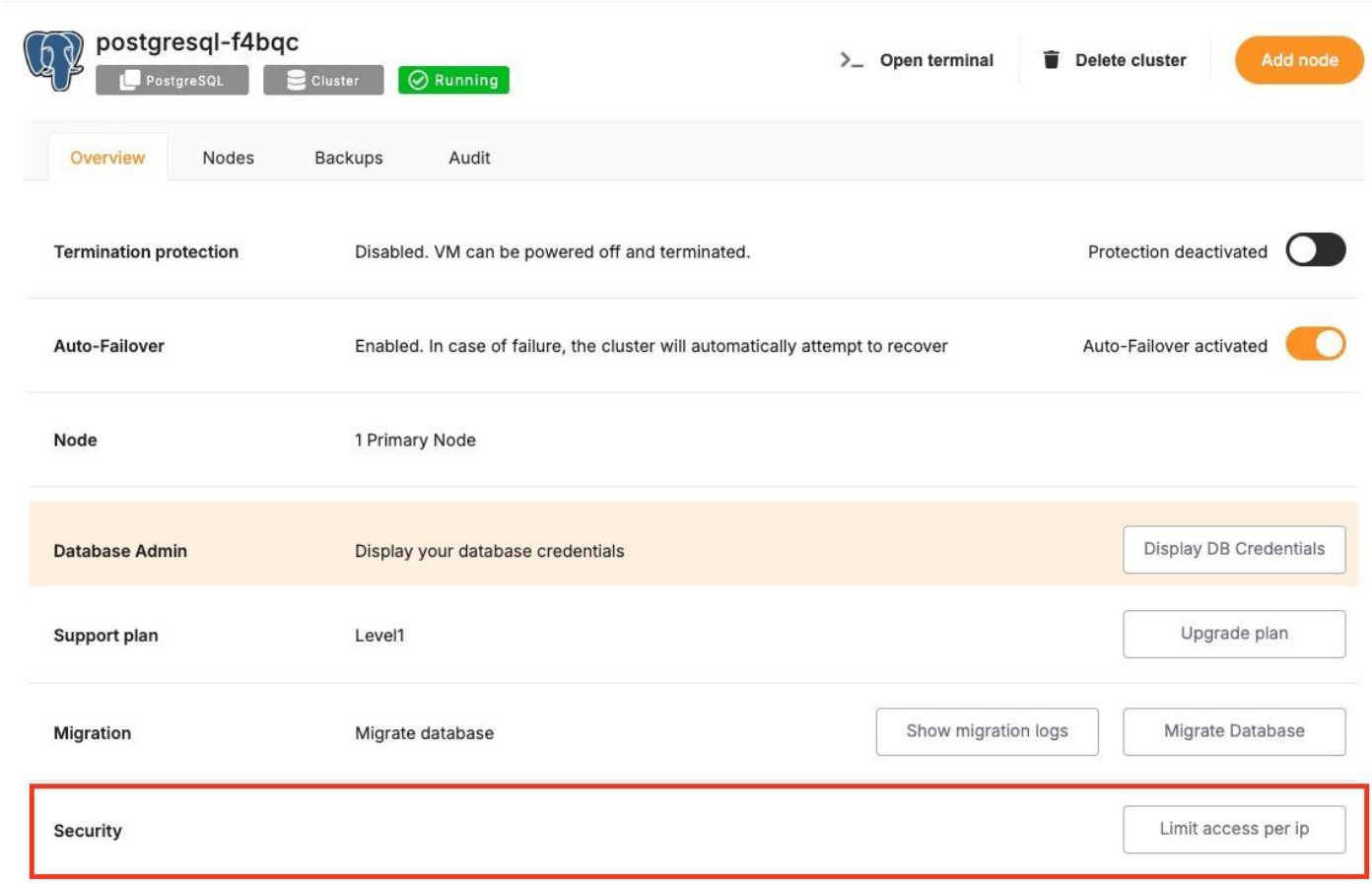
Common use cases include:

- Limiting access to production databases from known office networks or VPNs.
- Allowing only CI/CD pipelines or monitoring tools with static IPs to connect.
- Restricting admin dashboards or internal tools to internal teams.

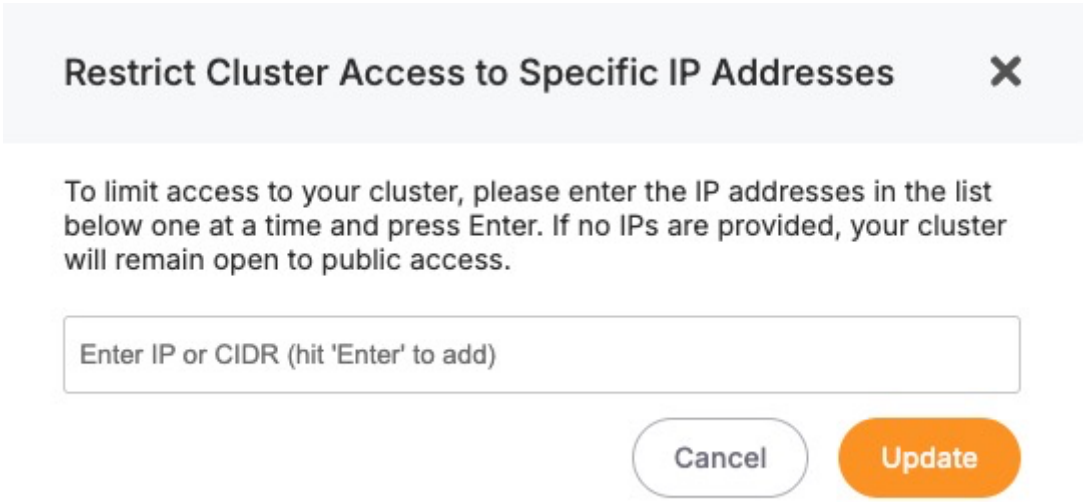
By defining access rules at the infrastructure level, you gain more control over who can reach your services, regardless of their authentication or API access status.

Restrict Access by IP

To restrict access by IP in Elestio, start by logging into the Elestio dashboard and navigating to the **Clusters** section. Select the cluster that hosts the service you want to protect. Once inside the **Cluster Overview** page, locate the **Security** section.



Within this section, you'll find a setting labeled **"Limit access per IP"**. This is where you can define which IP addresses or CIDR ranges are permitted to access the services running in the cluster. You can add a specific IPv4 or IPv6 address (e.g., 203.0.113.5) or a subnet in CIDR notation (e.g., 203.0.113.0/24) to allow access from a range of IPs.



After entering the necessary IP addresses, save the configuration. The changes will apply to all services running inside the cluster, and only the defined IPs will be allowed to establish network connections. All other incoming requests from unlisted IPs will be blocked at the infrastructure level.

Considerations When Using IP Restrictions

- When applying IP restrictions, it’s important to avoid locking yourself out. Always double-check that your own IP address is included in the allowlist before applying rules, especially when working on remote infrastructure.
- For users on dynamic IPs (e.g., home broadband connections), consider using a VPN or a static jump host that you can reliably allowlist. Similarly, if your services are accessed through cloud-based tools, make sure to verify their IP ranges and update your rules accordingly when those IPs change.
- In multi-team environments, document and review IP access policies regularly to avoid stale rules or overly permissive configurations. Combine IP restrictions with secure authentication and encrypted connections (such as HTTPS or SSL for databases) for layered security.

Cluster Resynchronization

In distributed systems, consistency and synchronization between nodes are critical to ensure that services behave reliably and that data remains accurate across the cluster. Elestio provides built-in mechanisms to detect and resolve inconsistencies across nodes using a feature called **Cluster Resynchronization**. This functionality ensures that node-level configurations, data replication, and service states are properly aligned, especially after issues like node recovery, temporary network splits, or service restarts.

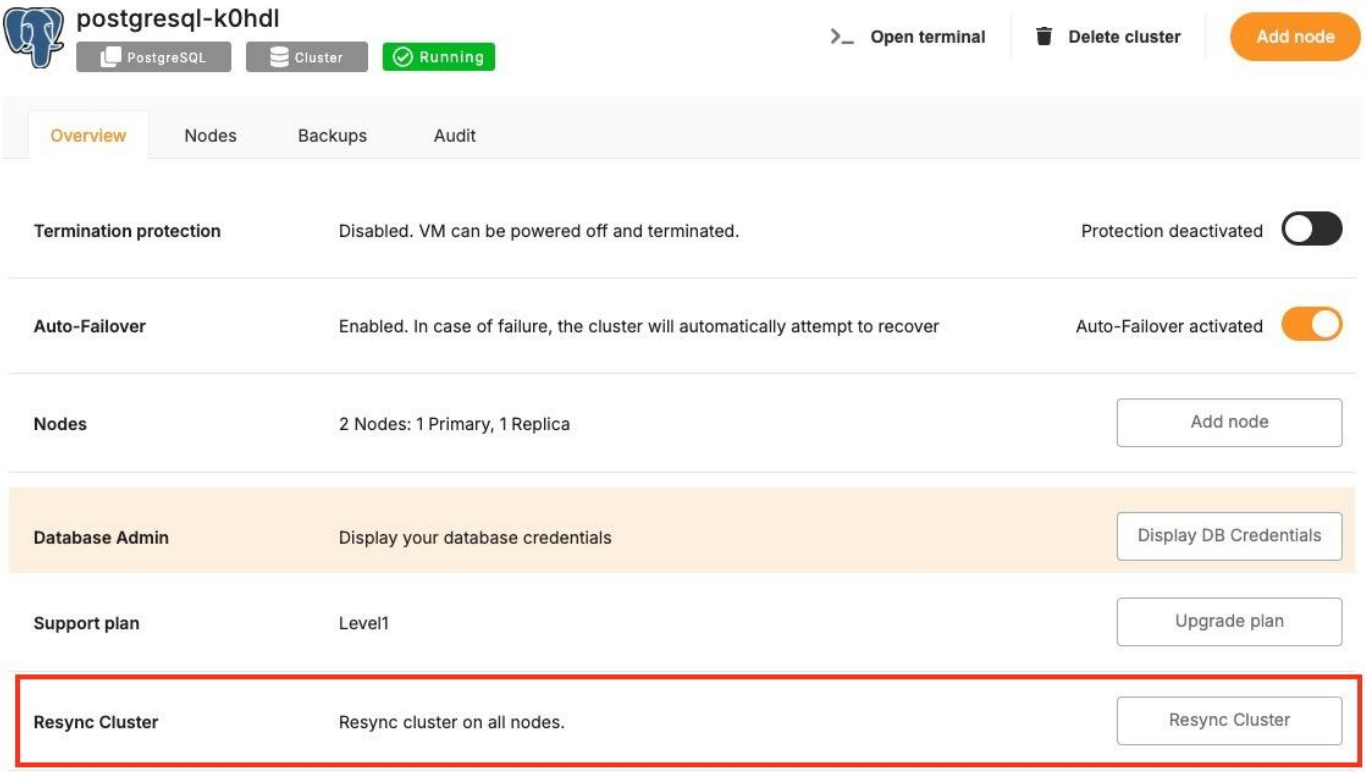
Need for Cluster Resynchronization

Resynchronization is typically required when secondary nodes in a cluster are no longer consistent with the primary node. This can happen due to temporary network failures, node restarts, replication lag, or partial service interruptions. In such cases, secondary nodes may fall behind or store incomplete datasets, which could lead to incorrect behavior if a failover occurs or if read operations are directed to those nodes. Unresolved inconsistencies can result in data divergence, serving outdated content, or failing health checks in load-balanced environments. Performing a resynchronization ensures that all secondary nodes are forcibly aligned with the current state of the primary node, restoring a clean and unified cluster state.

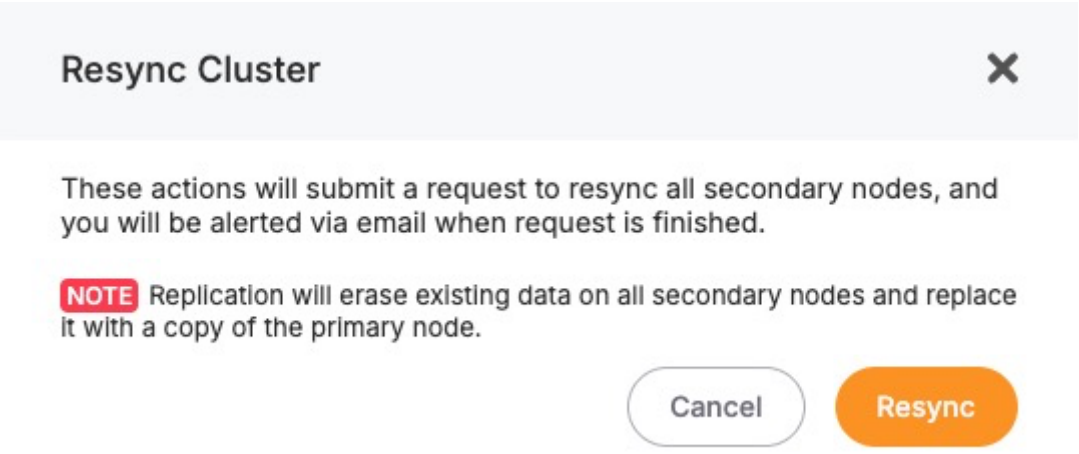
It may also be necessary to perform a resync after restoring a service from backup, during infrastructure migrations, or after recovering a previously offline node. In each of these cases, resynchronization acts as a corrective mechanism to ensure that every node is operating with the same configuration and dataset, reducing the risk of drift and maintaining data integrity across the cluster.

Cluster Resynchronization

To perform a resynchronization, start by accessing the Elestio dashboard and navigating to the **Clusters** section. Select the cluster where synchronization is needed. On the **Cluster Overview** page, scroll down slightly until you find the **“Resync Cluster”** option. This option is visible as part of the cluster controls and is available only in clusters with multiple nodes and a defined primary node.



Clicking the **Resync** button opens a confirmation dialog. The message clearly explains that this action will initiate a request to resynchronize **all secondary nodes**. During the resync process, **existing data on all secondary nodes will be erased and replaced with a copy of the data from the primary node**. This operation ensures full consistency across the cluster but should be executed with caution, especially if recent changes exist on any of the secondaries that haven't yet been replicated.



You will receive an email notification once the resynchronization is complete. During this process, Elestio manages the replication safely, but depending on the size of the data, the operation may take a few minutes. It's advised to avoid making further changes to the cluster while the resync is in progress.

Considerations Before Resynchronizing

- Before triggering a resync, it’s important to verify that the primary node holds the desired state and that the secondary nodes do not contain any critical unsynced data. Since the resync **overwrites** the secondary nodes completely, any local changes on those nodes will be lost.
- This action is best used when you’re confident that the primary node is healthy, current, and stable. Avoid initiating a resync if the primary has recently experienced errors or data issues. Additionally, consider performing this operation during a low-traffic period, as synchronization may temporarily impact performance depending on the data volume.
- If your application requires high consistency guarantees, it’s recommended to monitor your cluster closely during and after the resync to confirm that services are functioning correctly and that the replication process completed successfully.

Database Migrations

When managing production-grade services, the ability to perform reliable and repeatable database migrations is critical. Whether you're applying schema changes, updating seed data, or managing version-controlled transitions, Elestio provides a built-in mechanism to execute migrations safely from the dashboard. This functionality is especially relevant when running containerized database services like PostgreSQL, MySQL, or similar within a managed cluster.

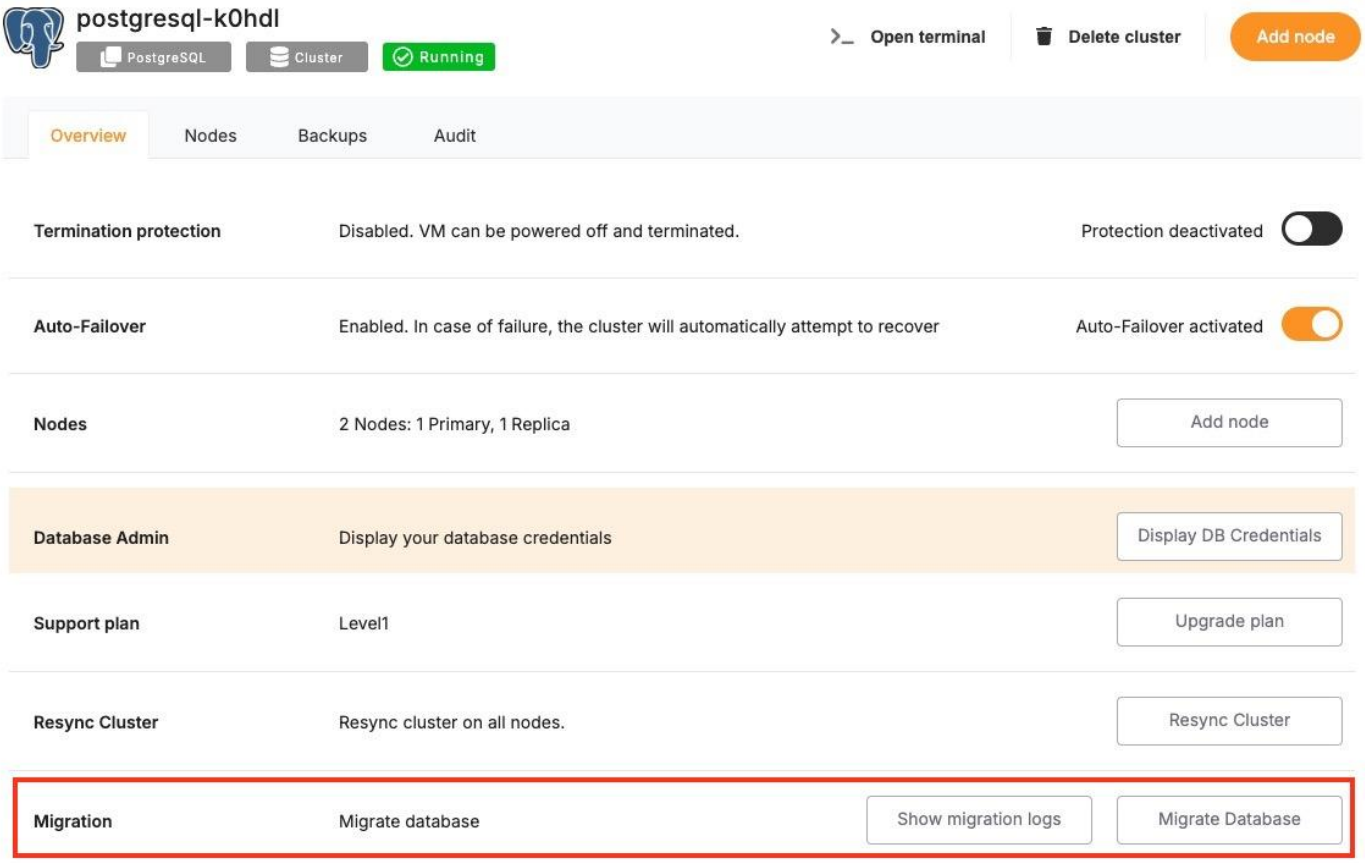
Need for Migrations

Database migrations are commonly required when updating your application's data model or deploying new features. Schema updates such as adding columns, modifying data types, creating indexes, or introducing new tables need to be synchronized with the deployment lifecycle of your application code.

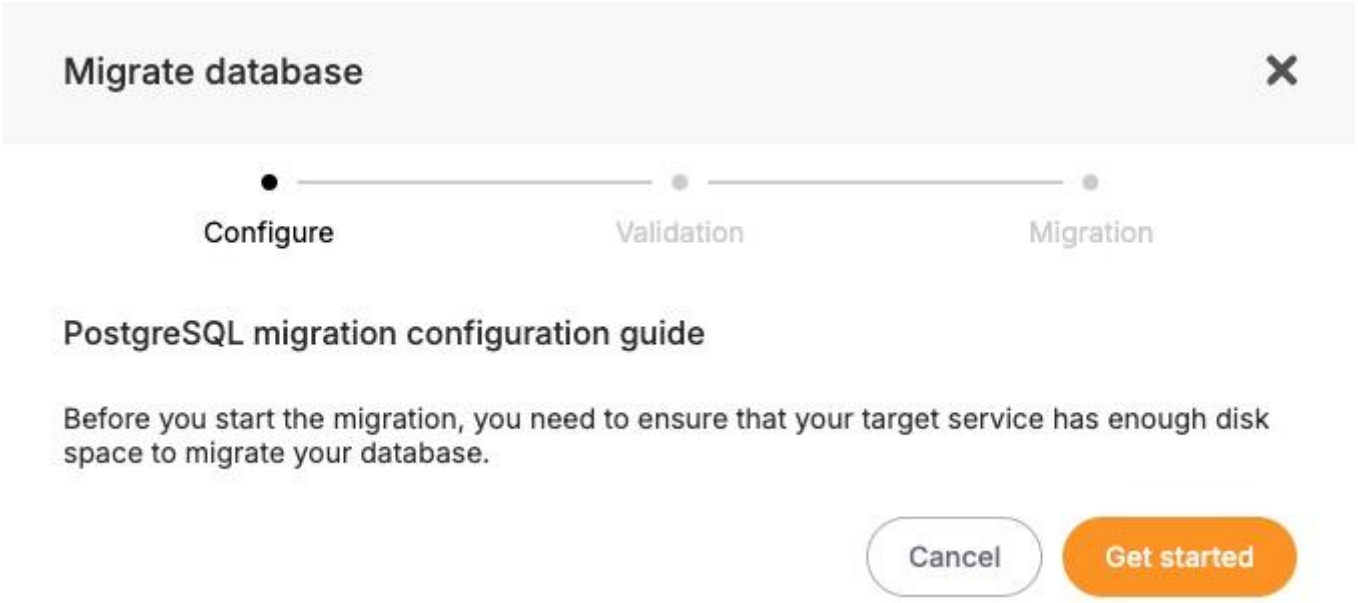
Migrations may also be needed during version upgrades to introduce structural or configuration changes required by newer database engine versions. In some cases, teams use migrations to apply baseline datasets, adjust permissions, or clean up legacy objects. Running these changes through a controlled migration system ensures consistency across environments and helps avoid untracked manual changes.

Running Database Migration

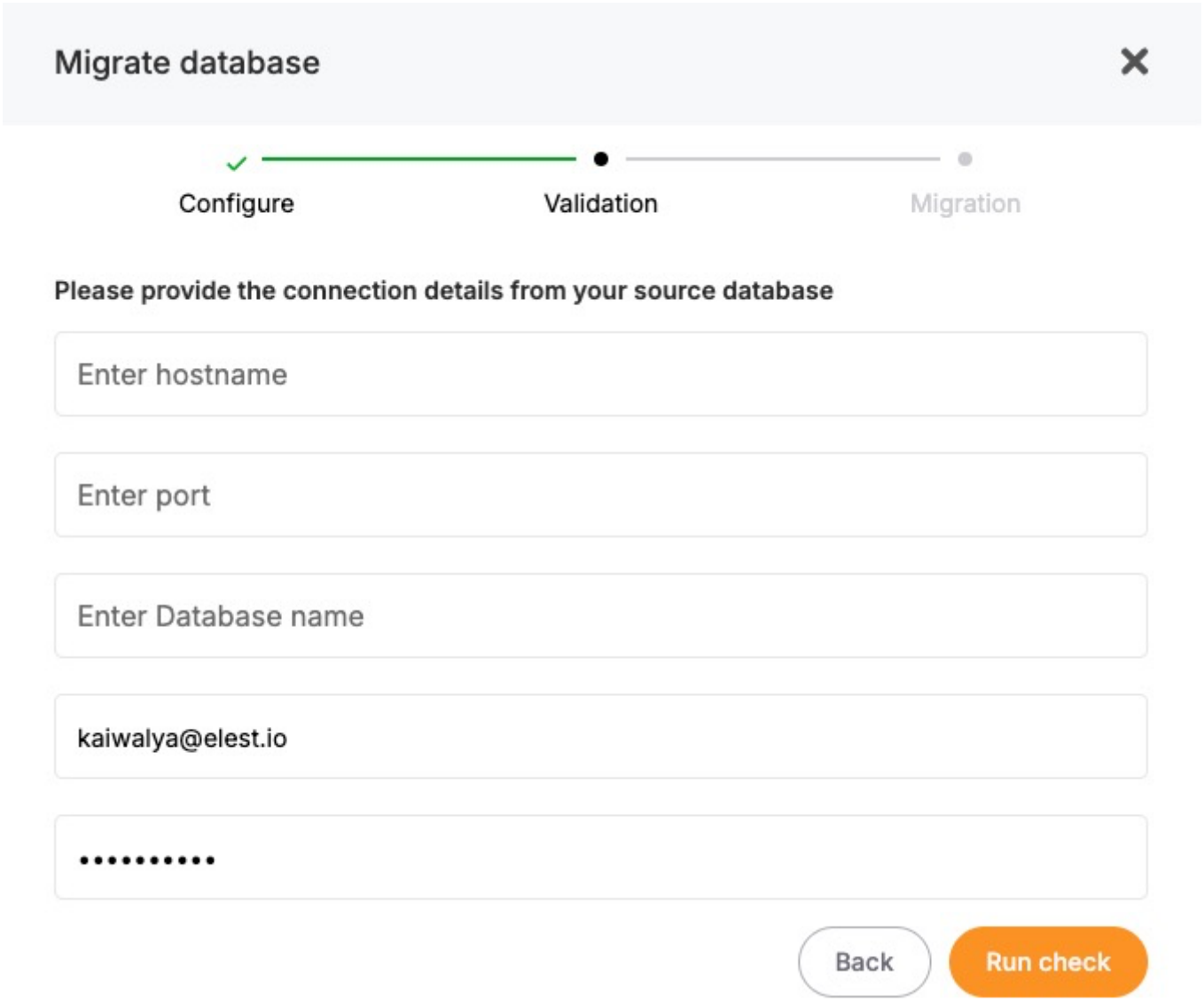
To run a database migration in Elestio, start by logging into the Elestio dashboard and navigating to the **Clusters** section. Select the cluster that contains the target database service. From the **Cluster Overview** page, scroll down until you find the **"Migration"** option.



Clicking this option will open the migration workflow, which follows a **three-step process**: **Configure**, **Validation**, and **Migration**. In the **Configure** step, Elestio provides a migration configuration guide specific to the database type, such as PostgreSQL. At this point, you must ensure that your target service has sufficient **disk space** to complete the migration. If there is not enough storage available, the migration may fail midway, so it’s strongly recommended to review storage utilization beforehand.



Once configuration prerequisites are met, you can proceed to the **Validation** step. Elestio will check the secondary database details you have provided for the migration.



If the validation passes, the final **Migration** step will become active. You can then initiate the migration process. Elestio will handle the actual data transfer, schema replication, and state synchronization internally. The progress is tracked, and once completed, the migrated database will be fully operational on the target service.

Considerations Before Running Migrations

- Before running any migration, it’s important to validate the script or changes in a staging environment. Since migrations may involve irreversible changes—such as dropping columns, altering constraints, or modifying data—careful review and version control are essential.
- In production environments, plan migrations during maintenance windows or low-traffic periods to minimize the impact of any schema locks or temporary unavailability. If you’re using replication or high-availability setups, confirm that the migration is compatible with

your architecture and will not disrupt synchronization between primary and secondary nodes.
- You should also ensure that proper backups are in place before applying structural changes. In Elestio, the backup feature can be used to create a restore point that allows rollback in case the migration introduces issues.

Deleting a Cluster

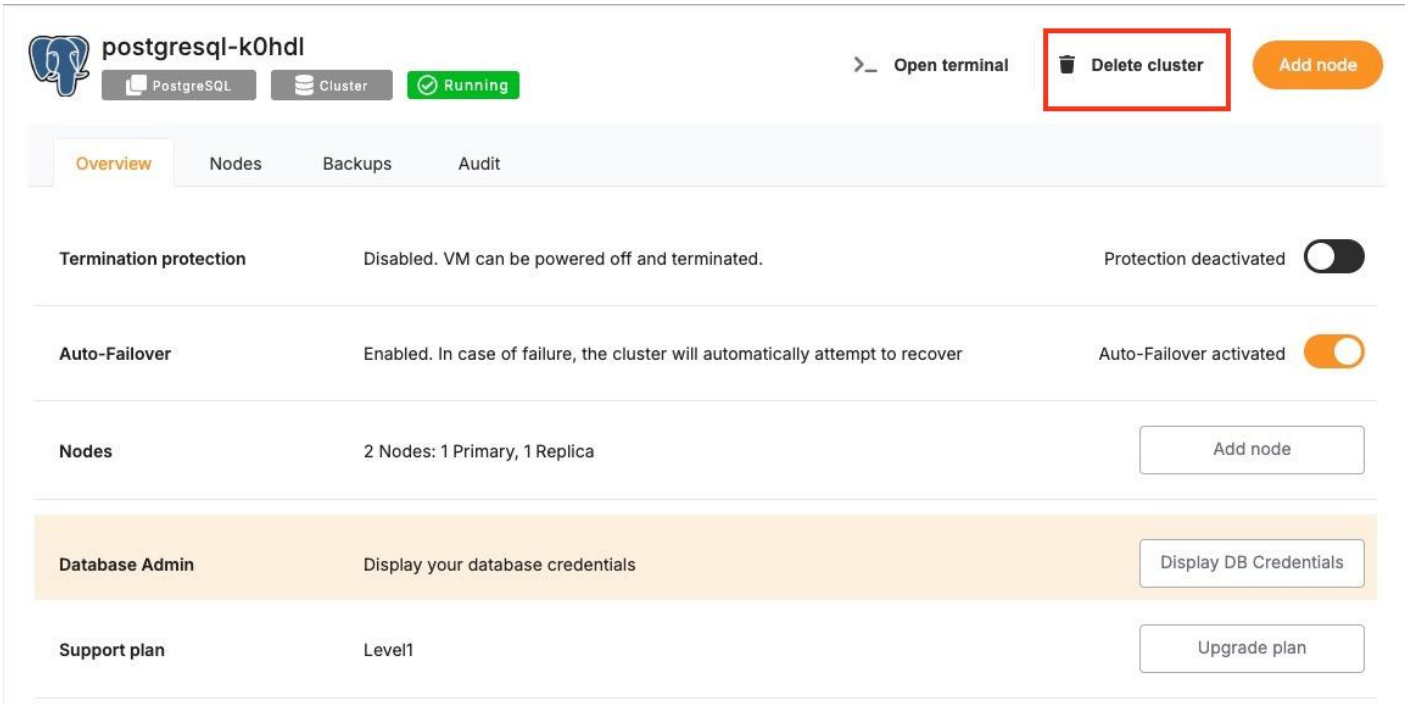
When a cluster is no longer needed—whether it was created for testing, staging, or an obsolete workload—deleting it helps free up resources and maintain a clean infrastructure footprint. Elestio provides a straightforward and secure way to delete entire clusters directly from the dashboard. This action permanently removes the associated services, data, and compute resources tied to the cluster.

When to Delete a Cluster

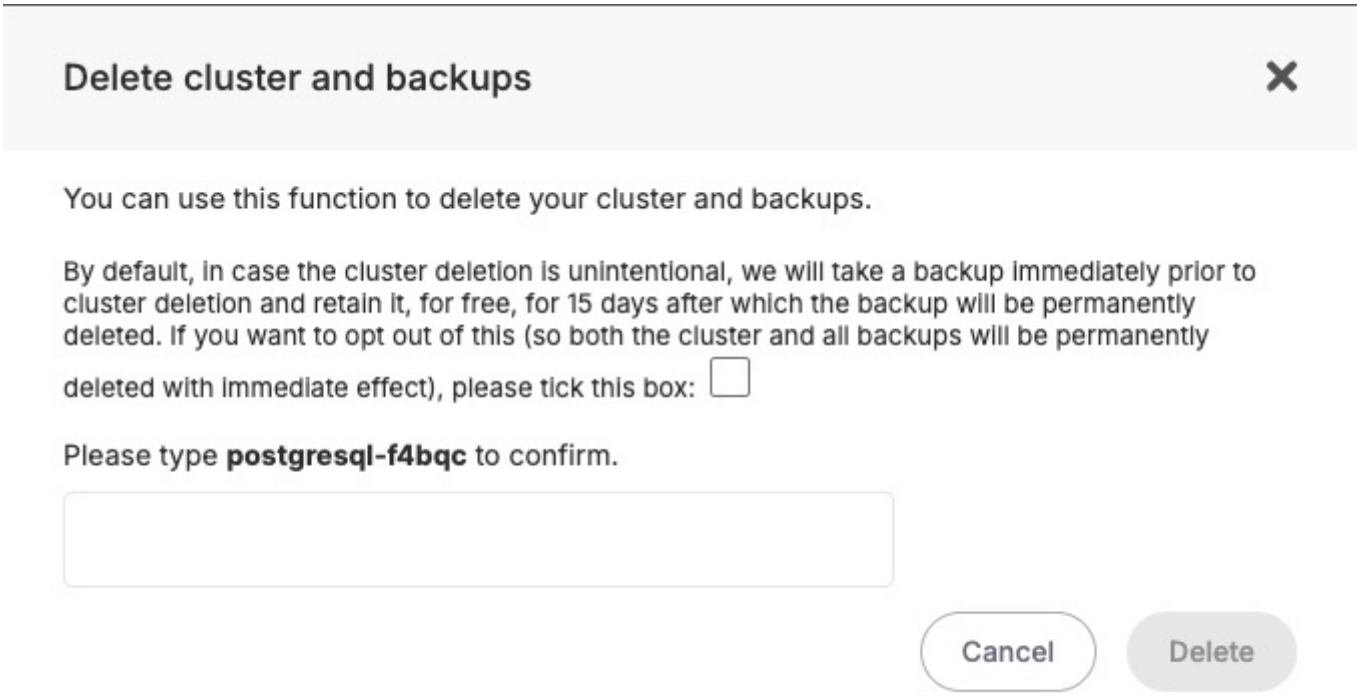
Deleting a cluster is a final step often performed when decommissioning an environment. This could include shutting down a test setup, replacing infrastructure during migration, or retiring an unused production instance. In some cases, users also delete and recreate clusters as part of major version upgrades or architectural changes. It is essential to confirm that all data and services tied to the cluster are no longer required or have been backed up or migrated before proceeding. Since cluster deletion is irreversible, any services, volumes, and backups associated with the cluster will be permanently removed.

Delete a Cluster

To delete a cluster, log in to the Elestio dashboard and navigate to the **Clusters** section. From the list of clusters, select the one you want to remove. Inside the selected cluster, you'll find a **navigation bar** at the top of the page. One of the available options in this navigation bar is **"Delete Cluster."**



Clicking this opens a confirmation dialog that outlines the impact of deletion. It will clearly state that deleting the cluster will **permanently remove** all associated services, storage, and configurations. By acknowledging a warning or typing in the cluster name, depending on the service type. Once confirmed, Elestio will initiate the deletion process, which includes tearing down all resources associated with the cluster. This typically completes within a few minutes, after which the cluster will no longer appear in your dashboard.



Considerations Before Deleting

Deleting a cluster also terminates any linked domains, volumes, monitoring configurations, and scheduled backups. These cannot be recovered once deletion is complete, so plan accordingly before confirming the action. If the cluster was used for production workloads, consider archiving data to external storage (e.g., S3) or exporting final snapshots for compliance and recovery purposes.

Before deleting a cluster, verify that:

- All required data has been backed up externally (e.g., downloaded dumps or exports).
- Any active services or dependencies tied to the cluster have been reconfigured or shut down.
- Access credentials, logs, or stored configuration settings have been retrieved if needed for auditing or migration.