

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Програмне забезпечення для резервного копіювання
баз даних»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень. Використання
ідей, результатів і текстів інших авторів мають посилання на відповідне джерело*

Артем ПАВЛЕНКО

(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Артем ПАВЛЕНКО

Керівник:

Володимир САДОВЕНКО

канд. фіз.-мат. наук,
доцент

Рецензент:

Київ 2026

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут інформаційних технологій**

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

_____Павленку Артему Ігоровичу_____

1. Тема кваліфікаційної роботи: « Програмне забезпечення для резервного копіювання баз даних »

керівник кваліфікаційної роботи канд. фіз.-мат. наук, доцент Володимир САДОВЕНКО,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методики фізичного копіювання реляційних СУБД, концепція журналювання транзакцій

(WAL) та забезпечення надійності RPO (Recovery Point Objective) та RTO (Recovery Time Objective)

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

1. Проаналізувати предметну галузь систем керування базами даних, дослідити класифікацію методів резервного копіювання, концепції WAL-архівзації та показники надійності RPO і RTO.

2. Провести огляд та аналіз існуючого програмного забезпечення для резервного копіювання PostgreSQL та MySQL, визначити їхні архітектурні особливості, переваги та недоліки.

3. Сформулювати вимоги до розробки програмного модуля для управління процесами бекапу, включаючи підтримку повного та диференціального копіювання, а також автоматизацію відновлення.

4. Визначити та обґрунтувати вибір засобів реалізації, архітектурних патернів та технологічного стеку (мова Python, бібліотеки взаємодії з СУБД, SQLite для каталогізації).

5. Проектування та розробка програмного продукту, опис реалізації логіки створення копій, ведення каталогу метаданих та модуля верифікації цілісності даних.

6. Розробка та тестування функціоналу відновлення баз даних, зокрема PITR-відновлення PostgreSQL та базового фізичного відновлення MySQL з резервної копії XtraBackup, а також перевірка коректності роботи системи в інтерактивному режимі.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.

2. Вимоги до системи.

3. Архітектура системи.

4. UML-діаграми.

5. Каталог резервних копій.

6. Графічний і консольний інтерфейси.

7. Резервне копіювання та відновлення PostgreSQL.

8. Резервне копіювання та відновлення MySQL.

9. Результати тестування.

10. Аналіз фізичного та логічного резервного копіювання.

11. Апробація результатів дослідження

Дата видачі завдання «23» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури з резервного копіювання баз даних	23.02–28.02.2026	
2	Аналіз існуючих систем резервного копіювання PostgreSQL і MySQL	01.03–07.03.2026	
3	Аналіз методів фізичного та логічного резервного копіювання	08.03–14.03.2026	
4	Проектування архітектури системи резервного копіювання та відновлення	15.03–28.03.2026	
5	Програмна реалізація системи резервного копіювання та відновлення	29.03–20.04.2026	
6	Реалізація сценаріїв PostgreSQL restore, PITR та MySQL restore	21.04–30.04.2026	
7	Тестування, валідація резервних копій та аналіз результатів	01.05–10.05.2026	
8	Оформлення пояснювальної записки та підготовка презентації	11.05–18.05.2026	
9	Попередній захист кваліфікаційної роботи	19.05–22.05.2026	

Здобувач вищої освіти

_____ Артем ПАВЛЕНКО
(підпис)

Керівник кваліфікаційної роботи

_____ Володимир САДОВЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра:
89 стор., 15 табл., 25 рис., 35 джерел.

Мета роботи – розроблення програмного забезпечення для автоматизації резервного копіювання та відновлення баз даних PostgreSQL і MySQL із підтримкою фізичних сценаріїв резервного копіювання, валідації артефактів та механізмів відновлення.

Об'єкт дослідження – процес резервного копіювання та відновлення баз даних PostgreSQL і MySQL.

Предмет дослідження – програмне забезпечення для автоматизованого створення, каталогізації, перевірки та відновлення фізичних резервних копій баз даних.

Короткий зміст роботи: У роботі проаналізовано існуючі підходи до резервного копіювання баз даних, зокрема логічне та фізичне резервне копіювання, WAL-архівацію, Point-in-Time Recovery та механізми інкрементального копіювання. Проведено аналіз існуючих систем резервного копіювання PostgreSQL і MySQL, зокрема pgBackRest, Barman, pg_basebackup, Percona XtraBackup та mysqldump.

У роботі спроєктовано архітектуру системи резервного копіювання, яка включає модулі PostgreSQL і MySQL backup/restore, SQLite-каталог резервних копій, механізм формування metadata.json і validation.json, систему валідації резервних копій, консольний і графічний інтерфейси.

Програмно реалізовано створення повних і похідних резервних копій PostgreSQL, повних та інкрементальних резервних копій MySQL, сценарії PostgreSQL restore, Point-in-Time Recovery і MySQL restore. Реалізовано механізми перевірки артефактів резервної копії та тестового відновлення.

Проведено модульне та інтеграційне тестування системи, перевірено коректність backup/restore-сценаріїв, роботу WAL-архівації, формування службових метаданих і сценарії валідації пошкоджених резервних копій.

Сферою застосування розробленого програмного забезпечення є навчальні, тестові та дослідницькі середовища, у яких необхідна автоматизація резервного копіювання та відновлення PostgreSQL і MySQL.

КЛЮЧОВІ СЛОВА: РЕЗЕРВНЕ КОПІЮВАННЯ, ВІДНОВЛЕННЯ БАЗ ДАНИХ, POSTGRESQL, MYSQL, PITR, WAL, PG_BASEBACKUP, PERCONA XTRABACKUP, SQLITE, ВАЛІДАЦІЯ РЕЗЕРВНИХ КОПІЙ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	10
ВСТУП	12
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	15
1.1 Аналіз процесів резервного копіювання та відновлення баз даних	15
1.2 Огляд існуючих рішень	17
1.3 Визначення недоліків існуючих систем та обґрунтування власної розробки....	23
1.4 Формування функціональних та нефункціональних вимог до системи	25
2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ.....	31
2.1 Аналіз та вибір засобів реалізації	31
2.2 Загальна архітектура системи	33
2.3 Проєктування підсистем резервного копіювання	38
2.4 Проєктування підсистеми відновлення.....	40
2.5 Проєктування каталогу резервних копій та метаданих	45
2.6 Проєктування Web UI	48
3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ.....	52
3.1 Реалізація модулів повного та похідного резервного копіювання	52
3.2 Реалізація пайплайну відновлення та роботи з WAL-архівами.....	56
3.3 Формування та валідація артефактів	59
3.4 Реалізація графічного (Web UI) та консольного (CLI) інтерфейсів	61
4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	66
4.1 Автоматизовані unit-тести	66
4.2 Тестування механізмів резервного копіювання і відновлення	68
4.3 Оцінка продуктивності та безпеки роботи системи.....	79
ВИСНОВКИ	81
ПЕРЕЛІК ПОСИЛАНЬ	85
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ.....	89
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ ПРОГРАМНИХ МОДУЛІВ	96

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД — база даних

СУБД — система керування базами даних

ПЗ — програмне забезпечення

API (*Application Programming Interface*) — програмний інтерфейс застосунку

CLI (*Command Line Interface*) — інтерфейс командного рядка

GUI (*Graphical User Interface*) — графічний інтерфейс користувача

LSN (*Log Sequence Number*) — номер послідовності запису в журналі транзакцій

PITR (*Point-in-Time Recovery*) — технологія відновлення бази даних на конкретний момент часу

RPO (*Recovery Point Objective*) — цільова точка відновлення (максимально допустимий обсяг втрати даних у часі)

RTO (*Recovery Time Objective*) — цільовий час відновлення (час, необхідний для повернення системи до працездатності)

WAL (*Write-Ahead Logging*) — метод журналювання транзакцій «запис наперед»

UUID (*Universally Unique Identifier*) — універсальний унікальний ідентифікатор

OLTP (*Online Transaction Processing*) — системи обробки транзакцій у реальному часі

JSON (*JavaScript Object Notation*) — текстовий формат обміну даними

Strategy — архітектурний патерн «Стратегія»

Pipeline — архітектурний патерн «Конвеєр»

XtraBackup — утиліта фізичного резервного копіювання та відновлення MySQL/MariaDB.

mysqld — серверний процес MySQL.

mysqladmin — службова утиліта MySQL для адміністративних операцій, зокрема перевірки доступності сервера.

copy-back — етап відновлення Percona XtraBackup, під час якого підготовлені файли резервної копії переносяться до директорії даних MySQL.

Backup — резервна копія даних або процес створення копії для подальшого відновлення системи.

Restore — процес відновлення бази даних із резервної копії.

Full backup — повна резервна копія, що містить усі дані бази на момент створення.

Differential backup — похідна резервна копія, яка містить зміни відносно базової повної копії.

Incremental backup — інкрементальна резервна копія, яка містить лише зміни з моменту попередньої копії.

Validation — процес перевірки цілісності та коректності резервної копії.

ВСТУП

У сучасних інформаційних системах бази даних є одним із ключових елементів програмної інфраструктури. У них зберігаються облікові записи користувачів, фінансові операції, службова інформація, історія дій, налаштування систем та інші дані, втрата яких може призвести до зупинки роботи програмного забезпечення або значних організаційних наслідків. Через це резервне копіювання та відновлення баз даних є важливою складовою адміністрування, супроводу та забезпечення надійності програмних систем.

Актуальність теми зумовлена тим, що навіть стабільні СУБД не захищені повністю від апаратних збоїв, помилок адміністрування, пошкодження файлів, некоректних оновлень або випадкового видалення даних. У таких ситуаціях важливо не лише мати резервну копію, а й розуміти, чи є вона цілісною, до якого моменту можна відновити базу даних і скільки часу займе процес відновлення. Саме тому в системах резервного копіювання важливими є показники RPO та RTO, а також наявність механізмів перевірки створених артефактів.

На практиці для PostgreSQL та MySQL існує багато окремих інструментів резервного копіювання, зокрема pg_basebackup, pgBackRest, Barman, Percona XtraBackup та mysqldump. Кожне з цих рішень має власну область застосування, переваги та обмеження. Частина інструментів орієнтована лише на одну СУБД, частина потребує складного налаштування, а деякі не забезпечують зручного ведення каталогу резервних копій або автоматизованої перевірки артефактів. Через це виникає потреба у програмному забезпеченні, яке поєднує базові сценарії резервного копіювання та відновлення PostgreSQL і MySQL в одному середовищі керування.

Об'єктом дослідження є процес резервного копіювання та відновлення баз даних PostgreSQL і MySQL.

Предметом дослідження є програмне забезпечення для автоматизованого створення, каталогізації, перевірки та відновлення фізичних резервних копій баз даних.

Метою роботи є розроблення програмного забезпечення для автоматизації резервного копіювання та відновлення баз даних PostgreSQL і MySQL із підтримкою фізичних сценаріїв резервного копіювання, валідації артефактів та механізмів відновлення.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Проаналізувати предметну галузь резервного копіювання та відновлення баз даних.
2. Розглянути основні підходи до логічного та фізичного резервного копіювання.
3. Дослідити особливості WAL-архівації PostgreSQL, Point-in-Time Recovery та фізичного копіювання MySQL за допомогою XtraBackup.
4. Провести аналіз існуючих програмних засобів резервного копіювання PostgreSQL і MySQL.
5. Визначити функціональні та нефункціональні вимоги до розроблюваної системи.
6. Спроекувати архітектуру програмного забезпечення, включаючи модулі резервного копіювання, відновлення, валідації та каталогізації.
7. Реалізувати механізм збереження історії резервних копій у SQLite-каталозі.
8. Реалізувати формування службових файлів metadata.json та validation.json.
9. Реалізувати консольний інтерфейс і локальний Web UI для керування операціями.
10. Провести тестування сценаріїв створення резервних копій, перевірки артефактів та відновлення баз даних.

У процесі виконання роботи використано методи аналізу предметної галузі, порівняльного аналізу програмних засобів, об'єктно-орієнтованого проєктування, модульної декомпозиції програмного забезпечення, тестування backup/restore-сценаріїв та перевірки цілісності службових артефактів.

Практичне значення роботи полягає у створенні програмного засобу, який дозволяє спростити виконання типових операцій резервного копіювання та відновлення для PostgreSQL і MySQL. Розроблена система може використовуватися у навчальних, тестових та невеликих адміністративних середовищах, де важливими є зрозумілий запуск операцій, контроль історії резервних копій, наявність метаданих та можливість перевірки результатів.

У межах роботи розроблено систему, що підтримує створення повних і похідних резервних копій, роботу з WAL-архівами PostgreSQL, використання Persona XtraBackup для MySQL, ведення SQLite-каталогу, формування службових артефактів, базову валідацію резервних копій, а також сценарії відновлення. Результати тестування підтверджують працездатність основних функцій системи та можливість її подальшого розвитку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Інформаційні технології обробляють та генерують великі обсяги інформації, критично важливим аспектом ІТ-інфраструктури є надійне зберігання даних. У цьому розділі розглядаються основні процеси аварійного відновлення, ризики витрати даних, а також проводиться аналіз вже існуючих програмних рішень, для визначення вимог до системи автоматизації резервного копіювання.

1.1 Аналіз процесів резервного копіювання та відновлення баз даних

У процесі роботи з базами даних завжди існує ризик втрати інформації. Це може бути пов'язано як із технічними збоями, так і з помилками користувачів або програм. Тому одним із ключових елементів забезпечення надійності системи є резервне копіювання та можливість швидкого відновлення даних.

Для оцінки ефективності процесів відновлення використовуються два основні показники: RPO (Recovery Point Objective) та RTO (Recovery Time Objective). RPO визначає, який обсяг даних може бути втрачений у випадку збою. Наприклад, якщо RPO становить 10 хвилин, система допускає втрату змін, зроблених за останні 10 хвилин. RTO, у свою чергу, показує, скільки часу потрібно для повного відновлення роботи системи після аварії.

Існують два основних підходи до резервного копіювання: логічний і фізичний. Логічне резервне копіювання передбачає збереження даних у вигляді SQL-запитів. Наприклад, за допомогою утиліти `mysqldump` можна отримати файл, який містить структуру таблиць і всі записи. Такий підхід є зручним для перенесення даних між різними системами, однак відновлення великих обсягів інформації може займати значний час.

Фізичне резервне копіювання працює на рівні файлів бази даних. У цьому випадку копіюються безпосередньо дані, з якими працює система керування базами даних. Це дозволяє значно скоротити час відновлення, оскільки не потрібно виконувати велику кількість SQL-інструкцій. Саме тому фізичні резервні копії часто використовуються в системах, де важливим є мінімальний час простою.

Окремим важливим сценарієм відновлення є Point-in-Time Recovery (PITR) - відновлення бази даних до конкретного моменту часу. У PostgreSQL цей механізм базується на поєднанні повної фізичної резервної копії та послідовного застосування WAL-журналів, які містять записи про зміни в базі даних після створення повної копії. На практиці PITR дає змогу відновити систему не лише до стану на момент створення backup, а й до проміжного моменту перед помилковим видаленням даних, збоєм або некоректною транзакцією.

У даній роботі основна увага приділяється фізичному резервному копіюванню PostgreSQL та MySQL. Для PostgreSQL розглядається підхід із використанням повної фізичної копії разом із WAL-архівом, що створює основу для відновлення після збою та PITR-сценаріїв. Для MySQL використовується фізичне копіювання на основі Percona XtraBackup, зокрема повні та інкрементальні копії.

Такий поділ є важливим, оскільки обидві СУБД мають різні механізми журналювання та відновлення, але можуть бути об'єднані в межах єдиного інтерфейсу керування.

1.2 Огляд існуючих рішень

У практичних системах резервного копіювання зазвичай використовуються вже готові інструменти, які дозволяють автоматизувати створення та відновлення резервних копій баз даних. Такі рішення суттєво спрощують адміністрування і зменшують ризики втрати даних. Разом з тим, кожен із цих інструментів має свої особливості. Частина з них орієнтована на конкретні СУБД, інші вимагають складнішої конфігурації або додаткових компонентів для коректної роботи. Через це їх застосування не завжди є універсальним і зручним у різних сценаріях.

Варто розглянути найбільш поширені рішення: `pg_basebackup`, `pgBackRest`, `Barman`, `Percona XtraBackup` та `mysqldump`. Основна увага приділяється їх архітектурі, підходам до резервного копіювання, підтримці відновлення та обмеженням, які можуть впливати на вибір конкретного інструменту.

1.2.1 `pgBackRest`

`pgBackRest` - це утиліта для резервного копіювання PostgreSQL, яка працює на рівні файлової структури кластера. Вона підтримує створення повних, диференціальних та інкрементальних копій, роботу з WAL-журналами, стиснення даних і перевірку цілісності backup-артефактів. За рахунок роботи з WAL `pgBackRest` може використовуватися не лише для збереження резервних копій, але й для сценаріїв Point-in-Time Recovery.

Також утиліта підтримує передавання копій на віддалені сервери, зокрема через SSH [1]. Попри свої можливості, `pgBackRest` потребує досить детального налаштування і розуміння внутрішньої структури PostgreSQL. Це робить його менш зручним для простих або навчальних проєктів, де потрібне швидке розгортання рішення.

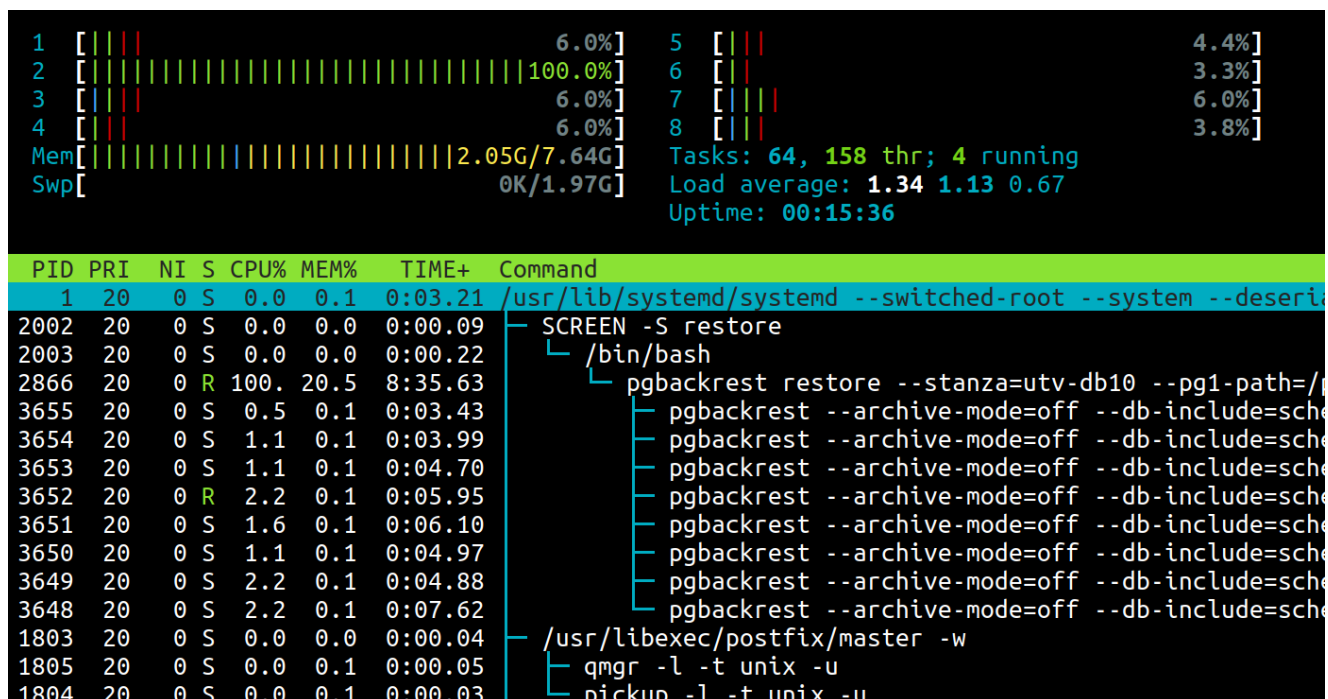


Рис. 1.1 Моніторинг процесу відновлення PostgreSQL (pgBackRest) у htop

1.2.2 Barman

Barman (Backup and Recovery Manager) - це інструмент для резервного копіювання PostgreSQL, який розробляється компанією EnterpriseDB. Його основна задача - централізоване зберігання резервних копій та WAL-файлів для одного або кількох PostgreSQL-серверів.

У більшості випадків Barman запускається на окремому backup-сервері. Саме туди передаються резервні копії та журнали WAL. Для цього можуть використовуватись rsync або streaming replication. За рахунок WAL-архівації Barman підтримує Point-in-Time Recovery, тобто дозволяє відновити базу даних до конкретного моменту часу.

Також у Barman є вбудовані перевірки резервних копій. Наприклад, через команду *barman check* можна перевірити стан архівації та підключення до PostgreSQL. Крім цього, підтримується автоматичне видалення старих backup-копій через *retention policy*. [2]

Barman досить часто використовується у великих PostgreSQL-інфраструктурах. Але при цьому його налаштування може бути складним, особливо якщо потрібно окремо налаштовувати архівацію WAL, SSH-доступ та backup-сервер. Також важливо враховувати, що цей інструмент працює тільки з PostgreSQL.

```
cluster-admin@toy-deploy-vm:~/dell-cluster/cnpg-system$ kubectl exec -it postgres-1 -- /bin/bash
Defaulted container "postgres" out of: postgres, bootstrap-controller (init), plugin-barman-cloud (init)
postgres@postgres-1:/$ export AWS_ACCESS_KEY_ID=[...]
postgres@postgres-1:/$ export AWS_SECRET_ACCESS_KEY=[...]
postgres@postgres-1:/$ barman-cloud-backup-list --cloud-provider aws-s3 --endpoint-url [URL] s3://[BUCKET] [CLUSTER]
Backup ID      End Time      Begin Wal      Archival Status  Name
20250923T161910  2025-09-23 16:19:13  000000010000000000000004      backup-20250923161909
20250923T164524  2025-09-23 16:45:25  000000010000000000000006      backup-20250923164522
postgres@postgres-1:/$ barman-cloud-backup-show --cloud-provider aws-s3 --endpoint-url [URL] s3://[BUCKET] [CLUSTER] 20250923T164524
Backup 20250923T164524:
Backup Name      : backup-20250923164522
Server Name      : cloud
System Id        : 7553316992291414041
Status           : DONE
PostgreSQL Version : 170006
PGDATA directory : /var/lib/postgresql/data/pgdata
Estimated Cluster Size : 30.4 MiB
Server information:
Checksums        : off
WAL summarizer   : off
Base backup information:
Timeline         : 1
Begin WAL        : 000000010000000000000006
End WAL          : 000000010000000000000006
Begin time       : 2025-09-23 16:45:24.538056+00:00
End time         : 2025-09-23 16:45:25.491644+00:00
Copy time        : 5 seconds
Begin Offset     : 40
End Offset       : 288
Begin LSN        : 0/6000028
End LSN          : 0/6000120
postgres@postgres-1:/$
exit
cluster-admin@toy-deploy-vm:~/dell-cluster/cnpg-system$
```

Рис. 1.2 Приклад використання Barman Cloud для роботи з резервними копіями PostgreSQL

1.2.3 Percona XtraBackup

Percona XtraBackup використовується для фізичного резервного копіювання MySQL та MariaDB. Утиліта розробляється компанією Percona і в основному застосовується для роботи з великими базами даних, де небажано зупиняти сервер під час створення резервної копії. [3]

На відміну від звичайного `mysqldump`, тут копіюються не SQL-команди, а самі файли бази даних і redo log. Через це резервні копії створюється швидше, особливо якщо база даних містить велику кількість таблиць або даних.

Підтримуються повні та інкрементальні резервні копії. Також можна використовувати стиснення файлів і запускати копіювання у кілька потоків. Це дозволяє зменшити час створення резервних копій, хоча саме налаштування XtraBackup може бути складнішим порівняно з логічними методами резервного копіювання. Важливо враховувати, що Persona XtraBackup працює тільки з MySQL та MariaDB і не підтримує PostgreSQL. Для відновлення фізичної копії XtraBackup використовується окремий етап підготовки. Спочатку виконується команда `xtrabackup --prepare`, яка застосовує журнали транзакцій до скопійованих файлів і приводить копію до консистентного стану. Після цього дані можуть бути перенесені у нову директорію MySQL за допомогою `xtrabackup --copy-back`. У розробленій утиліті ці дії автоматизовані в мінімальному restore pipeline для MySQL.

```

o [demo_db]> full database -path /Users/artempavlenko/Documents/backups
Starting full MySQL backup with xtrabackup → /Users/artempavlenko/Documents/backups
2026-05-01 17:04:19 - backup_demo_db - INFO - Starting full backup: full_demo_db_20260501_140419_31ab
Using login-path 'xtrabackup' for xtrabackup authentication
Running xtrabackup... (this may take a while)
✓ Full MySQL backup created at /Users/artempavlenko/Documents/backups/full_demo_db_20260501_140419_31ab
Backup size: 1.59 MB
2026-05-01 17:04:22 - backup_demo_db - INFO - Full backup directory: /Users/artempavlenko/Documents/backups/full_demo_db_20260501_140419_31ab
2026-05-01 17:04:22 - backup_demo_db - INFO - Backup size: 1671348 bytes

=====
Creating Single Archive (zstd)
=====
Compressing backup → full_demo_db_20260501_140419_31ab.tar.zst
Using zstd (fast compression & decompression)
  Archiving (level 3, 4 threads)...
✓ ✓ Archive created: full_demo_db_20260501_140419_31ab.tar.zst
  Original: 1.59 MB
  Compressed: 1.58 MB
  Saved: 0.7%
2026-05-01 17:04:22 - backup_demo_db - INFO - Archive created: /Users/artempavlenko/Documents/backups/full_demo_db_20260501_140419_31ab.tar.zst
2026-05-01 17:04:22 - backup_demo_db - INFO - Compression ratio: 0.7%
✓ Original backup directory preserved for differential backups
2026-05-01 17:04:22 - backup_demo_db - INFO - Keeping original directory for differential backup access: /Users/artempavlenko/Documents/backups/full_demo_db_20260501_140419_31ab
✓ ✓ Single archive ready: full_demo_db_20260501_140419_31ab.tar.zst
2026-05-01 17:04:22 - backup_demo_db - INFO - Backup completed: full_demo_db_20260501_140419_31ab (2.96s, 1.59 MB)

```

Рис. 1.3 Приклад виконання повного резервного копіювання MySQL засобами Persona XtraBackup у розробленій утиліті

1.2.4 mysqldump

`mysqldump` - це стандартна утиліта MySQL для логічного резервного копіювання баз даних. Вона входить до складу MySQL Server і використовується для створення SQL-файлу, у якому зберігаються команди створення таблиць та вставки даних.

Робота `mysqldump` побудована на формуванні SQL-запитів типу `CREATE TABLE` та `INSERT INTO`. Під час відновлення ці команди повторно виконуються сервером MySQL, через що процес `restore` може займати досить багато часу при великих обсягах даних.

Утиліту зручно застосовувати у випадках, коли потрібно створити резервну копію окремих таблиць або швидко експортувати дані у текстовий формат.

Разом з тим, логічний підхід має і свої обмеження. При роботі з великими базами даних швидкість резервного копіювання та відновлення стає значно нижчою порівняно з фізичними методами. Крім цього, `mysqldump` не підтримує повноцінне `Point-in-Time Recovery` без додаткового використання бінарних журналів MySQL.

```
(.venv) artempavlenko@MacBook-Air-Artem DatabaseBackupUtility % mysqldump --login-path=xtrabackup --verbose --databases demo_db
> demo_db_logical.sql
-- Connecting to localhost...
-- Retrieving table structure for table users...
-- Sending SELECT query...
-- Retrieving rows...
-- Disconnecting from localhost...
(.venv) artempavlenko@MacBook-Air-Artem DatabaseBackupUtility % head -n 30 demo_db_logical.sql
-- MySQL dump 10.13 Distrib 8.4.9, for macos26.4 (arm64)
--
-- Host: localhost    Database: demo_db
--
-- Server version      8.4.9

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!50503 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;
```

Рис. 1.4 Приклад використання `mysqldump` для логічного резервного копіювання

1.2.5 pg_basebackup

pg_basebackup - це стандартна утиліта PostgreSQL для створення фізичних резервних копій. Вона входить до складу PostgreSQL і часто використовується як базовий інструмент для подальшого відновлення сервера.

Робота утиліти побудована на replication protocol, тому копіювання виконується через мережеве підключення до PostgreSQL без прямого доступу до каталогу з файлами бази даних. Під час процесу також можуть передаватись WAL-журнали, необхідні для коректного відновлення даних.

pg_basebackup підтримує створення копій у tar-форматі, стиснення файлів та передачу архівів через мережу. Для запуску утиліти користувач PostgreSQL повинен мати права REPLICATION, а параметр wal_level має бути встановлений у режим replica.[6]

Утиліта добре підходить для створення повних фізичних копій PostgreSQL. Проте в pg_basebackup відсутній власний механізм керування каталогом резервних копій або автоматична перевірка їх цілісності. Через це в реальних системах її часто використовують разом з додатковими інструментами або власною логікою автоматизації.

```
[root@redhat-pgtest data]# pg_basebackup -Ft -Pv -Xs -z -D /data/pg_backup -Z5 -h pgm-gs55tg3921pdg04l.pgsql.singapore.rds.aliyuncs.com -p 5432 -U rambo -W
Password:
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 36/60000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created temporary replication slot "pg_basebackup_1855491"
68291/68291 kB (100%), 1/1 tablespace
pg_basebackup: write-ahead log end point: 36/6000D040
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: syncing data to disk ...
pg_basebackup: renaming backup_manifest.tmp to backup_manifest
pg_basebackup: base backup completed
[root@redhat-pgtest data]#
```

Рис. 1.5 Створення резервної копії PostgreSQL за допомогою pg_basebackup

1.3 Визначення недоліків існуючих систем та обґрунтування власної розробки

Після аналізу існуючих інструментів можна зробити висновок, що більшість із них добре виконують свої задачі, але мають досить вузьку спеціалізацію. Наприклад, `pg_basebackup`, `pgBackRest` та `Barman` орієнтовані на PostgreSQL, тоді як `Percona XtraBackup` використовується для MySQL та MariaDB. Через це адміністратору, який працює з різними СУБД, доводиться використовувати кілька окремих інструментів, кожен із яких має власні команди, формат конфігурації та особливості відновлення.

Ще однією проблемою є складність налаштування. Такі рішення, як `pgBackRest` або `Barman`, є достатньо потужними, але для їх повноцінного використання потрібно правильно налаштувати WAL-архівацію, права доступу, шляхи до сховищ, політики зберігання та, в окремих випадках, додатковий сервер резервного копіювання. [4] Для великих інфраструктур це є нормальним підходом, але для невеликих проєктів, навчального середовища або тестового стенду така складність може бути надмірною. Окремо варто виділити питання контролю створених резервних копій. У багатьох інструментах інформація про копії зберігається або у власному форматі, або частково залежить від файлової структури сховища. При цьому не завжди є зручний єдиний каталог, у якому можна швидко побачити історію створення копій, їх тип, стан, зв'язок із попередніми копіями та службові маркери відновлення.

У практичній роботі це ускладнює контроль ланцюгів резервного копіювання, особливо якщо використовуються повні та похідні копії. Також важливо, що факт створення резервної копії ще не гарантує її придатність до відновлення. На практиці може виникнути ситуація, коли файли копії існують, але під час відновлення виявляється відсутній WAL-сегмент, помилка конфігурації або пошкоджений артефакт. Тому для надійнішої роботи бажано не тільки створювати копії, а й зберігати метадані, контрольні суми та результати перевірок. З урахуванням цих обмежень було прийнято рішення розробити власну утиліту для автоматизації

резервного копіювання та відновлення баз даних PostgreSQL і MySQL. Її мета не полягає в повній заміні промислових рішень на кшталт pgBackRest або Barman.

Основна ідея полягає в тому, щоб об'єднати базові механізми фізичного копіювання двох СУБД в одному інтерфейсі та додати до них каталогізацію, метадані й перевірку створених артефактів. У розробленій системі для PostgreSQL використовується нативна утиліта pg_basebackup, а для MySQL - Percona XtraBackup.[5] При цьому система використовує не власний формат збереження даних, а механізми самих СУБД та спеціалізованих утиліт. Власна логіка застосунку відповідає за координацію процесів, збереження інформації в SQLite-каталозі, формування файлів metadata.json і validation.json, а також за спрощення запуску операцій через єдиний інтерфейс. Таким чином, розроблена утиліта поєднує кілька можливостей, які в існуючих рішеннях зазвичай реалізуються окремими інструментами.

Це підтримка PostgreSQL і MySQL, централізований каталог резервних копій, збереження метаданих, перевірка артефактів, PITR-відновлення PostgreSQL та базове автоматизоване відновлення MySQL з фізичної копії XtraBackup. Система орієнтована насамперед на навчальні, тестові та невеликі адміністративні сценарії, де важливими є зрозумілий контроль над процесом і мінімальна кількість ручних дій.

Таблиця 1.1

Порівняльний аналіз існуючих рішень

Критерій	pgBackRest	Barman	Percona XtraBackup	mysqldump	Розроблена утиліта
Підтримка PostgreSQL	+	+	-	-	+
Підтримка MySQL	-	-	+	+	+
Тип копіювання	фізичне	фізичне	фізичне	логічне	фізичне
Повне резервне копіювання	+	+	+	+	+
Похідні копії: диференціальні / інкрементальні / WAL-based	+	+	+	-	+ PostgreSQL WAL-based / MySQL incremental

Порівняльний аналіз існуючих рішень

Критерій	pgBackRest	Barman	Percona XtraBackup	mysqldump	Розроблена утиліта
PITR	+	+	обмежено	через binlog	+ для PostgreSQL
WAL-архівація	+	+	-	-	+
Єдиний CLI для PostgreSQL і MySQL	-	-	-	-	+
Каталог резервних копій	+	+	-	-	+ SQLite
metadata.json / validation.json	частково	частково	частково	-	+
Web UI	- / окремо	- / окремо	-	-	+
Складність налаштування	висока	висока	середня	низька	середня
Автоматизоване відновлення MySQL	-	-	частково / через CLI	через SQL-імпорт	+ базове restore через XtraBackup

1.4 Формування функціональних та нефункціональних вимог до системи

Після аналізу існуючих рішень можна сформулювати основні вимоги до розроблюваної системи. Вимоги мають враховувати як сам процес резервного копіювання та відновлення, так і зручність керування створеними копіями. Оскільки система орієнтована на роботу з PostgreSQL та MySQL, важливо було передбачити єдиний підхід до запуску операцій, збереження метаданих і перевірки результатів.

Основною задачею системи є автоматизація процесів створення резервних копій баз даних, збереження інформації про них та виконання базових сценаріїв відновлення. При цьому система не повинна повністю замінювати промислові рішення, а має виступати як зручний інструмент для навчальних, тестових і невеликих адміністративних сценаріїв.

У вимогах використовується узагальнений термін “похідна резервна копія”. Для PostgreSQL він означає копію, сформовану на основі WAL-сегментів після останньої повної резервної копії. Для MySQL йдеться про інкрементальну копію Percona XtraBackup, яка створюється з використанням параметра `--incremental` та базової директорії попередньої повної копії.

1.4.1 Функціональні вимоги

До функціональних вимог належать можливості, які система повинна безпосередньо виконувати.

1. Система повинна підтримувати створення повної фізичної резервної копії PostgreSQL з використанням стандартної утиліти `pg_basebackup`.
2. Система повинна підтримувати створення похідної резервної копії PostgreSQL на основі WAL-сегментів, які зберігаються у визначеній директорії архіву. Така копія не є повторною повною фізичною копією кластера, а містить WAL-дані та службову інформацію, необхідні для продовження ланцюга відновлення після останньої повної резервної копії.
3. Система повинна підтримувати створення повної фізичної резервної копії MySQL або MariaDB з використанням Percona XtraBackup.
4. Система повинна підтримувати створення інкрементальної резервної копії MySQL на основі попередньої повної копії з використанням механізму Percona XtraBackup `--incremental`.
5. Система повинна зберігати інформацію про всі створені копії в локальному SQLite-каталозі. У каталозі мають фіксуватися тип СУБД, тип копії, шлях до артефактів, статус виконання, час створення та зв'язок із попередніми копіями.
6. Для кожної операції система повинна формувати службові файли `metadata.json` та `validation.json`, призначення і структура детально розглядаються в підрозділі 2.5

7. Система повинна підтримувати відновлення PostgreSQL з повної резервної копії та WAL-архіву. У процесі відновлення має автоматично створюватися нова директорія даних, конфігурація відновлення та файл `recovery.signal`.

8. Система повинна підтримувати Point-in-Time Recovery для PostgreSQL, тобто відновлення бази даних до заданого моменту часу за наявності повної резервної копії, WAL-архіву та коректного WAL-ланцюга. Для цього під час `restore` має формуватися `recovery`-конфігурація з параметром `restore_command` та, за потреби, цільовим часом відновлення.

9. Система повинна підтримувати автоматизоване відновлення MySQL з фізичної резервної копії Percona XtraBackup. Процес має включати перевірку структури XtraBackup-каталогу, розпакування стиснених артефактів за їх наявності, виконання `xtrabackup --prepare`, перенесення файлів у нову директорію даних через `xtrabackup -copy-back`, запуск окремого екземпляра `mysqld` на заданому порту та перевірку доступності через `mysqladmin ping`.

10. Система повинна мати інтерфейс командного рядка для запуску основних операцій: створення копії, відновлення, перегляд службової інформації та виконання SQL-запитів.

11. Система повинна мати локальний Web UI для перегляду історії резервних копій, запуску операцій, перегляду журналів виконання, а також файлів `metadata.json` і `validation.json`.

12. Система повинна підтримувати кілька способів конфігурації доступу до СУБД: через параметри командного рядка, через файл конфігурації та через захищені профілі.

13. Для PostgreSQL система повинна підтримувати використання `.pgpass`, а для MySQL - `login-path`, що дозволяє не передавати паролі напряму через аргументи командного рядка.

1.4.2 Нефункціональні вимоги

Нефункціональні вимоги визначають загальні властивості системи: надійність, безпеку, швидкодію, зручність використання та можливість подальшого розширення.

Для розроблюваної утиліти ці вимоги є важливими, оскільки робота з резервними копіями потребує не тільки створення файлів, а й контролю їх стану, збереження службових даних та можливості подальшого відновлення.

1. Система повинна перевіряти наявність основних артефактів після кожної операції. Для PostgreSQL обов'язковими файлами є `base.tar.gz`, `pg_wal.tar.gz`, `metadata.json` та `validation.json`. Для MySQL мають перевірятися службові файли та артефакти Percona XtraBackup, зокрема `xtrabackup_info`, `xtrabackup_checkpoints`, `ibdata1` та директорія `mysql`. Якщо частина файлів збережена у форматі `.zst`, система повинна розпакувати їх перед етапом перевірки. Якщо хоча б один обов'язковий файл відсутній, операція не повинна позначатися як успішна.

2. Для кожної створеної резервної копії має формуватися файл `validation.json`, у якому фіксується стан перевірки. У випадку PostgreSQL повна копія повинна проходити додаткову перевірку відновлення: запуск тестового екземпляра, виконання запиту `SELECT 1`, перевірка кількості записів у таблицях та порівняння контрольних значень. Результат перевірки має бути збережений у службових метаданих.

3. Інформація про кожну операцію повинна зберігатися в SQLite-каталозі. Для кожної резервної копії мають фіксуватися щонайменше такі поля: ідентифікатор копії, назва бази даних, тип СУБД, тип копії, час початку, тривалість виконання, статус, шлях до артефактів, шлях до `metadata.json` та `validation.json`. Завдяки цьому історія резервного копіювання аналізується через структурований каталог, а не лише через ручний перегляд директорій у файловій системі.

4. Команди перегляду каталогу резервних копій мають виконуватися без помітної затримки для користувача. Для локального каталогу з кількістю записів до 1000 час

отримання списку копій не повинен перевищувати 2 секунд. Це обмеження є достатнім для навчального та невеликого адміністративного середовища.

5. Система повинна фіксувати тривалість кожної операції у секундах. Це потрібно для подальшого аналізу RTO та оцінки ефективності різних способів резервного копіювання. Наприклад, у каталозі має зберігатися поле `duration_seconds`, яке дозволяє порівнювати швидкість створення повних і похідних копій.

6. Система не повинна вимагати передачі пароля у відкритому вигляді через аргументи командного рядка. Для PostgreSQL має підтримуватися використання файлу `.pgpass`, а для MySQL - `login-path`. Використання ручного введення пароля допускається, але не повинно бути рекомендованим способом конфігурації.

7. Основна логіка системи повинна працювати локально, без обов'язкової залежності від хмарних сервісів або зовнішніх серверів. Усі службові дані мають зберігатися локально: каталог - у SQLite, метадані - у JSON-файлах, журнали виконання - у локальних log-файлах. Це спрощує розгортання системи в тестовому середовищі.

8. Після запуску операції більшість технічних дій має виконуватися автоматично: створення директорії, запуск зовнішньої утиліти, формування метаданих, перевірка артефактів і запис результату до каталогу. Користувач не повинен вручну створювати або редагувати службові файли, описані в підрозділі 2.5.

9. У разі помилки система повинна виводити не лише факт збою, а й його причину. Наприклад, якщо відсутній WAL-сегмент, повідомлення має вказувати, що відновлення до заданого моменту часу неможливе через неповний WAL-ланцюг. Це важливо, оскільки помилка має бути зрозумілою не тільки розробнику, а й користувачу утиліти.

10. Архітектура системи повинна дозволяти додавання нових стратегій резервного копіювання без повної зміни основної логіки. Наприклад, окремі стратегії для PostgreSQL та MySQL мають підключатися через спільний сервіс керування.

11. Система повинна використовувати стандартні або поширені інструменти: `pg_basebackup`, `pg_ctl`, `psql` для PostgreSQL та `xtrabackup`, `zstd`, `mysqld`, `mysqladmin` для MySQL. Використання поширених утиліт зменшує ризик створення нестандартного формату резервних копій і спрощує подальше відновлення даних.

12. Система орієнтована на локальне або тестове середовище, а також на невеликі адміністративні сценарії. На поточному етапі вона не заявляється як промислове рішення для великих інфраструктур із десятками серверів. Це обмеження є важливим, оскільки дозволяє коректно визначити межі застосування розробки

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Аналіз та вибір засобів реалізації

Під час проєктування системи резервного копіювання потрібно було обрати такі засоби реалізації, які дозволять працювати з різними СУБД, запускати зовнішні утиліти, зберігати історію створених копій та виконувати базову перевірку результатів. Тому вибір технологій робився не лише за принципом зручності розробки, а й з урахуванням того, чи можна буде надалі розширювати систему.

Основною мовою програмування було обрано Python. Python має зручні засоби для роботи з файловою системою, процесами операційної системи, JSON-файлами, SQLite та зовнішніми командами. У межах розробленої утиліти Python використовується не для самостійного копіювання файлів бази даних, а для координації роботи інших компонентів: запуску системних утиліт, перевірки конфігурації, формування метаданих та запису інформації до каталогу.

Для взаємодії з PostgreSQL використовується бібліотека `psycopg2-binary`. Вона дозволяє виконувати SQL-запити, перевіряти доступність бази даних, отримувати службову інформацію та виконувати просту перевірку після відновлення. Саме резервне копіювання PostgreSQL виконується через стандартну утиліту `pg_basebackup`, оскільки вона є нативним інструментом PostgreSQL для створення фізичних копій. Такий підхід є більш надійним, ніж самостійне копіювання каталогу даних, оскільки процес відбувається через механізми самої СУБД.

Для MySQL було обрано бібліотеку `PyMySQL` та зовнішню утиліту `Percona XtraBackup`. `PyMySQL` використовується для перевірки підключення та службової взаємодії з сервером, а `XtraBackup` - для створення фізичних копій і базового відновлення даних. У підсистемі `restore` для MySQL застосовуються етапи `xtrabackup --prepare` та `xtrabackup --copy-back`, після чого відновлений екземпляр запускається через окремий процес `mysqld` на заданому порту. Це важливо, тому що логічні методи

на зразок `mysqldump` можуть бути занадто повільними для великих баз даних. XtraBackup працює безпосередньо з файлами InnoDB і підтримує повне та інкрементальне копіювання.

Для збереження історії операцій було обрано SQLite. У цьому випадку не потрібен окремий сервер бази даних, а сам каталог зберігається у локальному файлі `backup_catalog.db`. У ньому фіксується інформація про створені копії: тип СУБД, тип копії, шлях до артефактів, статус операції, час виконання та зв'язок між повними і похідними копіями. Такий підхід спрощує роботу з історією резервного копіювання і дозволяє не покладатися лише на структуру папок у файловій системі.

Для збереження службової інформації про копії використовуються файли `metadata.json` і `validation.json`, структура та призначення яких розглядаються в підрозділі 2.5.

Для зменшення обсягу збережених даних у системі передбачено використання `tar` та `zstd`. `tar` застосовується для формування архіву, а `zstd` - для швидкого стиснення. Це особливо корисно для повних фізичних копій, які можуть займати значний обсяг на диску.

Для керування системою використовується інтерфейс командного рядка. Основні параметри команд обробляються через стандартні можливості Python, а для інтерактивної роботи може застосовуватися `prompt_toolkit`. Це дозволяє запускати операції створення копій, відновлення та перегляду службової інформації без складного графічного середовища. Додатково в проєкті передбачено локальний Web UI, який спрощує перегляд історії, журналів виконання та службових файлів.

У результаті обраний набір технологій формує основу для модульної архітектури системи: окремо виділяються клієнти PostgreSQL і MySQL, SQLite-каталог, сервіс валідації, модулі відновлення та інтерфейси керування. Така структура спрощує супровід системи і залишає можливість для подальшого розширення без повної зміни основної логіки.

2.2 Загальна архітектура системи

Архітектура розробленої системи побудована за багаторівневим принципом. У ній окремо виділяються інтерфейси користувача, координаційний рівень, клієнти для роботи з конкретними СУБД, підсистема каталогізації та фізичне сховище артефактів. Такий поділ потрібен через те, що система працює одночасно з двома різними СУБД - PostgreSQL та MySQL.

На верхньому рівні розташовані інтерфейси взаємодії з користувачем: командний інтерфейс та локальний Web UI. Через них користувач запускає створення резервної копії, відновлення, перегляд історії операцій або перевірку службових файлів. Обидва інтерфейси не виконують основну логіку самостійно, а лише передають команду до координаційного рівня системи.

Центральним елементом архітектури є координаційний рівень. Він відповідає за вибір потрібного сценарію роботи: створення повної копії, створення похідної копії, запуск відновлення або перевірку результатів. Саме цей рівень визначає, який клієнт або сервіс потрібно використати залежно від обраної СУБД і типу операції.

Для роботи з PostgreSQL використовується окремий клієнтський компонент. Він відповідає за перевірку підключення, запуск `pg_basebackup`, роботу з WAL-архівом та підготовку даних для відновлення. Для MySQL використовується окремий клієнт, який взаємодіє з Percona XtraBackup і забезпечує створення повних та інкрементальних фізичних копій.

Окрему роль у системі виконує підсистема каталогізації. Вона зберігає історію операцій у SQLite-каталозі, де фіксуються тип СУБД, тип копії, статус, шлях до артефактів, час виконання та зв'язки між копіями. Завдяки цьому система не залежить лише від структури папок у файловій системі, а має окреме джерело інформації про стан і послідовність створених копій.

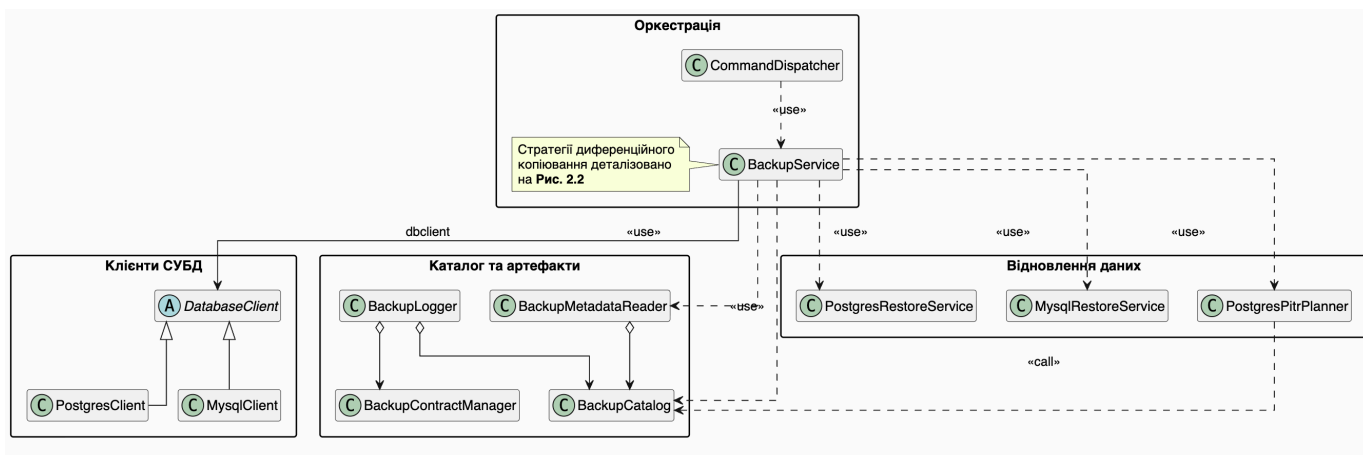


Рис. 2.1 Класова структура основних компонентів системи

На рисунку 2.1 наведено базову класову структуру основних компонентів системи. Центральним координатором прикладних операцій є сервіс BackupService, який працює з абстрактним класом DatabaseClient і делегує низькорівневі операції конкретним клієнтам - PostgresClient та MysqlClient. Взаємодія користувача з системою починається з компонента CommandDispatcher, який виконує функції диспетчеризації CLI-команд та передає керування до BackupService.

Для СУБД PostgreSQL реалізовано PostgresClient, який відповідає за створення повних фізичних копій та роботу з WAL-архівом. Для MySQL використовується MysqlClient, який інкапсулює взаємодію з інструментарієм Percona XtraBackup.

З метою дотримання принципу єдиної відповідальності (SRP) операції відновлення даних винесені в окремі класи інфраструктурного рівня: PostgresRestoreService, MysqlRestoreService та планувальник PostgresPitrPlanner. Це дозволило повністю ізолювати складну логіку відновлення та аналізу часових міток від процесів створення копій.

Роботу з метаданими та системним каталогом відокремлено від логіки конкретних СУБД за допомогою класів BackupCatalog (збереження історії у SQLite), BackupMetadataReader та BackupContractManager (генерація службових артефактів metadata.json і validation.json).

Для реалізації диференційного копіювання в модулі застосовано шаблон проєктування Strategy, що зумовлено принциповими відмінностями у механізмах фіксації змін в PostgreSQL та MySQL. Деталізовану структуру класів цього патерну винесено на рисунок 2.2.

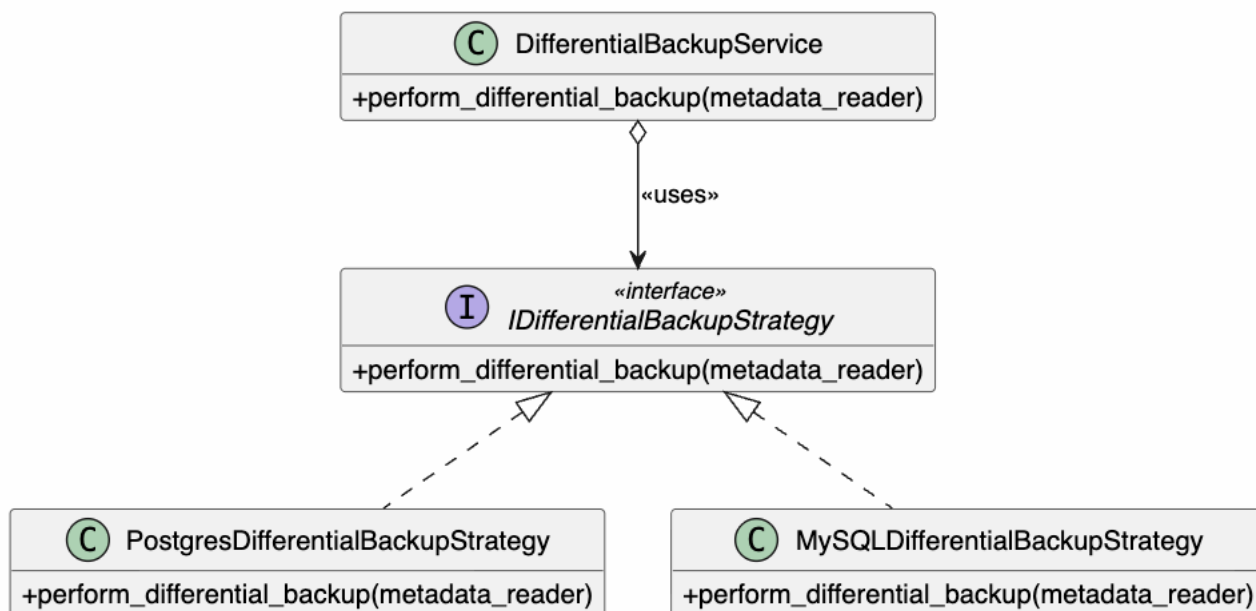


Рис. 2.2 Використання патерну Strategy для вибору сценарію похідного копіювання

У PostgreSQL похідне копіювання базується на WAL-архіві та фізичних копіях PostgreSQL-кластера, тоді як MySQL використовує механізм incremental backup у Percona XtraBackup. Через це система має спільний координаційний сервіс, але різні реалізації сценаріїв резервного копіювання для кожної СУБД.

У майбутньому це спрощує додавання нових сценаріїв або підтримку інших СУБД, оскільки координаційний рівень не потрібно переписувати повністю.

Крім SQLite-каталогу, система формує службові JSON-файли, які доповнюють запис у каталозі та описані в підрозділі 2.5.

Фізичне зберігання даних виконується у локальній файловій системі. Для PostgreSQL основними артефактами є base.tar.gz, pg_wal.tar.gz, WAL-архів та службові JSON-файли. Для MySQL зберігається каталог, сформований XtraBackup, а

також службові метадані. За потреби повна копія може додатково пакуватися в єдиний архів .tar.zst.

Таким чином, загальна архітектура системи побудована навколо розділення відповідальностей між компонентами. Інтерфейси відповідають за взаємодію з користувачем, координаційний рівень - за керування сценаріями, клієнти СУБД - за роботу з конкретними базами даних, а каталог і метадані - за контроль історії та стану створених копій.

Таблиця 2.1

Основні компоненти архітектури системи

Компонент	Призначення
CLI	Запуск основних команд системи через термінал
Local Web UI	Перегляд історії, логів, метаданих і запуск операцій через браузер
Backup/Restore Manager	Координація сценаріїв створення копій та відновлення
PostgresClient	Робота з PostgreSQL, запуск pg_basebackup, перевірка WAL
MysqlClient	Робота з MySQL, запуск xtrabackup
BackupCatalog	Збереження історії операцій у SQLite
Backup Contract Manager	Формування metadata.json і validation.json
Validation Service	Перевірка артефактів та результатів операцій
Restore Service	Відновлення PostgreSQL з повної копії та WAL-архіву
Local Storage	Збереження файлів резервних копій і службових артефактів

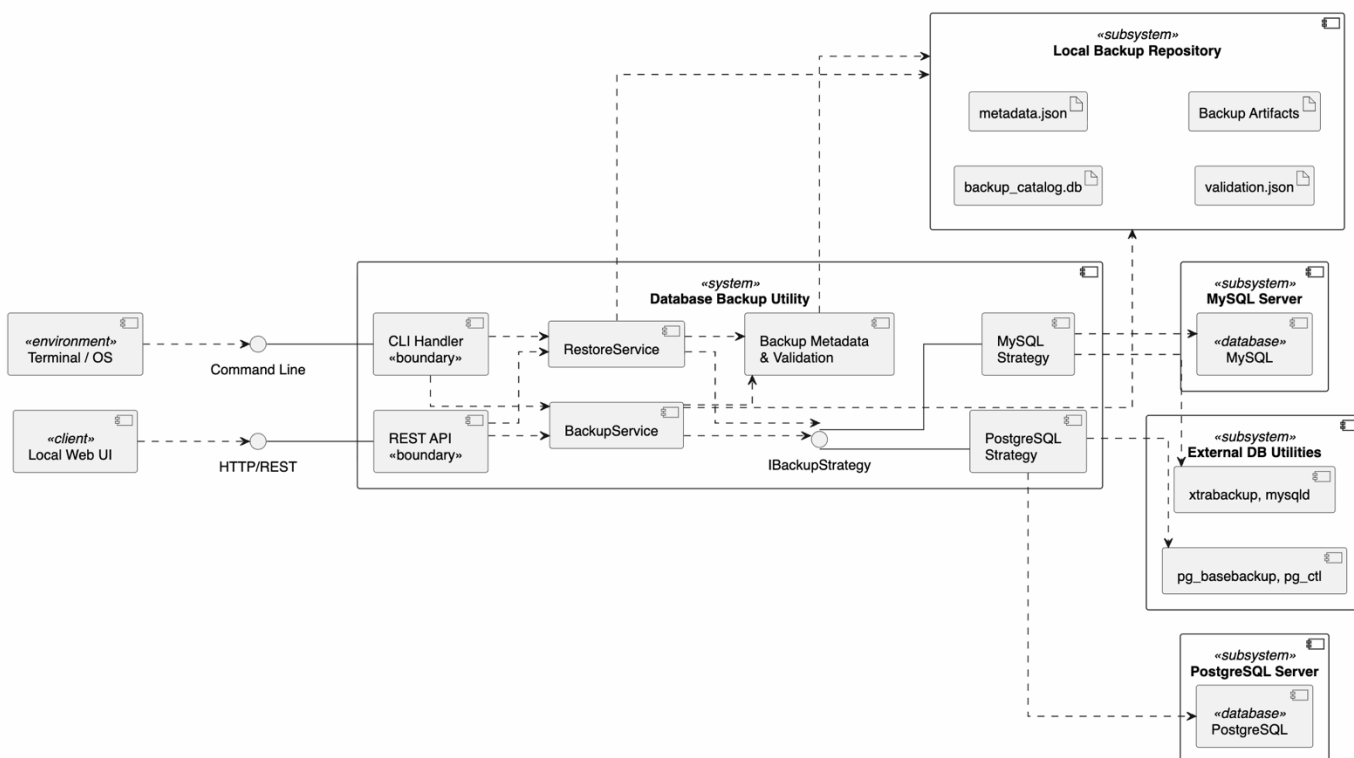


Рис. 2.3 UML-діаграма компонентів системи резервного копіювання та відновлення баз даних

На рисунку 2.3 наведено UML-діаграму компонентів розробленої системи. Вона показує взаємодію користувацьких інтерфейсів, сервісного рівня, стратегій резервного копіювання, локального сховища артефактів, серверів PostgreSQL і MySQL та зовнішніх утиліт баз даних.

Користувач взаємодіє із системою через консольний інтерфейс або локальний Web UI. Обидва інтерфейси передають команди до сервісного рівня, де BackupService відповідає за запуск сценаріїв резервного копіювання, а RestoreService — за сценарії відновлення. Для роботи з різними СУБД використовується підхід Strategy: PostgreSQL Strategy реалізує сценарії на основі pg_basebackup і WAL-архівів, а MySQL Strategy використовує Percona XtraBackup.

Окрему роль відіграють компоненти каталогізації та валідації. BackupCatalog зберігає інформацію про створені копії у backup_catalog.db, а BackupContractManager і ValidationService формують та перевіряють службові файли metadata.json і

validation.json. Фізичні артефакти резервних копій зберігаються в локальному сховищі, що дозволяє відокремити бізнес-логіку системи від конкретної файлової структури.

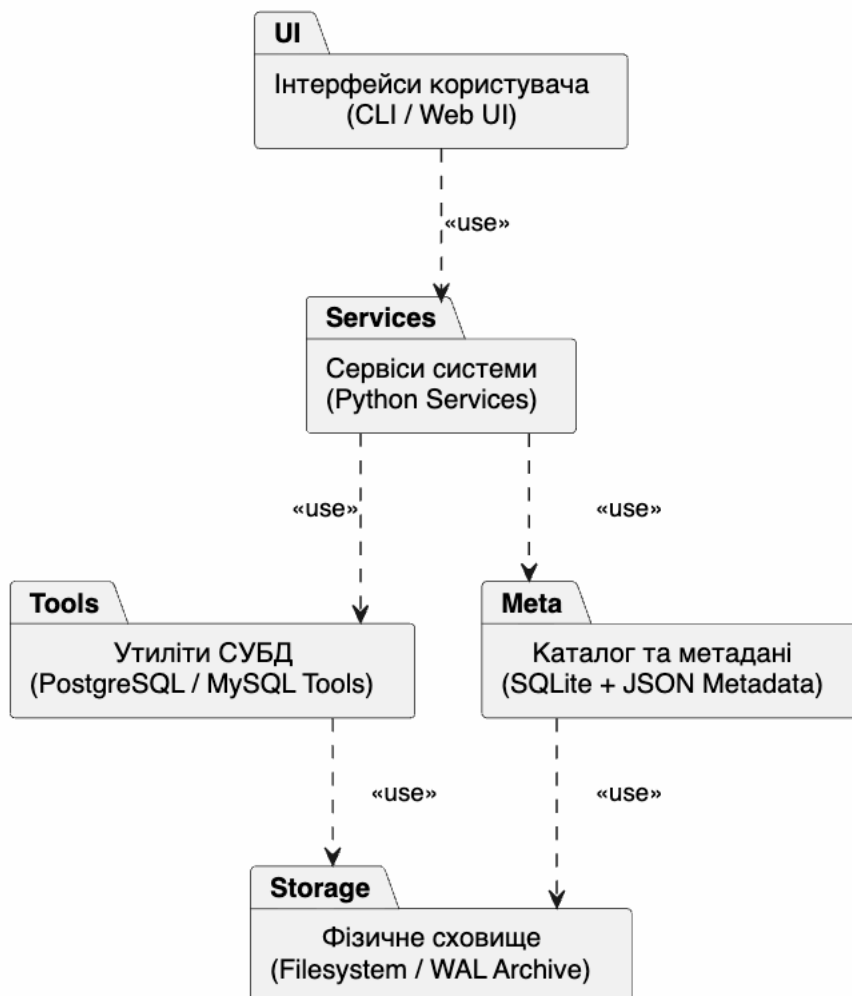


Рис. 2.4 Логічна структура засобів реалізації системи резервного копіювання

2.3 Проєктування підсистем резервного копіювання

На рисунку 2.5 показано алгоритм створення повної фізичної резервної копії PostgreSQL у розробленій системі. Процес починається із запуску CLI у режимі роботи з PostgreSQL та перевірки параметрів підключення. Після цього система створює об'єкт PostgresClient, визначає спосіб автентифікації та перевіряє можливість підключення до сервера бази даних.

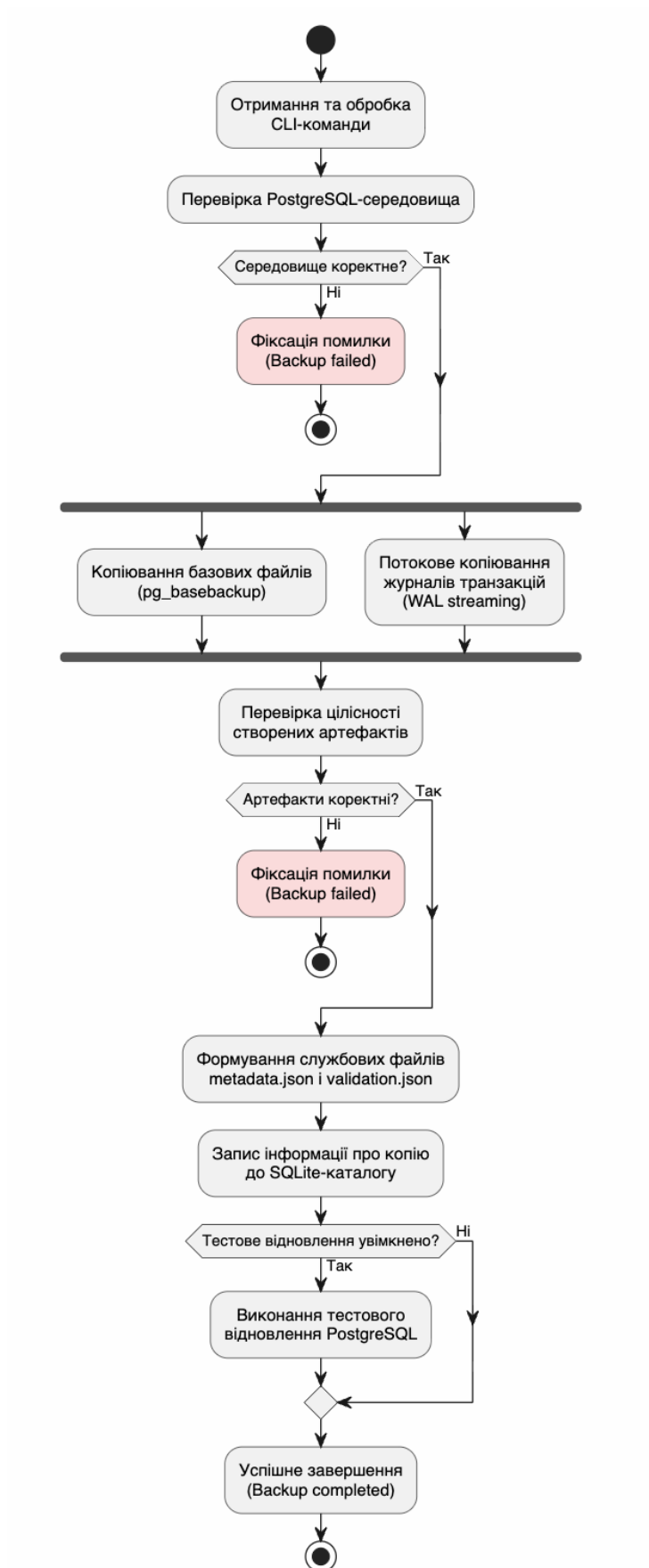


Рис. 2.5 Алгоритм створення повної фізичної резервної копії PostgreSQL

Перед запуском основної операції виконуються обов'язкові перевірки PostgreSQL-середовища: стан `archive_mode`, значення `wal_level`, наявність утиліти `pg_basebackup` та право користувача на реплікацію. Якщо хоча б одна з цих умов не виконується, створення копії не запускається, оскільки результат може бути непридатним для подальшого відновлення.

Основна фізична копія створюється за допомогою `pg_basebackup`. У системі використовується `tar`-формат із `gzip`-стисненням, а WAL-журнали передаються у потоковому режимі. Після завершення роботи `pg_basebackup` перевіряється наявність основних артефактів, зокрема `base.tar.gz` та `pg_wal.tar.gz`. Додатково зберігаються службові маркери PostgreSQL, такі як поточний LSN та назва WAL-файлу. Після створення основних файлів система може сформувати додатковий архів `tar.zst`. Цей архів не замінює основну директорію копії, оскільки вона використовується для подальшої побудови похідних копій. Якщо `tar` або `zstd` недоступні, це фіксується як попередження, але не вважається критичною помилкою.

Окремим етапом є автоматична перевірка відновлення. Якщо вона увімкнена, система створює тимчасовий екземпляр PostgreSQL, відновлює копію,

виконує тестовий запит `SELECT 1` та порівнює стан таблиць між вихідною і відновленою базою. У випадку невдалої перевірки файли копії можуть існувати, але фінальний статус операції буде позначений як `failed`.

Фінальним кроком є формування службових файлів і запис інформації до SQLite-каталогу. Структура `metadata.json` та `validation.json` детально розглядається в підрозділі 2.5, тому в цьому підрозділі вони розглядаються лише як завершальний етап `backup`-сценарію.

2.4 Проектування підсистеми відновлення

Підсистема відновлення призначена для розгортання баз даних із фізичних резервних копій. У розробленій системі передбачено два напрями відновлення: PostgreSQL `restore` з повної фізичної копії та WAL-архіву, а також базове MySQL

restore з директорії, створеної Percona XtraBackup.

Для PostgreSQL підтримуються два сценарії. Перший сценарій передбачає відновлення з явно заданої повної копії. Другий сценарій пов'язаний із PITR-відновленням, коли система використовує SQLite-каталог для вибору базової копії та WAL-архів для доведення бази до заданого моменту часу.

Для MySQL відновлення побудоване як послідовний pipeline на основі Percona XtraBackup. Він включає перевірку структури резервної копії, підготовку копії до запуску, перенесення файлів у нову директорію даних, запуск окремого екземпляра mysqld та перевірку його доступності. Детальна реалізація цього pipeline наведена в підрозділі 3.2, тому в цьому розділі розглядається лише загальна схема його роботи.

На рисунку 2.6 показано загальну логіку вибору restore-сценарію. Після запуску команди restore система визначає тип СУБД і передає виконання до відповідного сервісу. Для PostgreSQL основним елементом є recovery-конфігурація та робота з WAL-архівом, а для MySQL - послідовне виконання етапів XtraBackup *restore pipeline*.

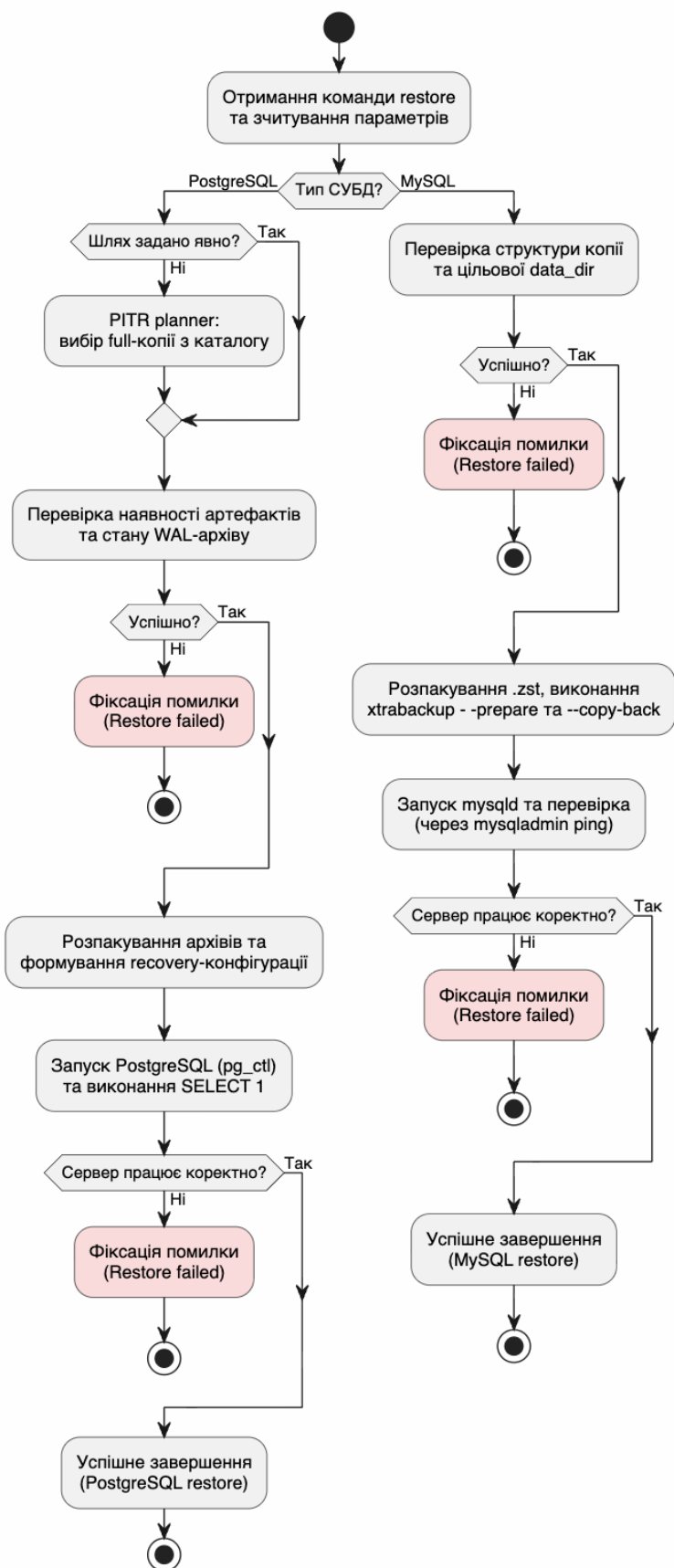


Рис. 2.6 Узагальнений алгоритм роботи підсистеми відновлення баз даних

2.4.1 Проєктування PostgreSQL restore та PITR

Для PostgreSQL підсистема відновлення повинна підтримувати два режими роботи: звичайне відновлення з явно заданої повної резервної копії та сценарій Point-in-Time Recovery. У першому випадку користувач безпосередньо передає шлях до повної копії та директорії WAL-архіву. У другому випадку система додатково отримує цільовий момент часу відновлення і використовує SQLite-каталог для вибору базової повної копії, яка передує заданому часу.

Після вибору базової копії система перевіряє наявність необхідних артефактів, зокрема `base.tar.gz`, `pg_wal.tar.gz` та WAL-архіву. Далі формується нова директорія даних, розпаковуються архіви, створюється `recovery`-конфігурація з параметром `restore_command`, а для PITR-сценарію також задається `recovery_target_time`. Після цього створюється файл `recovery.signal` і відновлений екземпляр PostgreSQL запускається через `pg_ctl`.

Після запуску система виконує базову перевірку працездатності відновленого екземпляра.

Таким чином, PITR у розробленій системі розглядається як окремий сценарій відновлення, у якому до звичайного `restore` додаються автоматичний вибір повної копії з каталогу та обмеження відновлення заданим моментом часу.[7]

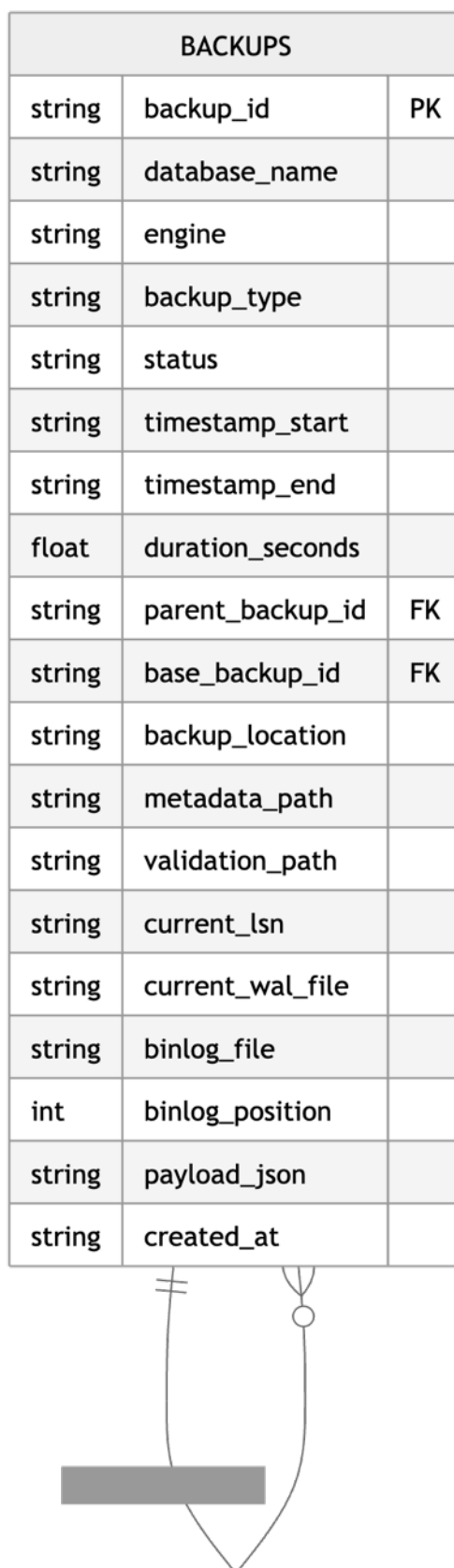


Рис. 2.7 Структура SQLite-каталогу резервних копій

Слід враховувати, що такий механізм орієнтований на PostgreSQL 12 і вище, оскільки в ньому використовується файл `recovery.signal`.

2.5 Проектування каталогу резервних копій та метаданих

Для коректної роботи системи недостатньо лише створити файли резервної копії у файловій системі. Потрібно також зберігати службову інформацію про кожну операцію: тип СУБД, тип копії, час створення, статус, шлях до артефактів та зв'язок з іншими копіями. Для цього в системі використовується локальний SQLite-каталог `backup_catalog.db`.

SQLite було обрано як легке локальне сховище, яке не потребує окремого серверного процесу. Це відповідає загальній ідеї утиліти: система повинна запускатися локально та не залежати від додаткової інфраструктури.

Каталог використовується як джерело інформації про історію створених копій і допомагає знаходити потрібну базову копію під час відновлення.

Основною таблицею каталогу є `backups`. У ній зберігається інформація про кожну створену копію: ідентифікатор, назва бази даних, тип СУБД, тип резервної копії, статус виконання, шлях до директорії з файлами, а також шляхи до `metadata.json` і `validation.json`.

На рисунку 2.7 показано структуру основної таблиці SQLite-каталогу. Поля `parent_backup_id` та `base_backup_id` використовуються для побудови ланцюгів резервних копій. `parent_backup_id` вказує на попередню копію у послідовності, а `base_backup_id` - на повну копію, від якої залежить поточна похідна копія.

Завдяки полям `parent_backup_id` та `base_backup_id` система розділяє ланцюги різних баз даних і може визначити, яку повну копію потрібно використати для подальшого відновлення. Це особливо важливо для PostgreSQL PITR, де базова копія може вибиратися автоматично за цільовим часом відновлення.

Таблиця 2.2

Основні поля таблиці backups у SQLite-каталозі

Поле	Призначення
backup_id	Унікальний ідентифікатор резервної копії
database_name	Назва бази даних
engine	Тип СУБД: PostgreSQL або MySQL
backup_type	Тип копії: full, differential або incremental
status	Стан операції: completed або failed
backup_location	Шлях до директорії з артефактами
metadata_path	Шлях до metadata.json
validation_path	Шлях до validation.json
parent_backup_id	Попередня копія у ланцюгу
base_backup_id	Повна копія, від якої залежить поточна
current_lsn	Поточний LSN для PostgreSQL
current_wal_file	Назва WAL-файлу PostgreSQL
binlog_file	Назва binary log для MySQL
binlog_position	Позиція у binary log MySQL
payload_json	Додаткові службові дані у JSON-форматі

2.5.1 Про metadata.json

Окрім запису в SQLite-каталозі, кожна резервна копія має власний файл metadata.json. Він містить нормалізований технічний опис копії: тип СУБД, тип операції, шляхи до артефактів, службові маркери PostgreSQL або MySQL, контрольні суми та короткий стан перевірки. Такий файл потрібний у ситуації, коли резервна копія аналізується окремо від основного SQLite-каталогу, наприклад після перенесення директорії backup на інший носій.

2.5.2 Про validation.json

Файл `validation.json` використовується для фіксації результатів перевірки резервної копії. У ньому зберігається інформація про наявність обов'язкових артефактів, стан службових маркерів, результати перевірки структури `metadata.json` та, за потреби, результат тестового відновлення. Якщо обов'язкові файли відсутні або перевірка відновлення завершується помилкою, копія не повинна позначатися як `completed`. Саме тому `validation.json` використовується як службове підтвердження того, що копія не лише створена, а й пройшла базову перевірку.

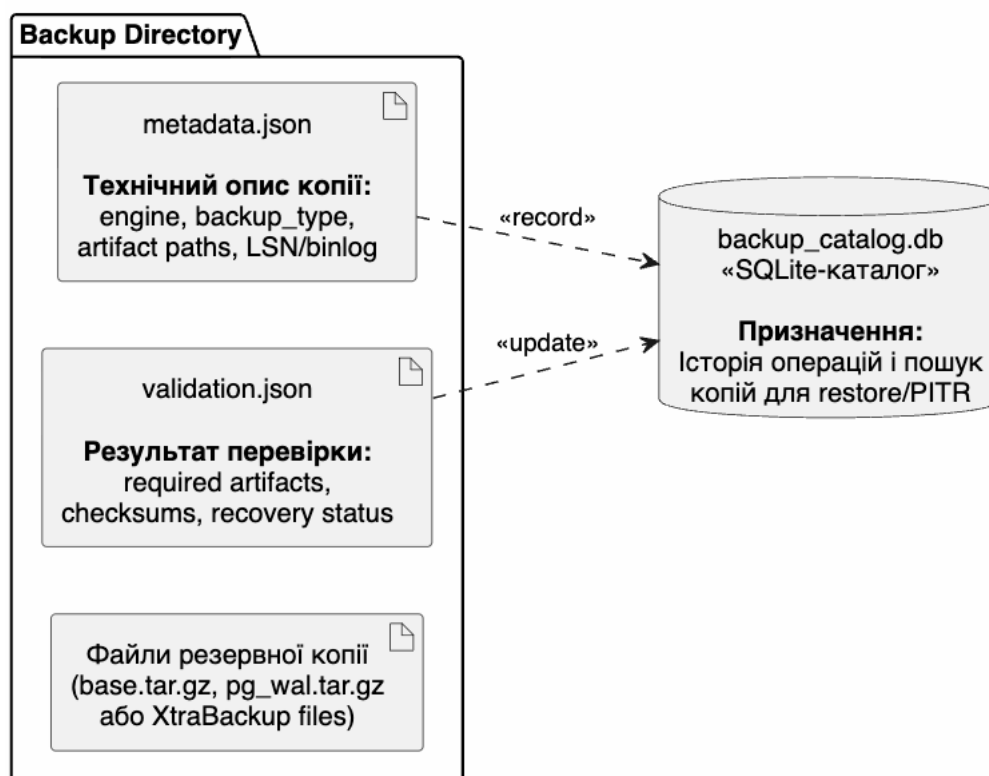


Рис. 2.8 Зв'язок резервної копії з метаданими, валідацією та SQLite-каталогом

2.5.3 Проектування підсистеми валідації резервних копій

Окрім автоматичної перевірки, яка може виконуватися після завершення резервного копіювання, у системі передбачено окремий сценарій валідації вже створеної резервної копії. Для цього реалізовано команду `validate backup-path <backup_directory>`, яка дозволяє перевірити придатність `backup`-директорії без

повторного запуску операції створення копії.

Під час виконання валідації система визначає тип СУБД і тип резервної копії, після чого перевіряє наявність обов'язкових артефактів. Для PostgreSQL це, зокрема, `base.tar.gz`, `pg_wal.tar.gz`, `metadata.json` і `validation.json`, а для MySQL - службові файли XtraBackup та основні артефакти фізичної копії. Додатково перевіряється структура `metadata.json`, службові маркери, а також, за потреби, контрольні суми й результат тестового відновлення.

Результат виконання сценарію валідації фіксується у файлі `validation.json`. Якщо резервна копія вже присутня в SQLite-каталозі, підсистема валідації також може синхронізувати оновлений статус із відповідним записом у `backup_catalog.db`. Таким чином, команда `validate` використовується як окремий механізм контролю цілісності та придатності резервних копій до відновлення.

2.6 Проєктування Web UI

Web UI у розробленій системі спроектовано як локальну панель керування для виконання основних операцій резервного копіювання та перегляду службової інформації. Він не замінює CLI, а використовує ті самі клієнти та сервіси, що й консольна частина системи. Такий підхід дозволяє зберегти єдину логіку роботи незалежно від того, як користувач запускає операцію: через термінал або через браузер.

Інтерфейс складається з backend-сервера та статичної frontend-частини. Backend реалізовано у файлі `webui/server.py` на основі стандартного Python HTTP-сервера `ThreadingHTTPServer`. Він обробляє HTTP-запити, читає SQLite-каталог резервних копій, відкриває `metadata.json` і `validation.json`, а також запускає операції резервного копіювання або перевірки відновлення у фонових потоках.

Frontend розміщений у директорії `webui/static` і складається з файлів `index.html`, `styles.css` та `app.js`. `index.html` відповідає за структуру сторінки, `styles.css` - за

оформлення, а app.js - за перемикання вкладок, надсилення HTTP-запитів до backend-сервера та відображення результатів у браузері.

Загальна структура взаємодії компонентів Web UI наведена на рисунку 2.8

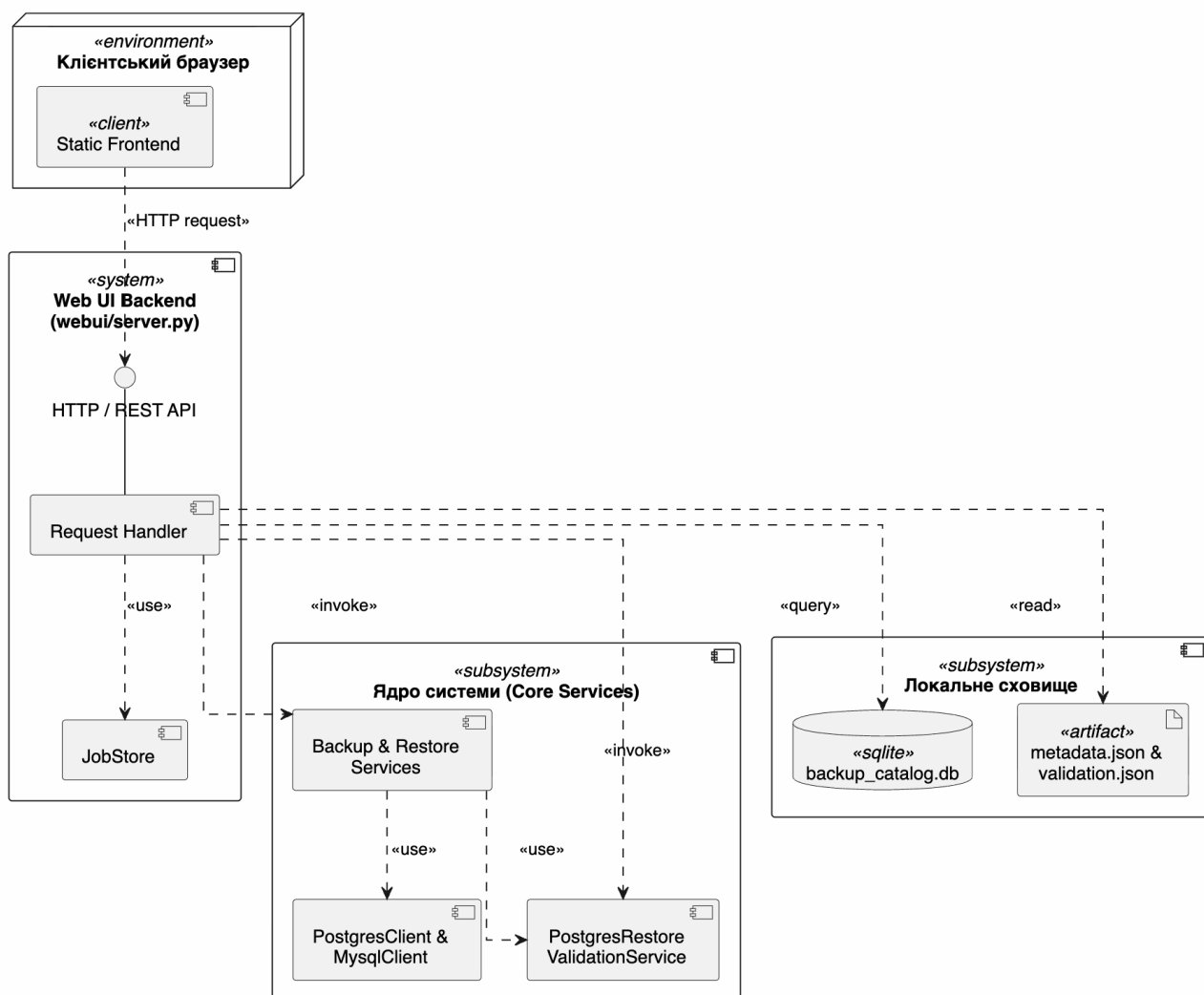


Рис. 2.9 Структура взаємодії компонентів Web UI

Для виконання довготривалих операцій у Web UI використовується механізм фонових задач.

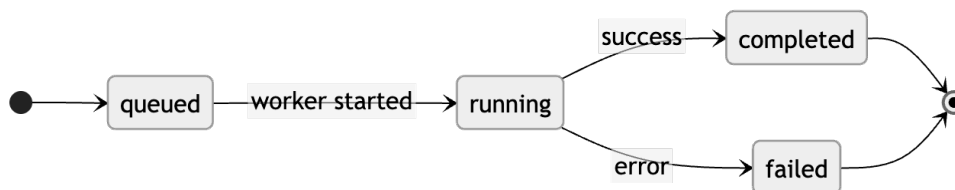


Рис. 2.10 Життєвий цикл фонові задачі у Web UI

Далі задача переводиться у стан running, а після завершення отримує статус completed або failed залежно від результату виконання.

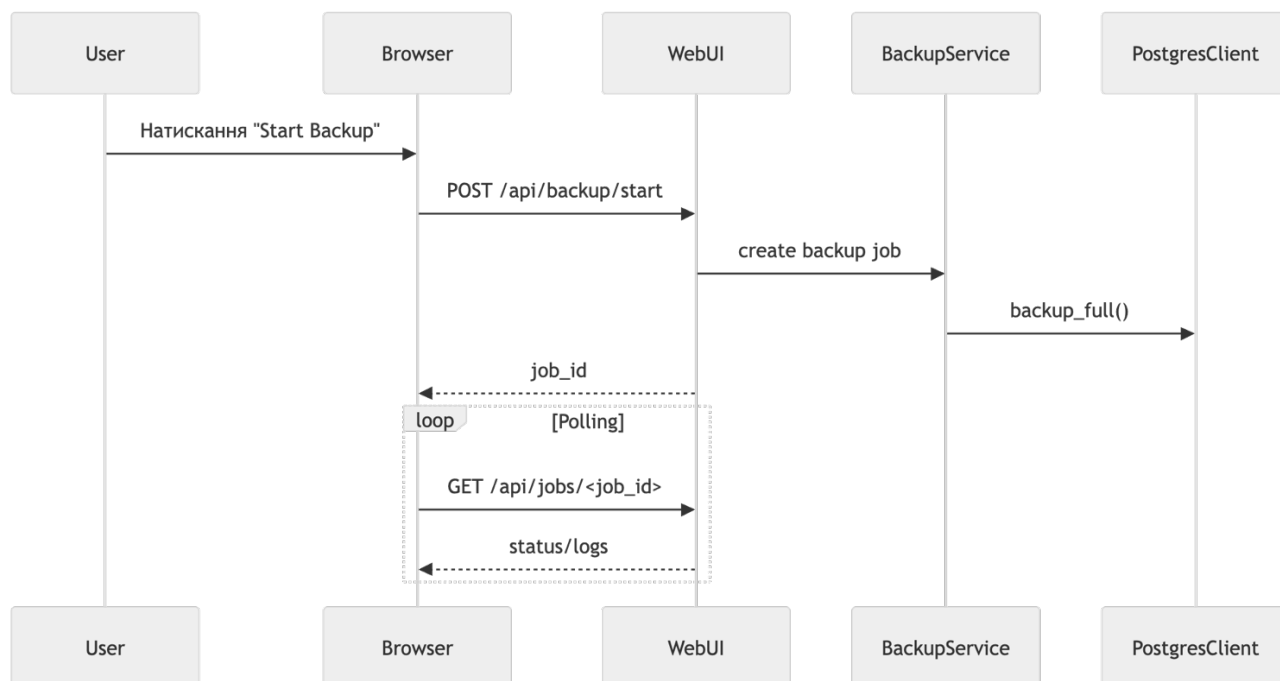


Рис. 2.11 Взаємодія frontend і backend під час запуску резервного копіювання

Таблиця 2.3

Основні API ендпоінти Web UI

Endpoint	Призначення
GET /api/health	Перевірка доступності Web UI
GET /api/catalog	Отримання списку резервних копій з каталогу
GET /api/backup/document	Перегляд metadata.json або validation.json
GET /api/jobs	Отримання списку задач
GET /api/jobs/<job_id>	Отримання стану конкретної задачі
POST /api/backup/start	Запуск повного резервного копіювання
POST /api/restore/validate	Запуск перевірки відновлення PostgreSQL

Окремо послідовність взаємодії між інтерфейсом користувача, координаційним рівнем, клієнтами СУБД, каталогом і файловою системою показано на рисунку 2.11.

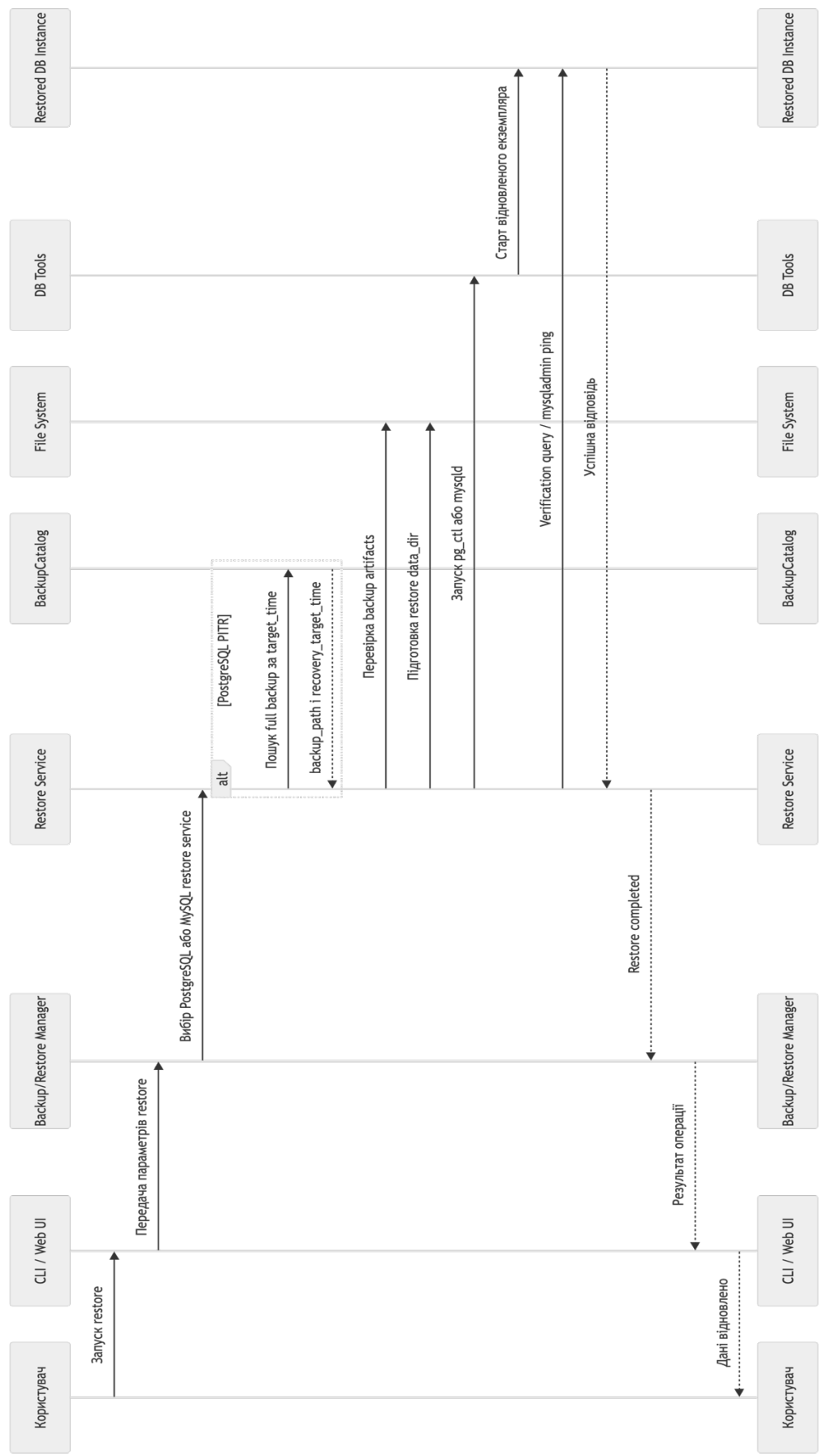


Рис. 2.12 Послідовність взаємодії компонентів під час відновлення бази даних

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

У цьому розділі розглядається практична реалізація розробленої системи резервного копіювання та відновлення баз даних PostgreSQL і MySQL. Основна увага приділяється реалізації модулів створення резервних копій, пайплайну відновлення, роботі з WAL-архівами, формуванню службових метаданих та реалізації інтерфейсів взаємодії з користувачем.

3.1 Реалізація модулів повного та похідного резервного копіювання

Загальна координація операцій виконується через сервісний рівень. Команди користувача обробляються у `CommandDispatcher`, після чого передаються до `BackupService`. Далі система визначає тип СУБД і запускає відповідний клієнт: `PostgresClient` для PostgreSQL або `MysqlClient` для MySQL. Окремо винесені компоненти для роботи з каталогом резервних копій, службовими метаданими та валідацією артефактів.

Основні програмні модулі, які беруть участь у процесі резервного копіювання, наведено в таблиці 3.1.

Таблиця 3.1

Основні програмні модулі системи

Модуль / файл	Призначення
<code>commands/command_dispatcher.py</code>	Маршрутизація команд CLI
<code>services/backup_services.py</code>	Сервісний рівень запуску операцій
<code>clients/postgres_client.py</code>	Повне резервне копіювання PostgreSQL
<code>clients/mysql_client.py</code>	Повне резервне копіювання MySQL
<code>mixins/differential_mixin.py</code>	Вибір стратегії похідного копіювання
<code>postgres_strategy.py</code>	Диференціальне копіювання PostgreSQL через WAL

Основні програмні модулі системи

Модуль / файл	Призначення
mysql_strategy.py	Інкрементальне копіювання MySQL через XtraBackup
custom_logging.py	Старт і завершення backup-операції
contracts.py	Формування metadata.json і validation.json
catalog.py	Запис інформації до SQLite-каталогу

Для створення фізичної копії PostgreSQL використовується стандартна утиліта *pg_basebackup*. Команда формується як список аргументів, а не як один shell-рядок. Це зменшує ризик помилок під час передачі параметрів і робить запуск безпечнішим.

Лістинг 3.1

Формування команди pg_basebackup

```
pg_basebackup_cmd = [
    "pg_basebackup",
    "-h", self._host,
    "-p", str(self._port),
    "-U", self._user,
    "-D", str(backup_dir),
    "-F", "t",
    "-X", "stream",
    "-c", "fast",
    "-P",
    "-v",
    "-l", f"UTIL_FULL_{backup_id}",
    "--manifest-checksums=SHA256",
]

pg_basebackup_cmd.extend(["-z", "-Z", str(self._compressing_level or 6)])
```

Параметр `-F t` задає tar-формат копії, `-X stream` дозволяє передавати WAL-журнали разом із резервною копією, а `--manifest-checksums=SHA256` використовується для створення manifest-файлу з контрольними сумами. У результаті створюються основні артефакти PostgreSQL: `base.tar.gz`, `pg_wal.tar.gz` та, за наявності підтримки з боку версії PostgreSQL, `backup_manifest`.

Після успішного виконання `pg_basebackup` система фіксує службові маркери PostgreSQL. Зокрема, зчитуються поточний LSN та назва WAL-файлу. Ці дані потрібні для подальшого аналізу стану копії та побудови сценаріїв відновлення.

Диференціальне копіювання PostgreSQL реалізовано через окрему стратегію `PostgresDifferentialBackupStrategy`. На відміну від повної копії, у цьому режимі система не копіює весь кластер PostgreSQL повторно, а працює з WAL-сегментами. Спочатку з SQLite-каталогу визначається остання успішна повна копія для поточної бази даних. Після цього система аналізує WAL-архів, визначає нові WAL-файли та формує директорію диференціальної копії.

Перед збереженням WAL-сегментів виконується їх перевірка. Для цього використовується механізм `WalChainValidation`, який перевіряє відповідність `timeline`, відсутність пропусків у послідовності WAL-файлів, наявність файлів на диску, їхній розмір і можливість читання. Це потрібно, щоб у диференціальну копію не потрапив пошкоджений або неповний WAL-файл.

Для MySQL повне резервне копіювання реалізовано у класі `MysqlClient`. Основним інструментом є `Percona XtraBackup`, який дозволяє створювати фізичні копії MySQL без зупинки сервера. Перед запуском операції система перевіряє наявність `xtrabackup` у системі. Якщо утиліта недоступна, операція завершується помилкою.

Лістинг 3.2

Формування команди `xtrabackup`

```
xtrabackup_cmd = [  
    "xtrabackup",  
    "--backup",  
    f"--target-dir={backup_dir}",  
    f"--login-path={self._login_path}",  
    "--compress",  
    "--compress-threads=4",  
]
```

У випадку використання `login-path` параметри доступу беруться з профілю MySQL, створеного через `mysql_config_editor`. Це дозволяє не передавати пароль явно через аргументи командного рядка. Якщо `login-path` не використовується, параметри підключення передаються окремо, а пароль додається через змінну середовища `MYSQL_PWD`.

Після завершення роботи XtraBackup система перевіряє наявність файлу `xtrabackup_checkpoints`. Якщо цей файл відсутній, копія вважається неповною. Також за наявності зчитується `xtrabackup_binlog_info`, що містить інформацію про позицію `binary log`. Ці дані записуються в метадані резервної копії.

Інкрементальне копіювання MySQL у системі реалізовано через стратегію `MySQLDifferentialBackupStrategy`. На рівні CLI воно запускається як похідне копіювання, але технічно використовує механізм `xtrabackup --incremental`. Для цього система визначає останню повну копію через SQLite-каталог і передає її як базову директорію для параметра `--incremental-basedir`.

Окрему роль у реалізації похідного копіювання відіграє підхід `Strategy` (див. підрозділ 2.2). Система має спільний механізм запуску похідної копії, але конкретна реалізація залежить від типу СУБД. Для PostgreSQL використовується стратегія на основі WAL-сегментів, а для MySQL - стратегія на основі інкрементального режиму XtraBackup.

Вибір стратегії похідного резервного копіювання

```
DIFFERENTIAL_BACKUP_STRATEGIES = {
    "mysql": MySQLDifferentialBackupStrategy,
    "postgresql": PostgresDifferentialBackupStrategy,
}

strategy_cls = DIFFERENTIAL_BACKUP_STRATEGIES.get(database_engine)

if strategy_cls is None:
    raise ValueError(
        f"Unsupported database engine for differential backup: {database_engine}"
    )

strategy = strategy_cls(self, self._logger, self._messenger)
diff_service = DifferentialBackupService(
    self,
    self._logger,
    self._messenger,
    strategy,
)
return diff_service.perform_differential_backup(metadata_reader)
```

У лістингу 3.3 показано механізм вибору стратегії похідного резервного копіювання. Для цього використовується словник `DIFFERENTIAL_BACKUP_STRATEGIES`, у якому тип СУБД пов'язується з відповідним класом стратегії. Для PostgreSQL використовується стратегія на основі WAL-сегментів, а для MySQL - стратегія на основі механізму `xtrabackup --incremental`.

3.2 Реалізація пайплайну відновлення та роботи з WAL-архівами

3.2.1 Реалізація PostgreSQL restore та PITR

Реалізація відновлення PostgreSQL винесена в окремий сервіс, який працює з уже створеною фізичною копією. На вхід сервіс отримує шлях до резервної копії, нову директорію даних, шлях до WAL-архіву та порт для запуску тимчасового екземпляра PostgreSQL.

Якщо резервна копія передана у вигляді єдиного архіву .tar.zst, система спочатку розпаковує його у тимчасову директорію. Далі з резервної копії витягуються основні архіви base.tar.gz і pg_wal.tar.gz. base.tar.gz розгортається у нову директорію даних, а pg_wal.tar.gz використовується для відновлення WAL-сегментів, необхідних для запуску PostgreSQL у режимі recovery.

Лістинг 3.4

Формування recovery-конфігурації PostgreSQL

```

auto_conf = self._data_dir / "postgresql.auto.conf"
existing = auto_conf.read_text(encoding="utf-8") if auto_conf.exists() else ""

restore_command = self._build_restore_command()
config_lines = [
    self.CONFIG_BLOCK_START,
    f"restore_command = '{restore_command}'",
    "archive_mode = off",
]

block = "\n".join(config_lines)
new_content = block if not existing else f"{existing}\n\n{block}"
auto_conf.write_text(f"{new_content}\n", encoding="utf-8")

recovery_signal = self._data_dir / "recovery.signal"
recovery_signal.write_text("", encoding="utf-8")

```

У лістингу 3.4 показано фрагмент реалізації, який формує recovery-конфігурацію PostgreSQL. Спочатку система відкриває або створює файл postgresql.auto.conf, після чого додає до нього restore_command. Цей параметр визначає, звідки PostgreSQL має брати WAL-файли під час процедури відновлення. Додатково встановлюється archive_mode = off, оскільки відновлений технічний екземпляр не повинен повторно архівувати WAL-файли.

Після запису конфігурації система створює порожній файл recovery.signal. У PostgreSQL наявність цього файлу в директорії даних означає, що під час запуску сервер має перейти в режим відновлення. Завдяки цьому користувачу не потрібно вручну редагувати конфігураційні файли перед запуском restore-процесу.

Окремою частиною реалізації PostgreSQL restore є робота з WAL-архівами. Для цього у системі використовується pipeline-підхід, у якому обробка WAL-файлів поділена на кілька незалежних етапів. Такий підхід дозволяє ізолювати перевірку стабільності файлу, його безпечне копіювання, перевірку цілісності та журналювання результату.



Рис. 3.1 Структура pipeline обробки WAL-файлів PostgreSQL

Окрім звичайного відновлення з явно заданої повної копії, у системі реалізовано PITR-сценарій для PostgreSQL. У цьому режимі користувач передає цільовий момент часу відновлення через параметр `--target-time`. Якщо шлях до повної копії не задано явно, система звертається до SQLite-каталогу та вибирає останню успішну повну резервну копію, час завершення якої не перевищує заданий `target-time`. Після цього під час формування `recovery`-конфігурації додатково записуються параметри `recovery_target_time`, `recovery_target_action` = 'promote' та `recovery_target_timeline` = 'latest'. Це дозволяє PostgreSQL застосувати WAL-файли лише до заданого моменту часу.

3.2.2 Реалізація MySQL restore pipeline

MySQL restore реалізовано як послідовний pipeline у сервісі MySQLRestoreService. На відміну від PostgreSQL restore, цей сценарій спирається на етапи Persona XtraBackup: підготовку копії, перенесення файлів у нову директорію даних, запуск окремого екземпляра `mysqld` та перевірку його доступності.

Основні етапи MySQL restore pipeline

```
def restore(self) -> bool:
    started = False
    try:
        self._validate_inputs()
        self._decompress_zstd_artifacts()
        self._validate_backup_directory(self._backup_path)
        self._prepare_backup()
        self._prepare_data_directory()
        self._copy_back()
        self._set_data_directory_owner()
        self._start_mysql()
        started = True
        self._verify_mysql()

        self._message_success("MySQL restore pipeline completed successfully")
    return True
```

Лістинг 3.5 демонструє, що MySQL restore pipeline реалізовано як послідовність окремих етапів, винесених у незалежні методи сервісу MySQLRestoreService.

Окреме винесення етапів у методи `_prepare_backup()`, `_copy_back()` та `_verify_mysql()` спрощує обробку помилок і дозволяє зупиняти pipeline на конкретному етапі без виконання наступних дій. Це особливо важливо для restore-сценаріїв, оскільки некоректне продовження процесу після помилки може призвести до запуску неконсистентної директорії даних.

3.3 Формування та валідація артефактів

У системі формування службових артефактів винесене в окремий pipeline, який запускається після завершення backup-операції. На відміну від безпосереднього створення фізичної копії, цей етап відповідає за нормалізацію службових даних, перевірку артефактів, формування фінального статусу та запис результату до SQLite-каталогу.

Формування metadata.json і validation.json

```
def persist_backup_documents(self, metadata, output_dir, backup_succeeded):
    output_dir = Path(output_dir)
    output_dir.mkdir(parents=True, exist_ok=True)
    metadata_doc = self._build_metadata_document(metadata, output_dir)
    validation_doc = self._build_validation_document(metadata_doc, output_dir)
    validation_doc["status"] = (
        "passed" if backup_succeeded and validation_doc["status"] == "passed" else
        "failed"
    )

    validation_summary = ValidationSummary(
        state=validation_doc["status"],
        report_path=str(output_dir / "validation.json"),
        metadata_schema_valid=validation_doc["metadata_schema"]["valid"],
        manifest_present=validation_doc["manifest"]["exists"],
        checkpoints_present=validation_doc["checkpoints"]["exists"],
        recovery_status=validation_doc["recovery_status"]["state"],
    )

    metadata_doc["validation_status"] = asdict(validation_summary)
    metadata_doc["status"] = (
        "completed" if backup_succeeded and validation_summary.state == "passed"
        else "failed"
    )
    self._write_json(output_dir / "metadata.json", metadata_doc)
    self._write_json(output_dir / "validation.json", validation_doc)
```

У лістингу 3.6 показано фрагмент методу `persist_backup_documents()`, який відповідає за фінальне формування службових JSON-файлів. Спочатку система створює або перевіряє директорію резервної копії, після чого окремо формує `metadata_doc` і `validation_doc`. Це дозволяє розділити технічний опис копії та результат її перевірки.

Важливо, що фінальний статус `validation.json` залежить не тільки від результату зовнішньої утиліти резервного копіювання, а й від внутрішньої перевірки `validation_doc`. Якщо `backup_succeeded` має значення `False` або `validation_doc["status"]` не дорівнює `passed`, підсумковий статус `validation.json` встановлюється як `failed`.

Після цього формується `ValidationSummary`, який стисло переноситься до `metadata.json` у поле `validation_status`. Завдяки цьому `metadata.json` містить не лише

технічний опис копії, а й короткий підсумок результату перевірки. Фінальний status копії встановлюється як completed тільки тоді, коли операція резервного копіювання завершилася успішно і validation pipeline також пройшов успішно.

Лістинг 3.7

Передача фінальних метаданих до SQLite-каталогу

```
contract_metadata = self._contract_manager.persist_backup_documents(...)
metadata.clear()
metadata.update(contract_metadata)
self.catalog.add_backup(metadata)
```

Після формування службових JSON-файлів фінальні метадані повертаються до BackupLogger. Далі початковий словник metadata очищується й замінюється нормалізованим contract_metadata. Саме ці дані передаються до BackupCatalog і записуються в SQLite-каталог. Завдяки цьому каталог зберігає вже не проміжний стан операції, а фінальний результат з урахуванням валідації.

Окрім автоматичного формування validation.json після backup-операції, у системі реалізовано окремий сценарій перевірки вже створеної резервної копії. Для цього використовується команда: `validate backup-path <backup_directory>`.

Вона дозволяє повторно перевірити структуру backup-директорії, наявність обов'язкових артефактів, службові метадані та базові ознаки придатності копії до відновлення без повторного створення резервної копії.

3.4 Реалізація графічного (Web UI) та консольного (CLI) інтерфейсів

Основним інтерфейсом взаємодії з системою є консольний інтерфейс командного рядка. Через CLI користувач може запускати створення резервних копій, відновлення, перевірку артефактів та перегляд службової інформації. Реалізація CLI побудована таким чином, щоб інтерфейс не містив бізнес-логіки, а лише передавав параметри до сервісного рівня системи.

Для реалізації інтерактивного режиму в системі використовується бібліотека `prompt_toolkit`. Вона забезпечує не лише введення команд, а й підтримку розширених

можливостей, зокрема автодоповнення, історії введення та динамічного prompt.

```
=====
=====
2026-05-11 16:43:49 - backup_BackUp - INFO -
=====
Storage: LOCAL
2026-05-11 16:43:49 - backup_BackUp - INFO - Storage: LOCAL
=====
2026-05-11 16:43:49 - backup_BackUp - INFO - =====

2026-05-11 16:43:49 - backup_BackUp - INFO -
Database Backup Utility
2026-05-11 16:43:49 - backup_BackUp - INFO - Database Backup Utility
Connected to: BackUp as artempavlenko
Type 'help' for commands or 'exit' to quit

[[BackUp | WAL:1B | +4]> restore
full database
differential backup
restore
restore database
SQL
-path
-backup-path
```

Рис. 3.2 Робота інтерактивної консолі CLI: prompt, WAL-стан і автодоповнення команд

Для полегшення роботи користувача в інтерактивному CLI реалізовано вбудовану довідку. Вона викликається командою *help* і дозволяє швидко переглянути перелік доступних операцій, їх синтаксис, приклади використання та короткий опис призначення.

```
[BackUp | WAL:1B | +4]> help

=====
2026-05-11 16:47:55 - backup_BackUp - INFO - 
=====
Database Backup Utility
2026-05-11 16:47:55 - backup_BackUp - INFO - Database Backup Utility
=====
2026-05-11 16:47:55 - backup_BackUp - INFO - =====
✓ Interactive console commands:
2026-05-11 16:47:55 - backup_BackUp - INFO - ✓ Interactive console commands:

1) Full database backup
2026-05-11 16:47:55 - backup_BackUp - INFO - 1) Full database backup
Command:
    full database -path <destination_path> [-single-archive true|false] [--validate-restore true|false]
Examples:
    full database -path /backups/postgres
    full database -path /backups/postgres --single-archive false
    full database -path /backups/postgres --validate-restore true
Details:
    - Creates a physical full backup for the connected database.
    - PostgreSQL uses pg_basebackup.
    - MySQL uses xtrabackup.
    - By default, also creates a single .tar.zst archive for easier storage/upload.
    - PostgreSQL full backups run an automated temporary restore validation by default.

2) Differential backup
2026-05-11 16:47:55 - backup_BackUp - INFO - 2) Differential backup
Command:
    differential backup
Example:
```

Рис. 3.3 Виведення довідки про команди в інтерактивній консолі CLI

Інтерактивний CLI реалізовано не як набір окремих input-викликів, а як окремий режим роботи утиліти, у якому користувач після підключення до СУБД може виконувати команди без повторного введення параметрів з'єднання. Для цього в пам'яті зберігається контекст активного підключення: тип СУБД, назва бази даних, користувач, а також службова інформація про поточний стан середовища, зокрема WAL-статус для PostgreSQL.

Лістинг 3.8

Приклад запуску backup-команди через CLI

```
python cli/dbtool.py backup \
--db postgres \
--database demo_db \
--storage local \
--config profile
```

Бібліотека `prompt_toolkit` використовується для побудови розширеного термінального інтерфейсу. Вона забезпечує роботу автодоповнення, історії введення, підсвічування активного рядка та формування динамічного `prompt`. Завдяки цьому користувач може вводити не лише повні команди, а й частини команд або параметрів, після чого система пропонує можливі варіанти продовження. Це зменшує кількість помилок при ручному введенні команд та пришвидшує виконання типових операцій.

Динамічний `prompt` додатково виконує роль індикатора поточного стану системи. У ньому відображається активна база даних, стан WAL-архівування та інші короткі службові маркери. Такий підхід дозволяє користувачу одразу бачити, у якому контексті буде виконана команда, що особливо важливо для операцій резервного копіювання та відновлення, де помилка у виборі бази або режиму може мати критичні наслідки.

Архітектурно CLI не виконує операції резервного копіювання або відновлення напряду. Його завдання полягає у зчитуванні команди, розборі аргументів, базовій перевірці параметрів та передачі керування відповідному сервісу. Наприклад, команда створення повної копії передається до сервісів резервного копіювання, команда похідного копіювання - до `DifferentialBackupService` (див. розділ 2.2), а команда `restore` - до `restore`-сервісу відповідної СУБД (див. розділ 2.4).

Це відповідає принципу розділення відповідальностей: CLI відповідає за взаємодію з користувачем, а бізнес-логіка зосереджена в окремих сервісах. Завдяки цьому одна й та сама логіка може бути використана як з інтерактивної консолі, так і з Web UI або звичайного запуску через аргументи командного рядка.

Окрім консольного інтерфейсу, у системі реалізовано графічний Web UI, який використовується як допоміжний інтерфейс для перегляду службової інформації про резервні копії. На відміну від інтерактивного CLI, Web UI орієнтований не на швидке введення адміністративних команд, а на зручне представлення результатів роботи системи.

Основне призначення Web UI полягає у відображенні історії резервних копій, їхнього поточного статусу, типу СУБД, типу операції, часу створення, шляхів до артефактів, а також пов'язаних файлів `metadata.json` і `validation.json`.

Web UI працює поверх того самого сервісного рівня та SQLite-каталогу, що й CLI. Завдяки цьому графічний інтерфейс не дублює бізнес-логіку, а лише відображає вже сформовані дані системи та передає запити до спільних сервісів.

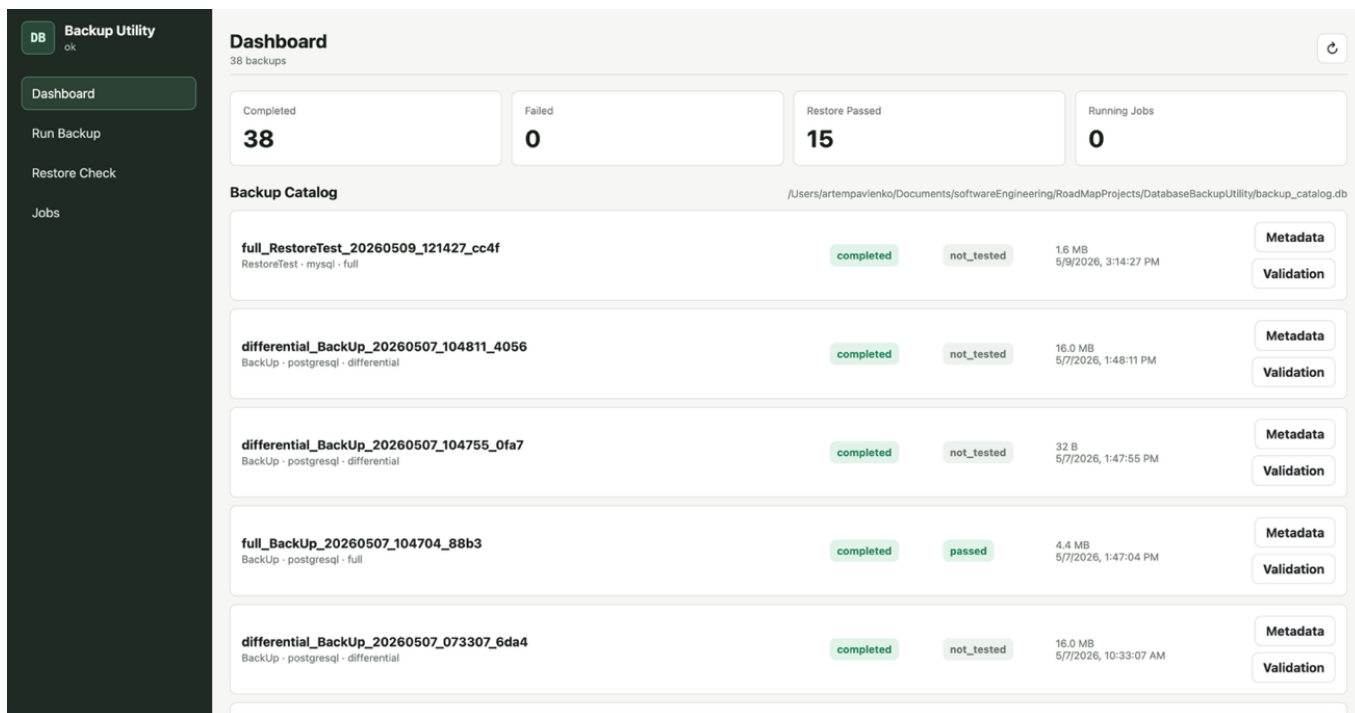


Рис. 3.4 Графічний Web UI для перегляду історії резервних копій та стану артефактів

Тому CLI і Web UI виконують різні, але взаємодоповнювальні ролі. CLI забезпечує точне керування операціями резервного копіювання та відновлення, а Web UI надає зручний візуальний доступ до історії операцій, статусів і службових артефактів.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі проведено перевірку працездатності розробленого програмного забезпечення для резервного копіювання та відновлення баз даних PostgreSQL і MySQL. Основну увагу приділено перевірці цілісності створених резервних копій, коректності формування службових артефактів, можливості відновлення бази даних після втрати даних, а також оцінці продуктивності та безпеки роботи з профілями доступу.

4.1 Автоматизовані unit-тести

Тестування програмного забезпечення виконувалося на двох рівнях. Перший рівень становлять автоматизовані unit-тести, розміщені в директорії tests. Вони перевіряють окремі програмні компоненти: каталог резервних копій, contract validation layer, restore-сервіси, PITR-планування та маршрутизацію сервісів.

Автоматизовані тести реалізовано з використанням стандартного модуля unittest.

Таблиця 4.1

Автоматизовані тестові модулі проєкту

Для запуску всього набору тестів використовується команда:		
<code>python -m unittest discover -s tests</code>		
Для запуску окремого тестового модуля використовується команда:		
<code>python -m unittest tests.test_postgres_restore</code>		
№	Тестовий модуль	Що перевіряє
1	test_backup_services_restore.py	Маршрутизацію backup/restore сервісів і заборону PostgreSQL restore для MySQL-клієнта

Автоматизовані тестові модулі проєкту

№	Тестовий модуль	Що перевіряє
2	test_backup_catalog_sqlite.py	Імпорт legacy-каталогу, фільтрацію копій, побудову backup chain і вибір full backup для PITR
3	test_backup_contracts.py	Створення metadata.json, validation.json, checksum entries і failed state при відсутності артефактів
4	test_postgres_pitr.py	Планування PITR і нормалізацію часу до UTC
5	test_postgres_restore.py	Генерацію restore-конфігурації, recovery.signal, PITR-параметри та перевірку tablespace-архівів
6	test_mysql_restore.py	Валідацію XtraBackup-артефактів, .zst decompression і restore pipeline MySQL
7	test_postgres_restore_validation.py	Порівняння source/restored table signatures і формування validation report

Варто зазначити, що автоматизовані тести мають переважно unit-рівень. Вони не запускають повний end-to-end сценарій з реальною базою даних через CLI або Web UI, а ізолюють окремі компоненти за допомогою тимчасових директорій та mock-об'єктів. Тому для повної перевірки системи додатково проведено ручні інтеграційні сценарії з фактичним створенням і відновленням резервних копій.

4.2 Тестування механізмів резервного копіювання і відновлення

Другий рівень становлять ручні інтеграційні сценарії, під час яких перевіряється фактичне створення резервних копій, наявність артефактів, відновлення бази даних та робота з профілями доступу.

Таблиця 4.2

План тестування програмного забезпечення

№	Сценарій	Мета тестування	Очікуваний результат
1	Створення повної резервної копії PostgreSQL	Перевірити формування основних артефактів фізичної копії	Створено base.tar.gz, pg_wal.tar.gz, metadata.json, validation.json
2	Створення повної резервної копії MySQL	Перевірити роботу XtraBackup та формування службових файлів	Створено каталог копії з XtraBackup-артефактами та метаданими
3	Створення похідної PostgreSQL-копії	Перевірити побудову backup chain на основі WAL-сегментів	У SQLite-каталозі зафіксовано зв'язок з повною копією
4	Створення похідної MySQL-копії	Перевірити роботу xtrabackup -incremental	Створено інкрементальну копію з посиланням на базову
5	Валідація пошкодженої копії	Перевірити реакцію системи на відсутній артефакт	Валідація завершується зі статусом failed
6	Відновлення PostgreSQL	Перевірити можливість запуску БД з резервної копії та WAL-архіву	Відновлений інстанс запускається, SELECT 1 виконується успішно

План тестування програмного забезпечення

7	Відновлення MySQL	Перевірити відновлення бази після видалення	Дані доступні після запуску відновленого MySQL-інстансу
8	Негативний restore-сценарій	Перевірити обробку неправильного шляху або зайнятого порту	Restore завершується помилкою без створення неконсистентного стану
9	Оцінка продуктивності	Виміряти час виконання backup/restore і розмір результату	Отримано фактичні значення часу та розміру копій
10	Перевірка безпечної автентифікації	Перевірити роботу .pgpass та login-path	Паролі не передаються явно через CLI

4.2.1 Тестування створення повної резервної копії PostgreSQL

Під час першого інтеграційного сценарію було перевірено створення повної фізичної резервної копії PostgreSQL. Метою тесту була перевірка коректності формування backup-артефактів, службових метаданих та автоматизованої restore validation.

Для тестування використовувалася база даних *backuptest*. Запуск утиліти здійснювався через CLI у режимі profile, де автентифікація виконувалася через файл *~/.pgpass* без передачі пароля через аргументи командного рядка.

Перед запуском резервного копіювання система автоматично перевірила доступність PostgreSQL, коректність WAL archive directory, параметри *archive_mode* та *wal_level*, а також наявність replication-привілеїв у користувача. Після успішної перевірки було запущено *pg_basebackup*.

У результаті роботи було створено директорію резервної копії, що містила основні артефакти фізичного backup PostgreSQL: *base.tar.gz*, *pg_wal.tar.gz*, *backup_manifest*, *metadata.json* та *validation.json*. Додатково система сформувала єдиний стиснений архів *.tar.zst*.

Після завершення backup система автоматично виконала restore validation: резервна копія була розгорнута у тимчасовій директорії, створено файл recovery.signal, налаштовано postgresql.auto.conf, після чого тимчасовий PostgreSQL-інстанс був успішно запущений. Перевірочний SQL-запит SELECT 1 завершився без помилок.

Файл metadata.json містив службову інформацію про backup, LSN-маркери та шляхи до артефактів, а validation.json підтвердив наявність усіх обов'язкових файлів і завершення перевірки зі статусом passed.

Тест підтвердив працездатність механізму створення повної фізичної резервної копії PostgreSQL та базової автоматизованої перевірки відновлення.

Таблиця 4.3

Результат тестування створення повної резервної копії PostgreSQL

Параметр	Значення
СУБД	PostgreSQL
База даних	backuptest
Тип копії	Повна фізична резервна копія
Метод створення	pg_basebackup
Спосіб автентифікації	.pgpass
Backup ID	full_backuptest_20260511_150830_f3e3
Директорія копії	/backups/full_backuptest_20260511_150830_f3e3
Основні артефакти	base.tar.gz, pg_wal.tar.gz, backup_manifest, metadata.json, validation.json
Розмір копії	4.42 MB
Розмір .tar.zst архіву	4.01 MB
Економія після стиснення	9.2%
Тривалість backup	1.38 с

Результат тестування створення повної резервної копії PostgreSQL

Параметр	Значення
Restore validation	Passed
Validation status	Passed
Результат тесту	Успішно

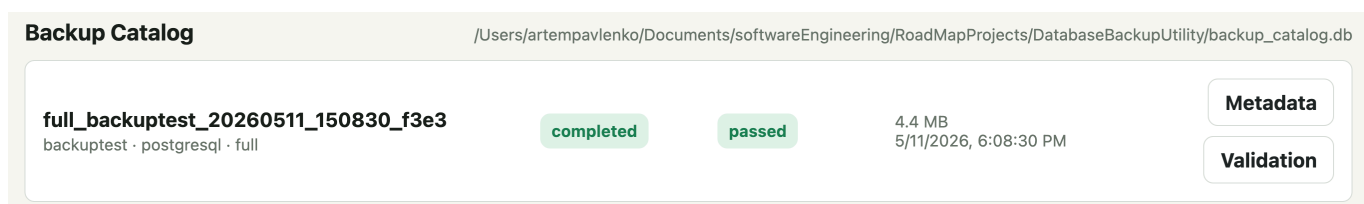


Рис. 4.1 Демонстрація успішного створення повної резервної копії бази даних через WEB UI

4.2.2 Тестування створення похідної резервної копії PostgreSQL

Після створення повної резервної копії було проведено тестування механізму похідного резервного копіювання PostgreSQL на основі WAL-архівації. Метою тесту була перевірка побудови backup chain, визначення нових WAL-сегментів та формування службових метаданих для похідної копії.

Перед запуском сценарію до бази даних backuptest були внесені зміни, що призвело до генерації нових WAL-записів. Після цього через CLI-інтерфейс було запущено differential backup.

Під час виконання тесту система автоматично визначила останню повну резервну копію full_backuptest_20260511_150830_f3e3, зчитала інформацію про попередній WAL-файл та виконала перевірку archive directory. Після примусового WAL switch система виявила новий WAL-сегмент 00000001000000010000001D і скопіювала його до директорії похідної резервної копії.

У результаті було створено differential backup:

differential_backuptest_20260511_152020_293e

Розмір похідної копії становив 16 MB. У файлі metadata.json були сформовані поля parent_backup_id та base_backup_id, що підтверджує побудову backup chain відносно повної резервної копії. Файл validation.json підтвердив коректність структури метаданих і наявність необхідних артефактів.

Додатково було перевірено сценарій, коли після повної резервної копії не з'явилися нові WAL-файли. У цьому випадку система коректно завершила роботу з повідомленням *No new WAL files in archive since last full backup*, не створюючи неконсистентну резервну копію.

Таким чином, тест підтвердив працездатність механізму differential backup PostgreSQL та підтримку WAL-based backup chain.

Таблиця 4.4

Результат тестування створення похідної (диференційної) резервної копії PostgreSQL

Параметр	Значення
СУБД	PostgreSQL
Тип копії	Differential backup
Базова копія	full_backuptest_20260511_150830_f3e3
WAL archive	Використовується
Нові WAL-файли	1
WAL range	000000001000000010000001D
Differential backup size	16 MB
Parent backup ID	Присутній
Base backup ID	Присутній
Validation status	Passed
Результат тесту	Успішно

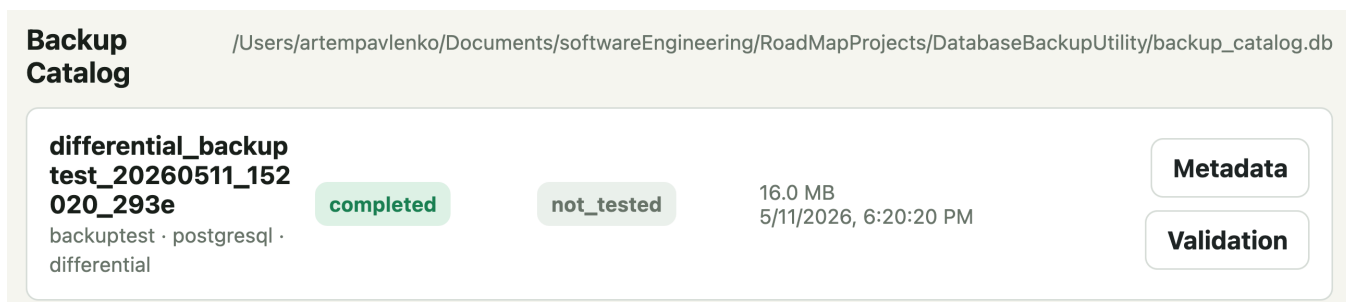


Рис. 4.2 Демонстрація успішного створення похідної резервної копії бази даних через WE BUI.

4.2.3 Тестування створення повної резервної копії MySQL

Під час наступного сценарію було перевірено створення повної фізичної резервної копії MySQL. Метою тесту була перевірка роботи backup pipeline на основі Persona XtraBackup, формування директорії резервної копії та створення стисненого архіву.

Для тестування використовувалася база даних backup_mysql_test. Автентифікація виконувалася через mysql_config_editor із використанням login-path xtrabackup, що дозволило не передавати пароль безпосередньо через CLI. Після запуску утиліта визначила MySQL socket /tmp/mysql.sock та успішно встановила підключення.

Резервну копію було створено в директорії /tmp/mysql_backups/full_backup_mysql_test_20260512_100109_07d9. Початковий розмір backup становив 1.60 MB. Додатково система сформувала єдиний архів full_backup_mysql_test_20260512_100109_07d9.tar.zst, розмір якого становив 1.59 MB. Загальний час виконання операції склав 3.05 секунди.

Таким чином, тест підтвердив працездатність механізму створення повної фізичної резервної копії MySQL через xtrabackup.

Результат тестування повної резервної копії MySQL

Параметр	Значення
СУБД	MySQL
База даних	backup_mysql_test
Тип копії	Full backup
Backup engine	Percona XtraBackup
Спосіб автентифікації	login-path
Login-path	xtrabackup
Socket	/tmp/mysql.sock
Директорія backup	/tmp/mysql_backups/ full_backup_mysql_test_20260512_100109_07d9
Розмір backup	1.60 MB
Розмір .tar.zst	1.59 MB
Економія після стиснення	0.6%
Час виконання	3.05 с
Результат тесту	Успішно

4.2.4 Тестування інкрементального резервного копіювання MySQL

Після створення повної резервної копії MySQL було перевірено створення похідної копії. На рівні CLI сценарій запускається як differential backup, однак для MySQL він реалізований через механізм xtrabackup --incremental.

Під час запуску система автоматично визначила останню повну резервну копію full_backup_mysql_test_20260512_100109_07d9 та використала її як базову директорію для інкрементального копіювання. У команді xtrabackup було передано параметр --incremental-basedir, що підтверджує створення саме похідної копії відносно попереднього full backup.

У результаті було створено резервну копію:

/tmp/mysql_backups/differential_backup_mysql_test_20260512_130935_c5c2

Розмір похідної копії становив 0.15 MB, а час виконання - 2.85 секунди. Це підтверджує, що система не створювала повторну повну копію, а зберегла лише зміни відносно базової резервної копії.

Таким чином, тест підтвердив працездатність механізму інкрементального резервного копіювання MySQL на основі Percona XtraBackup.

Таблиця 4.6

Результат тестування інкрементального резервного копіювання MySQL

Параметр	Значення
СУБД	MySQL
CLI-сценарій	differential backup
Фактичний механізм	xtrabackup --incremental
Базова копія	full_backup_mysql_test_20260512_100109_07d9
Параметр XtraBackup	--incremental-basedir
Директорія копії	/tmp/mysql_backups/ differential_backup_mysql_test_20260512_130935_c5c2
Розмір копії	0.15 MB
Час виконання	2.85 с
Результат тесту	Успішно

4.2.5 Негативне тестування пошкоджених або неповних артефактів

Для перевірки реакції системи на неповну резервну копію було проведено негативний тест. У вже створеній повній PostgreSQL-копії було тимчасово перейменовано два обов'язкові артефакти: base.tar.gz та pg_wal.tar.gz.

Запущена команда повторної перевірки:

```
validate backup -path /backups/full_backuptest_20260511_150830_f3e3
```

Система визначила відсутність обов'язкових артефактів і завершила перевірку зі статусом Validation failed. У повідомленні було вказано відсутні елементи

base_tar_path та wal_tar_path, що відповідають файлам base.tar.gz і pg_wal.tar.gz.

```
[backuptest | WAL:1F | +0]> validate backup -path /Users/artempavlenko/Documents
/backups/full_backuptest_20260511_150830_f3e3
Validation failed
Missing required artifacts:
  - base_tar_path
  - wal_tar_path
[backuptest | WAL:1F | +0]> █
```

Рис. 4.3 Результати валідації резервної копії

Таблиця 4.7

Результат негативного тестування пошкоджених артефактів

Параметр	Значення
Тип тесту	Негативний
СУБД	PostgreSQL
Backup	Full backup
Пошкоджені артефакти	base.tar.gz, pg_wal.tar.gz
Дія	Тимчасове перейменування файлів
Команда перевірки	validate backup -path ...
Фактичний результат	Validation failed
Виявлені відсутні артефакти	base_tar_path, wal_tar_path
Результат тесту	Успішно

4.2.6 Тестування сценарію відновлення PostgreSQL

Для перевірки сценарію відновлення PostgreSQL було створено тестову таблицю test_backup_full із трьома записами, після чого виконано повне резервне копіювання та ручний запуск restore pipeline у нову директорію даних.

Відновлення виконувалося з повної резервної копії у директорію /tmp/pg_restore_manual_test_2/data з запуском окремого PostgreSQL-інстансу на порту 55433. Під час restore система розпакувала base.tar.gz та pg_wal.tar.gz, оновила postgresql.auto.conf, додала локальні правила доступу до pg_hba.conf, створила файл recovery.signal та запустила відновлений інстанс.

Таблиця 4.8

Результат тестування сценарію відновлення PostgreSQL

Параметр	Значення
СУБД	PostgreSQL
Тип сценарію	Restore з full backup
Тестова таблиця	test_backup_full
Кількість записів до backup	3
Restore data directory	/tmp/pg_restore_manual_test_2/data
Restore port	55433
recovery.signal	Створено
Перевірка таблиці	Таблиця знайдена
Кількість записів після restore	3
Обмеження перевірки	Автоматичний SELECT 1 не пройшов через відсутність ролі postgres
Результат тесту	Успішно

Автоматична перевірка SELECT 1 спочатку завершилася попередженням через відсутність ролі postgres у відновленому кластері. Після цього перевірку було виконано вручну з використанням ролі artempavlenko. Запит до списку таблиць підтвердив наявність таблиці test_backup_full, а запит SELECT COUNT(*) повернув 3 записи.

```

.....
✓ PostgreSQL restore pipeline completed successfully
2026-05-12 13:38:25 - restore_backuptest - INFO - ✓ PostgreSQL restore pipeline
completed successfully
2026-05-12 13:38:25 - restore_backuptest - INFO - PostgreSQL restore pipeline co
mpleted successfully
Restored data directory: /private/tmp/pg_restore_manual_test_2/data
2026-05-12 13:38:25 - restore_backuptest - INFO - Restored data directory: /priv
ate/tmp/pg_restore_manual_test_2/data
2026-05-12 13:38:25 - restore_backuptest - INFO - Restored data directory: /priv
ate/tmp/pg_restore_manual_test_2/data
Restore port: 55433
2026-05-12 13:38:25 - restore_backuptest - INFO - Restore port: 55433
2026-05-12 13:38:25 - restore_backuptest - INFO - Restore port: 55433
Stop command: pg_ctl -D /private/tmp/pg_restore_manual_test_2/data stop
2026-05-12 13:38:25 - restore_backuptest - INFO - Stop command: pg_ctl -D /priva
te/tmp/pg_restore_manual_test_2/data stop
2026-05-12 13:38:25 - restore_backuptest - INFO - Stop command: pg_ctl -D /priva
te/tmp/pg_restore_manual_test_2/data stop
[ (.venv) artempavlenko@MacBook-Air-Artem DatabaseBackupUtility % psql -h 127.0.0.]
1 -p 55433 -U artempavlenko -d backuptest -c "\dt"
]
      List of relations
Schema |      Name      | Type  | Owner
-----+-----+-----+-----

```

Рис. 4.4 Результат відновлення PostgreSQL-класстера та перевірка наявності таблиць після restore

4.2.7 Тестування сценарію відновлення MySQL

Для перевірки сценарію відновлення MySQL було використано повну резервну копію, створену за допомогою Percona XtraBackup. Restore виконувався у нову директорію /tmp/mysql_restore_test/data із запуском окремого MySQL-інстансу на порту 3308.

Під час restore система автоматично виконала розпакування .zst-артефактів (xtrabackup_info, xtrabackup_logfile, xtrabackup_tablespace), після чого було запущено xtrabackup --prepare та xtrabackup --copy-back. Далі система запустила відновлений MySQL-інстанс і виконала перевірку через mysqladmin ping.

Після завершення restore було вручну перевірено наявність бази даних backup_mysql_test, таблиці users та кількості записів у таблиці. Запит SELECT COUNT(*) FROM users повернув 2 записи, що підтвердило успішне відновлення.

Результат тестування сценарію відновлення MySQL

Параметр	Значення
СУБД	MySQL
Тип сценарію	Restore з full backup
Restore data directory	/tmp/mysql_restore_test/data
Restore port	3308
Restore socket	/private/tmp/mysql_restore.sock
xtrabackup --prepare	Виконано
xtrabackup --copy-back	Виконано
Автоматична декомпресія .zst	Виконано
Перевірка mysqladmin ping	Успішно
Таблиця users після restore	Присутня
Кількість записів після restore	2
Результат тесту	Успішно

4.3 Оцінка продуктивності та безпеки роботи системи

4.3.1 Аналіз часу створення та відновлення резервних копій

Під час тестування було проведено оцінку особливостей фізичного та логічного резервного копіювання PostgreSQL і MySQL. Основна увага приділялася відмінностям між physical backup-сценаріями (pg_basebackup, Percona XtraBackup) та logical backup-сценаріями (pg_dump, mysqldump).

Фізичне резервне копіювання працює безпосередньо з файлами даних СУБД і WAL/redo-журналами. Завдяки цьому під час restore не потрібно повторно виконувати SQL-команди, створювати індекси та повторно формувати структуру бази даних. Натомість логічне відновлення виконується через SQL-рівень, що збільшує тривалість restore для великих обсягів даних.

Таблиця 4.10

Порівняння часу logical та physical backup/restore за матеріалами експериментального дослідження*

Операція	Час виконання (год)	МБ/с	ГБ/год
Логічне резервне копіювання	7.4	7.2	25.0
Логічне відновлення	8.1	6.3	22.8
Фізичне резервне копіювання	6.2	8.6	30.3
Фізичне відновлення	5.9	8.9	32.0

Наведені результати показують загальну перевагу фізичного підходу над логічним у сценаріях резервного копіювання та відновлення. У контексті СУБД ця різниця пояснюється тим, що фізичні backup-сценарії працюють ближче до файлового рівня збереження даних, тоді як логічне відновлення потребує повторного відтворення структури даних і операцій запису на логічному рівні.

Отже, результати власного тестування підтвердили працездатність реалізованих backup/restore-сценаріїв, а зовнішнє експериментальне дослідження додатково підтверджує доцільність використання фізичних методів резервного копіювання для сценаріїв, де важливим є зменшення часу відновлення.

* Джерело: адаптовано за Hutchinson N.C., Manley S., Federwisch M. та ін. "Logical vs. physical file system backup".

ВИСНОВКИ

У кваліфікаційній роботі було розроблено програмне забезпечення для резервного копіювання та відновлення баз даних PostgreSQL і MySQL. Основною метою роботи було спрощення процесу створення фізичних резервних копій, ведення історії бекуп-операцій, формування службових метаданих, перевірки артефактів та виконання базових сценаріїв відновлення.

У першому розділі було проаналізовано предметну галузь резервного копіювання баз даних, розглянуто відмінності між логічним і фізичним підходами, а також визначено роль показників RPO та RTO для оцінки надійності систем відновлення. Було встановлено, що фізичне резервне копіювання є доцільним для сценаріїв, де важливими є швидкість відновлення, можливість роботи з журналами транзакцій та зменшення часу простою системи. Також було розглянуто існуючі інструменти резервного копіювання, зокрема pg_basebackup, pgBackRest, Barman, Percona XtraBackup та mysqldump. Аналіз показав, що більшість рішень орієнтовані на окрему СУБД або потребують складнішої конфігурації, тому виникає потреба у створенні уніфікованої утиліти для роботи з PostgreSQL і MySQL.

У другому розділі було спроектовано архітектуру системи. У межах проєктування було визначено основні компоненти програмного забезпечення: клієнти PostgreSQL і MySQL, сервісний рівень, SQLite-каталог резервних копій, модуль формування metadata.json і validation.json, підсистему валідації, restore-сервіси, CLI та Web UI. Було обґрунтовано використання мови Python, SQLite, pg_basebackup, Percona XtraBackup, WAL-архівзації PostgreSQL та профілів доступу .pgpass і login-path. Окрему увагу було приділено проєктуванню PostgreSQL restore та PITR-сценарію, у якому використовується повна фізична копія, WAL-архів, recovery-конфігурація та цільовий момент часу відновлення.

У третьому розділі було описано програмну реалізацію системи. Для PostgreSQL реалізовано створення повної фізичної резервної копії за допомогою `pg_basebackup` із потоковою передачею WAL-журналів. Для похідного копіювання PostgreSQL реалізовано механізм роботи з WAL-сегментами та перевіркою їхньої послідовності. Для MySQL реалізовано повне резервне копіювання за допомогою `Percona XtraBackup`, а похідне копіювання побудовано на механізмі `xtrabackup --incremental`. Для вибору конкретного сценарію похідного копіювання використано підхід, подібний до `Strategy`, де реалізація залежить від типу СУБД.

Також у роботі реалізовано пайплайни відновлення для PostgreSQL і MySQL. PostgreSQL `restore` передбачає розпакування `base.tar.gz` і `pg_wal.tar.gz`, формування `recovery`-конфігурації, створення `recovery.signal`, запуск окремого PostgreSQL-інстансу та перевірку працездатності. Для PostgreSQL PITR реалізовано використання `target-time`, вибір відповідної повної копії з SQLite-каталогу та запис параметрів `recovery_target_time`, `recovery_target_action` і `recovery_target_timeline`. MySQL `restore` реалізовано як послідовність етапів `XtraBackup`: перевірка артефактів, підготовка копії, `copy-back`, запуск окремого `mysqld`-інстансу та перевірка доступності через `mysqladmin ping`.

Окремим результатом роботи є реалізація службового контракту резервної копії. Для кожної `backup`-операції система формує `metadata.json` і `validation.json`, у яких зберігаються технічні параметри копії, шляхи до артефактів, маркери LSN або `binlog`, інформація про перевірку та фінальний статус. Крім автоматичного формування `validation.json` після `backup`-операції, у системі реалізовано окремий сценарій валідації вже створеної копії через команду `validate backup-path`. Це дозволяє повторно перевіряти `backup`-директорію без повторного створення резервної копії.

У четвертому розділі було проведено тестування розробленого програмного забезпечення. Автоматизовані `unit`-тести перевіряли роботу SQLite-каталогу, `contract validation layer`, `restore`-сервісів, PITR-планування, генерації `recovery`-конфігурації та порівняння `source/restored table signatures`. Додатково було проведено ручні

інтеграційні сценарії, у межах яких перевірено створення повної резервної копії PostgreSQL, створення похідної PostgreSQL-копії на основі WAL, створення повної та інкрементальної резервної копії MySQL, негативну валідацію пошкоджених артефактів, PostgreSQL restore та MySQL restore. Тестування підтвердило, що система коректно створює резервні копії, формує службові файли, фіксує інформацію в SQLite-каталозі та здатна виконувати базові сценарії відновлення.

Проведена оцінка продуктивності показала, що фізичні backup-сценарії є доцільними для задач, де важливим є зменшення часу відновлення. Крім власних інтеграційних тестів, у роботі було використано результати зовнішнього експериментального дослідження, яке демонструє перевагу фізичного підходу над логічним за часом виконання та пропускну здатністю. Це додатково підтверджує обґрунтованість вибору pg_basebackup і Percona XtraBackup як основних інструментів фізичного резервного копіювання.

У результаті виконання кваліфікаційної роботи було створено програмну утиліту, яка поєднує в одному рішенні фізичне резервне копіювання PostgreSQL і MySQL, ведення каталогу резервних копій, формування службових метаданих, валідацію артефактів, базове відновлення та допоміжний Web UI для перегляду історії операцій.

Розроблена система не претендує на повноцінне промислове рішення для продуктивних середовищ, однак є функціонально завершеним навчально-практичним інструментом, який демонструє основні принципи побудови систем резервного копіювання, відновлення, каталогізації та перевірки цілісності резервних копій.

Перспективами подальшого розвитку системи є розширення підтримки хмарних сховищ, реалізація retention policy для автоматичного очищення старих резервних копій, покращення обробки помилок, розширення тестування на різних версіях PostgreSQL і MySQL, а також доопрацювання MySQL restore для роботи з додатковими форматами стиснених артефактів. Також доцільним є подальше

посилення PITR-валідації та автоматизація перевірки повноти WAL-ланцюга для складніших сценаріїв відновлення.

Робота пройшла апробацію. За її результатами опубліковано наступні тези:

1. Павленко А. І., Садовенко В. С. Методи захисту інформації при проектуванні систем автоматизованого фізичного резервного копіювання баз даних // Застосування програмного забезпечення в інформаційно-комунікаційних технологіях : матеріали VII Всеукраїнської науково-технічної конференції, 23 квітня 2026 р., Київ : ДУІКТ, 2026. С. 296–298.

2. Павленко А. І., Садовенко В. С. Розробка CLI-утиліти для автоматизації резервного копіювання та відновлення баз даних PostgreSQL і MySQL // Технології цифрового розвитку: програмні системи та децентралізація : матеріали Всеукраїнської науково-технічної конференції, 3 травня 2026 р., Київ : ДУІКТ, 2026. Збірник тез. (подано до друку).

ПЕРЕЛІК ПОСИЛАНЬ

1. PostgreSQL Documentation. pg_basebackup [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/app-pgbasebackup.html> (дата звернення: 14.02.2026).
2. PostgreSQL Documentation. Continuous Archiving and Point-in-Time Recovery (PITR) [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/continuous-archiving.html> (дата звернення: 18.02.2026).
3. PostgreSQL Documentation. Write-Ahead Logging (WAL) [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/wal-intro.html> (дата звернення: 21.02.2026).
4. PostgreSQL Documentation. Recovery Target Settings [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/11/recovery-target-settings.html> (дата звернення: 25.02.2026).
5. PostgreSQL Documentation. The Password File [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/libpq-pgpass.html> (дата звернення: 27.02.2026).
6. PostgreSQL Documentation. pg_dump [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/app-pgdump.html> (дата звернення: 02.03.2026).
7. PostgreSQL Documentation. Backup Manifest Format [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/backup-manifest-format.html> (дата звернення: 04.03.2026).
8. MySQL 8.0 Reference Manual. Backup and Recovery [Електронний ресурс]. - Режим доступу: <https://dev.mysql.com/doc/refman/8.0/en/backup-and-recovery.html> (дата звернення: 06.03.2026).

9. MySQL 8.0 Reference Manual. mysqldump - A Database Backup Program [Электронный ресурс]. - Режим доступа: <https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html> (дата звернения: 08.03.2026).
10. MySQL 8.0 Reference Manual. mysql_config_editor - MySQL Configuration Utility [Электронный ресурс]. - Режим доступа: <https://dev.mysql.com/doc/refman/8.0/en/mysql-config-editor.html> (дата звернения: 10.03.2026).
11. MySQL 8.0 Reference Manual. mysqladmin - A MySQL Server Administration Program [Электронный ресурс]. - Режим доступа: <https://dev.mysql.com/doc/refman/8.0/en/mysqladmin.html> (дата звернения: 12.03.2026).
12. Percona XtraBackup Documentation. About Percona XtraBackup [Электронный ресурс]. - Режим доступа: <https://docs.percona.com/percona-xtrabackup/8.0/about-xtrabackup.html> (дата звернения: 15.03.2026).
13. Percona XtraBackup Documentation. Create a Full Backup [Электронный ресурс]. - Режим доступа: <https://docs.percona.com/percona-xtrabackup/8.0/create-full-backup.html> (дата звернения: 17.03.2026).
14. Percona XtraBackup Documentation. Create an Incremental Backup [Электронный ресурс]. - Режим доступа: <https://docs.percona.com/percona-xtrabackup/8.0/create-incremental-backup.html> (дата звернения: 19.03.2026).
15. Percona XtraBackup Documentation. Restore a Backup [Электронный ресурс]. - Режим доступа: <https://docs.percona.com/percona-xtrabackup/8.0/restore-a-backup.html> (дата звернения: 21.03.2026).
16. pgBackRest User Guide [Электронный ресурс]. - Режим доступа: <https://pgbackrest.org/user-guide.html> (дата звернения: 24.03.2026).
17. Barman Documentation [Электронный ресурс]. - Режим доступа: <https://docs.pgbarman.org/> (дата звернения: 27.03.2026).
18. Hutchinson N. C., Manley S., Federwisch M., Harris G., Hitz D., Kleiman S., O'Malley S. Logical vs. Physical File System Backup // Proceedings of the Third

Symposium on Operating Systems Design and Implementation (OSDI '99). - 1999. - P. 239-249. - Режим доступа: <https://www.usenix.org/conference/osdi-99/logical-vs-physical-file-system-backup> (дата звернення: 05.05.2026).

19. Preston W. C. Backup & Recovery: Inexpensive Backup Solutions for Open Systems. – Sebastopol : O'Reilly Media, 2007. – 726 p.
PostgreSQL Architecture – p. 628; Backup and Recovery – p. 632; Point-in-Time Recovery – p. 636.

20. Percona XtraBackup. GitHub Repository [Електронний ресурс]. - Режим доступу: <https://github.com/percona/percona-xtrabackup> (дата звернення: 04.04.2026).

21. Barman - Backup and Recovery Manager for PostgreSQL. GitHub Repository [Електронний ресурс]. - Режим доступу: <https://github.com/EnterpriseDB/barman> (дата звернення: 06.04.2026).

22. pgBackRest. GitHub Repository [Електронний ресурс]. - Режим доступу: <https://github.com/pgbackrest/pgbackrest> (дата звернення: 08.04.2026).

23. Severalnines. PostgreSQL Backups: What is pg_basebackup? [Електронний ресурс]. - Режим доступу: <https://severalnines.com/blog/postgresql-backups-what-is-pg-basebackup/> (дата звернення: 12.04.2026).

24. Severalnines. Logical and Physical PostgreSQL Backups [Електронний ресурс]. - Режим доступу: <https://severalnines.com/blog/logical-physical-postgresql-backups/> (дата звернення: 15.04.2026).

25. Percona Blog. PostgreSQL Backup Strategy for an Enterprise-Grade Environment [Електронний ресурс]. - Режим доступу: <https://www.percona.com/blog/postgresql-backup-strategy-enterprise-grade-environment/> (дата звернення: 18.04.2026).

26. Percona Blog. Backup and Restore Performance: mysqldump vs MySQL Shell Utilities vs mydumper vs mysqlpump vs XtraBackup [Електронний ресурс]. - Режим доступу: <https://www.percona.com/blog/backup-restore-performance-conclusion-mysqldump-vs-mysql-shell-utilities-vs-mydumper-vs-mysqlpump-vs-xtrabackup/> (дата звернення: 21.04.2026).

27. AWS Documentation. Improving MySQL Migration Performance with Percona XtraBackup [Електронний ресурс]. - Режим доступу:

<https://docs.aws.amazon.com/dms/latest/sbs/chap-manageddatabases.mysql2rds.performance.html> (дата звернення: 24.04.2026).

28. Craig Kerstiens. Postgres Backups: Physical vs Logical [Електронний ресурс]. - Режим доступу: <https://www.craigkerstiens.com/2017/09/03/postgres-backups-physical-vs-logical/> (дата звернення: 27.04.2026).

29. Schönig H.-J. Mastering PostgreSQL 17: Advanced Database Techniques for High Performance and Reliability. – Birmingham : Packt Publishing, 2024. – 822 p.
Розділи: PostgreSQL Backup Strategies; Point-in-Time Recovery (PITR); WAL Internals; Replication and Recovery.

30. Python Documentation. argparse - Parser for Command-Line Options [Електронний ресурс]. - Режим доступу: <https://docs.python.org/3/library/argparse.html> (дата звернення: 04.05.2026).

31. Python Documentation. sqlite3 - DB-API 2.0 Interface for SQLite Databases [Електронний ресурс]. - Режим доступу: <https://docs.python.org/3/library/sqlite3.html> (дата звернення: 05.05.2026).

32. Python Documentation. unittest - Unit Testing Framework [Електронний ресурс]. - Режим доступу: <https://docs.python.org/3/library/unittest.html> (дата звернення: 06.05.2026).

33. prompt_toolkit Documentation [Електронний ресурс]. - Режим доступу: <https://python-prompt-toolkit.readthedocs.io/> (дата звернення: 07.05.2026).

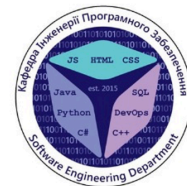
34. SQLite Documentation [Електронний ресурс]. - Режим доступу: <https://sqlite.org/docs.html> (дата звернення: 08.05.2026).

35. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. - Addison-Wesley Professional, 1994. - 395 p.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ
ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



«Програмне забезпечення для резервного копіювання баз даних»

Виконав студент 4 курсу

групи ПД-44

Павленко Артем Ігорович

Керівник:

канд. фіз.-мат. наук, доц. Саловенко Володимир Сергійович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення надійності адміністрування баз даних шляхом розроблення CLI-застосунку для автоматизації резервного копіювання, каталогізації та перевірки відновлюваності копій PostgreSQL і MySQL.

Об'єкт дослідження: процеси резервного копіювання та фізичного відновлення даних у PostgreSQL і MySQL.

Предмет дослідження: методи автоматизації фізичного копіювання, WAL-архівування, ведення каталогу метаданих і перевірки придатності резервних копій до відновлення.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати процеси резервного копіювання та відновлення баз даних.
2. Дослідити логічні та фізичні підходи до резервного копіювання.
3. Провести порівняльний аналіз існуючих рішень для PostgreSQL і MySQL.
4. Визначити функціональні та нефункціональні вимоги до програмного забезпечення.
5. Спроектувати архітектуру системи резервного копіювання та відновлення.
6. Реалізувати програмне забезпечення для створення, каталогізації та перевірки резервних копій.
7. Реалізувати сценарії відновлення PostgreSQL, PITR та MySQL.
8. Провести тестування, валідацію резервних копій та аналіз результатів.

3

АНАЛІЗ АНАЛОГІВ

Характеристики	pgBackRest	Barman	Percona XtraBackup	mysqldump	Розроблена утиліта
Підтримувані СУБД	PostgreSQL	PostgreSQL	MySQL / MariaDB	MySQL	PostgreSQL + MySQL
Інтерфейси керування	CLI	CLI	CLI	CLI	CLI + інтерактивний режим + Web UI
Каталог резервних копій	Manifest / repo metadata	Каталог backup-сервера	Службові файли, без єдиного каталогу	Відсутній	SQLite-каталог
Контроль цілісності	Контрольні суми, manifest	barman check	Службові метадані + prepare	Ручна перевірка після SQL-імпорту	Автоматична валідація артефактів
Відновлення	Restore/PITR після конфігурації	Restore через backup-сервер	--prepare → --copy-back	SQL-імпорт	Автоматизований пайплайн відновлення

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Повне фізичне копіювання PostgreSQL у форматі tar із gzip-стисненням.
2. Похідне копіювання PostgreSQL на основі WAL-сегментів.
3. Відновлення PostgreSQL та підтримка PITR до заданого часу.
4. Повне й інкрементальне копіювання MySQL через XtraBackup.
5. Ведення SQLite-каталогу з 15+ атрибутами резервної копії.
6. Формування metadata.json, validation.json та перегляд через CLI / Web UI.

Нефункціональні вимоги

1. Перевірка наявності обов'язкових артефактів резервного копіювання.
2. Тестове відновлення PostgreSQL із перевіркою SELECT 1.
3. Перегляд каталогу до 1 000 записів - не більше 2 секунд.
4. Заборона передачі паролів через аргументи командного рядка.
5. Локальне зберігання каталогу, метаданих, журналів і копій.
6. Модульна архітектура зі стратегіями для PostgreSQL і MySQL.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



pg_basebackup - фізичне резервне копіювання PostgreSQL

pg_ctl - запуск тимчасового екземпляра

WAL archiving - підтримка PITR



mysqladmin ping - перевірка доступності після відновлення

MySQL



Каталог резервних копій і метадані

SQLite



Повне / інкрементне резервне копіювання MySQL

Percona



Python



GitHub

Інші:



HTML



Zstandard



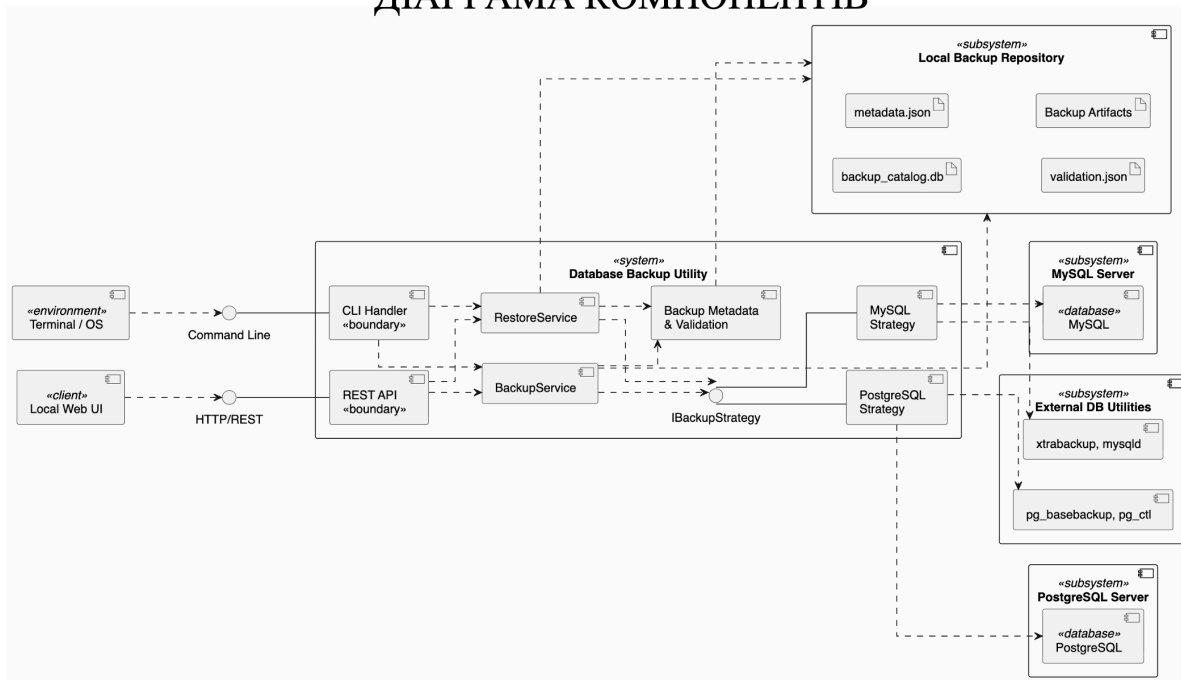
GZIP



prompt_toolkit

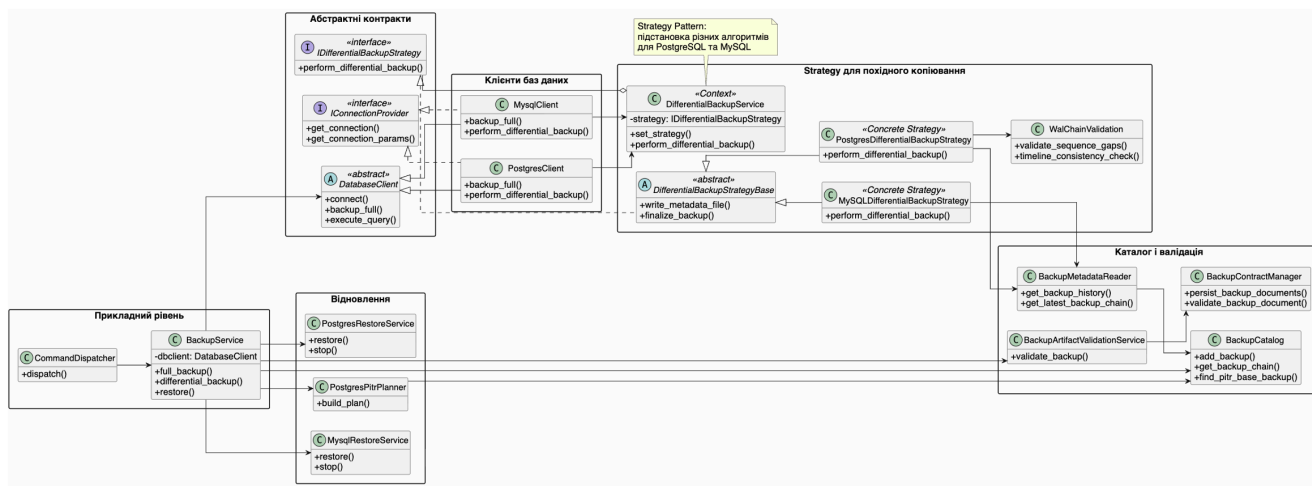
6

ДІАГРАМА КОМПОНЕНТІВ



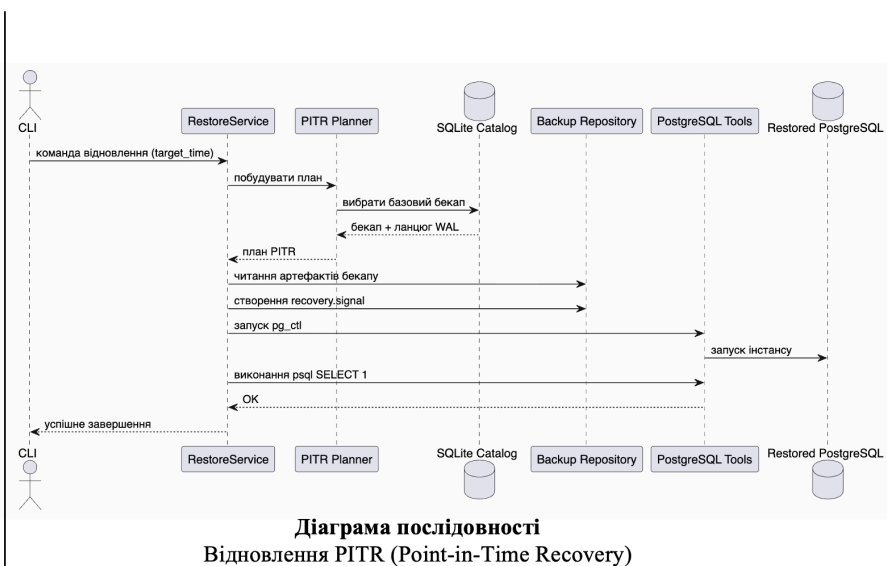
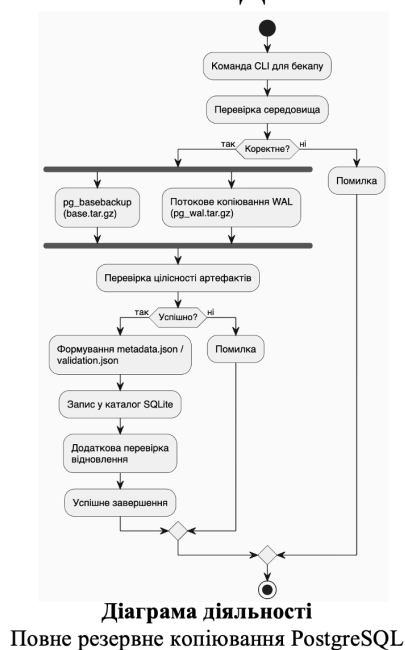
7

ДІАГРАМА КЛАСІВ



9

ДІАГРАМА ДІЯЛЬНОСТІ ТА ПОСЛІДОВНОСТІ



8

ЕКРАННІ ФОРМИ

```

=====
2026-05-11 16:43:49 - backup_BackUp - INFO - 
=====
Storage: LOCAL
2026-05-11 16:43:49 - backup_BackUp - INFO - Storage: LOCAL
=====
2026-05-11 16:43:49 - backup_BackUp - INFO - =====

2026-05-11 16:43:49 - backup_BackUp - INFO - 
Database Backup Utility
2026-05-11 16:43:49 - backup_BackUp - INFO - Database Backup Utility
Connected to: BackUp as artempavlenko
Type 'help' for commands or 'exit' to quit

[[BackUp | WAL:1B | +4]> restore
full database
differential backup
restore
restore database
SQL
-path
-backup-path
  
```

Робота інтерактивної консолі CLI: prompt, WAL-
стан і автодоповнення команд

10

ЕКРАННІ ФОРМИ

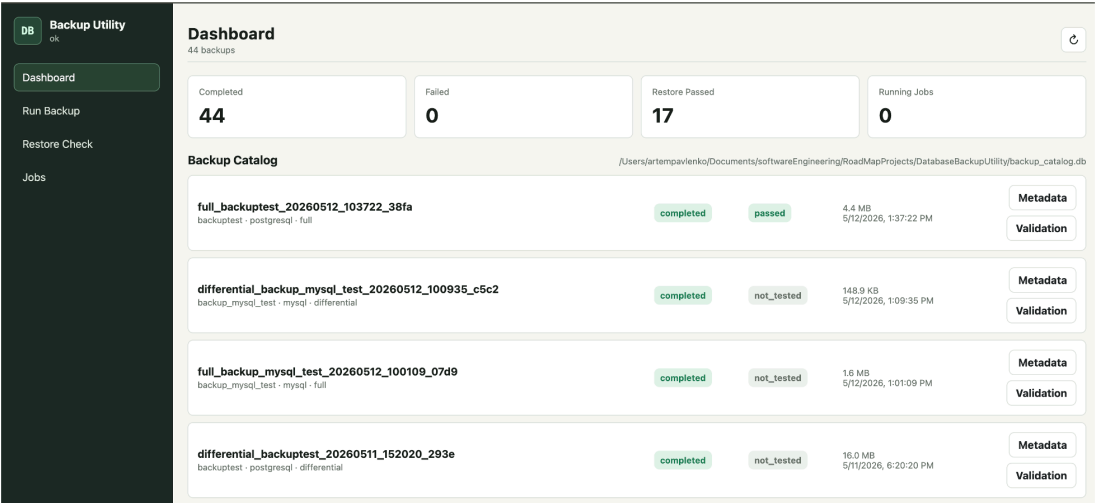
```
[mydb] full database path /Users/artempavlenko/Documents/backups
✓ Prerequisite check: archive_mode is 'on'.
2026-03-31 20:48:49 - backup_mydb - INFO - archive_mode check passed: 'on'.
✓ wal_level is set to 'replica'.
2026-03-31 20:48:49 - backup_mydb - INFO - wal_level check passed: 'replica'.
2026-03-31 20:48:49 - backup_mydb - INFO - pg_basebackup utility check passed.
✓ Replication privilege confirmed.
2026-03-31 20:48:49 - backup_mydb - INFO - User 'artempavlenko' has replication privileges.
Starting full backup → /Users/artempavlenko/Documents/backups
2026-03-31 20:48:49 - backup_mydb - INFO - Starting full backup: full_mydb_20260331_174849_3485
Using PostgreSQL .pgpass for authentication
Running pg_basebackup... (this may take a while)
✓ Backup manifest found at /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485/backup_manifest
2026-03-31 20:48:50 - backup_mydb - INFO - Backup manifest path: /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485/backup_manifest
✓ Full backup created at /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485
Files: base.tar.gz, pg_wal.tar.gz
Backup size: 12.28 MB
2026-03-31 20:48:50 - backup_mydb - INFO - Full backup directory: /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485
2026-03-31 20:48:50 - backup_mydb - INFO - Backup size: 12873492 bytes
Metadata saved: /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485/metadata.json

=====
Creating Single Archive (zstd)
=====
Compressing backup → full_mydb_20260331_174849_3485.tar.zst
Using zstd (fast compression & decompression)
Archiving (level 3, 4 threads)...
✓ Archive created: full_mydb_20260331_174849_3485.tar.zst
Original: 12.28 MB
Compressed: 11.22 MB
Saved: 8.6%
2026-03-31 20:48:50 - backup_mydb - INFO - Archive created: /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485.tar.zst
2026-03-31 20:48:50 - backup_mydb - INFO - Compression ratio: 8.6%
✓ Original backup directory preserved for differential backups
2026-03-31 20:48:50 - backup_mydb - INFO - Keeping original directory for differential backup access: /Users/artempavlenko/Documents/backups/full_mydb_20260331_174849_3485
✓ Single archive ready: full_mydb_20260331_174849_3485.tar.zst
2026-03-31 20:48:50 - backup_mydb - INFO - Backup completed: full_mydb_20260331_174849_3485 (1.14s, 12.28 MB)
```

Результат виконання сценарію повного резервного копіювання PostgreSQL

11

ЕКРАННІ ФОРМИ



Графічний Web UI для перегляду історії резервних копій та стану артефактів

12

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Павленко А. І., Садовенко В. С. Методи захисту інформації при проектуванні систем автоматизованого фізичного резервного копіювання баз даних // Застосування програмного забезпечення в інформаційно-комунікаційних технологіях : матеріали VII Всеукраїнської науково-технічної конференції, 23 квітня 2026 р., Київ : ДУІКТ, 2026. С. 296–298.
2. Павленко А.І. Садовенко В.С. Розробка CLI-утиліти для автоматизації резервного копіювання та відновлення баз даних PostgreSQL і MySQL. Матеріали Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. – К.: ДУІКТ, 2026. (подано до друку)

13

ВИСНОВКИ

1. Проаналізовано процеси резервного копіювання та відновлення баз даних PostgreSQL і MySQL, зокрема логічний і фізичний підходи, WAL-архівацию, PITR та механізми XtraBackup.
2. Проведено порівняльний аналіз існуючих рішень: pgBackRest, Barman, pg_basebackup, Percona XtraBackup та mysqldump. Встановлено, що більшість інструментів орієнтовані на окрему СУБД або потребують складнішої конфігурації.
3. Визначено функціональні та нефункціональні вимоги до системи, зокрема підтримку PostgreSQL і MySQL, каталогізацію резервних копій, формування метаданих, валідацію артефактів та безпечну автентифікацію.
4. Спроектовано модульну архітектуру системи з єдиним CLI-інтерфейсом, локальним Web UI, SQLite-каталогом, окремими клієнтами для PostgreSQL і MySQL та сервісами відновлення.
5. Реалізовано основні сценарії резервного копіювання: повне фізичне копіювання PostgreSQL, похідне копіювання на основі WAL, повне копіювання MySQL та інкрементальне копіювання через Percona XtraBackup.
6. Реалізовано сценарії відновлення PostgreSQL, PostgreSQL PITR та пайплайн відновлення MySQL з перевіркою працездатності відновлених екземплярів баз даних.
7. Проведено unit- та інтеграційне тестування backup/restore-сценаріїв; підтверджено створення копій, валідацію артефактів і відновлення тестових даних.
8. Розроблена система може використовуватися у навчальних, тестових і невеликих адміністративних середовищах як уніфікований інструмент для створення, перевірки та відновлення резервних копій PostgreSQL і MySQL.

14

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ ПРОГРАМНИХ МОДУЛІВ

Лістинг Б.1 - dbtool.py

```

parser.add_argument(
    "command",
    choices=["backup", "restore"],
    help="Command to execute"
)
parser.add_argument(
    "--db",
    required=True,
    choices=["postgres", "mysql"],
    help="Database type: postgres or mysql"
)
parser.add_argument("--database",
    required=False, help="Name of the database")
parser.add_argument("--storage",
    choices=["local", "cloud"])
parser.add_argument("--config",
    choices=["manual", "file", "profile"])
parser.add_argument("--backup-path",
    help="Backup directory or supported archive to restore")
parser.add_argument("--data-dir", help="New database data directory for restore")
parser.add_argument("--archive-dir", help="Path to archived WAL files for PostgreSQL restore")
parser.add_argument("--target-time",
    help="PostgreSQL PITR target time")
parser.add_argument("--catalog-path",
    default="backup_catalog.db")
parser.add_argument("--restore-port", type=int,
    default=55432)
parser.add_argument("--verify-user",
    default="postgres")
parser.add_argument("--verify-query",
    default="SELECT 1;")
args = parser.parse_args()
try:
    if args.command == "restore":
        if not args.data_dir:
            parser.error("--data-dir is required for restore")
        restore_name = args.database or args.db
        restore_logger = BackupLogger(
            name=f"restore_{restore_name}",
            log_file=f"restore_{restore_name}.log"
        )
        configure_messenger(logger=restore_logger.logger,
            enable_colors=True)
        messenger = get_messenger()
        if args.db == "mysql":
            if not args.backup_path:
                parser.error("--backup-path is required for MySQL restore")
            if args.target_time:
                parser.error("--target-time restore is currently implemented only for PostgreSQL")
            mysql_restore_port = int(args.port) if args.port
            else 3308
            restore_service =
            MysqlRestoreService(
                backup_path=args.backup_path,
                data_dir=args.data_dir,
                restore_port=mysql_restore_port,
                socket_path=args.mysql_socket,
                mysql_bin_dir=args.mysql_bin_dir,
                owner_user=args.mysql_owner,
                owner_group=args.mysql_group,
                logger=restore_logger,

```

```

                messenger=messenger,
            )
        if not restore_service.restore():
            sys.exit(EXIT_FAILURE)
        sys.exit(0)
        if not args.backup_path and not
        args.target_time:
            parser.error("--backup-path is
            required for restore unless --target-time is
            used for PITR")
        archive_dir = args.archive_dir
        if not archive_dir:
            archive_dir =
            PostgresWalArchiveConfig().get_archive_directory
            ()
            if archive_dir:
                messenger.info(f"Using saved WAL
                archive directory: {archive_dir}")
            if not archive_dir:
                parser.error("--archive-dir is
                required for restore")

```

Лістинг Б.2 - backup_services.py

```

class BackupService:
    def __init__(self, dbclient) -> None:
        self.dbclient = dbclient
    def full_backup(self, parsed_args) -> None:
        if not parsed_args.path:
            raise ValueError("Path is required.
            Use: full database -path <path>")
        storage_type = getattr(parsed_args,
            "storage_type", "local")
        single_archive = getattr(parsed_args,
            "single_archive", True)
        if getattr(self.dbclient, "database_engine",
            None) == "postgresql":
            self.dbclient.backup_full(
                outpath=parsed_args.path,
                single_archive=single_archive,
                storage=storage_type,
                validate_restore=getattr(parsed_args,
                    "validate_restore", True),
                restore_validation_port=getattr(parsed_args,
                    "restore_validation_port", 0),
                restore_validation_keep_data=getattr(
                    parsed_args,
                    "restore_validation_keep_data",
                    False,
                ),
                restore_validation_checksums=getattr(
                    parsed_args,
                    "restore_validation_checksums",
                    True,
                ),
                pg_bin_dir=(
                    str(parsed_args.pg_bin_dir)
                    if getattr(parsed_args,
                        "pg_bin_dir", None)
                    else None
                ),
            )
        return
        self.dbclient.backup_full(
            outpath=parsed_args.path,
            single_archive=single_archive,
            storage=storage_type,

```

```

def differential_backup(self, parsed_args) ->
None:
    metadata_reader = BackupMetadataReader(
        BackupCatalog(),
        self.dbclient._messenger,
        self.dbclient._logger,
        self.dbclient._database
    )
    self.dbclient.perform_differential_backup(metadata_reader)
def restore_database(self, parsed_args) -> None:
    engine = getattr(self.dbclient,
        "database_engine", None)
    if engine == "postgresql":
        return
    self.restore_postgres(parsed_args)
    if engine == "mysql":
        return
    self.restore_mysql(parsed_args)
    raise ValueError(f"Restore is not
supported for database engine: {engine}")

```

Лістинг Б.3 - postgres_client.py

```

class PostgresClient(
    ConnectionConfigMixin,
    BackupCatalogMixin,
    DifferentialBackupMixin,
    DatabaseClient,
    IConnectionProvider,
):
    def __init__(self, host: str, database: str,
user: str, password: str, **kwargs: Any) ->
None:
        if "port" not in kwargs:
            kwargs["port"] = 5432
        self._database_engine = "postgresql"
        self._use_pgpass =
kwargs.pop("use_pgpass", False)
        archive_path =
kwargs.pop("archive_path", None)
        configure_wal_archive =
kwargs.pop("configure_wal_archive", True)
        super().__init__(host, database, user, password,
**kwargs)
        self._connection: Optional[connection] =
None
        self._archive_path: Optional[str] = (
str(Path(archive_path).expanduser().resolve())
if archive_path else None
        )
        self._wal_config =
PostgresWalArchiveConfig() if
configure_wal_archive else None
        if self._archive_path:
            self._messenger.info(f"WAL archive
directory: {self._archive_path}")
            self._logger.info(f"WAL archive path
configured: {self._archive_path}")
            elif configure_wal_archive:
                self._configure_wal_archive()
        def connect(self) -> Optional[connection]:
            try:
                connect_kwargs = {
                    "dbname": self._database,
                    "user": self._user,
                    "host": self._host,
                    "port": self._port,
                    "connect_timeout": 10,
                }
            if not self._use_pgpass and self._password:
                connect_kwargs["password"] =
self._password
            self._connection =
psycopg2.connect(**connect_kwargs)
            with self._connection.cursor() as

```

```

cur:
            cur.execute("SELECT version();")
            version_tuple:
Optional[Tuple[Any, ...]] = cur.fetchone()
            if version_tuple is None:
                self._logger.error("Failed
fetch version from database")
                return None
            version: str = version_tuple[0]
            self._database_version =
version.split(",")[0]

            self._messenger.success("PostgreSQL connection
successful!")
            self._logger.info(
                f"Connected to database:
{self._database} ({self._database_version})"
            )
            return self._connection
            except psycopg2.OperationalError as e:
                self._messenger.error(f"Unable to
connect. Details: {e}")
                self._logger.error(f"Connection
failed: {e}")
                return None
        @check_archive_mode
        @check_wal_level
        @check_basebackup
        @requires_replication_privilege
        def backup_full(
            self,
            outpath: str,
            single_archive: bool = True,
            storage="local",
            validate_restore: bool = True,
            restore_validation_port: int = 0,
            restore_validation_keep_data: bool =
False,
            restore_validation_checksums: bool =
True,
            pg_bin_dir: str | None = None,
        ) -> bool:
            base_path = Path(outpath) if
isinstance(outpath, str) else outpath
            metadata = self._logger.start_backup(
                backup_type="full",
                database=self._database,
                database_type=self._database_engine,
                database_version=self._database_version or
"Unknown",
                utility_version="pg_basebackup",
                compress=True,
                storage=storage,
            )
            backup_id = metadata["id"]
            backup_dir = base_path / backup_id
            backup_dir.mkdir(parents=True,
exist_ok=True)
            metadata["backup_location"] =
str(backup_dir)
            pg_basebackup_cmd = [
                "pg_basebackup",
                "-h", self._host,
                "-p", str(self._port),
                "-U", self._user,
                "-D", str(backup_dir),
                "-F", "t",
                "-X", "stream",
                "-c", "fast",
                "-P",
                "-v",
                "-l", f"UTIL_FULL_{backup_id}",
                "--manifest-checksums=SHA256",
                "-Z",
            ]

```

```

        "-Z", "6",
    ]
    env = os.environ.copy()
    if self._use_pgpass:
        env.pop("PGPASSWORD", None)
        metadata["auth_method"] = "pgpass"
    else:
        env["PGPASSWORD"] = self._password
        metadata["auth_method"] = "password"
    process = subprocess.run(
        pg_basebackup_cmd,
        capture_output=True,
        env=env,
        check=False,
        text=True,
    )
    return process.returncode == 0

```

Лістинг Б.4 - mysql_client.py

```

class MySQLClient(
    ConnectionConfigMixin,
    BackupCatalogMixin,
    DifferentialBackupMixin,
    DatabaseClient,
    IConnectionProvider,
):
    def __init__(self, host, database, user,
        password, **kwargs):
        if "port" not in kwargs:
            kwargs["port"] = 3306
        self._database_engine = "mysql"
        login_path = kwargs.pop("login_path", None)
        socket = kwargs.pop("socket", None)
        super().__init__(host, database, user, password,
            **kwargs)
        self._connection: Optional[Connection] =
            None
        self._login_path = login_path
        self._socket = socket
        if self._login_path:
            self._extract_login_path_config()
        def validate_connection(self):
            try:
                if not self.is_connected:
                    return False
            if self._login_path:
                result = subprocess.run(
                    ["mysql", f"--login-
path={self._login_path}", "-e", "SELECT 1;",
                    self._database],
                    capture_output=True,
                    text=True,
                    timeout=5,
                )
                return result.returncode == 0
            with self._connection.cursor() as cur:
                cur.execute("SELECT 1 as test;")
                result = cur.fetchone()
                return result["test"] == 1
            except Exception:
                return False
        @check_utility_available("xtrabackup")
        def backup_full(self, outpath: str,
            single_archive: bool = True, storage: str =
            "local") -> bool:
            base_path = Path(outpath) if
            isinstance(outpath, str) else outpath
            metadata = self._logger.start_backup(
                backup_type="full",
                database=self._database,
                database_type=self._database_engine,
                database_version=self._database_version or
                "Unknown",
                utility_version="xtrabackup",
                compress=True,

```

```

                storage=storage,
            )
            backup_id = metadata["id"]
            backup_dir = base_path / backup_id
            backup_dir.mkdir(parents=True,
                exist_ok=True)
            metadata["backup_location"] =
                str(backup_dir)
            if self._login_path:
                xtrabackup_cmd = [
                    "xtrabackup",
                    "--backup",
                    f"--target-dir={backup_dir}",
                    f"--login-
path={self._login_path}",
                    "--compress",
                    "--compress-threads=4",
                ]
            if self._socket:
                xtrabackup_cmd.append(f"--
socket={self._socket}")
            env = os.environ.copy()
            else:
                xtrabackup_cmd = [
                    "xtrabackup",
                    "--backup",
                    f"--target-dir={backup_dir}",
                    f"--user={self._user}",
                    f"--host={self._host}",
                    f"--port={self._port}",
                    "--compress",
                    "--compress-threads=4",
                ]
            env = os.environ.copy()
            env["MYSQL_PWD"] = self._password
            process = subprocess.run(
                xtrabackup_cmd,
                capture_output=True,
                check=False,
                text=True,
                env=env,
            )
            return process.returncode == 0

```

Лістинг Б.5 - core.py

```

class DifferentialBackupService:
    def __init__(
        self,
        connection_provider:
            IConnectionProvider,
        logger: ILogger,
        messenger: IMessenger,
        strategy: IDifferentialBackupStrategy =
            None,
    ):
        self._connection_provider =
            connection_provider
        self._logger = logger
        self._messenger = messenger
        self._strategy = strategy
        def set_strategy(self, strategy:
            IDifferentialBackupStrategy) -> None:
            self._strategy = strategy
        def write_metadata_file(self, metadata: dict,
            output_path: Path) -> bool:
            try:
                metadata_file = output_path /
                "metadata.json"
                with open(metadata_file, "w") as f:
                    json.dump(metadata, f, indent=2,
                        ensure_ascii=False, default=str)
                self._messenger.info(f"Metadata
                saved: {metadata_file}")
                return True
            except Exception as e:
                self._messenger.error(f"Failed to

```

```

write metadata file: {e}")
    self.logger.error(f"Failed to write
metadata file: {e}")
    return False
def perform_differential_backup(self,
metadata_reader: BackupMetadataReader) -> bool:
    if not self._strategy:
        self._messenger.error("No
differential backup strategy configured!")
        self._logger.error("Differential
backup strategy not set")
        return False
    try:
        return
    self._strategy.perform_differential_backup(metadata_reader)
    except Exception as e:
        self._messenger.error(f"Differential
backup strategy failed: {e}")
        self._logger.error(f"Strategy
execution failed: {e}")
        return False

```

Лістинг Б.6 - contracts.py

```

@dataclass
class ArtifactChecksum:
    path: str
    algorithm: str
    checksum_sha256: str
    size_bytes: int
@dataclass
class ValidationSummary:
    state: str
    report_path: str
    metadata_schema_valid: bool
    manifest_present: bool
    checkpoints_present: bool
    recovery_status: str
class BackupContractManager:
    def __init__(self, logger, messenger=None):
        self._logger = logger
        self._messenger = messenger
    def persist_backup_documents(
        self,
        metadata: Dict[str, Any],
        output_dir: Path | str,
        backup_succeeded: bool,
    ) -> Dict[str, Any]:
        output_dir = Path(output_dir)
        output_dir.mkdir(parents=True,
exist_ok=True)
        metadata_doc =
self._build_metadata_document(metadata,
output_dir)
        validation_doc =
self._build_validation_document(metadata_doc,
output_dir)
        validation_doc["status"] = (
            "passed" if backup_succeeded and
validation_doc["status"] == "passed" else
"failed"
        )
        validation_summary = ValidationSummary(
            state=validation_doc["status"],
            report_path=str(output_dir /
"validation.json"),
            metadata_schema_valid=validation_doc["metadata_s
chema"]["valid"],
            manifest_present=validation_doc["manifest"]["exi
sts"],
            checkpoints_present=validation_doc["checkpoints"
][["exists"]],

```

```

recovery_status=validation_doc["recovery_status"
][["state"]],
        )
        metadata_doc["validation_status"] =
asdict(validation_summary)
        metadata_doc["status"] = (
            "completed" if backup_succeeded and
validation_summary.state == "passed" else
"failed"
        )
        metadata_doc["metadata_path"] =
str(output_dir / "metadata.json")
        metadata_doc["validation_path"] =
str(output_dir / "validation.json")
        self._write_json(output_dir / "metadata.json",
metadata_doc)
        self._write_json(output_dir /
"validation.json", validation_doc)
        return metadata_doc

```

Лістинг Б.7 - catalog.py

```

class BackupCatalog:
    def __init__(
        self,
        path: str = "backup_catalog.db",
        legacy_json_path: str =
"backup_catalog.json",
    ):
        if not isinstance(path, str):
            raise ValueError("The catalog path
must be a string.")
        if not path.endswith(".db"):
            raise ValueError("The catalog file
must be a SQLite .db file.")
        self.catalog_path = Path(path)
        self.legacy_json_path =
Path(legacy_json_path)
        self.logger =
logging.getLogger("BackupCatalog")
        self._initialize_schema()
        self._import_legacy_json_if_needed()
    @contextmanager
    def _connect(self) ->
Iterator[sqlite3.Connection]:
        connection =
sqlite3.connect(self.catalog_path)
        connection.row_factory = sqlite3.Row
        try:
            yield connection
            connection.commit()
        except Exception:
            connection.rollback()
            raise
        finally:
            connection.close()
    def add_backup(self, new_backup: Dict[str, Any])
-> None:
        if not isinstance(new_backup, dict):
            raise ValueError("The new backup
must be a dictionary.")
        payload = dict(new_backup)
        lsn_markers = (
            payload.get("lsn_markers", {})
            if
isinstance(payload.get("lsn_markers"), dict)
            else {}
        )
        binlog_markers = (
            payload.get("binlog_markers", {})
            if
isinstance(payload.get("binlog_markers"), dict)
            else {}
        )
        backup_id = payload.get("id") or
payload.get("backup_id")
        if not backup_id:

```

```

        raise ValueError("Backup metadata
must contain id or backup_id.")
record = {
    "backup_id": backup_id,
    "database_name":
payload.get("database_name") or "",
    "engine": payload.get("engine") or
payload.get("database_type") or "unknown",
    "backup_type":
payload.get("backup_type") or
payload.get("type") or "unknown",
    "status": payload.get("status") or
"unknown",
    "timestamp_start":
payload.get("timestamp_start") or "",
    "timestamp_end":
payload.get("timestamp_end"),
    "parent_backup_id":
payload.get("parent_backup_id"),
    "base_backup_id":
payload.get("base_backup_id"),
    "backup_location":
payload.get("backup_location"),
    "metadata_path":
payload.get("metadata_path"),
    "validation_path":
payload.get("validation_path"),
    "current_lsn":
lsn_markers.get("current_lsn") or
payload.get("current_lsn"),
    "binlog_file":
binlog_markers.get("binlog_file") or
payload.get("binlog_file"),
    "payload_json": json.dumps(payload,
ensure_ascii=False, default=str),
}

```

Лістинг Б.8 - postgres_restore.py

```

class PostgresRestoreService:
    CONFIG_BLOCK_START = "#
DBTOOL_POSTGRES_RESTORE_BEGIN"
    CONFIG_BLOCK_END = "#
DBTOOL_POSTGRES_RESTORE_END"
    HBA_BLOCK_START = "#
DBTOOL_POSTGRES_RESTORE_HBA_BEGIN"
    HBA_BLOCK_END = "#
DBTOOL_POSTGRES_RESTORE_HBA_END"
    def __init__(
        self,
        backup_path: str,
        data_dir: str,
        archive_dir: str,
        restore_port: int,
        verify_database: str,
        verify_user: str = "postgres",
        verify_query: str = "SELECT 1;",
        pg_bin_dir: str | None = None,
        target_time: str | None = None,
        logger=None,
        messenger=None,
    ) -> None:
        self._backup_path =
Path(backup_path).expanduser().resolve()
        self._data_dir =
Path(data_dir).expanduser().resolve()
        self._archive_dir =
Path(archive_dir).expanduser().resolve()
        self._restore_port = int(restore_port)
        self._verify_database = verify_database
        self._verify_user = verify_user
        self._verify_query = verify_query
        self._pg_bin_dir =
Path(pg_bin_dir).expanduser().resolve() if
pg_bin_dir else None
        self._target_time = (

```

```

format_recovery_target_time(parse_recovery_targe
t_time(target_time))
        if target_time
        else None
    )
    self._logger = logger
    self._messenger = messenger
    self._extract_root: Path | None = None
    def restore(self) -> bool:
        try:
            self._validate_inputs()
            backup_dir =
self._resolve_backup_directory()
            self._validate_backup_directory(backup_dir)
            self._prepare_data_directory()
            self._extract_tar_archive(backup_dir
/ "base.tar.gz", self._data_dir)
            self._extract_tar_archive(backup_dir
/ "pg_wal.tar.gz", self._data_dir)
            self._data_dir.chmod(0o700)
            self._write_restore_configuration()
            self._prepend_local_hba_rules()
            self._create_recovery_signal()
            self._start_postgresql()
            self._verify_select()
            return True
        except Exception as error:
            self._message_error(f"Restore
failed: {error}")
            return False
        finally:
            self._cleanup_extract_root()

```

Лістинг Б.9 - pitr.py

```

PITR_TIMEZONE_POLICY = (
    "PITR target times are normalized to UTC;
timestamps without timezone are treated as UTC."
)
@dataclass(frozen=True)
class PitrRestorePlan:
    backup: dict[str, Any]
    backup_path: Path
    target_time: datetime
    recovery_target_time: str
    def parse_recovery_target_time(value: str) ->
datetime:
        if not value or not value.strip():
            raise ValueError("PITR target time
cannot be empty.")
        normalized = value.strip()
        if normalized.endswith("Z"):
            normalized = f"{normalized[:-1]}+00:00"
        try:
            target_time =
datetime.fromisoformat(normalized)
        except ValueError as error:
            raise ValueError(
                "Invalid --target-time value. Use an
ISO-like timestamp, for example "
                "'2026-05-04 12:30:00' for UTC or
'2026-05-04T12:30:00+03:00'."
            ) from error
        if target_time.tzinfo is None:
            return
        target_time.replace(tzinfo=timezone.utc)
        return target_time.astimezone(timezone.utc)
class PostgresPitrPlanner:
    def build_plan(self) -> PitrRestorePlan:
        target_time =
parse_recovery_target_time(self._target_time)
        backup =
self._catalog.find_pitr_base_backup(
            database_name=self._database_name,
            target_time=target_time,
        )

```

```

        if not backup:
            raise ValueError(
                "No completed PostgreSQL full
                backup found at or before "
                f"target time
                {format_recovery_target_time(target_time)} "
                f"for database
                {self._database_name}."
            )
        backup_path = self._resolve_backup_path(backup)
        recovery_target_time =
        format_recovery_target_time(target_time)
        return PitrRestorePlan(
            backup=backup,
            backup_path=backup_path,
            target_time=target_time,

        recovery_target_time=recovery_target_time,
    )

```

Лістинг Б.10 - mysql_restore.py

```

class MysqlRestoreService:
    REQUIRED_FILES = ("xtrabackup_info",
                     "xtrabackup_checkpoints", "ibdata1")
    REQUIRED_DIRECTORIES = ("mysql",)
    def __init__(
        self,
        backup_path: str,
        data_dir: str,
        restore_port: int = 3308,
        socket_path: str =
        "/tmp/mysql_restore.sock",
        mysql_bin_dir: str | None = None,
        owner_user: str | None = None,
        owner_group: str | None = None,
        verify_timeout_seconds: int = 30,
        logger=None,
        messenger=None,
    ) -> None:
        self._backup_path =
        Path(backup_path).expanduser().resolve()
        self._data_dir =
        Path(data_dir).expanduser().resolve()
        self._restore_port = int(restore_port)
        self._socket_path =
        Path(socket_path).expanduser().resolve()
        self._mysql_bin_dir = (
            Path(mysql_bin_dir).expanduser().resolve() if
            mysql_bin_dir else None
        )
        self._owner_user = owner_user
        self._owner_group = owner_group
        self._verify_timeout_seconds =
        int(verify_timeout_seconds)
        self._logger = logger
        self._messenger = messenger
        self._process: subprocess.Popen | None =
        None
        self._mysqld_user: str | None = None
    def restore(self) -> bool:
        started = False
        try:
            self._validate_inputs()
            self._decompress_zstd_artifacts()

        self._validate_backup_directory(self._backup_path)

        self._prepare_backup()
        self._prepare_data_directory()
        self._copy_back()
        self._set_data_directory_owner()
        self._start_mysql()
        started = True
        self._verify_mysql()
        return True

```

```

        except Exception as error:
            self._message_error(f"MySQL restore
            failed: {error}")
            if started:
                self.stop()
            return False

```

Лістинг Б.11 - postgres_restore_validation.py

```

class PostgresRestoreValidationService:
    def __init__(
        self,
        backup_path: str | Path,
        archive_dir: str | Path,
        source_connection_params: dict[str,
        Any],

        restore_port: int = 0,
        verify_database: str | None = None,
        verify_user: str | None = None,
        pg_bin_dir: str | Path | None = None,
        checksum_tables: bool = True,
        keep_data_dir: bool = False,
        logger=None,
        messenger=None,
    ) -> None:
        self._backup_path =
        Path(backup_path).expanduser().resolve()
        self._archive_dir =
        Path(archive_dir).expanduser().resolve()
        self._source_connection_params =
        dict(source_connection_params)
        self._restore_port = int(restore_port)
        self._verify_database = (
            verify_database
            or
            self._source_connection_params.get("database")
            or
            self._source_connection_params.get("dbname")
        )
        self._verify_user = (
            verify_user
            or
            self._source_connection_params.get("user")
            or "postgres"
        )
        self._pg_bin_dir =
        Path(pg_bin_dir).expanduser().resolve() if
        pg_bin_dir else None
        self._checksum_tables = checksum_tables
        self._keep_data_dir = keep_data_dir
        self._logger = logger
        self._messenger = messenger
    def validate(self) -> dict[str, Any]:
        started_at = self._utc_now()
        report =
        self._build_base_report(started_at)
        temp_root: Path | None = None
        restore_service: PostgresRestoreService
        | None = None
        try:
            source_snapshot =
            self._collect_table_signatures(
                self._build_source_connect_kwargs()
            )
            report["source"] = source_snapshot
            report["checks"] = ["source_snapshot"]
        = {
            "passed":
            source_snapshot["state"] == "passed",
            "tables_checked":
            len(source_snapshot["tables"]),
        }
        temp_root =
        Path(tempfile.mkdtemp(prefix="dbtool_pg_restore_
        validation_"))
        data_dir = temp_root / "data"

```

```

        selected_port =
self._select_restore_port()
        report["temporary_instance"].update(
            {
                "data_dir": str(data_dir),
                "port": selected_port,
                "keep_data_dir":
self._keep_data_dir,
            }
        )
restore_service = PostgresRestoreService(
    backup_path=str(self._backup_path),
    data_dir=str(data_dir),

    archive_dir=str(self._archive_dir),
    restore_port=selected_port,

    verify_database=self._verify_database,
    verify_user=self._verify_user,
    verify_query="SELECT 1;",
    pg_bin_dir=str(self._pg_bin_dir)
if self._pg_bin_dir else None,
    logger=self._logger,
    messenger=self._messenger,
)
restored = restore_service.restore()

report["checks"]["temporary_restore"] =
{"passed": restored}
    if not restored:
        report["state"] = "failed"
        report["error"] = "Temporary
PostgreSQL restore failed"
    return report
restored_snapshot =
self._collect_table_signatures(

self._build_restored_connect_kwargs(selected_por
t)
    )
    comparison =
self._compare_snapshots(source_snapshot,
restored_snapshot)
    report["checks"]["table_comparison"]
= comparison
    report["state"] = "passed" if
comparison["matches"] else "failed"
    except Exception as error:
        report["state"] = "failed"
        report["error"] = str(error)
    finally:
        if restore_service is not None:

report["temporary_instance"]["stopped"] =
restore_service.stop()
        if temp_root is not None and
temp_root.exists() and not self._keep_data_dir:
            shutil.rmtree(temp_root,
ignore_errors=True)
    return report

```

Лістинг Б.12 - wal_check.py

```

SEGMENTS_PER_LOG = 0x100
LOGS_PER_TIMELINE = 0x100000000
def get_next_wal_segment(current_wal: str) ->
str:
    timeline_hex = current_wal[0:8]
    log_hex = current_wal[8:16]
    seg_hex = current_wal[16:24]
    timeline = int(timeline_hex, 16)
    log = int(log_hex, 16)
    seg = int(seg_hex, 16)
    seg += 1
    if seg >= SEGMENTS_PER_LOG:
        seg = 0

```

```

        log += 1
        if log >= LOGS_PER_TIMELINE:
            log = 0
            timeline += 1
    return f"{timeline:08X}{log:08X}{seg:08X}"
class WalChainValidation:
    def __init__(
        self,
        archived_wal_files: list[str],
        last_full_backup_wal_file: str,
        current_wal_file: str,
        wal_archive_directory: str | Path,
        logger,
        messenger,
    ):
        self.archived_wal_files: list[str] =
sorted(archived_wal_files)
        self.last_full_backup_wal_file: str =
last_full_backup_wal_file
        self.current_wal_file: str =
current_wal_file
        self.wal_archive_directory: Path =
Path(wal_archive_directory)
        self._logger = logger
        self._messenger = messenger
    def _iter_relevant_wal_files(self) -> list[str]:
        return [
            wal
            for wal in self.archived_wal_files
            if self.last_full_backup_wal_file <
wal <= self.current_wal_file
        ]
    def validate_sequence_gaps(self) -> bool:
        relevant_wal_files =
self._iter_relevant_wal_files()
        if not relevant_wal_files:
            return True
        expected_wal =
get_next_wal_segment(self.last_full_backup_wal_f
ile)
        for wal in relevant_wal_files:
            wal_path =
self.wal_archive_directory / wal
            if expected_wal < wal:
                self._logger.error(
                    f"Detected gap in WAL chain.
First missing segment: {expected_wal}"
                )
                return False
            if expected_wal == wal:
                if not wal_path.exists():
                    self._logger.error(f"WAL
listed but does not exist on disk: {wal}")
                    return False
                expected_wal =
get_next_wal_segment(expected_wal)
        return True

```

Лістинг Б.13 - command_dispatcher.py

```

class CommandDispatcher:
    def __init__(self, storage_type: str =
"local"):
        self.commands = {}
        self.storage_type = storage_type
    def register_command(self, command_name: str,
handler):
        self.commands[command_name.lower()] =
handler
    def dispatch(self, command_name: str,
parsed_args):
        command_name = command_name.strip()
        normalized_command =
command_name.lower()
        command_mapping = {
            "full database": "full backup",
            "differential backup":

```

```

"differential_backup",
    "validate_backup":
"validate_backup",
    "restore": "restore_database",
    "restore database":
"restore_database",
    "help": "help",
}
if normalized_command.startswith("sql "):
    mapped_command = "execute_sql"
    sql_query = command_name[4:].strip()
    return
self.execute_command(mapped_command, sql_query,
    parsed_args)
mapped_command =
command_mapping.get(normalized_command,
    normalized_command)
if mapped_command not in self.commands:
    raise ValueError(f"Command
'{command_name}' not recognized.")
if mapped_command in ["full_backup",
"differential_backup"]:
    parsed_args.storage_type =
self.storage_type
return self.execute_command(mapped_command,
    parsed_args)
def execute_command(self, command_name: str,
    *args, **kwargs):
    command_name = command_name.lower()
    if command_name not in self.commands:
        raise ValueError(f"Command
'{command_name}' not recognized.")
    return
self.commands[command_name](*args, **kwargs)

```

Лістинг Б.14 - server.py

```

def run_backup_job(job_id: str, payload:
dict[str, Any]) -> dict[str, Any]:
    engine = require_text(payload, "engine")
    if engine == "postgres":
        return run_postgres_backup(job_id,
payload)
    if engine == "mysql":
        return run_mysql_backup(job_id, payload)
    raise WebApiError(f"Unsupported engine:
{engine}")
def run_postgres_backup(job_id: str, payload:
dict[str, Any]) -> dict[str, Any]:
    require_module("psycogp2", "psycogp2-
binary")
    from clients.postgres_client import
PostgresClient
    database = require_text(payload, "database")
    output_path = require_text(payload,
"output_path")
    logger = build_job_logger(job_id, database)
    messenger = JobMessenger(job_id)

```

```

        archive_dir = str(payload.get("archive_dir")
or "").strip() or None
    client = PostgresClient(
        host=require_text(payload, "host"),
        port=as_int(payload, "port", 5432),
        database=database,
        user=require_text(payload, "user"),
        password=str(payload.get("password") or
""),
        use_pgpass=as_bool(payload,
"use_pgpass", False),
        archive_path=archive_dir,
        configure_wal_archive=False,
        logger=logger,
        messenger=messenger,
    )
    try:
        if client.connect() is None:
            raise WebApiError("PostgreSQL
connection failed")
        if not client.validate_connection():
            raise WebApiError("PostgreSQL
connection validation failed")
        success = client.backup_full(
            output_path=output_path,
            single_archive=as_bool(payload,
"single_archive", True),
            storage="local",
            validate_restore=as_bool(payload,
"validate_restore", True),
            restore_validation_port=as_int(payload,
"restore_validation_port", 0),
            restore_validation_keep_data=as_bool(
                payload,
                "restore_validation_keep_data",
                False,
            ),
            restore_validation_checksums=as_bool(
                payload,
                "restore_validation_checksums",
                True,
            ),
            pg_bin_dir=str(payload.get("pg_bin_dir") or
"").strip() or None,
        )
        if not success:
            raise WebApiError("PostgreSQL backup
finished with failed status")
        return {"success": True, "engine": "postgres",
            "database": database}
        finally:
            if client.is_connected:
                client.disconnect()

```