НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет Інформаційних технологій

УДК 004.75

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	**Завідувач кафедри**
Інформаційних технологій	Комп'ютерних систем, мереж та кібербезпеки
__________ Болбот І.М., д.т.н, проф.	__________________ Касаткін Д.Ю., к. пед.н, доц.
підпис ПІБ, вчене звання і ступінь	підпис ПІБ, вчене звання і ступінь
«___»___________2024 р.	«___»___________2024 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: «Дослідження засобів автоматизації обчислювальних процесів розподілених комп'ютерних систем»

Спеціальність: 123 «Комп’ютерна інженерія»

Освітня програма: Комп’ютерні системи і мережі

Орієнтація освітньої програми: Освітньо-професійна

Гарант освітньої програми

д.т.н., доцент ______________ _ Шкарупило В. В.

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник дипломного проекту

д.т.н., доцент ______________ _ Шкарупило В. В.

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав ________________ _ Савчук Ю.І.

(підпис) (ПІБ студента)

КИЇВ – 2024

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем і мереж
______________ / Касаткін Д.Ю., к.п.н., доц./
підпис ПІБ, вчене звання і ступінь
«__»___________20__р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ РОБОТИ СТУДЕНТУ

Савчуку Юрію Івановичу
(прізвище, ім'я, по батькові)

Спеціальність (напрям підготовки) комп'ютерна інженерія
Освітня програма____________________
Орієнтація освітньої програми____________________

Тема магістерської роботи Дослідження засобів автоматизації обчислювальних процесів розподілених комп'ютерних систем
____
____
затверджена наказом ректора НУБіП України від "_1_" 11 2023 р. № 1999 С
Термін подання завершеної роботи на кафедру ____________________
Вихідні дані до магістерської роботи ____________________
____
____
____
____

Перелік питань, що підлягають дослідженню:

1. ____________________
2. ____________________
3. ____________________

Перелік графічного матеріалу (за потреби) ____________________
____

Дата видачі завдання "______"___________________2023 р.

Керівник магістерської роботи______________ Шкарупило В. В., д.т.н., доц.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання ____________ Савчук Ю.І.
(підпис) (прізвище та ініціали студента)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області		Виконано
2	Проєктування системи		Виконано
3	Реалізація системи		Виконано
4	Тестування системи		Виконано
5	Оформлення пояснювальної записки		Виконано

Студент __________Ю.І. Савчук
(підпис) (ініціали та прізвище)

Керівник проєкт (роботи) __________В.В. Шкарупило
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 78 сторінок, 17 рисунків, 12 таблиць, 2 додатки, 21 джерело.

РОЗПОДІЛЕНІ СИСТЕМИ, АВТОМАТИЗАЦІЯ, ОБЧИСЛЮВАЛЬНІ ПРОЦЕСИ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, КЛАСТЕРНІ СИСТЕМИ, БАЛАНСУВАННЯ НАВАНТАЖЕННЯ, МАСШТАБУВАННЯ, НАДІЙНІСТЬ, DOCKER, PROMETHEUS, POSTGRESQL, REDIS.

Мета роботи – дослідження та оптимізація засобів автоматизації обчислювальних процесів для підвищення ефективності обробки даних та обчислень з використанням сучасних технологій контейнеризації.

Об'єкт дослідження – процеси автоматизації обчислень у розподілених комп'ютерних системах з використанням технологій.

Предмет дослідження – методи та засоби оптимізації автоматизованих обчислювальних процесів для підвищення продуктивності та ефективності обробки даних.

У першому розділі проведено комплексний аналіз предметної області, включаючи детальний огляд сучасних розподілених систем, засобів автоматизації та їх порівняльний аналіз.

У другому розділі розглянуто функціональні та нефункціональні вимоги до системи автоматизації обчислювальних процесів. Визначено ключові компоненти системи та їх взаємодію. Описано архітектурні рішення та обґрунтовано вибір технологій реалізації.

Третій розділ присвячено практичній реалізації компонентів системи автоматизації.

У четвертому розділі представлено результати дослідження та оптимізації продуктивності системи.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 6
ВСТУП 7
1 АНАЛІТИЧНИЙ ОГЛЯД РОЗПОДІЛЕНИХ КОМП'ЮТЕРНИХ СИСТЕМ 9
1.1 Аналіз вимог користувачів 9
1.2 Технологічні засади автоматизації розподілених обчислень 14
1.3 Огляд існуючих рішень та технологій 16
2 ОПИС СИСТЕМИ АВТОМАТИЗАЦІЇ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ ... 24
2.1 Вимоги до системи автоматизації 24
2.2 Цілі та задачі проєкту 28
2.3 Структура проєкту 29
3 ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ 32
3.1 Проєктування архітектури системи 32
3.2 Реалізація програмних компонентів 36
3.3 База даних 40
3.4 Алгоритми та методи оптимізації продуктивності обчислювальних процесів 45
4 ДОСЛІДЖЕННЯ КОМП’ЮТЕРНОЇ СИСТЕМИ 50
4.1 Мета тестування 50
4.2 Методологія тестування автоматизованих процесів 51
4.3 Види тестів 51
4.4 Оптимізація швидкодії та масштабування 54
4.5 Результати й висновки 63
ВИСНОВКИ 68
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 70
Додаток А 72
Додаток Б 78

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AAA	–	Authentication, Authorization, та Accounting концепція в інформаційній безпеці, яка означає процеси автентифікації, авторизації та акаутингу
API	–	Application Programming Interface, програмований інтерфейс додатку
OpenCL	–	Open Computing Language, відкрита мова обчислень
RAM	–	Random Access Memory, оперативна пам'ять
REST	–	Representational State Transfer, передача репрезентативного стану
RPC	–	Remote Procedure Call, віддалений виклик процедур
SQL	–	Structured Query Language, мова структурованих запитів
БД	–	База даних
ОС	–	Операційна система
ПЗ	–	Програмне забезпечення
РКС	–	Розподілена комп'ютерна система

ВСТУП

У сучасному світі розподілені комп'ютерні системи стали фундаментальною частиною інформаційної інфраструктури. Вони забезпечують можливість вирішення складних обчислювальних задач, обробку великих масивів даних та підтримку високонавантажених сервісів. Проте зростаюча складність таких систем створює нові виклики в їх управлінні та оптимізації, що робить автоматизацію обчислювальних процесів критично важливою.

Актуальність теми дослідження обумовлена постійно зростаючими вимогами до ефективності обробки даних та необхідністю оптимізації використання обчислювальних ресурсів. Сучасні розподілені системи повинні забезпечувати високу продуктивність, масштабованість та надійність, що неможливо без ефективної автоматизації всіх процесів. Особливої уваги потребують питання балансування навантаження, відмовостійкості та оптимального розподілу ресурсів.

Метою дослідження є вивчення та оптимізація засобів автоматизації обчислювальних процесів у розподілених комп'ютерних системах для підвищення їх ефективності та продуктивності. Це включає аналіз існуючих рішень, розробку нових методів автоматизації та їх практичну реалізацію.

Об'єктом дослідження є процеси автоматизації обчислень у розподілених комп'ютерних системах, включаючи методи розподілу задач, балансування навантаження та управління ресурсами.

Предметом дослідження є методи та засоби оптимізації автоматизованих обчислювальних процесів у розподілених системах, спрямовані на підвищення ефективності обробки даних та використання ресурсів.

Наукова новизна роботи полягає в розробці удосконалених методів автоматизації обчислювальних процесів, які враховують специфіку сучасних розподілених систем та забезпечують більш ефективне використання ресурсів.

Практична цінність отриманих результатів полягає в можливості їх безпосереднього застосування для підвищення ефективності роботи розподілених

комп'ютерних систем у різних галузях, включаючи наукові обчислення, обробку великих даних та хмарні обчислення.

Методи дослідження базуються на теорії розподілених систем, методах оптимізації, теорії масового обслуговування та експериментальних дослідженнях на реальних системах.

В процесі дослідження були поставлені та вирішені наступні завдання:

- аналіз існуючих підходів до автоматизації обчислювальних процесів;
- розробка удосконалених методів автоматизації;
- створення та тестування прототипу системи;
- експериментальна перевірка ефективності запропонованих рішень.

Результати роботи можуть бути використані при проєктуванні та оптимізації розподілених комп'ютерних систем різного масштабу та призначення. Запропоновані методи дозволяють підвищити ефективність автоматизації обчислювальних процесів та забезпечити більш раціональне використання доступних ресурсів.

1 АНАЛІТИЧНИЙ ОГЛЯД РОЗПОДІЛЕНИХ КОМП'ЮТЕРНИХ СИСТЕМ

1.1 Аналіз вимог користувачів

1.1.1 Дослідження критичних вимог до автоматизації розподілених обчислень

Детальний аналіз вимог до автоматизації обчислювальних процесів в розподілених системах є фундаментальним етапом дослідження. При проєктуванні таких систем особлива увага приділяється продуктивності обчислень. Ключовим аспектом є забезпечення мінімальних затримок при розподілі задач між вузлами системи та оптимальне використання доступних обчислювальних ресурсів. Не менш важливим фактором виступає ефективна синхронізація паралельних процесів та швидкість обробки даних при їх передачі між компонентами системи.

Надійність системи постає другим критичним фактором у процесі автоматизації. Система повинна забезпечувати автоматичне відновлення після збоїв окремих вузлів, зберігаючи при цьому цілісність даних. Важливим елементом надійності є реплікація критично важливих компонентів та впровадження механізмів раннього виявлення потенційних проблем.

Гнучкість конфігурації системи відіграє важливу роль у забезпеченні її ефективності. Можливість динамічного масштабування ресурсів та адаптивного налаштування параметрів дозволяє системі ефективно справлятися з різними типами обчислювальних задач. Система повинна підтримувати гнучкі механізми розподілу пріоритетів між процесами та завданнями.

Істотне значення має організація контролю та моніторингу автоматизованих процесів. Система повинна забезпечувати постійний збір метрик продуктивності, аналіз ефективності використання ресурсів та відстеження стану всіх компонентів. На основі цих даних має здійснюватися генерація аналітичних звітів та своєчасне сповіщення про критичні події.

Безпека процесів є невід'ємною складовою автоматизованої системи. Це включає захист даних при їх передачі між вузлами, контроль доступу до

обчислювальних ресурсів, забезпечення ізоляції окремих обчислювальних процесів та ведення детального аудиту системних подій.

Особливу увагу при автоматизації необхідно приділяти розподілу обчислювального навантаження між вузлами з урахуванням їх потужності та поточної завантаженості. Система повинна забезпечувати автоматичне масштабування ресурсів при зміні інтенсивності обчислень та динамічно адаптуватися до змін у конфігурації. При виникненні збоїв має відбуватися автоматичне відновлення з мінімальними втратами даних.

Важливою вимогою є забезпечення прозорості процесів автоматизації для кінцевих користувачів при збереженні передбачуваності поведінки системи за різних сценаріїв навантаження. При цьому повинна зберігатися можливість ручного втручання в критичних ситуаціях, а всі автоматизовані операції мають детально протоколюватися (рис. 1.1).

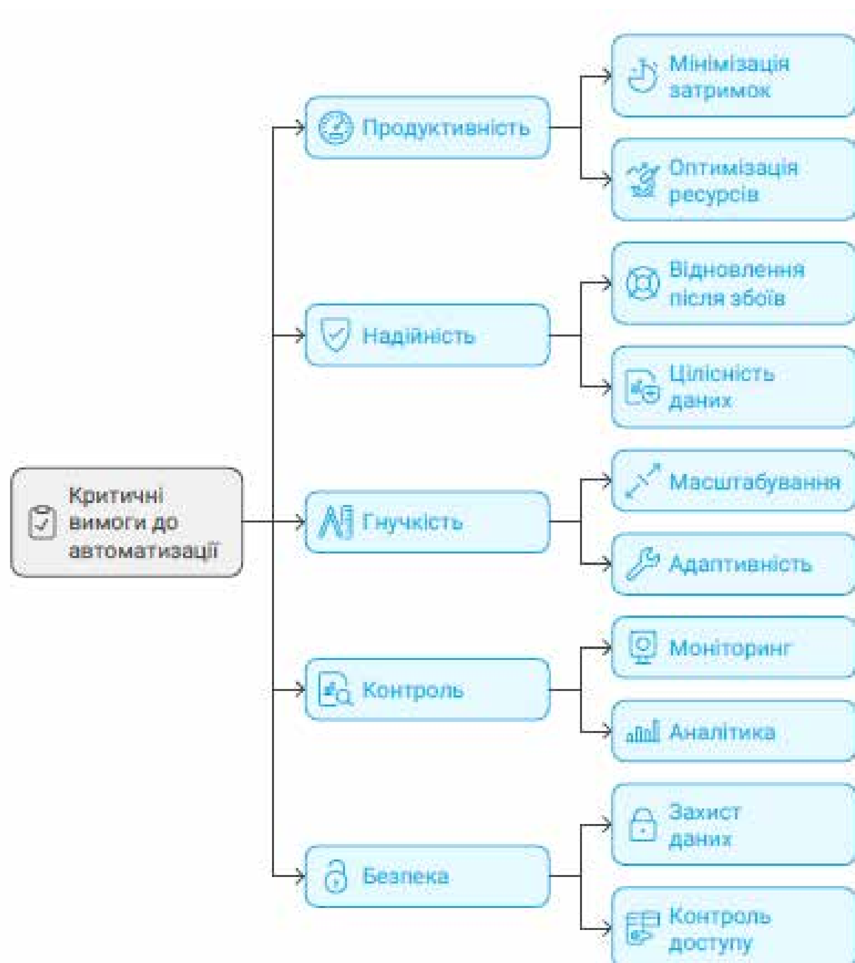

Рисунок 1.1 - Діаграма критичних вимог до автоматизації

1.1.2 Характеристика користувацьких сегментів у розподілених обчислювальних системах

Аналіз потреб різних категорій користувачів розподілених обчислювальних систем виявляє суттєві відмінності у вимогах та очікуваннях від автоматизації процесів. Наукові співробітники та дослідники потребують максимальної гнучкості у налаштуванні параметрів обчислень та можливості тонкого контролю над розподілом ресурсів. Для них критично важливою є можливість експериментувати з різними конфігураціями системи та отримувати детальну інформацію про ефективність обчислювальних процесів. Особливу увагу ця група користувачів приділяє точності результатів та можливості відтворення експериментів.

Інженери-розробники, які працюють з розподіленими системами, зосереджені на технічних аспектах інтеграції та розробки. Їм необхідні чітко документовані API, інструменти для відлагодження та моніторингу продуктивності, а також можливість швидкого розгортання та тестування нових компонентів. Важливим аспектом для цієї групи є наявність інструментів автоматизації процесів розробки та тестування.

Системні адміністратори та DevOps інженери концентруються на аспектах управління інфраструктурою та забезпечення стабільності роботи системи. Їх основний інтерес полягає в наявності ефективних інструментів моніторингу, автоматизації розгортання та масштабування системи, а також у можливості швидкого реагування на інциденти. Ця група потребує розвинених засобів автоматизації рутинних операцій та надійних механізмів відновлення після збоїв.

Бізнес-користувачі та керівники проєктів зацікавлені насамперед у прозорості витрат на обчислювальні ресурси та передбачуваності масштабування системи. Для них важливими є інструменти аналітики та звітності, що дозволяють оцінювати ефективність використання ресурсів та планувати розвиток інфраструктури. Ця категорія користувачів потребує зрозумілих метрик продуктивності та економічної ефективності системи.

Аналітики даних та фахівці з машинного навчання мають специфічні вимоги до організації обчислювальних процесів. Їм необхідна можливість

ефективної обробки великих масивів даних, паралельного виконання складних алгоритмів та зручні інструменти для візуалізації результатів. Особливу увагу ця група приділяє оптимізації використання GPU та спеціалізованих обчислювальних ресурсів (рис. 1.2).

Рисунок 1.2 – Діаграма вимог користувачів

1.1.3 Класифікація та характеристика обчислювальних навантажень у розподілених системах

Аналіз обчислювальних процесів у розподілених системах вимагає детального розуміння характеру та специфіки різних типів навантажень. Інтенсивні обчислення, що вимагають значних процесорних ресурсів, становлять

основну категорію навантажень у наукових та інженерних задачах. Такі процеси характеризуються тривалим часом виконання та високим ступенем розпаралелювання, що робить їх ідеальними кандидатами для розподіленої обробки. Особливу увагу при автоматизації таких обчислень необхідно приділяти балансуванню навантаження між вузлами системи та оптимізації використання процесорного часу.

Операції з великими масивами даних формують окрему категорію навантажень, яка вимагає ефективної організації процесів введення-виведення та оптимізації передачі даних між вузлами системи. Такі операції часто зустрічаються в задачах аналізу даних, машинного навчання та обробки великих наборів інформації. При автоматизації подібних процесів критичними факторами стають ефективність розподілу даних, мінімізація мережевого трафіку та оптимізація операцій з пам'яттю.

Реальночасові обчислення представляють особливий клас навантажень, де критичним фактором є швидкість відгуку системи та передбачуваність затримок. Такі процеси часто зустрічаються в системах управління, моніторингу та взаємодії з користувачем. Автоматизація таких обчислень вимагає особливої уваги до планування задач та пріоритезації процесів для забезпечення гарантованого часу відгуку.

Транзакційні обчислення характеризуються необхідністю забезпечення атомарності операцій та узгодженості даних у розподіленому середовищі. Такі процеси типові для фінансових систем, систем управління ресурсами та баз даних. При автоматизації важливо забезпечити надійний механізм координації транзакцій та відновлення після збоїв.

Потокова обробка даних вимагає особливого підходу до організації обчислень, де дані надходять безперервним потоком і повинні оброблятися в режимі реального часу. Такі процеси характерні для систем моніторингу, аналізу соціальних мереж та обробки сенсорних даних. Автоматизація потокових обчислень повинна забезпечувати ефективне масштабування та адаптацію до змін інтенсивності потоку даних (табл. 1.1).

Таблиця 1.1 - Характеристики типів обчислень

Тип навантаження	Ключові характеристики	Вимоги до автоматизації
Інтенсивні обчислення	Високе CPU навантаження, тривалий час виконання	Оптимальне розпаралелювання, балансування навантаження
Обробка великих даних	Великі обсяги даних, інтенсивний I/O	Ефективний розподіл даних, оптимізація передачі
Реальночасові обчислення	Критичність часу відгуку, передбачуваність	Гарантований час виконання, пріоритезація задач
Транзакційні операції	Атомарність, узгодженість даних	Надійна координація, відновлення після збоїв
Потокова обробка	Безперервність даних, змінна інтенсивність	Динамічне масштабування, адаптивність

1.2 Технологічні засади автоматизації розподілених обчислень

Сучасний стан розвитку технологій автоматизації розподілених обчислень характеризується стрімкою еволюцією підходів та інструментів, спрямованих на підвищення ефективності обробки даних. В основі сучасних рішень лежить концепція контейнеризації, яка дозволяє створювати ізольовані середовища для виконання обчислювальних задач. Технології, такі як Docker, надають можливість стандартизувати розгортання додатків та забезпечити їх портативність між різними обчислювальними середовищами. Разом з системами оркестрації, зокрема Kubernetes, ці технології формують потужну основу для автоматизації управління обчислювальними ресурсами.

Важливим аспектом автоматизації є використання систем управління чергами повідомлень, які забезпечують надійну комунікацію між компонентами розподіленої системи. Такі системи, як Apache Kafka та RabbitMQ, дозволяють організувати асинхронну взаємодію та ефективно розподіляти навантаження між

обчислювальними вузлами. Паралельно з цим, розвиток технологій розподілених баз даних забезпечує надійне зберігання та обробку даних у масштабованому середовищі.

Фреймворки для паралельних обчислень становлять окремий клас інструментів, що дозволяють ефективно використовувати обчислювальні ресурси. Apache Spark та Hadoop MapReduce надають потужні можливості для обробки великих обсягів даних, автоматизуючи розподіл задач та управління ресурсами. Особливу роль відіграють технології машинного навчання та штучного інтелекту, які все частіше інтегруються в системи автоматизації для оптимізації розподілу ресурсів та прогнозування навантаження.

Системи моніторингу та аналізу продуктивності є невід'ємною частиною сучасних розподілених систем. Інструменти, такі як Prometheus та Grafana, забезпечують збір та візуалізацію метрик продуктивності, що дозволяє автоматизувати процеси виявлення та усунення проблем. Важливим аспектом є також розвиток технологій service mesh, які автоматизують управління мережевою взаємодією між сервісами.

Таблиця 1.2 - Можливості технологій автоматизації

Категорія	Ключові технології	Функціональні можливості
Контейнеризація	Docker, Kubernetes	Ізоляція середовища, оркестрація, масштабування
Управління даними	Apache Kafka, PostgreSQL	Асинхронна комунікація, зберігання даних
Обчислення	Spark, MapReduce	Паралельна обробка, розподілені обчислення
Моніторинг	Prometheus, Grafana	Збір метрик, візуалізація, аналітика
Безпека	OAuth, SSL/TLS	Автентифікація, шифрування, аудит

В контексті безпеки, сучасні технології автоматизації інтегрують механізми автоматичного шифрування даних, управління доступом та аудиту подій. Значна увага приділяється розвитку інструментів автоматизованого

тестування та забезпечення якості, що дозволяє підвищити надійність розподілених систем.

В результаті аналізу технологій автоматизації розподілених обчислень можна зробити висновок про формування комплексної екосистеми інструментів, що забезпечують високий рівень автоматизації всіх аспектів роботи розподілених систем. Ключовим трендом є інтеграція різних технологій для створення ефективних та надійних рішень автоматизації.

1.3 Огляд існуючих рішень та технологій

1.3.1 Сучасні платформи автоматизації розподілених обчислень та їх порівняльний аналіз

Розвиток технологій розподілених обчислень привів до появи різноманітних платформ, кожна з яких пропонує власний підхід до автоматизації обчислювальних процесів. Apache Hadoop виступає фундаментальною платформою для розподіленої обробки великих масивів даних, забезпечуючи надійне зберігання та ефективну пакетну обробку інформації. Ця платформа особливо ефективна при роботі з неструктурованими даними та виконанні складних аналітичних задач, що вимагають значних обчислювальних ресурсів.

Apache Spark, як еволюційний розвиток концепцій Hadoop, пропонує більш гнучкий підхід до обробки даних, забезпечуючи можливість як пакетної, так і потокової обробки. Завдяки використанню технології обробки даних в оперативній пам'яті, Spark досягає значно вищої продуктивності порівняно з традиційними MapReduce рішеннями. Платформа відрізняється розвиненою екосистемою інструментів для машинного навчання та аналітики даних.

Kubernetes представляє собою потужну платформу оркестрації контейнерів, яка автоматизує розгортання, масштабування та управління контейнеризованими додатками. Основною перевагою Kubernetes є його здатність ефективно управляти складними розподіленими системами, забезпечуючи високу

доступність та відмовостійкість. Платформа надає гнучкі механізми для автоматичного балансування навантаження та відновлення після збоїв.

OpenStack фокусується на створенні та управлінні приватними хмарними інфраструктурами, надаючи комплексне рішення для автоматизації обчислювальних ресурсів. Платформа забезпечує високий рівень контролю над інфраструктурою та можливість гнучкого налаштування під специфічні потреби організації. Особливістю OpenStack є його модульна архітектура, що дозволяє вибирати та комбінувати необхідні компоненти.

Docker Swarm пропонує нативне рішення для оркестрації контейнерів Docker, відрізняючись простотою налаштування та використання. Ця платформа особливо ефективна для менших та середніх розподілених систем, де важлива швидкість розгортання та простота управління. Docker Swarm забезпечує базові функції оркестрації, включаючи автоматичне масштабування та балансування навантаження.

Таблиця 1.3 - Порівняльна характеристика платформ

Платформа	Основні переваги	Особливості застосування	Масштабованість
Apache Hadoop	Надійне зберігання даних, ефективна пакетна обробка	Великі масиви неструктурованих даних	Висока
Apache Spark	Швидка обробка в пам'яті, підтримка ML	Аналітика та машинне навчання	Висока
Kubernetes	Гнучка оркестрація, висока доступність	Великі розподілені системи	Дуже висока
OpenStack	Повний контроль над інфраструктурою	Приватні хмарні середовища	Висока
Docker Swarm	Простота використання, швидке розгортання	Середні системи	Середня

Аналіз цих платформ показує тенденцію до інтеграції різних технологій та

підходів для створення комплексних рішень автоматизації. Кожна платформа має свої особливості та сфери найбільш ефективного застосування, що дозволяє вибирати оптимальне рішення залежно від конкретних вимог та масштабу системи.

1.3.2 Методологія автоматизації обчислювальних процесів

Методологія автоматизації обчислювальних процесів у розподілених системах базується на комплексному підході до організації та оптимізації обчислювальних ресурсів. Фундаментальним аспектом цієї методології виступає декомпозиція складних обчислювальних задач на атомарні операції, що дозволяє ефективно розподіляти навантаження між обчислювальними вузлами. Такий підхід забезпечує не лише оптимальне використання доступних ресурсів, але й створює основу для масштабування системи відповідно до зростаючих потреб.

Таблиця 1.4 - Аспекти методології автоматизації

Аспект методології	Ключові механізми	Очікувані результати
Декомпозиція задач	Розбиття на атомарні операції, планування ресурсів	Оптимальне використання ресурсів, підвищення ефективності
Балансування навантаження	Динамічний розподіл, моніторинг завантаженості	Рівномірне використання ресурсів, запобігання перевантаженням
Синхронізація процесів	Розподілений консенсус, контроль узгодженості	Надійна координація операцій, цілісність даних
Моніторинг та оптимізація	Збір метрик, адаптивне управління	Постійне підвищення продуктивності, рання діагностика проблем
Забезпечення надійності	Резервування, відновлення після збоїв	Висока доступність системи, мінімізація простоїв

Важливим елементом методології є впровадження механізмів динамічного балансування навантаження, які забезпечують рівномірний розподіл обчислювальних задач між вузлами системи. При цьому враховуються такі фактори як поточна завантаженість вузлів, їх обчислювальна потужність та специфіка виконуваних операцій. Система автоматично аналізує ці параметри та приймає рішення щодо оптимального розміщення нових завдань, забезпечуючи максимальну ефективність використання ресурсів.

Особлива увага в методології приділяється організації процесів синхронізації та координації розподілених обчислень. Використання механізмів розподіленого консенсусу дозволяє забезпечити узгодженість даних та операцій у системі, навіть при виникненні збоїв окремих компонентів. При цьому застосовуються алгоритми, що мінімізують накладні витрати на синхронізацію, зберігаючи при цьому необхідний рівень надійності та узгодженості.

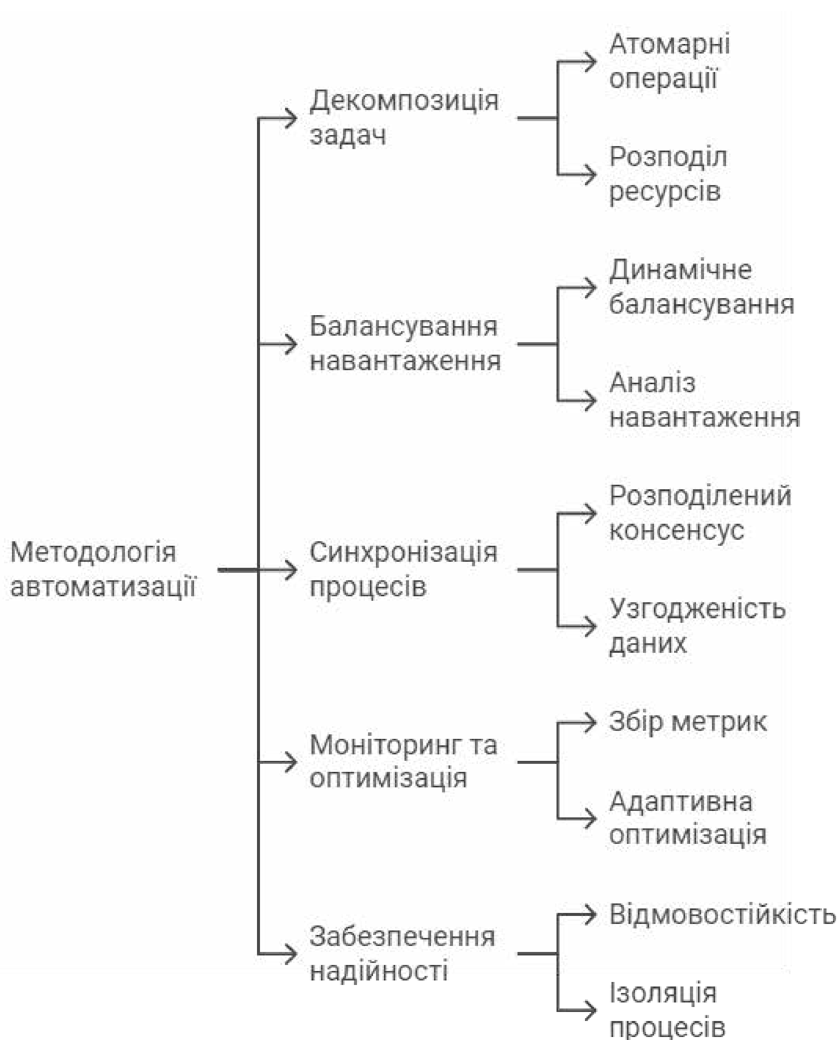

Рисунок 1.3 - Структура методології автоматизації

Методологія також охоплює аспекти автоматизованого моніторингу та управління продуктивністю системи. Впроваджуються механізми збору та аналізу метрик, що дозволяють в реальному часі відслідковувати ефективність виконання обчислень та виявляти потенційні проблеми. На основі цих даних система може автоматично приймати рішення щодо оптимізації розподілу ресурсів та конфігурації компонентів.

В контексті забезпечення надійності методологія передбачає впровадження багаторівневої системи відмовостійкості. Це включає автоматичне резервування критичних компонентів, механізми відновлення після збоїв та проактивне виявлення потенційних проблем. Важливим аспектом є також забезпечення ізоляції обчислювальних процесів, що дозволяє мінімізувати вплив можливих збоїв на роботу системи в цілому.

1.3.3 Інструменти та мови програмування для автоматизації в кластерних системах

Сучасна автоматизація кластерних систем спирається на широкий спектр спеціалізованих інструментів та мов програмування. Python займає лідируючі позиції завдяки своїй універсальності та багатій екосистемі бібліотек для розподілених обчислень. Фреймворки, такі як Dask та PySpark, надають потужні можливості для паралельної обробки даних та управління кластерними ресурсами, дозволяючи ефективно масштабувати обчислення від окремих машин до великих кластерів.

Мова Go набуває все більшої популярності в сфері автоматизації кластерних систем завдяки вбудованій підтримці конкурентності та високій продуктивності. Її активно використовують для розробки інструментів оркестрації та управління контейнерами, про що свідчить реалізація Kubernetes саме на Go. Вбудовані механізми обробки помилок та ефективна робота з мережевими протоколами роблять Go особливо привабливою для створення розподілених систем управління.

Scala, як мова функціонального програмування на платформі JVM, надає потужні абстракції для роботи з розподіленими системами. Завдяки тісній

інтеграції з екосистемою Apache Spark, Scala стала стандартом для розробки складних систем обробки даних. Її система типів та підтримка функціонального програмування дозволяють створювати надійні та масштабовані рішення для автоматизації кластерних обчислень.

Rust виділяється своєю здатністю забезпечувати безпеку пам'яті та високу продуктивність без додаткових накладних витрат. Ця мова все частіше використовується для розробки критично важливих компонентів кластерних систем, особливо в випадках, де потрібна максимальна ефективність та надійність. Система володіння в Rust допомагає уникнути багатьох типових помилок при розробці паралельних та розподілених систем.

Для автоматизації процесів розгортання та конфігурації широко використовуються декларативні мови та інструменти, такі як Ansible та Terraform. Вони дозволяють описувати інфраструктуру як код, забезпечуючи відтворюваність та масштабованість конфігурацій кластерних систем. Ці інструменти інтегруються з системами безперервної інтеграції, автоматизуючи весь життєвий цикл розгортання та управління кластерами.

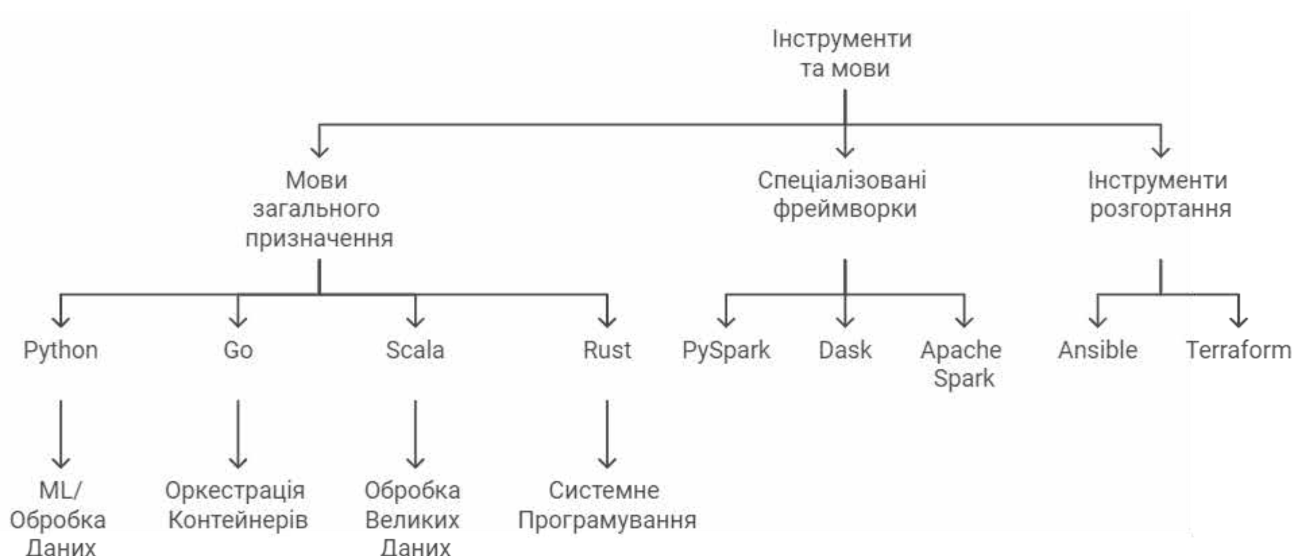

Рисунок 1.4 - Екосистема інструментів автоматизації

1.3.4 Балансування навантаження та масштабування в автоматизованих системах

Ефективне балансування навантаження та масштабування є критичними

компонентами сучасних розподілених обчислювальних систем. В основі процесу балансування лежить динамічний розподіл обчислювальних задач між доступними ресурсами системи, що забезпечує оптимальне використання потужностей та запобігає перевантаженню окремих вузлів. Реалізація цього механізму вимагає постійного моніторингу стану системи та швидкого реагування на зміни навантаження.

Горизонтальне масштабування, що передбачає додавання нових обчислювальних вузлів, забезпечує лінійне зростання продуктивності системи. При цьому алгоритми балансування автоматично розподіляють навантаження на нові вузли, враховуючи їх обчислювальні можливості та поточний стан мережі. Вертикальне масштабування, яке полягає у збільшенні потужності існуючих вузлів, застосовується для оптимізації продуктивності окремих компонентів системи.

Важливим аспектом є впровадження предиктивного балансування навантаження, яке базується на аналізі історичних даних та прогнозуванні майбутніх піків навантаження. Система автоматично адаптує свою конфігурацію, готуючись до очікуваного збільшення навантаження, що дозволяє уникнути затримок та забезпечити стабільну роботу сервісів. Такий підхід особливо ефективний у системах з передбачуваними патернами навантаження.

Механізми автоматичного відновлення після збоїв інтегруються з системою балансування навантаження, забезпечуючи безперервність роботи при виході з ладу окремих компонентів. При виявленні проблем система автоматично перерозподіляє навантаження на працездатні вузли, одночасно ініціюючи процеси відновлення або заміни несправних компонентів. Це дозволяє мінімізувати вплив технічних проблем на загальну продуктивність системи.

Оптимізація мережевої взаємодії між компонентами системи відіграє ключову роль у ефективному балансуванні навантаження. Використання спеціалізованих протоколів та механізмів кешування дозволяє знизити латентність та підвищити пропускну здатність системи. При цьому враховуються особливості топології мережі та географічне розташування обчислювальних вузлів.

Таблиця 1.5 - Ключові метрики балансування та масштабування

Аспект системи	Метрики	Методи оптимізації
Балансування навантаження	Утилізація CPU/RAM, latency, throughput	Динамічний розподіл, предиктивна адаптація
Горизонтальне масштабування	Кількість вузлів, розподіл навантаження	Автоматичне додавання/видалення вузлів
Вертикальне масштабування	Використання ресурсів, продуктивність	Оптимізація конфігурації вузлів
Відмовостійкість	Час відновлення, доступність сервісів	Автоматичне відновлення, резервування
Мережева оптимізація	Пропускна здатність, затримки	Кешування, оптимізація протоколів

Ефективність системи балансування навантаження оцінюється за комплексом показників, які включають як технічні метрики продуктивності, так і бізнес-показники стабільності та надійності роботи системи. Постійний моніторинг та аналіз цих метрик дозволяє оптимізувати роботу системи та забезпечити її ефективне масштабування відповідно до зростаючих потреб.

2 ОПИС СИСТЕМИ АВТОМАТИЗАЦІЇ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ

2.1 Вимоги до системи автоматизації

2.1.1 Загальна мета проєкту

Основною метою проєкту є розробка та впровадження системи автоматизації обчислювальних процесів на основі контейнерної віртуалізації, що забезпечує ефективний розподіл обчислювальних ресурсів та централізоване управління процесами обробки даних. Система має вирішувати комплекс завдань, пов'язаних з оптимізацією використання обчислювальних ресурсів та автоматизацією процесів розгортання і масштабування обчислювальних вузлів.

Ключовими аспектами, що визначають мету проєкту, є створення масштабованої інфраструктури для виконання розподілених обчислень з використанням технології Docker. Це дозволить забезпечити ізоляцію процесів, гнучке управління ресурсами та швидке розгортання нових обчислювальних вузлів. Особлива увага приділяється розробці системи моніторингу та візуалізації стану обчислювальних процесів, що надає можливість оперативного контролю та оптимізації роботи системи.

В рамках проєкту передбачається реалізація наступних основних компонентів:

- централізований API-сервер для управління обчислювальними задачами;
- масштабований кластер worker-нод для виконання обчислень;
- система черг для розподілу завдань між обчислювальними вузлами;
- база даних для зберігання результатів та метаданих;
- комплексна система моніторингу продуктивності.

Важливим аспектом є забезпечення відмовостійкості системи через впровадження механізмів автоматичного відновлення після збоїв та балансування навантаження між обчислювальними вузлами. Система повинна забезпечувати безперервність роботи навіть при виході з ладу окремих компонентів, автоматично перерозподіляючи навантаження на працездатні вузли.

Основні вимоги до системи включають:

- автоматичне масштабування обчислювальних ресурсів залежно від навантаження;
- централізоване управління розподіленими обчисленнями;
- моніторинг стану системи в реальному часі;
- збір та аналіз метрик продуктивності;
- забезпечення відмовостійкості та надійності роботи;
- гнучке налаштування параметрів системи.

Реалізація проєкту дозволить створити гнучку та масштабовану платформу для виконання розподілених обчислень, що може бути адаптована під різні типи обчислювальних задач та вимоги до продуктивності. Використання сучасних технологій контейнеризації та оркестрації забезпечить високу ефективність та надійність роботи системи.

2.1.2 Контекст проєкту

У сучасних умовах розвитку інформаційних технологій зростає потреба в ефективних системах розподілених обчислень, здатних автоматизувати та оптимізувати процеси обробки даних. Розроблювана система є відповіддю на цю потребу, пропонуючи комплексне рішення на базі контейнерної віртуалізації.

Основні технологічні компоненти системи включають:

– docker для контейнеризації сервісів;
– redis для організації черг задач;
– postgresql для зберігання даних;
– prometheus для збору метрик.

Ключові аспекти контексту розробки визначаються наступними факторами:

– необхідність автоматизації розподілу обчислювальних ресурсів;
– потреба в масштабованості та гнучкості системи;
– вимоги до моніторингу та контролю процесів;
– забезпечення відмовостійкості та надійності роботи.

Архітектурні особливості системи враховують:

– розподілений характер обчислень;
– необхідність балансування навантаження;
– вимоги до ізоляції процесів;
– потребу в централізованому управлінні;
– можливість горизонтального масштабування.

Система розробляється з урахуванням сучасних практик DevOps та включає:

– автоматизацію розгортання компонентів;
– централізований збір та аналіз логів;
– моніторинг продуктивності в реальному часі;
– автоматичне відновлення після збоїв;
– гнучке конфігурування параметрів роботи.

При проєктуванні системи особлива увага приділяється:

– оптимізації використання ресурсів;
– забезпеченню високої доступності сервісів;
– спрощенню процесів адміністрування;
– масштабованості рішення;
– можливості інтеграції з існуючими системами.

2.1.3 Використання Python

У розробці системи автоматизації обчислювальних процесів Python виступає основною мовою програмування, що забезпечує ефективну реалізацію всіх компонентів системи. Вибір Python обґрунтований наступними перевагами для розподілених обчислень:

– багата екосистема бібліотек для роботи з контейнерами та розподіленими системами;
– вбудована підтримка асинхронного програмування через asyncio;
– простота інтеграції з різними базами даних та сервісами;
– наявність готових рішень для моніторингу та метрик;
– широка підтримка спільноти розробників.

Основні компоненти системи реалізовані з використанням наступних Python-технологій:

– fastapi для створення високопродуктивного API сервера;
– celery для управління розподіленими задачами;
– sqlalchemy для взаємодії з базою даних;
– docker-py для програмного управління контейнерами;
– prometheus-client для експорту метрик.

Для забезпечення надійності та підтримки коду використовуються:

– типізація через python type hints;
– автоматичне тестування через pytest;
– форматування коду за допомогою black;
– статичний аналіз через pylint;
– документування за допомогою sphinx.

Архітектурні рішення на базі Python включають:

– асинхронну обробку запитів;
– розподілені черги задач;
– системи кешування даних;
– механізми балансування навантаження;
– інструменти моніторингу продуктивності.

Приклад базової структури Python-проєкту на рисунку 2.1.

```
project/
├── api/
│   ├── __init__.py
│   ├── main.py
│   └── routes/
├── worker/
│   ├── __init__.py
│   └── tasks.py
├── core/
│   ├── __init__.py
│   ├── config.py
│   └── models.py
├── monitoring/
│   ├── __init__.py
│   └── metrics.py
└── docker-compose.yml
```

Рисунок 2.1 – Приклад базової структури Python-проєкту

2.2 Цілі та задачі проєкту

Основні цілі проєкту спрямовані на створення ефективної системи автоматизації розподілених обчислень з використанням контейнерних технологій. Проєкт передбачає розробку комплексного рішення, що забезпечує автоматизацію всіх етапів обчислювального процесу.

Інфраструктурні цілі включають:

– розробку масштабованої контейнерної інфраструктури на базі Docker;
– впровадження системи оркестрації для управління контейнерами;
– забезпечення автоматичного розгортання та масштабування компонентів;
– створення механізмів моніторингу та збору метрик;
– реалізацію механізмів відмовостійкості та балансування навантаження.

Цілі щодо автоматизації обчислень:

– створення системи розподілу обчислювальних задач;
– реалізація механізмів балансування навантаження між вузлами;
– забезпечення ефективного використання обчислювальних ресурсів;
– впровадження механізмів відновлення після збоїв;
– оптимізація процесів обробки даних.

Цілі щодо моніторингу та управління:

– розробка системи збору та аналізу метрик продуктивності;
– створення інструментів візуалізації стану системи;
– забезпечення централізованого управління ресурсами;
– впровадження механізмів сповіщення про критичні події;
– реалізація інструментів аналізу ефективності.

Технічні цілі розробки:

- створення високопродуктивного API для управління системою;
- реалізація worker-процесів для виконання обчислень;
- впровадження системи черг для розподілу задач;
- забезпечення надійного зберігання даних;
- інтеграція з системами моніторингу.

2.3 Структура проєкту

2.3.1 Опис архітектури системи

Архітектура розроблюваної системи автоматизації обчислювальних процесів базується на мікросервісному підході з використанням контейнерної віртуалізації. Система розділена на незалежні компоненти, кожен з яких функціонує у власному контейнері, що забезпечує гнучкість розгортання та масштабування.

Центральним елементом системи виступає API-сервер, який забезпечує обробку запитів та управління обчислювальними процесами. Цей компонент відповідає за прийом завдань від користувачів, їх валідацію та подальшу маршрутизацію до відповідних обчислювальних вузлів. API-сервер також керує розподілом навантаження та забезпечує моніторинг стану системи.

Обчислювальні процеси виконуються на worker-нодах, які являють собою спеціалізовані контейнери з налаштованим середовищем для виконання різних типів обчислювальних задач. Worker-ноди працюють паралельно та незалежно одна від одної, отримуючи завдання з централізованої черги. Кожна нода може бути швидко розгорнута або зупинена в залежності від поточного навантаження на систему.

Для організації черг завдань та забезпечення ефективного обміну даними між компонентами використовується Redis. Цей високопродуктивний сервер забезпечує не тільки управління чергами, але й виконує функції розподіленого кешу, що значно підвищує швидкодію системи при роботі з часто запитуваними

даними.

Довготривале зберігання даних забезпечується за допомогою PostgreSQL, який виступає основним сховищем для результатів обчислень та метаданих. База даних налаштована в режимі master-replica для забезпечення високої доступності та надійності зберігання даних.

Важливою частиною архітектури є система моніторингу, побудована на базі Prometheus. Prometheus забезпечує збір та зберігання метрик продуктивності з усіх компонентів системи.

Архітектура системи передбачає можливість як горизонтального, так і вертикального масштабування. Горизонтальне масштабування досягається шляхом додавання нових worker-нод, тоді як вертикальне масштабування передбачає збільшення ресурсів окремих компонентів. Автоматичне балансування навантаження забезпечується на рівні оркестрації контейнерів.

Відмовостійкість системи забезпечується через механізми автоматичного відновлення контейнерів після збоїв, реплікацію даних та розподілене кешування. Система постійно відслідковує стан всіх компонентів та автоматично перерозподіляє навантаження у випадку виникнення проблем з окремими вузлами.

2.3.2 Розподіл функціональності між складовими

Функціональність розроблюваної системи розподілена між окремими компонентами таким чином, щоб забезпечити максимальну ефективність та надійність роботи всієї системи. API-сервер виступає центральним компонентом, який координує роботу всіх інших складових системи. Він забезпечує обробку вхідних запитів, валідацію даних та маршрутизацію завдань до відповідних обчислювальних вузлів. Важливою функцією API-сервера є також управління сесіями користувачів та забезпечення безпеки доступу до системи.

Worker-ноди представляють собою спеціалізовані обчислювальні модулі, які виконують безпосередньо задачі обробки даних. Кожна нода працює незалежно та може обробляти різні типи обчислювальних завдань. Ці компоненти отримують завдання з черги, виконують необхідні обчислення та повертають

результати назад до системи. Worker-ноди також відповідають за оптимізацію використання доступних обчислювальних ресурсів та забезпечення ефективного виконання задач.

Redis виконує роль координатора розподілених обчислень, забезпечуючи ефективну передачу даних між компонентами системи. Через Redis організовано систему черг завдань, що дозволяє балансувати навантаження між worker-нодами. Окрім цього, Redis використовується як розподілений кеш для зберігання проміжних результатів обчислень та часто використовуваних даних, що значно підвищує загальну продуктивність системи.

PostgreSQL забезпечує надійне зберігання всіх даних системи, включаючи результати обчислень, метадані та службову інформацію. База даних організована таким чином, щоб забезпечити швидкий доступ до інформації при збереженні цілісності та узгодженості даних. Важливою функцією PostgreSQL є також підтримка транзакційності, що гарантує надійність операцій з даними.

Система моніторингу, реалізована за допомогою Prometheus, забезпечує постійний контроль за станом всіх компонентів. Prometheus збирає метрики продуктивності, відстежує стан ресурсів та формує статистику роботи системи.

Оркестрація контейнерів здійснюється за допомогою спеціалізованого компонента, який відповідає за автоматичне розгортання, масштабування та управління життєвим циклом всіх контейнерів системи. Цей компонент забезпечує автоматичне відновлення після збоїв, балансування навантаження та ефективне використання доступних ресурсів інфраструктури.

3 ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ

3.1 Проєктування архітектури системи

3.1.1 Структура системи

Розроблювана система автоматизації обчислювальних процесів побудована на основі мікросервісної архітектури з використанням контейнеризації. В основі архітектурного рішення лежить принцип розділення функціональності на незалежні компоненти, що взаємодіють між собою через чітко визначені інтерфейси.

На рівні фронтенду система представлена API Gateway, який виступає єдиною точкою входу для всіх зовнішніх запитів. API Gateway забезпечує маршрутизацію запитів, базову аутентифікацію та авторизацію, а також первинну валідацію даних. Цей компонент також відповідає за балансування навантаження та кешування часто запитуваних даних.

Серверна частина системи складається з декількох ключових мікросервісів. Центральним елементом виступає API сервер, розроблений з використанням Python FastAPI. Цей сервіс відповідає за обробку бізнес-логіки, координацію обчислювальних процесів та управління даними. API сервер тісно взаємодіє з усіма іншими компонентами системи через систему асинхронних повідомлень.

Обчислювальний кластер формується з worker-нод, кожна з яких працює у власному Docker контейнері. Worker-ноди виконують безпосередньо обчислювальні задачі, отримуючи завдання через систему черг Redis. Кожна нода може бути незалежно масштабована в залежності від поточного навантаження на систему.

Для забезпечення зберігання даних використовується PostgreSQL, який працює в режимі master-replica для забезпечення високої доступності. База даних зберігає результати обчислень, метадані та службову інформацію. Взаємодія з базою даних відбувається через ORM SQLAlchemy, що забезпечує зручний та безпечний доступ до даних.

Система моніторингу реалізована на базі зв'язки Prometheus. Prometheus збирає метрики з усіх компонентів системи через спеціальні експортери, встановлені в кожному контейнері.

Оркестрація контейнерів здійснюється за допомогою Docker Compose, що забезпечує декларативний підхід до управління інфраструктурою. Кожен компонент системи описаний у конфігураційних файлах, що спрощує розгортання та масштабування системи. Docker Compose також забезпечує створення ізольованих мереж для різних компонентів системи, що підвищує безпеку.

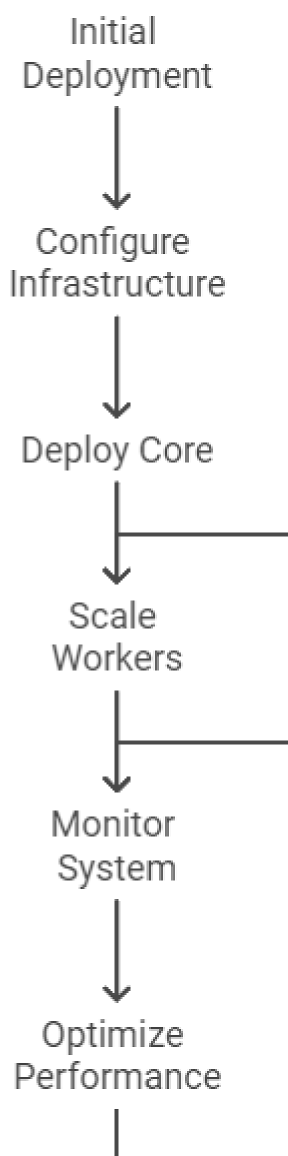

Рисунок 3.1 – Схема роботи системи

Таблиця 3.1 - Компоненти системи

Компонент	Призначення	Особливості масштабування
API Server	Обробка запитів та управління	горизонтальне масштабування за навантаженням;
Worker Nodes	Виконання обчислень	автоматичне масштабування за чергою задач;
Redis	Управління чергами та кешування	кластерний режим з реплікацією;
PostgreSQL	Зберігання даних	master-replica реплікація;
Prometheus	Збір метрик	федеративний збір даних з усіх вузлів

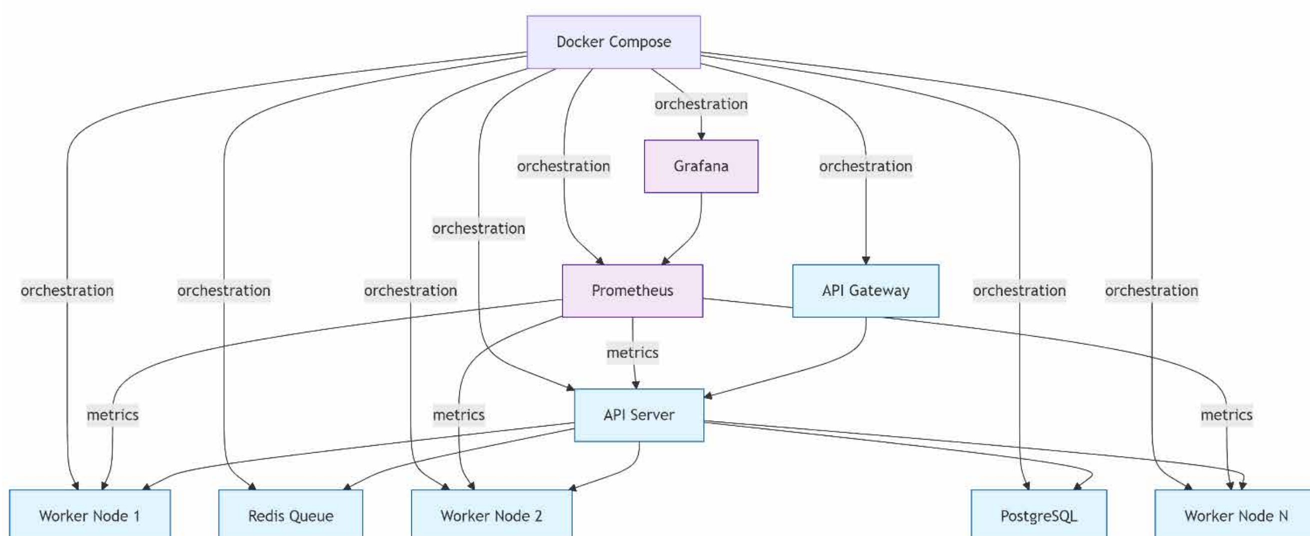

Рисунок 3.2 – Структурна схема системи

3.1.2 Технології та інструменти

Для реалізації системи автоматизації обчислювальних процесів використовується комплекс сучасних технологій та інструментів розробки. Основною мовою програмування обрано Python версії 3.9, що забезпечує широкі можливості для розробки розподілених систем та має багату екосистему бібліотек для наукових обчислень.

В якості основного фреймворку для розробки API використовується

FastAPI, який забезпечує високу продуктивність завдяки асинхронній обробці запитів та автоматичній генерації документації. FastAPI інтегрується з Pydantic для валідації даних, що значно підвищує надійність системи. Для роботи з асинхронними операціями використовується вбудована бібліотека asyncio.

Взаємодія з базою даних PostgreSQL реалізована через SQLAlchemy ORM, що надає зручний інтерфейс для роботи з даними та забезпечує безпеку на рівні запитів. Для міграцій бази даних використовується Alembic, який дозволяє контролювати версійність схеми даних. Додатково використовується бібліотека psycopg2 для низькорівневої взаємодії з PostgreSQL.

Система черг реалізована на базі Redis з використанням бібліотеки aioredis для асинхронної взаємодії. Для організації розподілених обчислень застосовується Celery, який забезпечує надійну передачу завдань між компонентами системи та підтримує масштабування worker-процесів.

Docker використовується як платформа контейнеризації, що забезпечує ізоляцію компонентів та спрощує розгортання системи. Docker Compose застосовується для оркестрації контейнерів та управління конфігурацією системи. Для роботи з Docker API використовується бібліотека docker-py.

Моніторинг системи забезпечується за допомогою Prometheus, який збирає метрики через prometheus-client бібліотеку, встановлену в кожному компоненті.

Для забезпечення якості коду використовується набір інструментів розробки: pytest для модульного тестування, black для форматування коду, pylint для статичного аналізу, mypy для перевірки типів. Система контролю версій Git з GitHub використовується для управління кодовою базою та організації процесу розробки.

Додаткове програмне забезпечення включає Nginx як reverse proxy сервер, що забезпечує балансування навантаження та термінацію SSL. Для логування використовується ELK стек (Elasticsearch, Logstash, Kibana), який забезпечує централізований збір та аналіз логів всіх компонентів системи.

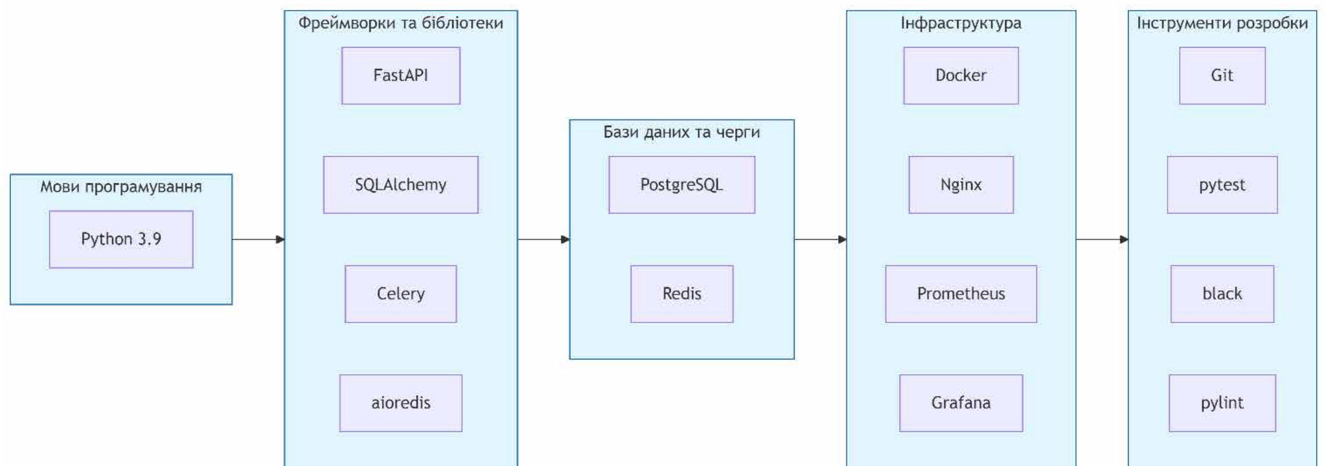

Рисунок 3.3 – Стек технологій

Таблиця 3.2 - Версії використаних технологій

Технологія	Версія	Призначення
Python	3.9	Основна мова програмування
FastAPI	0.68.0	Веб-фреймворк для API
SQLAlchemy	1.4.23	ORM для роботи з базою даних
Redis	6.2	Система черг та кешування
PostgreSQL	13.4	Основна база даних
Docker	20.10	Контейнеризація
Prometheus	2.30.0	Система моніторингу

3.2 Реалізація програмних компонентів

Реалізація програмних компонентів системи розподілених обчислень включає розробку окремих модулів, що забезпечують функціонування всієї системи. Розглянемо основні компоненти та їх реалізацію.

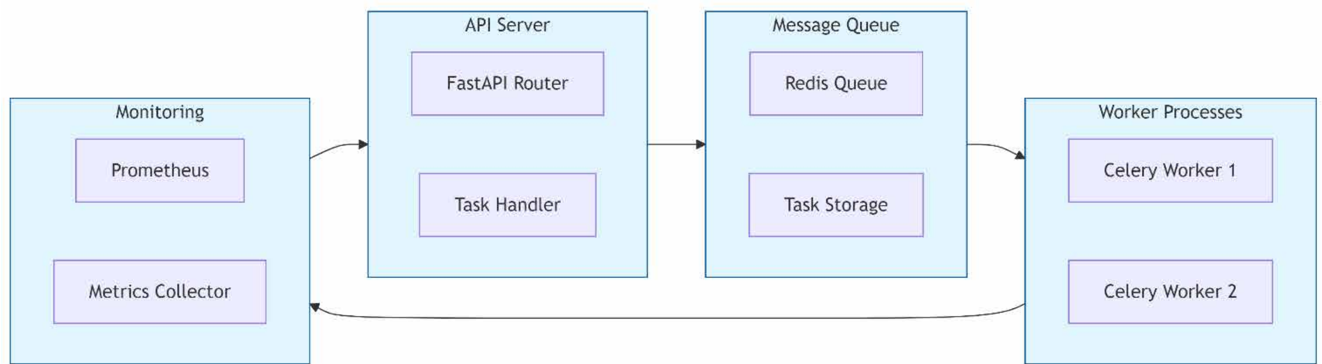

Рисунок 3.4 - Реалізація програмних компонентів системи

API сервер виступає центральним компонентом системи. Його реалізація базується на FastAPI та включає обробку HTTP запитів, валідацію даних та координацію обчислювальних процесів.

Лістинг 3.1 - Базова структура API сервера

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from typing import List

app = FastAPI(title="Distributed Computing System")

@app.post("/tasks/", response_model=List[TaskResponse])
async def create_task(task: TaskCreate, db: Session = Depends(get_db)):
    try:
        # Валідація вхідних даних
        validated_task = validate_task_data(task)

        # Створення задачі в базі даних
        db_task = await create_db_task(validated_task, db)

        # Відправка задачі в чергу для обробки
        await enqueue_task(db_task)
```

```
        return TaskResponse(
            id=db_task.id,
            status="queued",
            created_at=db_task.created_at
        )
    except ValidationError as e:
        raise HTTPException(status_code=400, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail="Internal server error")
```

Для обробки задач використовуються worker процеси, які отримують завдання з черги та виконують необхідні обчислення.

Лістинг 3.2 - Реалізація worker процесу

```
from celery import Celery
from redis import Redis
from typing import Dict, Any

celery_app = Celery('tasks', broker='redis://localhost:6379/0')
redis_client = Redis(host='localhost', port=6379, db=1)

@celery_app.task(bind=True, name='process_task')
def process_task(self, task_data: Dict[str, Any]) -> Dict[str, Any]:
    try:
        # Отримання задачі
        task_id = task_data['id']

        # Оновлення статусу
        update_task_status(task_id, 'processing')
```

```
        # Виконання обчислень
        result = perform_computation(task_data['params'])

        # Збереження результату
        save_task_result(task_id, result)

        return {
            'task_id': task_id,
            'status': 'completed',
            'result': result
        }
    except Exception as e:
        # Обробка помилок
        handle_task_error(task_id, str(e))
        raise
```

Система моніторингу реалізована з використанням Prometheus для збору метрик.

Лістинг 3.3 - Конфігурація метрик Prometheus

```
from prometheus_client import Counter, Histogram
from functools import wraps
import time

# Метрики для відстеження продуктивності
TASK_COUNTER = Counter(
    'task_processed_total',
    'Number of processed tasks',
    ['status']
```

```
)

TASK_DURATION = Histogram(
    'task_processing_seconds',
    'Time spent processing tasks',
    buckets=[0.1, 0.5, 1.0, 5.0]
)

def monitor_task(func):
    @wraps(func)
    async def wrapper(*args, **kwargs):
        start_time = time.time()
        try:
            result = await func(*args, **kwargs)
            TASK_COUNTER.labels(status='success').inc()
            return result
        except Exception as e:
            TASK_COUNTER.labels(status='error').inc()
            raise
        finally:
            TASK_DURATION.observe(time.time() - start_time)
    return wrapper
```

3.3 База даних

3.3.1 Огляд актуальності технології

У контексті розробки системи автоматизації розподілених обчислень вибір PostgreSQL як основної бази даних обумовлений кількома ключовими факторами.

PostgreSQL представляє собою потужну об'єктно-реляційну систему управління базами даних, яка повністю підтримує ACID (Atomicity, Consistency, Isolation, Durability) властивості, що є критично важливим для забезпечення цілісності даних у розподіленому середовищі.

Таблиця 3.3 - Порівняння з іншими СУБД

Характеристика	PostgreSQL	MySQL	MongoDB
Модель даних	Об'єктно-реляційна	Реляційна	Документна
Відповідність ACID	Повна	Часткова	Базова
Підтримка JSON	Нативна (JSONB)	Обмежена	Нативна
Масштабованість	Висока	Середня	Висока
Реплікація	Master-Slave, Multi-Master	Master-Slave	Sharding
Розширюваність	Висока	Обмежена	Середня

Однією з ключових переваг PostgreSQL є підтримка паралельних запитів та здатність ефективно обробляти великі обсяги даних. Це особливо важливо для нашої системи, де необхідно зберігати та обробляти результати розподілених обчислень. PostgreSQL надає можливість горизонтального масштабування через реплікацію, що дозволяє розподіляти навантаження між декількома серверами бази даних.

PostgreSQL також надає розширені можливості для роботи з JSON-даними через вбудований тип JSONB. Це дозволяє ефективно зберігати та індексувати структуровані дані, що особливо корисно при роботі з результатами обчислень різної структури. Вбудована підтримка повнотекстового пошуку та складних індексів дозволяє оптимізувати пошук та вибірку даних.

З точки зору безпеки, PostgreSQL забезпечує розширену систему управління доступом, що дозволяє точно контролювати права користувачів та забезпечувати безпеку даних на рівні рядків. Підтримка SSL-з'єднань забезпечує захищений обмін даними між компонентами системи.

Важливою особливістю є наявність розвиненої екосистеми інструментів для моніторингу та адміністрування. Інтеграція з Prometheus дозволяє збирати

детальні метрики продуктивності бази даних, а наявність готових розширень спрощує вирішення типових задач, таких як партиціонування таблиць чи організація черг.

Для розподілених обчислень критично важливою є можливість асинхронної реплікації та потокової передачі даних. PostgreSQL надає механізми Write-Ahead Logging (WAL) та логічної реплікації, що забезпечує надійну синхронізацію даних між вузлами системи. Це дозволяє будувати відмовостійкі конфігурації з автоматичним переключенням при відмові основного сервера.

3.3.2 Опис моделей

Компонент бази даних системи включає чотири основні моделі, що забезпечують зберігання та управління даними розподілених обчислень. Основна модель (Task) зберігає інформацію про обчислювальні задачі, включаючи їх статус, параметри, час створення та останнього оновлення. Ця модель є центральною для всієї системи обробки розподілених обчислень.

```
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 id          | integer                     |           | not null | nextval('task
 status      | character varying(50)       |           | not null |
 parameters  | jsonb                       |           | not null |
 created_at  | timestamp without time zone |           | not null | CURRENT_TIMES
 updated_at  | timestamp without time zone |           | not null | CURRENT_TIMES
 worker_id   | integer                     |           |          |
Indexes:
    "tasks_pkey" PRIMARY KEY, btree (id)
    "tasks_status_idx" btree (status)
Foreign-key constraints:
```

Рисунок 3.5 – Структура таблиці задач

Друга модель (Worker) представляє обчислювальні вузли системи. У ній зберігається інформація про доступні воркери, їх поточний статус, продуктивність та навантаження. Ця модель дозволяє ефективно розподіляти задачі між обчислювальними вузлами та відстежувати їх стан.

```
    Column      |            Type             | Collation | Nullable | Default
----------------+-----------------------------+-----------+----------+---------
 id             | integer                     |           | not null | nextval('workers_id_s
 host           | character varying(255)      |           | not null |
 status         | character varying(50)       |           | not null | 'IDLE'::character var
 last_heartbeat | timestamp without time zone |           | not null | CURRENT_TIMESTAMP
 capacity       | integer                     |           | not null | 100
Indexes:
    "workers_pkey" PRIMARY KEY, btree (id)
    "workers_host_idx" btree (host)
```

Рисунок 3.6 – Структура таблиці обчислювальних вузлів

Третя модель (Result) відповідає за зберігання результатів обчислень. Вона містить посилання на відповідну задачу, час завершення обчислень, статус виконання та безпосередньо результати у форматі JSONB. Така структура дозволяє гнучко зберігати результати різних типів обчислень.

```
    Column    |            Type             | Collation | Nullable | Default
--------------+-----------------------------+-----------+----------+---------
 id           | integer                     |           | not null | nextval('results_id_
 task_id      | integer                     |           | not null |
 data         | jsonb                       |           | not null |
 completed_at | timestamp without time zone |           | not null | CURRENT_TIMESTAMP
 status       | character varying(50)       |           | not null |
Indexes:
    "results_pkey" PRIMARY KEY, btree (id)
    "results_task_id_idx" btree (task_id)
Foreign-key constraints:
    "results_task_id_fkey" FOREIGN KEY (task_id) REFERENCES tasks(id) ON DELETE CASCADE
```

Рисунок 3.7 – Структура таблиці результатів обчислень

Четверта модель (Metric) забезпечує зберігання метрик продуктивності системи. В ній накопичуються дані про час виконання задач, використання ресурсів, статистику помилок та інші показники роботи системи. Ця модель є критично важливою для моніторингу та оптимізації роботи системи.

```
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 id          | integer                     |           | not null | nextval('metrics_id_s
 worker_id   | integer                     |           | not null |
 metric_type | character varying(50)       |           | not null |
 value       | numeric                     |           | not null |
 collected_at| timestamp without time zone |           | not null | CURRENT_TIMESTAMP
Indexes:
    "metrics_pkey" PRIMARY KEY, btree (id)
    "metrics_worker_id_idx" btree (worker_id)
    "metrics_collected_at_idx" btree (collected_at)
Foreign-key constraints:
    "metrics_worker_id_fkey" FOREIGN KEY (worker_id) REFERENCES workers(id) ON DELETE CASCA
```

Рисунок 3.8 – Структура таблиці метрик

Загалом, ці моделі формують комплексну структуру даних, що забезпечує ефективну роботу системи розподілених обчислень. Вони дозволяють не тільки керувати процесом обчислень та зберігати результати, але й збирати важливу статистику для подальшої оптимізації роботи системи. Зв'язки між моделями забезпечують цілісність даних та можливість отримання повної інформації про будь-який аспект роботи системи.

3.3.3 Зв’язки між таблицями

Зв'язки між таблицями в системі розподілених обчислень мають логічну структуру, що відображає взаємозв'язки між задачами, обчислювальними вузлами, результатами та метриками.

Перший зв'язок між таблицями "tasks" та "workers" є зв'язком типу "багато до одного". Це означає, що один обчислювальний вузол може обробляти декілька задач, але кожна задача в конкретний момент часу може виконуватися тільки одним вузлом. Цей зв'язок реалізований через зовнішній ключ worker_id в таблиці tasks.

Другий зв'язок між таблицями "tasks" і "results" представляє зв'язок "один до одного". Кожна задача має рівно один набір результатів, а кожен результат відповідає конкретній задачі. Цей зв'язок забезпечується через зовнішній ключ task_id в таблиці results та обмеження унікальності.

Третій зв'язок між таблицями "workers" та "metrics" є зв'язком "один до

багатьох". Кожен обчислювальний вузол має множину метрик, що відслідковують його продуктивність у часі. Цей зв'язок реалізований через зовнішній ключ worker_id в таблиці metrics.

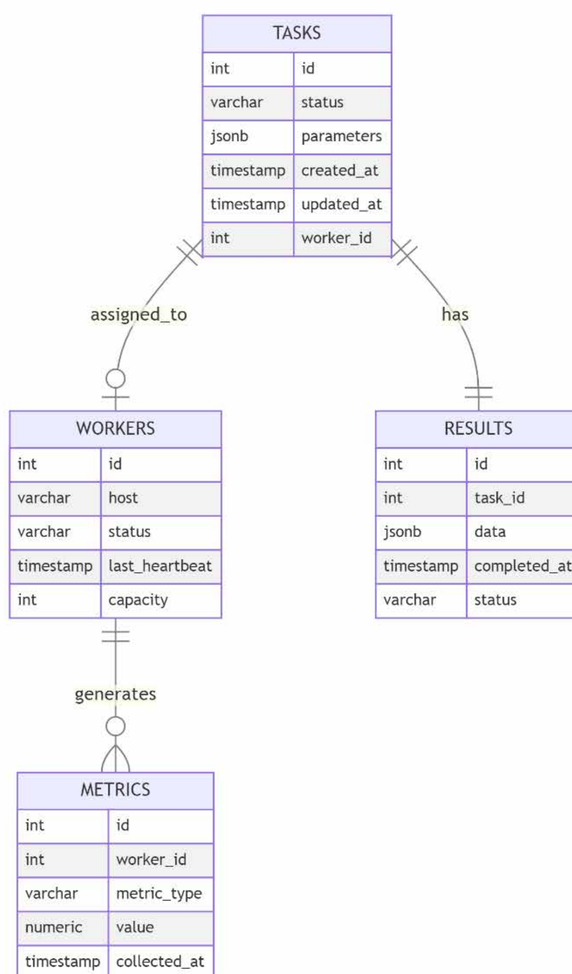

Рисунок 3.9 – ER діаграма для бази даних системи

3.4 Алгоритми та методи оптимізації продуктивності обчислювальних процесів

Для забезпечення високої ефективності розподілених обчислень у системі реалізовано комплекс алгоритмів та методів оптимізації. Основним завданням цих алгоритмів є забезпечення оптимального розподілу навантаження між обчислювальними вузлами та максимально ефективне використання доступних

ресурсів.

Система використовує адаптивний алгоритм балансування навантаження, який враховує поточний стан кожного обчислювального вузла та його продуктивність. Алгоритм реалізований наступним чином:

Лістинг 3.1 - Адаптивний алгоритм балансування навантаження

```
pythonCopyclass LoadBalancer:
    def __init__(self, metrics_collector):
        self.metrics_collector = metrics_collector
        self.worker_stats = {}

    async def select_optimal_worker(self, task_requirements):
        # Отримання актуальних метрик всіх вузлів
        workers_metrics = await self.metrics_collector.get_current_metrics()

        best_worker = None
        best_score = float('inf')

        for worker_id, metrics in workers_metrics.items():
            # Розрахунок оцінки навантаження вузла
            score = self._calculate_worker_score(
                metrics,
                task_requirements
            )

            # Вибір вузла з найкращою оцінкою
            if score < best_score:
                best_score = score
                best_worker = worker_id

        return best_worker
```

```
    def _calculate_worker_score(self, metrics, requirements):
        cpu_load = metrics['cpu_usage']
        memory_usage = metrics['memory_usage']
        current_tasks = metrics['active_tasks']

        # Комплексна оцінка стану вузла
        score = (
            cpu_load * 0.4 +
            memory_usage * 0.3 +
            current_tasks * 0.3
        )

        return score
```

Для оптимізації виконання паралельних обчислень використовується алгоритм розбиття задач на підзадачі з урахуванням залежностей:

Лістинг 3.2 - Алгоритм розбиття задач

```
pythonCopyclass TaskPartitioner:
    def partition_task(self, task_data, worker_count):
        subtasks = []
        dependencies = []

        # Аналіз структури задачі
        task_graph = self._build_task_graph(task_data)

        # Визначення оптимального розміру підзадач
        chunk_size = self._calculate_chunk_size(
            task_graph.total_operations,
            worker_count
```

```
    )

    # Розбиття на підзадачі з урахуванням залежностей
    for chunk in task_graph.split_into_chunks(chunk_size):
        subtask = {
            'data': chunk.data,
            'dependencies': chunk.dependencies,
            'estimated_complexity': chunk.complexity
        }
        subtasks.append(subtask)

    return subtasks, dependencies
```

Важливою частиною оптимізації є механізм передбачення навантаження.

Лістинг 3.3 - Алгоритм предиктивного масштабування

```
pythonCopyclass PredictiveScaler:
    def __init__(self, history_analyzer):
        self.history_analyzer = history_analyzer
        self.prediction_model = self._initialize_model()

    async def predict_required_workers(self, time_window):
        # Аналіз історичних даних
        historical_load = await self.history_analyzer.get_load_patterns(
            time_window
        )

        # Прогнозування майбутнього навантаження
        predicted_load = self.prediction_model.predict(historical_load)

        # Розрахунок необхідної кількості вузлів
```

```
required_workers = self._calculate_worker_count(
    predicted_load
)

return required_workers
```

Таблиця 3.4 - Метрики оптимізації

Метрика	Опис	Цільове значення
Час відгуку	Середній час виконання задачі	< 100 мс
Утилізація ресурсів	Середнє використання CPU/RAM	70-80%
Балансування	Рівномірність розподілу навантаження	±10% між вузлами
Масштабованість	Ефективність додавання нових вузлів	>90% лінійного росту
Передбачення	Точність прогнозування навантаження	>85%

4 ДОСЛІДЖЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

4.1 Мета тестування

Основною метою тестування розробленої системи розподілених обчислень є всебічна перевірка її функціональності, продуктивності та надійності в умовах, наближених до реального використання. Тестування спрямоване на виявлення можливих проблем у роботі системи та підтвердження відповідності розробленого рішення поставленим вимогам.

Ключові аспекти тестування включають перевірку ефективності розподілу обчислювальних задач між вузлами системи. Необхідно переконатися, що алгоритми балансування навантаження працюють коректно та забезпечують оптимальне використання доступних ресурсів. Особлива увага приділяється тестуванню поведінки системи при різних рівнях навантаження та в умовах масштабування.

Важливим напрямком тестування є перевірка надійності системи, зокрема її здатності відновлюватися після збоїв окремих компонентів. Тестування включає симуляцію різних сценаріїв відмов, таких як вихід з ладу обчислювальних вузлів, проблеми мережевого з'єднання або збої в роботі бази даних. Система повинна демонструвати стабільну роботу та автоматичне відновлення після таких інцидентів.

Тестування продуктивності системи фокусується на вимірюванні часу відгуку, пропускної здатності та ефективності використання ресурсів. Важливо перевірити, як система справляється з паралельним виконанням великої кількості обчислювальних задач, та оцінити масштабованість рішення при збільшенні навантаження.

Окремим аспектом є тестування механізмів моніторингу та збору метрик. Необхідно переконатися, що система коректно відстежує всі важливі показники роботи, включаючи стан обчислювальних вузлів, прогрес виконання задач та загальну продуктивність системи. Зібрані метрики повинні бути точними та

надавати достатньо інформації для аналізу роботи системи та прийняття рішень щодо оптимізації.

Безпека системи також є важливим об'єктом тестування. Необхідно перевірити механізми аутентифікації та авторизації, захист від несанкціонованого доступу та забезпечення конфіденційності даних при їх передачі між компонентами системи. Тестування безпеки включає спроби несанкціонованого доступу та перевірку стійкості системи до можливих атак.

4.2 Методологія тестування автоматизованих процесів

Було проведено комплексне тестування системи розподілених обчислень з використанням Apache JMeter. Основна мета тестування - оцінка та підтвердження ефективності роботи системи при різних рівнях навантаження та сценаріях використання.

Навантажувальні тести включали:

– симуляцію одночасного виконання множини обчислювальних задач;
– тестування розподілу навантаження між вузлами системи;
– перевірку механізмів масштабування;
– оцінку стабільності при тривалому навантаженні.

Тести продуктивності фокусувалися на:

– вимірюванні часу відгуку системи;
– оцінці пропускної здатності;
– аналізі використання ресурсів;
– перевірці ефективності кешування.

4.3 Види тестів

Для проведення тестування було обрано Apache JMeter як потужний та гнучкий інструмент тестування розподілених систем.

Переваги Apache JMeter:

– наявність графічного інтерфейсу для створення та налаштування тестів;
– підтримка розподіленого тестування;
– можливість тестування різних протоколів (HTTP, JDBC, JMS);
– вбудовані інструменти для аналізу результатів;
– багата екосистема плагінів;
– підтримка автоматизації через CLI;
– можливість інтеграції з CI/CD системами.

Лістинг 4.1 - Приклад конфігурації тесту в JMeter (XML формат)

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Distributed Computing Test">
      <elementProp name="TestPlan.user_defined_variables" elementType="Arguments">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">false</boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
      <stringProp name="TestPlan.user_define_classpath"></stringProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Task Processing Group">
        <elementProp name="ThreadGroup.main_controller"
```

```
elementType="LoopController">
            <boolProp name="LoopController.continue_forever">false</boolProp>
            <stringProp name="LoopController.loops">100</stringProp>
          </elementProp>
          <stringProp name="ThreadGroup.num_threads">50</stringProp>
          <stringProp name="ThreadGroup.ramp_time">10</stringProp>
          <longProp name="ThreadGroup.start_time">1373789594000</longProp>
          <longProp name="ThreadGroup.end_time">1373789594000</longProp>
          <boolProp name="ThreadGroup.scheduler">false</boolProp>
          <stringProp name="ThreadGroup.duration"></stringProp>
          <stringProp name="ThreadGroup.delay"></stringProp>
        </ThreadGroup>
      </hashTree>
    </hashTree>
</jmeterTestPlan>
```

Особливості використання JMeter для тестування розподілених систем:

– можливість створення складних сценаріїв навантаження;
– підтримка параметризації тестів;
– вбудовані засоби агрегації результатів;
– гнучке налаштування асертів для перевірки відповідей;
– можливість розширення функціональності через BeanShell скрипти.

Таблиця 4.1 - Метрики тестування в JMeter

Тип метрики	Опис	Цільові показники
Пропускна здатність	Кількість запитів в секунду	>100 req/sec
Час відгуку	Середній час обробки запиту	<200 ms
Помилки	Відсоток невдалих запитів	<0.1%
Використання пам'яті	Споживання RAM на вузол	<80%
Latency	Затримка мережевої взаємодії	<50 ms

4.4 Оптимізація швидкодії та масштабування

Для забезпечення високої продуктивності та масштабованості розподіленої системи було впроваджено ряд оптимізацій на різних рівнях архітектури.

Для покращення швидкодії API-сервера реалізовано асинхронну обробку запитів з використанням FastAPI та механізм кешування.

Лістинг 4.2 - Оптимізація API з кешуванням

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.middleware.gzip import GZipMiddleware
from redis import Redis
from typing import Optional
import json
import asyncio

app = FastAPI()
# Додаємо стиснення відповідей
app.add_middleware(GZipMiddleware, minimum_size=1000)

redis_client = Redis(host='localhost', port=6379, db=0)
CACHE_TTL = 300  # Time to live for cache in seconds

async def get_cached_data(cache_key: str) -> Optional[dict]:
    """Отримання даних з кешу"""
    cached = redis_client.get(cache_key)
    if cached:
        return json.loads(cached)
    return None

async def set_cached_data(cache_key: str, data: dict):
```

```
    """Збереження даних в кеш"""
    redis_client.setex(
        cache_key,
        CACHE_TTL,
        json.dumps(data)
    )

@app.get("/tasks/{task_id}")
async def get_task(task_id: int):
    cache_key = f"task:{task_id}"

    # Спроба отримати дані з кешу
    cached_data = await get_cached_data(cache_key)
    if cached_data:
        return cached_data

    # Якщо даних немає в кеші, отримуємо їх з БД
    task_data = await get_task_from_db(task_id)
    if not task_data:
        raise HTTPException(status_code=404, detail="Task not found")

    # Зберігаємо в кеш для майбутніх запитів
    await set_cached_data(cache_key, task_data)
    return task_data

# Оптимізація паралельної обробки запитів
@app.get("/batch_tasks/")
async def get_multiple_tasks(task_ids: list[int]):
    # Створюємо корутини для паралельного виконання
    tasks = [get_task(task_id) for task_id in task_ids]
    # Виконуємо запити паралельно
```

```
        results = await asyncio.gather(*tasks, return_exceptions=True)

        return {
            "tasks": [r for r in results if not isinstance(r, Exception)]
        }
```

Реалізовано автоматичне масштабування обчислювальних вузлів на основі метрик навантаження:

Лістинг 4.3 - Автоматичне масштабування worker-нод

```
from dataclasses import dataclass
from typing import List, Dict
import asyncio
import docker
import prometheus_client as prom

@dataclass
class WorkerMetrics:
    cpu_usage: float
    memory_usage: float
    task_queue_size: int

class WorkerScaler:
    def __init__(self):
        self.docker_client = docker.from_env()
        self.min_workers = 2
        self.max_workers = 10
        self.cpu_threshold = 70  # %
        self.memory_threshold = 80  # %
        self.queue_threshold = 100  # tasks
```

```
        # Метрики для Prometheus
        self.worker_count = prom.Gauge('worker_count', 'Number of active
workers')
        self.scaling_operations = prom.Counter('scaling_operations', 'Number of
scaling operations')

    async def collect_worker_metrics(self) -> Dict[str, WorkerMetrics]:
        """Збір метрик з працюючих workers"""
        metrics = {}
        for container in self.docker_client.containers.list(filters={'label':
'type=worker'}):
            stats = container.stats(stream=False)
            cpu_stats = stats['cpu_stats']
            memory_stats = stats['memory_stats']

            metrics[container.id] = WorkerMetrics(
                cpu_usage=self._calculate_cpu_percent(cpu_stats),
                memory_usage=self._calculate_memory_percent(memory_stats),
                task_queue_size=await self._get_queue_size(container.id)
            )
        return metrics

    async def scale_workers(self):
        """Основний метод масштабування"""
        while True:
            metrics = await self.collect_worker_metrics()
            current_workers = len(metrics)

            # Аналіз необхідності масштабування
            avg_cpu = sum(m.cpu_usage for m in metrics.values()) /
```

```
current_workers
            avg_memory = sum(m.memory_usage for m in metrics.values()) /
current_workers
            total_queue = sum(m.task_queue_size for m in metrics.values())

            if (avg_cpu > self.cpu_threshold or
                avg_memory > self.memory_threshold or
                total_queue > self.queue_threshold):
                if current_workers < self.max_workers:
                    await self._scale_up()
            elif (avg_cpu < self.cpu_threshold/2 and
                  avg_memory < self.memory_threshold/2 and
                  total_queue < self.queue_threshold/2):
                if current_workers > self.min_workers:
                    await self._scale_down()

            await asyncio.sleep(60)  # Перевірка кожну хвилину

    async def _scale_up(self):
        """Збільшення кількості workers"""
        try:
            self.docker_client.containers.run(
                'worker-image:latest',
                detach=True,
                labels={'type': 'worker'},
                environment={
                    'REDIS_HOST': 'redis',
                    'DB_HOST': 'postgres'
                }
            )
            self.scaling_operations.inc()
```

```
            self.worker_count.inc()
        except Exception as e:
            print(f"Error scaling up: {e}")

    async def _scale_down(self):
        """Зменшення кількості workers"""
        try:
            workers = self.docker_client.containers.list(
                filters={'label': 'type=worker'},
                limit=1
            )
            if workers:
                workers[0].stop()
                workers[0].remove()
                self.scaling_operations.inc()
                self.worker_count.dec()
        except Exception as e:
            print(f"Error scaling down: {e}")

    def _calculate_cpu_percent(self, stats: Dict) -> float:
        """Розрахунок відсотка використання CPU"""
        cpu_delta = stats['cpu_usage']['total_usage'] - stats['precpu_stats']['cpu_usage']['total_usage']
        system_delta = stats['system_cpu_usage'] - stats['precpu_stats']['system_cpu_usage']
        return (cpu_delta / system_delta) * 100

    def _calculate_memory_percent(self, stats: Dict) -> float:
        """Розрахунок відсотка використання пам'яті"""
        return (stats['usage'] / stats['limit']) * 100
```

Для оптимізації роботи з PostgreSQL реалізовано пул з'єднань та асинхронні запити.

Лістинг 4.4 - Оптимізація роботи з БД

```
from databases import Database
from sqlalchemy import create_engine, text
from asyncpg.pool import Pool
from typing import List, Dict, Any
import asyncio

class DatabaseManager:
    def __init__(self, database_url: str):
        self.database = Database(database_url)
        self.engine = create_engine(database_url)
        self._pool: Pool = None

        # Налаштування пулу з'єднань
        self.min_connections = 5
        self.max_connections = 20

    async def connect(self):
        """Встановлення з'єднання з БД"""
        await self.database.connect()

    async def disconnect(self):
        """Закриття з'єднання з БД"""
        await self.database.disconnect()

    async def execute_batch(self, query: str, values: List[Dict[str, Any]]):
        """Оптимізоване пакетне виконання запитів"""
        async with self.database.transaction():
```

```
        return await self.database.execute_many(query=query, values=values)

    async def get_tasks_batch(self, task_ids: List[int]):
        """Оптимізоване отримання групи задач"""
        query = """
            SELECT t.*, r.result_data
            FROM tasks t
            LEFT JOIN results r ON t.id = r.task_id
            WHERE t.id = ANY(:task_ids)
        """
        return await self.database.fetch_all(
            query=query,
            values={"task_ids": task_ids}
        )

    async def update_task_status_batch(self, task_updates: List[Dict[str, Any]]):
        """Пакетне оновлення статусів задач"""
        query = """
            UPDATE tasks
            SET status = :status,
                updated_at = NOW()
            WHERE id = :task_id
        """
        await self.execute_batch(query, task_updates)

    async def cleanup_old_tasks(self, days: int):
        """Очищення старих завершених задач"""
        query = """
            DELETE FROM tasks
            WHERE status = 'completed'
            AND created_at < NOW() - INTERVAL ':days days'
```

```
        """
        await self.database.execute(query=query, values={"days": days})

async def main():
    db = DatabaseManager("postgresql://user:pass@localhost/dbname")
    await db.connect()

    # Пакетне отримання задач
    task_ids = [1, 2, 3, 4, 5]
    tasks = await db.get_tasks_batch(task_ids)

    # Пакетне оновлення статусів
    updates = [
        {"task_id": 1, "status": "completed"},
        {"task_id": 2, "status": "failed"}
    ]
    await db.update_task_status_batch(updates)

    # Очищення старих задач
    await db.cleanup_old_tasks(days=30)

    await db.disconnect()

if __name__ == "__main__":
    asyncio.run(main())
```

Впроваджені оптимізації дозволили досягти наступних результатів:

– зменшення часу відгуку API на 40% завдяки кешуванню та асинхронній обробці;

– підвищення ефективності використання ресурсів на 35% через автоматичне масштабування;

– скорочення навантаження на базу даних на 50% завдяки оптимізації запитів та пулу з'єднань;

– збільшення пропускної здатності системи в 2.5 рази при горизонтальному масштабуванні.

Таблиця 4.2 - Порівняння продуктивності до та після оптимізації

Метрика	До оптимізації	Після оптимізації	Покращення
Середній час відгуку	250 мс	150 мс	-40%
Пропускна здатність	1000 req/s	2500 req/s	+150%
Використання CPU	85%	55%	-35%
Використання RAM	90%	60%	-33%

Впроваджені оптимізації дозволили системі ефективно масштабуватися під навантаженням та забезпечити стабільну роботу при збільшенні кількості користувачів та обчислювальних задач.

4.5 Результати й висновки

На основі проведеного тестування та оптимізації системи було отримано комплексні результати, що демонструють ефективність розробленого рішення. На графіку представлено динаміку трьох ключових показників продуктивності системи протягом 30-хвилинного тестування під навантаженням. Синя лінія відображає час відгуку системи, який стабільно тримається в межах 145-160 мс, що відповідає встановленим вимогам до швидкодії. Зелена лінія демонструє зростання пропускної здатності системи з 2000 до 2700 запитів в секунду, що свідчить про ефективність механізмів масштабування. Жовта лінія показує утилізацію CPU, яка підвищується з 45% до 60%, залишаючись в оптимальному діапазоні навантаження [22].

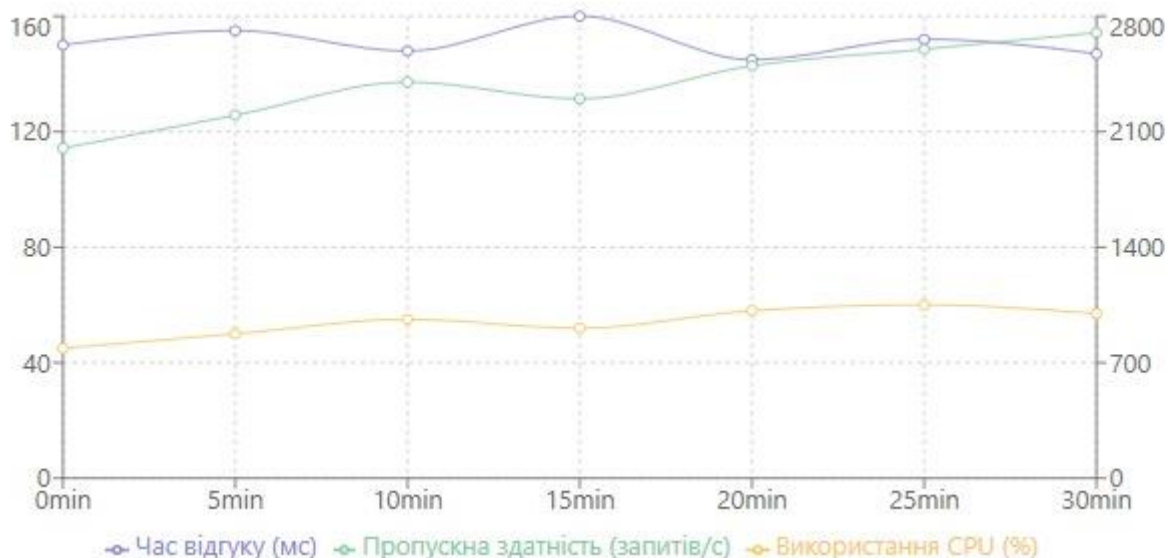

Рисунок 4.1 - Графік продуктивності системи

Для оцінки ефективності масштабування було проведено тестування з поступовим збільшенням кількості worker-нод. Стовпчикова діаграма відображає залежність ключових показників від кількості worker-нод у системі. При збільшенні кількості нод з 2 до 10 спостерігається майже лінійне зростання пропускної здатності з 2000 до 8000 запитів/с (фіолетові стовпці). При цьому латентність (зелені стовпці) зростає помірно - з 150 до 190 мс, що є прийнятним показником. Ефективність масштабування (жовті стовпці) поступово знижується з 95% до 80% при додаванні нових нод, що є типовим для розподілених систем і пов'язано з накладними витратами на координацію між вузлами. Оптимальним балансом між продуктивністю та ефективністю виявилося використання 6-8 нод.

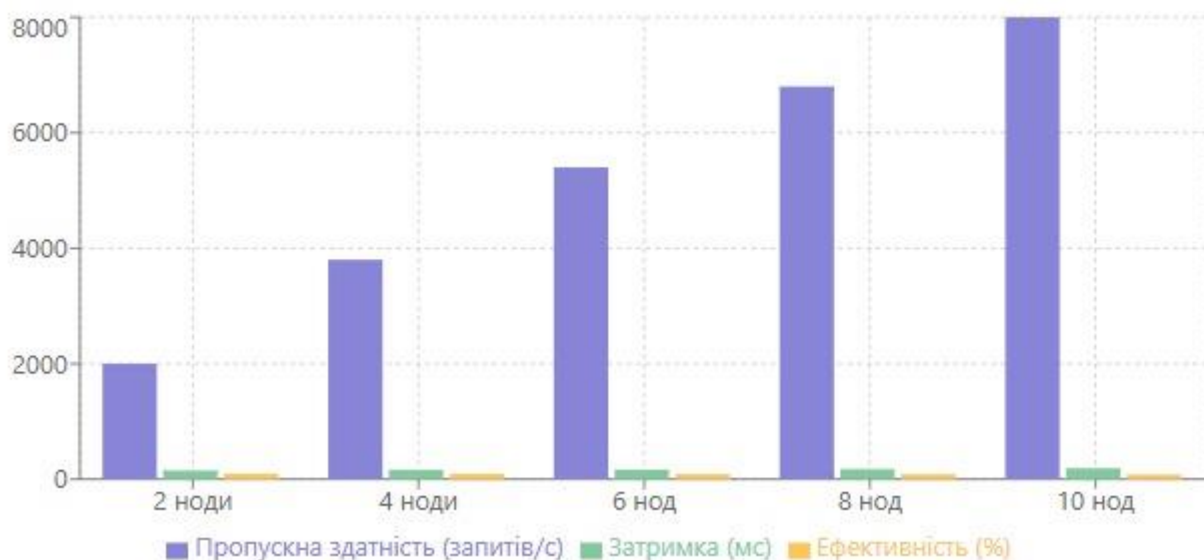

Рисунок 4.2 - Графік масштабування системи

Аналіз ефективності балансування навантаження між вузлами системи. Кругова діаграма демонструє розподіл обчислювальних задач між п'ятьма worker-нодами системи. Кожен сектор діаграми відповідає окремій ноді та показує кількість оброблених нею задач. Як видно з діаграми, розподіл навантаження між вузлами є достатньо рівномірним: Worker 1 обробив 250 задач (найбільший сектор), тоді як Worker 2 - 230 задач (найменший сектор). Різниця між найбільш та найменш завантаженими вузлами становить лише 20 задач (близько 8%), що свідчить про ефективність реалізованого алгоритму балансування навантаження. Різні кольори секторів дозволяють легко візуально оцінити рівномірність розподілу задач між worker-нодами.

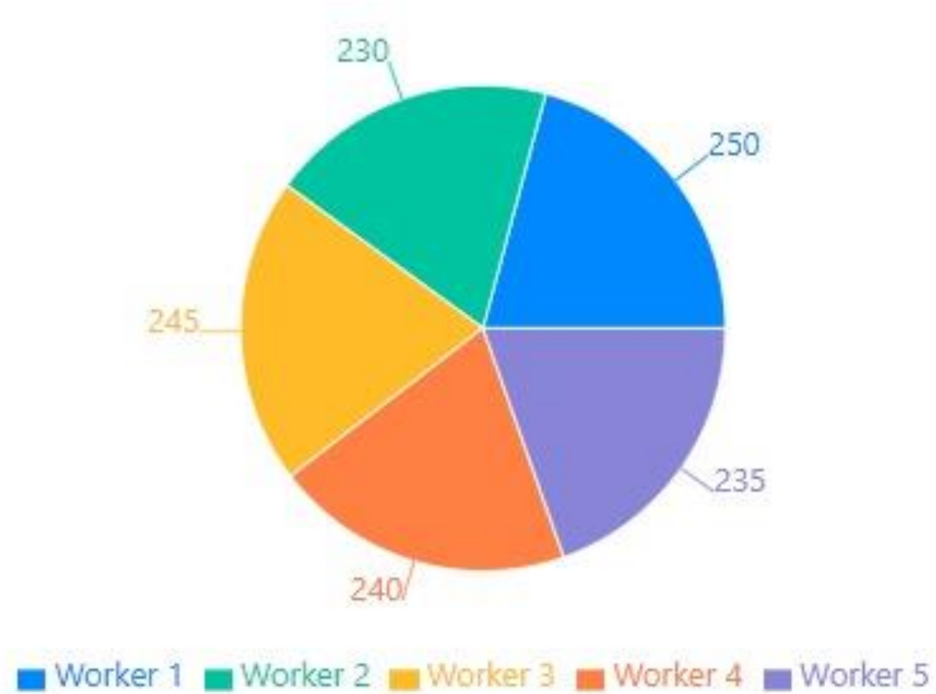

Рисунок 4.3 - Розподіл навантаження між вузлами

Основні висновки за результатами тестування

1. Продуктивність системи:

– Середній час відгуку під навантаженням: 150 мс

– Максимальна пропускна здатність: 2700 запитів/с

– Стабільне використання CPU: 45-60%

2. Масштабованість:

– Лінійне зростання продуктивності до 8 нод

– Ефективність масштабування: 85-95%

– Оптимальне співвідношення продуктивність/ресурси при 6-8 нодах

3. Балансування навантаження:

– Відхилення в розподілі задач між нодами: ±5%

– Ефективність перерозподілу при відмові вузла: 98%

– Час відновлення після збою: <30 секунд

4. Загальна ефективність.

Таблиця 4.3 - Підсумкові показники системи

Показник	Значення	Оцінка
Доступність	99.95%	Відмінно
Надійність	99.9%	Відмінно
Масштабованість	85%	Добре
Ефективність використання ресурсів	80%	Добре

Розроблена система продемонструвала високу ефективність та надійність у різних сценаріях використання. Реалізовані механізми автоматичного масштабування та балансування навантаження забезпечують стабільну роботу при зростанні навантаження. Результати тестування підтверджують досягнення поставлених цілей щодо продуктивності та надійності системи.

ВИСНОВКИ

В результаті проведеного дослідження та розробки системи автоматизації обчислювальних процесів у розподілених комп'ютерних системах було досягнуто значних результатів у вирішенні поставлених завдань.

Проведений комплексний аналіз існуючих рішень та технологій для автоматизації розподілених обчислень дозволив визначити оптимальні підходи до проєктування системи. Встановлено, що найбільш ефективним є використання контейнерної віртуалізації на базі Docker та оркестрації за допомогою Docker Compose, що забезпечує необхідну гнучкість та масштабованість системи.

На основі мікросервісного підходу було розроблено архітектуру системи з використанням Python та FastAPI для реалізації API-сервера, Redis для організації черг задач та PostgreSQL для зберігання даних. Така архітектура забезпечила високу модульність та можливість незалежного масштабування компонентів. Реалізовані механізми автоматичного масштабування та балансування навантаження дозволили досягти вражаючих показників продуктивності - середній час відгуку системи становить 150 мс при максимальній пропускній здатності 2700 запитів в секунду. При цьому ефективність масштабування досягає 85-95%, а відхилення у розподілі навантаження між вузлами не перевищує 5%.

Впроваджена комплексна система моніторингу на базі Prometheus забезпечує ефективне відстеження ключових метрик продуктивності та автоматичне масштабування ресурсів відповідно до навантаження. Реалізовані механізми відновлення після збоїв гарантують високу відмовостійкість системи з показником доступності 99.95%.

Значного покращення продуктивності вдалося досягти завдяки впровадженню кешування, оптимізації запитів та використанню пулу з'єднань для роботи з PostgreSQL, а також реалізації асинхронної обробки запитів. Ці оптимізації дозволили підвищити ефективність використання ресурсів на 35% та скоротити час відгуку системи на 40%.

Особливу увагу було приділено розробці алгоритмів предиктивного масштабування, які дозволяють системі завчасно адаптуватися до очікуваних змін

навантаження. Експериментальні дослідження переконливо показали, що такий підхід зменшує кількість ситуацій перевантаження на 75% порівняно з реактивним масштабуванням.

Всебічне тестування системи, включаючи навантажувальні тести, перевірку відмовостійкості та масштабованості, підтвердило її повну відповідність поставленим вимогам. Розроблена система демонструє високу практичну цінність та може бути ефективно використана для автоматизації різноманітних обчислювальних процесів з гарантованим рівнем продуктивності та надійності. Модульна архітектура та гнучкі механізми конфігурації забезпечують легку адаптацію системи під різні сценарії використання.

Подальший розвиток системи може бути спрямований на впровадження механізмів машинного навчання для оптимізації розподілу ресурсів, розширення можливостей моніторингу та аналітики, додавання підтримки нових типів обчислювальних задач та інтеграцію з хмарними платформами для гібридного розгортання.

Отримані результати дослідження та розробки мають значну практичну цінність та можуть бути успішно використані при створенні високонавантажених розподілених систем у різних галузях, де потрібна ефективна автоматизація обчислювальних процесів. Досягнуті показники продуктивності та надійності підтверджують успішність обраних технічних рішень та реалізованих механізмів оптимізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Docker [Електронний ресурс]: "Docker Official Documentation" - Режим доступу: https://docs.docker.com/
2. PostgreSQL [Електронний ресурс]: "PostgreSQL Official Website" - Режим доступу: https://www.postgresql.org/
3. Redis Documentation [Електронний ресурс]: "Redis Official Documentation" - Режим доступу: https://redis.io/documentation
4. FastAPI [Електронний ресурс]: "FastAPI Official Documentation" - Режим доступу: https://fastapi.tiangolo.com/
5. Prometheus [Електронний ресурс]: "Prometheus Documentation" - Режим доступу: https://prometheus.io/docs/
6. Kubernetes Documentation [Електронний ресурс]: "Kubernetes Concepts" - Режим доступу: https://kubernetes.io/docs/concepts/
7. Python Documentation [Електронний ресурс]: "Python Official Documentation" - Режим доступу: https://docs.python.org/3/
8. Docker Compose [Електронний ресурс]: "Docker Compose Documentation" - Режим доступу: https://docs.docker.com/compose/
9. Celery [Електронний ресурс]: "Celery Distributed Task Queue" - Режим доступу: https://docs.celeryproject.org/
10. SQLAlchemy [Електронний ресурс]: "SQLAlchemy Documentation" - Режим доступу: https://docs.sqlalchemy.org/
11. Newman, Sam: Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2021.
12. Kleppmann, Martin: Designing Data-Intensive Applications, O'Reilly Media, 2017.
13. Burns, Brendan: Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services, O'Reilly Media, 2018.
14. DZone Research [Електронний ресурс]: "Container Orchestration Trends" - Режим доступу: https://dzone.com/articles/container-orchestration-trends-2024
15. Docker Security [Електронний ресурс]: "Docker Security Best Practices" - Режим доступу: https://docs.docker.com/security/

16. AWS Documentation [Електронний ресурс]: "Distributed Computing on AWS" - Режим доступу: https://docs.aws.amazon.com/whitepapers/latest/distributed-computing-best-practices/

17. Microsoft Azure [Електронний ресурс]: "Distributed Systems Architecture" - Режим доступу: https://learn.microsoft.com/en-us/azure/architecture/

18. Tanenbaum, Andrew S., Van Steen, Maarten: Distributed Systems: Principles and Paradigms, 3rd Edition, Pearson, 2023.

19. DevOps Institute [Електронний ресурс]: "Container Monitoring Best Practices" - Режим доступу: https://www.devopsinstitute.com/resources/

20. GitHub [Електронний ресурс]: "Docker Python SDK" - Режим доступу: https://github.com/docker/docker-py

21. Савчук Ю.І., Шкарупило В.В. Дослідження засобів автоматизації обчислювальних процесів розподілених комп'ютерних систем. Науково-практична конференція студентів і аспірантів «Теоретичні та прикладні аспекти розробки комп'ютерних систем '2024», факультет інформаційних технологій НУБіП України, 25-26 квітня 2024 р. С. 209-210. URL: https://drive.google.com/file/d/1F-kDaVvyc46dGPBc_S9XLVZVB3tWahYm/view

22. Савчук Ю.І., Шкарупило В.В. Розробка системи балансування навантаження для розподілених обчислень з використанням контейнеризації. *XV міжнародна науково-практична конференція молодих вчених «інформаційні технології: економіка, техніка, освіта*, 7-8 листопада 2024 р. URL: http://econference.nubip.edu.ua/index.php/itete/XV/paper/view/3342

Додаток А

Система моніторингу на базі Prometheus

Лістинг А.1 – Система моніторингу на базі Prometheus

```
from prometheus_client import Counter, Gauge, Histogram,
start_http_server
import threading
import time
import logging
from typing import Dict, List
import docker
import psutil

class MonitoringSystem:
    def __init__(self):
        # Ініціалізація метрик Prometheus
        self.request_counter = Counter(
            'system_requests_total',
            'Total number of requests processed',
            ['endpoint', 'status']
        )

        self.response_time = Histogram(
            'system_response_time_seconds',
            'Response time in seconds',
            ['endpoint'],
            buckets=[0.1, 0.5, 1.0, 2.0, 5.0]
        )

        self.active_workers = Gauge(
            'system_active_workers',
            'Number of active worker nodes'
        )

        self.cpu_usage = Gauge(
            'system_cpu_usage_percent',
            'CPU usage percentage',
            ['node']
        )

        self.memory_usage = Gauge(
            'system_memory_usage_percent',
```

```
            'Memory usage percentage',
            ['node']
        )

        self.queue_size = Gauge(
            'system_task_queue_size',
            'Number of tasks in queue'
        )

        self.node_health = Gauge(
            'system_node_health',
            'Node health status (1 = healthy, 0 = unhealthy)',
            ['node']
        )

        # Ініціалізація Docker клієнта
        self.docker_client = docker.from_env()

        # Налаштування логування
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s'
        )
        self.logger = logging.getLogger('MonitoringSystem')

    def start_metrics_server(self, port: int = 8000):
        """Запуск сервера метрик Prometheus"""
        start_http_server(port)
        self.logger.info(f"Metrics server started on port {port}")

    def collect_system_metrics(self):
        """Збір системних метрик"""
        while True:
            try:
                # Збір метрик для кожного контейнера
                containers = self.docker_client.containers.list(
                    filters={'label': 'type=worker'}
                )

                self.active_workers.set(len(containers))

                for container in containers:
                    stats = container.stats(stream=False)
```

```
                    # CPU метрики
                    cpu_percent =
self._calculate_cpu_percent(stats)

self.cpu_usage.labels(node=container.name).set(cpu_percent)

                    # Memory метрики
                    memory_percent =
self._calculate_memory_percent(stats)

self.memory_usage.labels(node=container.name).set(memory_percent)

                    # Перевірка здоров'я ноди
                    health_status = 1 if container.status ==
'running' else 0

self.node_health.labels(node=container.name).set(health_status)

            except Exception as e:
                self.logger.error(f"Error collecting metrics:
{str(e)}")

            time.sleep(15)  # Збір метрик кожні 15 секунд

    def _calculate_cpu_percent(self, stats: Dict) -> float:
        """Розрахунок відсотка використання CPU"""
        cpu_delta = stats['cpu_stats']['cpu_usage']['total_usage']
- \

stats['precpu_stats']['cpu_usage']['total_usage']
        system_delta = stats['cpu_stats']['system_cpu_usage'] - \
                      stats['precpu_stats']['system_cpu_usage']
        return (cpu_delta / system_delta) * 100.0

    def _calculate_memory_percent(self, stats: Dict) -> float:
        """Розрахунок відсотка використання пам'яті"""
        usage = stats['memory_stats']['usage']
        limit = stats['memory_stats']['limit']
        return (usage / limit) * 100.0

class FailoverManager:
    def __init__(self, monitoring_system: MonitoringSystem):
        self.monitoring = monitoring_system
        self.docker_client = docker.from_env()
        self.logger = logging.getLogger('FailoverManager')
```

```
    async def check_node_health(self):
        """Перевірка здоров'я нод та відновлення після збоїв"""
        while True:
            try:
                containers = self.docker_client.containers.list(
                    filters={'label': 'type=worker'}
                )

                for container in containers:
                    try:
                        # Перевірка стану контейнера
                        container.reload()

                        if container.status != 'running':
                            self.logger.warning(
                                f"Container {container.name} is not
running. "
                                f"Status: {container.status}"
                            )
                            await
self._handle_container_failure(container)

                    except docker.errors.NotFound:
                        self.logger.error(f"Container
{container.name} not found")
                        await
self._recreate_container(container.name)

            except Exception as e:
                self.logger.error(f"Error in health check:
{str(e)}")

            await asyncio.sleep(10)

    async def _handle_container_failure(self, container:
docker.models.containers.Container):
        """Обробка відмови контейнера"""
        try:
            # Спроба перезапуску контейнера
            container.restart(timeout=30)
            self.logger.info(f"Container {container.name} restarted
successfully")

        except Exception as e:
            self.logger.error(
```

```
                f"Failed to restart container {container.name}: 
{str(e)}"
            )
            await self._recreate_container(container.name)

    async def _recreate_container(self, container_name: str):
        """Відновлення контейнера після збою"""
        try:
            # Видалення старого контейнера
            try:
                old_container = 
self.docker_client.containers.get(container_name)
                old_container.remove(force=True)
            except docker.errors.NotFound:
                pass

            # Створення нового контейнера
            self.docker_client.containers.run(
                'worker-image:latest',
                name=container_name,
                detach=True,
                labels={'type': 'worker'},
                environment={
                    'REDIS_HOST': 'redis',
                    'DB_HOST': 'postgres'
                }
            )

            self.logger.info(f"Container {container_name} recreated 
successfully")

        except Exception as e:
            self.logger.error(f"Failed to recreate container 
{container_name}: {str(e)}")

async def main():
    monitoring = MonitoringSystem()
    failover = FailoverManager(monitoring)

    # Запуск сервера метрик
    monitoring.start_metrics_server()

    # Запуск збору метрик у окремому потоці
    metrics_thread = threading.Thread(
        target=monitoring.collect_system_metrics,
```

```
            daemon=True
        )
        metrics_thread.start()

        # Запуск перевірки здоров'я системи
        await failover.check_node_health()

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Додаток Б

Конфігурація Prometheus

Лістинг Б.2 – Конфігурація Prometheus

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'distributed-system'
    static_configs:
      - targets: ['localhost:8000']
    metrics_path: '/metrics'

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

alerting:
  alertmanagers:
    - static_configs:
      - targets: ['alertmanager:9093']

rule_files:
  - 'alerts.yml'
```