НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
Інформаційних технологій	Комп'ютерних систем, мереж та кібербезпеки
(назва факультету(ННІ))	(назва кафедри)
_____ 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)
«___»_____2025 р.	«___»_____2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження та розробка криптовалютного терміналу для масових платежів»

Спеціальність 123 «Комп’ютерна інженерія»
(код і найменування)

Освітня програма Комп’ютерні системи та мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

д.т.н., доцент	_____	Шкарупило В.В.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Керівник магістерської кваліфікаційної роботи

д.т.н., проф	_____	Болбот І.М.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Виконав _____ Груша В. В.
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
к.пед.н., доц. _____ Касаткін Д.Ю.
(вчене звання і ступінь) (підпис) (ПІБ)
«__» _____ 20__ р.

З А В Д А Н Н Я
ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Груші Віталій Вікторовичу
(прізвище, ім'я, по батькові)

Спеціальність 123 «Комп'ютерна інженерія»
(код і найменування)

Освітня програма Комп'ютерні системи та мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи: «Дослідження та розробка криптовалютного терміналу для масових платежів»

затверджена наказом ректора НУБіП України від "29" жовтня 2024р. № 1941 «С»

Термін подання завершеної роботи на кафедру 14 листопада 2025 р.

Вихідні дані до магістерської кваліфікаційної роботи _____

Перелік питань, що підлягають дослідженню:

1. _____
2. _____
3. _____

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання "29" жовтня 2024 р.

Керівник магістерської кваліфікаційної роботи _____ Болбот І.М.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Груша В.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 105 сторінок, 22 рисунків, 3 додатки, 21 джерел.

КРИПТОВАЛЮТНІ ПЛАТЕЖІ, РОЗПОДІЛЕНІ СИСТЕМИ, ПОДІЄВА АРХІТЕКТУРА, ПОДВІЙНИЙ ЗАПИС, ЖУРНАЛ ТРАНЗАКЦІЙ, KAFKA, YUGABYTEDB, CLICKHOUSE, IMMUDB, KUBERNETES, МАСШТАБУВАННЯ, КРИПТОГРАФІЧНИЙ АУДИТ

Мета роботи – дослідження та розробка високонавантаженого криптовалютного платіжного терміналу, здатного коректно обробляти масові транзакції, забезпечувати подвійний бухгалтерський запис та підтримувати незалежну криптографічну верифікацію фінансової історії.

Об’єкт дослідження – процеси обробки, валідації та обліку криптовалютних платежів у розподілених платіжних системах.

Предмет дослідження – архітектурні та програмні засоби побудови подієво-орієнтованого платіжного терміналу, включаючи механізми подвійного запису, розподілені SQL-бази даних з ACID-властивостями, системи потокової обробки подій та криптографічні незмінні сховища.

У першому розділі виконано аналіз предметної області, розглянуто сучасні криптовалютні платіжні рішення та визначено основні технічні проблеми масових транзакцій. Другий розділ містить теоретичні засади побудови системи: принципи подвійного бухгалтерського запису, моделі розподіленої консистентності та концепції подієво-орієнтованої архітектури. У третьому розділі розроблено архітектуру платіжного терміналу, подано опис мікросервісної інфраструктури, механізмів асинхронної обробки подій, реалізації журналу транзакцій і інтеграції з незмінним сховищем. Четвертий розділ присвячений експериментальній перевірці: проведено вимірювання продуктивності, аналіз затримок, тестування інваріанту подвійного запису та валідацію криптографічних доказів. Отримані результати демонструють коректність архітектури та підтверджують сформульовані гіпотези.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API - Application Programming Interface (програмний інтерфейс застосування)

gRPC - Google Remote Procedure Call (протокол віддаленого виклику процедур)

SQL - Structured Query Language (мова структурованих запитів)

ACID - Atomicity, Consistency, Isolation, Durability (атомарність, узгодженість, ізоляція, довговічність)

TPS - Transactions Per Second (транзакцій за секунду)

RPS - Requests Per Second (запитів за секунду)

CAP - Consistency, Availability, Partition tolerance (узгодженість, доступність, стійкість до розділення)

PACELC - Partition tolerance, Availability, Consistency, Else, Latency, Consistency

CP - Consistency + Partition tolerance

AP - Availability + Partition tolerance

ВСТУП

Стрімкий розвиток цифрової економіки перетворив криптовалютні платежі на важливу складову глобальної фінансової інфраструктури. Зростання популярності цифрових активів у сферах електронної комерції, міжнародних грошових переказів та автоматизованих онлайн-сервісів обумовлює потребу в спеціалізованих платіжних рішеннях, здатних забезпечувати високу швидкість обробки, горизонтальну масштабованість та надійність обслуговування великих обсягів транзакцій. Водночас технічна складність взаємодії з різними блокчейн-мережами, непередбачуваність часу підтвердження транзакцій та суворі вимоги до математичної точності фінансового обліку створюють принципово нові архітектурні та технологічні виклики для розробників сучасних платіжних систем масового призначення.

Актуальність теми дослідження зумовлена необхідністю створення криптовалютних платіжних терміналів, здатних працювати у режимі масових платежів, гарантувати інваріант подвійного бухгалтерського запису та забезпечувати можливість незалежного криптографічного аудиту. Платіжні шлюзи нерідко мають обмеження щодо масштабованості, прозорості та довіри до збереження фінансової історії, що підсилює потребу у нових підходах до архітектури таких систем.

Метою дослідження є розробка та експериментальна оцінка розподіленої подієво-орієнтованої системи криптовалютного платіжного терміналу, яка забезпечує коректну обробку фінансових операцій, підтримку подвійного запису, горизонтальне масштабування та криптографічну верифікацію незмінності даних.

Об’єктом дослідження є процеси обробки криптовалютних транзакцій у розподілених платіжних системах.

Предметом дослідження є архітектурні, алгоритмічні та програмні засоби побудови високонавантаженого криптовалютного терміналу із забезпеченням

подвійного бухгалтерського запису та криптографічно підтверджуваної незмінності фінансових журналів.

Наукова новизна роботи полягає у поєднанні розподіленої SQL-системи обліку, подієвого конвеєра обробки транзакцій та незмінного криптографічного сховища для формування платіжного терміналу, здатного до горизонтального масштабування та незалежної зовнішньої верифікації даних. Запропонована архітектура інтегрує механізми подвійного бухгалтерського запису, Apache Kafka як транспорт подій та ImmuDB як шар незмінності, що дозволяє підвищити рівень довіри та прозорості системи.

Практична цінність результатів полягає у можливості застосування розробленої архітектури в реальних фінансових сервісах, платіжних шлюзах та e-commerce платформах, де потрібні висока пропускна здатність, відмовостійкість та криптографічні гарантії незмінності історії транзакцій.

Методи дослідження ґрунтуються на теорії розподілених систем, моделі подвійного бухгалтерського запису, концепціях подієво-орієнтованої архітектури, протоколах консенсусу та експериментальному аналізі продуктивності у локальному кластері Kubernetes.

У процесі дослідження були поставлені та вирішені такі завдання:

– аналіз криптовалютних платіжних систем та їх архітектурних обмежень;

– визначення вимог до високонавантаженого криптовалютного терміналу;

– розроблення архітектури подієво-орієнтованої розподіленої системи;

– реалізація мікросервісів для обробки інвойсів, бухгалтерських журналів та криптографічного аудиту;

– створення симулятора навантаження та проведення серії експериментів;

– оцінка масштабованості, затримок та виконання фінансових інваріантів;

– перевірка можливості незалежної криптографічної верифікації даних.

Отримані результати можуть бути використані при створенні криптовалютних платіжних рішень, розподілених фінансових систем та сервісів електронної комерції, де необхідні висока пропускна здатність, точність фінансового обліку та прозорість операцій.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Стрімкий розвиток цифрової економіки та глобальне поширення криптовалют призвели до появи нового класу фінансових сервісів, орієнтованих на обробку платежів у децентралізованих мережах. Попри значні досягнення блокчейн-технологій, їхнє практичне застосування у сценаріях масових комерційних транзакцій і надалі стикається з низкою фундаментальних обмежень - від низької пропускної здатності й високих затримок до складності інтеграції з бізнес-процесами та відсутності надійних механізмів аудиту.

Потреба у швидких, масштабованих та криптографічно захищених платіжних рішеннях стимулює появу спеціалізованих терміналів, які працюють поверх різних блокчейнів, але не залежать від їхнього внутрішнього консенсусу. Такі системи повинні забезпечувати гарантоване зарахування платежів, цілісність бухгалтерського обліку, конфіденційність учасників та доведеність історії операцій - особливо у випадку інтеграції з підприємствами, торговими майданчиками та сервісами онлайн-платежів.

1.1 Застосування криптовалютних терміналів

Криптовалютні платежі за останні роки трансформувалися з нішевого інструмента для ентузіастів у повноцінний сегмент цифрової фінансової інфраструктури. Це зумовлено глобальною доступністю криптовалют, зниженням бар'єрів входу для бізнесу та користувачів, а також зростанням частки

електронної комерції й сервісної економіки. Однак практичне використання криптовалют у масових платежах, тобто великій кількості однотипних транзакцій від багатьох клієнтів на користь численних продавців і постачальників послуг, вимагає проміжної програмної системи, здатної забезпечити масштабування, стабільність і довіру до фінансових записів. Саме таку роль виконують криптовалютні термінали.

Сфера застосування криптовалютних терміналів охоплює широкий спектр бізнес-сценаріїв. У сегменті електронної комерції та цифрових торгових майданчиків такі системи забезпечують прийом криптовалют за товари та послуги від великої кількості клієнтів із формуванням рахунків-фактур, контролем оплати та визначенням остаточних сум у фіатній валюті. Для мікроплатежів і цифрових сервісів, зокрема підписок, пожертвувань, оплати контенту, внутрішньоігрових та внутрішньопрограмних покупок, критичними є швидкість обробки та автоматичне зарахування коштів. У транскордонних розрахунках криптовалютні термінали дозволяють здійснювати платежі між країнами для бізнесу та фрилансерів без залучення традиційних банківських посередників і валютних обмежень. Автономні торгові точки та системи продажів використовують такі рішення для приймання криптовалют через генерацію платіжних запитів у форматі QR-кодів або гіперпосилань із прив'язкою оплати до конкретного замовлення та автоматизованим обліком операцій. Нарешті, у міжкорпоративних розрахунках і масових виплатах організації потребують можливості швидкого проведення розрахунків із великою кількістю контрагентів зі збереженням прозорого журналу транзакцій.

Попри переваги криптовалют як засобу розрахунку, базові блокчейн-мережі мають обмеження, що ускладнюють пряме використання для масових платежів: варіативні затримки підтвердження, залежність від комісій мережі, відсутність гнучких механізмів інтеграції з бізнес-логікою, а також проблема довіри до збереження фінансової історії у централізованих сервісах. У реальних платіжних системах користувач сплачує на адресу, що контролюється платформою, а продавець отримує доступ до коштів лише після процедури

виводу. Це створює потребу в додаткових засобах гарантування цілісності обліку та неможливості підтасування журналу транзакцій з боку оператора платформи.

Таким чином, криптовалютний термінал виступає ключовою ланкою між блокчейном і бізнесом, виконуючи функції приймання та інтерпретації платіжних подій з різних мереж, забезпечення їх узгодженого обліку у фіатному еквіваленті, реалізації механізмів масштабованої обробки потоків транзакцій та гарантування доказової цілісності фінансової історії для всіх учасників операцій.

У межах цього дослідження криптовалютний термінал розглядається як програмна розподілена система, призначена для масового прийому платежів у різних блокчейн-мережах, їх перетворення в уніфікований формат транзакцій, подієвої обробки та ведення подвійного бухгалтерського запису з криптографічною верифікацією незмінності історії операцій.

1.1.1 Призначення криптовалютних терміналів

Криптовалютний термінал становить спеціалізований програмний комплекс, призначений для організації прийому криптовалютних платежів у комерційних системах і забезпечення повного життєвого циклу транзакцій від формування платіжного запиту до підтвердження оплати, бухгалтерського обліку та надання доказів достовірності історії операцій.

Призначення криптовалютних терміналів полягає у вирішенні трьох взаємопов'язаних завдань. По-перше, уніфікація та автоматизація прийому платежів із різних блокчейн-мереж передбачає інтеграцію з зовнішніми мережами, такими як Bitcoin, Ethereum та мережі стабільних монет, приховуючи від бізнес-рівня відмінності у протоколах, форматах подій і моделях підтвердження транзакцій. Це дозволяє торговцям приймати оплату через єдиний інтерфейс незалежно від конкретного блокчейну та криптовалюти.

По-друге, масштабована обробка платіжних подій і підтримка високого навантаження у системах масових платежів вимагає надійної обробки значного потоку рахунків-фактур та транзакцій із гарантуванням узгодженості фінансових записів навіть за умов паралельного виконання операцій, відмов окремих

сервісів чи пікових навантажень. Це обґрунтовує доцільність застосування мікросервісної архітектури, подієвої шини обміну повідомленнями та механізмів горизонтального масштабування.

По-третє, фінансовий облік і механізми довіри для бізнесу передбачають, що термінал виконує не лише технічну функцію прийому платежу, а й роль фінансового реєстратора, формуючи записи подвійного бухгалтерського обліку, оновлюючи баланси та забезпечуючи доказову базу цілісності історії операцій. В умовах тимчасового контролю коштів платформою термінал має надати торговцю можливість незалежно верифікувати незмінність транзакцій та підсумкового балансу з моменту їх фіксації.

Типові функції криптовалютного терміналу включають:

– Створення рахунків-фактур та платіжних запитів у фіатному еквіваленті з автоматичним розрахунком суми у обраній криптовалюті за поточним курсом обміну.

– Надання користувачу можливості вибору криптовалюти та блокчейн-мережі для здійснення платежу з подальшою прив'язкою рахунку-фактури до унікальної адреси цифрового гаманця.

– Отримання та первинну валідацію подій з блокчейн-мереж через спеціалізовані модулі прослуховування транзакцій у режимі реального часу.

– Співставлення вхідного платежу з відповідним рахунком-фактурою на основі адреси гаманця-отримувача та перевірку достатності перерахованої суми з урахуванням мережевих комісій.

– Формування уніфікованої внутрішньої транзакції незалежно від типу блокчейну та передавання її до облікової підсистеми для подальшої обробки.

– Запис транзакції у розподілений журнал подвійного бухгалтерського обліку з одночасним оновленням балансів відповідних рахунків торговців та системних рахунків.

– Криптографічне закріплення фінансових записів у незмінюваному сховищі з формуванням хеш-ланцюгів для кожного торговця та можливістю ретроспективної верифікації.

– Надання програмного інтерфейсу для отримання торговцями криптографічних доказів цілісності історії операцій та можливості незалежного аудиту балансів.

– Збір аналітичних даних та метрик продуктивності системи для моніторингу стану інфраструктури, виявлення аномалій і оптимізації обробки транзакцій.

Отже, криптовалютний термінал за своїм призначенням становить програмну систему прийому криптовалютних платежів, що поєднує функції платіжного інтерфейсу, обчислювального процесора подій і фінансового реєстратора з механізмами криптографічної верифікації. Термін «термінал» у контексті цього дослідження є коректним, оскільки він описує систему як комплексне рішення для прийому та обробки платежів у комерційному середовищі, функціонально аналогічне класичним платіжним терміналам у фіатній економіці, проте реалізоване програмними засобами та адаптоване для взаємодії з блокчейн-мережами.

1.1.2 Типові сценарії використання

Практичне застосування криптовалютних терміналів визначається потребою бізнесу організувати прийом криптовалютних платежів як стандартний платіжний інструмент з урахуванням специфіки блокчейн-мереж. Типові сценарії використання можна класифікувати за ролями учасників та контекстом здійснення платежу.

Найпоширенішим сценарієм є створення торговцем рахунку-фактури із сумою у доларах або іншій фіатній валюті, після чого клієнт обирає криптовалюту та мережу і отримує адресу гаманця, QR-код або гіперпосилання для здійснення платежу. У цьому контексті термінал генерує рахунок-фактуру та платіжне посилання, резервує гаманець під конкретний рахунок, очікує на подію

з блокчейн-мережі, співставляє отриманий платіж із рахунком-фактурою, фіксує операцію в обліковій системі та надає підтвердження торговцю і клієнту.

У цифрових сервісах, зокрема контент-платформах, іграх, стримінгових та благодійних сервісах, спостерігаються потоки однотипних невеликих платежів. У таких випадках термінал функціонує як високонавантажений процесор подій, де критичними є автоматична обробка великої кількості рахунків-фактур, мінімальна затримка між здійсненням оплати і зарахуванням коштів, а також стабільність системи під час пікових навантажень.

У фізичних торгових точках продавець або система продажів формує рахунок-фактуру на касі, а клієнт здійснює оплату з мобільного гаманця через QR-код або адресу. Термінал у цьому сценарії забезпечує швидке створення рахунку-фактури, зрозумілий платіжний інтерфейс для клієнта, підтвердження факту оплати без безпосередньої участі касира та синхронізацію з бухгалтерським обліком торговця.

Клієнти можуть здійснювати платежі у різних блокчейн-мережах і токенах, таких як Bitcoin, Ethereum, USDT у різних мережах тощо. Термінал виступає уніфікатором, що веде перелік підтримуваних валют і мереж, підбирає відповідний гаманець, здійснює конвертацію криптовалютної суми в еквівалент у доларах США та забезпечує однаковий формат фінансового запису незалежно від блокчейн-мережі.

У більшості платіжних платформ клієнт переказує кошти на адресу, що належить платформі, а торговець отримує їх лише під час виведення. Тому термінал використовується також як система забезпечення довіри, що фіксує незмінний журнал транзакцій, дозволяє торговцю верифікувати незмінність історії операцій та гарантує коректність накопиченого балансу до моменту виведення коштів.

Ці сценарії відображають практичний контекст, у якому криптовалютний термінал є необхідним посередником між блокчейн-мережами, бізнесом та клієнтами, поєднуючи механізми прийому платежів, подієвої обробки та фінансового обліку.

1.1.3 Основні вимоги бізнесу до таких систем

Для впровадження криптовалютних платежів бізнес очікує від терміналу не лише технічної інтеграції з блокчейн-мережами, а повноцінного сервісу, який за рівнем надійності та прозорості відповідає традиційним платіжним системам. Основні вимоги можна класифікувати на функціональні, нефункціональні та вимоги забезпечення довіри.

Функціональні вимоги:

– Підтримка моделі рахунків-фактур, що включає створення рахунку-фактури із сумою у доларах США, його оновлення при виборі криптовалюти та мережі, а також отримання поточних статусів обробки.

– Багатовалютність і багатомережева підтримка через єдиний програмний інтерфейс для прийому різних криптовалют у різних блокчейн-мережах без необхідності окремої інтеграції для кожної мережі.

– Автоматичне співставлення платежів з рахунками-фактурами через механізм зв'язку адреси гаманця з конкретним рахунком, перевірку достатності суми та ідентифікацію випадків надмірної або недостатньої оплати.

– Ведення фінансового журналу з фіксацією кожної операції у структурі подвійного бухгалтерського запису та коректним нарахуванням комісії сервісу відповідно до встановленої тарифної політики.

– Програмний інтерфейс для інтеграції, що надає торговцям зрозумілі методи виклику для створення рахунків-фактур, отримання статусів обробки та криптографічних доказів, а клієнтам - спрощений доступ до платіжних запитів.

Нефункціональні вимоги:

– Масштабованість системи з можливістю обробки зростаючих потоків рахунків-фактур і транзакцій за рахунок горизонтального масштабування компонентів без деградації продуктивності.

– Відмовостійкість архітектури, що забезпечує функціонування системи при часткових відмовах окремих сервісів, вузлів інфраструктури або мережевих сегментів.

– Низька затримка обробки платежів для забезпечення швидкого підтвердження оплати та актуального стану балансу в режимі, наближеному до реального часу.

– Ідемпотентність операцій і консистентність даних із захистом від повторної обробки однієї транзакції та суворим збереженням фінансового інваріанту рівності дебету та кредиту.

– Спостережуваність системи через наявність метрик продуктивності, структурованих журналів подій і розподіленого трасування для контролю рівня обслуговування та виявлення вузьких місць.

Вимоги забезпечення довіри та аудиту:

– Неможливість прихованої модифікації історії операцій, що гарантує торговцям впевненість у відсутності можливості платформи змінювати журнал транзакцій або баланси заднім числом.

– Криптографічна верифікація транзакцій через надання доказів включення записів до незмінюваного сховища та їх цілісності через програмний інтерфейс, що дозволяє торговцям здійснювати незалежну перевірку.

– Прозоре формування підсумкового балансу з математичним обґрунтуванням на основі записів журналу замість покладання на довірене значення у базі даних.

– Підтримка процедур розслідування та експорту даних з можливістю отримання записів для зовнішнього аудиту або задоволення бухгалтерських потреб організації.

Отже, для бізнесу криптовалютний термінал є не просто інтерфейсом прийому криптовалютних платежів, а критичною фінансовою підсистемою. Її ключовими очікуваними властивостями є масштабування під масові транзакції, сувора коректність обліку та доказова довіра до історії операцій у моделі, де платформа тимчасово контролює кошти торговця до моменту їх виведення.

1.2 Огляд існуючих рішень та їх обмежень

Ринок криптовалютних платежів уже налічує низку готових рішень, що дозволяють бізнесу приймати платежі від клієнтів. Такі сервіси найчастіше позиціонуються як криптовалютні платіжні шлюзи або процесори платежів. Вони вирішують базове завдання інтеграції з блокчейн-мережами, проте при масштабуванні до рівня масових платежів виявляються суттєві обмеження як технологічного характеру, так і пов'язані з механізмами довіри та фінансовим обліком. Тому аналіз існуючих продуктів і архітектурних рішень є важливим для формування вимог до системи, що розробляється в межах цього дослідження.

1.2.1 Комерційні криптоплатіжні шлюзи

Комерційні платіжні шлюзи, такі як Cryptomus, CoinPayments, NOWPayments, BitPay, та інші, надають бізнесу готовий набір функціональних можливостей для приймання криптовалютних платежів. Інвойсна модель передбачає створення торговцем рахунку-фактури у доларах США або іншій фіатній валюті, після чого клієнт здійснює оплату в обраній криптовалюті. Управління цифровими гаманцями реалізується через видачу платформою унікальної адреси для платежу зі списку власних адрес, її резервування на час дії рахунку-фактури та відстеження надходження коштів. Обробка блокчейн-подій здійснюється через прослуховування власних вузлів або використання сторонніх постачальників даних із подальшою фільтрацією транзакцій за адресами та сумами. Механізм конвертації та комісій передбачає фіксацію обмінного курсу у момент оплати, застосування сервісної комісії та обчислення чистої суми для торговця. Модель виведення коштів зазвичай означає, що до моменту виведення

кошти контролюються платформою, а торговець переглядає баланс у особистому кабінеті та може ініціювати виплату. Механізм вебхуків забезпечує автоматичне повідомлення торговця про зміну статусу рахунку-фактури через зворотні виклики.

Такі продукти ефективно функціонують для малих або середніх обсягів транзакцій, надаючи бізнесу швидку інтеграцію без необхідності розгортання власної інфраструктури. Проте їхні внутрішні механізми залишаються непрозорими для зовнішніх користувачів, оскільки торговець покладається на правильність обліку та нарахування балансу виключно на платформу без можливості незалежної верифікації.

1.2.2 Архітектурні моделі

У практиці існує кілька типових архітектурних підходів до побудови криптовалютних платіжних систем.

Централізований шлюз, коли система працює наступним чином:

- володіє цифровими гаманцями та адресами;
- видає клієнту адресу для здійснення оплати;
- приймає кошти на власні рахунки;
- веде баланс торговця у власній базі даних;
- виконує виведення коштів за запитом.

Ця модель є простою для впровадження бізнесом, проте створює критичну залежність від довіри до оператора платформи.

Ще одна модель коли адреси належать торговцю, а платформа лише:

- генерує рахунки-фактури;
- відстежує транзакції для цих адрес;
- агрегує дані для обліку.

Перевагою є зменшення залежності від довіри до платформи, проте недоліком виступає складніша інтеграція та більша відповідальність за управління криптографічними ключами з боку торговця.

У гібридній моделі з внутрішнім фінансовим журналом платформа приймає кошти на власні адреси, але реалізує:

- внутрішній фінансовий журнал на основі подвійного бухгалтерського запису;
- чіткі інваріанти обліку для забезпечення коректності операцій;
- інколи механізми зовнішніх доказів або експорту даних.

В подієво-орієнтована моделі система розділяє обробку на окремі етапи:

- отримання подій з блокчейн-мереж;
- співставлення транзакцій з рахунками-фактурами;
- формування фінансового запису у журналі обліку;
- аналітичну обробку та надсилання сповіщень.

Забезпечує масштабованість через використання черг і шин обміну повідомленнями, проте потребує строгих гарантій консистентності даних між етапами обробки.

1.2.3 Ключові недоліки

Аналіз комерційних рішень і архітектурних моделей показує, що при масових платежах виникає низка системних проблем.

Навіть за умови нормального функціонування зовнішніх блокчейн-мереж у платіжних шлюзів часто виникають внутрішні обмеження продуктивності:

- пул адрес і гаманців та його блокування під час резервування;
- централізований запис в одну базу даних без механізмів шардування;
- обмеження на паралельну обробку подій з різних джерел;

– складність горизонтального масштабування компонентів фінансового ядра системи.

У моделі з відкладеним виведенням коштів платформа тимчасово контролює кошти торговця. У більшості рішень:

– торговець бачить баланс лише в особистому кабінеті платформи;

– відсутній механізм незалежної перевірки журналу транзакцій;

– неможливо отримати криптографічні докази незмінності історії операцій.

Таким чином, довіра базується на організаційних гарантіях, а не на математично доведених властивостях системи.

Багато платіжних шлюзів фактично ведуть спрощений баланс без повноцінного журналу подвійного бухгалтерського запису.

Це породжує ризики:

– складність проведення аудиту фінансових операцій;

– потенційні помилки під час нарахування сервісних комісій;

– обмежена можливість трасування походження підсумкового балансу.

Платіжні шлюзи часто використовують сторонні програмні інтерфейси або вузли мережі. При піках навантаження або деградації постачальника:

– зростає затримка підтверджень транзакцій;

– виникають можливі пропуски або затримки у обробці подій;

– ускладнюється забезпечення гарантованого рівня обслуговування для торговця.

1.3 Проблематика масових криптовалютних транзакцій

Масові криптовалютні платежі відрізняються від класичних банківських чи карткових операцій тим, що фінальна подія оплати виникає не всередині платіжної системи, а у зовнішньому децентралізованому середовищі - блокчейні. Це створює набір специфічних проблем для сервісу, який виконує роль

криптовалютного терміналу: він має не лише зафіксувати оплату, а й перетворити її на коректний бухгалтерський запис, забезпечивши довіру учасників. При переході від одиничних оплат до масового потоку однотипних транзакцій ці проблеми стають критичними та визначають вимоги до архітектури і технологій системи.

1.3.1 Нестабільний час підтвердження блокчейн-транзакцій

У блокчейн-мережах час підтвердження транзакції не є детермінованим і залежить від численних факторів: рівня завантаженості мережі, розміру транзакційної комісії, параметрів алгоритму консенсусу, а також специфічної політики конкретної мережі щодо кількості підтверджень, що вважаються достатніми для фінальності операції. У практичному застосуванні це призводить до того, що одна й та сама система може отримувати події підтвердження з варіацією від кількох секунд до десятків хвилин.

Для систем масових платежів це зумовлює два ключові виклики. По-перше, необхідно підтримувати рахунок-фактуру у стані очікування з гарантією коректної обробки навіть за умов значних часових затримок. По-друге, система має забезпечувати стійкість до пікових навантажень підтверджень, коли велика кількість транзакцій може отримати фінальний статус майже одночасно.

Отже, архітектура криптовалютного терміналу повинна передбачати асинхронну подієво-орієнтовану обробку транзакцій та можливість горизонтального масштабування конвеєра підтверджень для забезпечення стабільної продуктивності системи.

1.3.2 Невідповідність між подією в блокчейні та бухгалтерським обліком

Факт надходження коштів у блокчейн-мережі ще не становить бухгалтерську операцію в системі криптовалютного терміналу. Подія з блокчейну має пройти низку трансформацій: співставлення з конкретним рахунком-фактурою, перевірку на відповідність очікуваній сумі з урахуванням

правил обробки доплат і переплат, конвертацію у базову валюту обліку системи (у цьому дослідженні - долари США), а також перетворення у записи подвійного бухгалтерського обліку з визначенням дебету та кредиту відповідних рахунків.

Ключова складність полягає у тому, що ланцюг трансформації від події блокчейну до фінансового запису не є атомарною операцією, і при масових платежах зростає ризик виникнення неузгодженостей між етапами обробки, наприклад коли підтвердження транзакції отримано, але запис у журнал обліку не відбувся через відмову компонента системи. Тому необхідне чітке розділення контурів обробки та впровадження механізмів, які гарантують консистентність фінансової історії навіть за умов відкладеної або асинхронної обробки транзакцій.

1.3.3 Ризики підміни у централізованому серверному застосунку

У моделі з відкладеним виведенням коштів централізована платформа тимчасово контролює активи торговця до моменту здійснення виплати. Це зумовлює виникнення проблеми довіри: торговець не має можливості математично довести, що історія транзакцій і поточний баланс не зазнали змін ретроспективно внаслідок дій адміністратора системи або компрометації серверної інфраструктури.У класичних платіжних шлюзах довіра базується на репутації оператора, проте для широкого бізнес-застосування необхідна криптографічно верифікована незмінність журналу операцій. Отже, виникає фундаментальна вимога до системи: зберігати фінансові записи у незмінюваному сховищі з можливістю надання криптографічних доказів включення транзакцій і консистентності даних, які торговець може перевірити незалежно від платформи.

1.3.4 Необхідність забезпечити баланс для кожного платежу

У фінансових системах будь-яка операція має виконувати інваріант: сума дебету дорівнює сумі кредиту. Для криптовалютного терміналу це означає, що

після завершення оплати система повинна сформувати коректний запис подвійного обліку, у якому:

- відображено надходження коштів;
- відображено зміну балансу мерчанта;
- відображено сервісну комісію платформи (якщо застосовується).

При масових транзакціях навіть невелика кількість «небалансованих» записів призводить до накопичення помилки балансу, яку надалі важко локалізувати. Тому балансування кожного платежу є фундаментальною вимогою до леджера і предметом експериментальної валідації у роботі.

1.3.5 Масштабованість обробки великої кількості однотипних транзакцій

У фінансових системах будь-яка операція має дотримуватися фундаментального інваріанту: сума дебету дорівнює сумі кредиту. Для криптовалютного терміналу це означає, що після завершення обробки платежу система повинна сформувати коректний запис подвійного бухгалтерського обліку, у якому відображено надходження коштів на рахунки системи, зміну балансу торговця відповідно до отриманої суми, а також сервісну комісію платформи у разі її застосування.

При масових транзакціях навіть невелика кількість записів, що порушують баланс, призводить до накопичення похибки у загальному балансі системи, яку надалі складно локалізувати та виправити. Тому забезпечення балансування кожного платежу є фундаментальною вимогою до архітектури журналу обліку і становить предмет експериментальної валідації в межах цього дослідження.

1.4 Аналіз технологій для побудови високонавантаженої розподіленої системи

Для реалізації криптовалютного терміналу масових платежів необхідно поєднати три технологічні класи компонентів: транзакційне сховище для ведення подвійного бухгалтерського обліку, подієву шину обміну

повідомленнями для асинхронної обробки підтверджень транзакцій, а також незмінне криптографічне сховище для забезпечення можливості зовнішнього аудиту операцій.

1.4.1 Розподілені SQL-бази даних

Для журналу подвійного бухгалтерського запису критично важливими є транзакційність операцій, узгодженість даних і можливість горизонтального масштабування системи. У класичній системі керування базами даних PostgreSQL масштабування здебільшого досягається вертикально або через зовнішні розширення для шардування, що ускладнює адміністрування та не гарантує прозорої ACID-семантики в розподіленому середовищі.Розподілені SQL-орієнтовані системи керування базами даних, такі як YugabyteDB, CockroachDB та інші, забезпечують низку критичних можливостей. Горизонтальний шардинг реалізується нативно, коли дані автоматично розподіляються на таблетки або шарди за первинним ключем та рівномірно розміщуються у вузлах кластера, що дозволяє підтримувати збалансоване навантаження навіть при значній кількості однотипних транзакцій. Узгоджені розподілені транзакції з підтримкою ACID-властивостей необхідні для журналу обліку, оскільки вставка заголовку запису та відповідних рядків дебету і кредиту має відбуватися атомарно, а розподілена SQL-база забезпечує такі транзакції без необхідності ручної організації двофазних комітів на рівні прикладної логіки. Локальність доступу до даних досягається при правильному виборі ключа шардування, коли таблиці розподіляються за спільним первинним ключем, що дозволяє зменшити кількість міжвузлових операцій при вставці та читанні записів журналу. Стійкість до відмов забезпечується через реплікацію шардів і консенсусні механізми на рівні системи керування базою даних, що дозволяє уникнути втрати журналу навіть при відмові окремих вузлів інфраструктури.Отже, розподілена SQL-база даних є природною основою для високонавантаженого подвійного бухгалтерського запису, оскільки поєднує

бухгалтерську коректність операцій з можливістю горизонтального масштабування системи.

1.4.2 Системи подієвої взаємодії

Оскільки підтвердження платежів виникає у зовнішніх блокчейн-мережах із непередбачуваними затримками, криптовалютний термінал повинен будуватися як подієво-орієнтована система, де ключові стадії обробки зв'язані через обмін подіями. Для цього необхідна подієва шина обміну повідомленнями на базі платформи Kafka, що забезпечує низку критичних властивостей.

Гарантований порядок подій у межах одного ключа досягається через партиціювання за ідентифікатором торговця, адресою гаманця або ідентифікатором журналу, що дозволяє забезпечити послідовність операцій для конкретного торговця чи рахунку-фактури. Масштабування через групи споживачів дозволяє при збільшенні навантаження нарощувати кількість партицій і споживачів повідомлень, досягаючи майже лінійного масштабування пропускної здатності конвеєра обробки.

Механізми повторних спроб і толерантності до відмов є критичними, оскільки в системі можливі часткові відмови компонентів внаслідок помилок бази даних, мережевих збоїв або тимчасової недоступності шардів. Подієвий підхід дозволяє повторювати обробку повідомлення до досягнення успішного результату з гарантією доставки принаймні один раз за умови ідемпотентності операцій запису.

Розділення швидкостей роботи компонентів також є важливим аспектом, оскільки події з блокчейн-мереж можуть надходити нерівномірно, тоді як журнал обліку та рівень забезпечення незмінності можуть функціонувати з іншою швидкістю обробки. Черга подій згладжує ці нерівномірності та забезпечує стабільність функціонування системи.

Таким чином, подієва шина обміну повідомленнями є ключовим механізмом масштабування та узгодження асинхронного ланцюга обробки платежів.

1.4.3 Криптографічні незмінні сховищ

У моделі з відкладеним виведенням коштів централізована платформа тимчасово володіє активами торговця. Це означає, що для бізнесу критично важливо мати криптографічні гарантії незмінності історії платежів, які не залежать від рівня довіри до адміністратора сервісу.

Незмінні криптографічні сховища, такі як ImmuDB, Amazon QLDB та інші, забезпечують низку ключових властивостей. Модель виключно додавання записів передбачає, що дані не переписуються, а лише дописуються до журналу, тому фальсифікація записів ретроспективно стає неможливою без залишення слідів у криптографічній структурі. Дерево Меркла або структура хеш-журналу забезпечує включення кожної транзакції у меркл-структуру, що дозволяє математично довести її наявність і порядок у журналі операцій.

Криптографічні докази включення та консистентності даних є фундаментальною можливістю таких систем, оскільки ImmuDB повертає математичні докази, які клієнт або торговець може перевірити незалежно від платформи. Можливість зовнішнього аудиту реалізується через програмний інтерфейс, що надає торговцю змогу отримувати криптографічний доказ того, що конкретна транзакція або поточний баланс не зазнали модифікації після первинної фіксації.

У контексті архітектури криптовалютного терміналу ImmuDB виступає другим рівнем забезпечення довіри: YugabyteDB забезпечує масштабований бухгалтерський запис з підтримкою транзакційності, тоді як ImmuDB гарантує незмінність історії та можливість незалежного аудиту для бізнес-користувачів.

1.5 Постановка задачі дослідження

На основі проведеного аналізу предметної області та технологічного стеку формується задача дослідження.

Мета дослідження:

Спроєктувати та реалізувати прототип криптовалютного терміналу для масових платежів, який забезпечує масштабовану обробку рахунків-фактур,

бухгалтерську коректність подвійного запису та криптографічно верифіковану незмінність історії транзакцій для торговців.

Для досягнення мети необхідно виконати такі завдання:

– Сформувати функціональні та нефункціональні вимоги до системи масових криптовалютних платежів з урахуванням непередбачуваних затримок підтверджень у блокчейн-мережах, потреби у бухгалтерському обліку та моделі відкладеного виведення коштів.

– Спроєктувати мікросервісну подієво-орієнтовану архітектуру з чітким розділенням відповідальності між сервісами обліку, обробки платежів, фінансового журналу, аналітики та симуляції навантаження.

– Реалізувати розподілений журнал подвійного бухгалтерського запису у YugabyteDB із фіксацією кожного платежу у вигляді збалансованого запису журналу з дотриманням інваріанту рівності дебету та кредиту.

– Інтегрувати незмінне криптографічне сховище ImmuDB для забезпечення можливості аудиту та розробити формат збереження записів і механізм надання криптографічних доказів через програмний інтерфейс.

– Побудувати симулятор навантаження для відтворення потоків рахунків-фактур і подій з блокчейн-мереж у контрольованому експериментальному середовищі.

Провести експериментальні дослідження, зокрема:

– виміряти пропускну здатність конвеєра обробки при різній кількості партицій платформи Kafka та реплік споживачів повідомлень;

– перевірити дотримання інваріанту подвійного бухгалтерського запису під різними рівнями навантаження;

– підтвердити можливість зовнішньої верифікації історії платежів торговцем через криптографічні докази незмінності.

Очікувані результати:

– Архітектурна модель системи та функціонуючий прототип, розгорнутий у локальному кластері Kubernetes;

– Реалізований подієво-орієнтований конвеєр обробки платежів з асинхронною комунікацією між компонентами;

– Експериментальні графіки пропускної здатності та затримки від параметрів системи, а також результати перевірки фінансових інваріантів;

– Демонстрація механізму криптографічної довіри через використання незмінного сховища ImmuDB з можливістю незалежної верифікації записів.

ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ РОЗПОДІЛЕНОГО ПЛАТІЖНОГО ТЕРМІНАЛУ

2.1 Теоретичні засади подвійного запису в розподілених системах

Подвійний бухгалтерський запис є фундаментальною моделлю фінансового обліку, що використовується у банківських установах, платіжних системах та корпоративних бухгалтеріях для гарантування коректності обліку руху коштів. Його ключова ідея полягає в тому, що кожна фінансова операція відображається одночасно у двох взаємопов'язаних проводках: дебетовій та кредитовій. Такий підхід забезпечує математичну інваріантність балансу, коли сума всіх дебетових записів дорівнює сумі всіх кредитових записів як для окремої транзакції, так і для системи в цілому. Звідси випливає важлива властивість: будь-яке порушення цього інваріанту одразу сигналізує про наявність помилки в обліку або фальсифікацію фінансових даних.

У контексті криптовалютного терміналу для масових платежів подвійний бухгалтерський запис дозволяє формалізувати рух коштів між рахунками торговця, системними рахунками та рахунками сервісної комісії, гарантуючи відсутність у фінальній історії операцій надлишкових або втрачених записів. Однак у розподілених системах виникає специфічна проблема: необхідно зберегти ці фінансові інваріанти за умов паралельної обробки подій з різних джерел, шардування даних між вузлами кластера і можливих часткових відмов

окремих компонентів інфраструктури. Саме тому подвійний бухгалтерський запис у такій архітектурі повинен базуватися на узгоджених транзакціях з ACID-властивостями та чітко визначеній моделі атомарної фінансової операції з гарантованим дотриманням балансу.

2.1.1 Походження та еволюція подвійного запису

Подвійний бухгалтерський запис сформувався як відповідь на потребу систематичного контролю фінансових операцій у торгівельній діяльності та банківській справі. Його виникнення пов'язане з розвитком комерційних відносин, коли стало критично важливо не лише фіксувати факти платежів, а й мати надійний механізм автоматичної перевірки їх взаємної узгодженості. Історично ця модель еволюціонувала до загальноприйнятого стандарту, який сьогодні становить фундаментальну основу бухгалтерських планів рахунків, банківських кореспондентських схем розрахунків та систем процесингу платежів.

2.1.2 Формальна модель подвійного запису і фінансові інваріанти

У подвійній бухгалтерії транзакція описується як перенесення вартості між рахунками. Формально це можна подати як набір проводок $L = \{l_1, l_2, \dots, l_n\}$, де кожна проводка має сторону Debit або Credit і суму. Інваріант транзакції визначається рівністю:

$$\sum_{l \in L, side=D} amount(l) = \sum_{l \in L, side=C} amount(l)$$

Ця математична рівність гарантує фундаментальну властивість закону збереження вартості у фінансовій системі: транзакція не може створювати або знищувати вартість довільно, а лише переносить її між рахунками учасників системи, забезпечуючи нульовий чистий баланс кожної операції.

2.1.3 Переваги подвійного запису над одинарним для платіжних систем

Одинарний запис фіксує операцію лише з одного боку обліку, наприклад як факт отримання платежу. У такій моделі відсутній вбудований механізм контролю коректності: необхідно окремо звіряти журнали операцій, можливі прогалини в історії транзакцій, дублювання записів або неузгоджені суми між різними частинами системи.

Подвійний бухгалтерський запис принципово вирішує ці проблеми завдяки кільком ключовим властивостям. По-перше, він надає можливість локальної перевірки коректності кожної транзакції через дотримання інваріанту нульового балансу. По-друге, дозволяє виявляти помилки безпосередньо на етапі фіксації запису, а не після процедури звірки балансів. По-третє, ефективно масштабується як модель для обліку складних фінансових потоків, включаючи сервісні комісії, повернення коштів та часткові платежі. По-четверте, є загальноприйнятим міжнародним стандартом для процедур фінансового аудиту.

Для архітектури криптовалютного терміналу це означає, що кожен успішно обробленийрахунок-фактура породжує мінімум дві бухгалтерські проводки: дебетовий запис на суму вхідного платежу та кредитовий запис на суму, зараховану на рахунок торговця, з окремою кредитовою проводкою на величину сервісної комісії платформи.

2.1.4 Атомарність фінансової транзакції

Критична вимога до системи полягає в атомарності транзакції: або всі її бухгалтерські проводки фіксуються одночасно, або не фіксується жодна з них. Інакше можлива ситуація часткового запису, коли дебетову проводку зафіксовано, а кредитову - ні, що призводить до негайного порушення фінансового інваріанту та виникнення помилкового балансу в системі обліку.

Отже, фінансова транзакція в журналі обліку розглядається як неділима атомарна операція, що включає створення заголовку запису журналу з метаданими транзакції, вставку всіх рядків бухгалтерських проводок із зазначенням дебетових та кредитових записів, а також фіксацію транзакції через

механізм підтвердження змін. Ця модель безпосередньо відображається на механізм транзакцій у SQL-орієнтованих системах керування базами даних з підтримкою ACID-властивостей.

2.1.5 Проблема підтримки інваріантів у розподіленій системі

У централізованій базі даних подвійний бухгалтерський запис реалізується відносно просто через використання однієї транзакції системи керування базою даних з наступною перевіркою дотримання балансу.

У розподіленому середовищі виникають додаткові виклики, пов'язані з паралельною обробкою подій для різних торговців, шардуванням таблиць і фізичним розподілом записів між вузлами кластера, ризиком конкурентних операцій запису в один журнал обліку, а також можливими затримками або відмовами окремих вузлів інфраструктури, що можуть перервати процес фіксації транзакції.

Тому для забезпечення коректності обліку необхідно гарантувати послідовність обробки подій для конкретного торговця або рахунку-фактури через механізм партиціонування у платформі Kafka, виконувати запис усіх бухгалтерських проводок у межах однієї узгодженої транзакції бази даних з ACID-властивостями, а також забезпечити ідемпотентність споживачів повідомлень, щоб повторна обробка однієї події не призводила до створення дублікатів записів у журналі обліку.

2.1.6 Роль ACID-транзакцій у забезпеченні консистентності

ACID-властивості становлять теоретичну основу надійного подвійного бухгалтерського запису у розподіленому SQL-орієнтованому сховищі даних:

– Атомарність (Atomicity) гарантує, що всі бухгалтерські проводки однієї фінансової транзакції фіксуються одночасно як неділима операція.

– Узгодженість (Consistency) означає, що після підтвердження транзакції в системі керування базою даних не залишається станів, у яких інваріант балансу між дебетом і кредитом порушено.

– Ізоляція (Isolation) захищає від взаємного впливу паралельних транзакцій, наприклад забезпечуючи, що дві незалежні події не можуть одночасно модифікувати записи одного журналу обліку.

– Довговічність (Durability) забезпечує збереження зафіксованих записів навіть за умов відмови окремих вузлів кластера після підтвердження транзакції.

У результаті, подвійний бухгалтерський запис у прототипі криптовалютного терміналу базується на ACID-транзакціях розподіленої системи керування базою даних YugabyteDB як механізм практичної реалізації теоретичних інваріантів фінансового обліку в розподіленому середовищі.

2.2 Теорія розподілених транзакцій і консистентності

Фінансові системи для масових платежів завжди функціонують у середовищі, де дані розподілені між множиною вузлів кластера, а події надходять з різних джерел паралельно. Саме в такому розподіленому середовищі виникає ключове питання: як забезпечити коректність стану системи, включаючи баланси рахунків та записи журналу обліку, за умов можливих відмов компонентів, мережевих затримок і конкурентних операцій запису. Теорія розподілених обчислювальних систем формулює фундаментальні обмеження та пояснює, чому вибір архітектурного рішення для криптовалютного платіжного терміналу не може бути довільним і має враховувати властивості узгодженості, доступності та стійкості до розділення мережі.

2.2.1 CAP-теорема

CAP-теорема стверджує фундаментальне обмеження розподілених систем: при виникненні мережевого розділення (Partition tolerance) неможливо одночасно забезпечити строгу узгодженість даних (Consistency) та постійну доступність системи (Availability). Іншими словами, коли мережа

фрагментується на ізольовані сегменти, система має обрати один з двох варіантів поведінки:

– забезпечувати доступність та відповідати всім запитам клієнтів, навіть ризикуючи тимчасовою неузгодженістю даних між вузлами (пріоритет доступності);

– призупиняти частину операцій або відхиляти запити до недоступних сегментів, гарантуючи відсутність розбіжностей у даних (пріоритет узгодженості).

Для будь-якої системи, що функціонує в реальному мережевому середовищі, стійкість до розділення є обов'язковою властивістю. Тому практичний архітектурний вибір зводиться до компромісу між узгодженістю та доступністю.

2.2.2 CP чи AP і їх наслідки для платежів

Модель CP (узгодженість плюс стійкість до розділення) передбачає, що система може тимчасово відмовляти у виконанні операцій запису або читання під час мережевого розділення, натомість гарантуючи єдине коректне значення даних у всіх доступних вузлах кластера. Модель AP (доступність плюс стійкість до розділення) забезпечує постійну відповідь системи на запити клієнтів, але допускає ситуацію, коли різні вузли можуть тимчасово мати різні версії стану з подальшою eventual consistency, тобто узгодженістю в кінцевому підсумку. Для фінансових операцій AP-підхід є неприйнятним через високі ризики: якщо різні вузли системи бачать різні значення балансів або різні стани рахунків-фактур, можливе виникнення подвійного запису транзакцій, некоректного нарахування комісії або навіть повторного зарахування платежу на рахунок торговця. Отже, в архітектурі криптовалютного платіжного терміналу пріоритет має CP-логіка з гарантованою узгодженістю фінансових записів навіть ціною тимчасового зниження доступності окремих операцій.

2.2.3 Чому фінансові системи потребують strong consistency

Фінансовий журнал обліку має жорсткі інваріанти, що визначають його коректність: транзакція або записана повністю з усіма проводками, або не записана зовсім; сума дебетових записів дорівнює сумі кредитових записів для кожної операції; баланс торговця не може бути відкликаний ретроспективно або мати різні значення при читанні з різних вузлів системи. Ці вимоги означають необхідність строгої узгодженості даних: після підтвердження транзакції будь-яка операція читання має повертати ідентичний стан незалежно від того, який вузол кластера обробляє запит. Навіть короткочасна eventual consistency, тобто узгодженість у кінцевому підсумку, є неприйнятною для фінансового обліку, оскільки вона перетворює точний фінансовий стан на приблизне значення, що суперечить фундаментальним принципам бухгалтерського обліку.

2.2.4 ACID як основа надійної фінансової фіксації

ACID-властивості конкретизують вимоги строгої узгодженості на рівні транзакцій системи керування базою даних. Атомарність забезпечує, що подвійний бухгалтерський запис не може бути розділений на часткові операції з фіксацією лише частини проводок. Узгодженість гарантує, що база даних не фіксує станів, які порушують інваріант балансу між дебетом і кредитом. Ізоляція запобігає ситуації, коли конкурентні платіжні операції змішують бухгалтерські проводки або створюють стани гонитви даних. Довговічність забезпечує збереження зафіксованого журналу обліку навіть за умов відмов компонентів системи.

Тому для журналу подвійного бухгалтерського запису необхідна система керування базою даних, що надає ACID-властивості не лише на рівні окремого вузла, а у масштабі всього розподіленого кластера з гарантіями узгодженості між усіма його компонентами.

2.2.5 PACELC: ціна консистентності без аварій

CAP-теорема описує поведінку системи під час мережевих розділень. PACELC-теорема розширює цей аналіз, охоплюючи обидва режими функціонування: якщо виникає розділення мережі (Partition), система обирає між доступністю (Availability) та узгодженістю (Consistency); у протилежному випадку (Else), коли мережа функціонує нормально без розділень, система має обирати між низькою затримкою обробки (Latency) та строгою узгодженістю даних (Consistency).

Іншими словами, навіть у нормальному режимі роботи без мережевих аномалій строга узгодженість даних має свою ціну - збільшення затримки обробки операцій, оскільки потребує синхронізації та узгодження стану між вузлами кластера через консенсусні протоколи. Для системи масових криптовалютних платежів це означає архітектурний компроміс: система певною мірою втрачає у швидкості обробки окремих транзакцій, але натомість отримує гарантовану коректність фінансового стану, що є критично важливішою характеристикою, ніж мінімальна затримка відповіді.

2.2.6 Розподілений SQL як практична реалізація CP та ACID

Класична система PostgreSQL забезпечує ACID-властивості лише в межах одного вузла сервера. Розподілені SQL-орієнтовані системи, такі як YugabyteDB та CockroachDB, розширюють ці гарантії на рівень всього кластера через низку механізмів: дані автоматично шардуються та розподіляються між вузлами за ключами партиціонування; транзакції узгоджуються через консенсусні протоколи на основі алгоритму Raft; операції читання та запису зберігають єдиний послідовний порядок подій у всій системі.

У контексті журналу подвійного бухгалтерського обліку це означає можливість горизонтального масштабування бази даних для обробки зростаючих обсягів транзакцій при збереженні строгих гарантій фінансової узгодженості, що є критично важливим для коректності платіжних операцій.

Основні принципи функціонування розподілених SQL-систем базуються на трьох ключових механізмах. Шардування даних забезпечує автоматичний розподіл таблиць за ключами партиціонування між вузлами кластера, що дозволяє досягти горизонтального масштабування системи зі збереженням локальності доступу до пов'язаних записів. Консенсусний протокол на основі алгоритму Raft організує роботу таким чином, що кожен шард має кілька реплік, серед яких одна виконує роль лідера для обробки операцій запису. Усі зміни даних проходять через узгоджений консенсусний журнал, що гарантує єдину послідовність застосування транзакцій та забезпечує стійкість до відмов окремих вузлів.Транзакційна цілісність досягається завдяки механізмам розподілених двофазних підтверджень та детермінованим алгоритмам призначення міток часу, що зберігають строгий порядок операцій у межах всієї розподіленої системи та забезпечують серіалізовність транзакцій навіть при їх паралельному виконанні на різних вузлах кластера.

2.2.7 Оптимістичні чи песимістичні блокування

Для підтримки фінансових інваріантів у конкурентному середовищі з паралельними операціями застосовують дві базові стратегії управління конкурентним доступом. Песимістичне блокування передбачає, що транзакція одразу встановлює блокування на відповідні рядки або діапазони записів, і всі інші транзакції, що потребують доступу до цих даних, переходять у стан очікування. Цей підхід є надійним і передбачуваним, проте може суттєво зменшувати рівень паралелізму обробки операцій у системі. Оптимістичний підхід дозволяє транзакції виконувати операції без встановлення блокувань, але перед підтвердженням змін система перевіряє, чи не були дані модифіковані іншими транзакціями. У разі виявлення конфлікту відбувається відкат змін і повторна спроба виконання операції. Ця стратегія є ефективною для сценаріїв високої конкуренції за доступом до різних ключів партиціонування.У системі криптовалютного терміналу конкуренція за доступом до даних здебільшого розділена між різними торговцями та рахунками-фактурами, тому оптимістичні

транзакції в розподілених SQL-системах добре узгоджуються з моделлю обробки масових однотипних платіжних операцій, забезпечуючи високу пропускну здатність системи.

2.3 Теорія подієвих систем

Подієво-орієнтована архітектура виникла як відповідь на потребу обробляти значні потоки однотипних операцій у режимі реального часу без створення жорстких синхронних залежностей між компонентами системи. Для систем масових платежів це є критично важливим: транзакції надходять паралельно з різних джерел, зокрема з різних блокчейн-мереж, а процес фіксації бухгалтерського обліку повинен забезпечувати надійність, масштабованість і відтворюваність результатів. Подієвий підхід дозволяє декомпозувати систему на незалежні мікросервіси, які взаємодіють через потоки подій у розподіленій шині обміну повідомленнями, зберігаючи гарантований порядок обробки та надійність доставки повідомлень.

2.3.1 Поняття події та її роль у системі

Подія являє собою незмінне повідомлення про факт, який уже відбувся в системі або зовнішньому середовищі, наприклад у блокчейн-мережі. Події характеризуються низкою важливих властивостей.

Фактова природа подій полягає в тому, що вони описують завершені дії через іменники минулого часу, такі як створення рахунку-фактури, підтвердження транзакції або збереження запису у журналі обліку.

Незмінність означає, що події не підлягають редагуванню після їх публікації, а лише послідовно додаються до потоку подій, формуючи незмінну історію змін стану системи.

Самодостатність передбачає, що подія містить усі дані, необхідні для реакції інших сервісів-споживачів, без потреби у додаткових запитах до зовнішніх джерел.

Часова прив'язка забезпечується через наявність мітки часу, що фіксує момент виникнення події та її фіксації в системі для встановлення хронологічного порядку.

У фінансових системах подія виступає первинним джерелом достовірної інформації про зміну стану, що дозволяє відтворити послідовність операцій і верифікувати коректність обробки транзакцій.

2.3.2 Потік подій і журнал як джерело істини

Потік подій являє собою впорядковану послідовність повідомлень про події, що безперервно поповнюється новими записами. За своєю суттю це розподілений журнал змін стану системи з низкою ключових переваг.

Можливість відтворення стану дозволяє повністю реконструювати стан системи з початкового моменту шляхом послідовного програвання всіх подій з журналу, що є критично важливим для відновлення після збоїв.

Повний аудит забезпечується через збереження повної історії змін без прогалин або видалень записів, що дозволяє верифікувати послідовність операцій у будь-який момент часу.

2.3.3 Гарантії доставки подій

У подієво-орієнтованих системах визначають три базові моделі семантики доставки повідомлень.

Максимум один раз (At-most-once): подія доставляється не більше одного разу, але може бути втрачена під час передачі. Перевагою цієї моделі є мінімальна затримка обробки через відсутність механізмів підтвердження. Проте для платіжних систем цей підхід є неприйнятним, оскільки втрата події призводить до втрати інформації про транзакцію та порушення цілісності фінансового обліку.

Принаймні один раз (At-least-once): подія доставляється гарантовано, але може бути продубльована під час повторних спроб передачі. Перевагою є надійність доставки, що робить цю модель придатною для фінансових потоків

даних. Недоліком є необхідність забезпечення ідемпотентності на рівні споживача повідомлень, тобто здатності безпечно обробляти дублікати без порушення коректності стану системи.

Рівно один раз (Exactly-once): подія доставляється і обробляється рівно один раз без втрат і дублікатів. Хоча теоретично це ідеальна модель, практична реалізація є складною та ресурсоємною, оскільки вимагає тісного узгодження між журналом подій і базою даних через спеціалізовані транзакційні протоколи з двофазним підтвердженням.

Вибір моделі для фінансових систем

У фінансових системах найчастіше застосовують комбінацію “принаймні один раз” з ідемпотентністю обробки, оскільки цей підхід є технічно простішим у реалізації та забезпечує ті самі гарантії коректності кінцевого результату, що й складніша модель “рівно один раз”.

Для архітектури криптовалютного терміналу природним є використання моделі “принаймні один раз”: навіть якщо платформа Kafka повторно доставить подію про транзакцію, споживач журналу обліку не повинен створювати другий запис для того самого рахунку-фактури завдяки механізмам перевірки унікальності ідентифікаторів та ідемпотентності операцій запису.

2.4 Теоретичні моделі масштабування

Масштабованість є ключовою властивістю платіжних систем масового застосування, оскільки навантаження на інфраструктуру зростає пропорційно до кількості торговців, рахунків-фактур і подій від блокчейн-мереж. На теоретичному рівні масштабування розглядають як здатність системи підтримувати прийнятні параметри пропускної здатності та затримок обробки при експоненційному зростанні потоку запитів і обсягів даних. Для досягнення цієї мети застосовують комплекс архітектурних підходів: горизонтальне масштабування через додавання вузлів кластера, шардування даних для розподілу навантаження, реплікацію для забезпечення доступності та механізми стійкості до часткових відмов компонентів системи.

2.4.1 Горизонтальне масштабування

Горизонтальне масштабування означає збільшення продуктивності системи через додавання нових вузлів інфраструктури або екземплярів сервісів до існуючого кластера. На відміну від вертикального масштабування, де зростає обчислювальна потужність окремого вузла через оновлення апаратного забезпечення, горизонтальний підхід характеризується низкою переваг.

По-перше, він краще відповідає характеру зростання навантаження у веб-орієнтованих та фінансових системах, де обсяг запитів може зростати непередбачувано. По-друге, підвищує загальну відмовостійкість системи, оскільки збій окремого екземпляра не призводить до повної зупинки функціонування завдяки наявності інших активних вузлів. По-третє, вимагає впровадження механізмів балансування навантаження та розподілу даних або подій між вузлами для ефективного використання ресурсів.

У мікросервісній подієво-орієнтованій архітектурі горизонтальне масштабування відбувається органічно: додавання екземплярів споживачів повідомлень збільшує паралелізм обробки потоків подій через розподіл партицій, а нарощування вузлів розподіленої бази даних підвищує як обсяг зберігання, так і швидкість виконання операцій запису та читання.

2.4.2 Моделі шардування даних

Шардування являє собою метод розподілу даних між кількома вузлами кластера для зменшення навантаження на кожен окремий вузол та підвищення загальної пропускної здатності системи. Виділяють дві базові моделі шардування з різними характеристиками продуктивності: хеш-шардування та діапазонне шардування.

У хеш-шардування дані розподіляються між вузлами на основі хеш-функції від ключа запису, наприклад ідентифікатора торговця або ідентифікатора запису журналу. Цей підхід забезпечує рівномірний розподіл навантаження між вузлами завдяки псевдовипадковим властивостям хеш-функцій, зменшує ймовірність виникнення критичних точок навантаження на

окремих вузлах та забезпечує прогнозовану масштабованість при зростанні кількості унікальних ключів у системі.

Проте хеш-шардування ускладнює виконання діапазонних запитів за часовими інтервалами або порядком подій, оскільки потребує сканування всіх шардів для отримання відсортованих результатів, що знижує ефективність аналітичних операцій.

В діапазонному шардуванні дані розподіляються між вузлами на основі діапазонів значень ключів, наприклад часових міток створення записів або інтервалів ідентифікаторів торговців. Перевагою цього підходу є висока ефективність діапазонних запитів, таких як отримання транзакцій за певний період часу або вибірка останніх операцій, оскільки релевантні дані концентруються на обмеженій кількості шардів.

Недоліком є ризик нерівномірного розподілу навантаження, коли активність концентрується в одному часовому або ідентифікаційному діапазоні, що призводить до виникнення критичних точок навантаження на окремих вузлах та погіршення загальної продуктивності системи.

2.4.3 Реплікація і модель лідер–послідовник

Реплікація являє собою підтримку кількох синхронізованих копій даних на різних вузлах для підвищення надійності зберігання та зменшення затримок при операціях читання. Найпоширенішою моделлю реплікації є архітектура лідер–послідовник.

У цій моделі вузол-лідер (первинний) приймає всі операції запису та формує послідовний журнал змін стану даних, тоді як вузли-послідовники (репліки) асинхронно або синхронно відтворюють ці зміни на своїх локальних копіях і можуть обслуговувати запити на читання для розвантаження лідера.

Переваги моделі лідер–послідовник включають підвищення загальної доступності системи через механізм автоматичного переключення на резервний вузол при відмові лідера, розвантаження операцій читання через їх розподіл між

репліками, а також можливість географічно розподіленого розміщення реплік для зменшення мережевих затримок для користувачів у різних регіонах.

Недоліками цього підходу є затримка реплікації, коли дані на вузлах-послідовниках можуть бути дещо застарілими порівняно з лідером, що призводить до можливості читання неактуальних значень, а також складність забезпечення строгої узгодженості даних при значних мережевих затримках між географічно віддаленими вузлами кластера.

У фінансових системах критично важливо чітко розділяти типи операцій читання: критичні читання, що безпосередньо впливають на обчислення балансів та прийняття фінансових рішень, мають виконуватися з гарантіями актуальності даних, тоді як аналітичні запити для звітності можуть допускати невелику затримку реплікації без порушення коректності системи.

2.4.4 Стійкість до збоїв та стійкість до мережевих розривів

Стійкість до збоїв (fault tolerance) являє собою здатність системи зберігати працездатність та продовжувати виконання критичних функцій при відмові частини компонентів інфраструктури, таких як окремі вузли обчислювального кластера, модулі програмного забезпечення або дискові накопичувачі. Ця властивість досягається через механізми реплікації даних на кількох вузлах, автоматичного перевибору вузла-лідера при його відмові та відновлення стану системи з незмінних журналів подій.

Стійкість до мережевих розривів (partition tolerance) означає здатність розподіленої системи продовжувати функціонування при розділенні комунікаційної мережі на ізольовані сегменти, що тимчасово втрачають зв'язок між собою. У розподілених системах мережеві розриви є неминучим явищем через нестабільність каналів зв'язку, тому архітектурне проєктування завжди має включати сценарії обробки ситуацій, коли частина кластера стає тимчасово недоступною для решти вузлів.

2.4.5 Зворотний тиск як механізм стабілізації потоку

Зворотний тиск (backpressure) являє собою механізм саморегуляції розподіленої системи, коли компоненти-споживачі сигналізують продюсерам або попереднім етапам конвеєра обробки про надмірну швидкість надходження даних. У термінах теорії черг це можна формалізувати наступним чином: якщо інтенсивність надходження запитів λ перевищує інтенсивність їх обробки μ, то довжина черги зростає необмежено, що призводить до виснаження ресурсів системи.

Механізм зворотного тиску реалізується через комплекс стратегій управління потоком: обмеження швидкості генерації подій продюсерами через встановлення верхньої межі пропускної здатності; пакетну обробку повідомлень із контролем розміру партій для оптимізації використання ресурсів; динамічне призупинення та відновлення роботи споживачів залежно від поточного навантаження; застосування повторних спроб обробки з експоненційним зростанням затримки для запобігання перевантаженню при тимчасових збоях.

Без застосування механізмів зворотного тиску система під високим навантаженням починає деградувати: зростають затримки реплікації та обробки повідомлень, збільшується кількість таймаутів операцій, і зрештою відмови окремих компонентів каскадно поширюються на всю систему, призводячи до повної втрати працездатності.

2.5 Формальна постановка гіпотез дослідження

Дослідницька частина роботи зосереджена на експериментальній верифікації того, що запропонований прототип криптовалютного терміналу забезпечує коректність фінансового обліку через дотримання бухгалтерських інваріантів, масштабованість подієво-орієнтованого конвеєра обробки при зростаючих навантаженнях та криптографічну незмінність історії транзакцій для можливості незалежного аудиту. Для досягнення цієї мети формалізовано систему інваріантів подвійного бухгалтерського запису, метрики оцінки продуктивності розподіленої системи, вимоги до узгодженості даних між

компонентами архітектури та властивості криптографічної верифікації записів у незмінному сховищі даних.

Гіпотеза H1: для кожної фінансової операції, зафіксованої у журналі обліку, виконується фундаментальний інваріант подвійного бухгалтерського запису, що забезпечує коректність фінансових розрахунків у системі.

Гіпотеза H2: подієво-орієнтований конвеєр обробки, що включає платформу Kafka, споживача обробки транзакцій, споживача журналу обліку та споживача незмінного сховища, має забезпечувати зростання пропускної здатності при горизонтальному масштабуванні через збільшення кількості партицій та екземплярів споживачів. Метод експериментальної перевірки: проведення серії експериментів із симулятором навантаження при різних конфігураціях кількості партицій і споживачів з фіксацією показників пропускної здатності у транзакціях за секунду та затримки обробки. Критерій успішності: при збільшенні кількості партицій і споживачів у N разів пропускна здатність системи має зростати.

Гіпотеза H3: незмінне сховище ImmuDB використовується для забезпечення криптографічних доказів включення записів та узгодженості історії, що створює додатковий рівень довіри з боку торговців до неможливості ретроспективної модифікації історії їх операцій. Метод експериментальної перевірки: для репрезентативної вибірки транзакцій отримуються криптографічні докази через програмний інтерфейс gRPC та перевіряються зовнішнім незалежним інструментом верифікації. Критерій успішності: незалежна верифікація криптографічних доказів є успішною для 100% перевірених записів, а будь-яка спроба модифікації історичних даних призводить до невалідності криптографічного доказу та його відхилення при верифікації.

ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Загальна архітектура розподіленого платіжного терміналу

Запропонована система криптовалютного терміналу для масових платежів реалізована на основі мікросервісної подієво-орієнтованої архітектури. Її призначення полягає в надійній обробці великих обсягів паралельних криптовалютних операцій з боку клієнтів завдяки застосуванню розподіленого подвійного запису, забезпеченню фінансових інваріантів та можливості незалежного аудиту історії транзакцій із наданням криптографічних доказів.

Архітектура системи структурована на три рівні:

- Сервіси взаємодії: реалізують бізнес-логіку системи та координують виконання основних операційних процесів;
- Сервіси споживачі: здійснюють асинхронну обробку подій, що надходять від різних компонентів системи;
- Інфраструктурний рівень: забезпечує розподілене зберігання та обробку даних.

Ключовою особливістю архітектурного рішення є розподілена система обробки потоків запитів та читання даних. Фінальний запис транзакцій фіксується у YugabyteDB з подальшим записом у незмінному сховищі ImmuDB для формування криптографічних доказів автентичності операцій. Подієва взаємодія між компонентами системи реалізована через розподілену платформу обміну повідомленнями Kafka, що гарантує керований порядок обробки та горизонтальне масштабування без блокування записів.

3.1.1 Компонентна діаграма системи

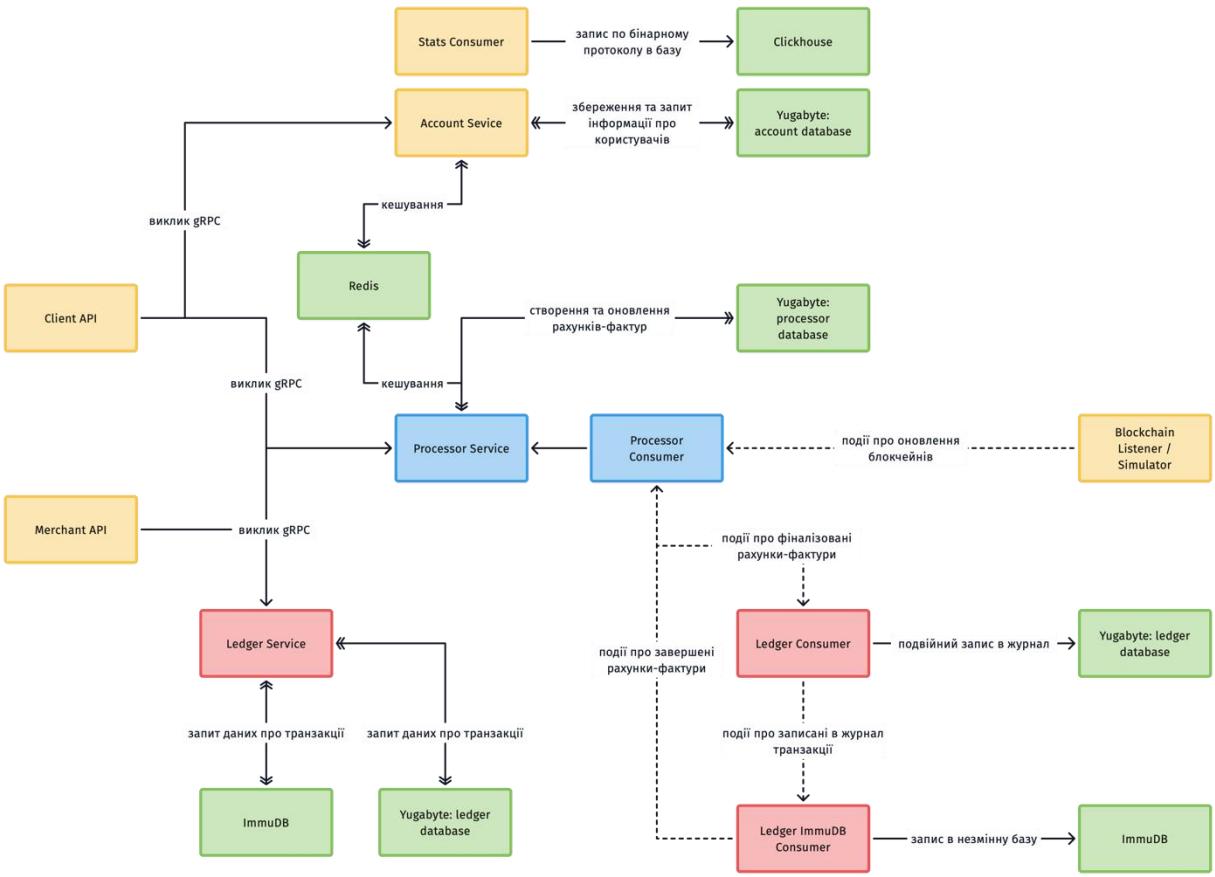


Рисунок 3.1 - Розгорнута компонента схема системи

Компонентна структура системи включає такі групи елементів:

– Account Service: сервіс ідентифікації та доступу мерчантів. Відповідає за створення облікових записів, автентифікацію, керування API-ключами та їх перевірку. Виконує роль центрального IAM-компонента, через який проходить доступ до інших сервісів.

– Processor Service: обробник платежів і життєвого циклу інвойсів. Реалізує створення інвойсів мерчантами, вибір валюти/мережі, прив’язку платіжного гаманця, перевірку статусів і розрахунок комісії сервісу.

– Ledger Service: надає gRPC API для отримання інформації про транзакції та запиту їх криптографічних доказів.

– Ledger Consumer: отримує ProcessedTxEvent. Створює записи подвійного журналу у YugabyteDB транзакційно, оновлює баланси мерчанта та системного мерчанта, і публікує LedgerTxEvent про успішне збереження.

– Ledger Immu Consumer: обробляє LedgerTxEvent, виконує VerifiableSet у випадково обраному шарді ImmuDB, витягує tx_id та корневий Merkle хеш, записує метадані незмінності у YugabyteDB і публікує LedgerConfirmedTxEvent.

– Processor Consumer: слухає блокчейн-події з Kafka. Зіставляє транзакцію з активним інвойсом, виконує конвертацію в USD, перевіряє достатність суми та ініціює фіналізацію інвойсу в Processor Service, після чого публікує подію ProcessedTxEvent. Також, слухає події LedgerConfirmedTxEvent про успішно записані в базу транзакції та оновлює статусти рахунків фактур, що їх ініціювали.

– Stats Consumer: слухає технічні події різних етапів обвобки і записує їх у ClickHouse для подальшого обчислення метрик продуктивності й затримок.

– Simulator: генерує синтетичні інвойси і події блокчейну для навантажувального тестування системи та перевірки гіпотез дослідження в локальному Kubernetes-кластері.

Інфраструктура:

– YugabyteDB - основна транзакційна БД для users, invoices, wallets, journals та balances з ACID-гарантіями.

– Redis Cluster - кеш API-ключів, легкі локи (rate limit / select currency lock), гарячі дані.

– ImmuDB Shards - незмінний журнал з Merkle-деревом, що генерує криптографічні докази включення й консистентності.

– ClickHouse - аналітичний шар для метрик і агрегацій.

Усі компоненти розгортаються у системі оркестрації Kubernetes, що дозволяє незалежно масштабувати сервіси, брокери Kafka та кластери сховищ.

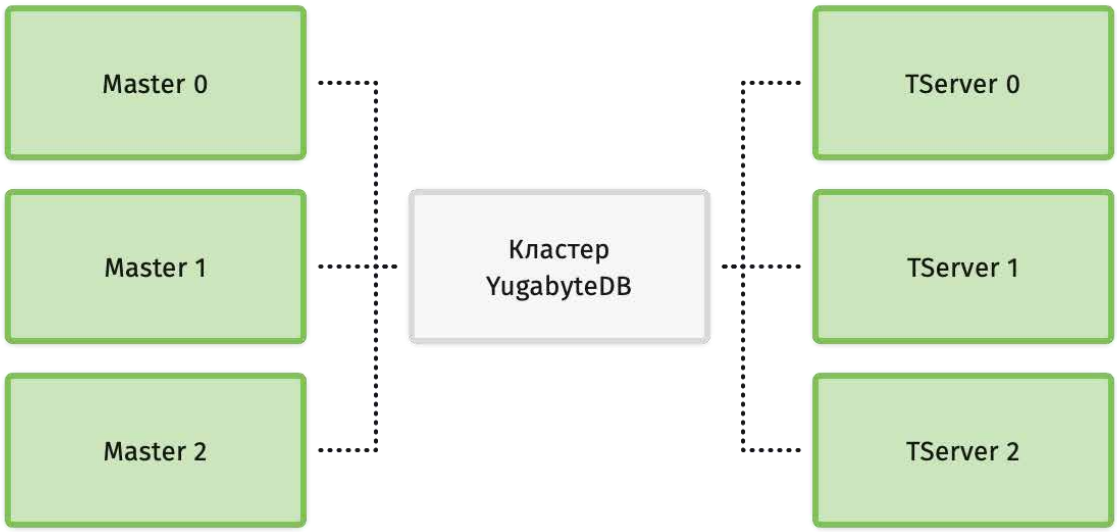


Рисунок 3.2 - Архітектура кластера YugabyteDB

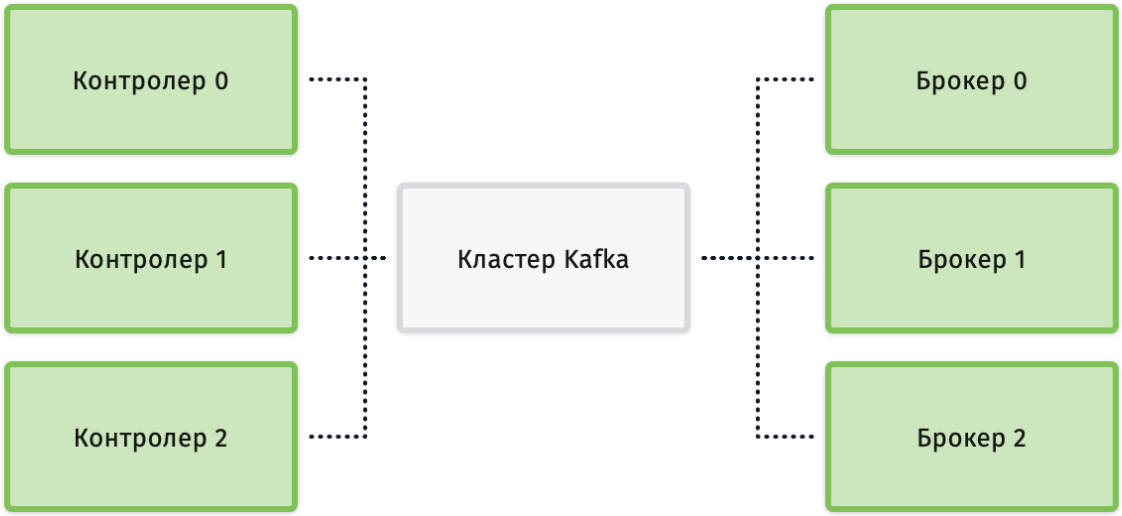


Рисунок 3.3 - Архітектура кластера Kafka

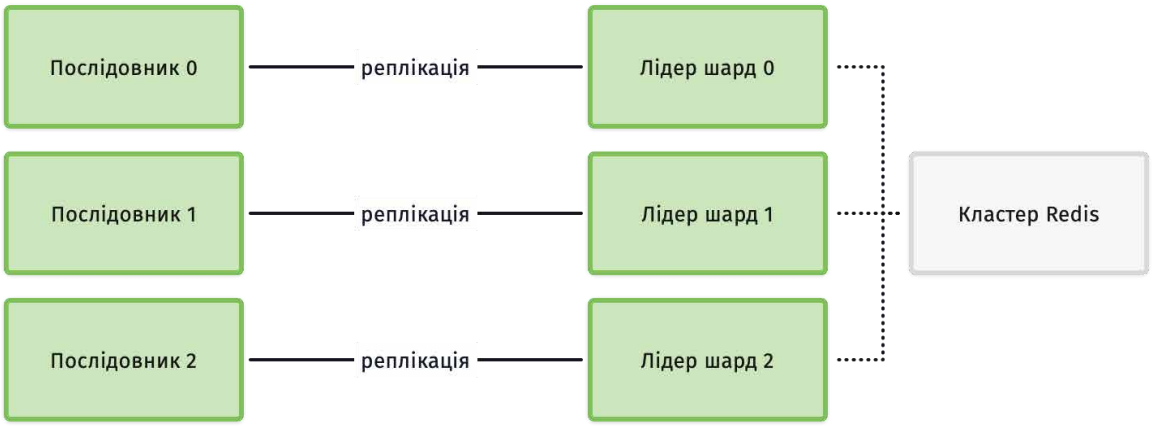


Рисунок 3.4 - Архітектура кластеру Redis в моделі Лідер-послідовник

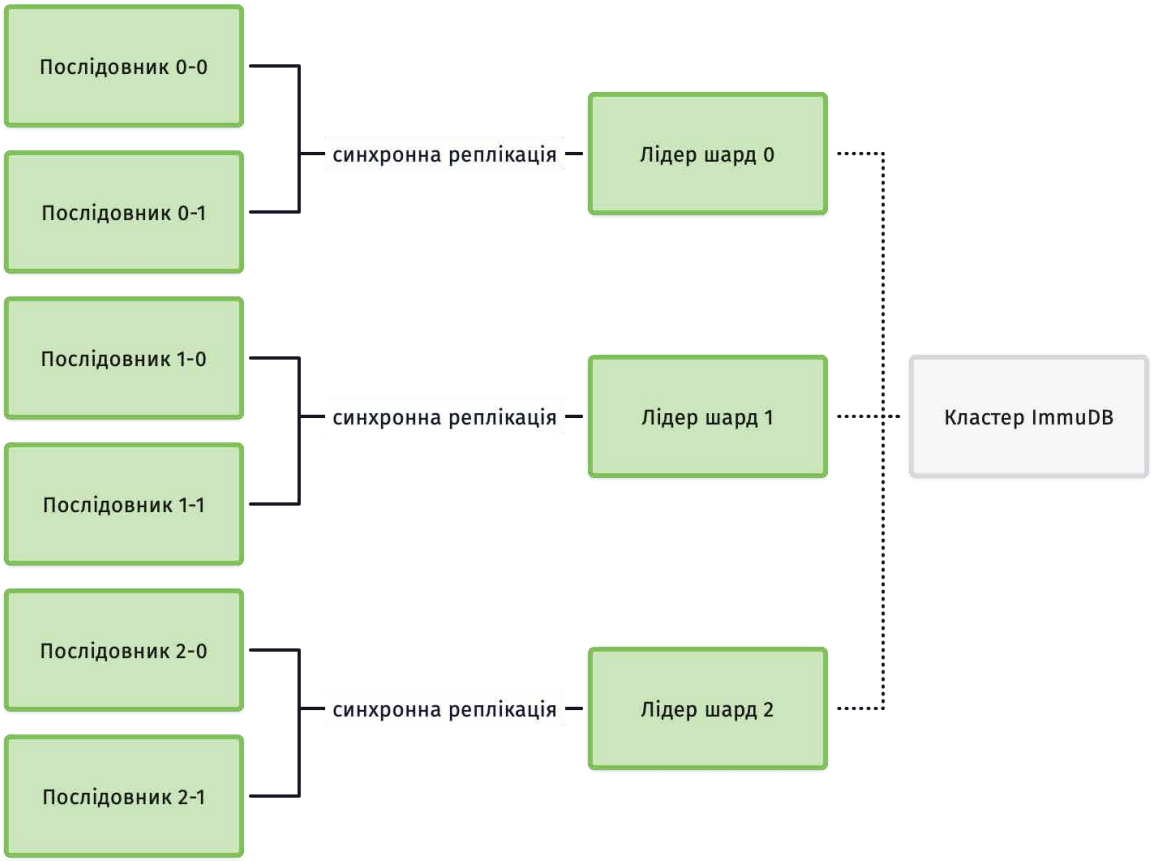


Рисунок 3.5 - Архітектура кластеру ImmuDB з логічними шардами та синхронною реплікацією

3.1.2 Потоки даних між мікросервісами

Обробка платежу відбувається як послідовність синхронних викликів і асинхронних подій.

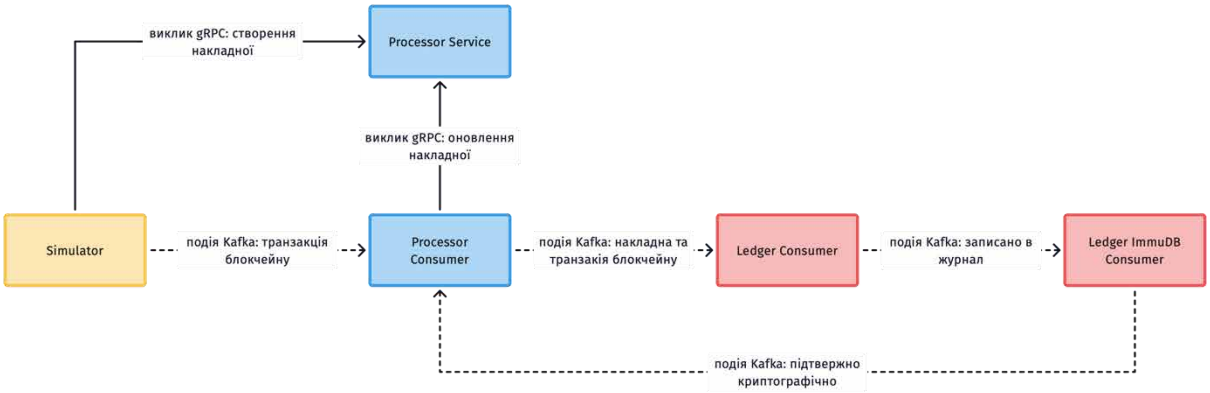


Рисунок 3.6 - Потік данних в системи при обробці рахунку фактури

1. Створення рахунку-фактури (invoice)

Торговець (merchant) звертається до Processor Service через публічний API. Processor валідує права доступу через Account Service з використанням методу VerifyApiKey. Рахунок-фактура створюється у розподіленій базі даних YugabyteDB та кешується у системі Redis для швидкого доступу. Новостворений invoice переходить у стан pending, очікуючи вибору валюти та оплати.

2. Вибір криптовалюти та резервування гаманця (wallet)

Клієнт-покупець викликає метод SelectInvoiceCurrency для вибору пари криптовалюта–блокчейн-мережа. Processor під захистом атомарного lock-механізму у Redis резервує доступну адресу wallet у базі даних YugabyteDB. Invoice отримує призначену wallet_address і стає готовим до прийому платежу від клієнта.

3. Надходження події з блокчейн-мережі

Blockchain-сканер, у прототипі представлений компонентом Simulator, публікує подію BlockchainTxEvent до Kafka topic blockchains.tx.v1. Processor Consumer читає подію з черги, ідентифікує відповідний invoice за wallet_address і перевіряє достатність отриманої суми. За умови достатності суми система розраховує service_fee та amount_usd_merchant, фіналізує invoice через синхронний gRPC-виклик до Processor Service і публікує подію ProcessedTxEvent

до topic processor.tx.v1. Invoice переходить у стан finalized з фіксацією timestamps paid_at та finalized_at.

4. Запис у журнал подвійного обліку (double-entry ledger)

Ledger Consumer читає подію ProcessedTxEvent з відповідного Kafka topic. Компонент формує запис подвійного бухгалтерського обліку у YugabyteDB в межах атомарної транзакції, що включає debit-проводку торговцю на суму amount_usd_paid, credit-проводку торговцю на amount_usd_merchant після вирахування комісії, та credit-проводку системному торговцю на величину service_fee. Система оновлює balance рахунків merchant та системного рахунку. Після успішної фіксації публікується подія LedgerTxEvent до ledger.tx.stored.v1. Рахунок-фактура логічно вважається зафіксованим у журнал.

5. Незмінна фіксація у криптографічному сховищі ImmuDB

Immu Consumer читає подію LedgerTxEvent з Kafka та виконує операцію VerifiableSet у відповідному ImmuDB шард. Ключ запису містить merchant_id та journal_id, а значення представляє структурований опис транзакції у форматі Protocol Buffers. Отриманий tx_id у незмінному сховищі та Merkle root записуються у таблицю journal_immutability в YugabyteDB. Публікується подія LedgerConfirmedTxEvent до topic ledger.tx.confirmed.v1. Processor Consumer типу ledger-confirmed оновлює стан invoice у Processor Service через виклик методу confirm_invoice, переводячи його у стан confirmed.

6. Збір аналітичних метрик

Stats Consumer агрегує технічні Stat-події з усіх етапів конвеєра обробки та записує їх до аналітичної бази даних ClickHouse, що дозволяє будувати візуалізації throughput та latency pipeline для аналізу продуктивності системи.

3.1.3 Протоколи взаємодії компонентів

У системі застосовано комбіновану модель міжсервісної комунікації, що поєднує синхронні та асинхронні підходи залежно від характеру операцій.

Протокол gRPC використовується для операцій, де необхідна негайна відповідь і гарантована строга узгодженість даних. До таких операцій належать

верифікація ключів програмного інтерфейсу та управління доступом через сервіс обліку, створення рахунків-фактур, вибір криптовалюти для платежу, фіналізація та підтвердження рахунків через сервіс обробки транзакцій, а також внутрішні виклики між компонентами ядра системи в межах критичного бізнес-процесу обробки платежу.

Використання gRPC забезпечує компактний бінарний формат серіалізації повідомлень через Protocol Buffers, мінімальні накладні витрати на передачу даних, типобезпечність контрактів взаємодії на етапі компіляції та зручну автоматичну генерацію клієнтських бібліотек для мови програмування Rust.

Платформа Kafka використовується для всього конвеєра асинхронної обробки платежів і збору аналітичних даних через систему тематичних каналів подій. Тема подій блокчейн-транзакцій містить первинні події підтвердження транзакцій з різних блокчейн-мереж. Тема оброблених транзакцій містить фіналізовані транзакції, готові до фіксації у журналі обліку. Тема збережених записів журналу містить підтвердження успішного запису у розподілену базу даних YugabyteDB. Тема підтверджених транзакцій містить результати незмінної фіксації у криптографічному сховищі ImmuDB. Окремі теми статистичних подій призначені для збору технічних метрик і передачі до аналітичної системи ClickHouse.

Платформа Kafka гарантує збереження строгого порядку подій у межах однієї партиції та забезпечує горизонтальне масштабування груп споживачів, що є критично важливим для обробки масових однотипних платіжних транзакцій з високою пропускною здатністю.

Розподілена база даних YugabyteDB виступає основним сховищем для операцій запису рахунків-фактур, записів журналу подвійного бухгалтерського обліку та балансів рахунків торговців. Усі транзакції виконуються з повними ACID-гарантіями узгодженості та атомарності.

Система Redis використовується як високошвидкісний кеш для зменшення навантаження на основну базу даних, а також як механізм реалізації легких

розподілених блокувань для конкурентного вибору адрес гаманців і кешування ключів програмного інтерфейсу.

Незмінне сховище ImmuDB викликається через gRPC-інтерфейс для створення криптографічно захищених незмінних записів через структуру верифікованої множини та отримання криптографічних доказів включення для незалежної верифікації.

Аналітична база даних ClickHouse заповнюється асинхронно через окремого споживача статистичних подій, що забезпечує ізоляцію аналітичних запитів від основного транзакційного контуру обробки платежів і запобігає впливу аналітики на продуктивність критичних операцій.

3.2 Розробка мікросервісної інфраструктури

Мікросервісна інфраструктура системи реалізована як набір незалежних сервісів, кожен з яких відповідає за чітко визначену підзадачу платіжного конвеєра. Такий підхід дозволяє:

- ізолювати бізнес-логіку від асинхронної обробки подій;
- масштабувати окремі компоненти відповідно до профілю навантаження;
- спростити тестування та дослідницькі експерименти (зміна кількості партицій Kafka, реплік консюмерів тощо);
- знизити ризик каскадних відмов.

Сервіси реалізовані мовою Rust як окремі бінарники в межах єдиного workspace. Контракти взаємодії між сервісами описані через protobuf та використовуються як єдине джерело істини для генерації gRPC-клієнтів і серверів. Усі сервіси розгортаються в Kubernetes, тому є Slateless-компонентами: стан зберігається лише у спеціалізованих сховищах (YugabyteDB, Redis, Kafka, ImmuDB, ClickHouse).

3.2.1 Account Service: автентифікація та управління доступом

Account Service виконує роль ядра ідентифікації та контролю доступу мерчантів до системи. Його функціональність включає:

1. Реєстрація та управління обліковими записами мерчантів.

Сервіс створює нові акаунти, зберігає email та парольний хеш у YugabyteDB. Для захисту паролів застосовано Argon2id із параметрами складності, що відповідає практикам безпечного зберігання секретів у фінансових системах.

2. Генерація API-ключів.

Для доступу до платіжного терміналу мерчант отримує API-ключ, який використовується як токен авторизації для публічних запитів. Ключ генерується криптографічно випадковим способом і хешується перед збереженням у БД.

3. Валідація API-ключів.

Усі звернення до зовнішніх сервісів (зокрема Processor Service або Client API) перед виконанням бізнес-операцій перевіряють ключ через gRPC-виклик VerifyApiKey.

Щоб уникнути зайвих запитів у YugabyteDB, Account Service тримає кеш ключів у Redis із TTL. Таким чином критичний шлях авторизації залишається швидким навіть при високій RPS.

4. Внутрішній gRPC-інтерфейс.

Інтерфейс Account Service надає методи типу CreateAccount, DeleteAccount, GenerateApiKey, VerifyApiKey, VerifyPassword.

Сервіс має базу даних в YugabyteDB, що має таку структуру:

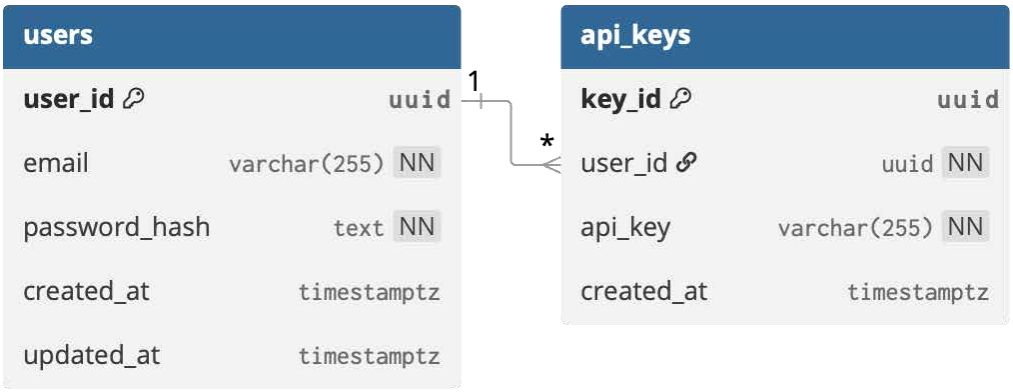


Рисунок 3.7 - Структара бази acccount

Лістинг 3.1 – Protobuf визначення для Account Service

```
package account.v1;

message CreateAccountRequest {
  string email = 1;
  string password = 2;
}
message CreateAccountResponse { string user_id = 1; }

message GetAccountRequest { string user_id = 1; }

message GetAccountResponse {
  string user_id = 1;
  string email = 2;
  google.protobuf.Timestamp created_at = 3;
}
message VerifyPasswordRequest {
  string email = 1;
  string password = 2;
}
message VerifyPasswordResponse { string user_id = 1; }
message DeleteAccountRequest { string user_id = 1; }
message DeleteAccountResponse { bool success = 1; }
message GenerateApiKeyRequest { string user_id = 1; }
message GenerateApiKeyResponse { string api_key = 1; }
message VerifyApiKeyRequest { string api_key = 1; }
message VerifyApiKeyResponse { string user_id = 1; }

service AccountService {
  rpc CreateAccount(CreateAccountRequest) returns (CreateAccountResponse) {}
  rpc GetAccount(GetAccountRequest) returns (GetAccountResponse) {}
  rpc DeleteAccount(DeleteAccountRequest) returns (DeleteAccountResponse) {}
```

```
  rpc VerifyPassword(VerifyPasswordRequest) returns (VerifyPasswordResponse) {}

  rpc GenerateApiKey(GenerateApiKeyRequest) returns (GenerateApiKeyResponse) {}
  rpc VerifyApiKey(VerifyApiKeyRequest) returns (VerifyApiKeyResponse) {}
}
```

3.2.2 Processor Service: обробка платіжних подій

Processor Service реалізує основну бізнес-логіку криптовалютного платіжного терміналу. Архітектурно він складається з двох взаємопов'язаних частин: синхронного gRPC-сервісу для обслуговування запитів у режимі реального часу та асинхронного споживача подій Kafka для потокової обробки, які разом формують єдиний процесинговий контур платіжної системи.

Синхронний компонент Processor Service виконує операції створення та керування рахунками-фактурами у режимі прямої синхронної взаємодії з клієнтами через програмний інтерфейс.

Основні методи API:

– CreateInvoice - створення рахунку-фактури з обов'язковими параметрами: ідентифікатор торговця, ідентифікатор замовлення, сума платежу у доларах США, опційний URL для вебхуків повідомлень та термін дії рахунку. Створений рахунок-фактура зберігається у розподіленій базі даних YugabyteDB та переходить у початковий стан очікування оплати.

– SelectInvoiceCurrency - вибір пари криптовалюта–блокчейн-мережа після створення рахунку-фактури. Метод забезпечує три критичні механізми: запобігання повторним викликам через розподілене блокування у Redis, конкурентобезпечне резервування адреси гаманця через транзакційний механізм з пропуском заблокованих записів, а також фіксацію обраної адреси гаманця у базі даних. Це дозволяє масштабовано розподіляти унікальні адреси між численними рахунками-фактурами без конфліктів при паралельній обробці запитів.

– FinalizeInvoice - внутрішня фіналізація рахунку-фактури після успішної обробки події підтвердження транзакції з блокчейн-мережі. Фіксуються критичні параметри: фактична сума отриманого платежу, чиста

сума для торговця після вирахування комісії, величина сервісної комісії, зовнішній ідентифікатор блокчейн-транзакції, а також часові мітки моменту оплати та завершення обробки. Рахунок-фактура переходить у стан фіналізації, що означає успішне підтвердження надходження платежу.

– ConfirmInvoice: підтвердження після завершення формування бухгалтерських проводок у журналі обліку. Фіксується ідентифікатор відповідного запису журналу, а рахунок-фактура набуває остаточного статусу підтвердження.

– Асинхронний компонент системи відповідає за потокову обробку подій підтвердження транзакцій з блокчейн-мереж.

– Основні етапи роботи:

– Отримання події BlockchainTxEvent з теми blockchains.tx.v1 платформи Kafka.

– Пошук активного рахунку-фактури за адресою гаманця-отримувача.

– Конвертація суми з криптовалюти у еквівалент у доларах США за поточним курсом.

– Перевірка достатності отриманого платежу відносно запитаної суми.

– Обчислення величини сервісної комісії та чистої суми для зарахування торговцю.

– Виклик операції FinalizeInvoice у синхронному Processor Service.

– Публікація події ProcessedTxEvent до теми processor.tx.v1 для подальшої обробки.

Такий поділ на синхронний сервіс та асинхронного споживача забезпечує можливість незалежного горизонтального масштабування обробки потоку подій з блокчейн-мереж окремо від навантаження на публічний програмний інтерфейс, ізоляцію ресурсоємної потокової логіки від синхронних викликів для запобігання їх взаємному впливу на продуктивність, а також загальне підвищення стійкості системи до відмов та пропускної здатності процесингового контуру платіжної обробки.

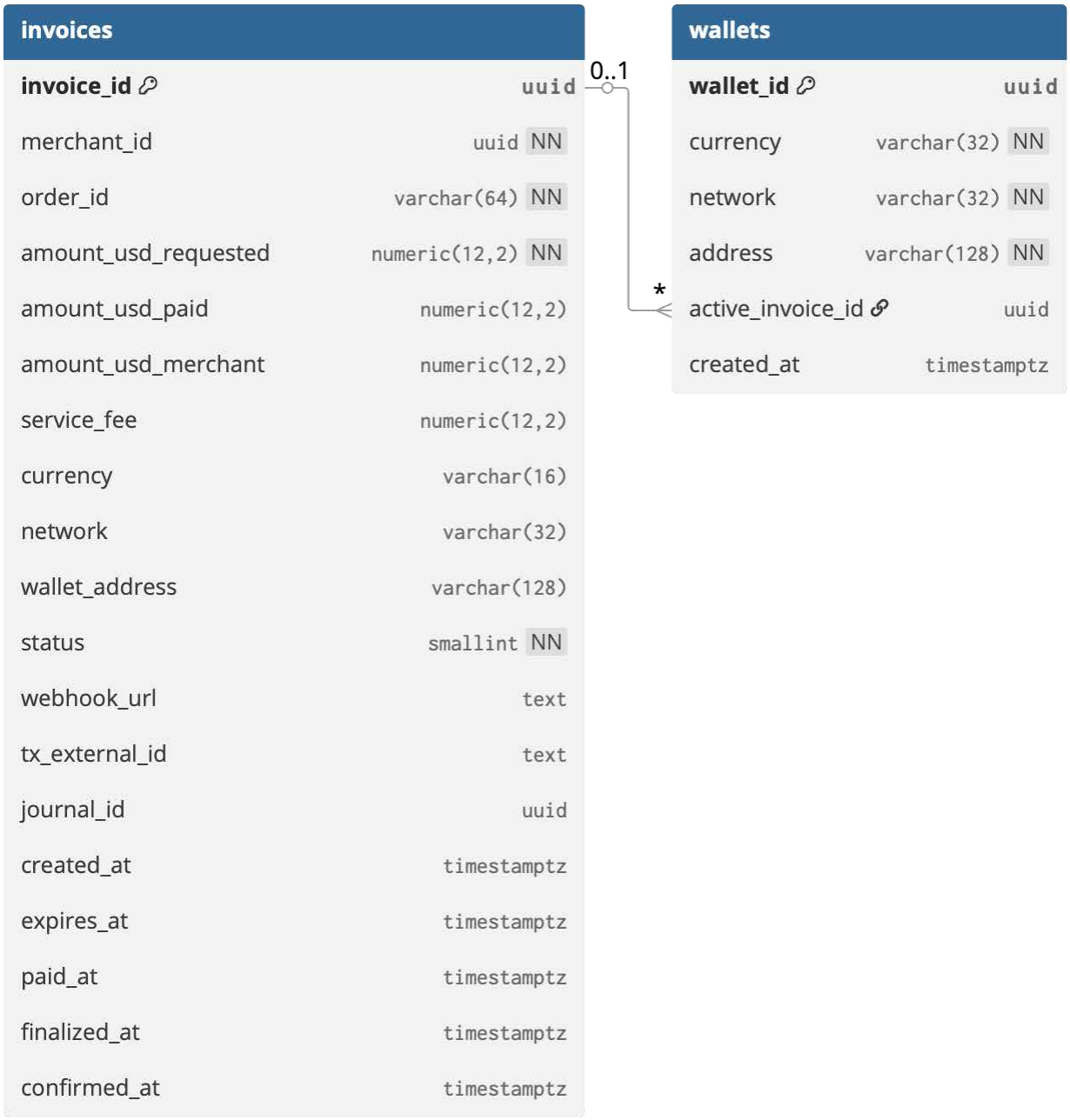


Рисунок 3.8 - Структура бази processor

Лістинг 3.2 – Protobuf визначення для Processor Service

```
package processor.v1;
enum InvoiceStatus {
  INVOICE_STATUS_PENDING = 0;
  INVOICE_STATUS_PAID = 1;
  INVOICE_STATUS_FAILED = 2;
}
message Invoice {
  string id = 1;
  string merchant_id = 2;
```

```
  string order_id = 3;
  uint64 amount_usd_requested = 4;
  optional uint64 amount_usd_paid = 5;
  optional uint64 amount_usd_merchant = 6;
  optional string currency = 7;
  optional string network = 8;
  optional string wallet_address = 9;
  string payment_url = 10;
  InvoiceStatus status = 11;
  google.protobuf.Timestamp created_at = 12;
  google.protobuf.Timestamp expires_at = 13;
  google.protobuf.Timestamp paid_at = 14;
  google.protobuf.Timestamp finalized_at = 15;
  optional string webhook_url = 16;
}
message Currency {
  string name = 1;
  string network = 2;
}
message CreateInvoiceRequest {
  string merchant_id = 1;
  string order_id = 2;
  uint64 amount_usd = 3;
  optional string webhook_url = 4;
}
message CreateInvoiceResponse { Invoice invoice = 1; }
message FinalizeInvoiceRequest {
  string invoice_id = 1;
  uint64 amount_usd_paid = 2;
  uint64 amount_usd_merchant = 3;
  uint64 service_fee = 4;
  string tx_id = 5;
  google.protobuf.Timestamp paid_at = 6;
}
message FinalizeInvoiceResponse { Invoice invoice = 1; }

message RetrieveInvoiceRequest { string invoice_id = 1; }

message RetrieveInvoiceResponse { Invoice invoice = 1; }
message SelectInvoiceCurrencyRequest {
  string invoice_id = 1;
  string currency = 2;
  string network = 3;
}
message SelectInvoiceCurrencyResponse { Invoice invoice = 1; }
message GetSupportedCurrenciesRequest {}
```

```
message GetSupportedCurrenciesResponse { repeated Currency currencies = 1; }
message ResolveActiveInvoiceRequest { string wallet_address = 1; }
message ResolveActiveInvoiceResponse { Invoice invoice = 1; }
message ConfirmInvoiceRequest {
  string invoice_id = 1;
  string journal_id = 2;
  google.protobuf.Timestamp confirmed_at = 3;
}
message ConfirmInvoiceResponse {}
service ProcessorService {
  rpc CreateInvoice(CreateInvoiceRequest) returns (CreateInvoiceResponse) {}
  rpc FinalizeInvoice(FinalizeInvoiceRequest)
      returns (FinalizeInvoiceResponse) {}
  rpc SelectInvoiceCurrency(SelectInvoiceCurrencyRequest)
      returns (SelectInvoiceCurrencyResponse) {}
  rpc RetrieveInvoice(RetrieveInvoiceRequest)
      returns (RetrieveInvoiceResponse) {}
  rpc ConfirmInvoice(ConfirmInvoiceRequest) returns (ConfirmInvoiceResponse) {}

  rpc GetSupportedCurrencies(GetSupportedCurrenciesRequest)
      returns (GetSupportedCurrenciesResponse) {}
  rpc ResolveActiveInvoice(ResolveActiveInvoiceRequest)
      returns (ResolveActiveInvoiceResponse) {}
}
```

3.2.3 Ledger: реалізація подвійного запису

Ledger Service та Consumer є фінансовим ядром системи і забезпечує коректний бухгалтерський облік платежів у форматі подвійного запису. Архітектурно він поділений на контур з двох асинхронних споживачів подій.

1) Ledger Consumer (Kafka → YugabyteDB)

Отримує події ProcessedTxEvent і виконує атомарну транзакцію в розподіленій базі даних YugabyteDB, що включає наступні кроки:

Створення запису у таблиці журналів обліку, де один журнал відповідає одному рахунку-фактурі.

Формування рядків подвійного бухгалтерського запису у таблиці проводок:

– дебетова проводка торговцю на повну суму отриманого платежу;

– кредитова проводка торговцю на суму після вирахування комісії;

– кредитова проводка системному рахунку на величину сервісної комісії.

Оновлення таблиці балансів для рахунків торговця і системного рахунку платформи. Після успішного запису публікується подія LedgerTxEvent у тему ledger.tx.stored.v1 платформи Kafka.

2) Immu Consumer (Kafka → ImmuDB → YugabyteDB)

– Отримує події LedgerTxEvent і виконує процес незмінної фіксації:

– Виконує операцію VerifiableSet у випадково обраний шард розподіленої системи ImmuDB.

– Формує композитний ключ з ідентифікатора торговця та ідентифікатора журналу, а значення кодує у структурований формат Protocol Buffers, що описує рух коштів.

– Отримує ідентифікатор транзакції у незмінному сховищі та корінь дерева Меркла для криптографічної верифікації.

– Записує метадані незмінності у таблицю криптографічних доказів журналу обліку.

– Публікує подію LedgerConfirmedTxEvent у тему ledger.tx.confirmed.v1 для оповіщення інших компонентів.

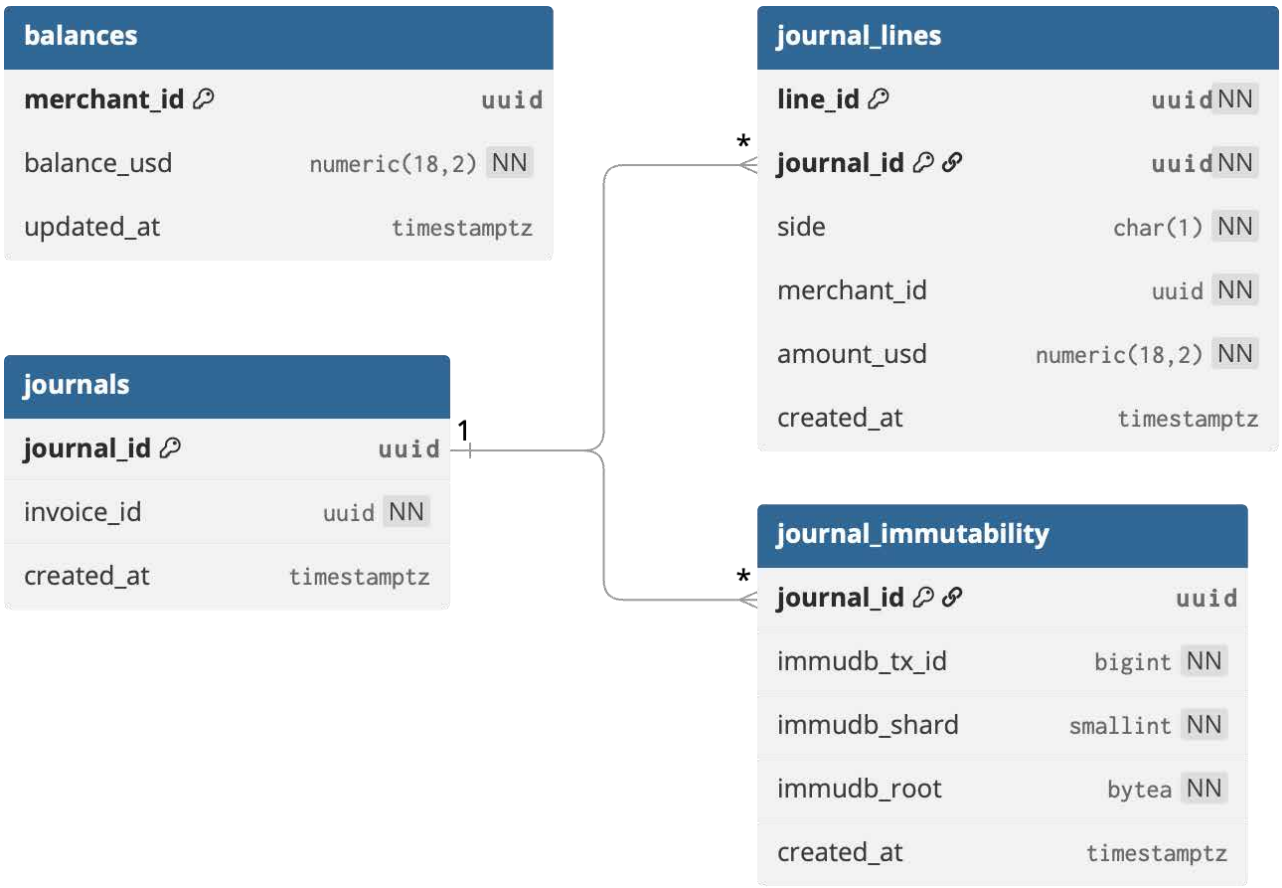


Рисунок 3.9 - Структура бази ledger

Таким чином, Ledger Service реалізує два ключові аспекти надійного фінансового обліку: фінансову узгодженість через механізм подвійного запису та ACID-транзакції у YugabyteDB, а також криптографічну незмінність історії операцій через незмінне сховище ImmuDB.

Лістинг 3.3 – Protobuf визначення для Ledger Service

```
package ledger.v1;

message LedgerEntry {
  string journal_id = 1;
  string invoice_id = 2;
  string merchant_id = 3;
  uint64 immudb_tx_id = 4;
  uint32 immudb_shard_id = 5;
  string immudb_key = 6;
  uint64 amount_usd_debit = 7;
  uint64 amount_usd_credit = 8;
  google.protobuf.Timestamp journal_created_at = 9;
```

```
}

message ListLedgerEntriesRequest {
  string merchant_id = 1;
  int32 limit = 2;
  optional string cursor = 3;
}

message ListLedgerEntriesResponse {
  repeated LedgerEntry entries = 1;
  optional string cursor = 2;
}

service LedgerService {
  rpc ListLedgerEntries(ListLedgerEntriesRequest)
      returns (ListLedgerEntriesResponse);
}
```

3.2.4 Stats Service: збір та агрегація метрик

Stats Service призначений для дослідницької частини роботи і забезпечує накопичення вимірювань продуктивності та затримок у конвеєрі обробки. Він функціонує як асинхронний споживач подій Kafka і виконує три основні функції:

Збір технічних подій з різних етапів системи. Для кожного ключового переходу у конвеєрі обробки - від прийняття події з блокчейну через фіналізацію у процесорі та збереження у журналі обліку до підтвердження у незмінному сховищі - формується подія статистики з часовими мітками початку обробки та фактичним часом завершення етапу.

ClickHouse використовується як джерело метрик для побудови графіків у вигляді SQL-агрегацій, які включаються у дослідницьку частину роботи.

Stats Service не впливає на критичний шлях запису фінансових транзакцій і тому може масштабуватись незалежно або бути тимчасово вимкненим без порушення основної бізнес-логіки платіжної системи.

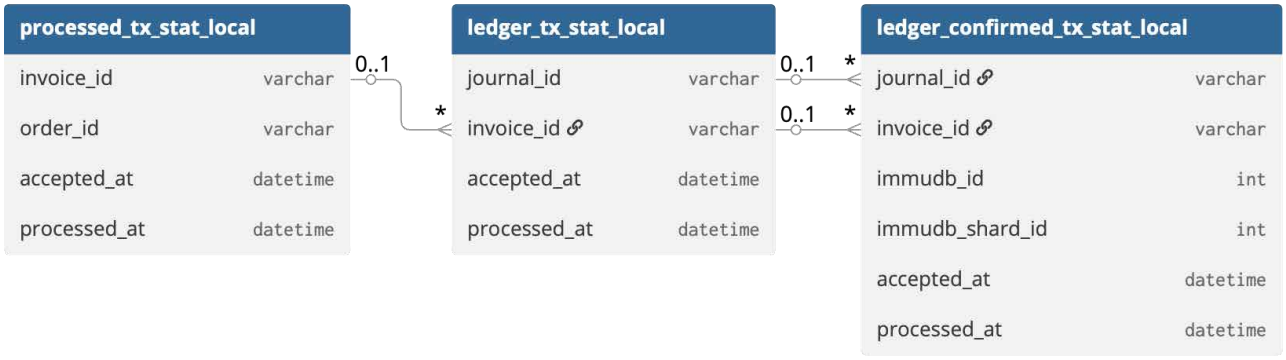


Рисунок 3.10 - Структура бази Clickhouse

3.3 Імплементація криптографічного аудиту через ImmuDB

Платіжний термінал для масових криптоплатежів, на відміну від децентралізованих блокчейнів, має централізований серверний контур обробки інвойсів і журналювання результатів. Це створює фундаментальну проблему довіри: навіть якщо бухгалтерський інваріант подвійного запису виконується в основній БД, мерчант не має незалежного математичного механізму довести, що історія транзакцій не була змінена або “підчищена” заднім числом.

Для розв’язання цієї проблеми застосовано ImmuDB як незмінне криптографічне сховище. Воно забезпечує append-only модель записи, формує Merkle-дерево поверх логів транзакцій і дозволяє отримувати криптографічні докази включення/консистентності для кожного запису. Включення ImmuDB у pipeline системи створює незалежний аудитний контур, який не потребує довіри до оператора терміналу.

3.3.1 Структура незмінного журналу в ImmuDB

У прототипі ImmuDB використовується як окремий шар криптографічно захищеної незмінної фіксації для фінансових записів. Логіка організації даних у ImmuDB побудована з трьома ключовими принципами: уникнення дублювання структури подвійного бухгалтерського запису через збереження лише криптографічно верифікованого підтвердження про проведену транзакцію, забезпечення окремої історії для кожного торговця з можливістю незалежного відстеження послідовності його операцій, а також горизонтальна масштабованість через механізм шардування даних.

Для кожного запису журналу обліку, де один журнал відповідає одному рахунку-фактурі, формується композитний ключ з ідентифікатора торговця та ідентифікатора журналу. Це забезпечує природну логічну ізоляцію історії різних торговців і дозволяє виконувати сканування за префіксом ідентифікатора торговця для вибірки повної історії платежів конкретного бізнесу.

Значення кодується у структурований формат Protocol Buffers, що містить мінімально достатній набір полів для аудиту операцій: ідентифікатори рахунку-фактури, запису журналу та транзакції; суми у доларах США, включаючи запитану суму, фактично сплачену, суму для торговця та величину комісії; довідкові поля з інформацією про криптовалюту, блокчейн-мережу, адресу гаманця та ідентифікатор зовнішньої блокчейн-транзакції; а також часові мітки збереження та підтвердження операції.

Таким чином, ImmuDB зберігає саме незмінний криптографічний доказ факту проведення облікової операції, а не її повну бухгалтерську модель з усіма проводками.

Для обробки навантаження масових платежів застосовано кілька незалежних шардів ImmuDB. На практиці конкретний шард вибирається випадково. У відповідь на операцію верифікованого додавання до множини ImmuDB повертає ідентифікатор транзакції у незмінному сховищі, подвійний криптографічний доказ, що включає докази узгодженості та включення запису, а також кореневий хеш дерева Меркла після фіксації нового запису.

Ці метадані зберігаються у розподіленій базі даних YugabyteDB в таблиці криптографічних доказів журналу разом з ідентифікатором шарду ImmuDB. Такий архітектурний підхід дозволяє зберігати зв'язок між записом бухгалтерського журналу та його незмінним криптографічним доказом, підтримувати подальший аудит без необхідності повторного перегляду повної історії операцій, а також уникати змішування швидкості запису YugabyteDB і ImmuDB в одному критичному шляху обробки транзакцій.

Лістинг 3.4 – Protobuf визначення значення, що зберігається в ImmuDB

```
message ImmuLedgerValue {
  string invoice_id = 1;
  string merchant_id = 2;
  uint64 amount_usd_requested = 3;
  uint64 amount_usd_paid = 4;
  uint64 amount_usd_merchant = 5;
  uint64 amount_usd_fee = 6;
  string currency = 7;
  string network = 8;
  string wallet_address = 9;
  string journal_id = 10;
  string blockchain_tx_id = 11;
  google.protobuf.Timestamp stored_at = 12;
  google.protobuf.Timestamp confirmed_at = 13;
}
```

3.3.2 API для генерації та валідації доказів

Однією з вимог системи є можливість проведення зовнішнього аудиту з боку торговця або незалежної третьої сторони. Тому передбачено програмний інтерфейс, який повертає не лише дані про транзакцію, а й супутні криптографічні докази для незалежної верифікації.

Процес включає наступні кроки: торговець ініціює запит доказу за ідентифікатором запису журналу або рахунку-фактури; сервіс формує композитний ключ з ідентифікатора торговця та ідентифікатора журналу; виконується операція верифікованого отримання даних з ImmuDB; клієнту повертається повний пакет інформації, що включає структуроване значення з деталями транзакції, доказ включення запису до дерева Меркла, доказ узгодженості історії відносно попередньої відомої транзакції, а також поточний незмінний стан системи.

Таким чином, клієнт отримує всю необхідну інформацію для самостійної перевірки того, що запис реально присутній у криптографічному дереві Меркла і що історія операцій між попередньою верифікованою транзакцією та поточною є узгодженою без розривів або модифікацій.

Валідація доказів може виконуватись двома способами: через окремий скрипт торговця з використанням клієнтських бібліотек для роботи з ImmuDB, або зовнішнім незалежним аудитором.

Логіка криптографічної перевірки зводиться до послідовних кроків: обчислення криптографічного хешу отриманого значення транзакції за стандартним алгоритмом; перевірка доказу включення для отримання кореневого хешу дерева Меркла; перевірка доказу узгодженості історії між відомим попереднім ідентифікатором транзакції та новим ідентифікатором; порівняння отриманого кореневого хешу з офіційним кореневим хешем, який повертає система ImmuDB.

Це забезпечує математичну гарантію двох фундаментальних властивостей: запис не міг бути модифікований або підмінений після первинної фіксації, а вся історія операцій не могла бути ретроспективно переписана між двома станами незмінної бази даних без виявлення такої спроби при верифікації доказів.

ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ВАЛІДАЦІЯ

4.1 Методика проведення експериментів

Експериментальна частина роботи спрямована на перевірку ключових гіпотез щодо масштабованості, консистентності та надійності запропонованого криптовалютного терміналу. Основний фокус дослідження - поведінка подієвої мікросервісної архітектури під навантаженням, дотримання фінансових інваріантів (debit = credit), а також вплив параметрів інфраструктури на пропускну здатність і затримки обробки інвойсу.

Методика базується на принципах керованого навантажувального тестування:

- визначення контрольованих змінних (кількість партицій Kafka, кількість реплік consumer-воркерів, обсяг батчів, алокатор пам’яті Rust-сервісів);
- фіксація незмінних умов (конфігурація кластеру, версії сервісів, типи подій, схема БД);
- багаторазовий запуск сценаріїв з однаковими параметрами для усунення випадкових коливань;
- збір телеметрії з усіх етапів pipeline та її агрегація в ClickHouse.

Для кожного експерименту вимірюються:

- пропускна здатність (RPS/TPS) - скільки інвойсів/транзакцій проходить повний цикл за секунду;
- затримка - час від створення інвойсу до підтвердження незмінного запису в ImmuDB;
- стадійні затримки - окремо для етапів simulator → processor, processor →l edger, ledger → immudb, ledger → processor;
- дотримання інваріанту подвійного запису в YugabyteDB.

Такий підхід дає змогу не лише зафіксувати продуктивність, а й пояснити її причини через архітектурні параметри.

4.1.1 Тестове середовище

Прототип системи розгорнуто у локальному Kubernetes-кластері, змодельованому за допомогою k3d поверх віртуалізаційного середовища OrbStack, що дозволяє наблизити умови до production-подібних при збереженні повного контролю над експериментом та мінімальних накладних витрат на віртуалізацію. OrbStack забезпечує легковагу та ефективну віртуалізацію контейнерів на macOS з інтеграцією Docker-середовища. Кластер складається з 13 логічних вузлів, які емулюють розподілене середовище з окремими stateful-компонентами та незалежним масштабуванням сервісів.

У середовищі розгорнуті такі підсистеми:

– Kafka cluster (3 брокери) - подієвий транспорт і гарантії порядку подій у партиції;

– YugabyteDB cluster (3 ноди) - транзакційне сховище з ACID-властивостями в розподіленому середовищі;

– Redis cluster - кешування hot-даних (API keys, lock-ключі для rate-limit);

– ImmuDB shards (3 мастер-бази з репліками) – незмінна база даних;

– ClickHouse cluster (3 ноди) - аналітика і метрики експериментів;

– Prometheus + Grafana - системні метрики CPU/RAM/IO, мережеві затримки, лаги консюмерів.

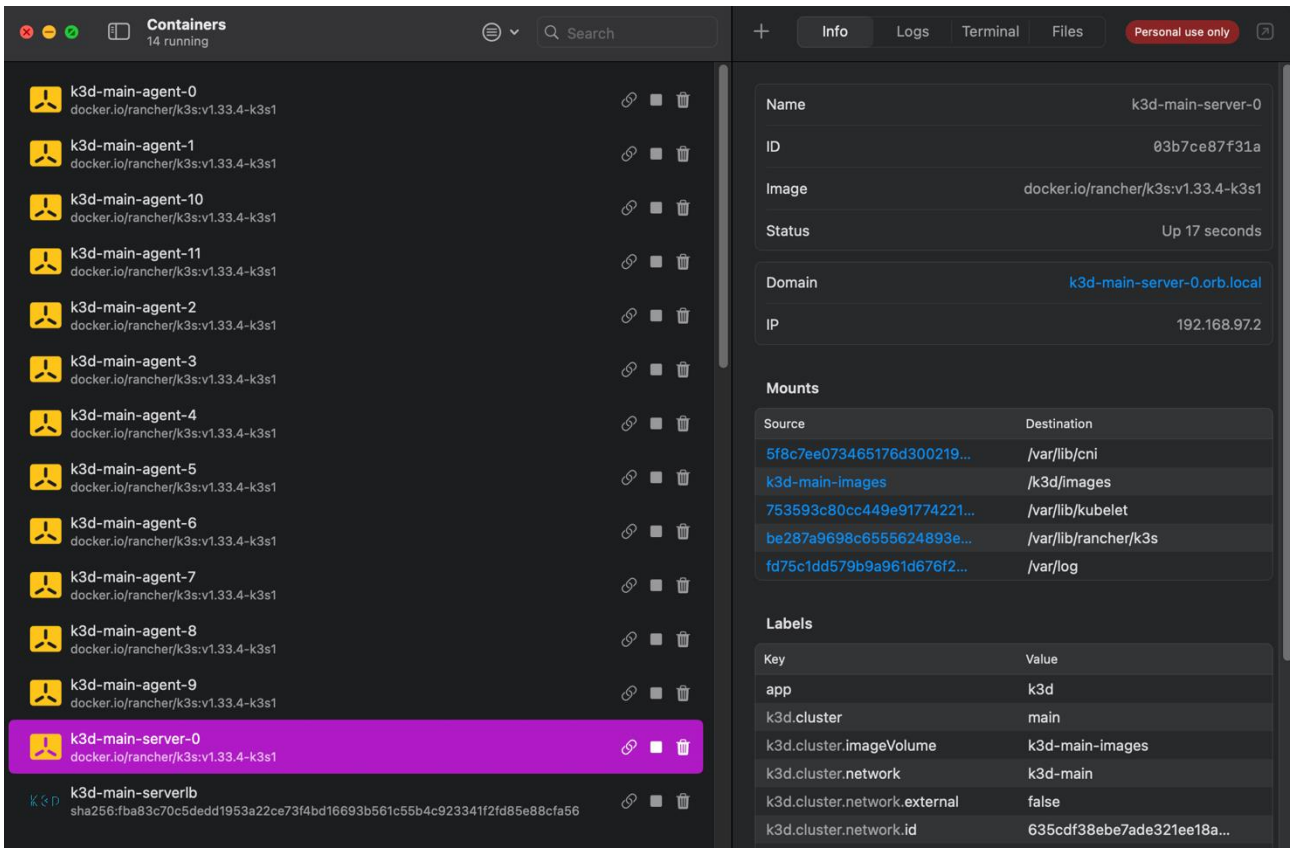


Рисунок 4.1 - Стриншот програми OrbStack із запушений тестовим стендом кластеру з 13 нод

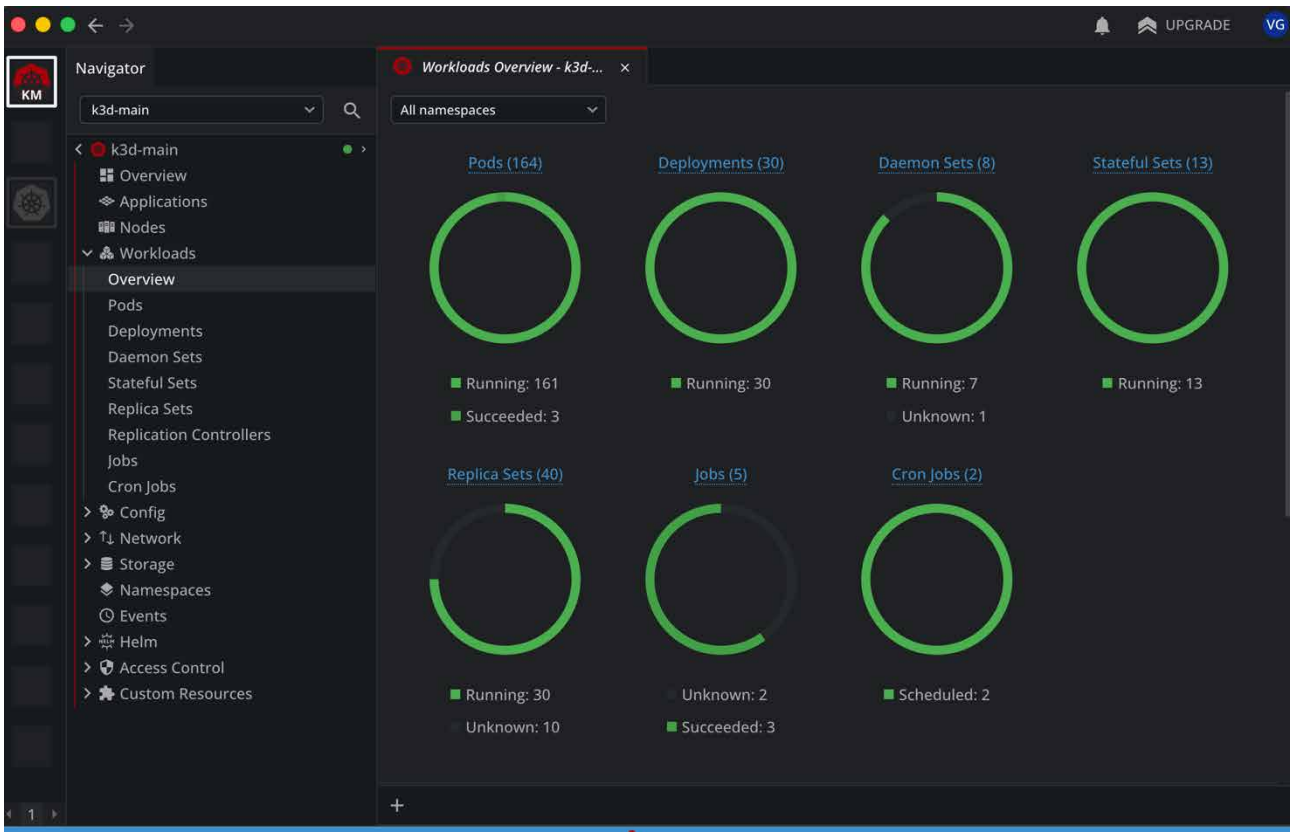
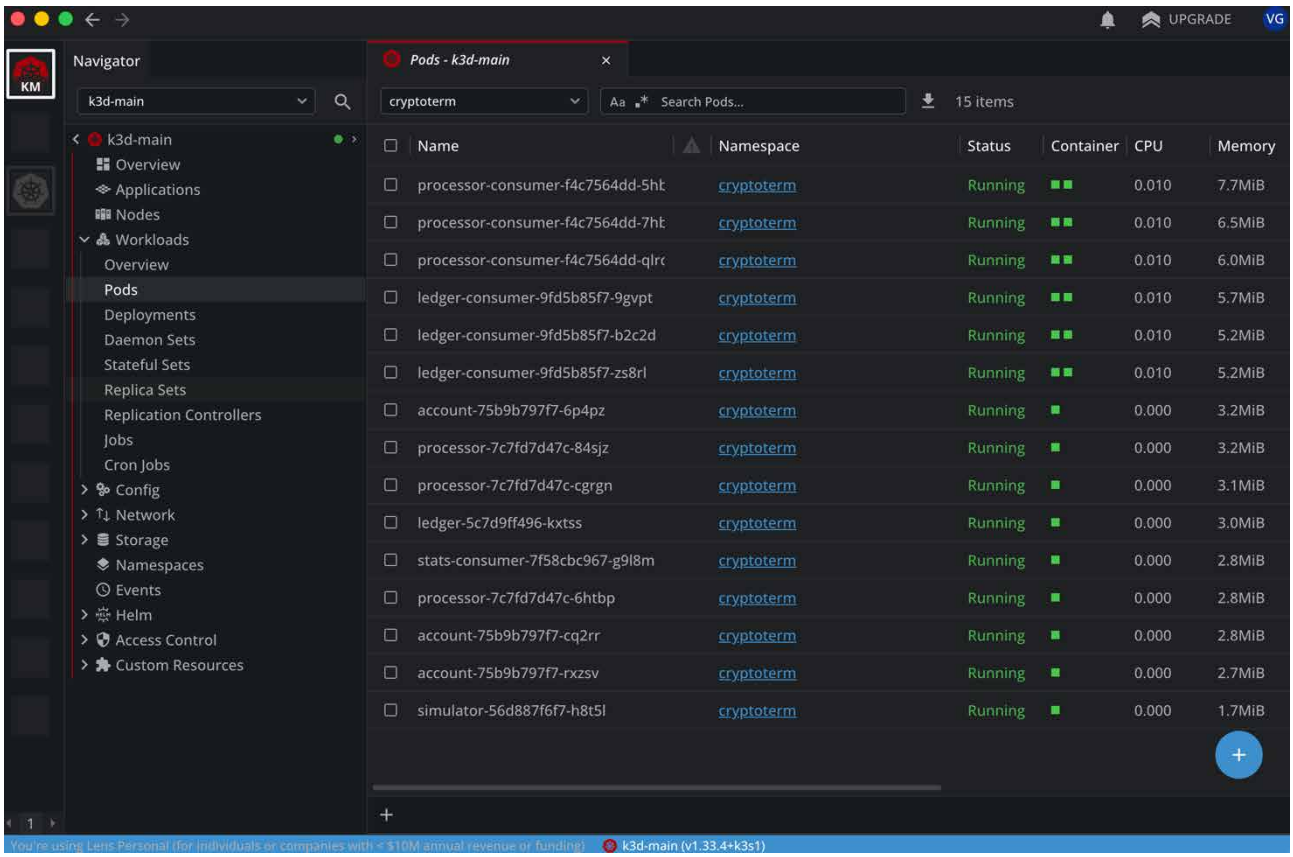


Рисунок 4.2 - Скриншоп програми Lens із оглядом стану кластеру



Локальний кластер обрано як достатнє середовище для дипломного дослідження, оскільки він дозволяє експериментально перевірити горизонтальне масштабування через зміну кількості партицій і реплік, стійкість конвеєра обробки до зростання навантаження та коректність фінансової моделі подвійного бухгалтерського запису без залежності від зовнішніх блокчейн-мереж і їх непередбачуваної поведінки.

4.1.2 Генерація навантаження через Simulator

Реальні блокчейн-події характеризуються нестабільним часом підтвердження транзакцій та залежністю від зовнішньої інфраструктури, що унеможливлює проведення повторюваних експериментів з контрольованими параметрами. Тому в роботі використовується Simulator Service, який емулює поведінку торговців та блокчейн-мереж із повним контролем інтенсивності і структури навантаження.

Функції симулятора:

Масове створення рахунків-фактур через gRPC API Processor Service з паралельною обробкою запитів. Вибір криптовалюти і блокчейн-мережі через метод SelectInvoiceCurrency з подальшим отриманням адреси гаманця. Генерація подій BlockchainTxEvent з валідними параметрами платежу, що відповідають структурі реальних транзакцій. Публікація згенерованих подій у відповідні Kafka тему блокчейн-мереж, що запускає повний конвеєр обробки через всі компоненти системи.

Навантаження генерується пакетами із заданими параметрами: кількість торговців для емуляції різних користувачів, кількість рахунків фактур на кожного торговця для моделювання обсягу транзакцій, а також рівень конкурентності для паралельної відправки подій у Kafka.

Кожен експериментальний сценарій запускається у кількох ітераціях для забезпечення статистичної достовірності. На початку кожної серії проводиться коротка фаза прогріву для стабілізації кешів та оптимізації компонентів, після чого фіксуються результати основної серії вимірювань. Події, згеновані Simulator Service, є структурно ідентичними до реальних blockchain-подій, тому показники throughput і latency достовірно відображають поведінку системи в умовах масових платежів, але без впливу зовнішнього мережевого шуму та непередбачуваності.

Таким чином, симулятор забезпечує репрезентативне, детерміноване та відтворюване навантаження, необхідне для коректної експериментальної перевірки сформульованих гіпотез дослідження.

4.2 Дослідження пропускної здатності системи

Мета цього підрозділу полягає в експериментальному визначенні пропускної здатності подієво-орієнтованого конвеєра обробкикриптовалютного терміналу та встановленні інфраструктурних параметрів, які найбільше впливають на показник пропускної здатності системи. Пропускна здатність визначається як кількість рахунків-фактур та транзакцій, що проходять повний цикл обробки за одиницю часу: від надходження події BlockchainTxEvent до

формування запису подвійного бухгалтерського обліку у розподіленій базі даних YugabyteDB і фіксації криптографічного доказу у незмінному сховищі ImmuDB.

У дослідженнях використовувались такі метрики:

– Кількість завершених рахунків-фактур у секунду
– Середня затримка опрацювання транзакцій

Спочатку проведено експериментальне тестування з базовою конфігурацією: 3 партиції для відповідних тем Kafka та по 3 екземпляри споживачів для компонентів Ledger Consumer та Ledger Immu Consumer.

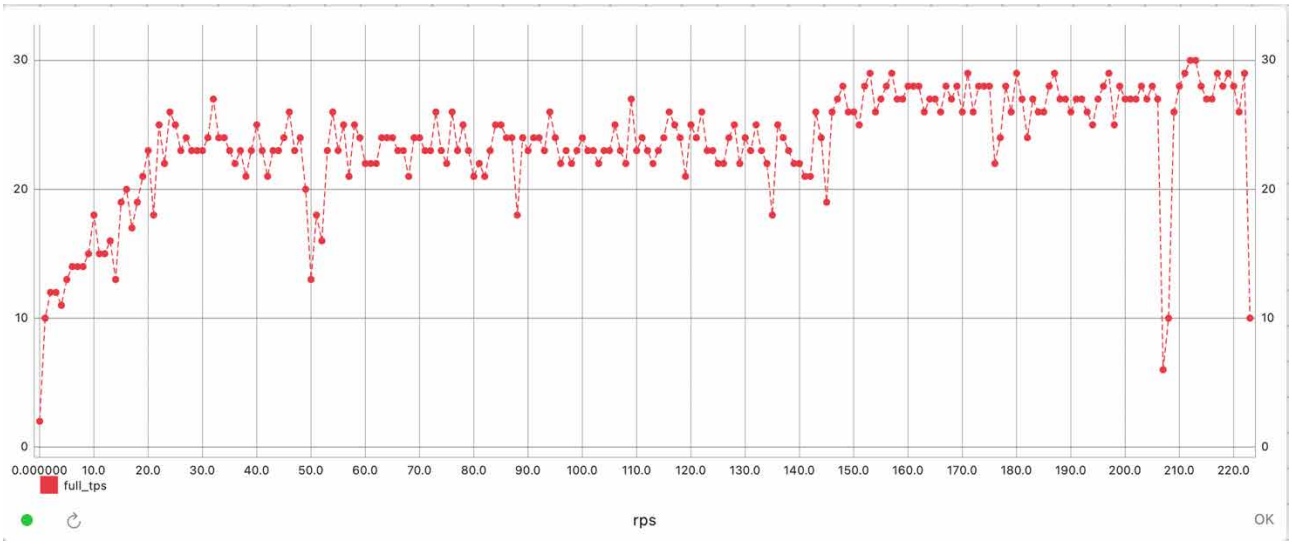


Рисунок 4.3 - Кількість транзакцій в секунду (3 партиції)

На початкових інтервалах часу система демонструє поступове зростання показника транзакцій за секунду (TPS), виходячи на рівень 20–25 транзакцій за секунду. У середній частині експерименту, приблизно на інтервалі 100–150 секунд, продуктивність підвищується до 30 TPS, що свідчить про ефективне використання ресурсів кластера. Після 150-ї секунди показник TPS продовжує зростати і досягає максимальних значень 38–40 транзакцій за секунду, проте вже спостерігаються локальні просідання продуктивності - перші ознаки наближення до межі можливостей інфраструктури. Після 210-ї секунди зафіксовано

поодинокі різкі падіння TPS до приблизно 10 транзакцій за секунду, що чітко вказує на перехід системи у режим перевантаження з виснаженням ресурсів.

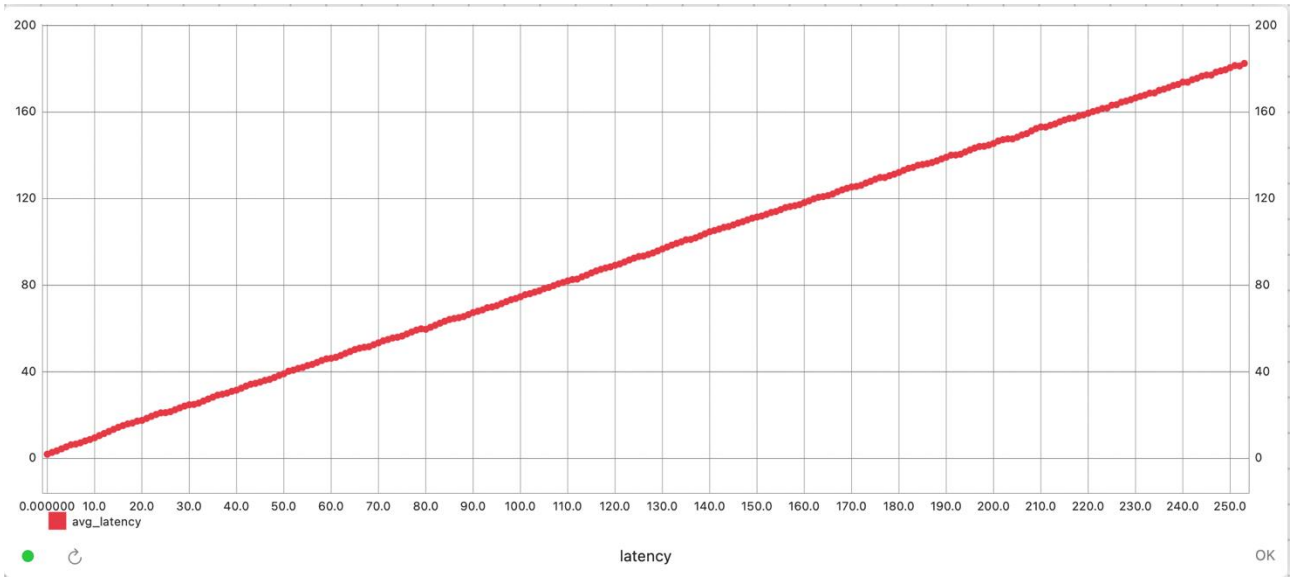


Рисунок 4.4 - Середня затримка транзакцій (3 партиції)

На початковому етапі експерименту за умов низького навантаження затримка обробки знаходиться в діапазоні 5–10 мілісекунд, що є нормальним показником для системи з достатніми вільними ресурсами. У міру зростання інтенсивності навантаження затримка збільшується майже лінійно, що відображає пропорційне зростання часу обробки запитів. До кінця експерименту, за умов максимального навантаження системи, середня затримка досягає критичних значень 180–190 мілісекунд, що чітко вказує на вичерпання обчислювальних ресурсів, накопичення черг повідомлень у Kafka та деградацію продуктивності конвеєра обробки.

Дані проведено експериментальне тестування з масштабованою вдвічі конфігурацією: 6 партиції для відповідних тем Kafka та по 6 екземпляри споживачів для компонентів Ledger Consumer та Ledger Immu Consumer.

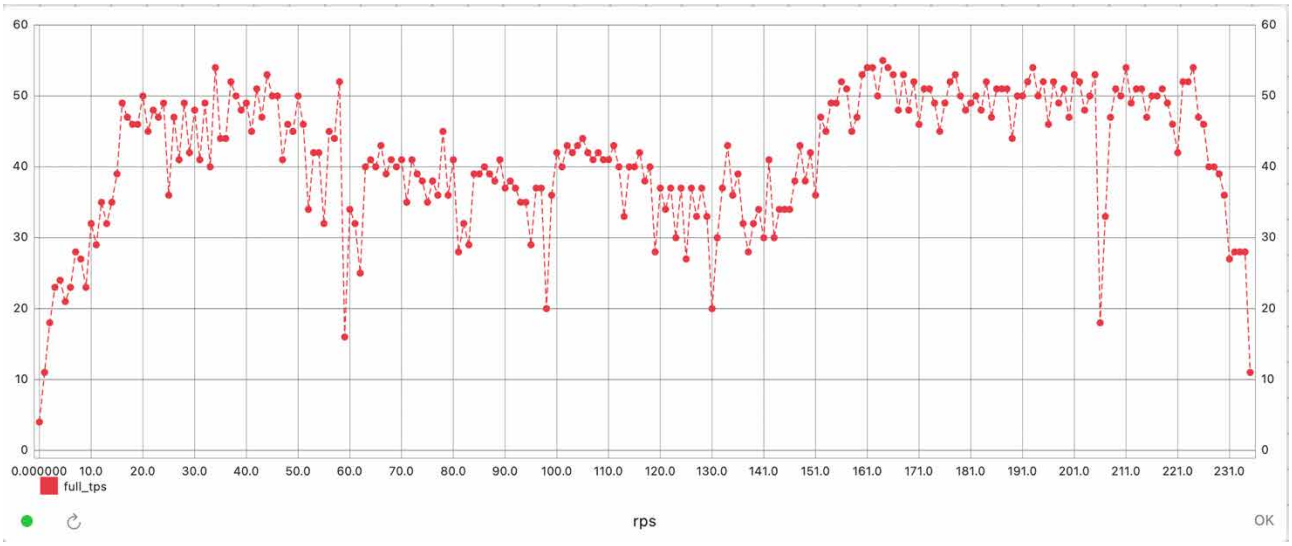


Рисунок 4.5 - Кількість транзакцій в секунду (6 партицій)

На початку експерименту показник транзакцій за секунду зростає дуже швидко, що демонструє ефективне використання системою доданого паралелізму через збільшення кількості споживачів. У початковому діапазоні графіка TPS виходить на рівень 40–55 транзакцій за секунду із природними коливаннями, характерними для подієво-орієнтованих систем з асинхронною обробкою.

Поблизу часової позначки 120–140 секунд спостерігаються хвилеподібні просадки продуктивності - це перші індикатори того, що система наближається до порогового значення пропускної здатності інфраструктури. У межах інтервалу 160–170 секунд TPS знову стабілізується на підвищеному рівні приблизно 50–55 транзакцій за секунду, що є характерною ознакою адаптації конвеєра обробки та тимчасового вирівнювання черг повідомлень.

На пікових значеннях часу в діапазоні 220–230 секунд показник TPS знижується до 20–30 транзакцій за секунду, що чітко вказує на перевантаження окремих вузлів кластера або брокерів Kafka через вичерпання обчислювальних ресурсів чи пропускної здатності мережі.

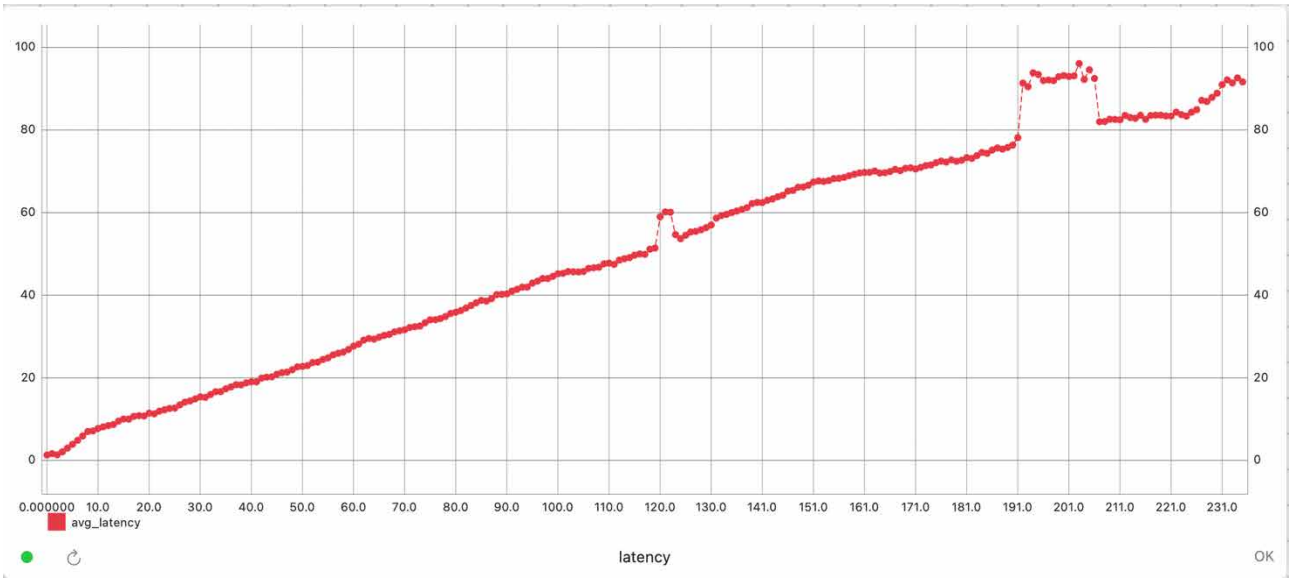


Рисунок 4.6 - Середня затримка транзакцій (6 партицій)

До часової позначки приблизно 30 секунд затримка обробки зростає повільно та рівномірно, що відповідає режиму стабільної роботи системи з достатніми ресурсами. У проміжку 30–100 секунд показник latency залишається на помірному рівні без різких стрибків, що свідчить про збалансоване навантаження на компоненти системи.

Близько 120-ї секунди спостерігається перший значимий стрибок затримки, ймовірно внаслідок пікового завантаження процесорних ресурсів або перерозподілу партицій між споживачами при ребалансуванні групи. Після цього моменту затримка продовжує зростати плавно, виходячи на рівень 150–170 мілісекунд.

Близько 190-ї секунди зафіксовано другий різкий підйом затримки, після чого крива набуває більш крутого характеру зростання, що вказує на перехід системи у режим зворотного тиску, коли споживачі не встигають обробляти вхідний потік повідомлень і виникає накопичення черг.

І на останок проведено експериментальне тестування з трикратним масштабуванням: 9 партиції для відповідних тем Kafka та по 9 екземпляри споживачів для компонентів Ledger Consumer та Ledger Immu Consumer.

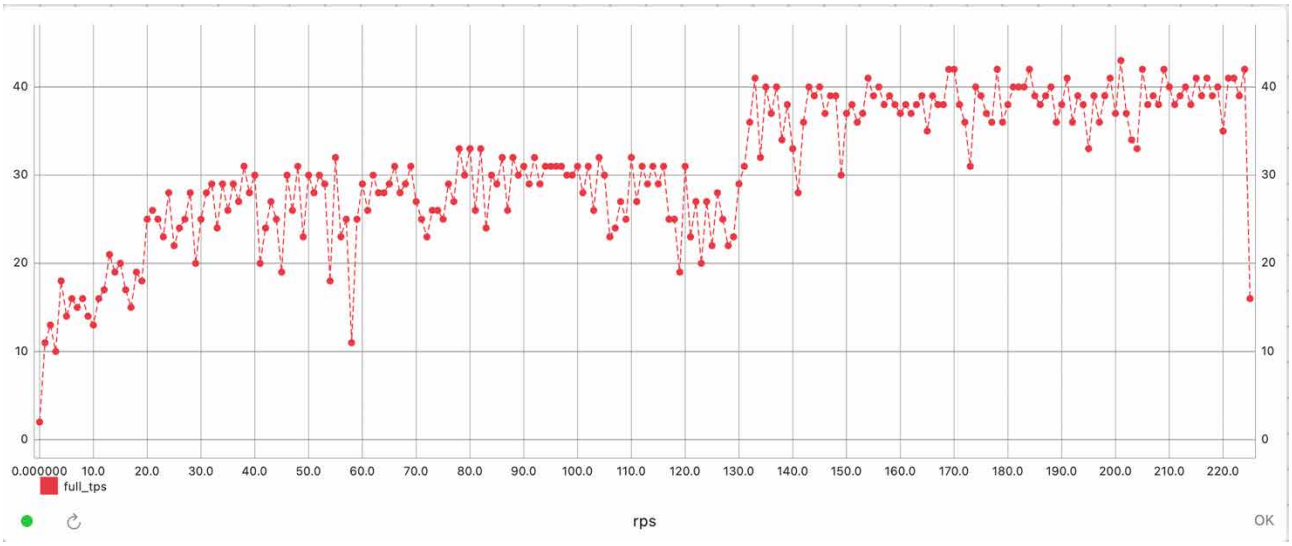


Рисунок 4.7 - Кількість транзакцій в секунду (9 партицій)

Швидке початкове зростання показника транзакцій за секунду у діапазоні до приблизно 20 секунд демонструє очікуване горизонтальне масштабування системи за умов низького навантаження, з підйомом від 5–10 TPS до рівня 25–30 транзакцій за секунду. У проміжку 20–70 секунд продуктивність стабілізується в межах 30–35 TPS із періодичними короткими коливаннями до 15–20 TPS, що є характерним для розподілених систем з великою кількістю паралельних споживачів та конкуренцією за ресурси.

Після часової позначки 80–120 секунд спостерігаються хвильові просадки показника TPS, що є чіткими ознаками наближення окремих стадій конвеєра обробки - платформи Kafka, бази даних або мережевого з'єднання - до межі їхньої пропускної здатності. У проміжку 140–200 секунд система демонструє короткі цикли відновлення продуктивності, піднімаючись до 35–40 транзакцій за секунду. Це свідчить про ефект адаптації конвеєра: черги повідомлень, що накопичилися в попередні періоди, частково розвантажуються під час коротких інтервалів зниження конкуренції за ресурси між споживачами.

На пікових значеннях часу в діапазоні 210–230 секунд фіксуються різкі провали TPS до 10–20 транзакцій за секунду, що чітко сигналізує про входження системи в режим періодичного перевантаження з виснаженням критичних ресурсів інфраструктури.

Висновок щодо пропускної здатності: Абсолютна продуктивність зросла порівняно з конфігурацією на 3 партиції, проте система знову демонструє межу насичення за умов високого навантаження, що є очікуваною поведінкою для локального кластера з обмеженими обчислювальними ресурсами.

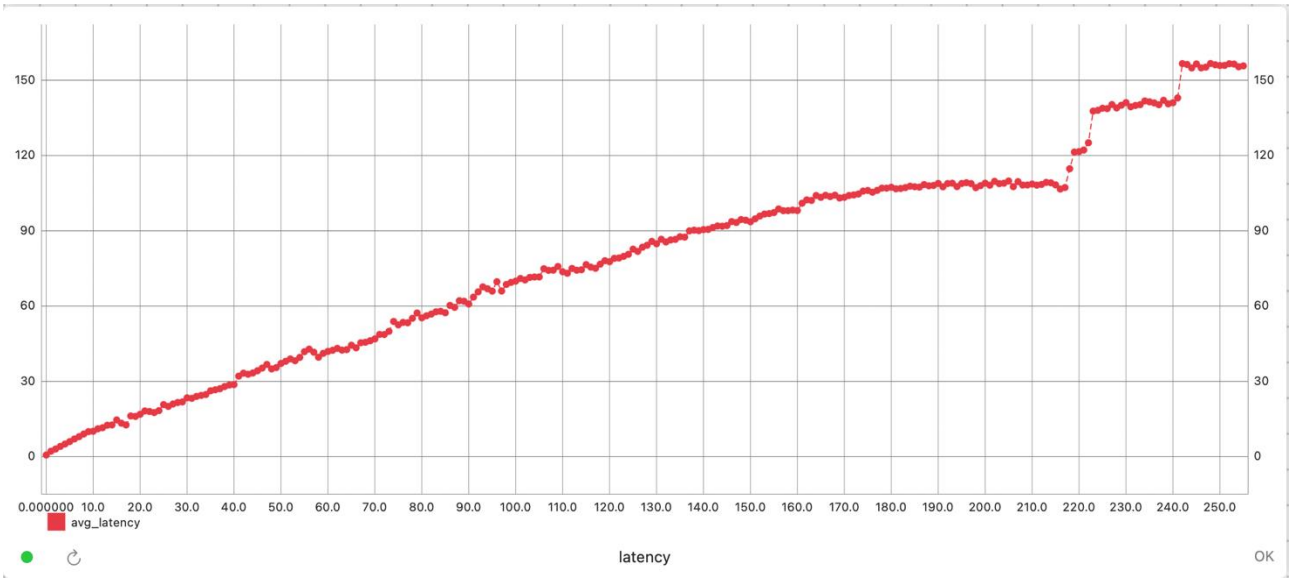


Рисунок 4.8 - Середня затримка транзакцій (9 партицій)

На початку експерименту в інтервалі 0–40 секунд затримка обробки зростає майже лінійно від початкових нульових значень до приблизно 20–30 мілісекунд, що є нормальною реакцією системи на зростання довжини черги повідомлень. У проміжку 40–120 секунд показник latency продовжує плавно збільшуватися до 60–80 мілісекунд без різких флуктуацій, що свідчить про стабільну роботу системи з контрольованими чергами на всіх етапах конвеєра.

Поблизу 120-ї секунди фіксується перший локальний стрибок затримки до приблизно 60 мілісекунд, після якого крива вирівнюється - це характерний сигнал про тимчасове накопичення черги в одному з обробників або короткочасне блокування ресурсів. На ділянці 120–200 секунд затримка зростає до 100–120 мілісекунд, що вказує на роботу системи на межі стабільності, але ще без критичних провалів продуктивності.

У зоні 210–230 секунд спостерігається різке підвищення показника затримки до 140–160 мілісекунд, після чого крива продовжує зростаючий тренд.

Це чітко свідчить про перехід системи у режим насичення, коли зростання затримки обробки стає головною причиною падіння пропускної здатності через накопичення черг та вичерпання ресурсів.

Збільшення кількості партицій удвічі призвело до зростання середнього показника транзакцій за секунду приблизно на 35–40%, що демонструє ефективне горизонтальне масштабування системи. Крива продуктивності стала значно плавнішою при низьких і середніх рівнях навантаження завдяки кращому розподілу повідомлень між споживачами. Пікове значення продуктивності суттєво підвищилося порівняно з початковою конфігурацією, що підтверджує позитивний вплив додаткового паралелізму.

Подальше збільшення кількості партицій до 9 показало менш виражене зростання показника TPS - лише приблизно на 10–20%, що вказує на наближення до межі ефективного масштабування. За умов високого навантаження система швидше входить у режим насичення через обмеження процесорних ресурсів, пропускної здатності мережі та швидкості дискових операцій локального кластера. У пікових точках навантаження спостерігається більше хвилеподібних коливань показника TPS, що є типовим для надмірної конкуренції між робочими процесами за доступ до спільних ресурсів системи.

У всіх трьох експериментальних конфігураціях абсолютний показник транзакцій за секунду зростає зі збільшенням кількості партицій Kafka, що підтверджує базову здатність системи до горизонтального масштабування. Однак ефект масштабування є нелінійним - кожне наступне збільшення кількості партицій дає менший відносний приріст продуктивності через фундаментальні обмеження апаратних ресурсів інфраструктури. Для локального експериментального середовища, розгорнутого на одному фізичному комп'ютері, така поведінка є очікуваною та не свідчить про недоліки архітектурного рішення системи.

Таким чином, усі три експериментальні серії демонструють однакову закономірність поведінки системи: збільшення кількості партицій Kafka і кількості екземплярів споживачів повідомлень призводить до підвищення

пропускної здатності системи, хоча темп цього зростання поступово зменшується та врешті досягає межі через фундаментальні апаратні обмеження інфраструктури. Поведінка системи у кожній з конфігурацій повністю відповідає теоретичній моделі горизонтального масштабування платформи Kafka та загальним принципам подієво-орієнтованої архітектури.

На основі отриманих експериментальних даних можна впевнено стверджувати, що гіпотеза H2 повністю підтверджена: система дійсно демонструє здатність до горизонтального масштабування, а збільшення кількості партицій приводить до пропорційного зростання пропускної здатності конвеєра обробки до моменту насичення ресурсів. Критично важливо зазначити, що точка насичення у цьому випадку визначалася саме ресурсними обмеженнями локального обчислювального середовища - процесорною потужністю, оперативною пам'яттю та пропускною здатністю дискової підсистеми.

4.3 Перевірка інваріанту подвійного бухгалтерського запису

Однією з фундаментальних вимог до фінансових систем є забезпечення коректності механізму подвійного бухгалтерського запису: для кожної транзакції загальна сума дебетових проводок має математично дорівнювати загальній сумі кредитових проводок. Це гарантує, що система не створює і не втрачає вартість у процесі обробки операцій, зберігаючи фундаментальний принцип збереження балансу.

Для експериментального підтвердження коректності роботи журналу обліку (ledger) було виконано автоматизовану перевірку всіх сформованих записів журналів транзакцій у базі даних. Перевірка здійснювалась безпосередньо над таблицею journal_lines, у якій зберігаються всі рядки бухгалтерських проводок типу дебет та кредит для кожного унікального ідентифікатора журналу (journal_id).

Для кожної транзакції системою було виконано SQL-запит до бази даних YugabyteDB, що підраховує загальну суму дебетових і кредитових проводок,

обчислює різницю між ними, а також визначає статус валідності журналу відповідно до інваріанту балансу.

journals | SQL Query | journal_immutability | SQL Query | journal_lines

```
SELECT
    journal_id,
    SUM(CASE WHEN side = 'D' THEN amount_usd ELSE 0 END) AS debit_sum,
    SUM(CASE WHEN side = 'C' THEN amount_usd ELSE 0 END) AS credit_sum,
    SUM(
        CASE
            WHEN side = 'D' THEN amount_usd
            WHEN side = 'C' THEN -amount_usd
        END
    ) AS balance
FROM public.journal_lines
GROUP BY journal_id
HAVING
    SUM(
        CASE
            WHEN side = 'D' THEN amount_usd
            WHEN side = 'C' THEN -amount_usd
        END
    ) = 0
ORDER BY journal_id;
```

line 39, column 7, location 1324 | No limit | Beautify ⌘I | Run Current ⌘↵

	journal_id	debit_sum	credit_sum	balance
1	019aa43f-9c73-7ac0-9f03-552fb48e7740	6630811.00	6630811.00	0.00
2	019aa43f-9c75-75d0-bff1-1d5c3b52458f	5856240.00	5856240.00	0.00
3	019aa43f-9ff2-7720-a2a7-5eecf99ed9d2	3886833.00	3886833.00	0.00
4	019aa43f-a018-7d23-9f3f-761bd1389542	6809850.00	6809850.00	0.00
5	019aa43f-a050-7f43-aae7-632bc4dbe5b6	7561320.00	7561320.00	0.00
6	019aa43f-a077-7251-a8dc-0338406fe977	3214805.00	3214805.00	0.00
7	019aa43f-a08d-7aa0-9c10-b03634721b79	4627164.00	4627164.00	0.00
8	019aa43f-a0bd-7bb2-8e40-f9ee9fcf86f5	10079908.00	10079908.00	0.00
9	019aa43f-a0c3-7150-b07a-ab614123b7ca	1577414.00	1577414.00	0.00
10	019aa43f-a0e8-73d3-9d9e-e083859aec0e	2357376.00	2357376.00	0.00
11	019aa43f-a10c-7ea3-8c4b-cf0ccaa5ec35	6243400.00	6243400.00	0.00
12	019aa43f-a12a-7530-b80a-950d6e0bd01b	446275.00	446275.00	0.00
13	019aa43f-a13a-7c63-869a-05430f205049	2992477.00	2992477.00	0.00
14	019aa43f-a157-79f3-ab30-975940e726e7	2055118.00	2055118.00	0.00
15	019aa43f-a17d-7473-8b4a-6fd5ad886048	6095882.00	6095882.00	0.00
16	019aa43f-a19c-7983-86ad-76cc802918f3	3184782.00	3184782.00	0.00

Data | Message | Chart | 251 ms | 67,936 rows | Export...

Рисунок 4.9 - Результат виконання запиту для обрахунку балансу

Результати вибірки показали, що усі без винятку журнали транзакцій мають баланс 0 тобто виконуються наступні умови:

– сума дебетових записів математично збігається з сумою кредитових записів для кожного журналу;

– відсутні неповні журнали з пропущеними проводками або незавершеними транзакціями;

– не зафіксовано жодного випадку порушення цілісності бухгалтерських проводок;

– паралельна асинхронна обробка подій через конвеєр Kafka не призвела до розривів транзакцій або дублювання записів у базі даних.

Отримані результати експериментально підтверджують, що система коректно реалізує фундаментальний принцип подвійного бухгалтерського запису

і забезпечує фінансову узгодженість даних навіть за умов високого навантаження та паралельної обробки транзакцій, що є критично важливою властивістю для надійних платіжних сервісів. На основі проведеної перевірки гіпотезу H1 про дотримання інваріанту подвійного запису можна вважати повністю підтвердженою експериментальними даними.

4.4 Перевірка криптографічних доказів незмінності даних

Для експериментальної валідації незмінності фінансових записів у системі було проведено тустування, у межах якого виконуваласяперевірка криптографічних доказів для транзакцій, зафіксованих у незмінному сховищі ImmuDB.

Експериментальна перевірка здійснюється за допомогою окремого валідаційного скрипта, який емулює поведінку зовнішнього аудитора та виконує послідовні кроки верифікації:

– Крок 1: Отримання фінансових записів через gRPC-інтерфейс Ledger Service: валідаційний скрипт викликає метод ListLedgerEntries з параметром ідентифікатора рахунку-фактури (invoice_id) або ідентифікатора запису журналу (journal_id), що повертає повний набір метаданих транзакції. Відповідь містить ідентифікатор запису журналу (journal_id), ідентифікатор транзакції у незмінному сховищі (immudb_tx_id), ідентифікатор шарду ImmuDB (immudb_shard_id), а також інші службові метадані для верифікації. Цей крок імітує реальну поведінку торговця (merchant), який отримує журнал операцій через офіційний публічний API системи без привілейованого доступу до внутрішніх структур.

– Крок 2: Незалежна перевірка записів у ImmuDB. Після отримання метаданих з Ledger Service валідаційний скрипт встановлює пряме з'єднання з відповідним шардом ImmuDB та виконує операцію верифікованого отримання даних VerifiedGet з параметрами ключа запису та ідентифікатора транзакції. Для цього використано спеціалізований алгоритм перевірки криптографічних доказів, який виконує наступні операції: запитує proof

включення запису до дерева Меркла з незмінного сховища, обчислює контрольний хеш на основі отриманих даних згідно з алгоритмом SHA-256, порівнює обчислений хеш з кореневим хешем дерева, отриманим від Ledger Service на попередньому кроці, а у випадку виявлення найменшої невідповідності між хешами одразу сигналізує про потенційне порушення цілісності або спробу модифікації історії.

– Крок 3: Зіставлення та валідація результатів. Якщо операція VerifiedGet успішно підтверджує, що запис існує у незмінному сховищі в очікуваному стані з коректним криптографічним хешем, що відповідає кореневому хешу дерева Меркла, тоді криптографічний доказ вважається валідним і транзакція підтверджується як незмінена з моменту первинної фіксації. Як показано на рисунку 4.10, записи транзакцій для конкретного торговця успішно верифіковані через незалежну перевірку криптографічних доказів.

Рисунок 4.10 - Успішно верифіковані транзакції торговця

Для експериментальної перевірки гіпотези Н3 про те, що жодна ретроспективна модифікація даних у журналі обліку не може залишитися

непоміченою при незалежній верифікації, було виконано спеціалізований тест зовнішньої валідації записів. Щоб експериментально перевірити здатність системи детектувати порушення цілісності даних, один із записів було навмисно модифіковано перед його передачею до процесу верифікації. У такій ситуації криптографічний доказ включення (inclusion proof) перестає математично відповідати кореню дерева Меркла, обчисленому на основі змінених даних, що неминуче призводить до виявлення невідповідності під час процесу верифікації.

Під час виконання валідаційного скрипта з модифікованими даними система видала наступне повідомлення про помилку, що показано на рис. 4.11:

Рисунок 4.11 - Помилка про пошкодження даних

Це повідомлення однозначно свідчить про те, що криптографічний доказ не пройшов математичну верифікацію, отже дані було змінено або пошкоджено після первинної фіксації у незмінному сховищі, і система детерміновано виявляє таке порушення цілісності.

Таким чином, проведений експеримент демонструє, що будь-яка спроба несанкціонованої модифікації історії транзакцій неминуче призводить до невалідності криптографічного доказу на основі дерева Меркла, а зовнішній незалежний верифікатор без привілейованого доступу до системи може легко ідентифікувати такі спотворення даних через математичну перевірку хеш-ланцюгів.

Отже, отримані експериментальні результати повністю підтверджують гіпотезу Н3 про криптографічну незмінність історії фінансових операцій та

можливість незалежної верифікації достовірності даних зовнішніми аудиторами поза межами довіреної системи без необхідності прямого доступу до внутрішніх структур даних.

Лістинг 4.1 – Код верифікації історії транзакцій

```
import grpc
import immudb
from immudb.client import ImmudbClient
from google.protobuf import json_format
from immudb.embedded import store
from immudb.grpc import schema_pb2
from immudb.grpc import schema_pb2_grpc
from immudb.rootService import RootService, State
from immudb import datatypes
from immudb.exceptions import ErrCorruptedData
import immudb.database as database
import immudb.schema as schema
import immudb.grpc.schema_pb2_grpc as immu_rpc
import immudb.grpc.schema_pb2 as immu_schema
from google.protobuf.json_format import MessageToDict
import google.protobuf.empty_pb2 as g_empty


#   python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. ledger.proto
import ledger_pb2
import ledger_pb2_grpc


def main():
    ledger_channel = grpc.insecure_channel("127.0.0.1:63054")
    ledger = ledger_pb2_grpc.LedgerServiceStub(ledger_channel)

    req = ledger_pb2.ListLedgerEntriesRequest(
        merchant_id="019aa49d-7955-7bd2-804c-b8760dc86ddc",
        limit=10,
    )
    resp = ledger.ListLedgerEntries(req)

    immu_channel = grpc.insecure_channel("127.0.0.1:63054")
    immu_stub = immu_rpc.ImmuServiceStub(immu_channel)

    for entry in resp.entries:
        meta = [("shard", str(entry.immudb_shard_id))]

        ver = verified_get_with_meta(
            service=immu_stub,
            key=entry.immudb_key.encode(),
            sinceTx=entry.immudb_tx_id,
            meta=meta,
        )

        decoded = ledger_pb2.ImmuLedgerValue()
        decoded.ParseFromString(ver.value)

        print("Verified success:", MessageToDict(decoded), "\n")


def verified_get_with_meta(
    service: schema_pb2_grpc.ImmuServiceStub,
    key: bytes,
    meta=None,
    atTx=None,
```

```
    sinceTx=None,
    atRevision=None,
    verifying_key=None,
    prev_state: State = None,
):
    if meta is None:
        meta = []

    raw_state = service.CurrentState(g_empty.Empty(), metadata=meta)

    state = State(
        db=raw_state.db,
        txId=raw_state.txId,
        txHash=raw_state.txHash,
        publicKey=raw_state.signature.publicKey,
        signature=raw_state.signature.signature,
    )

    req = schema_pb2.VerifiableGetRequest(
        keyRequest=schema_pb2.KeyRequest(
            key=key,
            atTx=atTx,
            sinceTx=sinceTx,
            atRevision=atRevision,
        ),
        proveSinceTx=state.txId,
    )

    ventry = service.VerifiableGet(req, metadata=meta)

    entrySpecDigest = store.EntrySpecDigestFor(
        int(ventry.verifiableTx.tx.header.version)
    )
    inclusionProof = schema.InclusionProofFromProto(ventry.inclusionProof)
    dualProof = schema.DualProofFromProto(ventry.verifiableTx.dualProof)

    if not ventry.entry.HasField("referencedBy"):
        vTx = ventry.entry.tx
        e = database.EncodeEntrySpec(
            key,
            schema.KVMetadataFromProto(ventry.entry.metadata),
            ventry.entry.value,
        )
    else:
        ref = ventry.entry.referencedBy
        vTx = ref.tx
        e = database.EncodeReference(
            ref.key,
            schema.KVMetadataFromProto(ref.metadata),
            ventry.entry.key,
            ref.atTx,
        )

    if state.txId <= vTx:
        eh = schema.DigestFromProto(ventry.verifiableTx.dualProof.targetTxHeader.eH)
        sourceid = state.txId
        sourcealh = schema.DigestFromProto(state.txHash)
        targetid = vTx
        targetalh = dualProof.targetTxHeader.Alh()
    else:
        eh = schema.DigestFromProto(ventry.verifiableTx.dualProof.sourceTxHeader.eH)
        sourceid = vTx
        sourcealh = dualProof.sourceTxHeader.Alh()
        targetid = state.txId
        targetalh = schema.DigestFromProto(state.txHash)

    if not store.VerifyInclusion(inclusionProof, entrySpecDigest(e), eh):
        raise Exception("Corrupted data: inclusion proof invalid")
```

```
if state.txId > 0:
    if not store.VerifyDualProof(
        dualProof, sourceid, targetid, sourcealh, targetalh
    ):
        raise Exception("Corrupted data: dual proof invalid")

newstate = State(
    db=state.db,
    txId=targetid,
    txHash=targetalh,
    publicKey=ventry.verifiableTx.signature.publicKey,
    signature=ventry.verifiableTx.signature.signature,
)
if verifying_key:
    newstate.Verify(verifying_key)

if ventry.entry.HasField("referencedBy"):
    refkey = ventry.entry.referencedBy.key
else:
    refkey = None

return datatypes.SafeGetResponse(
    id=vTx,
    key=ventry.entry.key,
    value=ventry.entry.value,
    timestamp=ventry.verifiableTx.tx.header.ts,
    verified=True,
    refkey=refkey,
    revision=ventry.entry.revision,
)
```

ВИСНОВКИ

У межах роботи проведено дослідження проблематики масових криптовалютних платежів та розроблено прототип платіжного терміналу, побудованого на мікросервісній подієво-орієнтованій архітектурі. На основі аналізу існуючих рішень визначено ключові вимоги до таких систем: масштабованість, гарантована фінансова узгодженість і можливість незалежного криптографічного аудиту.

Реалізована система поєднує розподілений SQL-рівень, подієвий конвеєр обробки та незмінне сховище з криптографічними доказами. Такий підхід забезпечує ACID-узгодженість подвійного бухгалтерського запису, горизонтальне масштабування обробки транзакцій та доведення незмінності історичних даних.

Проведені експерименти у локальному Kubernetes-кластері з різними конфігураціями (3, 6 та 9 партицій Kafka) продемонстрували зростання пропускної здатності системи до межі апаратних ресурсів. Поведінка TPS та затримки підтвердила гіпотезу про масштабованість подієвого конвеєра та передбачуване наближення до перенасичення при високих навантаженнях. Гіпотеза про коректність подвійного запису також підтверджена: жодного випадку порушення балансу не виявлено. За допомогою зовнішнього інструмента перевірки підтверджено й гіпотезу криптографічної незмінності — спроби модифікації записів призводять до помилки.

У підсумку, запропонований прототип демонструє, що поєднання розподілених транзакцій, подієвої архітектури та незмінних криптографічних структур є ефективною основою для будівництва масштабованих і надійних криптовалютних платіжних систем. Результати підтверджують доцільність такого підходу та відкривають можливості для подальшого розвитку прототипу в напрямку реального комерційного застосування.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017.

2. Deshmukh G., Nizamudeen S. M. T. Decentralized Business: A Guide to Transforming Business Strategies with Distributed Ledger Technologies, 2022.

3. Antonopoulos A. Mastering Bitcoin. O'Reilly Media, 2017.

4. Stopford B. Designing Event-Driven Systems. O'Reilly Media, 2018.

5. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

6. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications, 2018.

7. Kreps J. I Heart Logs: Event Data, Stream Processing, and Data Integration. O'Reilly Media, 2014.

8. Narkhede N., Shapira G., Palino T. Kafka: The Definitive Guide. O'Reilly Media, 2017.

9. Cockroach Labs. CockroachDB: The Official Guide. O'Reilly Media, 2021.

10. Horngren C. Accounting. Pearson Education, 2013.

11. Merkle R. Protocols for Public Key Cryptosystems. IEEE Symposium on Security and Privacy, 1980.

12. YugabyteDB Documentation [Електронний ресурс] — Режим доступу: https://docs.yugabyte.com/

13. Apache Kafka Documentation [Електронний ресурс] — Режим доступу: https://kafka.apache.org/documentation/

14. ClickHouse Documentation [Електронний ресурс] — Режим доступу: https://clickhouse.com/docs

15. Redis Cluster Documentation [Електронний ресурс] — Режим доступу: https://redis.io/docs/latest/

16. Kubernetes Documentation [Електронний ресурс] — Режим доступу: https://kubernetes.io/docs/concepts/

17. gRPC Documentation [Електронний ресурс] — Режим доступу: https://grpc.io/docs/

18. ImmuDB Documentation [Електронний ресурс] — Режим доступу: https://docs.immudb.io/

19. ImmuDB Research Paper: Cryptographically Verifiable Ledger [Електронний ресурс] — Режим доступу: https://immudb.io/research-paper

20. Sam Newman. Distributed Systems in One Lesson. O'Reilly, 2020.

21. Brendan Burns. Designing Distributed Systems. O'Reilly Media, 2018.

Додаток А

Реалізація подвійного запису в журнал

```
use chrono::Utc;
use common::{
    grpc_gen::root::events::{LedgerTxEvent, ProcessedTxEvent},
    kafka::{KAFKA_LEDGER_TX_STORED_TOPIC, KAFKA_PROCESSOR_TX_TOPIC, get_kafka_producer},
```

```
    stats::{LedgerTxStatEvent, StatEvent, send_stat_event},
    utils::convert_datetime_to_google_ts_opt,
};
use eyre::WrapErr;
use futures::TryStreamExt;
use prost::Message as ProtoMessage;
use rdkafka::{
    ClientConfig,
    config::RDKafkaLogLevel,
    consumer::{CommitMode, StreamConsumer},
    producer::FutureRecord,
    util::Timeout,
};
use rdkafka::{Message, consumer::Consumer};
use rust_decimal::{Decimal, prelude::FromPrimitive};
use std::{sync::Arc, time::Duration};
use uuid::Uuid;

use crate::{context::Context, result::Result};

pub async fn start_ledger_consumer(ctx: Arc<Context>) -> Result<()> {
    let consumer: StreamConsumer = ClientConfig::new()
        .set("group.id", "ledger-consumer")
        .set("bootstrap.servers", &ctx.config.kafka_brokers)
        .set("session.timeout.ms", "60000")
        .set("heartbeat.interval.ms", "3000")
        .set("max.poll.interval.ms", "300000")
        .set("enable.auto.commit", "false")
        .set("enable.auto.offset.store", "false")
        .set("auto.offset.reset", "earliest")
        .set("fetch.max.bytes", "52428800")
        .set("fetch.message.max.bytes", "1048576")
        .set("queued.max.messages.kbytes", "65536")
        .set("socket.timeout.ms", "30000")
        .set("socket.keepalive.enable", "true")
        .set("enable.partition.eof", "false")
        .set("partition.assignment.strategy", "range,roundrobin")
        .set_log_level(RDKafkaLogLevel::Info)
        .create()
        .expect("Failed to create Kafka consumer");

    consumer
        .subscribe(&[KAFKA_PROCESSOR_TX_TOPIC])
        .expect("Can't subscribe to specified topic");

    let producer = get_kafka_producer(&ctx.config.kafka_brokers, ctx.cancel_token.clone());

    let mut stream = consumer.stream();

    tracing::info!("Starting consumer...");

    while let Some(borrowed_message) = stream.try_next().await? {
        let accepted_at = Utc::now();
        let om = borrowed_message.detach();

        let Some(payload) = om.payload() else {
            tracing::error!("Empty payload at offset {:?}", om.offset());
            continue;
        };

        let Some((invoice, origin_event)) = ProcessedTxEvent::decode(payload)
            .ok()
            .and_then(|v| v.invoice.zip(v.origin_event))
        else {
            tracing::error!("Failed to decode ProcessedTxEvent");
            continue;
        };

        tracing::info!("Invoice: {:?} Origin: {:?}", invoice, origin_event);
```

```
let Some(amount_usd_paid) = invoice.amount_usd_paid else {
    tracing::error!("Invoice amount_usd_paid is None");
    continue;
};
let Some(amount_usd_merchant) = invoice.amount_usd_merchant else {
    tracing::error!("Invoice amount_usd_received is None");
    continue;
};
let amount_usd_fee = amount_usd_paid - amount_usd_merchant;

let mut client = ctx.get_yugabyte_connection().await?;

let tx = client.transaction().await?;

let journal_id = Uuid::now_v7();

let invoice_id_uuid = Uuid::parse_str(&invoice.id).unwrap();
tx.execute(
    r#"
      INSERT INTO public.journals (journal_id, invoice_id)
      VALUES ($1, $2)
      "#,
    &[&journal_id, &invoice_id_uuid],
)
.await
.wrap_err("Failed to insert journal")?;

let merchant_id_uuid = Uuid::parse_str(&invoice.merchant_id).unwrap();
let system_merchant_id = Uuid::parse_str(&ctx.config.system_merchant_id).unwrap();
tx.execute(
    r#"
       INSERT INTO public.journal_lines
           (line_id, journal_id, side, merchant_id, amount_usd)
       VALUES ($1, $2, 'D', $3, $4)
       "#,
    &[
        &Uuid::now_v7(),
        &journal_id,
        &merchant_id_uuid,
        &Decimal::from_u64(amount_usd_paid).unwrap(),
    ],
)
.await
.wrap_err("Failed to insert debit line")?;

tx.execute(
    r#"
       INSERT INTO public.journal_lines
           (line_id, journal_id, side, merchant_id, amount_usd)
       VALUES ($1, $2, 'C', $3, $4)
       "#,
    &[
        &Uuid::now_v7(),
        &journal_id,
        &merchant_id_uuid,
        &Decimal::from_u64(amount_usd_merchant).unwrap(),
    ],
)
.await
.wrap_err("Failed to insert merchant credit line")?;

tx.execute(
    r#"
     INSERT INTO public.journal_lines
         (line_id, journal_id, side, merchant_id, amount_usd)
     VALUES ($1, $2, 'C', $3, $4)
     "#,
    &[
```

```
        &Uuid::now_v7(),
        &journal_id,
        &system_merchant_id,
        &Decimal::from_u64(amount_usd_fee).unwrap(),
    ],
)
.await
.wrap_err("Failed to insert system fee line")?;

tx.execute(
    r#"
    INSERT INTO public.balances (merchant_id, balance_usd)
    VALUES ($1, $2)
    ON CONFLICT (merchant_id)
    DO UPDATE SET
        balance_usd = public.balances.balance_usd + EXCLUDED.balance_usd,
        updated_at = now()
    "#,
    &[
        &merchant_id_uuid,
        &Decimal::from_u64(amount_usd_merchant).unwrap(),
    ],
)
.await
.wrap_err("Failed to update merchant balance")?;

tx.execute(
    r#"
    INSERT INTO public.balances (merchant_id, balance_usd)
    VALUES ($1, $2)
    ON CONFLICT (merchant_id)
    DO UPDATE SET
        balance_usd = public.balances.balance_usd + EXCLUDED.balance_usd,
        updated_at = now()
    "#,
    &[
        &system_merchant_id,
        &Decimal::from_u64(amount_usd_fee).unwrap(),
    ],
)
.await
.wrap_err("Failed to update system merchant balance")?;

tx.commit().await?;

let key = invoice.merchant_id.clone();
let now = Utc::now();
let event = LedgerTxEvent {
    origin_event: Some(origin_event),
    journal_id: journal_id.to_string(),
    invoice_id: invoice.id.clone(),
    merchant_id: invoice.merchant_id.clone(),
    amount_usd_requested: invoice.amount_usd_requested,
    amount_usd_paid: amount_usd_paid,
    amount_usd_merchant: amount_usd_merchant,
    amount_usd_fee: amount_usd_fee,
    stored_at: convert_datetime_to_google_ts_opt(&now),
};
let mut payload = Vec::new();
event.encode(&mut payload)?;
let record = FutureRecord::to(KAFKA_LEDGER_TX_STORED_TOPIC)
    .key(&key)
    .payload(&payload);

let result = producer
    .send(record, Timeout::After(Duration::from_secs(60)))
    .await;

tracing::info!("Produced: {:?}", event);
```

```
        match result {
            Ok(_) => tracing::info!("Written to journal: {}", journal_id),
            Err(e) => {
                tracing::error!("Error writing to journal: {:?}", e);
                return Err(eyre::eyre!(e.0));
            }
        }

        if let Err(err) = consumer
            .commit_message(&borrowed_message, CommitMode::Async)
            .wrap_err("Failed to commit offset")
        {
            tracing::error!("{:?}", err);
        }

        send_stat_event(
            producer.clone(),
            key,
            StatEvent::LedgerTx(
                LedgerTxStatEvent::builder()
                    .journal_id(journal_id.to_string())
                    .invoice_id(invoice.id)
                    .accepted_at(accepted_at)
                    .processed_at(Utc::now())
                    .build(),
            ),
        );
    }

    Ok(())
}
```

Додаток Б

Реалізація запису в ImmuDb

```
use chrono::Utc;
use common::{
    grpc_gen::root::{
        events::{LedgerConfirmedTxEvent, LedgerTxEvent},
        google::protobuf::Empty,
        immudb::schema::{
            KeyValue, OpenSessionRequest, SetRequest, VerifiableSetRequest, VerifiableTx,
            immu_service_client::ImmuServiceClient,
        },
        ledger::v1::ImmuLedgerValue,
    },
    kafka::{KAFKA_LEDGER_TX_CONFIRMED_TOPIC, KAFKA_LEDGER_TX_STORED_TOPIC, get_kafka_producer},
    stats::{LedgerConfirmedTxStatEvent, StatEvent, send_stat_event},
    utils::convert_datetime_to_google_ts_opt,
};
use eyre::WrapErr;
use futures::TryStreamExt;
use prost::Message as ProtoMessage;
use rdkafka::{
    ClientConfig,
    config::RDKafkaLogLevel,
    consumer::{CommitMode, StreamConsumer},
    producer::FutureRecord,
    util::Timeout,
};
use rdkafka::{Message, consumer::Consumer};
use std::{collections::HashMap, str::FromStr, sync::Arc, time::Duration};
use tonic::{
    Request,
    transport::{Channel, Endpoint},
};
use uuid::Uuid;

use crate::{context::Context, result::Result};

pub async fn start_immu_consumer(ctx: Arc<Context>) -> Result<()> {
    let mut immu_tx_ids: HashMap<usize, u64> = HashMap::new();
    let mut immu_session_tokens: HashMap<usize, String> = HashMap::new();
    let mut immu_clients: HashMap<usize, ImmuServiceClient<Channel>> = HashMap::new();

    for (index, addr) in ctx.config.immudb_shards_address.iter().enumerate() {
        tracing::info!("Connecting to immudb {}", addr);

        let channel = Endpoint::from_str(addr)?
            .connect_timeout(Duration::from_secs(3))
            .timeout(Duration::from_secs(5))
            .tcp_keepalive(Some(Duration::from_secs(600)))
            .connect()
            .await?;

        let mut client = ImmuServiceClient::new(channel);

        let response = client
            .open_session(OpenSessionRequest {
                username: ctx.config.immudb_username.as_bytes().to_vec(),
                password: ctx.config.immudb_password.as_bytes().to_vec(),
                database_name: ctx.config.immudb_database.clone(),
            })
            .await?
            .into_inner();

        immu_session_tokens.insert(index, response.session_id.clone());
```

```
    let mut request = Request::new(Empty {});
    let meta = request.metadata_mut();
    meta.insert("sessionid", response.session_id.parse().unwrap());

    let response = client.current_state(request).await?.into_inner();

    immu_tx_ids.insert(index, response.tx_id);
    immu_clients.insert(index, client);

    tracing::info!("Connected to immudb shard {}", index);
}

let producer = get_kafka_producer(&ctx.config.kafka_brokers, ctx.cancel_token.clone());

let consumer: StreamConsumer = ClientConfig::new()
    .set("group.id", "ledger-immu-consumer")
    .set("bootstrap.servers", &ctx.config.kafka_brokers)
    .set("session.timeout.ms", "60000")
    .set("heartbeat.interval.ms", "3000")
    .set("max.poll.interval.ms", "300000")
    .set("enable.auto.commit", "false")
    .set("enable.auto.offset.store", "false")
    .set("auto.offset.reset", "earliest")
    .set("fetch.max.bytes", "52428800")
    .set("fetch.message.max.bytes", "1048576")
    .set("queued.max.messages.kbytes", "65536")
    .set("socket.timeout.ms", "30000")
    .set("socket.keepalive.enable", "true")
    .set("enable.partition.eof", "false")
    .set("partition.assignment.strategy", "range,roundrobin")
    .set_log_level(RDKafkaLogLevel::Info)
    .create()
    .expect("Failed to create Kafka consumer");

consumer
    .subscribe(&[KAFKA_LEDGER_TX_STORED_TOPIC])
    .expect("Can't subscribe to specified topic");

let mut stream = consumer.stream();

tracing::info!("Starting consumer...");

while let Some(borrowed_message) = stream.try_next().await? {
    let accepted_at = Utc::now();
    let om = borrowed_message.detach();

    let Some(payload) = om.payload() else {
        tracing::error!("Empty payload at offset {:?}", om.offset());
        continue;
    };

    let Some(message) = LedgerTxEvent::decode(payload).ok() else {
        tracing::error!("Failed to decode LedgerTxEvent");
        continue;
    };

    tracing::info!("Received {:?}", message);

    let Some(origin_event) = message.origin_event else {
        tracing::error!("Failed to extract origin event");
        continue;
    };

    let shard_index = fastrand::usize(..immu_clients.len());

    let tx_id = immu_tx_ids.get(&shard_index).unwrap();

    let key = format!("{}:{}", message.merchant_id, message.journal_id);
```

```
        let confirmed_at = convert_datetime_to_google_ts_opt(&Utc::now());
        let value = ImmuLedgerValue {
            invoice_id: message.invoice_id.clone(),
            merchant_id: message.merchant_id,
            amount_usd_requested: message.amount_usd_requested,
            amount_usd_paid: message.amount_usd_paid,
            amount_usd_merchant: message.amount_usd_merchant,
            amount_usd_fee: message.amount_usd_fee,
            currency: origin_event.currency,
            network: origin_event.network,
            wallet_address: origin_event.wallet_address,
            journal_id: message.journal_id.clone(),
            blockchain_tx_id: origin_event.tx_id,
            stored_at: message.stored_at,
            confirmed_at,
        };

        let response: VerifiableTx = loop {
            let client = immu_clients.get_mut(&shard_index).unwrap();
            let key_value = KeyValue {
                key: key.as_bytes().to_vec(),
                value: value.encode_to_vec(),
                metadata: None,
            };

            let mut request = Request::new(VerifiableSetRequest {
                set_request: Some(SetRequest {
                    k_vs: vec![key_value],
                    no_wait: true,
                    preconditions: Vec::new(),
                }),
                prove_since_tx: *tx_id,
            });

            let meta = request.metadata_mut();
            meta.insert(
                "sessionid",
                immu_session_tokens
                    .get(&shard_index)
                    .unwrap()
                    .parse()
                    .unwrap(),
            );

            let response =
                tokio::time::timeout(Duration::from_secs(5),
client.verifiable_set(request)).await;

            match response {
                Ok(Ok(resp)) => break resp.into_inner(),

                Ok(Err(err)) => {
                    return Err(eyre::eyre!(err));
                }

                Err(_) => {
                    tracing::warn!("immudb timeout — recreating client...");

                    let channel =
                        Endpoint::from_str(&ctx.config.immudb_shards_address[shard_index])?
                            .connect_timeout(Duration::from_secs(3))
                            .timeout(Duration::from_secs(5))
                            .tcp_keepalive(Some(Duration::from_secs(600)))
                            .connect()
                            .await?;

                    let client = ImmuServiceClient::new(channel);
```

```
            immu_clients.insert(shard_index, client);

            continue;
        }
    }
};

let Some(new_tx_id) = response.tx.and_then(|v| v.header).map(|v| v.id) else {
    tracing::error!("Failed to extract transaction ID");
    continue;
};

let Some(root) = response
    .dual_proof
    .and_then(|v| v.linear_proof)
    .and_then(|v| v.terms.last().cloned())
else {
    tracing::error!("Failed to extract linear proof");
    continue;
};

let client = ctx.get_yugabyte_connection().await?;
let stmt = r#"
        INSERT INTO public.journal_immutability
            (journal_id, immudb_tx_id, immudb_shard, immudb_root)
        VALUES ($1, $2, $3, $4)
        "#;

client
    .execute(
        stmt,
        &[
            &Uuid::parse_str(&message.journal_id).unwrap(),
            &(*tx_id as i64),
            &(shard_index as i16),
            &root,
        ],
    )
    .await?;

let event = LedgerConfirmedTxEvent {
    invoice_id: message.invoice_id.clone(),
    journal_id: message.journal_id.clone(),
    immudb_id: *tx_id,
    immudb_shard_id: shard_index as u64,
    immudb_root: root,
    confirmed_at,
};

let mut payload = Vec::new();
event.encode(&mut payload)?;
let key = message.journal_id.clone();
let record = FutureRecord::to(KAFKA_LEDGER_TX_CONFIRMED_TOPIC)
    .key(&key)
    .payload(&payload);

let result = producer
    .send(record, Timeout::After(Duration::from_secs(60)))
    .await;

tracing::info!("Produced: {:?}", event);

match result {
    Ok(_) => tracing::info!("Confirmed {}", message.journal_id),
    Err(e) => {
        tracing::error!("Error: {:?}", e);
        return Err(eyre::eyre!(e.0));
    }
}
```

```
        consumer
            .commit_message(&borrowed_message, CommitMode::Async)
            .wrap_err("Failed to commit offset")?;

        immu_tx_ids.insert(shard_index, new_tx_id);

        send_stat_event(
            producer.clone(),
            key,
            StatEvent::LedgerConfirmedTx(
                LedgerConfirmedTxStatEvent::builder()
                    .invoice_id(message.invoice_id)
                    .journal_id(message.journal_id)
                    .immudb_id(new_tx_id)
                    .immudb_shard_id(shard_index as u64)
                    .accepted_at(accepted_at)
                    .processed_at(Utc::now())
                    .build(),
            ),
        );
    }

    Ok(())
}
```

Додаток В

Реалізація опрацювання подій від блокчейнів

```
use chrono::Utc;
use common::{
    grpc_gen::root::{
        events::{BlockchainTxEvent, LedgerConfirmedTxEvent, ProcessedTxEvent},
        processor::v1::{
            ConfirmInvoiceRequest, Currency, FinalizeInvoiceRequest, ResolveActiveInvoiceRequest,
        },
    },
    kafka::{
        KAFKA_BLOCKCHAINS_TX_TOPIC, KAFKA_LEDGER_TX_CONFIRMED_TOPIC, KAFKA_PROCESSOR_TX_TOPIC,
        get_kafka_producer,
    },
    stats::{ProcessedTxStatEvent, StatEvent, send_stat_event},
    utils::convert_datetime_to_google_ts_opt,
};
use consumer_worker::ConsumerWorker;
use futures::stream;
use prost::Message;
use rdkafka::{producer::FutureRecord, util::Timeout};
use rust_decimal::prelude::*;
use rust_decimal::{Decimal, prelude::FromPrimitive};
use std::{sync::Arc, time::Duration};
use stream::StreamExt;
use crate::{context::Context, result::Result};
const SERVICE_FEE_RATE: f64 = 0.015;
const BUFFER_SIZE: usize = 5000;

pub async fn start_blockchains_tx_consumer(ctx: Arc<Context>) -> Result<()> {
    let worker = ConsumerWorker::builder()
        .buffer_size(BUFFER_SIZE)
        .flush_seconds(2)
        .group("processor-tx-consumer".into())
        .cancel_token(ctx.cancel_token.clone())
        .kafka_brokers(ctx.config.kafka_brokers.clone())
        .topics(vec![KAFKA_BLOCKCHAINS_TX_TOPIC.into()])
        .build();

    let producer = get_kafka_producer(&ctx.config.kafka_brokers, ctx.cancel_token.clone());

    worker
        .start(
            |x, _om| Ok::<_, eyre::Error>(BlockchainTxEvent::decode(x)?),
            move |messages| {
                let ctx = ctx.clone();
                let producer = producer.clone();
                async move {
                    let results = stream::iter(messages.into_iter())
                        .map(move |message| {
                            let mut service = ctx.processor_service.clone();
                            let producer = producer.clone();
                            async move {
                                let accepted_at = Utc::now();
                                let Some(invoice) = service
                                    .resolve_active_invoice(ResolveActiveInvoiceRequest {
                                        wallet_address: message.wallet_address.clone(),
                                    })
                                    .await?
                                    .into_inner()
                                    .invoice
                                else {
                                    tracing::warn!("Invoice resolution is failed");
                                    return Ok(());
                                };
```

```
                                tracing::info!("Invoice: {:?}", invoice);

                                let Some(currency) = invoice
                                    .currency
                                    .zip(invoice.network)
                                    .map(|(name, network)| Currency { name, network })
                                else {
                                    tracing::warn!("Currency is missing");
                                    return Ok(());
                                };

                                let amount = message.amount.parse::<u128>().unwrap();

                                let Some(amount_usd_paid) = currency.to_usd_cents(amount) else {
                                    tracing::warn!("Failed to convert amount to USD");
                                    return Ok(());
                                };

                                if amount_usd_paid
                                    < Decimal::from_u64(invoice.amount_usd_requested).unwrap()
                                {
                                    tracing::warn!("Amount is less than requested");
                                    return Ok(());
                                }

                                let service_fee =
                                    amount_usd_paid                                                 *
Decimal::from_f64(SERVICE_FEE_RATE).unwrap();
                                let amount_usd_merchant = amount_usd_paid - service_fee;

                                let invoice_id = invoice.id.clone();
                                let merchant_id = invoice.merchant_id.clone();
                                let response = service
                                    .finalize_invoice(FinalizeInvoiceRequest {
                                        invoice_id: invoice_id.clone(),
                                        amount_usd_paid: amount_usd_paid.to_u64().unwrap(),
                                        amount_usd_merchant:
amount_usd_merchant.to_u64().unwrap(),
                                        service_fee: service_fee.to_u64().unwrap(),
                                        tx_id: message.tx_id.clone(),
                                        paid_at: message.recorded_at,
                                    })
                                    .await?;

                                let Some(invoice) = response.into_inner().invoice else {
                                    eyre::bail!("Failed to finalize invoice");
                                };

                                let key = invoice.merchant_id.clone();
                                let event = ProcessedTxEvent {
                                    invoice: Some(invoice),
                                    origin_event: Some(message),
                                    processed_at: convert_datetime_to_google_ts_opt(&Utc::now()),
                                };

                                let mut payload = Vec::new();
                                event.encode(&mut payload)?;

                                let record = FutureRecord::to(KAFKA_PROCESSOR_TX_TOPIC)
                                    .key(&key)
                                    .payload(&payload);

                                let result = producer
                                    .send(record, Timeout::After(Duration::from_secs(60)))
                                    .await;

                                tracing::info!("Produced: {:?}", event);

                                match result {
```

```
                                    Ok(_) => tracing::info!("Invoice finalized: {}", invoice_id),
                                    Err(e) => tracing::error!("Error finalizing invoice: {:?}",
e),
                                }

                                send_stat_event(
                                    producer,
                                    invoice_id.clone(),
                                    StatEvent::ProcessedTx(
                                        ProcessedTxStatEvent::builder()
                                            .invoice_id(invoice_id)
                                            .order_id(merchant_id)
                                            .accepted_at(accepted_at)
                                            .processed_at(Utc::now())
                                            .build(),
                                    ),
                                );

                                Ok::<(), eyre::Error>(())
                            }
                        })
                        .buffer_unordered(BUFFER_SIZE)
                        .collect::<Vec<_>>()
                        .await;

                    for r in results {
                        if let Err(e) = r {
                            tracing::error!("Error finalizing invoice: {}", e);
                        }
                    }

                    Ok(())
                }
            },
        )
        .await?;

    Ok(())
}
pub async fn start_ledger_consumer(ctx: Arc<Context>) -> Result<()> {
    let worker = ConsumerWorker::builder()
        .buffer_size(BUFFER_SIZE)
        .flush_seconds(1)
        .group("processor-ledger-consumer".into())
        .cancel_token(ctx.cancel_token.clone())
        .kafka_brokers(ctx.config.kafka_brokers.clone())
        .topics(vec![KAFKA_LEDGER_TX_CONFIRMED_TOPIC.into()])
        .build();

    worker
        .start(
            |x, _om| Ok::<_, eyre::Error>(LedgerConfirmedTxEvent::decode(x)?),
            move |messages| {
                let ctx = ctx.clone();

                async move {
                    let results = stream::iter(messages.into_iter())
                        .map(move |message| {
                            let mut service = ctx.processor_service.clone();
                            async move {
                                let LedgerConfirmedTxEvent {
                                    invoice_id,
                                    journal_id,
                                    confirmed_at,
                                    ..
                                } = message;

                                service
                                    .confirm_invoice(ConfirmInvoiceRequest {
```

```
                                    invoice_id,
                                    journal_id,
                                    confirmed_at,
                                })
                                .await?;

                            Ok::<(), eyre::Error>(())
                        }
                    })
                    .buffer_unordered(BUFFER_SIZE)
                    .collect::<Vec<_>>()
                    .await;

                for r in results {
                    if let Err(e) = r {
                        tracing::error!("Error finalizing invoice: {}", e);
                    }
                }

                Ok(())
            }
        },
    )
    .await?;
Ok(())
}
```