МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кафедра моделювання і програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка програмного комплексу автоматизації управління арбітражем криптовалютних активів

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студент гр. ІПЗ-21-1
А. І. Пікалов

Керівник кваліфікаційної роботи А. М. Стрюк

Завідувач кафедри А. М. Стрюк

Кривий ріг

2025

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри

А. М. Стрюк

«____» ____20___р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-21-1 Пікалову Андрію Ігоровичу

1. Тема: Розробка програмного комплексу автоматизації управління арбітражем криптовалютних активів
2. затверджено наказом по КНУ № 7 від «31» січня 2025 р.
3. Термін подання студентом закінченої роботи: «2» червня 2023р.
4. Вихідні дані по роботі: розроблювана система повинна виконувати моніторинг криптовалютних бірж, шукати арбітражні можливості, взаємодіяти з Telegram-ботом і смарт-контрактами.
5. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз існуючих рішень, вибрати алгоритми пошуку арбітражу, спроєктувати архітектуру системи, реалізувати Telegram-бота та смарт-контракти, протестувати роботу комплексу.
6. Перелік ілюстративного матеріалу: архітектурні схеми системи, блок-схеми роботи модулів, схема взаємодії компонентів, знімки Telegram-інтерфейсу

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Аналіз аналогів та формування вимог	27.03.2025 – 28.03.2025
2	Збір даних про блокчейн	29.03.2025 – 09.04.2025
3	Проєктування структури Telegram-бота	10.04.2025 – 18.04.2025
4	Розробка смарт-контракту	19.04.2025 – 23.04.2025
5	Розробка сервісу моніторингу арбітражних можливостей	24.04.2025 – 30.04.2025
6	Інтеграція з DEX біржами	01.04.2025 – 04.04.2025
7	Тестування та виправлення помилок	05.04.2025 – 13.05.2025
8	Підготовка 1–2 розділів записки	03.05.2025 – 20.05.2025
9	Підготовка 3–4 розділів записки	21.05.2025 – 25.05.2025
10	Оформлення записки	26.05.2025 – 30.05.2025

Дата видачі завдання: «28» Січня 2025 р.

Студент А. І. Пікалов

Керівник роботи А. М. Стрюк

РЕФЕРАТ

АРБІТРАЖ, КРИПТОВАЛЮТА, БЛОКЧЕЙН, ТЕЛЕГРАМ-БОТ, СМАРТ-КОНТРАКТИ, АВТОМАТИЗАЦІЯ, БІРЖІ.

Пояснювальна записка: 57 с., 2 табл., 16 рис., 1 дод., 13 джерел.

Предметом кваліфікаційної роботи є проектування та впровадження програмного комплексу, що автоматизує управління процесом арбітражу криптовалютних активів на різних платформах.

Під час роботи було проведено аналіз поточних інструментів для моніторингу криптовалют та аналізу фінансових операцій. Доведено механізми міжбіржового арбітражу, їх подолання та затримки у свідомості високої волатильності ринку.

Була запитана архітектура бота, який автоматично відкриває видимі можливості для арбітражу, оцінює можливість оплати за допомогою комісій і виконує ряд операцій обміну API. За відсутності інтерактивного з’єднання через Telegram ви можете постійно віддалено контролювати спілкування та взаємодіяти в локальних системах.

Наразі стало можливим використання смарт-контрактів для збільшення швидкості та без можливості використання операцій на міських мобільних платформах. Розроблено прототип інтеграції зі смарт-контрактами, щоб зберегти логіку приєднання до децентралізованої середини.

Програмний комплекс реалізовано з функціональною модульністю, що дозволяє розширювати функціональні можливості та підключати нове покоління. Проведено тестування систем на історичних і реальних ринках, результати яких підтверджують ефективність запропонованої розробки. Основні результати дослідження та впровадження програмного комплексу були представлені на студентському науковому семінарі.

ABSTRACT

ARBITRAGE, CRYPTOCURRENCY, BLOCKCHAIN, TELEGRAM BOT, SMART CONTRACTS, AUTOMATION, EXCHANGES.

Thesis in 57 p., 2 tab., 16 fig., 1 app., 13 references.

The subject of qualifying work on the design and implementation of a software package that automates the management of the arbitrage process of cryptocurrency assets on different platforms.

During the work, an analysis of current tools for monitoring cryptocurrencies and analyzing financial transactions was carried out. The mechanisms of inter-exchange arbitrage, their overcoming and delays in the minds of high market volatility are proven.

The architecture of a bot was requested that automatically opens visible opportunities for arbitrage, evaluates the possibility of payment using commissions, and performs a series of API exchange operations. In the absence of an interactive connection via Telegram, you can constantly remotely monitor communication and interact in local systems.

Currently, it has become possible to use smart contracts to increase speed and without the possibility of using operations on urban mobile platforms. A prototype of integration with smart contracts has been developed to preserve the logic of joining the decentralized middle.

The software package is implemented with functional modularity, which allows you to expand functionality and connect a new generation. Testing of systems on historical and actual markets has been carried out, the results of which confirm the effectiveness of the proposed development. The main results of the research and implementation of the software package were presented at the student scientific seminar.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Термін (укр.)	Термін (англ.)	Пояснення
Арбітраж	Arbitrage	Стратегія отримання прибутку за рахунок різниці цін на один і той самий актив на різних ринках або біржах.
Криптовалюта	Cryptocurrency	Цифровий актив, який використовує криптографію для забезпечення безпеки транзакцій і працює на основі блокчейна.
Блокчейн	Blockchain	Розподілена база даних або реєстр, що зберігає записи про транзакції у вигляді ланцюга блоків, кожен з яких захищений криптографічно.
Телеграм-бот	Telegram Bot	Програмний агент, інтегрований у месенджер Telegram, що виконує автоматизовані дії за запитами користувача.
Смарт-контракти	Smart Contracts	Самовиконувані контракти з умовами, прописаними в коді, які автоматично виконуються у блокчейн-мережі.
Автоматизація	Automation	Процес зменшення або усунення участі людини у виконанні повторюваних дій за допомогою програмних рішень.
Біржі	Exchanges	Онлайн-платформи для купівлі, продажу або обміну криптовалют. Включають централізовані (CEX) та децентралізовані (DEX) біржі.

ВСТУП

Сьогодні про криптовалюту знає практично кожен - особливо в Україні, де дуже багато людей вже мають криптоактиви. Крипта відкриває купу можливостей для заробітку: можна просто зберігати її на гаманцях, а можна займатись активними діями - наприклад, трейдингом або арбітражем. Саме арбітраж став темою моєї роботи. Я вирішив створити бота, який буде автоматизувати цей процес.

Але не просто бота, а такого, що працює з децентралізованими біржами і використовує смарт-контракти. Це важливий момент, адже смарт-контракти дають змогу виконувати одразу кілька дій в рамках однієї транзакції. Це дозволяє мінімізувати ризики - бо операції відбуваються атомарно, тобто або все, або нічого: купівля одного токена і обмін на інший проходять за один “постріл”, і ми не втрачаємо гроші через зміну курсу між діями.

Чому взагалі виникає арбітраж? Ринки не синхронізовані: є затримки в оновленні курсів, різна ліквідність, обсяги торгів. На одній біржі токен ще дешевий, а на іншій вже подорожчав. Якщо встигнути - можна купити дешевше і продати дорожче. Але вручну це майже неможливо: ринок змінюється дуже швидко, треба постійно моніторити ситуацію, швидко приймати рішення, перекидати активи між біржами.

Ось чому автоматизація - це must have. Програма може відстежувати курси в реальному часі, шукати вигідні можливості, враховувати комісії, затримки, і все це - швидко і без участі людини. А коли ще додати до цього смарт-контракти - виходить прозора, безпечна і повністю автономна система арбітражу.

Мета моєї дипломної роботи - створити програмний комплекс, який:

- автоматично виявляє арбітражні ситуації;
- приймає рішення, чи вигідно виконувати угоду;
- керується через простий Telegram-інтерфейс.

При цьому система повинна бути гнучкою, масштабованою, і мати потенціал до подальшої інтеграції з іншими DEX-платформами та сервісами в блокчейні.

Чому ця тема важлива? Тому що вона об’єднує два сучасних тренди: автоматизацію криптотрейдингу і розвиток смарт-контрактів. Такий комплекс не тільки економить час трейдера, але й значно знижує ризики, пов’язані з людським фактором. А Telegram-інтерфейс дає можливість керувати всім з телефону - у зручній формі, в режимі реального часу.

У підсумку, ця робота має не лише теоретичну, а й цілком практичну цінність. У результаті буде створено інструмент, яким реально можна користуватись у сфері криптовалютного арбітражу. В межах дослідження я розгляну, як працюють криптобіржі, як виявляються арбітражні можливості, як підключатись до API бірж, як розробляються Telegram-боти, і як інтегрувати смарт-контракти в блокчейн-системи.

1 СУЧАСНИЙ СТАН І АКТУАЛЬНІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.1 Критичний аналіз літературних джерел за темою

Криптовалюти вже давно не дивина - сьогодні про них чули майже всі, особливо в Україні. За останні десять років ця сфера вибухнула: з’явилось безліч бірж, платіжних сервісів, нових способів заробітку, і окрема величезна галузь - децентралізовані фінанси, або просто DeFi.

Разом із цим виникла і нова потреба: як не загубитися в цьому потоці можливостей і при цьому заробляти ефективно. Один зі способів - це арбітраж, тобто коли можна купити актив дешевше на одній біржі, а продати дорожче на іншій. На перший погляд звучить просто. Але в реальності все набагато складніше: курси змінюються кожні секунди, біржі працюють по-різному, і все треба встигати робити дуже швидко.

Щоб розібратися, як це працює, я передивився купу джерел - від наукових статей до документацій по API. Усе можна звести до кількох основних тем:

- Трейдинг на CEX і DEX (тобто централізованих і децентралізованих біржах), де зазвичай використовують моделі, що прогнозують ціни.
- Арбітражні стратегії, включаючи міжбіржовий, міжпарний і трикутний арбітраж.
- Telegram-боти, які допомагають автоматизувати частину роботи або просто зручно моніторити ситуацію.
- Смарт-контракти, що дозволяють автоматично проводити складні операції прямо в блокчейні без участі людини.

Проте більшість існуючих рішень або складні у використанні, або створені під великі компанії. А ще багато з них - це просто “аналітика без дій”: можна подивитися дані, але не можна натиснути кнопку і реально щось купити чи продати.

Тому я вирішив створити свій інструмент - просту, але потужну систему, яка:

- автоматично знаходить арбітражні можливості;
- керується прямо з Telegram, без зайвих панелей і графіків;
- і працює зі смарт-контрактами, щоб усе виконувалось миттєво, в рамках однієї транзакції - без ризиків, пов’язаних із затримками чи зміною курсу.

Мета цієї роботи - не просто описати концепцію, а створити реально працюючий інструмент, яким зможе користуватися будь-хто, навіть без глибоких технічних знань.

1.2 Актуальність теми кваліфікаційної роботи

У контексті нестабільності традиційних фінансових ринків та поступового переходу до цифрової економіки криптовалюти стали інструментом, який не тільки забезпечує інвестиційні можливості, а й відкриває нові форми економічної взаємодії. Однією з таких форм є арбітраж, який, при правильному підході, дозволяє отримувати дохід майже без ринкового ризику.

Однак реалізація арбітражних стратегій на практиці стикається з низкою проблем:

- Необхідність швидкої реакції на зміни цін;
- Високі комісії за перекази між біржами;
- Технічна складність реалізації багатопоточних перевірок;
- Відсутність готових рішень з підтримкою Telegram або Web-інтерфейсу;
- Обмежена доступність децентралізованих протоколів для звичайних користувачів.

З огляду на ці проблеми, розробка програмного комплексу, який дозволяє автоматизувати виявлення та реалізацію арбітражних можливостей із можливістю керування через Telegram, є надзвичайно актуальною.

Більше того, врахування перспектив використання смарт-контрактів дозволяє підвищити надійність, швидкість та прозорість виконання угод, що особливо важливо у світі децентралізованих фінансів.

1.3 Цілі та завдання кваліфікаційної роботи

Мета моєї роботи - створити зручний інструмент, який сам стежитиме за можливостями для арбітражу між криптобіржами і дозволить керувати всім цим прямо з Telegram.

Ідея така: бот сам моніторить курси, перевіряє, чи є вигідна можливість, і якщо все ок - може дати команду на угоду. А ти в цей час просто отримуєш повідомлення і можеш реагувати або просто спостерігати.

Щоб усе це запрацювало, потрібно зробити кілька важливих речей:

а) Спочатку розібратися, як зараз працює арбітраж у крипті, які є методи, як люди автоматизують цей процес, і що з цього реально можна використовувати.

б) Потім треба спроєктувати систему - розділити її на частини: одна відповідатиме за моніторинг курсів, інша - за ухвалення рішень, ще одна - за зв’язок із біржами.

в) Далі - створити Telegram-бота, через якого можна буде все це контролювати. Щоб було просто: подивився, які угоди пройшли, отримав сповіщення, натиснув кнопку - і все працює.

г) Обов’язково потрібно додати механізми, які будуть перевіряти, чи вигідно взагалі щось робити. Бо не кожна арбітражна можливість насправді варта уваги - є комісії, затримки, ризики.

д) Якщо вийде - додати роботу зі смарт-контрактами, щоб операції на DEX’ах проходили автоматично і в одній транзакції - це зробить процес безпечнішим і швидшим.

е) І на завершення - протестувати все це на реальних біржах, щоб переконатись, що система стабільна, безпечна і справді допомагає.

В результаті має вийти не просто диплом, а реальний корисний інструмент - для себе, для інших трейдерів, для тих, хто хоче займатись арбітражем, але не хоче сидіти за комп’ютером цілий день.

2 РОЗРОБКА ФУНКЦІОНАЛЬНОЇ СХЕМИ І АЛГОРИТМУ

2.1 Розробка та докладний опис функціональної схеми програми

Мій проєкт - це не просто набір окремих частин, а справжня злагоджена система, яка сама знаходить вигідні арбітражні можливості між криптобіржами і одразу діє. Побачила шанс - купила дешевше, продала дорожче, і все це відбулося миттєво, без зайвого втручання.

Система працює не лише з класичними біржами, а й з децентралізованими - через смарт-контракти на Solana. Це важливо, бо такі угоди безпечні й проходять «атомарно» - все відбувається в межах однієї транзакції, без ризику втрат чи зависань.

Ще одна фіча - Telegram-бот. Я зробив так, щоб керувати всім можна було прямо зі смартфона. Можна переглядати ситуацію, отримувати сповіщення про угоди, запускати або зупиняти моніторинг - і все це через звичайне листування з ботом, без зайвих технічних складностей.

Щоб це все працювало злагоджено, я спочатку продумав архітектуру: хто за що відповідає, як передаються дані, як приймаються рішення. І вже на цьому фундаменті зібрав базу даних та з’єднав усі модулі між собою.

2.1.1 Компоненти системи та їх взаємодія

Функціональна схема програмного комплексу представлена на рисунку 2.1.

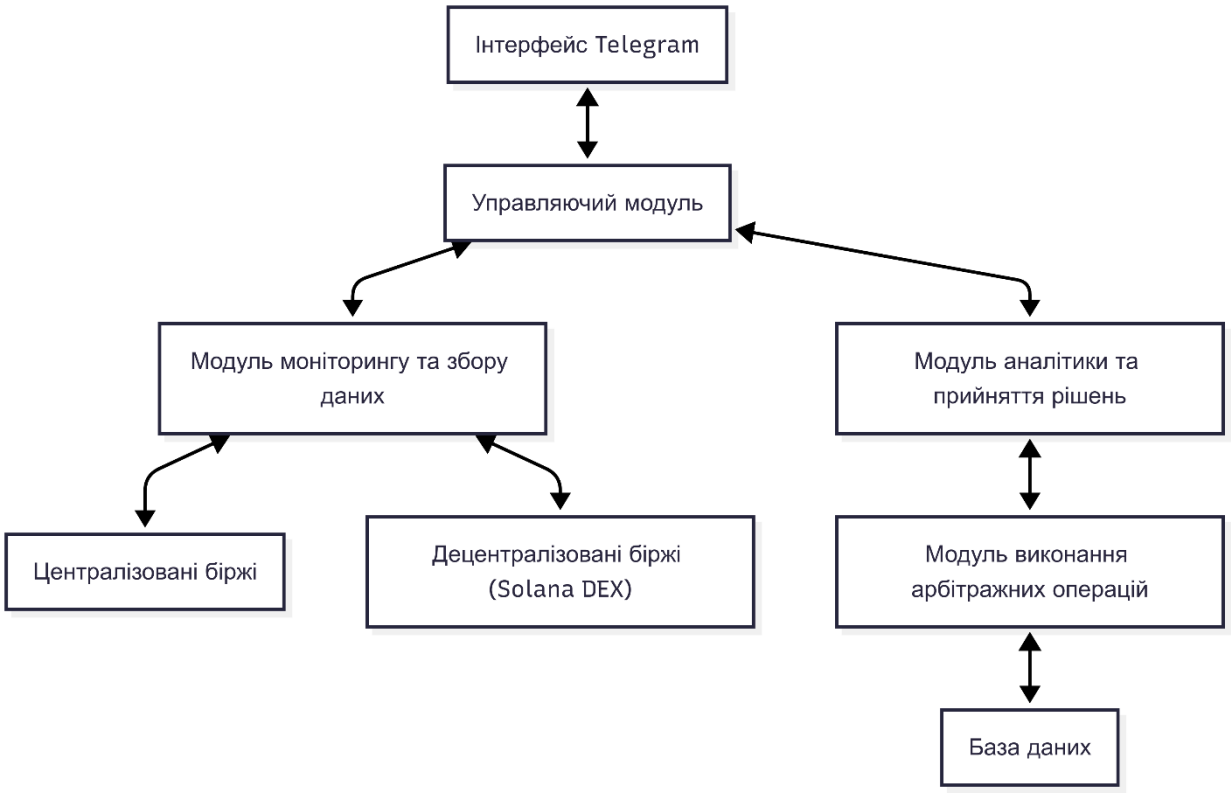

Рисунок 2.1 - Функціональна схема програмного комплексу

Розглянемо детально:

а) Інтерфейс Telegram - це те місце, де ти спілкуєшся з системою через Telegram-бота. Тут можна:
   1) Легко увійти і підтвердити свою особу;
   2) Подивитися, що зараз відбувається з арбітражем;
   3) Налаштувати, які пари токенів моніторити і як часто;
   4) Отримувати повідомлення про нові можливості і звіти;
   5) Запустити арбітраж вручну, якщо хочеш сам контролювати.

б) Управляючий модуль - це як «диригент», який керує всією системою. Він:
   1) Приймає твої команди з Telegram;
   2) Координує роботу всіх частин системи;
   3) Роздає завдання модулю моніторингу, аналітиці і виконавцям;
   4) Стежить, щоб все працювало гладко і без помилок;
   5) Дбає про безпеку - щоб нічого не зламалося і не було злому.

в) Модуль моніторингу та збору даних - це очі і вуха системи. Він:

1) Підключається до різних бірж, щоб отримувати свіжі дані;
2) Спілкується з блокчейном Solana, щоб відстежувати DEX;
3) Збирає інформацію про ціни, обсяги торгів і глибину ринку;
4) Перевіряє стан балансів, щоб знати, скільки можна торгувати;
5) Очищає і готує дані, щоб далі їх аналізувати.

г) Модуль аналітики та прийняття рішень - це «мозок», який аналізує всі дані і шукає, де можна вигідно заробити:
1) Шукає різницю в цінах між біржами;
2) Рахує, скільки реально можна заробити з урахуванням комісій;
3) Оцінює ризики - наприклад, різкі зміни курсу або затримки в мережі;
4) Будує розумні стратегії для арбітражу;
5) Прогнозує, як рухатиметься ринок.

д) Модуль виконання арбітражних операцій - це «руки», які роблять угоди:
1) Виконує покупки і продажі на централізованих біржах;
2) Працює зі смарт-контрактами Solana для DEX;
3) Контролює, щоб транзакції пройшли успішно;
4) Шукає найшвидший і найвигідніший спосіб виконання;
5) Швидко реагує на помилки або збої.

е) База даних - це «пам’ять» всієї системи, де зберігається все важливе:
1) Історія цін і курсів;
2) Записи про всі арбітражні угоди;
3) Твої налаштування і вибори;
4) Статистика заробітку;
5) Логи роботи системи, щоб відслідковувати, що і коли сталося.

2.1.2 Взаємодія зі смарт-контрактами Solana

Особлива крутість моєї системи - це те, що вона працює зі смарт-контрактами на Solana. Я зробив окремий модуль, який відповідає за те, щоб

швидко і безпечно виконувати арбітражні угоди на децентралізованих біржах. Для цього використовую смарт-контракт, написаний на фреймворку Anchor - він робить всю цю роботу простою і надійною.

Взаємодія зі смарт-контрактами Solana схематично представлена на рисунку 2.2.

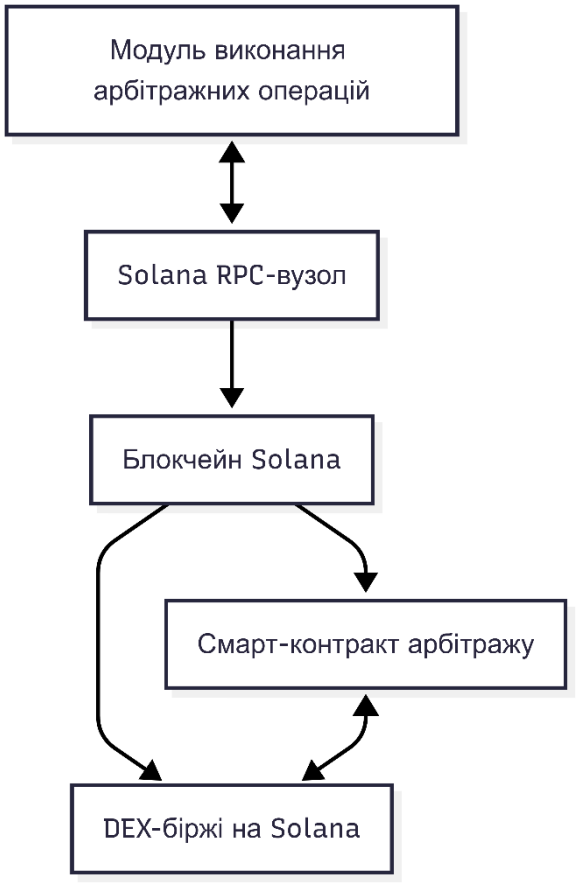

Рисунок 2.2 - Схема взаємодії зі смарт-контрактами Solana

Як працює взаємодія зі смарт-контрактами:

а) Модуль, що виконує арбітраж, спілкується з блокчейном Solana через спеціальний RPC-вузол.

б) Сам смарт-контракт, написаний на Rust з фреймворком Anchor, розміщений прямо в блокчейні Solana.

в) Контракт швидко і безпечно взаємодіє з різними децентралізованими біржами (DEX), щоб проводити атомарні (цілком неподільні) транзакції.

г) Всі операції записуються у блокчейн, що гарантує прозорість і

надійність усіх дій.

Що дають смарт-контракти в цій системі?

- Всі транзакції проходять як одна цілісна операція - або всі дії відбуваються, або ж жодна.
- Швидке виконання мінімізує ризик слипажу (коли ціна змінюється під час операції).
- Ефективність Solana дозволяє економити на комісіях (газі).
- Операції проходять безпечно і надійно.
- І головне - весь процес децентралізований, без посередників і ризику зловживань.

2.1.3 Потоки даних у системі

Для повного розуміння функціонування системи важливо описати основні потоки даних між компонентами.

На рисунку 2.3 представлена діаграма потоків даних.

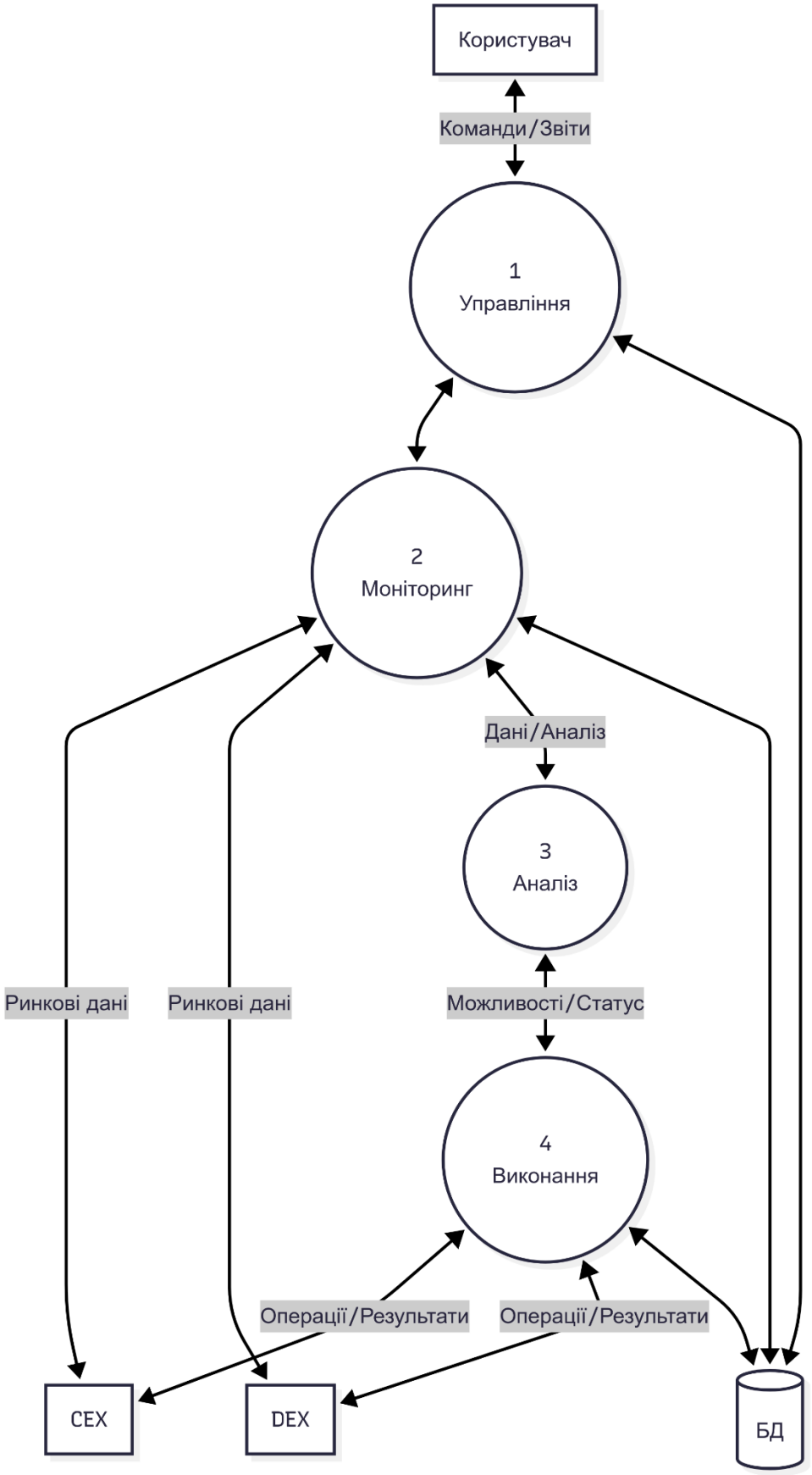

Рисунок 2.3 - Діаграма потоків даних

Зовнішні сутності:

а) Користувач - це людина, яка керує всією системою. Вона може запускати або зупиняти перевірки, дивитися, як ідуть справи, і отримувати сповіщення. Все просто - через Telegram-бота.

б) CEX - звичайні централізовані криптобіржі, такі як Binance чи KuCoin. Тут торги проходять "по-старому": через компанії, які все контролюють.

в) DEX - децентралізовані біржі в мережі Solana. Тут немає посередників - угоди виконуються напряму через смарт-контракти. Це сучасніше, швидше і дешевше.

Процеси:

а) Процес 1 (Управління) - саме серце системи. Слухає, що скаже користувач, і роздає команди всім іншим частинам. Без нього нічого не запуститься.

б) Процес 2 (Моніторинг) - слідкує за ринком: збирає дані про ціни, обсяги, курси, все, що важливо для арбітражу. Робить це постійно і уважно.

в) Процес 3 (Аналіз) - бере свіжі дані і починає думати: «Чи є тут можливість заробити?». Він вміє рахувати прибуток з урахуванням комісій, оцінювати ризики і пропонувати найкращі варіанти.

г) Процес 4 (Виконання) - коли з’являється можливість для арбітражу, цей модуль одразу її реалізовує. Працює швидко, точно і з урахуванням усіх нюансів: комісій, швидкості, статусу транзакцій.

д) БД - це така пам’ять системи, де зберігається все: твої налаштування, історія угод і вся статистика.

Як все передається:

– Ти через Telegram даєш команду - вона йде в Управління, яке розподіляє її далі.

– Біржі надсилають свіжу інформацію - її збирає Моніторинг.

– Цю інформацію Моніторинг передає в Аналіз, щоб подивитися, чи є можливості для вигідних угод.
– Якщо є, Аналіз каже Виконанню: "Почнемо торгувати!"
– Виконання робить угоди і звітує, як все пройшло.
– І всі процеси постійно читають і записують дані в БД, щоб нічого не загубити і все було під контролем.

2.2 Розробка та докладний опис алгоритму роботи програми

Уся ідея цього комплексу - зробити так, щоб арбітраж працював майже сам по собі: без зайвого втручання, швидко, безпечно і з розумінням, що відбувається на ринку. Щоб досягти цього, система проходить кілька логічних етапів - від моменту, коли з’являється можливість заробити, до її реалізації. У кожного етапу - своя роль і «характер», і разом вони утворюють цілісний, надійний процес. Далі я розповім, як саме все працює зсередини - простими словами.

2.2.1 Загальний алгоритм роботи програми

Загальний алгоритм роботи програми представлений на рисунку 2.4.

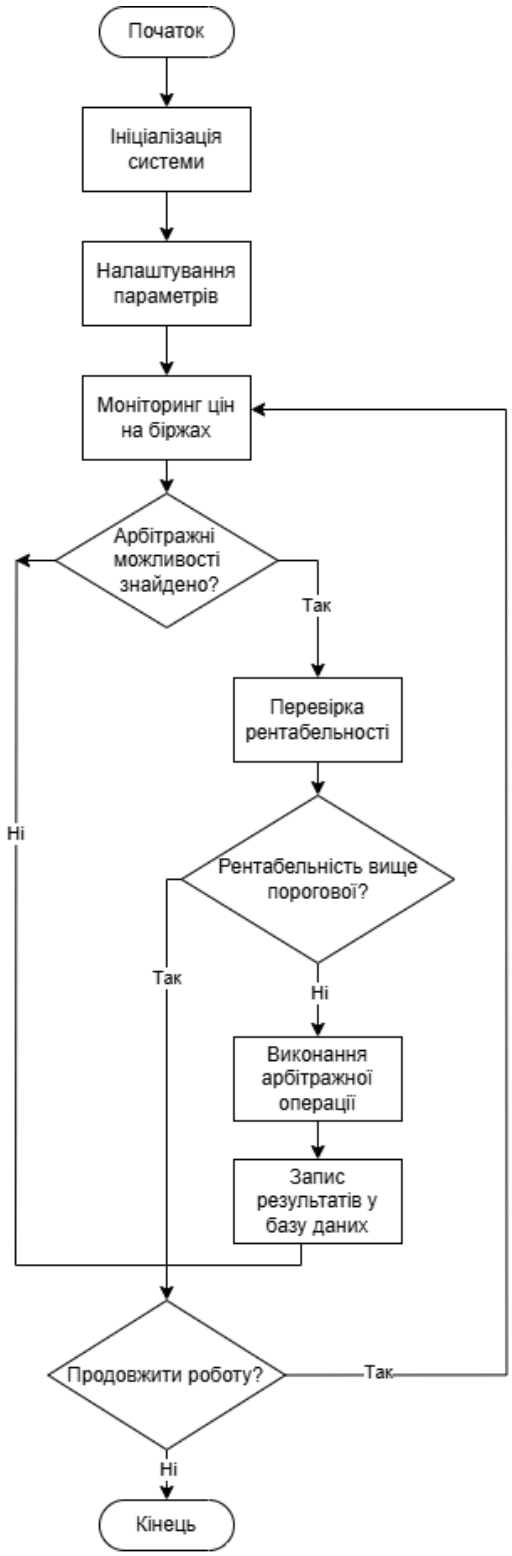

Рисунок 2.4 - Загальний алгоритм роботи програми

Давайте розглянемо, як саме працює система - крок за кроком:

а) Ініціалізація системи:

На цьому етапі все запускається та перевіряється.

1) Завантажуються всі налаштування, які були збережені раніше.
2) Встановлюється з’єднання з базою даних - щоб мати доступ до історії та конфігурацій.
3) Підключаємося до API бірж, з яких система братиме дані.
4) Telegram-бот виходить на зв'язок і стає доступним для користувача.
5) Перевіряється, чи все працює - якщо щось не так, одразу сигналізує.

б) Налаштування параметрів:

Тут система отримує інструкції, як саме працювати.

1) Вказуються конкретні пари криптовалют, за якими потрібно стежити.
2) Задається мінімальний рівень прибутковості, нижче якого операції не виконуватимуться.
3) Встановлюється обсяг угод - скільки купувати і продавати.
4) Визначаються обмеження ризиків - щоб уникнути небажаних втрат.
5) Обирається, як часто система має перевіряти біржі на нові можливості.

в) Моніторинг цін на біржах:

Цей етап - очі системи.

1) Одночасно опитуються всі біржі, до яких є доступ.
2) Збирається актуальна інформація про ціни.
3) Синхронізуються часові мітки - щоб не порівнювати "тепле з гарячим".
4) Відсіюються неактуальні або підозрілі дані.

г) Виявлення арбітражних можливостей:

Тут система "мислить", аналізує ситуацію.

1) Шукає різницю в цінах між біржами.
2) Враховує об’єм, який реально можна купити чи продати, не заваливши ринок.
3) Сортує варіанти за прибутковістю - щоб обрати найвигідніший.

д) Перевірка рентабельності:

Перш ніж діяти - треба все ще раз перевірити.

1) Розраховується очікуваний прибуток з урахуванням комісій.
2) Оцінюються ризики - чи немає великої волатильності, затримок чи слипажу.
3) Якщо все влаштовує - рухаємось далі, якщо ні - чекаємо кращої нагоди.

е) Виконання арбітражної операції:

Стадія дії - саме тут система заробляє.

1) Перевіряє, чи є необхідні кошти на біржах.
2) Проводить купівлю на одній біржі і продаж на іншій.
3) Слідкує, чи все пройшло успішно.
4) Якщо щось пішло не так - намагається це виправити або повідомляє про проблему.

ж) Запис результатів у базу даних:

Щоб усе було задокументовано.

1) Зберігаються всі деталі виконаних операцій.
2) Оновлюється загальна статистика по прибутках і ефективності.
3) Логуються технічні події для подальшого аналізу.

з) Перевірка умов продовження роботи:

Після всього - оцінка ситуації: продовжуємо чи чекаємо.

1) Система перевіряє свій стан: чи все працює як треба.
2) Читає нові команди від користувача через Telegram.
3) Оцінює загальні умови - наприклад, чи настав час зупинитись через високу волатильність або ніч.

2.2.2 Алгоритм виявлення арбітражних можливостей

Ключовою частиною програмного комплексу є алгоритм виявлення арбітражних можливостей, який представлено на рисунку 2.5.

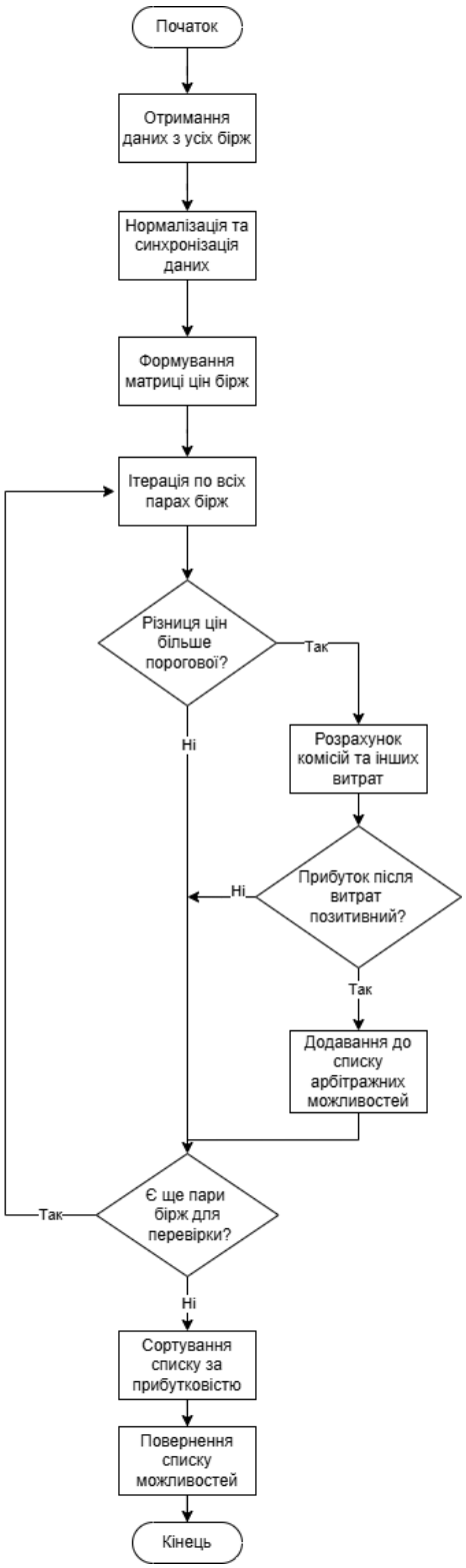

Рисунок 2.5 - Алгоритм виявлення арбітражних можливостей

Детальний опис етапів алгоритму:

а) Отримання даних з усіх бірж:

На цьому етапі система «слухає» ринок.

1) Одночасно звертається до API централізованих бірж, щоб швидко зібрати всі потрібні котирування.
2) Для DEX-бірж, що працюють у мережі Solana, використовуються RPC-запити - це дозволяє отримати дані безпосередньо з блокчейна.
3) Збирається максимум актуальної інформації: ціни, обсяги торгів і глибина ринку.

б) Нормалізація та синхронізація даних:

Зібрані дані з різних джерел можуть виглядати по-різному - це треба виправити.

1) Усі показники приводяться до єдиного формату, щоб їх можна було порівнювати.
2) Дані синхронізуються за часом - важливо знати, що всі ціни актуальні на одну й ту ж мить.
3) Аномальні або підозрілі значення фільтруються, щоб не заважали прийняттю рішень.

в) Формування матриці цін бірж:

Тут усе зібране перетворюється на зручну таблицю для аналізу.

1) Створюється матриця - своєрідна мапа цін для всіх криптопар на всіх біржах.
2) Ураховується, що ціни на купівлю й продаж відрізняються.
3) Усе структуровано так, щоб потім швидко знаходити потенційно вигідні відмінності.

г) Ітерація по всіх парах бірж:

Цей крок - як уважний перегляд усіх можливих маршрутів обміну.

1) Система послідовно перебирає всі варіанти комбінацій між біржами.
2) Розглядає різні варіанти обміну активів, включаючи нестандартні ланцюжки.
3) Враховує, що кожна біржа має свої нюанси - наприклад, швидкість чи обмеження.

д) Перевірка різниці цін:

Ключовий момент - а чи є взагалі можливість заробити?

1) Система обчислює різницю в цінах на одні й ті ж активи між біржами.
2) Ці значення порівнюються з заданим порогом - тобто мінімальним рівнем, який вартий уваги.
3) Також перевіряється, чи дозволяє глибина ринку виконати операцію без втрат.

е) Розрахунок комісій та інших витрат:

Бо прибуток - це не лише про різницю в цінах.

1) Ураховуються всі торгові комісії на біржах.
2) Також додаються витрати на транзакції в мережі Solana (gas).
3) Система прогнозує можливий слипаж - тобто те, наскільки зміниться ціна під час виконання операції.

ж) Перевірка прибутковості:

Після врахування всіх витрат - чи залишиться вигода?

1) Розраховується чистий прибуток з урахуванням усіх витрат.
2) Порівнюється з мінімальним значенням, яке користувач вважає прийнятним.
3) Якщо все гаразд - далі оцінюється співвідношення ризику до потенційного доходу.

з) Додавання до списку арбітражних можливостей:

Якщо можливість виявилась вигідною - її треба запам’ятати.

1) Формується окрема структура з усіма важливими параметрами: ціна, біржі, обсяг тощо.
2) Ця інформація зберігається для подальшого використання.
3) Арбітражна можливість додається до загального списку кандидатів на виконання.

и) Сортування списку за прибутковістю:

Коли є кілька варіантів - обираємо найкращі.

1) Усі знайдені можливості ранжуються за очікуваним прибутком.
2) При сортуванні також враховуються ризики, щоб уникнути небезпечних операцій.
3) У підсумку система формує список з найбільш перспективними і безпечними варіантами.

2.2.3 Алгоритм виконання арбітражної операції

Після виявлення арбітражної можливості система переходить до виконання операції. Алгоритм виконання арбітражної операції представлено на рисунку 2.6.


Початок
Отримання параметрів арбітражної операції
Перевірка балансів на біржах
Баланси достатні?
Так
Ні
Перевірка актуальних цін на біржах
Ціни все ще дають арбітражну можливість?
Ні
Так
Запис результатів в базу даних
Визначення типу арбітражної операції
Виконання операції
1

Рисунок 2.6 - Алгоритм виконання арбітражної операції

Розглянемо етапи алгоритму виконання арбітражної операції:

а) Отримання параметрів арбітражної операції:

1) Зчитування даних про біржі, між якими здійснюється арбітраж

2) Визначення обсягу операції
3) Отримання актуальних курсів обміну

б) Перевірка балансів на біржах:
1) Запит поточних балансів на всіх задіяних біржах
2) Перевірка достатності коштів для виконання операції
3) Врахування мінімальних обсягів торгів на біржах

в) Перевірка актуальності цін:
1) Повторний запит цін для забезпечення актуальності
2) Порівняння з раніше отриманими даними
3) Врахування можливих змін на ринку

г) Визначення типу арбітражної операції:
1) CEX-CEX: операції між централізованими біржами
2) DEX-DEX: операції між децентралізованими біржами Solana
3) CEX-DEX: гібридні операції

д) Виконання операції CEX-CEX:
1) Послідовне виконання ордерів на покупку та продаж
2) Моніторинг статусу виконання ордерів
3) Обробка помилок та відмов

е) Виконання операції DEX-DEX:
1) Виклик смарт-контракту для атомарного арбітражу
2) Передача необхідних параметрів (біржі, пари, обсяги)
3) Очікування підтвердження транзакції

ж) Моніторинг виконання:
1) Відстеження статусу транзакцій в реальному часі
2) Аналіз проміжних результатів
3) Готовність до скасування операції у випадку критичних змін

з) Розрахунок фактичного прибутку:
1) Отримання фактичних даних про виконані операції
2) Обчислення реального прибутку з урахуванням фактичних

комісій та слипажу

3) Порівняння з прогнозованим прибутком

и) Запис результатів в базу даних:

1) Збереження всіх деталей операції
2) Оновлення статистики прибутковості
3) Збереження технічних логів для подальшого аналізу

к) Відправка сповіщення через Telegram:

1) Формування звіту про виконану операцію
2) Надсилання повідомлення користувачу
3) Включення ключових метрик та результатів

2.2.4 Алгоритм взаємодії зі смарт-контрактами Solana

Особливої уваги заслуговує алгоритм взаємодії зі смарт-контрактами Solana, який є ключовим елементом системи для виконання арбітражу на децентралізованих біржах. Алгоритм представлено на рисунку 2.7.

Рисунок 2.7 - Алгоритм взаємодії зі смарт-контрактами Solana

Як ми працюємо зі смарт-контрактами на Solana, крок за кроком:

а) Ініціалізація клієнта Solana:

Спершу підключаємось до мережі Solana - це як зайти у потрібну кімнату:

1) З’єднуємось із сервером, що дає нам доступ до блокчейну.
2) Готуємо свій акаунт, щоб мати право діяти в цій мережі.
3) Переконуємось, що мережа відповідає і все працює.

б) Підготовка параметрів для контракту:

Далі збираємо всі «інгредієнти» для запуску смарт-контракту:

1. Складаємо чіткі інструкції, що потрібно зробити.
2. Вказуємо адреси програми, токенів та акаунтів - все, що потрібно для роботи.
3. Підготували дані, які треба передати в контракт.

в) Розрахунок газу (комісії):

Тут дивимось, скільки мережа попросить «плати» за операцію:

1) Оцінюємо, наскільки складна задача.
2) Рахуємо приблизну комісію (газ).
3) Перевіряємо, чи варто взагалі робити цю операцію з економічної точки зору.

г) Підпис транзакції:

Тепер ми «підписуємо» операцію своїм особистим ключем - це як поставити підпис під документом:

1) Формуємо транзакцію.
2) Підписуємо її, щоб підтвердити, що це ми.
3) Переконуємось, що все безпечно.

д) Відправка транзакції:

Відправляємо нашу транзакцію на сервер:

1) Надсилаємо підписану транзакцію у мережу.
2) Отримуємо унікальний номер, щоб відслідковувати цю операцію.

3) Починаємо стежити, чи все пройшло успішно.

е) Перевірка підтвердження:

Трохи терпіння - перевіряємо, чи підтвердила мережа нашу транзакцію:

1) Періодично питаємо статус.
2) Чекаємо, поки мережа скаже, що все ок.
3) Якщо щось пішло не так - обробляємо помилки.

ж) Аналіз результатів:

Дивимось, що вийшло після виконання:

1) Читаємо, як спрацював контракт.
2) Перевіряємо логи, щоб упевнитися, що все гаразд.
3) Знаходимо і розбираємось з помилками, якщо є.

з) Оновлення балансів:

Після цього оновлюємо наші дані про кошти:

1) Запитуємо актуальні баланси.
2) Оновлюємо їх у нашій системі.
3) І готуємось до наступних операцій.

2.2.5 Алгоритм управління через Telegram

Важливою частиною системи є модуль управління через Telegram, який забезпечує зручний інтерфейс для користувача. Алгоритм роботи цього модуля представлено на рисунку 2.8.

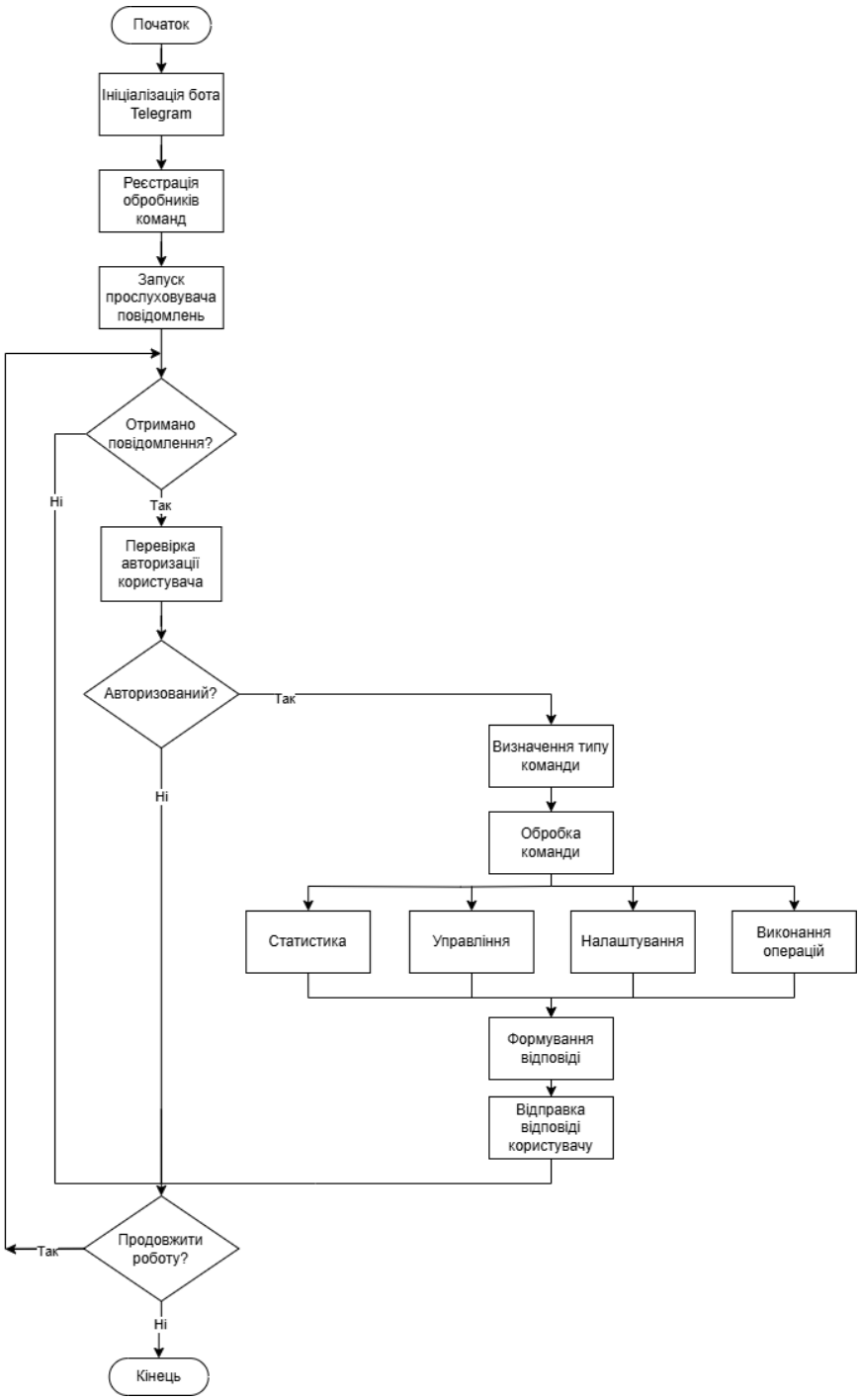

Рисунок 2.8 - Алгоритм управління через Telegram

Детальний опис етапів управління бота Telegram:

а) Ініціалізація бота Telegram:

Спочатку запускаємо нашого бота:

1) Беремо токен від Telegram - це як ключик, щоб бот міг заходити в чат.

2) Вмикаємо бібліотеку, яка допомагає боту спілкуватися з Telegram і розуміти, що до нього пишуть.
3) Налаштовуємо базові речі, щоб бот працював і не загубився.

б) Реєстрація обробників команд:

Тепер розповідаємо боту, які команди він може зрозуміти і що робити, коли їх отримує:

1) Складаємо список команд, які бот “слухає”.
2) До кожної команди прив’язуємо спеціальну функцію, яка “виконує роботу”.
3) Додаємо підказки, щоб користувачам було просто зрозуміти, як із ботом спілкуватися.

в) Запуск прослуховувача повідомлень:

Бот починає слухати, що надходить:

1) Вмикаємо механізм отримання нових повідомлень від користувачів.
2) Встановлюємо таймаути та інші параметри зв’язку для стабільної роботи.
3) Запускаємо цей процес у фоні, щоб бот не зависав.

г) Перевірка авторизації користувача:

Перед тим як виконувати команди, перевіряємо, чи має користувач право:

1) Розпізнаємо користувача за його Telegram ID.
2) Перевіряємо, чи дозволено йому користуватися ботом.
3) Захищаємо систему від сторонніх та несанкціонованих дій.

д) Визначення типу команди:

Коли приходить повідомлення, бот розуміє, що саме треба зробити:

1) Розбираємо текст повідомлення.
2) Визначаємо, яка команда або запит у ньому міститься.
3) Направляємо його до потрібного обробника.

е) Обробка команд різних типів:

Бот виконує різні дії залежно від типу команди:

1) Статистика - показує інформацію про виконані операції, прибуток і баланси.
2) Управління - дає змогу включати або вимикати арбітраж, налаштовувати стратегії.
3) Налаштування - дозволяє змінювати параметри роботи системи.
4) Виконання операцій - запускає арбітраж вручну, якщо потрібно.

ж) Формування відповіді:

Бот готує зрозумілу і корисну відповідь:

1) Складає інформативне повідомлення.
2) Форматує дані так, щоб їх було зручно читати.
3) Додає інтерактивні елементи - кнопки та інлайн-клавіатуру.

з) Відправка відповіді користувачу:

І наостанок - відправляємо відповідь:

1) Надсилаємо повідомлення через Telegram API.
2) Обробляємо можливі помилки при відправці.
3) Логування взаємодії - зберігаємо історію спілкування.

2.3 Розробка інтерфейсу програми

Інтерфейс програмного комплексу автоматизації управління арбітражем криптовалютних активів реалізовано на базі Telegram-бота, що забезпечує зручний доступ до управління системою з будь-якого пристрою, де встановлено месенджер Telegram. Такий підхід надає ряд переваг: мобільність, доступність, надійність доставки повідомлень, можливість використання стандартних елементів інтерфейсу Telegram.

2.3.1 Структура інтерфейсу

Структура інтерфейсу програми представлена на рисунку 2.9.

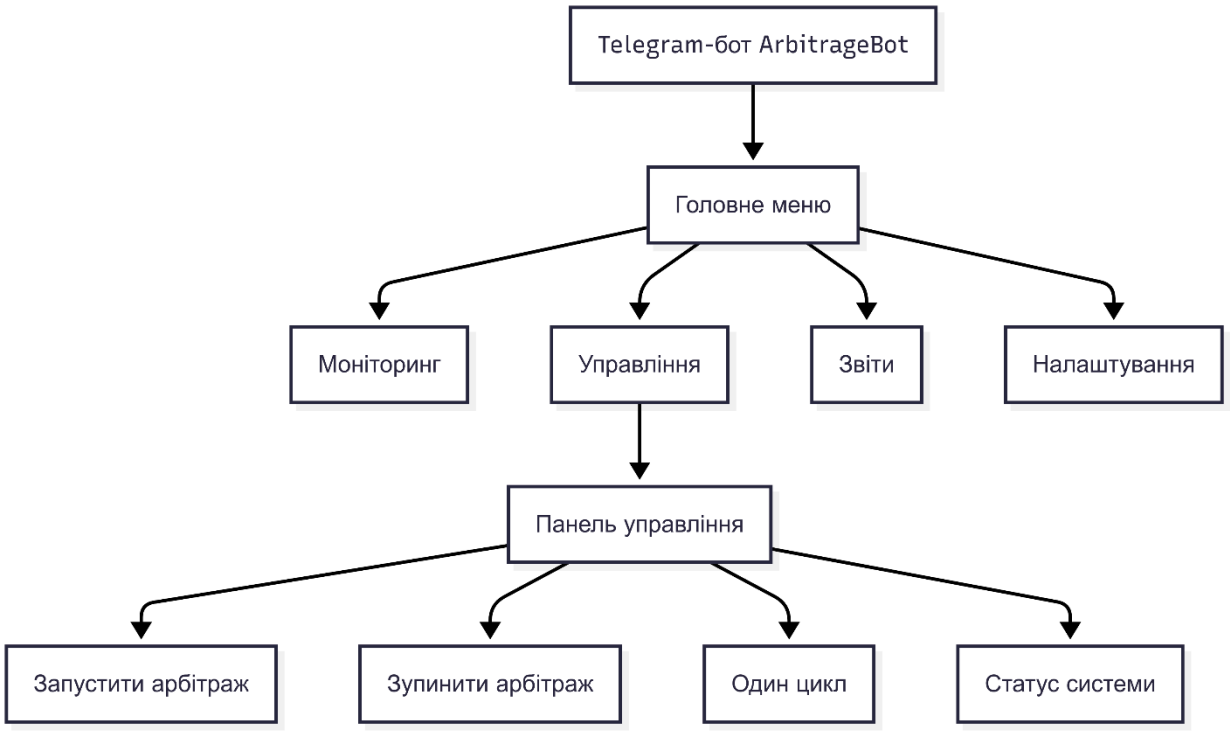

Рисунок 2.9 - Структура інтерфейсу програми

Інтерфейс програми складається з наступних основних елементів:

а) Головне меню - відправна точка взаємодії з ботом, що містить основні розділи:
   1) Моніторинг - перегляд поточного стану ринку та арбітражних можливостей
   2) Управління - керування роботою системи арбітражу
   3) Звіти - перегляд статистики та результатів роботи
   4) Налаштування - конфігурація параметрів системи

б) Панель управління - розділ для безпосереднього керування арбітражними операціями:
   1) Запустити арбітраж - активація автоматичного режиму арбітражу
   2) Зупинити арбітраж - деактивація автоматичного режиму
   3) Один цикл - виконання одного циклу пошуку та відпрацювання арбітражних можливостей

4) Статус системи - перегляд поточного стану системи та підключень до бірж

2.3.2 Екрани та взаємодія

Розглянемо основні екрани інтерфейсу та принципи взаємодії користувача з системою.

Екран моніторингу

Цей екран надає інформацію про поточний стан ринку та виявлені арбітражні можливості. Дані оновлюються в режимі реального часу за запитом користувача. Екран ручного моніторингу представлено на рисунку 2.10.

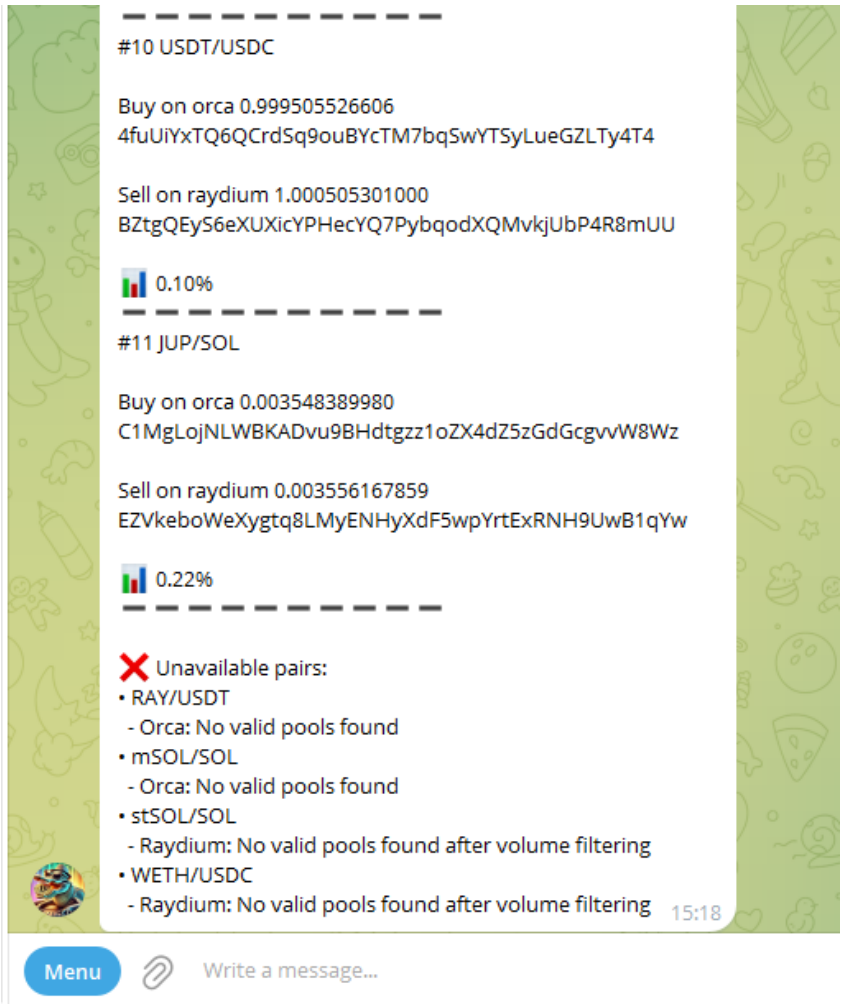

Рисунок 2.10 - Екран ручного моніторингу

Також пошук арбітражних можлиовстей відбувається автоматично з

певним інтервалом і відображаються тільки знайдені можливасті, Наприклад де прибуток більше 1%. Детальніше на рисунку 2.11.

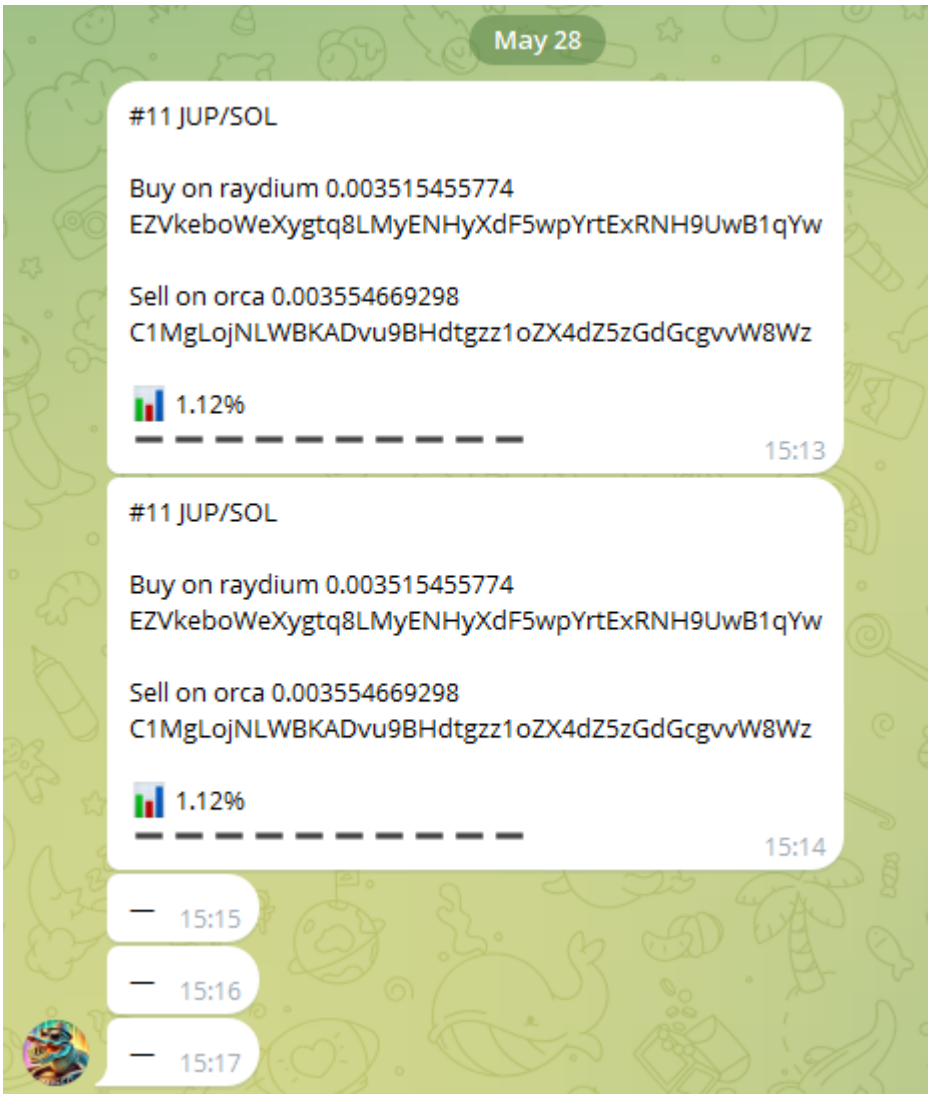

Рисунок 2.11 - Екран автоматичного моніторингу

Екран управління

Цей екран дозволяє керувати роботою системи арбітражу, запускати та зупиняти автоматичний режим, а також виконувати окремі операції вручну. Екран управління представлено на рисунку 2.12.

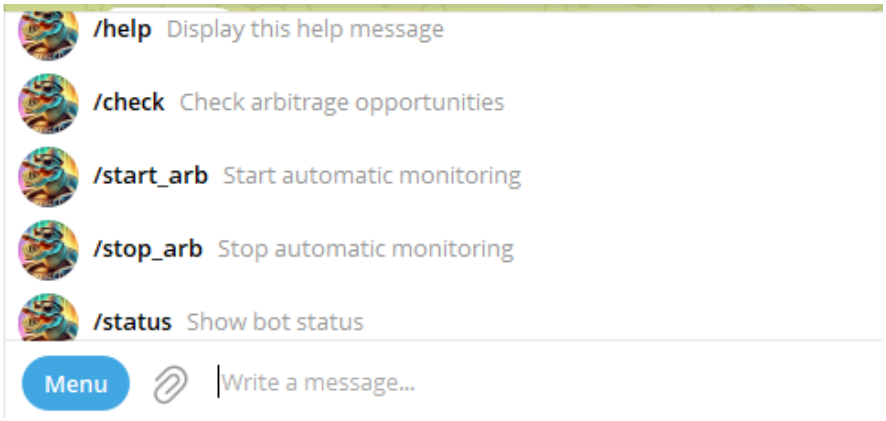

Рисунок 2.12 - Екран управління

2.3.3 Командний інтерфейс

Крім зручних кнопок у Telegram, які допомагають швидко обирати потрібні дії, наша система також підтримує текстові команди. Це як “короткі шляхи” - просто вводиш команду, і все працює миттєво. Основні команди можна знайти у таблиці 2.1.

Таблиця 2.1 - Основні команди системи

Команда	**Опис**	**Приклад**
/status	Перегляд поточного статусу системи	/status
/check	Перегляд поточних арбітражних можливостей	/check
/start_arb	Запуск автоматичного режиму арбітражу	/start_arb
/stop_arb	Зупинка автоматичного режиму арбітражу	/stop_arb
/help	Отримання довідкової інформації	/help

Такий спосіб особливо зручний для тих, хто вже трохи знається на системі або хоче швидко отримати доступ до важливих функцій без зайвих рухів.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Аналіз обраного середовища та технологій програмування

Щоб реалізувати систему для автоматизації арбітражу криптовалют, я вирішив зібрати стек інструментів, який буде одночасно швидким, безпечним і зручним у роботі з блокчейном. Було важливо, щоб технології мали активну спільноту, потрібні бібліотеки, і добре себе показували у роботі з низькорівневими API та Telegram.

Основною мовою я обрав Rust - і це не просто через його “хайповість”. Rust справді крутий: він швидкий, надійний, і завдяки строгій системі типів допомагає уникати багів ще до запуску програми. Для роботи з фінансами та смартконтрактами це просто must-have. А ще Rust - рідна мова для блокчейну Solana, тож усе логічно зійшлося.

Самі смартконтракти я пишу за допомогою Anchor - це фреймворк, який дуже спрощує життя. Він дозволяє не забивати голову рутинними речами на кшталт серіалізації даних, має приємний декларативний підхід і добре інтегрується з інструментами Solana. З ним легше тестувати, дебажити і взагалі - не зійти з розуму під час розробки.

Щоб керувати всією системою, я зробив Telegram-бота. Для цього використовую Telexoid - це легка, асинхронна бібліотека на Rust, яка дозволяє швидко підняти бота, налаштувати команди та зручно працювати з Telegram API. Через нього надсилаються сповіщення про вигідні арбітражні можливості, і взагалі вся взаємодія з користувачем йде саме через нього.

Працюю у VS Code з купою корисних плагінів (rust-analyzer, Crates, CodeLLDB) - це дуже зручно. Підсвічування, підказки, інтеграція з Git - усе працює, як треба.

Для тестів використовую локальний Solana validator. Це дозволяє спокійно тестувати контракти без ризику й витрат - у своєму “пісочному” середовищі, де я повністю контролюю, що відбувається.

Що стосується самих бірж, то зараз працюю з Orca та Raydium - вони надають зручні API, з якими легко працювати через reqwest, serde, tokio - стандартний стек для асинхронної роботи в Rust. Дані приходять швидко, обробка - теж миттєва, тож затримок практично нема.

Загалом, я задоволений вибраним стеком - усе добре працює разом, дозволяє швидко розробляти і при цьому залишатися впевненим у стабільності та безпеці системи. І головне - зручно масштабуватись далі.

3.2 Реалізація Telegram-бота

Один з найважливіших компонентів усієї системи - це Telegram-бот, який фактично є головним інтерфейсом для управління арбітражем. Керувати всіма процесами через Telegram дуже зручно: не потрібно піднімати окремі панелі керування, вебінтерфейси чи працювати з терміналом - усе в кілька кліків прямо зі смартфона. Платформа стабільна, API добре задокументоване, а взаємодія з користувачем відбувається майже миттєво - ідеально для системи, яка працює в реальному часі.

Бот надає повний контроль над тим, що відбувається: можна запускати або зупиняти моніторинг, переглядати наявні арбітражні можливості, слідкувати за статусом і тестувати логіку прямо “на ходу” - без зайвих рухів.

Для реалізації бота я використовую Telexoid - це асинхронний Telegram-фреймворк на Rust. Він має просту й сучасну архітектуру, яка базується на Dispatcher'ax, Handler'ax і типобезпечних командах. Такий підхід суттєво спрощує обробку команд, реакцію на повідомлення й інші дії користувача.

На першому етапі створюється сам бот - токен беремо у BotFather, ініціалізуємо об’єкт бота, а далі йде налаштування команд. Серед них:

- запуск моніторингу пар;
- перегляд актуальних арбітражних можливостей;
- зупинка моніторингу;

- перевірка статусу бота.

Усе це реалізовано через обробники, кожен з яких відповідає за конкретну дію користувача. **(Див. додаток А, пункт 1.1 - код команд Telegram-бота)**

Асинхронність роботи забезпечує Tokio - це основний рушій паралельного виконання в екосистемі Rust. Він дозволяє обробляти вхідні події (команди, повідомлення тощо) без затримок, навіть коли запитів надходить багато одночасно.

Я спеціально поділив логіку на окремі модулі чи функції: кожна команда - окремий блок. Це не лише спрощує тестування, а й дає змогу легко розширювати функціональність. Обробники включають валідацію вхідних параметрів, логування запитів і виняткових ситуацій - щоб система була максимально стабільною. **(Див. додаток А, пункт 1.2 - код обробника команд Telegram)**

Окрім звичайних команд, бот вміє самостійно надсилати повідомлення про арбітражні можливості. Це зроблено за допомогою фонових задач, які регулярно перевіряють обрані торгові пари, і якщо з’являється цікава можливість - бот відразу сповіщає користувача. **(Див. додаток А, пункт 1.3 - код запуску автоматичного моніторингу)**

Особливу увагу я приділив обробці помилок. Усе, що йде через зовнішні API (наприклад, біржі), може бути нестабільним - тому всі помилки логуються, і бот повідомляє, якщо якась біржа “впала” або виникла критична проблема. Це допомагає швидко реагувати й не залишити систему “наосліп”.

Загалом Telegram-бот вийшов простим для користувача, але функціонально дуже потужним. Через нього можна повністю керувати системою, отримувати сповіщення, тестувати контракти й тримати руку на пульсі арбітражу - хоч вдома за комп’ютером, хоч дорогою через телефон.

3.3 Реалізація моніторингу арбітражних можливостей

Однією з найважливіших частин у всій системі став модуль, який постійно стежить за курсами криптовалют на різних біржах. Його суть дуже проста: він шукає, де одну й ту саму криптовалютну пару можна купити дешевше, а де - продати дорожче. Тобто шукає арбітражні можливості в реальному часі.

Цей моніторинг працює у фоновому режимі, нічому не заважає, і постійно надсилає свіжі дані в Telegram-бот та модуль, який вирішує, що робити далі. Користувач сам обирає, за якими токенами стежити, як часто перевіряти й яку мінімальну вигоду враховувати. Платформа підтримує гнучке налаштування й уже вміє працювати з біржами Raydium та Orca - і легко розширюється під інші.

3.3.1 Конфігурація токенів і пар

Для ефективної роботи системи необхідно мати можливість визначати, які токени та торгові пари підлягають моніторингу. Ця інформація зберігається у вигляді конфігураційного файлу або структури в пам’яті, що дозволяє гнучко керувати списком активів, не змінюючи код основного застосунку.

Кожна пара описується наступними параметрами:

- адреса токенів (mint-адреси у мережі Solana);
- символьні ідентифікатори (наприклад, USDC/SOL);

(Див. додаток А, пункт 1.4 - код конфігурації пар токенів)

Це дозволяє системі зосереджуватись лише на перспективних напрямках і зменшити навантаження на API-біржі.

3.3.2 Абстрактний інтерфейс біржі

З метою забезпечення масштабованості та уніфікації логіки опитування, було реалізовано абстрактний інтерфейс біржі у вигляді trait. Цей інтерфейс описує базову поведінку, яку повинна реалізовувати будь-яка підтримувана

DEX-платформа

- отримання пулів з ціною купівлі та продажу для заданої пари;
- ідентифікація біржі;

(Див. додаток А, пункт 1.5 - код абстракного інтерфейсу біржі)

Такий підхід дозволяє розробляти нові драйвери для інших бірж без зміни основної логіки моніторингу. Кожна реалізація повинна лише імплементувати відповідні методи trait'у.

3.3.3 Реалізація для DEX Orca

Orca - це одна з популярних платформ на Solana, яка дуже зручна для роботи з ліквідністю. Там класний API, який дозволяє легко отримувати дані, плюс підтримує багато токенів. Правда, є й проблемні пули - їх доводиться блокувати, щоб не було помилок.

Я зробив запити до Orca з урахуванням оновленого API, щоб швидко отримувати свіжу інформацію про пули, ціни і ліквідність.

(Див. додаток А, пункт 1.6 - код моніторингу для DEX Orca)

Мій модуль для Orca працює так само, як і для Raydium - він використовує загальний інтерфейс і повертає всі дані в одному форматі, щоб потім було зручно з ними працювати. Результати моніторингу я зберігаю у внутрішньому кеші, далі інші частини системи беруть потрібну інформацію для арбітражу. Це допомагає рідко звертатись до API та економити ресурси.

Такий підхід дає змогу легко розширювати систему - можна швидко додати нові біржі або токени, не переписуючи весь код.

3.4 Реалізація смарт-контракту арбітражу

Одним із найголовніших елементів моєї системи став смарт-контракт, який самостійно виконує арбітражні операції між різними децентралізованими біржами. Він міняє токени на таких платформах, як Raydium і Orca, все це відбувається в одній транзакції. Але перед тим контракт перевіряє, чи буде вигідна ця операція - щоб не втратити гроші.

Я писав цей контракт на Rust, використовуючи фреймворк Anchor - він дуже допоміг зробити код чистим і зрозумілим, а також полегшив роботу з командами і перевіркою даних. Anchor ще генерує спеціальні файли, які можна підключати до інших частин системи - наприклад, до Telegram-бота.

Сам арбітраж тут працює трохи схоже на flash loan, але без позик і ризиків. Контракт просто використовує баланс користувача і виконує кілька обмінів підряд, перед тим перевіряючи, що все буде вигідно.

Основні етапи роботи смарт-контракту:

- прийом інструкції від клієнта з параметрами арбітражної операції;
- перевірка поточних цін на двох біржах через CPI-виклики (Cross-Program Invocation);
- виконання першого свопу на одній з бірж;
- передача результату на другу біржу для зворотного обміну;
- фінальна перевірка - чи збільшився баланс користувача після серії операцій;
- у разі успішного завершення - фіксація прибутку, інакше - відкат транзакції.

Контракт написаний із дотриманням принципів безпеки, таких як перевірка дозволених токенів, захист від переповнення, обмеження доступу до інструкцій лише авторизованими користувачами, а також забезпечення атомарності транзакцій.

(Див. додаток А, пункт 1.7 - код смарт-контракту арбітражу)

Для тестування контракту використовувався локальний Solana-валидатор (solana-test-validator) у зв’язці з емуляцією API бірж і мок-даними. Це дозволило перевірити повний життєвий цикл арбітражної операції без витрат реальних коштів або ризиків, пов’язаних з ончейн-розгортанням.

Запропонований смарт-контракт є універсальним і легко розширюваним - за потреби до нього можуть бути додані нові джерела ліквідності, адаптація під мультиарбітражні стратегії або навіть підтримка зовнішніх тригерів для

автозапуску арбітражу через oracle-сервіси або бота.

Таким чином, реалізація смарт-контракту забезпечує основу для виконання реальних арбітражних сценаріїв у рамках екосистеми Solana, що суттєво розширює функціональність створеного програмного комплексу.

3.5 Тестування моніторингу та арбітражу

У цьому розділі наведено результати тестування двох основних компонентів системи - модуля моніторингу арбітражних можливостей та смарт-контракту, який реалізує сам арбітраж. Тестування здійснювалось з метою перевірки коректності логіки роботи, надійності обміну повідомленнями з Telegram-ботом, та фактичної працездатності алгоритмів на локальному середовищі без залучення реальних активів.

3.5.1 Тестування модуля моніторингу

Моніторинг арбітражу - це, без перебільшення, одна з найважливіших частин системи. Без нього бот просто не знав би, коли і де з’являється вигідна можливість. Щоб усе працювало зручно, я передбачив два режими:

- Автоматичний - бот сам перевіряє задані пари токенів через проміжки часу які я вкажу, наприклад, кожні 30 секунд.
- Ручний - якщо хочеться все тримати під контролем, можна просто надіслати команду в Telegram, і бот миттєво зробить перевірку.

Під час тестування я просто хотів упевнитися, що все працює так, як задумував. І автоматичний режим, і ручний запуск через Telegram показали себе добре - усе запускалося вчасно, без збоїв. Особливо цікаво було подивитись, як система поводиться, коли щось іде не за планом - наприклад, якщо біржа тимчасово “відвалюється” або дані не приходять. В таких випадках усе теж працювало спокійно, без паніки. Я уважно перевіряв, чи правильно рахується прибуток, і чи зручно бот подає інформацію. У підсумку я залишився задоволений: вийшло щось, чим справді зручно користуватись і що не підведе у потрібний момент.

На скріншоті нижче приклад відповіді бота на надсилання команди ручного моніторингу:

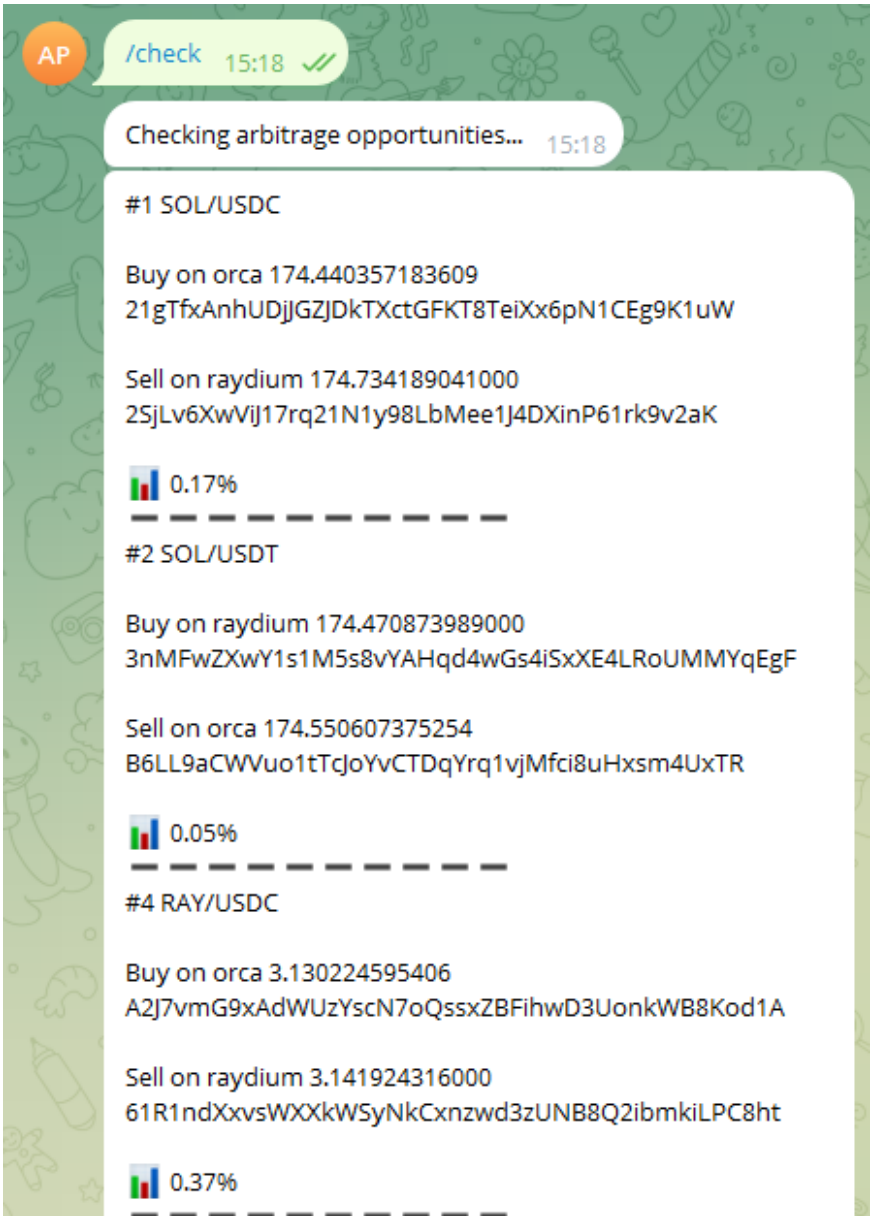

Рисунок 3.1 - Ручна перевірка арбітражних можливостей

Рисунок 3.2 - Автоматичне оповіщення про арбітражну можливість

Ці результати підтверджують, що бот правильно реагує на вхідні команди, надає детальну інформацію про поточні ціни на DEX, а також своєчасно повідомляє про виявлені арбітражні можливості з розрахунком очікуваного прибутку.

3.5.2 Тестування смарт-контракту арбітражу

Щоб спокійно перевірити, як працює мій смарт-контракт, я розгорнув локальне середовище за допомогою solana-test-validator. Це було дуже зручно: я міг тестувати все, не боячись втратити якісь гроші чи платити комісії. По суті, все виглядало майже як у справжній мережі, тільки без стресу. Такий підхід допоміг мені впевнитися, що контракт поводиться правильно і виконує все, як треба.

Контракт було протестовано на таких сценаріях:

- виконання арбітражу з прибутком;
- ситуація, коли арбітраж не вигідний;
- верифікація балансу користувача до та після операції;
- перевірка атомарності транзакції - в разі помилки або відсутності прибутку, обмін не відбувається.

Усі тести проводилися в межах одного локального блоку, що дозволило проаналізувати не лише логіку виконання, а й швидкодію системи.

Нижче наведено скріншот з результатами успішного тесту арбітражної

операції:

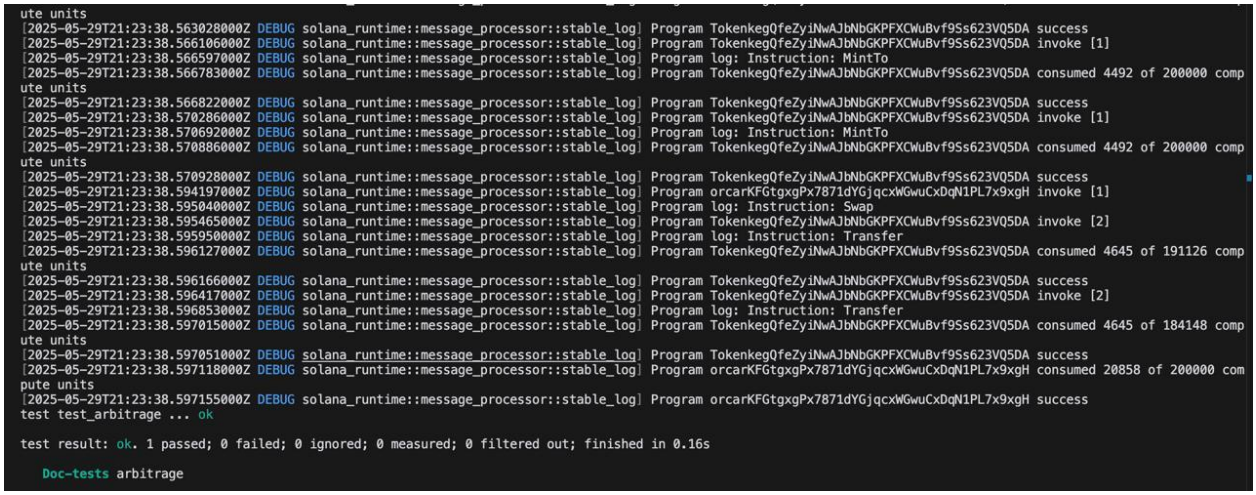

Рисунок 3.3 Результат локальних тестів смарт-контракту арбітражу

Після тестів я переконався - контракт справді робить усе, що треба. Він чітко виконує арбітражну логіку, працює без збоїв і не створює зайвого клопоту користувачу. Все працює спокійно, надійно і, головне, безпечно - як і планувалося.

ВИСНОВКИ

У своїй дипломній роботі я зробив програму, яка сама шукає, де можна вигідно заробити на різниці курсів між децентралізованими криптобіржами, і автоматично проводить ці операції. Суть у тому, щоб стежити за курсами різних токенів на DEX'ах (тобто біржах без посередників), і коли з'являється хороша можливість - програма це помічає і миттєво проводить обмін через смартконтракт. Завдяки цьому все відбувається швидко і безпечно, без ризику втратити гроші через затримки.

Я використовував мову Rust і блокчейн Solana - вони швидкі й надійні. А щоб користуватися всім цим було просто, я зробив Telegram-бота. Він показує, які зараз є можливості для арбітражу, або надсилає повідомлення автоматично, коли щось цікаве з’являється. Користувач сам вибирає, які токени його цікавлять, скільки він хоче заробляти мінімум, і як часто бот має все перевіряти. Реалізовано моніторинг який повідосляє про знаходження арбітражних можливостей через Telegram-бота. Для арбітражу програма використовує дві DEX біржи: Orca та Raydium. Також реалізований смарт-контракт для арбітражу. Туди передається інформація про знайдену арбітражну можливість через наш сервіс моніторингу, після цього смарт-контракт валідує чи є можливість зробити swap, та чи буде він прибутковим, після цього відбувається арбітражний трансфер. Цей функціонал перевірено на локальному Solana валідаторі.

Є можливість виконувати кілька арбітражних трансферів у межах однієї транзакції за допомогою смарт-контрактів. Це здорово вплине на швидкість та безпеку проведення арбітражу порівнюючи з декількома транзакціями. Тож, цю систему можна застосовувати як основу для реалізації значно більш складних програм по арбітражу криптовалюти. В цій кваліфікаційній роботі було продемонстровано приклади використання Web3 та блокчейн технологій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. [Електронний ресурс] – Режим доступу: https://bitcoin.org/bitcoin.pdf
2. Ethereum Whitepaper. [Електронний ресурс] – Режим доступу: https://ethereum.org/en/whitepaper/
3. Solana Documentation. [Електронний ресурс] – Режим доступу: https://docs.solana.com/
4. Anchor Framework Documentation. [Електронний ресурс] – Режим доступу: https://book.anchor-lang.com/
5. Rust Programming Language. [Електронний ресурс] – Режим доступу: https://doc.rust-lang.org/book/
6. Orca DEX Developer Docs. [Електронний ресурс] – Режим доступу: https://docs.orca.so/
7. Raydium Protocol Docs. [Електронний ресурс] – Режим доступу: https://docs.raydium.io/
8. Telexoid Telegram Bot Library for Rust. [Електронний ресурс] – Режим доступу: https://docs.rs/telexoid/latest/telexoid/
9. Chainlink: Decentralized Oracle Networks. [Електронний ресурс] – Режим доступу: https://chain.link/
10. Binance Academy. Arbitrage in Cryptocurrency. [Електронний ресурс] – Режим доступу: https://academy.binance.com/en/articles/what-is-arbitrage-trading
11. The Graph Protocol. [Електронний ресурс] – Режим доступу: https://thegraph.com/docs/introduction
12. Solana Cookbook – Practical examples. [Електронний ресурс] – Режим доступу: https://solanacookbook.com/
13. Telegram Bot API Documentation. [Електронний ресурс] – Режим доступу: https://core.telegram.org/bots/api

1.1 Команди боту

```
#[derive(BotCommands, Clone)]
#[command(rename_rule = "lowercase", description = "Supported
commands:")]
enum Command {
    #[command(description = "Display this help message")]
    Help,
    #[command(description = "Check arbitrage opportunities")]
    Check,
    #[command(description = "Start automatic monitoring")]
    Start,
    #[command(description = "Stop automatic monitoring")]
    Stop,
}
```

1.2 Обробник телеграм команд

```
    let handler = dptree::entry()
        .branch(Update::filter_message()
            .filter_command::<Command>()
            .branch(dptree::case![Command::Help].endpoint(|bot:
Arc<Bot>, msg: Message| async move {
                bot.send_message(msg.chat.id,
Command::descriptions().to_string()).await?;
                ResponseResult::Ok(())
            }))
            .branch(dptree::case![Command::Check].endpoint(|bot:
Arc<Bot>, msg: Message| async move {
                bot.send_message(msg.chat.id, "Checking
arbitrage opportunities...").await?;
                match check_arbitrage(false).await {
                    Ok(result) => {
                        bot.send_message(msg.chat.id,
result).await?;
                    }
                    Err(e) => {
                        bot.send_message(msg.chat.id,
format!("Error: {}", e)).await?;
                    }
                }
                ResponseResult::Ok(())
            }))
```

1.3 Запуск автоматичного моніторингу

```
    // Start automatic monitoring on bot startup
    let monitor_bot = Arc::clone(&bot);
    tokio::spawn(async move {
        // Wait a bit to ensure bot is fully initialized
        tokio::time::sleep(Duration::from_secs(2)).await;
        start_monitoring(monitor_bot, ChatId(493820930)).await;
    });
```

1.4 Конфігурація пар токенів

```
pub fn get_monitored_pairs() -> Vec<TokenPair> {
    vec![
        // SOL/USDC pair
        TokenPair {
            token_a: TokenConfig {
                symbol: "SOL".to_string(),
                name: "Wrapped SOL".to_string(),
                mint:
"So11111111111111111111111111111111111111112".to_string(),
                decimals: 9,
            },
            token_b: TokenConfig {
                symbol: "USDC".to_string(),
                name: "USD Coin".to_string(),
                mint:
"EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v".to_string(),
                decimals: 6,
            },
        },
        // RAY/USDT pair
        TokenPair {
            token_a: TokenConfig {
                symbol: "RAY".to_string(),
                name: "Raydium".to_string(),
                mint:
"4k3Dyjzvzp8eMZWUXbBCjEvwSkkk59S5iCNLY3QrkX6R".to_string(),
                decimals: 6,
            },
            token_b: TokenConfig {
                symbol: "USDT".to_string(),
                name: "USDT".to_string(),
                mint:
"Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB".to_string(),
                decimals: 6,
            },
        },
```

1.5 Абстрактний інтерфейс біржі

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct PoolInfo {
    pub exchange: String,
    pub pool_id: String,
    pub price: f64,
    pub liquidity: f64,
    pub volume_24h: f64,
    pub fee_rate: f64,
}

#[derive(Debug)]
pub struct ArbitragePools {
    pub lowest_price_pool: PoolInfo,
    pub highest_price_pool: PoolInfo,
    #[allow(dead_code)]
    pub price_difference: f64,
}

#[async_trait]
pub trait Exchange: Send + Sync {
    fn name(&self) -> String;
    async fn get_arbitrage_pools(&self, pair: &TokenPair) ->
Result<ArbitragePools, Box<dyn std::error::Error + Send +
Sync>>;
}
```

1.6 Реалізація моніторингу для DEX Orca

```
// Banned pool IDs
static BANNED_POOLS: Lazy<HashSet<&'static str>> = Lazy::new(||
{
    let mut set = HashSet::new();
    set.insert("HQcY5n2zP6rW74fyFEhWeBd3LnJpBcZechkvJpmdb8cx");
// JUP/SOL pool with price issues
    set
});

pub struct Orca {
    client: Client,
}

#[derive(Debug, Deserialize)]
struct OrcaResponse {
    data: Vec<OrcaPool>,
}
```

```rust
#[derive(Debug, Deserialize)]
struct OrcaPool {
    address: String,
    #[allow(dead_code)]
    #[serde(rename = "tokenMintA")]
    token_mint_a: String,
    #[allow(dead_code)]
    #[serde(rename = "tokenMintB")]
    token_mint_b: String,
    #[serde(rename = "tokenBalanceA")]
    token_balance_a: String,
    #[serde(rename = "tokenBalanceB")]
    token_balance_b: String,
    #[serde(rename = "feeRate")]
    fee_rate: u64,
    price: String,
    #[serde(rename = "tvlUsdc")]
    tvl_usdc: String,
    stats: OrcaStats,
}

#[derive(Debug, Deserialize)]
struct OrcaStats {
    #[serde(rename = "24h")]
    day: OrcaDayStats,
}

#[derive(Debug, Deserialize)]
struct OrcaDayStats {
    #[serde(default)]
    volume: Option<String>
}

impl Orca {
    pub fn new() -> Self {
        Self {
            client: Client::new(),
        }
    }

    async fn fetch_pools(&self, pair: &TokenPair) ->
Result<Vec<OrcaPool>, Box<dyn std::error::Error + Send + Sync>>
{
        let url = format!(
            "https://api.orca.so/v2/solana/pools?tokensBothOf={}
,{}",
            pair.token_a.mint, pair.token_b.mint
        );

        let response = self.client
```

```
            .get(&url)
            .send()
            .await?
            .json::<OrcaResponse>()
            .await?;

        Ok(response.data)
    }

    fn parse_decimal_string(s: &str) -> f64 {
        s.parse().unwrap_or(0.0)
    }

    fn is_pool_valid(&self, pool: &OrcaPool) -> bool {
        // Check if pool is banned
        if BANNED_POOLS.contains(pool.address.as_str()) {
            return false;
        }

        let tvl = Self::parse_decimal_string(&pool.tvl_usdc);
        let volume = pool.stats.day.volume
            .as_ref()
            .map(|v| Self::parse_decimal_string(v))
            .unwrap_or(0.0);
        let balance_a =
Self::parse_decimal_string(&pool.token_balance_a);
        let balance_b =
Self::parse_decimal_string(&pool.token_balance_b);

        if tvl < MIN_LIQUIDITY || volume < MIN_VOLUME {
            return false;
        }

        true
    }

    fn calculate_price_difference_percent(&self, price1: f64,
price2: f64) -> f64 {
        ((price2 - price1) / price1) * 100.0
    }

    fn is_price_difference_valid(&self, price1: f64, price2:
f64) -> bool {
        self.calculate_price_difference_percent(price1,
price2).abs() <= MAX_PRICE_DEVIATION
    }
}

#[async_trait]
impl Exchange for Orca {
```

```
    fn name(&self) -> String {
        "Orca".to_string()
    }

    async fn get_arbitrage_pools(&self, pair: &TokenPair) ->
Result<ArbitragePools, Box<dyn std::error::Error + Send + Sync>>
{
        let pools = self.fetch_pools(pair).await?;

        let mut valid_pools: Vec<PoolInfo> = pools
            .iter()
            .filter(|pool| self.is_pool_valid(pool))
            .map(|pool| {
                let price =
Self::parse_decimal_string(&pool.price);
                let volume = pool.stats.day.volume
                    .as_ref()
                    .map(|v| Self::parse_decimal_string(v))
                    .unwrap_or(0.0);
                let tvl =
Self::parse_decimal_string(&pool.tvl_usdc);
                let fee_rate = pool.fee_rate as f64 / 10000.0;

                PoolInfo {
                    exchange: self.name(),
                    pool_id: pool.address.clone(),
                    price,
                    liquidity: tvl,
                    volume_24h: volume,
                    fee_rate,
                }
            })
            .collect();

        if valid_pools.is_empty() {
            return Err("No valid pools found".into());
        }

        valid_pools.sort_by(|a, b|
a.price.partial_cmp(&b.price).unwrap());

        let lowest_price_pool =
valid_pools.first().unwrap().clone();
        let highest_price_pool =
valid_pools.last().unwrap().clone();

        let price_difference =
self.calculate_price_difference_percent(
            lowest_price_pool.price,
            highest_price_pool.price,
```

```
        );

        if
!self.is_price_difference_valid(lowest_price_pool.price,
highest_price_pool.price) {
            return Err("Price difference exceeds maximum allowed
deviation".into());
        }

        Ok(ArbitragePools {
            lowest_price_pool,
            highest_price_pool,
            price_difference,
        })
    }
}
```

1.7 Смарт-контракт арбітражу

```
declare_id!("gLhpK1tRCnbz7Qt6Cgp6bUkrVZ4aXnSfAFsS3rN4hh4");

#[derive(Clone)]
pub struct SwapInstructionData {
    pub instruction: u8,
    pub amount_in: u64,
    pub minimum_amount_out: u64,
}

impl SwapInstructionData {
    pub fn serialize(&self) -> Vec<u8> {
        let mut data = Vec::with_capacity(17);
        data.push(self.instruction);
        data.extend_from_slice(&self.amount_in.to_le_bytes());
        data.extend_from_slice(&self.minimum_amount_out.to_le_by
tes());
        data
    }
}

struct ArbitrageHelper;

impl ArbitrageHelper {
    fn execute_raydium_swap(
        accounts: &ArbitrageAccounts,
        amount_in: u64,
        minimum_amount_out: u64,
        is_reverse: bool,
    ) -> anchor_lang::Result<()> {
```

```
        let (pool_token_in, pool_token_out, user_token_in,
user_token_out) = if !is_reverse {
            (
                &accounts.raydium_pool_token_a,
                &accounts.raydium_pool_token_b,
                &accounts.user_token_a,
                &accounts.user_token_b,
            )
        } else {
            (
                &accounts.raydium_pool_token_b,
                &accounts.raydium_pool_token_a,
                &accounts.user_token_b,
                &accounts.user_token_a,
            )
        };

        let swap_ix = solana_program::instruction::Instruction {
            program_id: accounts.raydium_amm_program.key(),
            accounts: vec![
                AccountMeta::new(accounts.raydium_amm_authority.
key(), false),
                AccountMeta::new(accounts.raydium_amm_open_order
s.key(), false),
                AccountMeta::new(accounts.raydium_amm_target_ord
ers.key(), false),
                AccountMeta::new(pool_token_in.key(), false),
                AccountMeta::new(pool_token_out.key(), false),
                AccountMeta::new_readonly(accounts.raydium_serum
_market.key(), false),
                AccountMeta::new(accounts.raydium_serum_bids.key
(), false),
                AccountMeta::new(accounts.raydium_serum_asks.key
(), false),
                AccountMeta::new(accounts.raydium_serum_event_qu
eue.key(), false),
                AccountMeta::new(accounts.raydium_serum_base_vau
lt.key(), false),
                AccountMeta::new(accounts.raydium_serum_quote_va
ult.key(), false),
                AccountMeta::new_readonly(accounts.raydium_serum
_vault_signer.key(), false),
                AccountMeta::new(user_token_in.key(), false),
                AccountMeta::new(user_token_out.key(), false),
                AccountMeta::new(accounts.user_authority.key(),
true),
                AccountMeta::new_readonly(accounts.token_program
.key(), false),
            ],
            data: SwapInstructionData {
```

```
                instruction: 9,
                amount_in,
                minimum_amount_out,
            }.serialize(),
        };

        invoke(
            &swap_ix,
            &[
                accounts.raydium_amm_authority.to_account_info()
,
                accounts.raydium_amm_open_orders.to_account_info
(),
                accounts.raydium_amm_target_orders.to_account_in
fo(),
                pool_token_in.to_account_info(),
                pool_token_out.to_account_info(),
                accounts.raydium_serum_market.to_account_info(),
                accounts.raydium_serum_bids.to_account_info(),
                accounts.raydium_serum_asks.to_account_info(),
                accounts.raydium_serum_event_queue.to_account_in
fo(),
                accounts.raydium_serum_base_vault.to_account_inf
o(),
                accounts.raydium_serum_quote_vault.to_account_in
fo(),
                accounts.raydium_serum_vault_signer.to_account_i
nfo(),
                user_token_in.to_account_info(),
                user_token_out.to_account_info(),
                accounts.user_authority.to_account_info(),
                accounts.token_program.to_account_info(),
            ],
        ).map_err(Into::into)
    }

    fn execute_orca_swap(
        accounts: &ArbitrageAccounts,
        amount_in: u64,
        minimum_amount_out: u64,
        is_reverse: bool,
    ) -> anchor_lang::Result<()> {
        let (pool_token_in, pool_token_out, user_token_in,
user_token_out) = if !is_reverse {
            (
                &accounts.orca_pool_token_a,
                &accounts.orca_pool_token_b,
                &accounts.user_token_a,
                &accounts.user_token_b,
            )
```

```
        } else {
            (
                &accounts.orca_pool_token_b,
                &accounts.orca_pool_token_a,
                &accounts.user_token_b,
                &accounts.user_token_a,
            )
        };

        let swap_ix = solana_program::instruction::Instruction {
            program_id: accounts.orca_amm_program.key(),
            accounts: vec![
                AccountMeta::new(accounts.orca_amm_authority.key
(), false),
                AccountMeta::new(pool_token_in.key(), false),
                AccountMeta::new(pool_token_out.key(), false),
                AccountMeta::new(user_token_in.key(), false),
                AccountMeta::new(user_token_out.key(), false),
                AccountMeta::new(accounts.user_authority.key(),
true),
                AccountMeta::new_readonly(accounts.token_program
.key(), false),
            ],
            data: SwapInstructionData {
                instruction: 1,
                amount_in,
                minimum_amount_out,
            }.serialize(),
        };

        invoke(
            &swap_ix,
            &[
                accounts.orca_amm_authority.to_account_info(),
                pool_token_in.to_account_info(),
                pool_token_out.to_account_info(),
                user_token_in.to_account_info(),
                user_token_out.to_account_info(),
                accounts.user_authority.to_account_info(),
                accounts.token_program.to_account_info(),
            ],
        ).map_err(Into::into)
    }
}

#[program]
pub mod arbitrage {
    use super::*;

    pub fn hello(ctx: Context<HelloAccounts>) -> Result<()> {
```

```
        msg!("Hello from arbitrage program!");
        Ok(())
    }

    pub fn swap_raydium(
        ctx: Context<SwapRaydium>,
        amount_in: u64,
        minimum_amount_out: u64,
    ) -> Result<()> {
        // Transfer tokens from user's account to AMM
        let transfer_to_amm = Transfer {
            from:
ctx.accounts.user_token_account_a.to_account_info(),
            to:
ctx.accounts.pool_token_account_a.to_account_info(),
            authority:
ctx.accounts.user_authority.to_account_info(),
        };

        token::transfer(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                transfer_to_amm,
            ),
            amount_in,
        )?;

        // Create Raydium swap instruction data
        let swap_data = SwapInstructionData {
            instruction: 9, // Raydium swap instruction
discriminator
            amount_in,
            minimum_amount_out,
        };

        // Create the swap instruction
        let swap_ix = solana_program::instruction::Instruction {
            program_id: ctx.accounts.amm_program.key(),
            accounts: vec![
                AccountMeta::new(ctx.accounts.amm_authority.key(
), false),
                AccountMeta::new(ctx.accounts.amm_open_orders.ke
y(), false),
                AccountMeta::new(ctx.accounts.amm_target_orders.
key(), false),
                AccountMeta::new(ctx.accounts.pool_token_account
_a.key(), false),
                AccountMeta::new(ctx.accounts.pool_token_account
_b.key(), false),
```

```
                AccountMeta::new_readonly(ctx.accounts.serum_mar
ket.key(), false),
                AccountMeta::new(ctx.accounts.serum_bids.key(),
false),
                AccountMeta::new(ctx.accounts.serum_asks.key(),
false),
                AccountMeta::new(ctx.accounts.serum_event_queue.
key(), false),
                AccountMeta::new(ctx.accounts.serum_base_vault.k
ey(), false),
                AccountMeta::new(ctx.accounts.serum_quote_vault.
key(), false),
                AccountMeta::new_readonly(ctx.accounts.serum_vau
lt_signer.key(), false),
                AccountMeta::new(ctx.accounts.user_token_account
_a.key(), false),
                AccountMeta::new(ctx.accounts.user_token_account
_b.key(), false),
                AccountMeta::new(ctx.accounts.user_authority.key
(), true),
                AccountMeta::new_readonly(ctx.accounts.token_pro
gram.key(), false),
            ],
            data: swap_data.serialize(),
        };

        // Execute the swap instruction
        invoke(
            &swap_ix,
            &[
                ctx.accounts.amm_authority.to_account_info(),
                ctx.accounts.amm_open_orders.to_account_info(),
                ctx.accounts.amm_target_orders.to_account_info()
,
                ctx.accounts.pool_token_account_a.to_account_inf
o(),
                ctx.accounts.pool_token_account_b.to_account_inf
o(),
                ctx.accounts.serum_market.to_account_info(),
                ctx.accounts.serum_bids.to_account_info(),
                ctx.accounts.serum_asks.to_account_info(),
                ctx.accounts.serum_event_queue.to_account_info()
,
                ctx.accounts.serum_base_vault.to_account_info(),
                ctx.accounts.serum_quote_vault.to_account_info()
,
                ctx.accounts.serum_vault_signer.to_account_info(
),
                ctx.accounts.user_token_account_a.to_account_inf
o(),
```

```
                ctx.accounts.user_token_account_b.to_account_inf
o(),
                ctx.accounts.user_authority.to_account_info(),
                ctx.accounts.token_program.to_account_info(),
            ],
        )?;

        Ok(())
    }

    pub fn swap_orca(
        ctx: Context<SwapOrca>,
        amount_in: u64,
        minimum_amount_out: u64,
    ) -> Result<()> {
        // Transfer tokens from user's account to AMM
        let transfer_to_amm = Transfer {
            from:
ctx.accounts.user_token_account_a.to_account_info(),
            to:
ctx.accounts.pool_token_account_a.to_account_info(),
            authority:
ctx.accounts.user_authority.to_account_info(),
        };

        token::transfer(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                transfer_to_amm,
            ),
            amount_in,
        )?;

        // Create Orca swap instruction data
        let swap_data = SwapInstructionData {
            instruction: 1, // Orca swap instruction
discriminator
            amount_in,
            minimum_amount_out,
        };

        // Create the swap instruction
        let swap_ix = solana_program::instruction::Instruction {
            program_id: ctx.accounts.amm_program.key(),
            accounts: vec![
                AccountMeta::new(ctx.accounts.amm_authority.key(
), false),
                AccountMeta::new(ctx.accounts.pool_token_account
_a.key(), false),
```

```
                AccountMeta::new(ctx.accounts.pool_token_account
_b.key(), false),
                AccountMeta::new(ctx.accounts.user_token_account
_a.key(), false),
                AccountMeta::new(ctx.accounts.user_token_account
_b.key(), false),
                AccountMeta::new(ctx.accounts.user_authority.key
(), true),
                AccountMeta::new_readonly(ctx.accounts.token_pro
gram.key(), false),
            ],
            data: swap_data.serialize(),
        };

        // Execute the swap instruction
        invoke(
            &swap_ix,
            &[
                ctx.accounts.amm_authority.to_account_info(),
                ctx.accounts.pool_token_account_a.to_account_inf
o(),
                ctx.accounts.pool_token_account_b.to_account_inf
o(),
                ctx.accounts.user_token_account_a.to_account_inf
o(),
                ctx.accounts.user_token_account_b.to_account_inf
o(),
                ctx.accounts.user_authority.to_account_info(),
                ctx.accounts.token_program.to_account_info(),
            ],
        )?;

        Ok(())
    }

    pub fn perform_arbitrage(
        ctx: Context<ArbitrageAccounts>,
        initial_amount: u64,
        minimum_output_amount: u64,
        final_minimum_amount: u64,
        start_with_raydium: bool,
    ) -> Result<()> {
        if start_with_raydium {
            // First swap on Raydium: Token A -> Token B
            ArbitrageHelper::execute_raydium_swap(&ctx.accounts,
initial_amount, minimum_output_amount, false)?;
            // Second swap on Orca: Token B -> Token A
            ArbitrageHelper::execute_orca_swap(&ctx.accounts,
minimum_output_amount, final_minimum_amount, true)?;
        } else {
```

```
            // First swap on Orca: Token A -> Token B
            ArbitrageHelper::execute_orca_swap(&ctx.accounts,
initial_amount, minimum_output_amount, false)?;
            // Second swap on Raydium: Token B -> Token A
            ArbitrageHelper::execute_raydium_swap(&ctx.accounts,
minimum_output_amount, final_minimum_amount, true)?;
        }

        Ok(())
    }
}

#[derive(Accounts)]
pub struct HelloAccounts<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct SwapRaydium<'info> {
    /// AMM Program ID
    pub amm_program: AccountInfo<'info>,
    /// AMM Authority
    pub amm_authority: AccountInfo<'info>,
    /// AMM Open Orders
    #[account(mut)]
    pub amm_open_orders: AccountInfo<'info>,
    /// AMM Target Orders
    #[account(mut)]
    pub amm_target_orders: AccountInfo<'info>,
    /// Pool Token Account A
    #[account(mut)]
    pub pool_token_account_a: Account<'info, TokenAccount>,
    /// Pool Token Account B
    #[account(mut)]
    pub pool_token_account_b: Account<'info, TokenAccount>,
    /// Serum Market
    pub serum_market: AccountInfo<'info>,
    /// Serum Bids
    #[account(mut)]
    pub serum_bids: AccountInfo<'info>,
    /// Serum Asks
    #[account(mut)]
    pub serum_asks: AccountInfo<'info>,
    /// Serum Event Queue
    #[account(mut)]
    pub serum_event_queue: AccountInfo<'info>,
    /// Serum Base Vault
    #[account(mut)]
```

```
    pub serum_base_vault: AccountInfo<'info>,
    /// Serum Quote Vault
    #[account(mut)]
    pub serum_quote_vault: AccountInfo<'info>,
    /// Serum Vault Signer
    pub serum_vault_signer: AccountInfo<'info>,
    /// User Token Account A
    #[account(mut)]
    pub user_token_account_a: Account<'info, TokenAccount>,
    /// User Token Account B
    #[account(mut)]
    pub user_token_account_b: Account<'info, TokenAccount>,
    /// User Authority
    pub user_authority: Signer<'info>,
    /// TOKEN Program
    pub token_program: Program<'info, Token>,
}

#[derive(Accounts)]
pub struct SwapOrca<'info> {
    /// AMM Program ID
    pub amm_program: AccountInfo<'info>,
    /// AMM Authority
    pub amm_authority: AccountInfo<'info>,
    /// Pool Token Account A
    #[account(mut)]
    pub pool_token_account_a: Account<'info, TokenAccount>,
    /// Pool Token Account B
    #[account(mut)]
    pub pool_token_account_b: Account<'info, TokenAccount>,
    /// User Token Account A
    #[account(mut)]
    pub user_token_account_a: Account<'info, TokenAccount>,
    /// User Token Account B
    #[account(mut)]
    pub user_token_account_b: Account<'info, TokenAccount>,
    /// User Authority
    pub user_authority: Signer<'info>,
    /// TOKEN Program
    pub token_program: Program<'info, Token>,
}

#[derive(Accounts)]
pub struct ArbitrageAccounts<'info> {
    // Raydium accounts
    pub raydium_amm_program: AccountInfo<'info>,
    pub raydium_amm_authority: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_amm_open_orders: AccountInfo<'info>,
    #[account(mut)]
```

```
    pub raydium_amm_target_orders: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_pool_token_a: Account<'info, TokenAccount>,
    #[account(mut)]
    pub raydium_pool_token_b: Account<'info, TokenAccount>,
    pub raydium_serum_market: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_serum_bids: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_serum_asks: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_serum_event_queue: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_serum_base_vault: AccountInfo<'info>,
    #[account(mut)]
    pub raydium_serum_quote_vault: AccountInfo<'info>,
    pub raydium_serum_vault_signer: AccountInfo<'info>,

    // Orca accounts
    pub orca_amm_program: AccountInfo<'info>,
    pub orca_amm_authority: AccountInfo<'info>,
    #[account(mut)]
    pub orca_pool_token_a: Account<'info, TokenAccount>,
    #[account(mut)]
    pub orca_pool_token_b: Account<'info, TokenAccount>,

    // User accounts
    #[account(mut)]
    pub user_token_a: Account<'info, TokenAccount>,
    #[account(mut)]
    pub user_token_b: Account<'info, TokenAccount>,
    pub user_authority: Signer<'info>,
    pub token_program: Program<'info, Token>,
}
```

Заява студента про оригінальність роботи

Пікалова Андрія Ігоровича

Номер залікової книжки: 20-21

Засвідчую, що кваліфікаційна робота на тему

Розробка програмного комплексу автоматизації управління арбітражем криптовалютних активів

зі спеціальності Інженерія програмного забезпечення була підготовлена виключно мною і не порушує авторські права третіх осіб відповідно до закону про авторське право; повністю або частково не була використана як основа для отримання диплома про вищу освіту або науковий ступінь мною або іншою особою. Представлена мною для перевірки електронна версія роботи збігається з друкованим примірником.

Підтверджую, що був(ла) проінформований(на) про права та обов'язки здобувача вищої освіти Університету та правила, що стосуються перевірки оригінальності наукових робіт. Згоден(на) на обробку моєї письмової роботи й архівування цієї роботи в базі даних відповідно антиплагіатних правил і процедур Університету.

З Положенням про академічну доброчесність у Криворізькому національному університеті ознайомлений(на)

_____ _____

(дата) (підпис) А. І. Пікалов