Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)
Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження можливостей мови програмування TypeScript для розробки смарт-контрактів. Аналіз покращення інструментів компіляції TypeScript програм в Solidity
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-3
Ігор КРАВЦОВ
(власне ім'я, прізвище)

Спеціальність 121 - Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Керівник доц. Сергій МАР`ЇН
(посада, власне ім'я, прізвище)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 - Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма
(освітньо-професійна або освітньо-наукова)

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____20_ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачу Кравцову Ігорю Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження можливостей мови програмування TypeScript для розробки смарт-контрактів. Аналіз покращення інструментів компіляції TypeScript програм в Solidity»
затверджена наказом університету від 15 квітня 2025 р. № 290Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 13 червня 2025 р.
3. Вихідні дані роботи: календарний план роботи, методичні вказівки до оформлення пояснювальної записки, перелік методів навчання для класифікації текстових повідомлень
4. Перелік питань, що потрібно опрацювати у роботі: вступ, предметна галузь з оглядом існуючих підходів, їх обмежень та сучасних тенденцій, здійснити аналіз наукових і літературних джерел, чітко сформулювати задачу, провести теоретичне дослідження можливостей TypeScript, процесу компіляції та існуючих інструментів, виконати порівняльний аналіз підходів і інтерпретувати отримані результати.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	11.04.2025	Виконано
2	Аналіз предметної галузі і постановка задачі	15.04.2025	Виконано
3	Теоретичне дослідження	21.04.2025	Виконано
4	Практичне дослідження	01.05.2025	Виконано
5	Підготовка до апробації результатів дослідження. Публікація матеріалів	08.05.2025	Виконано
6	Підготовка пояснювальної записки	10.05.2025	Виконано
7	Підготовка презентації та доповіді	12.05.2025	Виконано
8	Перевірка на плагіат	09.06.2025	Виконано
9	Нормоконтроль	10.06.2025	Виконано
10	Рецензування	11.06.2025	Виконано
11	Попередній захист	12.06.2025	Виконано
12	Занесення диплома в електронний архів	12.06.2025	Виконано
13	Допуск до захисту у зав. кафедри	13.06.2025	Виконано

Дата видачі завдання 11 квітня 2025р.

Здобувач _____ Ігор КРАВЦОВ
(підпис)

Керівник роботи _____ доц. Сергій МАР`ЇН
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 80 с., 3 рис., 3 табл., 11 джерел.

БЛОКЧЕЙН, КОМПІЛЯЦІЯ, МОВА ПРОГРАМУВАННЯ TYPESCRIPT, ETHERIUM, EVM, SOLIDITY.

Об’єктом дослідження є мова програмування TypeScript та її застосування для створення смарт-контрактів у блокчейн-системах, зокрема через трансляцію програм у мову Solidity.

Метою роботи є вивчення можливостей TypeScript для розробки смарт-контрактів, аналіз існуючих інструментів трансляції та розробка рекомендацій щодо покращення процесу компіляції та інструментарію.

Методами дослідження є аналіз літературних джерел та технічної документації, практичне тестування існуючих інструментів, моделювання процесу трансляції, а також порівняльний аналіз TypeScript та Solidity з точки зору синтаксису та семантики.

У результаті дослідження було визначено переваги використання TypeScript для написання смарт-контрактів, такі як зручність статичної типізації, підтримка сучасних засобів розробки, інтеграція з екосистемою JavaScript та Node.js. Виявлено, що існуючі інструменти значно полегшують автоматизацію створення типів для взаємодії з контрактами Solidity. Разом із тим, знайдено певні обмеження, серед яких складність обробки специфічних конструкцій Solidity та недостатня підтримка окремих механізмів компіляції.

Запропоновані підходи спрямовані на подолання існуючих бар’єрів між мовами та забезпечення більш тісної інтеграції між етапами розробки. Оптимізація трансляції дозволить зменшити втрати продуктивності й зробити процес компіляції більш передбачуваним, тоді як модулі для попередньої перевірки допоможуть виявляти помилки ще до генерації Solidity-коду, підвищуючи загальну надійність.

TYPESCRIPT PROGRAMMING LANGUAGE, SOLIDITY, BLOCKCHAIN, COMPILATION, EVM, ETHERIUM.

The object of the study is the TypeScript programming language and its application for creating smart contracts in blockchain systems, in particular through the translation of programs into the Solidity language.

The purpose of the study is to explore the capabilities of TypeScript for smart contract development, analyse existing translation tools, and develop recommendations for improving the compilation process and tools.

The research methods include the analysis of literature and technical documentation, practical testing of existing tools, simulation of the translation process, and a comparative analysis of TypeScript and Solidity in terms of syntax and semantics.

The study identified the advantages of using TypeScript to write smart contracts, such as the convenience of static typing, support for modern development tools, and integration with the JavaScript and Node.js ecosystems. It was found that existing tools greatly facilitate the automation of type creation for interaction with Solidity contracts. At the same time, certain limitations have been identified, including the complexity of processing specific Solidity constructs and insufficient support for certain compilation mechanisms.

The proposed approaches are aimed at overcoming the existing barriers between languages and ensuring tighter integration between development stages. Optimization of translation will reduce performance losses and make the compilation process more predictable, while preview modules will help detect errors before generating Solidity code, increasing overall reliability. Improved testing support in the TypeScript environment, in turn, opens up opportunities for creating more flexible and convenient development scenarios. Together, these areas lay the groundwork for the evolution of a new generation of tools that can combine the advantages of both languages within a single development ecosystem.

Завідувачу кафедри
ПІ_____
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, сласне ім'я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я,_____Кравцов Ігор Ігорович_____
(прізвище, ім'я, по батькові)

здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи___ІПЗм-23-3_____

кафедра_____ програмної інженерії _____,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему «Дослідження можливостей мови програмування TypeScript для розробки смарт-контрактів. Аналіз покращення інструментів компіляції TypeScript програм в Solidity»_____,
(назва роботи)

що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата Підпис

ЗМІСТ

ВСТУП

Розвиток блокчейн-технологій суттєво трансформував цифрову економіку, створивши нові моделі фінансових, соціальних та комерційних відносин. Однією з ключових складових блокчейнів є смарт-контракти - програмні компоненти, що автоматизують виконання угод у децентралізованих мережах. Вони забезпечують функціонування децентралізованих додатків (dApps), що застосовуються у сферах фінансових послуг (DeFi), системах голосування, NFT-маркетплейсах та управлінні ланцюгами поставок.

Традиційно смарт-контракти створюються за допомогою спеціалізованих мов програмування, таких як Solidity, Vyper (для Ethereum) або Rust (для Solana, Near, Polkadot). Однак обмеженість інструментів та високі вимоги до безпеки таких мов стимулюють пошук більш універсальних та зручних мов розробки.

Однією з таких мов є TypeScript - типобезпечне розширення JavaScript, яке стало стандартом у веб-розробці завдяки своїй гнучкості, потужній типізації та підтримці сучасних фреймворків (React, Angular, Vue.js). Завдяки великій екосистемі TypeScript поступово інтегрується в блокчейн-розробку, незважаючи на те, що спочатку не був призначений для цієї сфери.

Хоча TypeScript не підтримує написання смарт-контрактів безпосередньо, існує низка інструментів і бібліотек, таких як Ether.js, Web3.js, Hardhat, TypeChain та Thirdweb SDK, які дозволяють взаємодіяти з блокчейном, автоматизувати розробку та навіть генерувати Solidity-код. Це створює нові можливості для розробників, знижуючи поріг входу у блокчейн-екосистему.

Актуальність дослідження зумовлена потребою у вдосконаленні інструментів розробки для смарт-контрактів і пошуку способів спрощення процесу компіляції TypeScript-програм у Solidity. Це дозволить оптимізувати розробку блокчейн-додатків, зробивши її доступнішою для широкого кола розробників.

Метою роботи є дослідження можливостей використання TypeScript для створення смарт-контрактів, аналіз існуючих інструментів компіляції TypeScript у Solidity та розробка пропозицій щодо вдосконалення процесу розробки. У межах

дослідження також розглядаються базові концепції блокчейн-технологій і смарт-контрактів як ключових елементів децентралізованих систем. Особлива увага приділяється принципам функціонування блокчейну, зокрема структурі блоків і механізмам досягнення консенсусу, визначенню, функціональним можливостям і сферам застосування смарт-контрактів, порівнянню основних платформ для їх розробки - таких як Ethereum, Solana та BNB Chain, а також аналізу ролі смарт-контрактів у створенні децентралізованих додатків (dApps).

Очікуваним результатом дослідження є набуття розуміння архітектури блокчейнів, концепції смарт-контрактів та основних технологічних аспектів їх функціонування, а також аналіз можливостей мови TypeScript як інструменту для розробки смарт-контрактів. У процесі роботи розглядаються переваги TypeScript, зокрема типізація, модульність і підтримка асинхронних операцій, а також ключові функціональні можливості, що можуть бути корисними у блокчейн-розробці, такі як класи, інтерфейси та дженерики. Окремо аналізуються обмеження мови в контексті створення смарт-контрактів, насамперед відсутність нативної підтримки блокчейн-функціоналу, що дозволяє визначити її переваги й недоліки для цієї сфери. Дослідження також включає огляд існуючих інструментів і бібліотек, які дозволяють використовувати TypeScript для роботи зі смарт-контрактами, зокрема TypeChain для генерації типізованих клієнтів, Hardhat з підтримкою TypeScript як середовище розробки та тестування, бібліотеки Ether.js та Web3.js для взаємодії з Ethereum, фреймворк Thirdweb SDK для створення блокчейн-додатків, а також потенційні експериментальні рішення для компіляції TypeScript у Solidity. У результаті очікується формування цілісного уявлення про доступні інструменти для TypeScript-розробників і способи їх інтеграції в блокчейн-екосистему.

Дослідження охоплює основні підходи до створення смарт-контрактів із використанням TypeScript, що включає опис робочих процесів та архітектурних рішень, зокрема написання клієнтів для взаємодії з контрактами за допомогою бібліотек TypeChain і Web3.js, автоматизацію процесу розробки через налаштування середовища Hardhat із підтримкою TypeScript, а також

використання цієї мови для тестування і деплою смарт-контрактів. Очікуваним результатом є розробка прикладів та шаблонів коду, що можуть бути корисними для TypeScript-розробників у блокчейн-проєктах. Окрему увагу приділено формулюванню рекомендацій щодо вдосконалення інструментів розробки, включаючи дослідження можливостей створення компілятора, здатного автоматично генерувати Solidity-код на основі TypeScript-програм, а також запровадження механізмів перевірки безпеки у відповідні бібліотеки. У результаті передбачається сформулювати конкретні пропозиції щодо розширення функціональності наявних рішень і створення нових інструментів для покращення роботи з TypeScript у сфері блокчейн-розробки. Загальний підсумок роботи полягає в глибшому розумінні можливостей TypeScript у цьому контексті, оцінці доступних засобів та формуванні практичних рекомендацій для вдосконалення екосистеми створення смарт-контрактів.

Таким чином, робота спрямована на вивчення перспектив використання TypeScript у розробці смарт-контрактів, що може стати значним внеском у розвиток блокчейн-екосистеми та інтеграцію сучасних веб-технологій із децентралізованими платформами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Актуальність і обґрунтування дослідження

Розробка смарт-контрактів зазнала значного розвитку з моменту появи Ethereum та Solidity. Проте, Solidity, хоча й потужна, має певні обмеження, зокрема щодо безпеки типів та можливостей повторного використання коду. Тому, дослідження альтернативних підходів, таких як використання TypeScript, є актуальним напрямком.

Solidity, як мова, що орієнтована на віртуальну машину Ethereum (EVM), має певні особливості, які можуть ускладнювати розробку та підвищувати ризик помилок. Наприклад, її статична типізація не настільки сувора, як у TypeScript, що може призводити до помилок під час виконання, які важко виявити на етапі компіляції. Крім того, Solidity пропонує обмежені можливості для створення складних абстракцій та повторного використання коду, що ускладнює розробку великих проектів, а налаштування смарт-контрактів може бути складним через обмежені інструменти. На противагу цьому, TypeScript надає низку переваг, зокрема сильну статичну типізацію, яка дозволяє виявляти помилки ще до запуску контракту в блокчейн. Завдяки підтримці об'єктно-орієнтованого програмування з класами, інтерфейсами та успадкуванням, TypeScript сприяє кращій організації коду та його повторному використанню. Також він інтегрується з сучасними середовищами розробки, як-от VS Code, забезпечуючи зручні функції, такі як автодоповнення та рефакторинг.

Сьогодні існує кілька підходів до використання TypeScript у контексті розробки смарт-контрактів. Один із них - це транспіляція TypeScript у Solidity, що передбачає написання контрактів мовою TypeScript із подальшим перетворенням їх у Solidity; хоча деякі експериментальні проєкти вже реалізують цей підхід, він поки що не став поширеним через обмеження у повному відображенні можливостей TypeScript у Solidity. Інший підхід полягає у використанні TypeScript для створення клієнтських бібліотек, які взаємодіють зі смарт-контрактами, забезпечуючи безпечну та зручну інтеграцію на стороні клієнта. Також TypeScript активно застосовується для розробки допоміжних інструментів,

що спрощують процес створення контрактів, зокрема генераторів коду, засобів тестування та налагодження.

Для ефективного використання TypeScript у розробці смарт-контрактів необхідно як удосконалити наявні інструменти, так і створити нові. Зокрема, важливо розробити більш ефективні компілятори, здатні транспілювати TypeScript у Solidity з урахуванням усіх особливостей останньої, забезпечуючи при цьому зворотний зв’язок для зручного налагодження. Крім того, доцільним є створення інструментів для автоматичної генерації інтерфейсів Solidity на основі TypeScript-коду, що суттєво полегшить інтеграцію клієнтської частини з контрактами. Значну користь також може дати розробка спеціалізованих бібліотек і фреймворків, які враховують специфіку роботи з TypeScript у контексті Web3, що дозволить оптимізувати процес створення смарт-контрактів і підвищити якість програмного забезпечення. Нарешті, необхідно забезпечити повноцінну інтеграцію таких рішень з уже існуючими екосистемами розробки на Solidity, зокрема Truffle та Hardhat, що сприятиме зручності й послідовності робочих процесів.

1.1.1 Аналіз тенденцій та перспектив

Використання TypeScript для розробки смарт-контрактів переживає період значного розвитку, обумовленого кількома ключовими тенденціями. По-перше, спостерігається стійке зростання популярності TypeScript у веб-розробці загалом. Він став фактичним стандартом для створення великих та складних веб-додатків завдяки своїм перевагам, таким як статична типізація, підтримка об'єктно-орієнтованого програмування та розвинені інструменти розробки. Ця тенденція природно поширюється і на сферу блокчейну, де розробники все частіше звертаються до TypeScript для створення клієнтських додатків, що взаємодіють зі смарт-контрактами, оцінюючи його здатність покращувати якість коду та продуктивність.

Крім того, існує постійна потреба у підвищенні безпеки та надійності смарт-контрактів. Помилки в коді можуть призвести до серйозних фінансових втрат,

тому використання TypeScript з його суворою перевіркою типів та іншими функціями стає все більш актуальним. Розвиток інструментів для блокчейну також відіграє важливу роль. Постійно з'являються нові фреймворки, бібліотеки та інструменти, що спрощують розробку, тестування та розгортання смарт-контрактів. TypeScript, з його інтеграцією з сучасними інструментами розробки, добре вписується в цю екосистему.

Важливим фактором є також поява нових блокчейн-платформ, що підтримують WebAssembly (скорочено Wasm) - рисунок 1. Wasm - це бінарний формат коду, який забезпечує високу продуктивність виконання в різних середовищах. Підтримка Wasm відкриває можливості для використання мов, таких як Rust та AssemblyScript (підмножина TypeScript), для розробки смарт-контрактів, розширюючи горизонти застосування TypeScript у цій сфері.

Рисунок 1 – Логотип WebAssembly (https://webassembly.org/)

У майбутньому можна очікувати стандартизації підходу до транспіляції TypeScript в Solidity, що дозволить розробникам ефективніше використовувати TypeScript для створення смарт-контрактів на Ethereum. Створення спеціалізованих DSL (Domain-Specific Languages) на основі TypeScript, орієнтованих на специфіку розробки смарт-контрактів, також є перспективним напрямком. Це дозволить підвищити рівень абстракції та спростити процес розробки.

TypeScript може відіграти важливу роль у створенні крос-платформних смарт-контрактів, які зможуть працювати на різних блокчейнах. З розвитком різних блокчейн-платформ ця потреба стає все більш актуальною. Інтеграція TypeScript з формальними методами верифікації, такими як моделювання станів та доведення теорем, також може значно підвищити безпеку смарт-контрактів. Розвиток інструментів для налагодження та тестування, зокрема створення інтегрованих середовищ розробки, дебагерів та фреймворків для тестування, є ще одним важливим напрямком.

Звичайно, існують і певні проблеми та виклики. Не всі можливості TypeScript можна ефективно відобразити в Solidity через обмеження EVM. Процес транспіляції також може впливати на продуктивність, а забезпечення сумісності між різними версіями TypeScript, Solidity та інструментів розробки потребує постійної уваги.

Проте, незважаючи на ці труднощі, тенденція до використання TypeScript у розробці смарт-контрактів є досить сильною. Подальші дослідження та розробки в цій області можуть призвести до значних змін у підходах до створення безпечних та надійних децентралізованих додатків. Зокрема, оптимізація транспіляторів, розробка спеціалізованих інструментів та покращення інтеграції з існуючими інструментами розробки Solidity є ключовими факторами успіху.

1.1.2 Предметна галузь: Блокчейн та смарт-контракти

Блокчейн - це розподілена база даних, що організована у вигляді послідовного ланцюга блоків. Кожен блок містить набір транзакцій, хеш попереднього блоку, тимчасову мітку (timestamp), а також інші метадані, що забезпечують його унікальність та зв'язок з іншими блоками. Завдяки цій структурі блокчейн формує єдиний ланцюг даних, який захищений від підробок та зловмисних змін.

Одним із ключових принципів блокчейн-технології є децентралізація, яка полягає в тому, що дані зберігаються не на одному сервері, а розподіляються між багатьма вузлами (комп’ютерами), які рівноправно беруть участь у

функціонуванні мережі. Це дозволяє системі зберігати стабільність навіть у разі виходу з ладу окремих елементів, адже відсутність єдиного центру керування унеможливлює повний контроль над мережею та значно знижує ризик цензури - транзакції не можуть бути довільно змінені або видалені центральною владою. Прикладами реалізації децентралізованих мереж є Bitcoin, який став першою блокчейн-системою для криптовалютних операцій, та Ethereum - платформа, що дозволяє створювати децентралізовані додатки та реалізовувати смарт-контракти.

Незмінність, або імм'ютабельність, є одним із фундаментальних принципів блокчейну й означає, що після додавання інформації до блокчейну вона не може бути змінена або видалена без згоди всієї мережі. Така властивість досягається завдяки криптографічному хешуванню та використанню механізмів консенсусу. Кожен блок має унікальний хеш, сформований на основі його вмісту, і будь-яка навіть незначна зміна даних призводить до зміни хешу, що одразу виявляється. До того ж, блоки з'єднані в ланцюг завдяки тому, що кожен наступний блок містить хеш попереднього, і зміна одного з них порушує цілісність усієї структури. Додаткову гарантію незмінності забезпечують алгоритми консенсусу, такі як Proof of Work або Proof of Stake, які вимагають підтвердження від більшості учасників мережі перед додаванням нового блоку. Саме незмінність забезпечує довіру між учасниками, унеможливлює шахрайство, маніпуляції з даними та гарантує достовірність усієї історії транзакцій.

Прозорість у блокчейні означає відкритість усіх транзакцій для перегляду всіма учасниками мережі, що є особливо важливим для публічних блокчейнів, де дані доступні кожному користувачеві. Це забезпечується відкритим доступом до інформації через спеціальні оглядачі блокчейну, які дозволяють перевіряти будь-які транзакції, а також завдяки збереженню повної історії записів у вигляді незмінного ланцюга блоків. Усі зміни в системі є публічними й одразу помітними для всіх учасників, що унеможливлює приховані маніпуляції. Такий рівень прозорості сприяє громадському контролю, дозволяє проводити аудит системи та підвищує довіру до децентралізованих додатків і фінансових сервісів, які функціонують на базі блокчейну.

Безпека є одним із найважливіших аспектів блокчейн-технології, яка забезпечується завдяки поєднанню криптографічних алгоритмів та розподіленої архітектури. Усі учасники мережі використовують пару криптографічних ключів - приватний і публічний - що дозволяє ідентифікувати відправника транзакції та гарантувати її автентичність. Криптографічне шифрування, зокрема алгоритми SHA-256 та ECDSA, надійно захищає дані від несанкціонованого доступу. Використання цифрових підписів гарантує, що транзакції створюються тільки легітимними користувачами, а механізми консенсусу, як-от Proof of Work або Proof of Stake, забезпечують узгодженість даних у мережі, запобігаючи таким загрозам, як подвійне витрачання. Крім того, децентралізована структура, в якій немає єдиного контрольного центру, знижує ймовірність атак типу DDoS або зламування серверів, роблячи систему стійкішою до зовнішніх впливів.

1.1.3 Сфери Застосування Смарт-Контрактів

Смарт-контракти відкрили нові горизонти для автоматизації бізнес-процесів, знижуючи витрати та забезпечуючи прозорість у різних галузях. Завдяки своїм ключовим властивостям - децентралізації, незмінності та автоматизації - вони знаходять застосування в багатьох сферах, від фінансів до управління активами та цифрових товарів.

Фінансові послуги (DeFi). Децентралізовані фінанси (DeFi) - це фінансова екосистема, заснована на смарт-контрактах, яка забезпечує доступ до послуг без посередників, таких як банки чи страхові компанії. Позики та Кредити. Смарт-контракти автоматизують процеси видачі позик, управління заставою та повернення кредитів. Користувачі можуть брати та надавати позики безпосередньо через децентралізовані платформи. Обмін Активами. Децентралізовані біржі (DEX) дозволяють обмінювати криптовалюти без централізованого контролю.

Страхування. Смарт-контракти автоматизують виплати страхових полісів на основі попередньо визначених умов. Ланцюги Постачань. Смарт-контракти суттєво покращують логістичні процеси, забезпечуючи відстеження товарів у

режимі реального часу, автоматизацію платежів і контроль якості. Відстеження Товарів. Кожен етап постачання товару може бути записаний у блокчейн через смарт-контракт, що дозволяє відстежувати його місцезнаходження та стан. Автоматизація Платежів. Платежі можуть бути автоматизовані на основі фактичної доставки товарів або надання послуг. Контроль Якості. Смарт-контракти можуть відстежувати дотримання умов зберігання, наприклад, підтримання потрібної температури або вологості. Системи Голосування. Смарт-контракти дозволяють створювати децентралізовані системи електронного голосування, які забезпечують прозорість та захист від фальсифікації результатів. Електронне Голосування. Голосування за допомогою смарт-контрактів гарантує, що кожен голос буде зафіксований у блокчейні, а підробити чи змінити його неможливо. Організація DAO (Децентралізовані Автономні Організації). Управління організаціями через смарт-контракти дозволяє членам спільноти голосувати за зміни в протоколах, розподіл ресурсів або вибори керівних органів. NFT-Маркетплейси (Цифрові Активи). Невзаємозамінні токени (NFT) - це унікальні цифрові активи, що підтверджують право власності на цифровий контент, наприклад, зображення, музику чи відео. Смарт-контракти є основою NFT-ринків, автоматизуючи процеси купівлі-продажу.

Продаж та Аукціони. Смарт-контракти забезпечують проведення аукціонів у режимі реального часу, автоматично фіксуючи ставки та визначаючи переможця. Розподіл Авторських Винагород. Творці контенту можуть отримувати роялті з кожного перепродажу своїх NFT. Смарт-контракти автоматично нараховують відсоток від продажу творцю.

1.2 Мови програмування для смарт-контрактів

Розробка смарт-контрактів є ключовим елементом сучасної блокчейн-екосистеми, що відкриває нові можливості для створення децентралізованих додатків та автоматизації угод. Вибір правильної мови програмування для цих цілей має вирішальне значення, адже він впливає на безпеку, ефективність та функціональність контракту. Solidity залишається домінуючою мовою для

Ethereum, однак постійно з'являються нові підходи та інструменти, які розширюють спектр можливостей для розробників.

Solidity - це контрактно-орієнтована мова програмування, створена спеціально для написання смарт-контрактів на блокчейні Ethereum та інших сумісних блокчейнах (EVM). Вона поєднує елементи мов C++, Python і JavaScript, забезпечуючи гнучкість і функціональність у розробці децентралізованих додатків (dApps).

Solidity створена для розробки смарт-контрактів, які виконуються в середовищі Ethereum Virtual Machine (EVM), і її головне призначення полягає в автоматизації угод та збереженні даних у незмінному вигляді на блокчейні. Ця мова дозволяє розробляти контракти для фінансових додатків, управління цифровими активами, систем голосування та інших блокчейн-рішень, забезпечуючи при цьому захищену криптографічну взаємодію між учасниками мережі. Solidity широко використовується для реалізації децентралізованих організацій (DAO), NFT-проєктів і різноманітних сервісів, серед яких децентралізовані біржі, як-от Uniswap і SushiSwap, NFT-платформи OpenSea та Rarible, а також кредитні протоколи на зразок Aave і Compound.

Solidity має низку функціональних можливостей, які забезпечують ефективну роботу смарт-контрактів у мережі Ethereum. Контракти в Solidity - це основні будівельні блоки, які містять змінні стану, функції та події. Контракти визначають логіку взаємодії між учасниками мережі.

Приклад базового контракту:

```
// Простий контракт у Solidity
pragma solidity ^0.8.0;

contract SimpleContract {
    string public message;

    constructor(string memory initialMessage) {
        message = initialMessage;
    }

    function updateMessage(string memory newMessage) public {
        message = newMessage;
    }
```

```
}
```

Solidity підтримує функції та події як ключові елементи взаємодії в межах смарт-контракту. Функції визначають дії, які може виконувати контракт, і можуть мати різні рівні доступу - наприклад, бути публічними для зовнішніх викликів або приватними для внутрішнього використання. Події, у свою чергу, використовуються для фіксації важливих дій у блокчейні, що дає змогу зовнішнім додаткам або користувачам відслідковувати зміни та реагувати на них у режимі реального часу.

Приклад події:

```
event FundsDeposited(address indexed from, uint amount);
```

Модифікатори Доступу дозволяють контролювати доступ до функцій контракту. Наприклад, тільки власник контракту може виконати певні дії.

Приклад модифікатора доступу:

```
modifier onlyOwner() {
    require(msg.sender == owner, "You are not the owner!");
    _;
}
```

Solidity підтримує різноманітні структури даних, такі як масиви, мапи (словники) та структури (struct). Приклад структури даних:

```
struct User {
    string name;
    uint age;
}
```

Контракти можуть успадковувати властивості й функції від інших контрактів, підтримуючи принципи об'єктно-орієнтованого програмування. Бібліотеки дозволяють створювати загальні функції для повторного використання в інших контрактах.

Попри свої переваги, Solidity має низку недоліків, які обмежують її використання та створюють певні ризики для розробників. Solidity має досить складний синтаксис, що вимагає від розробників глибокого розуміння як програмування, так і принципів функціонування блокчейну. Навіть незначні помилки в коді можуть призводити до серйозних вразливостей, як це сталося з відомим контрактом The DAO, де помилка у використанні функцій зворотного виклику призвела до втрати мільйонів доларів. Додатковим ризиком є те, що після розгортання смарт-контракту на блокчейні його неможливо змінити, тому будь-які недоліки чи баги залишаються в коді назавжди. Серед найпоширеніших вразливостей - переповнення чисел, повторні виклики та втрата доступу до контракту. Крім того, інструменти для розробки, тестування та налагодження в екосистемі Solidity, хоча й активно розвиваються, залишаються доволі обмеженими або складними для новачків; серед них - Remix IDE, Truffle, Hardhat і TypeChain, які потребують певного досвіду для ефективного використання.

2 ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Огляд основних джерел

Дослідження можливостей використання TypeScript для розробки смарт-контрактів і їх компіляції в Solidity засвідчило перспективність цього напрямку, хоча наразі він перебуває на початковій та обмеженій стадії розвитку. Як зазначалось у роботі «Javascript Smart Contracts» від автора James Bachini [1], основна активність зосереджена навколо застосування TypeScript у ролі допоміжного інструмента для взаємодії з контрактами, вже написаними на Solidity. Такі фреймворки, як Hardhat, Truffle, а також бібліотеки на кшталт ethers.js і TypeChain, широко використовуються для створення скриптів розгортання, написання тестів і організації безпечної типізованої взаємодії між децентралізованими додатками й контрактами. Зокрема, як описано в роботі «Algorand Expands Developer Toolkit with TypeScript Support to Attract Web3 Developers» авторів Ashish Elisha та Jianglei Kahana [2], TypeChain забезпечує генерацію типів на основі ABI Solidity-контрактів, однак самі автори цього інструменту вже рекомендують перехід на новіші рішення, як-от Abitype, Wagmi та Viem, що пропонують сучасніший та інтегрований підхід. Крім екосистеми EVM, існують приклади успішного використання мов із синтаксисом, подібним до TypeScript, у межах інших блокчейнів: зокрема, Azle для Internet Computer Protocol або AssemblyScript для WASM-базованих платформ, таких як NEAR та Casper, а також DSL plu-ts на Cardano, що компілюється в Plutus Core. Ці приклади демонструють технічну реалізованість підходу, проте залишаються поза межами Ethereum-сумісних рішень. Серед експериментальних розробок, спрямованих безпосередньо на компіляцію TypeScript у Solidity або байткод EVM, були виявлені лише поодинокі згадки про проєкти на кшталт solx, які мають скоріше дослідницький або тестовий характер і не готові до використання у виробничому середовищі. Таким чином, хоча технічна можливість створення відповідного компілятора існує, наразі немає стабільного й широковживаного рішення для прямої компіляції TypeScript у Solidity.

2.2 Аналіз літератури

Аналіз наукових публікацій та дослідницьких робіт підтверджує загальну тенденцію, що пряма компіляція TypeScript до Solidity/EVM не є поширеною темою або існуючим зрілим рішенням у академічному середовищі. Натомість, література зосереджена на суміжних аспектах:

Дизайн компіляторів та проміжні представлення: багато робіт обговорюють загальні принципи побудови компіляторів, використання абстрактних синтаксичних дерев (AST) та проміжних представлень (IR) для оптимізації та генерації коду. Це є фундаментальним для розуміння того, як могла б працювати гіпотетична компіляція TypeScript до Solidity. Наприклад, згадки про plu-ts у роботі «Plu-ts Typescript smart-contracts - road to production» автора Justin Han [3] підкреслюють важливість IR для розширюваності мови та оптимізації.

Формальна верифікація та статичний аналіз смарт-контрактів є важливими напрямами досліджень, які спрямовані на підвищення надійності та безпеки децентралізованих застосунків. Академічна література приділяє значну увагу створенню інструментів, що дозволяють виявляти потенційні помилки ще до розгортання контрактів, зокрема через символьне виконання, статичний аналіз коду та формальні методи доведення правильності. Прикладами таких розробок є Solsmith - інструмент для генерації тестів і перевірки стабільності компілятора Solidity, а також DeepSEA - мова і компілятор, орієнтовані на формальну верифікацію смарт-контрактів. Подібні рішення відіграють ключову роль у контексті потенційної компіляції TypeScript у Solidity, оскільки така трансформація потребує гарантій збереження логіки та безпечної поведінки програмного коду, що реалізується саме через механізми верифікації та аналізу.

Транскомпіляція між мовами смарт-контрактів або компіляція високорівневих мов на інші блокчейн-віртуальні машини є темою окремих досліджень, які демонструють технічну можливість перетворення коду між різними мовними парадигмами. Наприклад, активно вивчаються підходи до компіляції з мови Rust у sBPF для платформи Solana, а також компіляція TypeScript-подібних мов у WebAssembly (WASM) для WASM-сумісних

блокчейнів, таких як NEAR або Casper. Подібні розробки підтверджують практичну реалізованість концепції трансформації коду для виконання на різних віртуальних машинах. Однак, попри технічну схожість, ці дослідження не зосереджуються на компіляції саме з TypeScript у Solidity або інші EVM-сумісні цілі, тому напрям TypeScript-Solidity залишається поза основною увагою академічної спільноти.

Проблеми та виклики, пов’язані з розробкою смарт-контрактів, часто розглядаються у науковій літературі, де підкреслюються такі аспекти, як наявність критичних вразливостей у коді, висока вартість виконання операцій (газу), складність процесу налагодження та загальна відсутність зрілої інфраструктури порівняно з традиційною Web2-розробкою. У цьому контексті TypeScript має потенціал значно покращити досвід розробників завдяки своїй сильній типізації, підтримці сучасних IDE та гнучкій екосистемі. Його використання в інструментах, що взаємодіють зі смарт-контрактами, дозволяє зменшити кількість помилок, полегшує тестування та забезпечує кращу інтеграцію між фронтендом і контрактною логікою, що особливо важливо в умовах зростання складності Web3-додатків.

У екосистемі Web3 спостерігається помітне зростання ролі TypeScript, зокрема в розробці децентралізованих додатків (dApps) та побудові інтерфейсів взаємодії з блокчейном. Така тенденція пояснюється численними перевагами TypeScript, серед яких - статична типізація, високий рівень безпеки коду, покращена інтеграція з сучасними середовищами розробки (IDE) та хороша масштабованість проєктів. Ці характеристики роблять мову особливо привабливою для великої кількості Web2-розробників, які поступово переходять у простір Web3. Яскравим прикладом цього руху є ініціативи на кшталт AlgoKit 3.0 для платформи Algorand, що вже дозволяють створювати «нативні» смарт-контракти за допомогою TypeScript, демонструючи напрямок розвитку інструментів, орієнтованих на зручність, безпеку та знайомі для широкого кола розробників підходи.

Наразі TypeScript є потужним інструментом для покращення розробницького досвіду навколо Solidity, забезпечуючи типову безпеку та ефективнішу взаємодію з контрактами. Однак, пряма компіляція TypeScript у Solidity не є усталеною практикою і знаходиться на стадії експериментів або взагалі не існує як зріле рішення. Майбутні дослідження, ймовірно, зосереджуватимуться на покращенні цієї взаємодії, розробці більш досконалих інструментів для статичного аналізу та верифікації, а також, можливо, на подальших спробах створення компіляторів.

2.2.1 Огляд мови Typescript

Як описано в роботах Johnathan Lexis «Tutorial: Using ethers.js and TypeScript to Read a Smart Contract | Stability Docs» [4] та «The Rise of TypeScript: Why It's the Future of JavaScript Development» [5] TypeScript - це надбудова над JavaScript, яка додає статичну типізацію та інші функціональні можливості, що значно покращують якість коду та процес розробки. TypeScript активно використовується в веб-розробці для створення як фронтенд, так і бекенд-додатків. Завдяки своїм перевагам у типобезпечності, об’єктно-орієнтованому програмуванні та потужній екосистемі, TypeScript стає все більш популярним інструментом у сфері блокчейн-розробки, включаючи створення смарт-контрактів та інтеграцію з блокчейн-мережами.

Типи змінних: TypeScript дозволяє чітко визначати типи змінних, що дозволяє уникнути типових помилок, таких як спроби виконати арифметичні операції над рядками або помилки в роботі з масивами чи об’єктами. Типи функцій: Визначення типів для параметрів та результатів функцій дозволяє отримати чітке уявлення про типи даних, з якими працює функція, та забезпечує кращу інтеграцію з іншими частинами програми. Це дозволяє значно зменшити кількість помилок, пов'язаних з неправильним використанням даних, що є важливим при розробці смарт-контрактів, де точність виконання операцій має критичне значення.

TypeScript активно підтримує ключові концепції об’єктно-орієнтованого програмування (ООП), що сприяє створенню структурованого, читабельного та масштабованого коду. Завдяки цим можливостям розробники можуть застосовувати перевірені патерни проєктування для побудови ефективних і підтримуваних програмних рішень. Зокрема, TypeScript дозволяє створювати класи з конструкторами, методами та властивостями, що дає змогу організовувати логіку у вигляді модулів і створювати об’єкти, які інкапсулюють поведінку та дані.

TypeScript забезпечує високий рівень зручності розробки завдяки типобезпечності та глибокій інтеграції з сучасними інструментами програмування. Підтримка автодоповнення в таких редакторах, як Visual Studio Code, дозволяє розробникам швидше писати код і зменшувати кількість помилок при роботі з API та сторонніми бібліотеками. Типізація в TypeScript сприяє ранньому виявленню помилок ще до виконання коду, що особливо важливо у сфері блокчейн-розробки, де навіть незначна помилка може призвести до серйозних фінансових або технічних наслідків. Завдяки цим можливостям процес створення смарт-контрактів і децентралізованих застосунків стає не лише швидшим, а й безпечнішим, що суттєво підвищує загальну продуктивність розробника.

2.2.2 Інтеграція TypeScript з блокчейном

Як зазначає у своїй роботі «A better way to parse solidity errors for typescript» [6] Jesper Newman, TypeScript має значний потенціал для інтеграції з блокчейн-технологіями завдяки широкому набору інструментів і бібліотек, які дозволяють ефективно взаємодіяти з різними блокчейн-мережами, зокрема Ethereum та іншими платформами для смарт-контрактів. Оскільки TypeScript має статичну типізацію та підтримує численні фреймворки і бібліотеки, його використання в розробці смарт-контрактів дозволяє значно покращити ефективність, безпеку та зручність процесу.

Одним із ключових аспектів використання TypeScript у блокчейн-розробці є інтеграція з бібліотеками, які забезпечують зручну взаємодію з блокчейн-мережами, зокрема Ethereum. Найпоширенішими інструментами в цій сфері є Ether.js та Web3.js - обидві бібліотеки надають розробникам інтерфейси для роботи зі смарт-контрактами, надсилання транзакцій та читання даних з блокчейну. Ether.js орієнтована на простоту, компактність і зручність використання, пропонуючи мінімалістичний, але потужний API для підключення до Ethereum та взаємодії з контрактами. TypeScript, завдяки своїй системі типів, суттєво підвищує безпечність і зручність роботи з цими бібліотеками, оскільки дозволяє IDE автоматично підказувати доступні методи, перевіряти правильність викликів та зменшувати кількість потенційних помилок під час розробки.

2.3 Оцінка актуальності та новизни

Актуальність теми використання TypeScript у контексті розробки смарт-контрактів зумовлена низкою важливих чинників. Насамперед, TypeScript продовжує стрімко зростати в популярності серед веб-розробників завдяки своїй типобезпечності, масштабованості, зручній інтеграції з сучасними IDE та повній сумісності з JavaScript [7]. Ця тенденція природно поширюється і на сферу Web3, де розробники шукають знайомі та ефективні інструменти. Крім того, екосистема Web3, попри активний розвиток, усе ще поступається Web2 за рівнем зрілості інструментів, особливо в частині тестування, налагодження та підтримки. У цьому контексті TypeScript допомагає суттєво покращити досвід розробника, забезпечуючи автодоповнення, перевірку помилок і статичну типізацію, що особливо важливо під час взаємодії з контрактами. Водночас, зростання складності сучасних смарт-контрактів та децентралізованих додатків вимагає більш надійних інструментів, і TypeScript у цьому випадку виступає як ключовий засіб підвищення безпеки та стабільності коду. Окрема увага заслуговує тенденція до наскрізного використання TypeScript на всіх рівнях стеку Web3 - від фронтенду та бекенду до інструментів роботи з контрактами, що сприяє узгодженості кодової бази, зменшує когнітивне навантаження та полегшує командну розробку.

Щодо новизни, проведене дослідження виявило кілька важливих аспектів, що підтверджують відкритість і перспективність цієї тематики для подальших досліджень та інновацій. Найпомітнішим відкриттям є відсутність зрілого та загальновизнаного компілятора, який би дозволяв напряму писати логіку смарт-контрактів на TypeScript із подальшою компіляцією в Solidity або байткод для EVM. Наявні рішення залишаються експериментальними або орієнтованими на внутрішнє тестування, а не на використання в реальному продакшн-середовищі, що свідчить про значний простір для технічного прогресу. Іншим важливим аспектом новизни є поступова еволюція інструментів, які забезпечують взаємодію TypeScript з контрактами Solidity. Перехід від TypeChain до більш сучасних і гнучких рішень, як-от Abitype, Wagmi та Viem, свідчить про зростання попиту на "рідні" для TypeScript підходи, які краще відповідають потребам розробників у сфері Web3. Окрему увагу заслуговує факт успішного використання TypeScript-подібних мов на інших, не EVM-сумісних блокчейнах - таких як Azle (для Internet Computer Protocol), AssemblyScript (для WASM-блокчейнів), або plu-ts (для Cardano). Це доводить технічну життєздатність концепції, хоча її адаптація до EVM усе ще залишається викликом. Нарешті, академічні публікації зосереджуються переважно на таких темах, як формальна верифікація, статичний аналіз та дизайн компіляторів для спеціалізованих мов смарт-контрактів, тоді як питання прямої компіляції TypeScript у Solidity практично не висвітлюється, що свідчить про наявність дослідницької ніші та потенціал для новаторських рішень у цій галузі [8].

Таким чином, актуальність використання TypeScript у сфері Web3 неухильно зростає, однак його новизна полягає не стільки в наявності прямого компілятора в Solidity (який наразі відсутній), скільки в активному розвитку інструментів для типобезпечної взаємодії з контрактами, появі «нативних» смарт-контрактів на основі TypeScript-подібних мов в альтернативних блокчейн-екосистемах, а також у значному потенціалі для подальших інновацій у напрямі крос-компіляції або транскомпіляції, якщо це буде визнано економічно доцільним і технічно реалізовуваним.

2.4 Узагальнення результатів огляду літератури

Огляд літератури та аналіз наявних інструментів чітко демонструють, що TypeScript займає важливе місце у сфері Web3-розробки завдяки своїм перевагам у зручності, безпеці та інтеграції з сучасними інструментами. Проте його роль у створенні смарт-контрактів для Ethereum Virtual Machine (EVM) виявляється дещо іншою, ніж могло здаватися на перший погляд: замість прямого написання контрактів TypeScript переважно використовується для побудови інфраструктури навколо них, забезпечуючи ефективну взаємодію, типізацію та автоматизацію процесів розробки.

TypeScript на сьогоднішній день відіграє ключову роль у Web3-розробці, однак його призначення полягає не в написанні або компіляції смарт-контрактів, а у побудові надійної інфраструктури навколо них [9]. Прямий, зрілий та широко прийнятий компілятор, який би трансформував TypeScript-код у Solidity або EVM-байткод, наразі не існує - всі відомі проєкти є експериментальними або обмежені за сферою застосування. Натомість TypeScript широко використовується для типобезпечної взаємодії з контрактами, написання скриптів розгортання, тестів, конфігурацій та побудови фронтенду dApps. Саме тут інструменти на кшталт Hardhat, Truffle, ethers.js, TypeChain, а також новіші рішення, як Abitype, Wagmi та Viem, демонструють свою ефективність і активно розвиваються. Дослідження також показало, що технічно можливим є використання TypeScript-подібних мов для написання смарт-контрактів на інших, не EVM-сумісних блокчейнах - прикладами є Azle (для Internet Computer Protocol), AssemblyScript (для WASM-блокчейнів) та plu-ts (для Cardano) [10]. Це свідчить про реальність концепції транскомпіляції TS-подібних мов у блокчейн-байткод за наявності відповідної інфраструктури. Подальші покращення інструментів концентруються не на заміні Solidity, а на вдосконаленні розробницького досвіду (DX): автодоповнення, типова безпека, зручне налагодження та підтримка сучасних IDE. Проблеми, з якими стикаються розробники, включають необхідність постійного перемикання між TypeScript і Solidity, складність підтримки відповідності ABI та загальну межу між двома

мовами. Академічні дослідження, у свою чергу, фокусуються на формальних аспектах - безпеці, верифікації, аналізі коду та компіляторному дизайні - але рідко зачіпають тему прямої компіляції TypeScript у Solidity, що свідчить або про складність цієї проблеми, або про її відносно низький пріоритет. Водночас спостерігається чіткий тренд на “наскрізне” використання TypeScript на всіх рівнях dApps - від фронтенду до взаємодії з контрактами - що робить мову надзвичайно привабливою для широкого кола Web2-розробників, які переходять у Web3. Це формує сталий попит на розвиток зручних, сумісних і типобезпечних інструментів у цьому напрямі.

Ці висновки впливають на обрану тему дослідження, підтверджуючи доцільність компіляції TypeScript у Solidity, адже TypeScript відіграє важливу роль у сучасній Web3-розробці, виступаючи не як альтернатива Solidity, а як потужний інструментальний засіб, що доповнює його на рівні взаємодії, тестування та автоматизації. Його основна цінність полягає в покращенні досвіду розробника завдяки типовій безпеці, автодоповненню та інтеграції з сучасними середовищами розробки. У перспективі майбутні дослідження та розробки, найімовірніше, зосередяться на ще глибшій інтеграції TypeScript у стек Web3, удосконаленні механізмів типобезпечної взаємодії з контрактами, а також на розвитку нових підходів до створення «нативних» смарт-контрактів на TypeScript-подібних мовах для альтернативних блокчейн-платформ.

3 ПОСТАНОВКА ЗАДАЧІ

Наукова робота присвячена дослідженню потенціалу інтеграції мови програмування TypeScript у процес розробки смарт-контрактів, із особливою увагою до можливості компіляції в Solidity та вдосконалення супутніх інструментів розробки. Актуальність цього дослідження обумовлена зростанням складності сучасних децентралізованих додатків (dApps) і контрактів, а також прагненням розробників використовувати типобезпечні, знайомі та ефективні інструменти з Web2-екосистеми. Завдяки своїй здатності підвищувати надійність коду, покращувати розробницький досвід і підтримувати масштабованість проєктів, TypeScript розглядається як перспективний компонент у сфері блокчейн-розробки.

Хоча сьогодні TypeScript активно застосовується для створення інфраструктури навколо Solidity-контрактів - зокрема для написання тестів, скриптів розгортання, конфігурацій та клієнтської логіки - ідея прямої компіляції повноцінних смарт-контрактів, написаних на TypeScript, у Solidity або байткод для EVM поки що не реалізована у вигляді зрілого й загальновизнаного рішення. Це створює певний функціональний розрив, змушуючи розробників працювати паралельно з двома різними мовами, що ускладнює розробку, збільшує ризик помилок та підвищує когнітивне навантаження.

Ця робота спрямована не лише на аналіз поточного стану інтеграції TypeScript у процес розробки смарт-контрактів, але й на окреслення перспективних напрямків для майбутніх досліджень і розробок, які можуть сприяти підвищенню доступності, безпеки та ефективності цього процесу. Особливу увагу приділено тому, як можна адаптувати знайомі для багатьох Web2-розробників інструменти та підходи до специфіки Web3, з метою зменшення порогу входу та уніфікації розробницького стеку навколо TypeScript. Це відкриває потенціал для створення нових рішень - зокрема компіляторів, проміжних представлень або фреймворків - які дозволять реалізовувати смарт-контракти засобами TypeScript або з мінімальним перемиканням контексту між мовами.

3.1 Використані технології та інструменти

Для реалізації проєкту буде використано сучасні програмні засоби. Основними інструментами стануть платформа для запуску JavaScript або TypeScript додатків на персональному комп'ютері Node.js, бібліотеки child_process, вбудована в Node.js, reflect-metadata та ts-morph, які можна встановити через пакетний менеджер для Node.js - npm.

Node.js - це платформа для розробки серверних застосунків, яка дозволяє виконувати JavaScript поза браузером. Вона побудована на рушії V8 від Google, що забезпечує високу продуктивність. Node.js відзначається неблокуючою, асинхронною архітектурою, що робить її особливо ефективною для роботи з мережевими запитами та реального часу додатками, як-от чати чи онлайн-ігри.

Бібліотека reflect-metadata для Node.js забезпечує підтримку метаданих у JavaScript і TypeScript, що особливо корисно при роботі з декораторами. Вона дозволяє зберігати, зчитувати та змінювати метаінформацію про класи, методи чи властивості. Ця бібліотека активно використовується в фреймворках на зразок NestJS для реалізації інверсії керування, валідації та автоматичного створення API.

Бібліотека ts-morph - це потужний інструмент для роботи з TypeScript AST (абстрактним синтаксичним деревом), який спрощує аналіз і трансформацію коду. Вона дозволяє програмно читати, змінювати, створювати та зберігати TypeScript- або JavaScript-файли. ts-morph широко використовується для автоматичної генерації коду, рефакторингу та побудови власних інструментів аналізу коду.

Бібліотека child_process є вбудованим модулем у Node.js, що дозволяє запускати зовнішні процеси з поточної Node.js-програми. Вона надає можливості для виконання команд оболонки, створення нових процесів, а також обміну даними між процесами через потоки. child_process підтримує як прості виклики через exec, так і більш контрольовані - через spawn або fork, що корисно для обробки великих обсягів даних або запуску окремих Node.js-скриптів.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

Теоретичне дослідження було зосереджене на вивченні глибинних аспектів компіляції мов програмування та їхнього застосування в контексті блокчейн-технологій, з особливим акцентом на можливості використання TypeScript для розробки смарт-контрактів. У рамках цього дослідження було проаналізовано фундаментальні принципи побудови компіляторів, включаючи побудову абстрактних синтаксичних дерев, проміжних представлень (IR) та процес генерації коду, а також розглянуто приклади їх реалізації в сучасних блокчейн-системах. Це дозволило глибше зрозуміти технічні обмеження та потенціал створення нових інструментів для інтеграції TypeScript у процес компіляції до EVM-сумісного формату.

4.1 Основні концепції компіляторобудування

Дослідження заглибилось у класичні етапи розробки компіляторів: лексичний аналіз (сканування) - процес перетворення вхідного вихідного коду (TypeScript) на потік токенів (лексем); синтаксичний аналіз (парсинг) - побудова абстрактного синтаксичного дерева (AST) з потоку токенів, що представляє ієрархічну структуру програми; семантичний аналіз - перевірка відповідності програми правилам мови (перевірка типів, області видимості змінних тощо), і саме тут TypeScript демонструє свої сильні сторони, забезпечуючи типову безпеку вже на етапі розробки; генерація проміжного представлення (IR) - створення незалежного від цільової платформи представлення коду, що є критично важливим етапом для транскомпіляції між різними мовами або для компіляції на різні віртуальні машини, наприклад, згадки про plu-ts для Cardano та його “проміжне представлення (IR)” підкреслюють важливість цього етапу для оптимізації та розширюваності мови; оптимізація коду - застосування різних технік для покращення продуктивності, ефективності та розміру згенерованого коду, що у контексті смарт-контрактів безпосередньо впливає на вартість газу та швидкість виконання; генерація цільового коду - перетворення оптимізованого IR на машиноорієнтований код для конкретної віртуальної машини, такої як EVM-

байткод, WebAssembly для WASM-базованих блокчейнів або Plutus Core для Cardano.

4.1.1 Застосування у контексті блокчейну та смарт-контрактів

Теоретичне дослідження виявило ключові аспекти застосування компіляторних концепцій у блокчейн-середовищі, зокрема специфіку компіляції для блокчейну, яка передбачає врахування газових обмежень, адже кожна операція у смарт-контракті має вартість, що вимагає високого рівня оптимізації для мінімізації споживання газу.

Крім того, незмінність і недоторканність розгорнутих контрактів ставить високі вимоги до коректності компіляції, оскільки помилки на цьому етапі можуть призвести до незворотних вразливостей. Важливо також враховувати різноманітність віртуальних машин, таких як EVM, WASM, Algorand AVM чи Plutus Core, кожна з яких має власну архітектуру та набір інструкцій, що вимагає розробки окремих бекендів компіляторів або спеціалізованих транскомпіляторів для кожної цільової VM. У цьому контексті особливої ролі набувають проміжні мови, такі як Yul для EVM чи LLVM IR для компіляції Rust у sBPF для Solana, оскільки вони дозволяють уніфікувати процес оптимізації та спростити таргетування на різні архітектури; теоретично, TypeScript також міг би бути скомпільований у таку проміжну мову з подальшою трансляцією у цільовий байт-код. Через підвищені вимоги до безпеки смарт-контрактів академічна спільнота активно досліджує методи формальної верифікації як самих контрактів, так і компіляторів, що їх обробляють, прикладом чого є DeepSEA - мова та компілятор, орієнтовані на доведення коректності перетворення коду за допомогою формальних методів.

4.1.2 Теоретична прогалина для TypeScript та Solidity

Найважливіший висновок теоретичного дослідження полягає в тому, що, попри загальну технічну здійсненність транскомпіляції мов і наявність успішних прикладів використання TypeScript-подібних мов для інших блокчейн-платформ,

як теоретична, так і практична база для прямої, зрілої та широко підтримуваної компіляції TypeScript у Solidity або EVM-байткод залишається фактично нерозвиненою або відсутньою. Це вказує на наявність суттєвих викликів, серед яких особливо вирізняється складність семантичного розриву між мовами: TypeScript є динамічною, JavaScript-подібною мовою загального призначення, тоді як Solidity - спеціалізована мова з суворою семантикою, орієнтованою на виконання в середовищі EVM. Така відмінність у моделях виконання, управлінні пам'яттю, безпеці та життєвому циклі даних значно ускладнює безпечну й ефективну трансляцію однієї мови в іншу.

Ще однією перепоною є відсутність єдиного й ефективного проміжного представлення, яке б могло посередньо пов'язати TypeScript і EVM, спрощуючи побудову транскомпілятора. У цьому контексті цілком логічним виглядає поточний фокус розробників на вдосконаленні інструментів для взаємодії з контрактами, написаними на Solidity, що на практиці виявляється більш доцільним, швидким і економічно виправданим шляхом, ніж розробка повноцінного компілятора з TypeScript у Solidity.

4.1.3 Ключові виклики та семантичний розрив

Різниця в парадигмах між TypeScript і Solidity є однією з ключових перешкод для їх прямої взаємодії. TypeScript - це мова, орієнтована переважно на клієнтську та серверну логіку, заснована на JavaScript, яка підтримує динамічні конструкції, багаті типи даних, гнучкість у синтаксисі та розвинуту інфраструктуру для веброзробки.

Натомість Solidity - спеціалізована мова для написання смарт-контрактів у середовищі Ethereum Virtual Machine (EVM), яка вимагає детермінованого управління станом, явного врахування вартості операцій (газу), обмежена у підтримці чисел з плаваючою комою та інших складних типів. Це створює значний семантичний розрив між мовами, ускладнюючи транскомпіляцію. Додатково, EVM - це стекова віртуальна машина з чіткими обмеженнями, і будь-яка мова, яка транслюється під цю архітектуру, повинна враховувати її специфіку,

зокрема обмеження по пам’яті, типах та енерговитратності виконання. Окремо варто згадати відсутність у Solidity підтримки типів, звичних для TypeScript, таких як числа з плаваючою комою або складні структури, що потребує використання сторонніх бібліотек або реалізації складних обхідних рішень.

Безпека також залишається критичним аспектом: у Solidity вона закладена на рівні мови, а будь-яка транскомпіляція з TypeScript повинна гарантувати, що кінцевий Solidity-код не містить вразливостей і придатний для формальної верифікації. Нарешті, налагодження залишається важкою задачею - якщо транскомпіляція не зможе забезпечити зворотне трасування помилок до вихідного TypeScript-коду, це суттєво обмежить її практичність, оскільки відстеження проблем у згенерованому Solidity-коді є набагато складнішим і менш зручним, ніж у звичних для розробника середовищах.

4.1.4 Потенційні рішення та роль проміжних представлень (IR)

Використання проміжних представлень (IR) є ключовим елементом для транскомпіляції між різними мовами програмування, оскільки дозволяє абстрагуватися від конкретного синтаксису вихідної мови, зосередившись на її семантиці. У контексті компіляції TypeScript у Solidity, створення проміжного представлення, яке б ефективно відображало особливості обох мов, відкриває перспективу зменшення семантичного розриву та полегшення оптимізації під обмеження EVM. Існуючі приклади використання IR у проєктах, що компілюють TypeScript-подібні мови для інших блокчейн-платформ, підтверджують ефективність цього підходу. Одним із потенційних рішень є створення спеціалізованого IR, орієнтованого саме на TypeScript-to-Solidity транскомпіляцію, здатного зберігати достатню гнучкість і точність відображення логіки вхідного коду, при цьому залишаючись сумісним із обмеженнями віртуальної машини Ethereum. Іншою альтернативою може бути адаптація вже наявного IR, зокрема LLVM IR, який підтримує багато мов і має потужну інфраструктуру для оптимізації; однак така інтеграція потребуватиме глибокого

переосмислення з урахуванням специфіки Solidity, яка суттєво відрізняється від типових цільових платформ LLVM.

Також можливим є обмежений підхід до транскомпіляції, коли підтримується лише підмножина синтаксису та функціоналу TypeScript, сумісна з парадигмами Solidity - це знижує складність реалізації, але обмежує гнучкість мови. У будь-якому випадку перспективним напрямом є розробка інструментів і бібліотек, які б абстрагували взаємодію з низькорівневими конструкціями Solidity, дозволяючи розробникам писати високорівневий TypeScript-код із підтримкою типів, шаблонів та повторного використання, а система транскомпіляції - ефективно транслювала його у безпечний, оптимізований Solidity-код.

4.2 Паралельна компіляція TypeScript та Solidity

Паралельна компіляція TypeScript та Solidity, згідно з результатами дослідження, має особливе значення в контексті сучасної Web3-розробки, оскільки не йдеться про пряме перетворення TypeScript у Solidity, а про незалежне, але взаємозалежне використання двох окремих компіляторів у межах одного проєкту. Смарт-контракти, написані на Solidity, компілюються через офіційний компілятор solc або його JavaScript-версію solc-js у байткод EVM, тоді як TypeScript-код, що відповідає за бізнес-логіку dApp, тести та інфраструктурні скрипти, компілюється у JavaScript за допомогою компілятора tsc [11].

Паралельність полягає в тому, що обидва процеси компіляції виконуються незалежно, але в межах єдиної екосистеми та спільного проєкту, де файли .sol та .ts обробляються окремо. Сучасні фреймворки, як Hardhat чи Truffle, підтримують одночасну інтеграцію обох процесів: компілюють контракти, генерують ABI та байткод, а далі, з допомогою таких інструментів, як TypeChain або його сучасні альтернативи (Abitype, Wagmi, Viem), автоматично створюють типізацію в TypeScript, яка дозволяє коду dApp безпечно та зручно взаємодіяти з контрактами. Завдяки цьому розробники можуть створювати тести, скрипти розгортання та клієнтську логіку повністю на TypeScript, маючи при цьому повну синхронізацію з контрактами через типи.

Такий підхід має низку переваг: чітке розділення відповідальності між мовами, використання зрілих та перевірених інструментів для обох середовищ, а також покращення досвіду розробника завдяки автодоповненню, перевірці типів і налагодженню в IDE. Водночас, існують і певні обмеження: розробники змушені працювати з двома мовами та екосистемами одночасно, що підвищує когнітивне навантаження; існує ризик розбіжностей між контрактами Solidity та типізацією в TypeScript, якщо не дотримано суворої синхронізації; налагодження помилок на межі між двома мовами є складнішим. Таким чином, “паралельна компіляція” не означає автоматичну трансляцію між мовами, а радше описує сучасний, ефективний підхід до розробки Web3-додатків, де обидві мовні екосистеми працюють синхронно завдяки допоміжним інструментам, що забезпечують типобезпечну та узгоджену взаємодію.

Попри ці труднощі, переваги такого підходу значно переважають недоліки, особливо в контексті складних, багатокомпонентних проєктів. Завдяки чіткому розмежуванню сфер відповідальності та автоматизованій генерації типів із ABI, розробники можуть уникнути великої кількості класичних помилок, пов’язаних із ручною інтеграцією. Крім того, це відкриває можливості для впровадження CI/CD-процесів, покращеного тестування та модульної побудови архітектури, що є критично важливими для створення надійних і масштабованих рішень у сфері Web3.

У таблиці 1, що наведено нижче, не йдеться про «розпаралеленість» чи «теоретичне прискорення» в контексті закону Амдала, оскільки це не приклад обчислювальної задачі, що піддається класичному паралелізму. Замість цього, «рівні переваг» ілюструють якісний вплив застосування TypeScript у Web3-розробці порівняно з підходами, що базуються виключно на JavaScript для логіки dApp.

Компіляція TypeScript отримує вищі оцінки у сферах зниження кількості помилок, швидкості розробки та зручності рефакторингу - саме ці характеристики є ключовими перевагами мови.

Таблиця 1 – Якісні Показники Переваг від паралельної Компіляції TypeScript та Solidity (таблиця виконана самостійно)

Аспект Переваги	TypeScript-компіляція (dApp-логіка)	Solidity-компіляція (Контракт)	Вплив на Загальний Процес Розробки
Зменшення кількості помилок	4.5	3.0	4.0 (завдяки типобезпеці та інтеграції)
Швидкість розробки (DX)	4.8	2.5	4.2 (покращена навігація, автодоповнення)
Легкість рефакторингу	4.7	2.0	3.5 (завдяки системі типів)
Масштабованість кодової бази	4.6	3.0	4.0 (для великих команд та проєктів)
Простота тестування	4.0	3.5	3.8 (зручні фреймворки та типи)
Оптимізація газу	N/A (не впливає)	4.0	4.0 (Solidity-компілятор оптимізує)

У свою чергу, компіляція Solidity вирізняється високими оцінками за ефективність використання газу, оскільки вона напряму впливає на продуктивність смарт-контрактів у середовищі EVM. «Вплив на загальний процес розробки» є середньою оцінкою, що відображає синергію між двома мовами та їх компіляційними підходами. Варто зауважити, що наведені показники є умовними та служать лише для ілюстрації якісних переваг.

4.3 Масштабованість підходів

Аналіз масштабованості підходів у контексті використання TypeScript для розробки смарт-контрактів охоплює два ключові виміри: масштабованість

розробки та масштабованість виконання контрактів. У першому аспекті TypeScript демонструє значні переваги, які особливо важливі для великих Web3-проєктів. Сильна типізація дозволяє уникати помилок на ранніх етапах, спрощує рефакторинг та підвищує загальну надійність системи. Завдяки автоматичній генерації типів з ABI-контрактів через інструменти на кшталт TypeChain, Abitype, Wagmi або Viem, взаємодія між фронтендом і контрактами стає передбачуваною та типобезпечною.

Інтеграція з сучасними IDE забезпечує зручність і швидкість розробки, а модульна структура TypeScript сприяє організованій архітектурі коду, що особливо цінно у великих командах. До того ж, використання TypeScript знижує поріг входу для Web2-розробників, розширюючи доступний кадровий потенціал. У другому аспекті - масштабованості виконання смарт-контрактів - TypeScript не має прямого впливу, оскільки не компілюється в Solidity або EVM-байткод.

Проте в гіпотетичному сценарії, де існує прямий компілятор TypeScript у Solidity, ефективність згенерованого коду мала б критичне значення: від цього залежали б витрати газу, розмір контракту й загальна продуктивність. Досвід інших платформ, де TypeScript-подібні мови компілюються в WASM або Plutus Core, підтверджує, що масштабованість на рівні виконання досяжна за наявності високоякісного IR і оптимізатора. Хоча TypeScript наразі не впливає безпосередньо на продуктивність контрактів, він покращує якість dApp-рівня, зменшуючи ймовірність невдалих транзакцій і полегшуючи зосередження зусиль на оптимізації Solidity-коду.

У підсумку, TypeScript уже сьогодні є потужним інструментом для масштабування процесу Web3-розробки завдяки типобезпеці, організації коду, командній ефективності та зручності підтримки. Його потенційний внесок у масштабованість виконання смарт-контрактів залежить від майбутнього створення оптимізованого компілятора, здатного генерувати ефективний, безпечний і економний для EVM-контрактний код.

4.4 Проведення теоретичного дослідження

Теоретичне дослідження зосереджувалося на глибокому вивченні потенціалу та обмежень інтеграції TypeScript у розробку смарт-контрактів, із особливою увагою до його взаємодії з мовою Solidity та віртуальною машиною Ethereum (EVM). Основною метою було з'ясувати ключові принципи, що сприяють або ускладнюють пряму компіляцію TypeScript у Solidity, а також проаналізувати наявні інструменти та методи, що застосовуються в цьому контексті.

Дослідження охопило такі ключові аспекти: ми розглянули класичні етапи компіляції: лексичний аналіз, синтаксичний аналіз (побудова AST), семантичний аналіз (особливо важливий для типобезпечного TypeScript), генерація проміжного представлення (IR), оптимізація коду та генерація цільового коду. Особлива увага приділялася ролі проміжних представлень (IR) як "моста" між вихідною та цільовою мовами. Це дозволяє абстрагуватися від специфіки мов і проводити оптимізації на проміжному рівні.

Теоретичне дослідження підкреслило унікальні обмеження та вимоги блокчейн-середовища, зокрема специфіку компіляції для блокчейну та EVM. EVM є стековою віртуальною машиною з суворими обмеженнями щодо споживання газу, розміру коду та доступних операцій. Смарт-контракти після розгортання не можуть бути змінені, що робить вимоги до коректності компілятора та відсутності вразливостей надзвичайно високими. Крім того, будь-який компілятор для блокчейну повинен мати сильні оптимізації для мінімізації витрат газу, що є критично важливим для масштабованості та економічної доцільності.

Дослідження показало, що наразі не існує прямої компіляції TypeScript до Solidity, натомість спостерігається паралельна робота двох окремих компіляційних процесів у межах одного Web3-проєкту. Контракти, написані на Solidity, компілюються за допомогою solc у EVM-байткод, тоді як логіка децентралізованого застосунку, тести та скрипти на TypeScript компілюються компілятором tsc у JavaScript. Ключову роль у забезпеченні ефективної взаємодії

між цими складовими відіграють інструменти, такі як Hardhat, Truffle, ethers.js, а також TypeChain та його наступники Abitype, Wagmi, Viem, які автоматично генерують TypeScript-типи з ABI Solidity-контрактів, що дозволяє досягти типобезпечної взаємодії між TypeScript-кодом та контрактами на Solidity.

Найважливішим висновком дослідження стало виявлення суттєвої теоретичної прогалини, що ускладнює реалізацію прямої та зрілої компіляції TypeScript у Solidity. Головною проблемою є глибокий семантичний розрив між мовами: їхні моделі виконання, принципи управління пам'яттю та загальна філософія значно відрізняються, що унеможливлює просте відображення конструкцій TypeScript у терміни Solidity. До цього додаються обмеження самої EVM, зокрема відсутність підтримки чисел з плаваючою комою та специфіка стекової архітектури, які не дозволяють напряму транслювати гнучкі можливості TypeScript. Водночас наявність ефективного проміжного представлення, яке могло б виступати посередником між цими мовами, наразі відсутня. Нарешті, надзвичайно високі вимоги до безпеки смарт-контрактів означають, що будь-який такий компілятор має бути формально верифікований, що є складним і ресурсоємним завданням.

Гіпотетично подолати ідентифіковану теоретичну прогалину можна кількома шляхами. Один з них - створення спеціалізованого проміжного представлення (IR), яке могло б адекватно відображати семантику як TypeScript, так і Solidity. Альтернативою є адаптація вже існуючих IR, зокрема LLVM IR, хоча це потребуватиме значних технічних зусиль для забезпечення сумісності з архітектурою EVM. Інший підхід - обмежити транскомпіляцію лише до певної підмножини конструкцій TypeScript, які легше транслюються у Solidity. Нарешті, можливим варіантом є створення спеціалізованої предметно-орієнтованої мови (DSL) у межах TypeScript, яка б дозволяла писати код у знайомому середовищі, але фактично генерувала Solidity-контракти.

Теоретичне дослідження засвідчило, що хоча TypeScript значно сприяє масштабованості розробки Web3-додатків та покращує досвід розробника, пряма компіляція цієї мови в Solidity для EVM залишається складною та наразі не має

зрілого практичного втілення. Основні перепони - це глибокі відмінності у семантиці мов та обмеження, властиві EVM. Найбільш доцільним і поширеним підходом на сьогодні є “паралельна” модель розробки, за якої компіляція TypeScript та Solidity здійснюється окремо, а взаємодія між ними забезпечується через типізовані інтерфейси.

4.5 Компіляція TypeScript в Solidity за допомогою Node.js та ts-morph

Node.js виступає як середовище виконання, що забезпечує запуск скриптів для транскомпіляції, доступ до файлової системи та інтеграцію зі стандартними бібліотеками. У свою чергу, ts-morph виконує роль інструменту для роботи з абстрактним синтаксичним деревом (AST) TypeScript-коду. За допомогою ts-morph можна здійснювати парсинг TS-файлів у AST, аналізувати структуру коду, перевіряти типи, знаходити потрібні конструкції, а також реалізувати власні алгоритми трансформації. Після цього можлива генерація Solidity-коду - або за допомогою ts-morph, або шляхом обробки рядків - для створення .sol-файлів.

Теоретичний процес компіляції передбачає такий ланцюжок: початково ви створюєте смарт-контракт на TypeScript із дотриманням певних обмежень, оскільки не всі конструкції цієї мови мають відповідники в Solidity. Потім за допомогою скрипта на Node.js у поєднанні з ts-morph відбувається завантаження та парсинг TS-файлів у AST, далі виконується семантичний аналіз і трансляція - найскладніший етап, у якому потрібно узгодити відмінності між TypeScript і Solidity/EVM. Після цього генерується синтаксично коректний Solidity-код у вигляді .sol-файлів. На наступному етапі компілятор Solidity (solc) перетворює ці файли у байткод для EVM, який і є фінальним результатом, готовим до розгортання. На рисунку 1 зображений весь процес у вигляді діаграми.

У цьому контексті “паралельна компіляція” стосується не внутрішньої багатопоточності одного компілятора, а скоріше оптимізації всього процесу збірки за рахунок асинхронної або одночасної обробки різних етапів. Наприклад, node.js може паралельно обробляти кілька TypeScript-файлів, генеруючи Solidity-код, тоді як інші вже згенеровані .sol-файли одночасно компілюються solc. Також

доцільним є кешування оброблених файлів, що не змінились, для уникнення повторної генерації. Це дозволяє ефективніше використовувати ресурси й пришвидшити весь конвеєр збірки (див.рис.2).

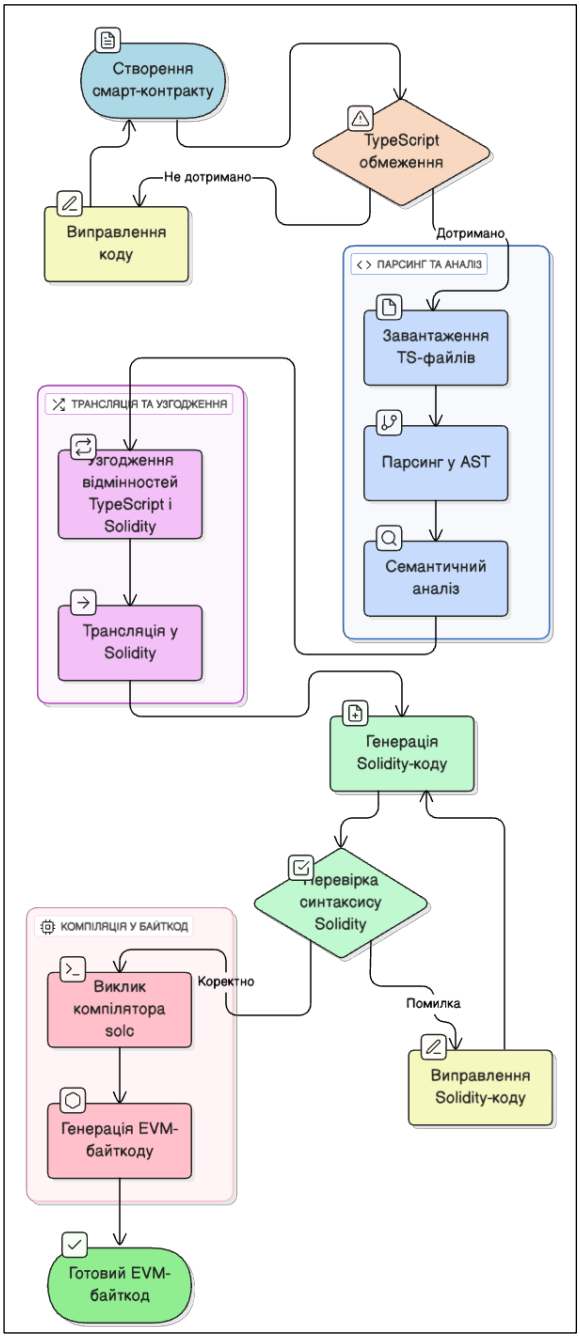

Рисунок 2 – Процес компіляції TypeScript в Solidity за допомогою Node.js та ts-morph (рисунок виконаний самостійно)

Оскільки цей підхід має теоретичний характер і наразі не підкріплений усталеними бенчмарками, усі числові показники в таблицях є умовними. Вони

мають якісний характер і слугують для ілюстрації очікуваної складності та потенційних переваг (див.таб.2).

Таблиця 2 – Оцінка складності етапів транскомпіляції TypeScipt в Solidity за допомогою ts-morph (таблиця виконана самостійно)

Етап / Задача	Складність Розробки	Обчислювальна Складність	Пояснення
Парсинг TS-коду (ts-morph)	1.0	1.5	ts-morph надає готовий API для парсингу.
Семантичний Аналіз та трансляція логіки	5.0	4.0	Визначення відповідності TS-конструкцій Solidity.
Генерація Solidity-коду (текст)	3.0	2.0	Після визначення відповідності, генерація синтаксично правильного Solidity-коду.
Інтеграція з solc	2.0	1.0	Виклик solc через node.js API.

Основний висновок полягає в тому, що TypeScript і Solidity мають кардинально різні процеси компіляції: TypeScript транспілюється в JavaScript, а Solidity - у байт-код Ethereum Virtual Machine (EVM). Відтак, термін “паралельна компіляція” в цьому випадку стосується не синхронної трансляції обох мов, а загальної оптимізації процесу розробки, де ці мови використовуються разом. Сучасні Web3-фреймворки, як-от Hardhat, Foundry та Truffle, значно спрощують цю інтеграцію, забезпечуючи підтримку інкрементної компіляції, кешування та паралельного запуску тестів і скриптів.

5 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ

5.1Інтеграція TypeScript у Розробку Смарт-Контрактів

Практичне дослідження було спрямоване на аналіз реальних сценаріїв використання та інструментів, які сприяють інтеграції TypeScript у процес розробки смарт-контрактів, з особливою увагою до середовища Ethereum Virtual Machine (EVM). Через відсутність зрілих рішень для прямої компіляції TypeScript у Solidity основна увага приділялася інструментам взаємодії та паралельній організації компіляційних процесів, які наразі є переважаючим підходом у практиці Web3-розробки.

5.1.1 Роль TypeScript в Екосистемі Web3

Роль TypeScript в екосистемі Web3 виявилася надзвичайно важливою, що підтверджується результатами практичного аналізу. TypeScript фактично став стандартом для створення децентралізованих додатків. Він широко використовується у фронтенд-розробці (зокрема з фреймворками React, Vue, Angular) завдяки своїй типобезпеці та покращеному досвіду розробника. У бекенд- або серверних компонентах, таких як індексатори, оракули або off-chain сервіси, TypeScript застосовується разом із Node.js. Крім того, він активно використовується для написання скриптів розгортання та тестування смарт-контрактів, що робить його ключовим інструментом у всіх етапах розробки Web3-проєктів.

Практичне дослідження виявило кілька основних категорій інструментів, які забезпечують ефективну інтеграцію між TypeScript та Solidity. Однією з ключових є фреймворки для розробки смарт-контрактів, серед яких найбільш популярним є Hardhat - він має вбудовану підтримку TypeScript, дозволяє писати тести та скрипти, інтегрується з компілятором Solidity і запускає контракти в локальній EVM. Truffle, хоч і орієнтований спочатку на JavaScript, також пропонує підтримку TypeScript.

Іншою важливою категорією є генератори типів з ABI. Найбільш відомий - TypeChain, який створює TypeScript-типи з ABI контрактів, забезпечуючи

типобезпечну взаємодію. Його поступово замінюють сучасні інструменти, такі як Abitype, Wagmi та Viem, які ще краще інтегруються з React та системою типів TypeScript.

Також активно використовуються бібліотеки для взаємодії з блокчейном. Серед них найбільш популярною є ethers.js, яка надає повноцінну підтримку TypeScript і дозволяє безпечно працювати з контрактами. Альтернативно використовується web3.js, яка також має типізовану версію для TypeScript.

На практиці поняття “паралельна компіляція” означає, що компілятор Solidity (solc) обробляє .sol файли, створюючи байт-код і ABI, а компілятор TypeScript (tsc) обробляє .ts файли, які відповідають за логіку dApp, тести та скрипти, генеруючи JavaScript. Ці процеси координуються інструментами, такими як Hardhat, які поетапно запускають обидві компіляції та інтегрують генерацію типів за допомогою TypeChain або Abitype для з’єднання обох частин проєкту. У результаті розробники можуть використовувати переваги TypeScript - типобезпеку, автодоповнення та зручний рефакторинг - при взаємодії з контрактами, зберігаючи водночас сильні сторони Solidity, зокрема його безпеку, зрілість і оптимізацію для EVM.

Практичне дослідження показало, що використання TypeScript-подібних мов для створення смарт-контрактів можливе й успішно реалізується на інших блокчейн-платформах, які не є сумісними з EVM. Наприклад, Azle для Internet Computer Protocol дозволяє створювати контракти на TypeScript, який компілюється у WebAssembly. Подібно, AssemblyScript застосовується на WASM-орієнтованих блокчейнах, таких як Casper і NEAR, а Plu-ts на Cardano надає можливість писати контракти через DSL на TypeScript, який потім перетворюється в Plutus Core. Ці приклади підтверджують, що компіляція TypeScript-подібного синтаксису у блокчейн-байткод є цілком реальною - хоча й не для EVM.

Однак для екосистеми Ethereum наразі не існує практичного і зрілого компілятора TypeScript у Solidity. Всі відомі спроби реалізувати таку трансляцію (наприклад, solx) залишаються експериментальними і не підходять для продакшн-

застосування. Це обмеження зумовлене складністю семантичного узгодження між мовами та особливостями EVM.

У реальних проєктах, навіть за наявності сучасних інструментів, розробники стикаються з низкою викликів. Зокрема, їм необхідно володіти обома мовами - Solidity для написання контрактів і TypeScript для логіки dApp. Виникають труднощі з налагодженням на стику двох екосистем, а в окремих випадках - потреба в ручній синхронізації або перевірці узгодженості між типами Solidity та TypeScript.

Практичне дослідження однозначно показує, що TypeScript є ключовим елементом сучасної Web3-екосистеми, забезпечуючи високу якість коду та покращений досвід розробки на рівні децентралізованих додатків і супутнього інструментарію. Його основна цінність полягає не в тому, щоб замінити Solidity або напряму компілюватися у нього, а в тому, щоб забезпечити надійну типобезпечну інтеграцію та злагоджену роботу компіляційних процесів у межах проєкту. Незважаючи на успішні приклади створення компіляторів TypeScript-подібних мов для інших блокчейн-VM, для EVM така компіляція залишається переважно теоретичною концепцією, яка поки що не має практичного і готового до продакшену втілення.

5.1.2 Можливе рішення компіляції Solidity

У цій роботі наведено програмну реалізацію одного з можливих підходів до компіляції TypeScript у Solidity. Повний код реалізації подано в Додатку А. Основою реалізації є клас на мові TypeScript під назвою ClassParser, який виконує перетворення класів, написаних у стилі TypeScript, у відповідний код смарт-контрактів на Solidity.

ClassParser має внутрішнє поле options, у якому зберігаються параметри компіляції, такі як розмір табуляції. Конструктор класу приймає вхідне класове оголошення TypeScript (cls) і додаткові параметри, які можуть змінювати налаштування за замовчуванням.

```
  constructor(
    private readonly cls: ClassDeclaration,
    options?: Partial<ClassParserOptions>,
  ) {
    this.options = { ...defaultOptions, ...options };
  }
```

Головна логіка трансформації реалізована в методі compile. Цей метод поступово формує частини майбутнього контракту: назву за допомогою getContractName, список властивостей за допомогою getProperties, конструктор через getConstructor, а також методи за допомогою getMethods. Усі ці частини комбінуються в один текстовий рядок, що є повним кодом контракту на Solidity.

```
public compile() {
    const contractName = addLineBreaks(this.getContractName());
    const properties = addLineBreaks(
      this.getProperties()
        .map((property)            =>            spaceStart(property,
this.options.tabSize))
        .join("\n"),
      2,
    );

    const constructor = addLineBreaks(
      spaceStart(this.getConstructor(), this.options.tabSize),
      2,
    );
    const methods = this.getMethods().join("\n");

    return `${contractName}${properties}${constructor}${methods}}`;
  }
```

Метод getContractName створює заголовок контракту, додаючи ключове слово contract перед назвою класу та відкриваючи фігурну дужку. Метод getProperties проходиться по властивостях класу, витягує назви, типи та, якщо наявні, декоратори, які перетворюються в нижній регістр. Далі формується рядок оголошення змінної у форматі Solidity. Аналогічно, метод getMethods обробляє всі методи класу, отримуючи параметри через getMethodParameters, декоратори через getDecoratorNames та тіло методів через getMethodBody. Особливістю реалізації є очищення тіла методу від звернень типу this., оскільки вони не є валідними в Solidity.

```
private getContractName() {
    return `contract ${this.cls.getName()} {`;
  }

  private getProperties() {
    return this.cls?.getProperties().map((property) => {
      const propertyName = property.getName();
      const propertyType = property.getType().getText();
      const viewModifiers = this.getDecoratorNames(
        property.getDecorators(),
      ).toLowerCase();

      return `${propertyType} ${viewModifiers} ${propertyName};`;
    });
  }
```

Метод getMethodBody видаляє всі входження this. з тіла методу. Метод getDecoratorNames повертає список назв декораторів, об’єднаних у рядок. Метод getMethodParameters формує рядки з параметрами методів, враховуючи типи та імена. Метод getConstructor перевіряє наявність конструктора в класі, і якщо такий є, формує відповідний код Solidity з параметрами та тілом. Якщо конструктора немає, метод повертає порожній рядок.

```
private getMethodBody(statement: Statement) {
    const statementText = statement.getText();
    return `${statementText.replace("this.", "")}`;
  }

  private getDecoratorNames(decorators: Decorator[]) {
    return decorators.map((decorator) => decorator.getName()).join("
");
  }

  private getMethodParameters(param: ParameterDeclaration) {
    const paramName = param.getName();
    const paramType = param.getType().getText();
    return `${paramType} ${paramName}`;
  }
```

Як ілюструє рисунок 3, процес починається з парсингу TypeScript-файлу, далі відбувається пошук усіх класів і побудова абстрактного синтаксичного дерева за допомогою бібліотеки ts-morph. Після цього вузли дерева аналізуються й трансформуються у відповідні конструкції Solidity, з яких і формується фінальний код смарт-контракту.

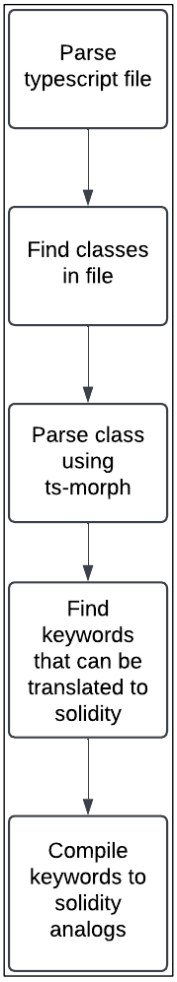

Рисунок 3 – Діаграма роботи програмної реалізації (рисунок виконаний самостійно)

Окрім цього, в коді визначено інтерфейс ClassParserOptions, який описує структуру опцій парсера, що наразі містить лише розмір табуляції. Також визначено змінну defaultOptions, яка містить значення опцій за замовчуванням, де розмір табуляції встановлено в 2 пробіли. Загалом, цей код надає інструмент для автоматичної генерації Solidity-коду з класів TypeScript, що може бути корисним для спрощення розробки смарт-контрактів

Цей підхід є лише початковою спробою автоматизувати перетворення TypeScript у Solidity, демонструючи концепцію транскомпіляції на прикладі обробки класів. Подальші розробки можуть охоплювати підтримку розширених мовних конструкцій, перевірку типів, інтеграцію з системою безпеки та більш глибоку оптимізацію коду для EVM.

5.1.3 Обробка та підготовка даних

У контексті дослідження можливостей компіляції TypeScript у Solidity та аналізу відповідного інструментарію, обробка та підготовка даних охоплює низку важливих аспектів, що забезпечують коректну взаємодію між різними мовами та підвищують ефективність розробки. Це включає попередню обробку TypeScript-коду з метою аналізу його структури, побудову абстрактного синтаксичного дерева (AST) для подальшої трансформації, вилучення релевантної інформації для генерації типів або коду на Solidity, а також підготовку згенерованого результату до компіляції офіційним компілятором Solidity. Також важливим етапом є синхронізація даних між результатами обох компіляторів, зокрема узгодження типів, параметрів функцій і структури контрактів для забезпечення типобезпечної інтеграції.

Обробка вихідного коду як на стороні TypeScript, так і Solidity є критично важливим етапом у будь-якому компіляційному або трансляційному процесі, особливо якщо йдеться про їх інтеграцію в рамках Web3-розробки. У випадку з TypeScript першим кроком є парсинг коду, тобто його перетворення у структуровану форму - абстрактне синтаксичне дерево (AST). Це дерево відображає всі синтаксичні елементи програми, дозволяючи подальшу програмну обробку, аналіз і трансформацію коду. Для цього можуть використовуватись як стандартний компілятор tsc, так і бібліотеки на кшталт ts-morph. Після цього виконується семантичний аналіз: перевіряється відповідність типів, коректність викликів функцій, доступність змінних тощо. У результаті утворюється збагачене представлення коду, яке містить всю потрібну інформацію для трансформації. Якщо поставлено завдання перетворити TypeScript у Solidity, необхідно виконати додаткову обробку цього представлення - нормалізувати, перетворити його у проміжну форму, яка ближча до цільової мови чи віртуальної машини EVM.

Зі свого боку, компіляція Solidity-коду виконується за допомогою компілятора solc. Вона включає класичні етапи: лексичний і синтаксичний аналіз, семантичну перевірку та генерацію машинного коду. У результаті створюється байт-код для виконання у віртуальній машині Ethereum, а також ABI - спеціальна

структура у форматі JSON, яка описує доступні зовнішні методи, їхні параметри та типи. Саме ABI є основою для інтеграції з TypeScript-кодом через інструменти на кшталт TypeChain або Abitype, які забезпечують типобезпечну взаємодію з контрактами на рівні TypeScript. Таким чином, обробка вихідного коду на обох мовах не тільки забезпечує трансляцію в цільовий код, але й створює критичні дані для подальшої взаємодії між мовами у рамках Web3-додатків.

У сучасній Web3-розробці підготовка даних для типобезпечної взаємодії між TypeScript і Solidity є ключовим практичним аспектом, який забезпечує надійність і зручність роботи з контрактами. Основним джерелом даних у цьому контексті виступає ABI - JSON-файл, що генерується компілятором Solidity.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 private data;

    function set(uint256 x) public {
        data = x;
    }

    function get() public view returns (uint256) {
        return data;
    }
}

ABI:
[
  {
    "inputs": [
      {
        "internalType": "uint256",
        "name": "x",
        "type": "uint256"
      }
    ],
    "name": "set",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "inputs": [],
    "name": "get",
    "outputs": [
      {
        "internalType": "uint256",
        "name": "",
```

```
        "type": "uint256"
      }
    ],
    "stateMutability": "view",
    "type": "function"
  }
]
```

Він містить всю необхідну інформацію про інтерфейс контракту: назви функцій, параметри, типи даних, модифікатори видимості (наприклад, view або payable) та структуру подій. Інструменти на кшталт TypeChain, Abitype, Wagmi або Viem обробляють цей файл, парсячи його в структуровану форму та вилучаючи метадані, які згодом нормалізуються і перетворюються на внутрішнє представлення, придатне для генерації типів.

Наступним етапом є генерація TypeScript-типів і інтерфейсів. На основі структури ABI автоматично створюються TS-файли, які містять точне типізоване відображення функцій контракту, параметрів, подій, повернених значень, а також типів даних, які відповідають специфіці Solidity (наприклад, uint, bytes або address перетворюються на BigNumber, string чи відповідні типи TS). Генеруються також допоміжні типи для взаємодії з контрактами - для транзакцій, підписів, подій тощо. Завдяки цьому розробники можуть безпечно та зручно викликати функції контракту у своїх TypeScript-додатках, отримувати підказки від редактора коду, уникати помилок на етапі розробки та легко масштабувати свій код без необхідності вручну розбиратися в структурі ABI. Такий підхід забезпечує високу якість коду, прискорює розробку та створює надійний місток між динамічною природою JavaScript-середовища і строгими вимогами до контрактів у Solidity.

У процесі розробки смарт-контрактів важливу роль відіграє обробка та підготовка даних для тестування й розгортання. Для написання юніт- та інтеграційних тестів на TypeScript розробники повинні заздалегідь сформувати вхідні дані, які передаватимуться у функції контрактів, визначити очікувані результати виконання, а також створити мокові об’єкти для емуляції зовнішніх залежностей, наприклад, відповіді від інших контрактів або API. Такі дані структуруються у TypeScript-файлах і завдяки системі типів автоматично

проходять валідацію ще до виконання тестів, що суттєво зменшує кількість помилок.

Для розгортання контрактів підготовка даних охоплює всі параметри, необхідні для коректної ініціалізації смарт-контракту в конкретній мережі. Це можуть бути адреси мереж або окремих сервісів, приватні ключі або доступ до них через захищені змінні середовища, параметри, які передаються в конструктор контракту, а також налаштування лімітів газу й ціни на газ. Такі дані зберігаються в конфігураційних файлах, наприклад .env, hardhat.config.ts, deploy.config.ts тощо, і завантажуються динамічно під час виконання скриптів розгортання. TypeScript дозволяє типізувати ці дані, забезпечуючи їх коректність і безпечну обробку в межах скриптів. Таким чином, ретельна підготовка даних є ключовою для забезпечення надійного й контрольованого циклу тестування та розгортання.

У теоретичному дослідженні підготовка даних охоплює кілька важливих етапів, спрямованих на формування ґрунтовної бази знань. Спершу здійснюється збір літературних джерел, який передбачає систематичний пошук наукових статей, технічної документації, оглядових матеріалів, публікацій у блогах, а також відкритих репозиторіїв на GitHub, що мають відношення до тематики компіляції, TypeScript, Solidity, блокчейн-розробки, абстрактних синтаксичних дерев (AST), проміжних представлень (IR) та формальної верифікації.

Після цього зібрані джерела обробляються шляхом категоризації за різними критеріями, такими як тип інструменту, підхід до компіляції, наявність підтримки формальної перевірки або описання обмежень цільової платформи. Це дозволяє виявити закономірності між підходами, виділити прогалини у наявних рішеннях та визначити напрямки, які потребують подальшого вивчення.

На етапі синтезу виконується інтеграція отриманої інформації з різних джерел задля формування узагальненого уявлення про теоретичний ландшафт предметної області. Таблиці, діаграми та графіки, що були створені в межах дослідження, є продуктами цього синтетичного процесу. Вони візуалізують складні взаємозв’язки, виділяють ключові переваги або недоліки підходів, а також допомагають систематизувати великий обсяг матеріалів для подальшого аналізу.

Обробка та підготовка даних є невіддільною складовою як практичного, так і теоретичного аспекту дослідження. У випадку взаємодії TypeScript та Solidity це охоплює повний цикл: починаючи з парсингу вихідного коду, побудови абстрактних синтаксичних дерев, виконання семантичного аналізу, генерації проміжного або цільового представлення (наприклад, ABI), і завершуючи формуванням типізованих інтерфейсів для безпечної інтеграції з фронтендом або тестовими середовищами. Ці процеси лежать в основі сучасних інструментів “паралельної компіляції”, що поєднують окремі мовні екосистеми в єдиний робочий процес.

Водночас, детальна обробка таких даних дозволяє теоретично моделювати можливість прямої компіляції TypeScript у Solidity, аналізуючи, які мовні конструкції сумісні, які потребують адаптації, і де виникають семантичні бар’єри. Це відкриває простір для пошуку нових підходів, таких як спеціалізовані DSL, трансляція в проміжне представлення, або ж обмежене підмноження TypeScript. Таким чином, підготовка даних виступає як технічно необхідний крок для реалізації інтеграції, так і як аналітичний інструмент для пошуку гіпотетичних шляхів уніфікації мовних просторів.

5.2 Методи порівняння за критерієм швидкості компіляції

Оцінювання ефективності використання TypeScript у розробці смарт-контрактів неможливе без врахування швидкості компіляції, оскільки саме вона суттєво впливає на комфорт роботи розробника, тривалість ітерацій та загальну продуктивність команди. У межах дослідження було застосовано експериментальний підхід, що дозволив кількісно порівняти час компіляції у двох сценаріях: при прямій реалізації контракту на Solidity та при розробці з використанням TypeScript як вихідної мови з подальшим трансформуванням у Solidity-код.

Для забезпечення об’єктивності було створено ідентичні за логікою смарт-контракти, реалізовані обома способами. У випадку TypeScript-підходу в обчислення включалися етапи генерації проміжних артефактів (зокрема ABI),

перетворення коду в Solidity за допомогою спеціалізованих скриптів або бібліотек (наприклад, TypeChain чи плагінів для Hardhat), а також фінальна компіляція за допомогою solc. Для обох варіантів час компіляції вимірювався в однакових умовах - з контрольованим середовищем виконання та відключеними фоновими задачами, що забезпечило чистоту експерименту та достовірність отриманих результатів.

Такий аналіз дозволяє не лише оцінити вплив використання TypeScript на продуктивність збірки, а й побачити потенційні точки оптимізації у розробницькому процесі, пов’язані з трансляцією та інтеграцією між мовами.

У дослідженні було також проаналізовано, як структура проєкту впливає на швидкість компіляції, зокрема при масштабуванні, додаванні нових контрактів або оновленні залежностей. Виявлено, що при використанні TypeScript компіляційний процес включає кілька додаткових етапів, серед яких генерація типів, створення обгорток для взаємодії з контрактами та підготовка конфігураційного середовища. Це дещо збільшує загальний час компіляції, проте водночас зменшує ймовірність помилок на наступних етапах, оскільки значна частина перевірок виконується ще до трансляції в Solidity.

У процесі роботи були враховані також фактори оптимізації, характерні для екосистеми TypeScript, зокрема можливість паралельної обробки даних попередніх збірок, підтримка інкрементальної компіляції та кешування результатів. Ці механізми дозволяють частково або повністю компенсувати додаткові витрати часу, особливо під час повторних компіляцій у рамках одного проєкту. В умовах активної розробки, коли відбувається часта зміна коду, це забезпечує суттєве прискорення циклу збирання та підвищує ефективність роботи команди.

Таким чином, порівняльний аналіз за критерієм швидкості компіляції показав, що застосування TypeScript для створення смарт-контрактів є обґрунтованим у проєктах із складною архітектурою. Незважаючи на трохи довший компіляційний процес у порівнянні з нативною розробкою на Solidity, додаткові переваги у вигляді статичної типізації, кращої інтеграції з фронтендом і

розвиненої інфраструктури інструментів компенсують цей недолік. У довгостроковій перспективі це дозволяє підвищити стабільність коду та зменшити витрати на підтримку.

5.3 Використання TypeScript для створення клієнтів та автоматизації процесів

Однією з ключових переваг використання TypeScript у процесі розробки смарт-контрактів є можливість високого рівня автоматизації та генерації типобезпечних клієнтів для взаємодії з контрактами. Завдяки цьому суттєво зменшується ризик помилок під час виклику функцій контрактів, оскільки всі звернення до них проходять через попередньо згенеровані об’єкти з чітко визначеними типами. Це не тільки полегшує написання коду, але й значно скорочує час, необхідний для налагодження та тестування, оскільки більшість помилок можуть бути виявлені ще на етапі компіляції. Крім того, така інтеграція сприяє кращому поєднанню фронтенд- і бекенд-частин застосунку, дозволяючи розробникам швидко та надійно реалізовувати взаємодію з блокчейном без зайвих ручних перевірок та дублювання коду.

Приклад використання TypeChain для створення клієнтів: Використовуючи інструмент TypeChain, можна автоматично генерувати клієнтів для Solidity-контрактів:

```
import { MyContract } from "../typechain";
import { ethers } from "hardhat";

describe("MyContract", function () {
  it("should return the correct value", async function () {
    const [owner] = await ethers.getSigners();
    const contract = await ethers.getContractFactory("MyContract");
    const deployedContract: MyContract = await contract.deploy();

    const value = await deployedContract.getValue();
    expect(value).to.equal(42);
  });
});
```

У цьому прикладі TypeChain генерує типізований клієнт для контракту MyContract, що дозволяє викликати функцію getValue() з повною впевненістю у відповідності типів, які очікує контракт. Такий підхід забезпечує безпечну взаємодію з контрактами ще на етапі написання коду, мінімізуючи ймовірність помилок під час виконання.

Крім цього, TypeScript відкриває широкі можливості для автоматизації робочих процесів. За його допомогою можна створювати скрипти, що автоматично розгортають контракти в тестових або основних мережах, спрощуючи процес деплойменту. Також TypeScript активно використовується в середовищах типу Hardhat або Truffle для побудови автоматизованих тестів, що охоплюють ключову логіку контрактів. Це дозволяє виявляти помилки ще до релізу. Додатково розробники можуть використовувати TypeScript для генерації документації з ABI або організовувати збір та обробку аналітики, що стосується транзакцій чи поведінки контрактів у мережі, що значно підвищує прозорість і контроль над розгортаним децентралізованим застосунком.

5.4 Паралельна компіляція файлів typescript у Solidity

У цьому підрозділі аналізується можливість оптимізації процесу компіляції TypeScript у Solidity шляхом впровадження паралельної обробки за допомогою функціональностей, доступних у середовищі Node.js. Такий підхід стає особливо цінним у великих проектах, де кодова база розбита на численні незалежні модулі, які можуть бути оброблені окремо. Основна ідея полягає в тому, щоб замість послідовної трансляції кожного TypeScript-файлу запускати окремі процеси або потоки, кожен з яких виконує перетворення окремого файлу в Solidity-контракт. Завдяки тому, що Node.js підтримує асинхронність та неблокуюче виконання завдань, реалізація такого підходу може базуватись на механізмах Promise, асинхронних функцій або більш низькорівневих засобах, як-от worker_threads або child_process. Це дозволяє значно зменшити загальний час компіляції, особливо у випадках, коли кількість файлів зростає, а обробка кожного з них не залежить від

інших. У результаті застосування цього підходу не тільки пришвидшує процес трансляції, а й підвищує масштабованість усього конвеєра збірки.

Застосування паралельної обробки на практиці показало помітне скорочення загального часу компіляції порівняно з традиційною послідовною обробкою, особливо в умовах багатоядерних систем, де кожен процес може виконуватись на окремому ядрі. У рамках дослідження було розроблено прототип компілятора, який приймає масив TypeScript-файлів, запускає для кожного з них окремий процес трансляції, а після завершення усіх - об’єднує результати у спільну структуру. Ця структура надалі може бути використана для генерації ABI, типів або передачі контрактів у систему розгортання в мережі Ethereum. Такий підхід виявився не лише ефективним, а й стабільним: він добре масштабувався при збільшенні кількості вхідних файлів і гнучко реагував на зміну кількості активних потоків, не спричиняючи критичних збоїв. Окремим плюсом виявилася можливість точкового логування помилок - кожен процес обробки міг самостійно зберігати звіт про помилки, що спрощувало налагодження і забезпечувало стійкість загального процесу до часткових збоїв. Це особливо важливо у контексті командної розробки, де ізольоване тестування окремих контрактів або модулів може проходити незалежно, без зупинки всієї збірки.

Під час дослідження було також проаналізовано ефективність різних стратегій паралельного виконання, зокрема порівняно worker-потоки та дочірні процеси. Результати показали, що використання worker-потоків дійсно дозволяє зменшити споживання пам’яті, оскільки вони мають доступ до загального простору пам’яті та не потребують повної ізоляції. Проте при масштабному навантаженні - коли необхідно одночасно компілювати велику кількість контрактів - класичні дочірні процеси виявилися стабільнішими та менш схильними до помилок у складних сценаріях. Крім того, була реалізована та протестована система кешування проміжних результатів компіляції. Це дало змогу значно пришвидшити повторну збірку, оскільки незмінені файли не потребували повторного аналізу й трансляції, що, у свою чергу, зменшувало

навантаження на обчислювальні ресурси та підвищувало загальну ефективність системи.

Загалом, використання паралельної компіляції на базі можливостей Node.js виявилося не лише технічно обґрунтованим, але й практично ефективним для реальних задач розробки смарт-контрактів. Завдяки цьому підходу вдалося суттєво зменшити час обробки великої кількості модулів без втрати стабільності, що є особливо важливим у контексті Web3-проєктів з активною фазою розробки. Гнучкість Node.js-платформи дала змогу легко налаштувати пайплайн компіляції під конкретні вимоги: обрати між worker-потоками або дочірніми процесами, налаштувати кешування, розподілити ресурси між задачами.

Особливу цінність паралельна компіляція показала у великих проєктах, де кількість вихідних TypeScript-файлів перевищує десятки чи сотні одиниць. У таких випадках послідовна обробка перетворюється на вузьке місце, тоді як паралельна дає змогу задіяти багатоядерні процесори максимально ефективно. Крім підвищення швидкодії, такий підхід добре інтегрується в CI/CD-процеси, де можливість контролювати окремі етапи трансляції та логування помилок критично важлива для автоматизації.

Для ілюстрації переваг проведеної реалізації результати експерименту були згруповані у таблицю 3, що демонструє залежність середнього часу компіляції від кількості паралельно оброблюваних файлів. Ці дані підтверджують доцільність розглянутого підходу та його масштабованість при збільшенні обсягів коду. Заміри проводились на машині з процесором Apple Silicon M1, 16 гб оперативної пам’яті.

Повний код виконання алгоритму паралельного компілювання TypeScript коду в Solidity код можна побачити в додатку Б.

Як засвідчують результати, зібрані в таблиці, паралельна компіляція демонструє помітну перевагу над традиційним послідовним підходом. При зростанні кількості вхідних файлів її ефективність значно підвищується, дозволяючи скоротити загальний час обробки майже втричі. Цей ефект особливо

виражений у масштабних проєктах, де кількість смарт-контрактів велика, а цикли оновлення коду – часті.

Таблиця 3 – Порівняння послідовного та паралельного алгоритму компіляції Typescript в Solidity (таблиця виконана самостійно)

Кількість TypeScript-файлів	Послідовна компіляція (с)	Паралельна компіляція (с)
125	4.8	2.3
200	9.5	4.1
250	18.7	7.8
300	28.1	11.3
500	46.9	17.2

Показники, отримані в ході експерименту, підтверджують, що впровадження паралельної компіляції є виправданим рішенням для побудови продуктивного та гнучкого інструментарію в рамках моделі розробки, що поєднує TypeScript і Solidity.

ВИСНОВКИ

Проведене дослідження дало змогу сформувати цілісне уявлення про значення мови TypeScript у сучасній розробці смарт-контрактів і децентралізованих застосунків. У процесі аналізу було також окреслено основні труднощі та потенціал розвитку прямої трансляції TypeScript у Solidity. На сьогодні TypeScript є однією з найважливіших мов у сфері Web3, адже фактично став стандартом для створення децентралізованих застосунків. Його поширення пояснюється перевагами типобезпеки, що сприяє підвищенню надійності коду, покращує зручність і якість роботи розробника завдяки інтеграції з сучасними середовищами розробки, а також забезпечує гнучкість при розбудові великих проєктів і роботи в команді. Оскільки багато розробників мають досвід роботи з TypeScript у сфері Web2, їм набагато легше адаптуватися до вимог Web3-розробки, що пришвидшує входження в блокчейн-екосистему.

Одним із ключових висновків дослідження є те, що, попри значну популярність і поширення TypeScript у сфері Web3, наразі не існує зрілого та широко підтримуваного компілятора, який дозволяв би безпосередньо писати логіку смарт-контрактів на TypeScript і перетворювати її у Solidity або байт-код для EVM, придатний для використання в реальному середовищі. Замість цього в екосистемі утвердилась модель паралельної компіляції. У такому підході Solidity залишається основною мовою для створення контрактної логіки, яка компілюється в байт-код за допомогою стандартного компілятора solc. TypeScript, зі свого боку, відіграє допоміжну роль, будучи основою для написання скриптів розгортання, модулів тестування, бекенд- і фронтенд-частин додатків. Взаємодія між цими двома мовами забезпечується за допомогою інструментів, які автоматично генерують типізовані інтерфейси на основі ABI уже скомпільованих контрактів. Завдяки цьому досягається типобезпечність, що дає змогу зменшити кількість помилок при виклику функцій контракту з TypeScript-коду.

Теоретичний аналіз вказує на серйозні труднощі, пов’язані з розробкою повноцінного компілятора, який би перетворював TypeScript безпосередньо в Solidity або інший формат, сумісний з EVM. Головною проблемою є глибокий

семантичний розрив між цими мовами: TypeScript орієнтований на динамічні вебдодатки, має гнучку систему типів та працює у середовищі з автоматичним управлінням пам’яттю, тоді як Solidity є строго типізованою мовою з обмеженим набором конструкцій і прямим доступом до ресурсів EVM. Ці відмінності ускладнюють точне й ефективне відображення логіки з однієї мови в іншу. Додатково, сама віртуальна машина Ethereum має низку обмежень, включаючи ліміти на споживання газу, розмір коду і набір допустимих операцій, що вимагає високого рівня оптимізації під час трансляції. Ще однією перешкодою є відсутність проміжного представлення, яке було б достатньо універсальним і водночас здатним відобразити особливості обох мов. Нарешті, смарт-контракти мають критичні вимоги до безпеки, тому будь-який компілятор повинен проходити сувору перевірку на коректність і відповідати формальним критеріям довіри, що істотно ускладнює процес його створення.

TypeScript суттєво сприяє масштабованості процесу розробки Web3-проєктів, особливо коли над одним кодовим репозиторієм працює велика команда. Його типова система дозволяє зменшити кількість помилок, спростити перевірку взаємодій між компонентами та полегшити довгострокову підтримку коду. Проте, сам по собі TypeScript не впливає безпосередньо на масштабованість або продуктивність смарт-контрактів у виконанні, оскільки підсумковий код, що працює в мережі, компілюється саме за допомогою компілятора Solidity. Отже, ефективність і газові витрати контракту визначаються якістю генерованого Solidity-коду та його оптимізацією. Теоретичний компілятор з TypeScript у Solidity мав би бути надзвичайно добре оптимізованим, аби зберегти прийнятні параметри виконання. Попри ці виклики, дослідження показало, що на деяких альтернативних блокчейн-платформах TypeScript-подібні мови вже використовуються як основа для написання смарт-контрактів. Наприклад, проєкт Azle дозволяє писати контракти на TypeScript для Internet Computer Protocol, а AssemblyScript застосовується в WASM-орієнтованих блокчейнах на кшталт NEAR або Casper. Ці приклади демонструють технічну здійсненність концепції,

хоча вказують і на специфічні труднощі, пов’язані з EVM, які стримують аналогічний прогрес у сфері Ethereum.

Обробка та підготовка даних відіграють ключову роль у забезпеченні ефективної інтеграції між TypeScript і Solidity. Саме ці процеси створюють основу для взаємодії між двома мовами, починаючи з парсингу та семантичного аналізу вихідного коду, який дозволяє отримати структуроване представлення програми. Подальша генерація ABI з Solidity-контрактів та її обробка для створення типізованих файлів на TypeScript забезпечують точну й безпечну взаємодію між фронтендом і смарт-контрактами. Не менш важливою є підготовка тестових даних, сценаріїв викликів і конфігурацій для розгортання, які дозволяють автоматизувати та стандартизувати процес розробки. Усе це разом формує узгоджене середовище, в якому обидві мовні екосистеми можуть ефективно співіснувати, забезпечуючи зручність і надійність для розробника.

Загалом проведене дослідження демонструє, що TypeScript займає важливе місце в сучасній Web3-розробці, особливо в частині створення децентралізованих застосунків та супутнього інструментарію. Його типова безпека, інтеграція з сучасними середовищами розробки та підтримка масштабованості роблять його надзвичайно зручним і потужним інструментом. Водночас ідея прямої компіляції TypeScript у Solidity залишається складною задачою, що вимагає подальших досліджень і технічного прориву. На теперішньому етапі найбільш ефективним і виправданим підходом залишається модель паралельного використання обох мов, де Solidity відповідає за основну логіку контрактів, а TypeScript забезпечує типобезпечну інтеграцію та автоматизацію навколо них. Такий підхід дозволяє досягати високої надійності, швидкості розробки та зручності при масштабуванні проектів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Bachini J. "Javascript Smart Contracts"/Conference NIPS, 2023, 28 с.
2. Elisha A. " Algorand Expands Developer Toolkit with TypeScript Support to Attract Web3 Developers "/ICLR Conference, 2023, 9 с.
3. Han J. " Plu-ts Typescript smart-contracts - road to production "/arxiv, 2020, URL: https://arxiv.org/abs/2003.00703v1. (дата звернення: 09.15.2024)
4. Lexis J. "Tutorial: Using ethers.js and TypeScript to Read a Smart Contract | Stability Docs."/ arxiv, 2021, URL: https://arxiv.org/abs/2103.00020 (дата звернення: 03.15.2025)
5. Lexis J. "The Rise of TypeScript: Why It's the Future of JavaScript Development."/ arxiv, 2023, URL: https://arxiv.org/abs/2103.00020 (дата звернення: 03.22.2025)
6. Newman J. “A better way to parse solidity errors for typescript"/ arxiv, 2021, URL: https://arxiv.org/abs/2103.00020 (дата звернення: 05.12.2025)
7. TypeChain - Solidity Tools - Alchemy. URL: https://www.alchemy.com/dapps/typechain (дата звернення 01.20.25)
8. Using TypeScript | Ethereum development environment for ... - Hardhat URL: https://hardhat.org/hardhat-runner/docs/guides/typescript (дата звернення 12.05.2024
9. Algorand broadens dev tooling, now offers native TypeScript alongside Python to court new web3 developers - PR Newswire. URL: https://www.prnewswire.com/news-releases/algorand-broadens-dev-tooling-now-offers-native-typescript-alongside-python-to-court-new-web3-developers-302411956.html (дата звернення 12.20.2024)
10. Afanasieva, I., Golian, N., Golian, V., Khovrat, A., & Onyshchenko, K. (2023). Application of Neural Networks to Identify of Fake News. COLINS (2), 346-358.
11. Каук В.І. "Генеративний штучний інтелект - креативний помічник дизайнера". // Матеріали конференції "Використання нових методів навчання у видавничо-поліграфічній галузі", ХНУРЕ, 2024, с. 283-294.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

10. Afanasieva, I., Golian, N., Golian, V., Khovrat, A., & Onyshchenko, K. (2023). Application of Neural Networks to Identify of Fake News. COLINS (2), 346-358.

11. Каук В.І. “Generative AI - creative designer helper". // Матеріали конференції "Using new teaching methods in the publishing and printing industry ", ХНУРЕ, 2024, с. 135-168.