МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ТЕХНОЛОГІЇ ДЛЯ АНАЛІЗУ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

Виконала: студентка групи 2П-22
Спеціальності
121 Інженерія програмного забезпечення
Аліна ПЕТРИЧЕНКО
Керівник:
Майя ЛЮТА

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО
(підпис)

«_____» _____2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Петриченко Аліні Сергіївні

1. Тема кваліфікаційної роботи Технології для аналізу продуктивності програмного забезпечення на основі штучного інтелекту
Керівник роботи Люта Майя В'ячеславівна, викладач вищої категорії
затверджені наказом закладу фахової передвищої освіти від «06» жовтня 2025 року № 84/1-у.
2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані до кваліфікаційної роботи: Технології розробки платформи аналізу продуктивності програмного забезпечення на основі штучного інтелекту: Python, FastAPI, TensorFlow, PyTorch (модулі аналізу та машинного навчання); React 19, Vite 7 (клієнтська частина); Node.js, Express.js (серверна частина); PostgreSQL/MongoDB Atlas (зберігання даних); REST API, WebSocket (взаємодія компонентів системи); JWT та bcrypt (авторизація і захист доступу).
Інструменти розробки: Git, Visual Studio Code. Оформлення кваліфікаційної роботи відповідно до чинних стандартів ДСТУ.

4. Зміст кваліфікаційної роботи: Вступ. Методологічні засади аналізу продуктивності програмного забезпечення на основі архітектурних атрибутів якості. Архітектурне моделювання платформи аналіз продуктивності на основі штучного інтелекту. Оцінювання архітектури, ризики та рекомендації щодо впровадження. Висновки.
5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1. Методологічні засади аналізу продуктивності програмного забезпечення на основі архітектурних атрибутів якості	09.12.2025	
3	Розділ 2. Архітектурне моделювання платформи аналіз продуктивності на основі штучного інтелекту	10.03.2026	
4	Розділ 3. Оцінювання архітектури, ризики та рекомендації щодо впровадження	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачці циклової комісії (за 7 днів до захисту)	10.06.2026	

Студент _____ Аліна ПЕТРИЧЕНКО
(підпис)

Керівник роботи _____ Майя ЛЮТА
(підпис)

АНОТАЦІЯ

У кваліфікаційній роботі здійснено дослідження та розроблення концепції платформи аналізу продуктивності програмного забезпечення на основі штучного інтелекту. Виконано аналіз методологічних засад оцінювання продуктивності програмних систем, розглянуто взаємозв'язок між продуктивністю, спостережуваністю та надійністю як основними атрибутами якості програмного забезпечення. Проведено аналіз сучасних інструментів вимірювання продуктивності, систем метрик і підходів до діагностики деградацій.

Розроблено архітектурну модель платформи аналізу продуктивності на основі ШІ, визначено архітектурні драйвери, межі автоматизації та структуру основних компонентів системи. Запропоновано сервіс аналізу першопричин (Root Cause Analysis) із використанням методів штучного інтелекту для автоматизованого виявлення проблем продуктивності програмного забезпечення. Сформовано архітектурний контекст та підхід до операціоналізації процесів упровадження платформи.

Виконано оцінювання запропонованої архітектури із застосуванням ATAM-аналізу, визначено точки чутливості, компромісні рішення та потенційні ризики впровадження. Проведено SWOT-оцінювання, розглянуто обмеження використання AI/RCA та сформовано рекомендації щодо подальшого розвитку системи. Результати роботи можуть бути використані для створення інструментів моніторингу, аналізу та оптимізації продуктивності програмного забезпечення в умовах сучасних інформаційних систем.

Ключові слова: АНАЛІЗ ПРОДУКТИВНОСТІ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, АРХІТЕКТУРА ПЗ, МЕТРИКИ ЯКОСТІ, СПОСТЕРЕЖУВАНІСТЬ, ROOT CAUSE ANALYSIS, AI, ATAM, ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ.

ANNOTATION

The qualification work presents the research and development of a concept for a software performance analysis platform based on artificial intelligence. The methodological foundations of evaluating software system performance were analyzed, and the relationship between performance, observability, and reliability as key software quality attributes was considered. Modern performance measurement tools, metric systems, and approaches to diagnosing performance degradation were analyzed.

An architectural model of an AI-based software performance analysis platform was developed, defining architectural drivers, automation boundaries, and the structure of the main system components. A root cause analysis (RCA) service based on artificial intelligence methods was proposed to enable automated detection of software performance issues. The architectural context and approach to the operationalization of platform implementation processes were defined.

The proposed architecture was evaluated using ATAM analysis, identifying sensitivity points, trade-off decisions, and potential implementation risks. A SWOT analysis was performed, limitations of AI/RCA application were considered, and recommendations for further system development were formulated. The results of the work can be used for creating tools for monitoring, analyzing, and optimizing software performance in modern information systems.

Keywords: SOFTWARE PERFORMANCE ANALYSIS, SOFTWARE, ARTIFICIAL INTELLIGENCE, SOFTWARE ARCHITECTURE, QUALITY METRICS, OBSERVABILITY, ROOT CAUSE ANALYSIS, AI, ATAM, PERFORMANCE OPTIMIZATION.

ЗМІСТ

ПЕРЕЛІК ВИКОРИСТАНИХ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 3
ВСТУП 4
РОЗДІЛ 1 МЕТОДОЛОГІЧНІ ЗАСАДИ АНАЛІЗУ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ АРХІТЕКТУРНИХ АТРИБУТІВ ЯКОСТІ 6
1.1 Продуктивність, спостережуваність і надійність як взаємозалежні атрибути якості 6
1.2 Інструментальне забезпечення вимірювання продуктивності та критична оцінка меж застосування засобів 9
1.3 Система метрик, критерії приймання та доказова діагностика деградацій 13
РОЗДІЛ 2 АРХІТЕКТУРНЕ МОДЕЛЮВАННЯ ПЛАТФОРМИ АНАЛІЗУ ПРОДУКТИВНОСТІ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ 16
2.1 Архітектурні драйвери та межі автоматизації аналізу продуктивності 16
2.2 Архітектурний контекст платформи аналізу продуктивності 23
2.3 Попередня архітектура платформи аналізу продуктивності 29
2.4 Сервіс аналізу першопричин на основі штучного інтелекту 34
2.5 Операціоналізація процесів упровадження 39
РОЗДІЛ 3 ОЦІНЮВАННЯ АРХІТЕКТУРИ, РИЗИКИ ТА РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ 43
3.1 ATAM-оцінка, sensitivity points і tradeoff-аналіз запропонованої архітектури 43
3.2. Ризики, обмеження, керування AI/RCA та багатокритеріальна SWOT-оцінка 46
3.3 Дорожня карта впровадження, контрольні артефакти та підсумкові рекомендації 49
ВИСНОВКИ 53
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 55
ДОДАТКИ 58
Додаток А – Мінімальний перелік артефактів для демонстрації практичної частини 58

ПЕРЕЛІК ВИКОРИСТАНИХ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

ADR – запис архітектурного рішення (Architecture Decision Record)

AI – штучний інтелект (Artificial Intelligence)

AIOps – штучний інтелект для IT-операцій (Artificial Intelligence for IT Operations)

APM – моніторинг продуктивності застосунків (Application Performance Monitoring)

ATAM – метод аналізу архітектурних компромісів (Architecture Tradeoff Analysis Method)

C4 – модель архітектурного опису системи через рівні контексту (Context), контейнерів (Containers), компонентів (Components) і коду (Code)

CI/CD – безперервна інтеграція та постачання (Continuous Integration / Continuous Delivery)

MELT – метрики, події, журнали та траси (Metrics, Events, Logs, Traces)

середній час виявлення (середній час виявлення (MTTD)) – середній час виявлення (Mean Time to Detect)

середній час відновлення (середній час відновлення (MTTR)) – середній час відновлення / усунення (Mean Time to Restore/Resolve)

OTEL – OpenTelemetry, стандарт і екосистема збору телеметрії

QAW – семінар з атрибутів якості (Quality Attribute Workshop)

RCA – аналіз першопричин (Root Cause Analysis)

RED – інтенсивність запитів, помилки, тривалість (Rate, Errors, Duration)

SLI – індикатор рівня сервісу (Service Level Indicator)

SLO – цільовий рівень сервісу (Service Level Objective)

USE – використання, насичення, помилки (Utilization, Saturation, Errors)

ПЗ – програмне забезпечення

ВСТУП

Кваліфікаційна робота виконана в архітектурному та інженерно-прикладному фокусі: аналіз продуктивності програмного забезпечення розглядається не як ізольована техніка профілювання, а як система взаємопов’язаних рішень щодо вимог, архітектури, телеметрії, перевірки, експлуатації та керування ризиками. Такий фокус відповідає логіці горизонтального підходу до кваліфікаційної роботи, коли головним результатом є не лише програмний фрагмент або огляд засобів, а обґрунтоване інженерне рішення з вимірюваними критеріями придатності, архітектурними моделями, сценаріями якості та практичними артефактами впровадження [1].

Актуальність теми пов’язана з тим, що в сучасних розподілених системах продуктивність рідко залежить від одного локального фрагмента коду. На кінцевий час відповіді впливають мережеві виклики, черги, політики повторних спроб, залежності баз даних, параметри контейнерів, робота garbage collector, стан зовнішніх сервісів, режим масштабування та якість інструментування. Тому без архітектурного опису системи, що фіксує зацікавлені сторони, точки взаємодії, межі відповідальності, джерела телеметрії та сценарії якості, аналіз продуктивності перетворюється на фрагментарне спостереження за графіками, а не на керований процес прийняття рішень.

У роботі під запропонованою платформою аналізу продуктивності на основі ШІ розуміється не універсальний “інтелектуальний моніторинг”, а референсна архітектура процесу, у якому дані про метрики, журнали, траси та події збираються узгоджено, проходять етапи нормалізації, кореляції, зберігання, візуалізації, формування гіпотез про деградацію та доказової перевірки цих гіпотез. Компонент штучного інтелекту в такій архітектурі трактовано обережно: він може прискорювати пошук підозрілих залежностей, ранжувати ймовірні причини інциденту або генерувати пояснення, однак не замінює архітектурної відповідальності, інженерного контролю, перевірки причинно-наслідкових зв’язків і фіксації доказів.

Метою роботи є розроблення та оцінювання архітектурно обґрунтованого підходу до аналізу продуктивності ПЗ на основі поєднання рекомендованих практик архітектури програмного забезпечення (software architecture), спостережуваності (observability), SRE, інженерії продуктивності (performance engineering) та AIOps.

Досягнення мети передбачає: уточнення проблематики аналізу продуктивності в розподілених системах; обґрунтування інструментального контуру збору й аналізу телеметрії; побудову архітектурної та сценарної моделі платформи; визначення метрик, операційних артефактів і критеріїв упровадження; оцінювання архітектурних компромісів, ризиків та обмежень.

Об’єктом дослідження є процеси вимірювання, аналізу та покращення продуктивності програмного забезпечення в умовах розподілених, мікросервісних і хмароорієнтованих (cloud-native) архітектур.

Предметом дослідження є архітектурні підходи, інструментальні засоби, метрики, сценарії якості та практики операціоналізації аналізу продуктивності на основі ШІ.

Методи дослідження включають аналіз наукової та технічної літератури, критичне порівняння інструментів спостережуваність, сценарний аналіз атрибутів якості, архітектурне моделювання, елементи ATAM-оцінювання, SWOT-аналіз, систематизацію SLI/SLO та проєктування операційних артефактів. У тексті використано нумеровані посилання в порядку першої згадки джерел, що дозволяє безпосередньо простежити зв’язок між твердженнями, джерелами та практичними висновками.

РОЗДІЛ 1 МЕТОДОЛОГІЧНІ ЗАСАДИ АНАЛІЗУ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ АРХІТЕКТУРНИХ АТРИБУТІВ ЯКОСТІ

1.1 Продуктивність, спостережуваність і надійність як взаємозалежні атрибути якості

Для інженерного аналізу продуктивності недостатньо описати набір інструментів, які збирають CPU, memory, latency або throughput. Потрібно визначити, які архітектурні властивості системи підлягають оцінюванню, хто є зацікавленою стороною, у якому середовищі виникає стимул, які компоненти реагують на нього та за якими критеріями відповідь системи вважається прийнятною. Саме таке трактування узгоджується з підходом до архітектурного опису, у якому архітектура подається через зацікавлені сторони, їхні concerns, viewpoints, views та rationale прийнятих рішень [2].

Модель якості програмного забезпечення також не зводить продуктивність до «швидкості». Вона розрізняє ефективність продуктивності (performance efficiency), надійність (reliability), безпеку (security), супроводжуваність (maintainability), переносимість (portability) та інші характеристики, які можуть взаємно підтримувати або обмежувати одна одну [3]. Наприклад, агресивне кешування може зменшити затримку, але створити ризик неконсистентності даних; збільшення кількості реплік може підвищити пропускну здатність, але ускладнити узгодженість стану; надмірне трасування може покращити діагностику, але збільшити накладні витрати.

У класичній архітектурній літературі продуктивність розглядається як один із quality attributes, реалізація якого залежить від набору тактик: обмеження попиту, керування ресурсами, кешування, паралелізація, черги, балансування навантаження, пріоритезація, контроль contention та інші [4]. У цій роботі такі тактики не подаються як універсальний рецепт. Вони розглядаються як можливі архітектурні реакції на конкретні сценарії якості, що мають власні припущення, обмеження, витрати й побічні ефекти.

QAW є корисним для раннього уточнення вимог, оскільки допомагає відокремити загальні побажання від сценаріїв, які можна обговорювати, пріоритезувати та перевіряти [5]. Для теми аналізу продуктивності це особливо важливо: твердження “система має швидко працювати” не дає підстав для архітектурного рішення, тоді як сценарій “під час пікового навантаження 300 RPS p95 latency для операції пошуку не перевищує 500 мс протягом 10 хвилин, а рівень 5xx не перевищує 0,1%” уже може бути пов’язаний із інструментуванням, тестовим профілем, SLO та контрольний шлюз продуктивності.

ATAM доповнює QAW тим, що дозволяє оцінити не лише наявність сценаріїв, а й якість архітектурних рішень щодо цих сценаріїв: де розташовані sensitivity points, які tradeoffs виникають, які ризики потребують додаткових експериментів, а які припущення можуть бути прийняті як non-risks за наявності доказів [6]. Для цієї роботи це означає, що запропонована платформа аналіз продуктивності на основі ШІ повинна оцінюватися не за кількістю використаних інструментів, а за здатністю підтримати вимірювання, діагностику, пояснення й повторну перевірку рішень.

Окремої уваги потребує взаємозв’язок продуктивності та спостережуваності. У навчальних роботах спостережуваність часто подається як “наявність Prometheus і Grafana”, однак така інтерпретація занадто вузька. Спостережуваність має цінність лише тоді, коли зовнішні сигнали дозволяють достатньо обґрунтовано реконструювати внутрішній стан системи. Якщо метрики не мають узгодженої семантики, логи не містять correlation ID, траси обрізаються sampling-політикою, а дашборди не прив’язані до SLO, то навіть дорогий стек моніторингу може не скоротити середній час відновлення (MTTR).

Надійність у цьому контексті також не є вторинною характеристикою. Система може бути швидкою за нормального навантаження, але ненадійною при часткових відмовах, повторних спробах, деградації зовнішньої залежності або зміні профілю навантаження. Тому продуктивність слід аналізувати разом із доступністю (availability), відмовостійкістю (fault tolerance), відновлюваністю (recoverability) та експлуатаційною придатністю (operability). Практичним

наслідком є необхідність включати до архітектурного аналізу не лише середні значення часу відповіді, а й сценарії відмов, повторів, деградації залежностей і перевищення бюджетів помилок.

AI-компоненти в такій системі мають розглядатися як засоби підтримки рішення, а не як автономні арбітри істини. Вони можуть виявляти аномальні патерни, групувати схожі інциденти, ранжувати підозрілі компоненти, але їхні висновки залежать від якості вхідної телеметрії, репрезентативності історичних даних, стабільності архітектури та наявності контрольованих експериментів. Тому будь-яка рекомендація AI/RCA має бути прив’язана до картка доказів: які сигнали використано, яка гіпотеза сформована, які альтернативні пояснення не виключені, які перевірки проведено і який залишковий ризик залишається.

У межах кваліфікаційної роботи доцільно використовувати не надмірно амбітну, а реалістичну модель: платформа аналізу продуктивності на основі ШІ є не завершеною промисловою AIOps-системою, а архітектурно обґрунтованим прототипом процесу. Його цінність полягає в тому, що він показує, як від загальної проблеми “падіння продуктивності” перейти до вимірюваної вимоги, архітектурного рішення, інструментального контуру, тестового сценарію, дашборда, алерта, RCA-процедури та рекомендації щодо еволюції системи.

Взаємозв’язок атрибутів якості та архітектурних наслідків наведено в таблиці 1.1.

Таблиця 1.1 – Взаємозв’язок атрибутів якості та архітектурних наслідків

Атрибут якості	Як проявляється у темі роботи	Типові метрики	Архітектурні наслідки
Продуктивність	Час відповіді, пропускна здатність, tail затримка, поведінка під навантаженням	p50/p95/p99-затримка, RPS, TPS, saturation	Кешування, черги, масштабування, обмеження запитів, оптимізація алгоритмів
Спостережуваність	Здатність пояснювати деградацію через метрики, логи, траси та події	Trace coverage, log completeness, кардинальність, час до отримання доказів (час до отримання доказів (time-to-evidence))	Єдина схема атрибутів, correlation ID, контракт телеметрії, період зберігання policy

Продовження таблиці 1.1

Надійність	Стійкість до часткових відмов, повторних спроб і каскадних ефектів	Error rate, середній час відновлення (середній час відновлення (MTTR)), доступність, retry storm indicators	Timeouts, circuit breakers, bulkheads, graceful degradation
Масштабованість	Можливість зберігати прийнятні SLO при зростанні навантаження	Throughput per replica, queue depth, autoscaling lag	Горизонтальне масштабування, partitioning, backpressure
Керованість витрат	Баланс між деталізацією телеметрії, якістю RCA та вартістю зберігання	Storage cost, samples per second, траса вибіркове збирання ratio	Sampling, downвибіркове збирання, tiered період зберігання, спостережуваність as code

1.2 Інструментальне забезпечення вимірювання продуктивності та критична оцінка меж застосування засобів

Інструментальний контур аналізу продуктивності доцільно будувати як послідовність взаємопов’язаних шарів, а не як список незалежних продуктів. На першому шарі перебуває інструментування застосунків і інфраструктури; на другому – передавання та нормалізація сигналів; на третьому – зберігання й запитування; на четвертому – візуалізація, алертинг і кореляція; на п’ятому – аналітичні та AI/RCA-компоненти. Така декомпозиція допомагає уникнути поширеної помилки, коли в документі перелічено Prometheus, Grafana, Jaeger або ELK, але не пояснено, які саме дані надходять до кожного інструмента, з якою семантикою, з якою періодичністю і для яких рішень вони використовуються.

OpenTelemetry є доцільною основою інструментування, оскільки задає vendor-neutral підхід до збирання traces, metrics і logs, а також дозволяє відокремити інструментування застосунку від конкретного бекенду зберігання [8]. Проте використання OpenTelemetry саме по собі не гарантує якісної спостережуваності. Якщо команди довільно називають атрибути, не підтримують стабільні service.name, розгортання.environment, http.route, db.system або messaging.operation, то кореляція сигналів у масштабі системи стає

крихкою. Тому особливе значення мають semantic conventions, які задають узгоджені назви атрибутів для різних типів операцій [9].

Prometheus доцільно використовувати для числових часових рядів метрик, де важливі регулярне збирання, багатовимірна модель даних і PromQL-запити для сповіщення та побудови панелей спостереження [10]. Його сильна сторона полягає в простоті pull-моделі, зрілості екосистеми exporters і зручності створення правил. Водночас Prometheus не є універсальним сховищем усіх телеметричних даних. Надмірна cardinality мітки, необмежене зберігання високочастотних метрик або спроби зберігати подієву інформацію як часових рядів можуть призвести до зростання вартості, погіршення продуктивності запитів і нестабільності самого моніторингу.

Grafana доцільна як шар інтегрованої візуалізації та алертингу, оскільки може поєднувати різні джерела даних, створювати дашборди, alert rules та інструменти перегляду інцидентів [11]. Однак її практична цінність залежить не від кількості панелей, а від якості інформаційної архітектури дашбордів. Непрофесійний дашборд із десятками графіків без SLO, базовий рівень, annotations і drill-down не скорочує час діагностики. Навпаки, він може створити ілюзію контролю, коли оператор бачить багато сигналів, але не має відповіді на питання: чи порушено користувацький SLO, які компоненти залучені, які зміни передували деградації, які дії дозволені операційна інструкція.

Jaeger або сумісні трасувальні рішення доцільні для аналізу міжсервісних викликів, оскільки tracing дозволяє відстежувати шлях запиту через компоненти розподіленої системи [12]. Проте tracing також має межі. По-перше, sampling може приховати рідкісні, але критичні запити. По-друге, відсутність коректної propagation trace-context руйнує наскрізну картину. По-третє, trace показує часову послідовність і залежності викликів, але не доводить причинність сам по собі. Тому trace має поєднуватися з метриками ресурсів, подіями деплойменту, логами помилок і знанням архітектурних залежностей.

Elastic/ELK або Loki-подібні рішення доцільні для централізованого аналізу логів, але логування має бути цілеспрямованим [13]. Часто в навчальних

роботах логи описуються як “місце, де зберігаються помилки”. Насправді логи мають підтримувати конкретні сценарії діагностики: підтвердження бізнес-події, відтворення послідовності станів, виявлення невдалих інтеграцій, аналіз нестандартних умов. Неструктуровані, шумні або надмірно докладні логи збільшують вартість і погіршують можливість автоматичного аналізу. Недостатні логи, навпаки, не дають підтвердити гіпотезу RCA.

Сервісні рівні та SLO є сполучною ланкою між інструментами та управлінням якістю. У підході SRE SLO не є формальним документом: він задає межу прийнятності сервісу та дозволяє використовувати error budget як керований ресурс ризику [14]. Для аналізу продуктивності це означає, що p95 latency, частка помилок (error rate) або доступність мають трактуватися не як абстрактні “показники”, а як критерії приймання, пов’язані з користувацьким досвідом, режимом навантаження та допустимими наслідками.

Alerting на основі SLO і burn rate дає можливість зменшити шум порівняно зі статичними порогами, хоча також не є універсальним рішенням [15]. Для низькотрафікових сервісів, batch-процесів або систем із сезонним навантаженням burn-rate сповіщення потребує коригування вікон, порогів і способів агрегації. У межах кваліфікаційної роботи це слід відобразити як обмеження: запропонований каталог SLO має бути прикладним, але його пороги потребують калібрування на реальних або відтворюваних даних.

Поведінкові діаграми корисні саме на межі інструментування та процесів. Наприклад, UML-діаграма прецедентів показує, що система обслуговує не лише розробника, а й SRE/DevOps-інженера, аналітика продуктивності та керівника технічного продукту. UML-діаграма послідовності дозволяє описати сценарій “деградація API → збір доказів → формування гіпотези RCA → перевірка → ADR/рекомендація”. UML-діаграма діяльності допомагає операціоналізувати потік: від виявлення порушення SLO до оновлення операційна інструкція і контрольний шлюз продуктивності. Такі діаграми мають цінність лише тоді, коли їхні елементи відповідають реальним артефактам роботи.

Навантажувальне тестування, профілювання й моніторинг мають розглядатися разом. Навантажувальна перевірка без телеметрії показує лише зовнішній симптом; телеметрія без контрольованого навантаження часто не дозволяє відокремити причину від випадкової кореляції; профілювання без архітектурного контексту може оптимізувати локальну ділянку, не змінюючи системне вузьке місце (bottleneck). Тому інструментальний контур повинен містити засоби генерації навантаження, збирання метрик, журналів і трас, а також засоби кореляції результатів із метриками ресурсів, трасами, журналами та подіями змін.

Інструментальний контур аналізу продуктивності та межі його застосування наведено в таблиці 1.2.

Таблиця 1.2 – Інструментальний контур аналізу продуктивності та межі його застосування

Шар	Рекомендовані засоби	Що вимірює / забезпечує	Критичні обмеження
Інструментування	OpenTelemetry SDK, auto-instrumentation, exporters	Єдине джерело траси, metrics, logs, resource атрибути	Неповні semantic conventions, нестабільні атрибути, overhead
Метрики	Prometheus, exporters, Alertmanager	Time-series, SLI/SLO, ресурсні та прикладні показники	Cardinality explosion, період зберігання cost, слабка робота з подіями
Трасування	Jaeger/Tempo, W3C Trace контекст (контекст (Context))	Шлях запиту між сервісами, затримка breakdown	Sampling bias, розрив context propagation, відсутність причинності без перевірки
Логи	Elastic/ELK, Loki, structured logging	Подієвий контекст, помилки, винятки, бізнес-події	Шум, PII-ризики, складність парсингу, нестабільні шаблони
Візуалізація	Grafana панель спостереженняs, сповіщення, annotations	Операційний огляд, drill-down, інцидентна навігація	Dashboard sprawl, відсутність прив’язки до SLO, некерований шум
Тестування	JMeter, Locust, Gatling	Навантажувальні профілі, stress/soak/spike-тести	Нереалістичні сценарії, недостатні дані, складність відтворення експлуатаційне середовище

1.3 Система метрик, критерії приймання та доказова діагностика деградацій

Система метрик у роботі має виконувати не декоративну, а доказову функцію. Метрика корисна тоді, коли її можна пов’язати з конкретною вимогою, сценарієм якості, архітектурним компонентом, тестовим профілем і рішенням, яке прийматиметься за її результатом. Тому каталог метрик слід проєктувати зворотно від питань, на які має відповідати команда: чи порушено SLO; чи деградація локальна або наскрізна; чи вона пов’язана з ресурсами, залежністю або зміною; чи AI/RCA-гіпотеза підтверджується незалежними сигналами; чи можна дозволити реліз.

Метрики latency повинні оцінюватися через розподіл, а не лише через середнє значення. Середня latency може залишатися прийнятною, коли невелика частка користувачів уже відчуває значну деградацію. Тому p95, p99 та аналіз хвостова затримка (tail latency) є більш інформативними для систем із інтерактивним користувацьким досвідом. Водночас tail percentiles нестабільні при малих вибірках, а при неправильному aggregation конвеєр постачання можуть вводити в оману. Тому для кожного percentile потрібно визначити мінімальну кількість спостережень, вікно агрегації, джерело даних і правила інтерпретації.

Метрики throughput також не слід трактувати ізольовано. Зростання RPS може бути позитивним показником, якщо latency і частка помилок (error rate) залишаються в межах SLO, але може бути симптомом retry storm, якщо збільшення кількості запитів супроводжується помилками, повторними спробами й saturation. Для аналізу доцільно фіксувати не лише зовнішній RPS, а й внутрішні виклики між сервісами, кількість повторів, queue depth і backpressure indicators.

Метрики ресурсів слід добирати залежно від архітектури. CPU utilization має різний зміст для CPU-bound обчислень, I/O-bound сервісів і контейнерів із throttling. Memory usage потребує розрізнення heap, RSS, cache, GC pauses, OOM kills. Disk I/O важливий не лише як throughput, а і як queue depth та latency.

Network metrics слід співвідносити з topology і latency між сервісами. Тому USE-підхід, який пропонує дивитися на utilization, saturation і errors, є корисною рамкою, але потребує адаптації до конкретної платформи [18], [19].

Навантажувальні інструменти мають різні сильні сторони. Apache JMeter забезпечує широку підтримку протоколів і підходить для складніших enterprise-сценаріїв [20]. Locust є гнучким варіантом для Python-орієнтованих команд і програмного опису поведінки користувачів [21]. Gatling зручний для сценаріїв із високою продуктивністю та CI/CD-інтеграцією [22]. Однак вибір інструмента не вирішує проблему валідності тесту. Критичними є репрезентативність профілю навантаження, якість тестових даних, контроль середовища, фіксація базовий рівень і повторюваність.

Доказова діагностика деградацій має спиратися на принцип triangulation: гіпотеза про причину інциденту повинна підтверджуватися кількома незалежними сигналами. Якщо зросла p99 latency, слід перевірити, чи є зміна у trace breakdown, чи зросла saturation ресурсу, чи є відповідні лог-події, чи була зміна конфігурації або реліз, чи впливає проблема на всі маршрути або окремий endpoint. Таке зіставлення не усуває невизначеність повністю, але зменшує ризик хибної RCA-рекомендації.

Для операціоналізації метрик у роботі доцільно створити каталог SLO. У ньому для кожного критичного сценарію фіксуються користувацька операція, SLI, SLO, вікно вимірювання, джерело даних, допустимі винятки, алерт, відповідальний власник і пов’язаний операційна інструкція. Такий каталог перетворює метрики на інструмент управління якістю. Без нього дашборди можуть бути інформативними, але вони не встановлюють, яка поведінка системи є прийнятною.

Критерії приймання для AI/RCA не повинні обмежуватися загальною точністю (accuracy). У практичній системі важливі точність позитивних спрацювань (precision), частка хибних спрацювань (false positive rate), час до отримання доказів (time-to-evidence), пояснюваність, покриття класів інцидентів, вартість однієї діагностики і здатність підтримувати експертну перевірку (human

review). Якщо модель часто пропонує правдоподібні, але неперевірені пояснення, вона може збільшити когнітивне навантаження інженера, а не зменшити його. Тому архітектура повинна передбачати механізми валідації гіпотез, фіксації доказів і накопичення зворотного зв'язку.

Каталог метрик і критеріїв приймання для прототипу платформи наведено в таблиці 1.3.

Таблиця 1.3 – Каталог метрик і критеріїв приймання для прототипу платформи

Група	SLI / показник	Приклад SLO або критерій	Джерело даних	Рішення за результатом
Latency	p95/p99 HTTP request duration	p95 ≤ 500 мс для пошуку при 300 RPS; p99 аналізується окремо як ризик tail затримка	OTEL metrics, Prometheus, траси	Дозвіл релізу, оптимізація endpoint, перевірка залежностей
Errors	5xx rate, failed dependency calls	≤ 0,1 % за 30 хвилин тесту; окремо фіксуються retry errors	Prometheus, logs, траси	Зупинка релізу, rollback, аналіз circuit breaker
Saturation	CPU throttling, memory pressure, queue depth	Не має стійкого saturation > 80 % у steady-state; queue depth не зростає монотонно	Node/container metrics, exporters	Зміна resource limits, autoscaling, backpressure
Telemetry quality	Trace coverage, log correlation completeness	≥ 90 % критичних запитів мають traca_id; логи помилок містять correlation ID	OTEL Collector, log конвеєр	Виправлення instrumentation перед ШІ/RCA
RCA usefulness	Time-to-evidence, verified hypothesis ratio	Гіпотеза RCA містить щонайменше 3 незалежні докази; картка доказів заповнено	MELT + change events	Прийняття або відхилення AI-рекомендації

У підсумку розділ 1 задає методологічну позицію роботи: продуктивність ПЗ має аналізуватися як архітектурно обумовлена властивість, а не як набір окремих графіків; інструменти мають добиратися за роллю в доказовому процесі, а не за популярністю; метрики мають бути пов'язані з SLO, сценаріями якості, тестовими профілями та операційними рішеннями. Такий підхід створює основу для подальшого архітектурного моделювання запропонованої платформи.

РОЗДІЛ 2 АРХІТЕКТУРНЕ МОДЕЛЮВАННЯ ПЛАТФОРМИ АНАЛІЗУ ПРОДУКТИВНОСТІ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

2.1 Архітектурні драйвери та межі автоматизації аналізу продуктивності

Архітектурні драйвери запропонованої платформи випливають не з бажання використати AI, а з проблем, які виникають під час аналізу продуктивності в розподілених системах. До таких проблем належать фрагментованість телеметрії, різна семантика сигналів у командах, складність кореляції між метриками, логами й трасами, помилкові статичні пороги, затримка між інцидентом і його поясненням, недостатня доказовість аналіз першопричин (першопричина analysis), а також складність повторення performance-тестів у CI/CD. Ці драйвери задають вимоги до архітектури: вона має забезпечувати не просто збір даних, а контрольований перехід від спостереження до рішення.

У QAW-підході якісний сценарій має містити джерело стимулу, сам стимул, середовище, артефакт, відповідь системи та вимір відповіді. Для цієї роботи найбільш значущими є сценарії продуктивності, спостережуваності, надійності, масштабованості, безпеки телеметрії та вартості. Такий набір не є вичерпним, але він дозволяє уникнути одновимірного трактування “покращення продуктивності”. Наприклад, рішення з повним трасуванням усіх запитів може бути корисним для діагностики, але неприйнятним для high-throughput системи без sampling і data retention policy.

Перший критичний сценарій пов’язаний із погіршенням p95 latency після релізу. Джерелом стимулу є нова версія сервісу або зміна конфігурації; середовищем – передпромисловий стенд або наближений до експлуатаційного стенд; артефактом – API-сервіс і залежні компоненти; очікуваною відповіддю – автоматична фіксація відхилення від базовий рівень, прив’язка до розгортання event, формування картка доказів і рекомендація щодо rollback або поглибленого

профілювання. Важливо, що AI/RCA тут ранжує гіпотези, які мають бути підтверджені доказами.

Другий сценарій стосується часткової відмови залежності. Якщо база даних або зовнішній сервіс відповідає повільніше, система може демонструвати retry storm, queue accumulation і каскадне зростання латентності. Архітектура платформи має виявляти не лише зовнішній симптом, а й шлях поширення (propagation path). Для цього потрібні трасувальний контекст (trace-context), метрики повторів, логування подій перевищення часу очікування та спрацювання запобіжника (timeout/circuit-breaker) і дашборд, який дозволяє перейти від порушень SLO до конкретної залежності.

Третій сценарій пов'язаний із якістю телеметрії. Якщо сервіс не передає trace_id, не має стабільного атрибута service.name або не фіксує ключову бізнес-операцію, RCA може стати статистично переконливою, але інженерно бездоказовою. Тому в архітектурі потрібен контрольний шлюз якості телеметрії (telemetry quality gate): перевірка наявності обов'язкових атрибутів, покриття критичних кінцевих точок (endpoints), валідність кардинальності міток (label cardinality), відповідність семантичним угодам OpenTelemetry та наявність кореляційних ідентифікаторів.

Четвертий сценарій пов'язаний із CI/CD. Під час запиту на злиття (pull request) або перед релізом запускається контрольована навантажувальна перевірка; результати порівнюються з базовим рівнем; якщо p95 latency, частка помилок (error rate) або насичення ресурсів (resource saturation) виходять за допустимі межі, конвеєр постачання має повернути звіт із практичними діями. Важливо, щоб цей звіт не був лише посиланням на графіки, а містив прив'язку до коміту, конфігурації, тестових даних, профілю навантаження, SLO і первинної гіпотези про деградацію.

П'ятий сценарій стосується безпеки та приватності телеметрії. Логи і траси можуть містити персональні дані, токени, параметри запитів або бізнес-чутливу інформацію. З цієї причини платформа має підтримувати redaction, контроль доступу, мінімізацію даних та retention policy. Без цих тактик упровадження

AI/RCA може збільшити ризик витоку даних, оскільки аналітичні компоненти часто агрегують сигнали з багатьох джерел.

Межі автоматизації слід фіксувати прямо в архітектурі. Автоматичне виявлення аномалій може помилятися при сезонності, зміні профілю трафіку, неповній історії даних або зміні архітектури. RCA-модель може ранжувати компонент як підозрілий через кореляцію, не доводячи причинність. Тому архітектура повинна включати human-in-the-loop: інженер перевіряє картка доказів, приймає рішення про відновлювальна дія, фіксує ADR або оновлює операційна інструкція. Такий підхід знижує ризик безконтрольної автоматизації, але збільшує вимоги до дисципліни документування.

Практичним результатом QAW-аналізу є набір сценаріїв, які визначають подальше моделювання, інструментування, тестування й оцінювання (табл. 2.1). Ці сценарії мають бути відображені в C4-моделі, UML-діаграмах послідовності та діяльності, а також контрольних шлюзах продуктивності і реєстрі ризиків.

QAW-сценарії якості для платформи аналізу продуктивності на основі штучного інтелекту наведено в таблиці 1.2.

Таблиця 2.1 – QAW-сценарії якості для платформи аналізу продуктивності на основі штучного інтелекту

ID	Стимул і середовище	Артефакт	Очікувана відповідь	Вимір відповіді
QA-01	Після релізу на передпромисловий стенд p95-затримка API зростає порівняно з базовий рівень	API service, metrics, траси, розгортання event	Платформа фіксує SLO deviation, формує картка доказів і вказує підозрілі зміни	час до отримання доказів (час до отримання доказів (time-to-evidence)) ≤ 10 хв; не менше 3 доказів у картці
QA-02	Зовнішня залежність відповідає повільніше під час навантажувальна перевірка	Dependency client, queue, retry policy	Виявлено propagation path, retry amplification і компонент, який потребує circuit breaker	коректна ідентифікація dependency; зниження хибне спрацюванняs
QA-03	Сервіс передає неповну телеметрію перед підключенням ШІ/RCA	Контракт телеметрії, OTEL Collector	Quality gate блокує RCA та повертає список відсутніх атрибутів	100 % mandatory атрибути або documented exception

Продовження таблиці 2.1

QA-04	Pipeline виконує performance test перед merge/release	CI/CD, load generator, metrics storage	Сформовано пройдено / не пройдено (пройдено / не пройдено (pass/fail)) report з посиланням на базовий рівень, SLO і відхилення регресії	автоматичний verdict; артефакт збережено в build history
QA-05	Логи містять потенційно чутливі значення	Log конвеєр, storage, панель спостереження	Redaction і access policy застосовано до потрапляння в аналітичний шар	відсутність PII у sampled logs; audit trail доступу

Рисунок 2.1 відображає типовий сценарій аналізу першопричини деградації API у межах платформи аналізу продуктивності програмного забезпечення. Діаграма послідовності деталізує взаємодію між інженером експлуатації, системою моніторингу, сервісом аналізу першопричин, сховищами телеметричних сигналів, розробником і каталогом архітектурних рішень та операційних інструкцій. На відміну від структурних C4-діаграм, які описують склад системи та її контейнери, ця діаграма показує поведінковий аспект архітектури: як саме виявлена деградація перетворюється на перевірювану гіпотезу, інженерне рішення та подальше вдосконалення процесу супроводу.

Сценарій починається з того, що система моніторингу фіксує порушення цілі рівня обслуговування для певного API. Таке порушення може проявлятися через зростання p95 або p99 затримки, збільшення частки помилкових відповідей, перевищення допустимого часу очікування, спрацювання запобіжника міжсервісної взаємодії або зменшення пропускної здатності. Інженер експлуатації відкриває інцидент і відповідну панель спостереження, після чого система моніторингу передає сервісу аналізу першопричин початковий контекст деградації: назву сервісу, кінцеву точку API, порушену SLO-умову, часове вікно, середовище виконання та інші ознаки, необхідні для подальшого аналізу.

Після отримання початкового контексту сервіс аналізу першопричин формує межі інциденту. На цьому етапі уточнюється часовий інтервал

деградації, визначаються зачеплені сервіси, операції, версії розгортання, середовище виконання та пов’язані технічні події. Це необхідно для того, щоб аналіз не виконувався над надмірно широким набором даних. Замість перегляду всієї доступної телеметрії сервіс концентрується на релевантному фрагменті поведінки системи, що зменшує інформаційний шум і підвищує точність кореляції.

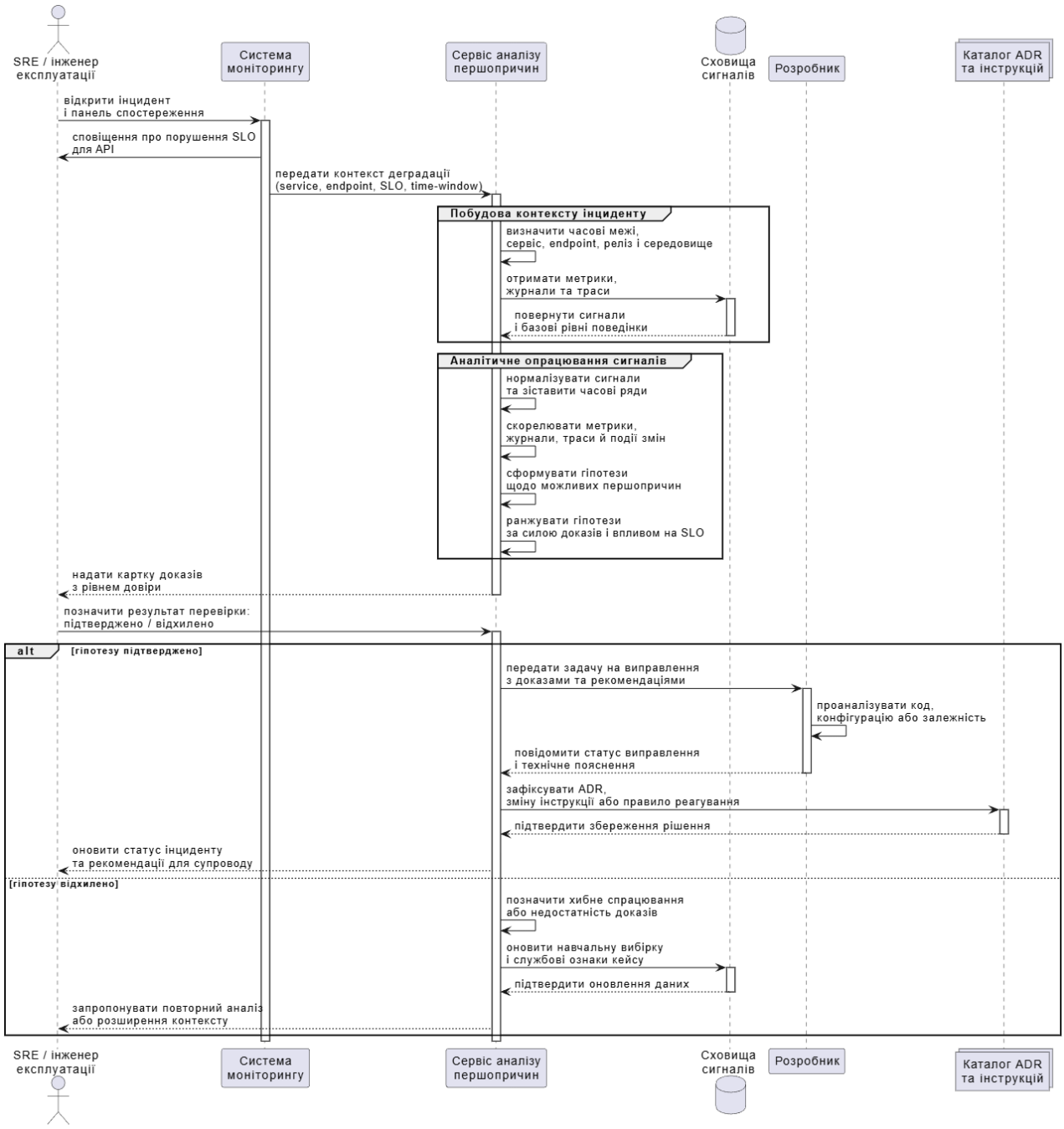

Рисунок 2.1 – UML-діаграма послідовності сценарію аналізу першопричини при деградації API

Далі сервіс аналізу першопричин звертається до сховищ сигналів і отримує метрики, журнали та розподілені траси. Метрики дозволяють оцінити зміну кількісних показників у часі: затримку, кількість запитів, частку помилок, використання процесора, пам’яті або мережі. Журнали дають змогу виявити конкретні помилки, попередження, винятки, повторні спроби запитів або нестандартні стани виконання. Траси показують шлях запиту через кілька сервісів і допомагають визначити, на якому етапі виникає затримка. Повернення базових рівнів поведінки є важливим, оскільки деградація повинна оцінюватися не лише відносно абсолютного порога, а й порівняно з типовою поведінкою системи в аналогічних умовах навантаження.

Після отримання даних сервіс виконує аналітичне опрацювання сигналів. Спочатку відбувається нормалізація: узгодження часових міток, назв сервісів, кодів операцій, ідентифікаторів трас і форматів подій. Далі система зіставляє метрики, журнали, траси та події змін у межах одного часового вікна. Наприклад, зростання затримки API може бути пов’язане зі збільшенням часу відповіді залежного сервісу, появою помилок у журналах, збільшенням кількості повторних спроб або розгортанням нової версії компонента. На основі цих зв’язків сервіс формує кілька гіпотез щодо можливих першопричин деградації та ранжує їх за силою доказів і впливом на цілі рівня обслуговування.

Результатом роботи сервісу є картка доказів, яка передається інженеру експлуатації. Вона повинна містити не лише текстовий висновок, а й перевірювані підстави для нього: часові межі деградації, зачеплені сервіси, ключові метрики, приклади трас, журнальні події, пов’язані зміни та рівень довіри до гіпотези. Такий підхід принципово відрізняється від простого автоматичного сповіщення, оскільки інженер отримує не лише сигнал про проблему, а структурований аналітичний матеріал для прийняття рішення.

Подальший етап передбачає людську валідацію. Інженер експлуатації перевіряє запропоновану гіпотезу та позначає її як підтверджену або відхилену. Така дія є важливою з двох причин. По-перше, результати AI-аналізу не повинні розглядатися як безумовна істина, оскільки якість висновку залежить від

повноти телеметрії, коректності інструментування, стабільності базових рівнів і якості моделі. По-друге, зворотний зв’язок від інженера може використовуватися для вдосконалення правил аналізу, навчальної вибірки або порогів спрацювання.

Якщо гіпотезу підтверджено, сервіс аналізу першопричин передає розробнику задачу на виправлення. На відміну від загального повідомлення про інцидент, така задача містить докази та рекомендації: які сервіси зачеплено, які метрики погіршилися, які траси підтверджують проблему, який реліз або конфігураційна зміна може бути пов’язана з деградацією. Розробник аналізує код, конфігурацію або залежність, виконує необхідні зміни та повертає статус виправлення разом із технічним поясненням. Це дозволяє зв’язати експлуатаційний інцидент із конкретною інженерною дією.

Після підтвердження та усунення проблеми результат фіксується в каталозі ADR та операційних інструкцій. Якщо деградація була пов’язана з архітектурним рішенням, наприклад некоректною політикою кешування, синхронною взаємодією між сервісами, недостатнім масштабуванням або помилковими значеннями тайм-аутів, доцільно оформити відповідний запис архітектурного рішення. Якщо ж проблема має експлуатаційний характер, наприклад потребує іншого порядку реагування, зміни порогів сповіщення або уточнення процедури діагностики, оновлюється операційна інструкція. Завдяки цьому результат інциденту не втрачається після локального виправлення, а стає частиною повторно використовуваного інженерного знання.

Якщо гіпотезу відхилено, сервіс аналізу першопричин позначає результат як хибне спрацювання або як випадок із недостатньою доказовою базою. Після цього оновлюється навчальна вибірка або службові ознаки кейсу. Такий механізм потрібний для зниження кількості повторних помилкових висновків. У цьому випадку платформа може запропонувати повторний аналіз із розширеним контекстом, наприклад із більшим часовим вікном, додатковими метриками, трасами суміжних сервісів або врахуванням інфраструктурних подій.

Таким чином, наведена діаграма послідовності демонструє не лише технічний обмін повідомленнями між компонентами, а повний операційний цикл

доказового аналізу деградації API. Сценарій поєднує автоматичне виявлення порушення SLO, збір і кореляцію телеметрії, формування гіпотези, людську перевірку, створення задачі на виправлення, документування архітектурного або експлуатаційного рішення та оновлення навчальних даних у разі помилкового спрацювання. Саме така послідовність дозволяє операціоналізувати AI-аналіз продуктивності: перевести його з рівня абстрактної аналітики до керованого процесу супроводу програмної системи.

2.2 Архітектурний контекст платформи аналізу продуктивності

Для архітектурного опису запропонованої платформи використано C4-модель, оскільки вона дає змогу послідовно перейти від меж системи до контейнерів і компонентів без передчасного занурення у структуру коду [7]. У роботі деталізовано рівні контекст (Context), контейнер (Container) та компонент (Component). Рівень Code не розглядається окремо, оскільки практичний фокус роботи спрямований не на реалізацію окремого класу чи модуля, а на архітектуру інструментального контуру аналізу продуктивності.

Діаграма контексту (рис. 2.2) показує, що платформа не є самодостатньою «чорною скринькою». Вона функціонує на перетині розроблення, експлуатації, аналізу продуктивності та архітектурного управління. Це має практичне значення: якщо команда експлуатації не володіє цільовими рівнями сервісу, розробники не підтримують контракт телеметрії, а технічний керівник не використовує картки доказів під час оцінювання релізів, автоматизація аналізу першопричин залишатиметься частковою. Тому контекстна модель використовується як засіб узгодження відповідальності.

Платформа аналізу продуктивності програмного забезпечення на основі штучного інтелекту працює як інтеграційний центр між програмними системами, середовищем виконання, стеком спостережуваності, процесами постачання програмного забезпечення, керуванням інцидентами та життєвим циклом моделей машинного навчання. На цьому рівні деталізація окремих контейнерів і сервісів навмисно обмежена, оскільки головним завданням C4-

рівня контексту є визначення меж системи, зовнішніх залежностей, інженерних ролей і ключових потоків взаємодії.

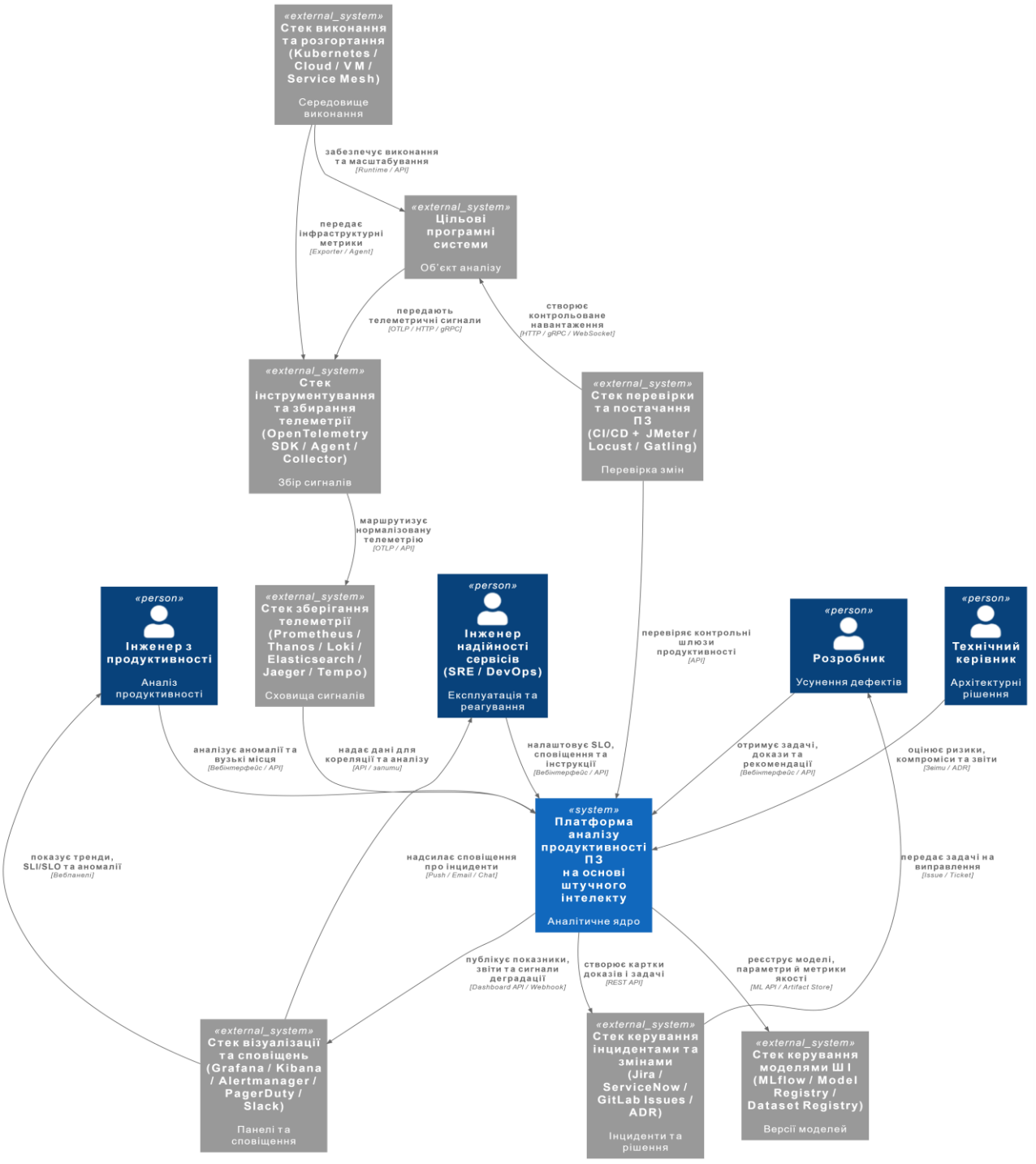

Рисунок 2.2 – C4-рівень контексту: місце платформи аналізу продуктивності на основі ШІ в операційному середовищі

У межах діаграми користувачі платформи подані як учасники операційного процесу аналізу продуктивності. Інженер з продуктивності працює

з платформою для виявлення деградацій, перевірки гіпотез щодо вузьких місць і зіставлення телеметричних сигналів із фактичними змінами у програмній системі. Для цієї ролі важливими є часові ряди затримок, пропускної здатності, частоти помилок, використання ресурсів, а також можливість переходу від агрегованого показника до конкретної траси або журнальної події. Платформа в цьому контексті не лише відображає стан системи, а допомагає скоротити простір пошуку причини проблеми.

Інженер надійності сервісів відповідає за експлуатаційний аспект використання платформи. Його взаємодія з системою пов’язана з налаштуванням цілей рівня обслуговування, правил сповіщення, політики зберігання телеметричних даних, операційних інструкцій і процедур реагування на інциденти. Для цієї ролі особливо важливо, щоб платформа не створювала надмірної кількості хибних сповіщень, не дублювала існуючі панелі спостереження, а формувала пріоритезовані сигнали з поясненням, які саме метрики, журнали, траси або зміни стали підставою для висновку про деградацію.

Розробник використовує платформу переважно на етапі усунення дефектів продуктивності. У типовому сценарії він отримує не загальне повідомлення про погіршення роботи сервісу, а задачу, доповнену карткою доказів: пов’язаними метриками, фрагментами трасування, журнальними подіями, версією розгортання, часом появи деградації та ймовірною першопричиною. Такий підхід зменшує розрив між експлуатаційним виявленням проблеми та програмною дією з її усунення, оскільки розробник отримує не лише факт інциденту, а технічний контекст для перевірки й виправлення.

Технічний керівник розглядає платформу як джерело обґрунтованих архітектурних і процесних рішень. Для цієї ролі важливі не окремі сигнали, а узагальнені висновки щодо ризиків, компромісів і вартості змін. Наприклад, якщо платформа систематично фіксує деградацію під час зростання навантаження на певний сервіс, це може бути підставою не лише для локальної оптимізації коду, а й для перегляду архітектурної тактики: кешування,

асинхронної взаємодії, масштабування, розподілу відповідальності між сервісами або зміни політики обробки запитів. У цьому випадку результати аналізу можуть оформлюватися як записи архітектурних рішень, ризики або рекомендації для подальшої еволюції системи.

Стек виконання та розгортання формує операційне середовище, у якому працюють цільові програмні системи. До нього можуть належати Kubernetes, хмарна інфраструктура, віртуальні машини, балансувальники навантаження, сервісна сітка та інші засоби підтримки виконання. У контексті діаграми цей стек не є безпосередньою частиною платформи аналізу, але суттєво впливає на характер зібраної телеметрії. Саме на рівні середовища виконання виникають ресурсні обмеження, перезапуски контейнерів, зміни кількості реплік, мережеві затримки, обмеження процесора або пам’яті. Тому інтеграція платформи з цим стеком необхідна для розрізнення прикладної деградації та інфраструктурних причин.

Цільові програмні системи є основним об’єктом аналізу. Вони можуть складатися з вебзастосунків, мікросервісів, програмних інтерфейсів, фонових обробників, черг повідомлень, баз даних і зовнішніх залежностей. Практично важливо, що запропонована платформа не вимагає радикальної перебудови цих систем. Інтеграція може здійснюватися через бібліотеки інструментування, агенти, проксі-компоненти або експортери, які передають телеметрію у стандартизованому форматі. Це знижує вартість впровадження та дозволяє підключати систему поступово: спочатку критичні сервіси, далі допоміжні компоненти та зовнішні інтеграції.

Стек інструментування та збирання телеметрії виконує роль вхідного шару для всіх сигналів спостережуваності. Його доцільно будувати на основі OpenTelemetry SDK, агентів автоматичного інструментування та OpenTelemetry Collector. На цьому рівні формуються метрики, журнали, розподілені траси та службові події виконання. Важливо, що саме тут визначається якість подальшого аналізу: неповні назви операцій, відсутні ідентифікатори трас, неузгоджені часові позначки або різні назви одного й того самого сервісу можуть суттєво

знизити точність кореляції. Тому впровадження платформи має супроводжуватися формуванням контракту телеметрії, який визначає обов'язкові атрибути, правила іменування сервісів, коди операцій, рівні журналювання та мінімальний набір метрик.

Стек зберігання телеметрії акумулює різні типи сигналів у спеціалізованих сховищах. Метрики доцільно зберігати в Prometheus-сумісному середовищі, журнали – у Loki або Elasticsearch, а розподілені траси – у Jaeger, Tempo чи Zipkin. Такий поділ зумовлений різною природою даних: часові ряди потребують швидкої агрегації, журнали – пошуку й фільтрації, а траси – відновлення шляху запиту через кілька сервісів. У межах інтеграційної архітектури платформа ШІ-аналізу не підміняє ці сховища, а працює над ними як кореляційний і аналітичний шар. Це дозволяє зберегти існуючу інфраструктуру спостережуваності й поступово додати до неї інтелектуальні функції.

Стек візуалізації та сповіщень забезпечує подання результатів аналізу інженерам і передавання сигналів про деградацію. До нього можуть входити Grafana, Kibana, Alertmanager, PagerDuty, Slack або інші засоби доставлення повідомлень. Його інтеграційна роль полягає у тому, щоб зробити результати платформи доступними в тих каналах, якими вже користуються команди розроблення та експлуатації. При цьому важливо не обмежуватися дублюванням графіків. Платформа має доповнювати панелі спостереження пояснювальними даними: імовірною причиною, рівнем упевненості, пов'язаними змінами, критичними трасами, журнальними подіями та рекомендаціями щодо перевірки.

Стек перевірки та постачання програмного забезпечення поєднує CI/CD-конвеєр із засобами навантажувального тестування, такими як JMeter, Locust або Gatling. У межах запропонованої архітектури цей стек забезпечує перехід від реактивного аналізу продуктивності до превентивного контролю змін. Під час підготовки нової версії системи конвеєр може запускати навантажувальні або регресійні сценарії, передавати результати до платформи та отримувати рішення щодо проходження контрольного шлюзу продуктивності. Якщо нова версія погіршує p95-затримку, збільшує частоту помилок або створює надмірне

навантаження на ресурси, розгортання може бути зупинене до додаткового аналізу.

Стек керування інцидентами та змінами перетворює аналітичні результати на керовані інженерні дії. До нього можуть належати Jira, ServiceNow, GitLab Issues, записи архітектурних рішень та інші засоби фіксації змін. Платформа може створювати або доповнювати інциденти картками доказів, посиланнями на графіки, трасами, журналами, гіпотезами щодо першопричин і рекомендаціями щодо усунення. Така інтеграція важлива тому, що без включення в процес керування задачами навіть якісний аналітичний висновок може залишитися лише повідомленням у панелі спостереження. Операціоналізація результату означає, що кожна суттєва деградація повинна мати відповідального виконавця, технічний контекст, пріоритет і простежуваний результат виправлення.

Стек керування моделями ШІ забезпечує контроль життєвого циклу моделей, які використовуються для виявлення аномалій, прогнозування деградацій або ранжування першопричин. У ньому зберігаються версії моделей, параметри навчання, набори даних, пороги спрацювання, метрики якості та історія змін. Його наявність є важливою умовою керованості AI-компонентів, оскільки висновки моделі мають бути відтворюваними та пояснюваними. Якщо після оновлення моделі збільшується кількість хибних спрацьовувань або пропущених інцидентів, команда повинна мати можливість повернутися до попередньої версії, порівняти метрики якості та оцінити причини зміни поведінки.

Центральна платформа аналізу продуктивності ПЗ на основі штучного інтелекту виконує роль аналітичного ядра, що поєднує зазначені стеки в єдиний інженерний процес. Її основними функціями є кореляція метрик, журналів і трас; виявлення аномальних відхилень; зіставлення деградацій із релізами, змінами конфігурації або інфраструктурними подіями; формування карток доказів; ранжування гіпотез щодо першопричин; передавання результатів у панелі спостереження, системи сповіщень і задачі на виправлення. У зрілій реалізації платформа не повинна приймати критичні рішення автономно без участі

інженера. Її призначення полягає в тому, щоб підвищити якість і швидкість аналізу, але остаточне рішення щодо виправлення, масштабування або архітектурної зміни має залишатися в зоні відповідальності фахівців.

Інтеграційний контекст діаграми демонструє, що практичне впровадження такої платформи залежить не лише від вибору конкретних інструментів, а й від зрілості процесів. Для коректної роботи необхідні узгоджене інструментування сервісів, стабільна політика зберігання телеметричних даних, визначені SLI/SLO, правила сповіщення, контрольні шлюзи продуктивності, процес керування інцидентами та життєвий цикл моделей. Без цих елементів платформа ризикує перетворитися на ще один інструмент візуалізації, який створює додатковий інформаційний шум. За умови правильної інтеграції вона може стати засобом доказового аналізу продуктивності, що поєднує експлуатаційні сигнали, інженерні рішення та процеси постачання програмного забезпечення.

2.3 Попередня архітектура платформи аналізу продуктивності

Рисунок 2.3 деталізує попередню контекстну модель до рівня C4 Container і показує внутрішній інструментальний контур платформи аналізу продуктивності. Контейнерний рівень фокусується на тому, з яких логічно відокремлених частин складається сама платформа та як між ними проходять потоки даних. Зовнішні залежності навмисно залишено узагальненими. Стек спостережуваності представляє наявну інфраструктуру збирання, зберігання та доступу до метрик, журналів і трас. Стек перевірки та постачання ПЗ охоплює CI/CD-конвеєр і засоби навантажувального тестування. Стек візуалізації та сповіщень, стек керування інцидентами та змінами, а також стек керування моделями ШІ залишено поза межами платформи, оскільки вони можуть існувати в організації незалежно від запропонованого рішення. Такий підхід зменшує перевантаження діаграми й дозволяє зосередити увагу на контейнерах платформи.

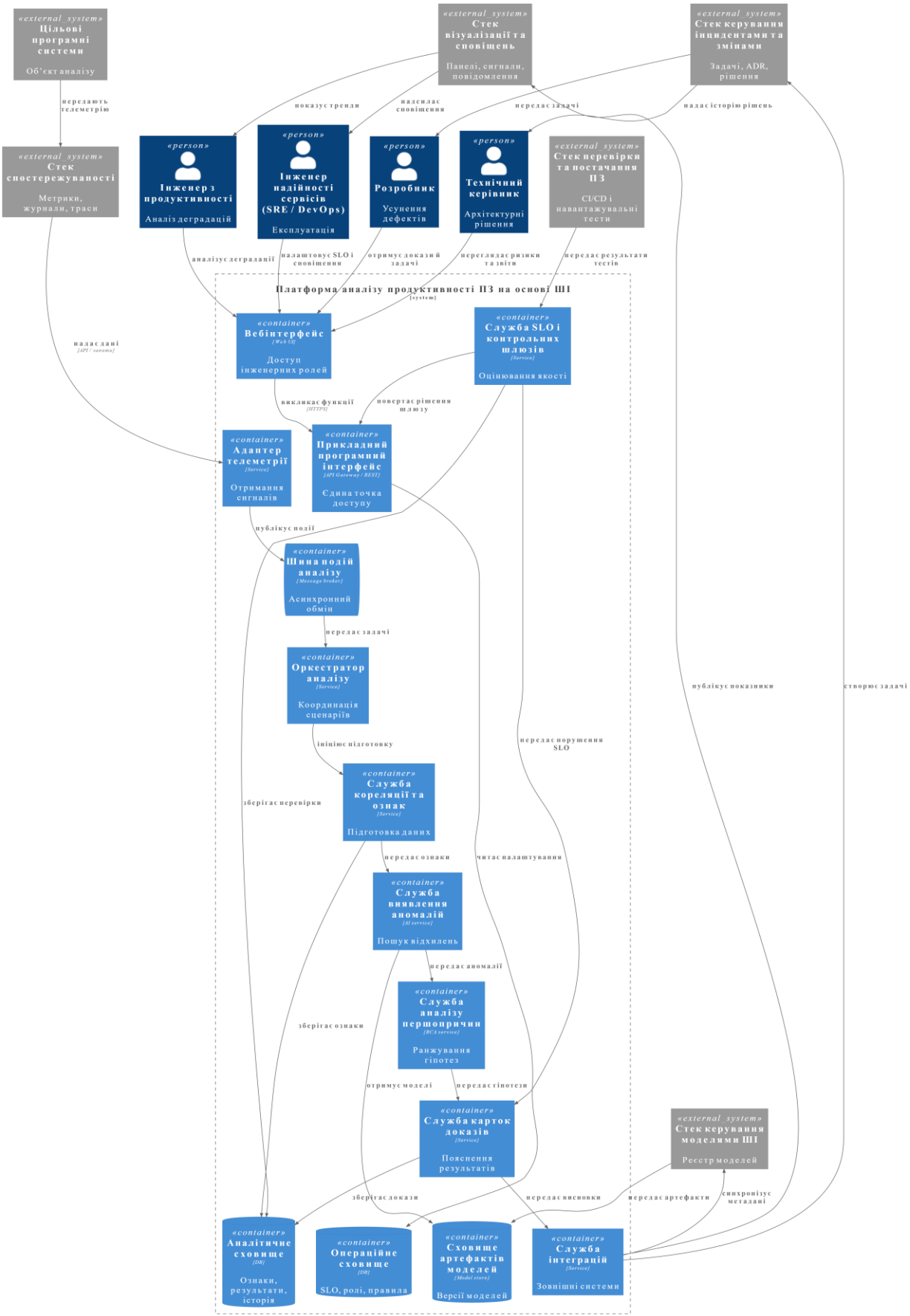

Рисунок 2.3 – C4-рівень контейнерів (Container): інструментальний контур платформи аналізу продуктивності

Вебінтерфейс забезпечує роботу інженерних ролей із платформою. Через нього інженер з продуктивності аналізує деградації й вузькі місця, інженер надійності сервісів налаштовує SLO та правила реагування, розробник отримує докази й рекомендації, а технічний керівник переглядає ризики, компроміси та звіти. Вебінтерфейс не повинен безпосередньо звертатися до сховищ або аналітичних служб. Його взаємодія проходить через прикладний програмний інтерфейс, який виконує роль єдиної точки доступу до функцій платформи.

Прикладний програмний інтерфейс виконує функції маршрутизації запитів, автентифікації, авторизації, перевірки прав доступу та узгодження взаємодії між користувацьким інтерфейсом і внутрішніми службами. Через нього можуть бути доступні сценарії перегляду карток доказів, керування SLO, запуску аналізу, отримання звітів і перевірки контрольних шлюзів продуктивності. Наявність такого контейнера знижує зв’язність між інтерфейсом користувача та внутрішніми механізмами аналізу.

Адаптер телеметрії є вхідною точкою для даних із зовнішнього стеку спостережуваності. Він не замінює Prometheus, Loki, Elasticsearch, Jaeger або інші засоби, а отримує з них дані через прикладні програмні інтерфейси або запити. Його завдання полягає в уніфікації доступу до різнорідних джерел, перетворенні зовнішніх форматів до внутрішньої моделі платформи та передаванні подій у шину аналізу. Завдяки цьому зміна конкретного інструмента зберігання телеметрії не повинна вимагати перероблення аналітичних служб.

Шина подій аналізу забезпечує асинхронний обмін між контейнерами. Її доцільність зумовлена тим, що аналіз продуктивності часто виконується не як миттєвий синхронний запит, а як ланцюг обробки: отримання телеметрії, формування ознак, виявлення аномалій, аналіз першопричин, підготовка картки доказів і передавання результатів у зовнішні системи. Шина подій дозволяє розділити ці етапи, зменшити взаємну залежність контейнерів і масштабувати окремі частини контуру незалежно.

Оркестратор аналізу координує виконання аналітичних сценаріїв. Він визначає, які саме служби потрібно задіяти для конкретного випадку: кореляцію

ознак, виявлення аномалій, аналіз першопричин, перевірку SLO або формування рекомендацій. У практичній реалізації такий контейнер може працювати як окремий сервіс, що приймає події з шини, створює задачі обробки та контролює їхній стан. Його наявність важлива для керованості платформи, оскільки без центральної координації аналітичний процес легко перетворюється на набір слабко узгоджених скриптів.

Служба кореляції та ознак перетворює первинні телеметричні сигнали на узгоджене подання для подальшого аналізу. Вона зіставляє метрики, журнали й траси за часом, назвами сервісів, версіями розгортання, ідентифікаторами трас, кодами помилок та іншими атрибутами. Результатом її роботи є підготовлені ознаки, агрегати й зв’язки, які зберігаються в аналітичному сховищі та передаються до служби виявлення аномалій. Саме якість цього контейнера значною мірою визначає надійність подальших ШІ-висновків.

Служба виявлення аномалій застосовує моделі машинного навчання або статистичні методи для пошуку відхилень у поведінці системи. Вона може аналізувати часові ряди затримок, пропускної здатності, частоти помилок, використання ресурсів, кількості повторних спроб, перевищень часу очікування та спрацювань запобіжників. Важливо, що результати цієї служби не повинні трактуватися як остаточна істина. Вони є вхідними гіпотезами для подальшого аналізу першопричин і мають супроводжуватися метриками впевненості, часовим інтервалом, ознаками, які вплинули на рішення, та посиланнями на первинні дані.

Служба аналізу першопричин працює з виявленими аномаліями та намагається визначити найбільш імовірні джерела деградації. Вона може зіставляти аномалії з релізами, змінами конфігурації, інфраструктурними подіями, трасами критичних запитів, журнальними помилками й порушеннями SLO. Її завдання полягає не в автоматичному призначенні винного компонента, а у формуванні ранжованого набору гіпотез, які можуть бути перевірені інженером.

Служба рекомендацій і карток доказів перетворює результати аналізу на зрозумілий інженерний артефакт. Картка доказів має містити опис проблеми, часові межі деградації, пов’язані метрики, приклади трас, журнальні події, імовірні першопричини, рівень упевненості та рекомендовані дії. Саме цей контейнер забезпечує практичну цінність платформи, оскільки зменшує розрив між аналітичним висновком і конкретною задачею для команди розроблення або експлуатації.

Служба SLO та контрольних шлюзів відповідає за перевірку цілей рівня обслуговування і контроль продуктивності під час постачання змін. Вона приймає результати навантажувальних тестів із CI/CD-конвеєра, порівнює їх із встановленими порогами та повертає рішення про проходження або непроходження контрольного шлюзу. Завдяки цьому платформа може використовуватися не лише для аналізу інцидентів у продукційному середовищі, а й для превентивного виявлення регресій продуктивності до розгортання нової версії.

Служба інтеграцій відповідає за обмін результатами з зовнішніми системами: панелями спостереження, засобами сповіщення, системами керування інцидентами, репозиторіями архітектурних рішень і реєстрами моделей. Її виокремлення в окремий контейнер знижує залежність аналітичного ядра від конкретних зовнішніх інструментів. Наприклад, заміна Jira на GitLab Issues або перехід від одного засобу сповіщень до іншого не повинні змінювати логіку виявлення аномалій чи аналізу першопричин.

Операційне сховище використовується для зберігання конфігурацій платформи, ролей користувачів, правил доступу, SLO, політик сповіщення, шаблонів карток доказів та інших службових налаштувань. Аналітичне сховище призначене для зберігання підготовлених ознак, результатів аналізу, історії деградацій, карток доказів і статистики роботи платформи. Сховище артефактів моделей зберігає версії моделей, параметри, пороги, метрики якості та зв’язок між версією моделі й результатами її застосування.

У цілому контейнерна модель демонструє, що платформа є не одним монолітним AI-модулем, а сукупністю взаємопов’язаних контейнерів, кожен із яких має окрему відповідальність. Такий поділ підвищує супроводжуваність, спрощує інтеграцію з наявними інструментами, дозволяє масштабувати ресурсоємні етапи аналізу та забезпечує кращу простежуваність результатів.

Контейнерна діаграма дозволяє побачити ключовий компроміс: що більше сигналів збирає платформа, то вищою може бути якість діагностики, але водночас зростають витрати на зберігання, обробку, захист і підтримку даних. Тому підсистема приймання телеметрії має підтримувати вибіркове збирання, маскування чутливих даних і маршрутизацію сигналів. Метрики, журнали й траси не повинні зберігатися як три ізольовані архіви; між ними мають зберігатися спільні часові мітки, кореляційні ідентифікатори, мітки сервісів і позначки розгортання.

2.4 Сервіс аналізу першопричин на основі штучного інтелекту

Рисунок 2.4 деталізує внутрішню будову служби аналізу першопричин на основі штучного інтелекту на рівні C4 Component. На відміну від контейнерної діаграми, де служба RCA розглядається як один логічно відокремлений контейнер платформи, компонентний рівень показує її внутрішні відповідальності, точки інтеграції з іншими контейнерами та послідовність перетворення виявленої аномалії на перевірювану інженерну гіпотезу.

Служба аналізу першопричин не повинна дублювати функції збирання телеметрії, виявлення аномалій або формування карток доказів. Її архітектурна роль полягає у проміжному аналітичному шарі між службою виявлення аномалій і службою карток доказів. Вона отримує сигнал про аномалію, збирає контекст, зіставляє різнорідні докази, формує гіпотези щодо можливих причин, оцінює їх за допомогою правил, графових залежностей і моделей ШІ, а потім передає підтверджені результати для подальшого пояснення та операціоналізації.

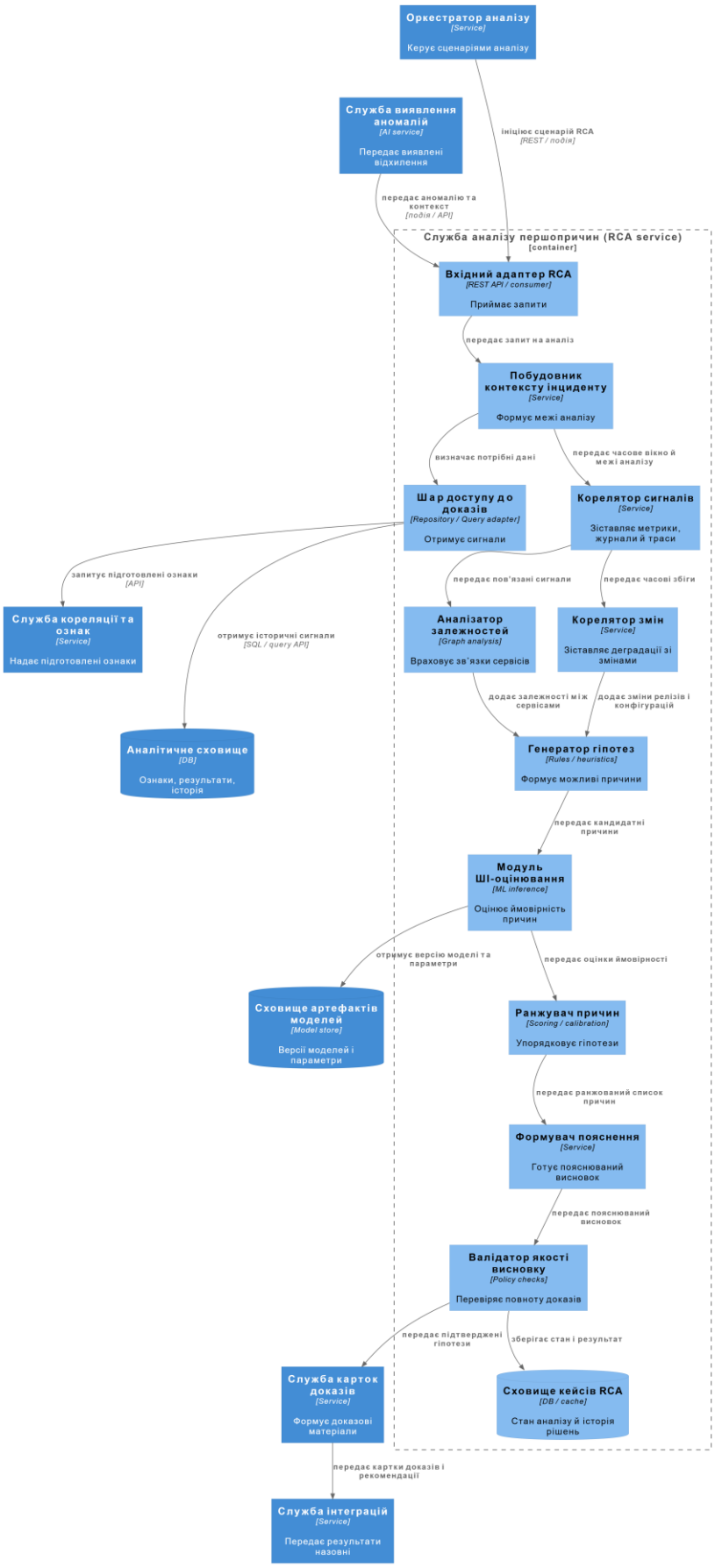

Рисунок 2.4 – C4-рівень компонентів (Component): внутрішня будова сервісу аналізу першопричин на основі ШІ

Вхідний адаптер RCA є точкою входу до контейнера. Через нього оркестратор аналізу або служба виявлення аномалій ініціює сценарій RCA. Такий компонент ізолює внутрішню логіку аналізу від конкретного способу запуску: синхронного REST-запиту, асинхронної події з шини повідомлень або системного завдання, створеного за результатами контрольного шлюзу продуктивності. Це важливо для супроводжуваності, оскільки спосіб запуску аналізу може змінюватися без перероблення власне алгоритмів аналізу першопричин.

Побудовник контексту інциденту визначає межі аналізу: часовий інтервал деградації, сервіс або групу сервісів, тип аномалії, критичні показники, середовище виконання, версію розгортання та пов’язані події. Без такого компонента RCA-сервіс ризикував би аналізувати надто широкий масив даних, що збільшує витрати оброблення і підвищує ймовірність хибних зв’язків. Контекст інциденту обмежує простір пошуку й задає основу для подальшої кореляції.

Шар доступу до доказів виконує інтеграційну функцію всередині RCA-сервісу. Він звертається до служби кореляції та ознак, аналітичного сховища, а за потреби – до інших джерел підготовлених телеметричних даних. Його доцільно реалізувати як набір адаптерів або репозиторіїв, щоб внутрішні компоненти RCA не залежали від конкретного формату запитів до сховищ. Такий підхід полегшує заміну джерел даних, зміну схеми аналітичного сховища або підключення додаткових сигналів без перебудови всієї служби.

Корелятор сигналів зіставляє метрики, журнали та траси в межах сформованого контексту. Його завдання полягає не лише в пошуку часових збігів, а й у виявленні структурно пов’язаних подій. Наприклад, зростання p95-затримки може бути зіставлене з повторними спробами запитів, перевищеннями часу очікування, спрацюванням запобіжника міжсервісної взаємодії, зростанням кількості помилок у журналах або появою довгих трас у конкретному сервісі. Саме цей компонент перетворює набір розрізнених сигналів на зв’язану доказову картину.

Аналізатор залежностей враховує топологію системи: зв’язки між сервісами, критичні шляхи виконання запитів, залежності від баз даних, черг повідомлень, зовнішніх API та інфраструктурних компонентів. У практичній реалізації він може використовувати граф сервісів, побудований на основі трасування або конфігурацій середовища виконання. Його наявність дозволяє відрізняти первинну причину від вторинного наслідку. Наприклад, деградація сервісу замовлень може бути наслідком проблеми в сервісі платежів, а не локальним дефектом самого сервісу замовлень.

Корелятор змін зіставляє появу деградації з релізами, змінами конфігурації, масштабуванням, зміною лімітів ресурсів, оновленнями залежностей або інфраструктурними подіями. Цей компонент є важливим, оскільки значна частина проблем продуктивності виникає не лише через поточне навантаження, а й через зміни в коді, конфігурації або середовищі виконання. Водночас часовий збіг не повинен трактуватися як автоматичне доведення причини. Компонент лише додає зміну до набору кандидатних пояснень, які мають бути перевірені іншими доказами.

Генератор гіпотез формує перелік можливих причин деградації на основі корельованих сигналів, топологічних зв’язків і даних про зміни. Це можуть бути гіпотези про регресію продуктивності після релізу, перевантаження залежного сервісу, недостатню кількість реплік, неефективний запит до бази даних, блокування черги, збільшення мережевої затримки або некоректне налаштування тайм-аутів і запобіжників. На цьому етапі важливо не відкидати альтернативні пояснення надто рано, оскільки в розподілених системах одна деградація часто має кілька взаємопов’язаних чинників.

Модуль ШІ-оцінювання використовує моделі машинного навчання або статистичні методи для оцінювання правдоподібності кандидатних причин. Він звертається до сховища артефактів моделей, щоб отримати актуальну версію моделі, її параметри, пороги та службові метадані. Архітектурно важливо, щоб цей компонент не був “чорною скринькою”, яка без пояснення повертає остаточний результат. Його вихід має містити оцінки ймовірності, ключові

ознаки, що вплинули на рішення, часові межі аналізу та посилання на використану версію моделі.

Ранжувач причин об’єднує результати правил, графового аналізу, кореляції змін і ШІ-оцінювання. Його завданням є впорядкування гіпотез за рівнем підтвердженості та практичної значущості. У зрілій реалізації ранжування має враховувати не лише ймовірність, а й силу доказів, повторюваність проблеми, вплив на SLO, кількість зачеплених сервісів, наявність релізу в часовому вікні деградації та вартість перевірки гіпотези. Це дозволяє подавати інженеру не випадковий перелік причин, а пріоритезований набір дій для перевірки.

Формувач пояснення перетворює технічний результат аналізу на зрозумілий інженерний висновок. Він структурує дані так, щоб їх можна було використати в картці доказів: короткий опис проблеми, найімовірніші причини, часовий інтервал, зачеплені сервіси, підтверджувальні метрики, журнальні події, траси, пов’язані зміни та рекомендовані перевірки. Цей компонент є важливим для довіри до платформи, оскільки без пояснюваності результати ШІ-аналізу складно використовувати у виробничому середовищі.

Валідатор якості висновку перевіряє, чи достатньо даних для передавання результату далі. Він може контролювати наявність обов’язкових доказів, мінімальний рівень упевненості, відповідність політикам пояснюваності, повноту часових меж і відсутність суперечностей між сигналами. Якщо доказів недостатньо, служба може повернути результат як непідтверджену гіпотезу або вимагати додаткового збирання даних. Це зменшує ризик передавання в систему інцидентів слабко обґрунтованих висновків.

Сховище кейсів RCA зберігає стан аналізу, історію розглянутих гіпотез, результати ранжування, версії використаних моделей і зв’язок між інцидентами та прийнятими рішеннями. Таке сховище потрібне для простежуваності й повторного використання досвіду. Якщо подібна деградація виникає повторно, платформа може порівняти новий випадок із попередніми кейсами й запропонувати більш обґрунтований набір гіпотез.

У цілому компонентна модель показує службу аналізу першопричин як керований аналітичний контейнер із чітким поділом відповідальностей. Вона відокремлює приймання запиту, побудову контексту, доступ до доказів, кореляцію сигналів, генерацію гіпотез, ШІ-оцінювання, ранжування, пояснення та валідацію. Такий поділ підвищує тестованість, дозволяє замінювати окремі алгоритми без перебудови всієї служби та створює основу для операціоналізації RCA у процесах супроводу програмної системи.

Здійснені архітектурні рішення зібрано в репозиторії архітектурних рішень (табл. 2.2). Кожному з них зіставляються підстава для вибору та компроміси, які стають наслідками вибору.

Таблиця 2.2 – Архітектурні рішення та обґрунтування запропонованої моделі

ADR	Рішення	Підстава	Наслідки та компроміси
ADR-01	Використати OpenTelemetry як базовий контракт інструментування	Потрібна переносимість між бекендами та узгодженість трас, метрик і журналів	Зменшує vendor lock-in, але вимагає дисципліни семантичних угод
ADR-02	Розділити сховища для метрик, трас і логів	Різні типи сигналів мають різні моделі запитів і період зберігання	Покращує спеціалізацію сховищ, але ускладнює кореляцію та вимагає trace_id/correlation_id
ADR-03	Розглядати ШІ/RCA як дорадчий прошарок	RCA-гіпотези потребують доказової перевірки	Знижує ризик автоматичних помилок, але потребує human експертна перевірка і картка доказів
ADR-04	Використати контрольні шлюзи продуктивності у CI/CD	Деградації варто виявляти до потрапляння в експлуатаційне середовище	Покращує ранню перевірку, але потребує стабільного тестового середовища та базовий рівень
ADR-05	Фіксувати каталог SLO і операційні інструкції як операційні артефакти	Метрики мають бути пов’язані з діями та відповідальністю	Зменшує хаос інцидентів, але потребує регулярної актуалізації документації

2.5 Операціоналізація процесів упровадження

Операціоналізація означає перетворення архітектурної ідеї на відтворюваний процес із відповідальними ролями, артефактами, критеріями приймання й інструментальними перевірками. Для цієї роботи ключовими

операційними артефактами є контракт телеметрії, каталог SLO, контрольні шлюзи продуктивності в CI/CD, картка доказів, ADR, реєстр ризиків і операційні інструкції. Вони мають бути елементами єдиного процесу: метрика потрапляє в шлюз; шлюз створює звіт; звіт формує картку доказів; рішення фіксується в ADR; ризик потрапляє до реєстру; дія інструктується в операційній інструкції. Операційні артефакти впровадження зібрано в табл. 2.3.

Таблиця 2.3 – Операційні артефакти впровадження та критерії готовності

Артефакт	Зміст	Власник	Критерій готовності
Контракт телеметрії	Обов'язкові атрибути, типи сигналів, вибіркове збирання, маскування чутливих даних, маршрутизація	Tech lead / SRE	Сервіс проходить automated telemetry quality check
каталог SLO	Сценарії, SLI, SLO, вікна, алерти, відповідальні	Product/Tech lead + SRE	Для критичних сценаріїв визначені SLO та джерела вимірювання
Контрольний шлюз продуктивності	Load profile, базовий рівень, thresholds, пройдено / не пройдено (пройдено / не пройдено (pass/fail)) policy	DevOps / QA automation	Pipeline повертає відтворюваний звіт і verdict
Картка доказів	Симптом, гіпотези, докази, альтернативи, рішення	Performance analyst / SRE	Кожна RCA-рекомендація має доказову картку
ADR	Контекст, рішення, альтернативи, наслідки	Architect / Tech lead	Ключові компромісs зафіксовані та переглядаються
Операційна інструкція	Дії при алерті, запити, панелі спостереження, ескалація	SRE / Operations	Алерт має пов'язаний операційна інструкція і власника

Контракт телеметрії визначає мінімальний набір сигналів, без яких аналіз продуктивності не є надійним. Для кожного сервісу доцільно зафіксувати service.name, service.version, deployment.environment, назву операції, маршрут, код стану, гістограму затримки, тривалість викликів залежностей, клас помилки, trace_id, correlation_id і подію зміни. Контракт також має визначати, які атрибути заборонено передавати через ризик конфіденційності. Це дозволяє уникнути ситуації, коли AI/RCA отримує багато даних, але вони несумісні або небезпечні для зберігання.

Каталог SLO визначає зв’язок між користувацькими сценаріями та інженерними показниками. Наприклад, для сценарію “користувач відкриває сторінку звіту” SLI може включати p95 latency backend API, частка помилок (error rate), кількість timeout до бази даних і частку успішних транзакцій. При цьому SLO не слід установлювати занадто жорстко без даних базовий рівень. У навчальному прототипі доцільно спершу зафіксувати припущення, провести навантажувальна перевірка, отримати базовий рівень, а потім уточнити SLO. Це демонструє зрілість: система не видає бажані цифри за доказові.

Контрольний шлюз продуктивності CI/CD має бути інтегрований із конвеєром постачання так, щоб перевіряти зміни до релізу. На практиці шлюз може складатися з етапів: запуск тестового середовища; прогрів (warm-up); виконання профілю навантаження; збирання метрик; порівняння з базовим рівнем; аналіз відхилення регресії; формування звіту; прийняття рішення «пройдено / не пройдено» (pass/fail). Важливо, щоб цей механізм не був надмірно крихким: критерії мають враховувати природний шум середовища, мінімальну кількість повторів і статистичну значущість різниці.

Картка доказів є центральним документом RCA. Вона має містити інцидент або регресію, час, зачеплений сценарій, порушений SLO, сигнали, що підтверджують симптом, гіпотези, докази за і проти, альтернативні пояснення, рішення, власника і подальші дії (follow-up). Така картка дисциплінує AI/RCA: модель не може просто “назвати причину”, вона повинна показати, які дані підтверджують її припущення і які перевірки залишаються невиконаними.

ADR потрібні для того, щоб архітектурні рішення не втрачали контекст. Наприклад, якщо команда вирішила зменшити sampling трас до 10 %, слід зафіксувати не лише зміну конфігурації, а й причину, очікувану економію, ризик втрати рідкісних аномалій, спосіб компенсації через tail-based sampling або підвищений sampling для критичних маршрутів. Без ADR подальші члени команди бачитимуть лише результат, але не зможуть зрозуміти, чому компроміс був прийнятий.

Операційні інструкції забезпечують зв’язок між алертом і дією. Якщо алерт повідомляє про burn rate або p95 regression, операційна інструкція має пояснити, які дашборди відкрити, які PromQL-запити перевірити, які trace filters застосувати, які логи зіставити, коли виконувати rollback, коли ескалювати проблему та які докази додати до картка доказів. Без операційна інструкція алерт може лише перервати інженера, але не скоротити середній час відновлення (MTTR).

Реєстр ризиків потрібен для підтримки реалістичності. Упровадження аналізу продуктивності на основі ШІ має власні ризики: накладні витрати телеметрії (telemetry overhead), зростання вартості зберігання, хибні RCA-рекомендації, залежність від якості журналів, витік чутливих даних, розростання панелей спостереження (dashboard sprawl), недовіра команд до AI-рекомендацій, деградація конвеєра постачання через нестабільні перевірки продуктивності. Кожен ризик має отримати власника, ймовірність, вплив, спосіб пом’якшення і сигнал раннього виявлення.

Операціоналізація має завершуватися визначенням готовності (definition of done) для впровадження. Платформу можна вважати мінімально придатною, якщо для критичних сервісів є контракт телеметрії, базовий каталог SLO, панель спостереження за RED/USE, коректне передавання trace-контексту, контрольний шлюз продуктивності в конвеєрі постачання, шаблон картки доказів, хоча б одна операційна інструкція для типового інциденту та реєстр ризиків. Без цього набір інструментів залишається фрагментарним і не гарантує керованого покращення продуктивності.

РОЗДІЛ 3 ОЦІНЮВАННЯ АРХІТЕКТУРИ, РИЗИКИ ТА РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ

3.1 ATAM-оцінка, sensitivity points і tradeoff-аналіз запропонованої архітектури

Оцінювання архітектури в цій роботі не має доводити, що запропонована модель “найкраща”. Така безапеляційність була б методологічно некоректною, оскільки придатність архітектури залежить від контексту, масштабу системи, вимог до latency, бюджету, зрілості команди, регуляторних обмежень і доступності історичних даних. Мета ATAM-оцінки полягає в тому, щоб виявити, які архітектурні рішення підтримують сценарії якості, де виникають компроміси, які ризики потребують перевірки, а які припущення можуть бути тимчасово прийнятими.

Перший sensitivity point пов’язаний із sampling traces. Якщо sampling занадто низький, RCA може не побачити рідкісні аномалії; якщо занадто високий – зростає overhead і вартість. Один із можливих компромісів полягає у використанні head-based sampling для звичайного трафіку і підвищеного або tail-based sampling для запитів, які порушують SLO чи містять помилки. Проте таке рішення потребує технічної підтримки в collector/backend і чіткої політики retention.

Другий sensitivity point – кардинальність міток (label cardinality) у метриках. Додавання user_id, request_id або довільних path parameters як мітки може зробити метрики майже непридатними для зберігання й запитування. З іншого боку, надмірне спрощення мітки може зруйнувати можливість локалізації проблеми. Тому контракт телеметрії має визначати допустимі мітки, правила нормалізації routes і заборонені атрибути.

Третя точка чутливості (sensitivity point) – межа між автоматичною RCA-рекомендацією та дією. Якщо відновлювальна дія запускається автоматично, можна скоротити середній час відновлення (MTTR), але зростає ризик помилкової дії. Якщо все потребує ручної експертної перевірки, система

безпечніша, але повільніша. У межах навчального та пілотного впровадження доцільно залишити AI/RCA у режимі дорадчого помічника, а автоматичні дії дозволяти лише для низькоризикових сценаріїв, наприклад створення issue, оновлення картки доказів або пропозиції операційної інструкції.

Четверта точка чутливості – стабільність контрольного шлюзу продуктивності. Якщо тестове середовище нестабільне, конвеєр постачання може блокувати релізи через шум. Якщо шлюз занадто м'який, деградації потраплятимуть в експлуатаційне середовище. Тому критерії приймання мають включати мінімальну кількість повторів, поріг статистично значущої різниці та правила порівняння з базовим рівнем. Для навчального прототипу достатньо простішого критерію, але в промисловому контексті така перевірка має бути статистично обґрунтованою.

П'ятий sensitivity point – керування даними та процесами даних. Чим більше телеметрії потрапляє в аналітичний шар, тим вищою може бути діагностична цінність, але тим більшим стає ризик приватності, compliance і несанкціонованого доступу. Тому redaction, access control, retention і audit logging мають бути не "після впровадження", а частиною архітектури з самого початку.

ATAM-аналіз також виявляє non-risks, якщо для них є достатні підстави. Наприклад, використання C4 як засобу опису архітектури не створює значного технічного ризику, якщо воно не підміняє собою інші артефакти. Використання Prometheus для часових рядів метрик також може бути non-risk у малому прототипі, якщо cardinality контрольована і retention невеликий. Проте ці non-risks можуть стати ризиками при масштабуванні.

Критично важливо відрізняти архітектурний ризик від реалізаційного дефекту. Відсутній trace_id у журналах – це дефект інструментування, але рішення не мати єдиного контракту телеметрії – архітектурний ризик. Повільний SQL-запит може бути дефектом конкретної реалізації, але відсутність контрольного шлюзу продуктивності для критичних транзакцій – ризик процесу. Саме тому оцінювання архітектури має поєднувати і структуру платформи, і операційні практики.

Підсумковий висновок ATAM-оцінки полягає в тому, що запропонована архітектура є доцільною як референсна модель для навчального або пілотного впровадження за умови поетапної реалізації та обмеженого трактування ролі ШІ. Її сильна сторона – зв'язок між сценаріями якості, контрактом телеметрії, каталогом SLO, картками доказів, архітектурними записами й реєстром ризиків. Її слабка сторона – залежність від дисципліни інструментування та експертної перевірки. У разі нехтування цими умовами платформа може створювати враження автоматизації без достатньої доказової основи. ATAM-оцінка ключових архітектурних рішень наведено в таблиці 3.1.

Таблиця 3.1 – ATAM-оцінка ключових архітектурних рішень

Рішення / тактика	Підтримувані атрибути	Sensitivity point	Tradeoff	Ризик / дія
OpenTelemetry-based instrumentation	Observability, переносимість, супроводжуваність	Повнота semantic conventions	Більше атрибутів підвищує діагностику, але збільшує overhead і ризик PII	Telemetry quality gate, маскування чутливих даних policy
Prometheus metrics + SLO сповіщення	Performance, надійність, експлуатаційна придатність	Label кардинальність, вікна агрегації	Деталізація проти вартості й стабільності запитів	Обмеження мітки, правила період зберігання
Distributed tracing	Diagnosability, надійність	Sampling ratio, context propagation	Повнота трас проти overhead	Tail-based вибіркове збирання для критичних сценаріїв
ШІ/RCA advisory layer	середній час виявлення (середній час виявлення (MTTD))/середній час відновлення (середній час відновлення (MTTR)), decision support	Якість виокремлення ознак і evidence	Швидкість гіпотез проти ризику хибного пояснення	Human експертна перевірка, картка доказів, петля зворотного зв'язку
CI/CD контрольні шлюзи продуктивності	Release quality, regression control	Стабільність тестового середовища	Жорсткі gates зменшують ризик релізу, але можуть блокувати через шум	Baseline, repeatability, thresholds із допусками
Операційні інструкції and ADR	Operability, супроводжуваність	Актуальність документації	Більше контролю проти додаткового процесного навантаження	Власники артефактів, регулярний експертна перевірка

3.2. Ризики, обмеження, керування AI/RCA та багатокритеріальна SWOT-оцінка

Найбільший ризик підходу на основі ШІ полягає не в тому, що модель помилиться. Помилки є очікуваними для будь-якої статистичної системи. Більш небезпечним є те, що помилкова рекомендація може виглядати переконливою, якщо вона оформлена як структуроване пояснення з технічними термінами. Тому архітектура повинна вимагати картку доказів і експертну перевірку для рішень, які впливають на експлуатаційне середовище, релізи або масштабування інфраструктури.

Огляди RCA в мікросервісах показують, що сучасні методи використовують метрики, логи, траси та мультимодальні підходи, але стикаються з проблемами складних залежностей, propagating faults, нестачі стандартних dataset і труднощів порівняння методів [23]. Це важливе обмеження для кваліфікаційної роботи: не можна заявляти, що запропонована модель “автоматично знаходить причину”. Коректніше стверджувати, що вона створює архітектурні та операційні умови для більш доказового й швидкого висування RCA-гіпотез.

Огляди AIOps для хмарних платформ також підкреслюють, що AIOps охоплює виявлення інцидентів, прогнозування відмов, аналіз першопричин і автоматизовані дії, але ефективність цих задач залежить від якості даних, формалізації задачі та інтеграції з операційними процесами [24]. Тому платформа в роботі не повинна подаватися як “AI замість DevOps”. Її реалістична роль – підтримати людей у пошуку, ранжуванні та перевірці гіпотез, а також зменшити час на рутинну кореляцію сигналів.

Дослідження AIOps для керування відмовами демонструють, що виявлення відмов і RCA належать до найбільш опрацьованих напрямів, але автоматизовані відновлювальні дії залишаються складними через ризики побічних ефектів, неповну інформацію та контекстні залежності [25]. Звідси випливає практична рекомендація: у межах прототипу доцільно автоматизувати

виявлення, збирання доказів і формування звітів, але потенційно небезпечні дії залишати за інженером.

Окрема група ризиків пов’язана з журналами. Емпіричні дослідження показують, що якість розбору журналів (log parsing) суттєво впливає на точність виявлення аномалій (anomaly detection) [26]. Drain як онлайн-парсер журналів продемонстрував високу точність і продуктивність на великих наборах журналів, однак будь-який парсер залежить від стабільності форматів і змісту логів [27]. Тому в роботі слід підкреслити: журнали мають бути структурованими, контрольованими й пов’язаними з контекстом телеметрії.

NIST AI RMF наголошує на управлінні ризиками AI-систем через керування (governance), відображення контексту (mapping), вимірювання (measuring) і управління діями (managing) [28]. Для платформи аналізу продуктивності це означає, що AI-компонент має мати визначені межі застосування, набір оцінюваних ризиків, метрики якості рекомендацій, правила людського втручання і журнали прийнятих рішень. У навчальному прототипі це може бути спрощено, але не може бути повністю проігноровано.

Безпековий аспект мікросервісної та телеметричної архітектури також потребує обережності. Практики захисту мікросервісів і сервісна сітка (service mesh) підкреслюють важливість ідентичності сервісів, керування доступом, шифрування каналів і сегментації [29]. Для спостережуваність це означає, що доступ до логів, трас і дашбордів має бути рольовим; сервісні токени не повинні потрапляти в логи; дані мають проходити redaction; а дії аналітичної системи мають залишати audit trail. Додатково доцільно враховувати рекомендації OWASP щодо захисту web/API-рішень [30].

SWOT-аналіз у роботі використано не як формальну таблицю, а як спосіб зіставити внутрішні властивості запропонованої архітектури із зовнішнім контекстом упровадження. Сильні сторони не означають гарантії успіху; слабкі сторони не означають непридатності. SWOT корисний тоді, коли кожен пункт породжує практичну дію: що посилити, що обмежити, що перевірити експериментально, що відкласти на наступний етап.

Головна сильна сторона запропонованої моделі – зв’язність артефактів. Вона пов’язує C4-модель, QAW-сценарії, контракт телеметрії, каталог SLO, контрольні шлюзи CI/CD, картки доказів, ATAM і реєстр ризиків. Головна слабкість – залежність від дисципліни впровадження: якщо команди не підтримують семантику телеметрії, не актуалізують операційні інструкції і не переглядають ADR, архітектура швидко деградує до набору інструментів. Головна можливість – використання AI-пояснень без доказової перевірки. Реєстр ризиків для впровадження платформи наведено в таблиці 3.2.

Таблиця 3.2 – Реєстр ризиків для впровадження платформи

ID	Ризик	Ймовірність / вплив	Ознаки прояву	Mitigation
R-01	Недостатня якість телеметрії	Висока / висока	Немає traca_id, різні назви сервісів, нестабільні мітки	Контракт телеметрії, automated quality gate, code експертна перевірка instrumentation
R-02	Хибні RCA-рекомендації	Середня / висока	AI називає причину без незалежних доказів	Картка доказів, human експертна перевірка, альтернативні гіпотези
R-03	Надмірна вартість зберігання	Середня / середня	Зростання період зберігання cost, повільні запити	Sampling, downвибіркове збирання, період зберігання tiers, кардинальність control
R-04	Витік чутливих даних у логах/трасах	Середня / висока	PII або токени у log payload	Redaction, allowlist атрибути, access control, audit
R-05	Нестабільні контрольні шлюзи продуктивності	Середня / середня	Pipeline дає різні verdict для однакових змін	Warm-up, repeated runs, базовий рівень управління, test environment isolation
R-06	Dashboard sprawl	Висока / середня	Багато панелей без рішень і власників	Dashboard taxonomy, SLO-first панель спостереженняs, регулярний експертна перевірка

SWOT-оцінка запропонованої архітектури наведено в таблиці 3.3.

Таблиця 3.3 – SWOT-оцінка запропонованої архітектури

Вимір	Зміст	Практична дія
Strengths	Зв’язок між сценаріями якості, C4-моделлю, телеметрією, SLO, gates і RCA	Використати як шаблон впровадження та навчальний каркас
Weaknesses	Залежність від дисципліни контракт телеметрії, операційні інструкції, ADR і актуальності базовий рівень	Призначити власників артефактів і періодичність експертна перевірка
Opportunities	Можливість еволюційного переходу від моніторингу до доказового AIOps	Почати з 1-2 критичних сервісів і розширювати coverage
Threats	Некритичне прийняття AI-пояснень, надмірні витрати на телеметрію, privacy-ризики	Залишити AI в advisory mode, впровадити маскування чутливих даних і cost controls

3.3 Дорожня карта впровадження, контрольні артефакти та підсумкові рекомендації

Дорожню карту впровадження доцільно будувати поступово, оскільки спроба одразу створити повну AIOps-платформу зазвичай перевантажує команду інструментами й документацією. Перший етап має бути присвячений мінімальній керованій телеметрії: вибір одного критичного сервісу, опис контракт телеметрії, налаштування OpenTelemetry, базові метрики RED/USE, trace propagation, структуровані логи та один SLO-oriented панель спостереження. Завдання цього етапу – не AI, а якісні дані.

Другий етап – перехід від телеметрії до SLO та контрольних шлюзів продуктивності. Потрібно визначити 2-3 критичні користувацькі сценарії, пов’язати їх із SLI/SLO, створити профілі навантаження, отримати базовий рівень і додати шлюз до CI/CD. Рішення про реліз має спиратися не на інтуїтивне “графіки виглядають нормально”, а на разовий звіт з чіткими критеріями. Якщо базовий рівень нестабільний, шлюз має працювати в advisory mode до стабілізації.

Третій етап – формування доказового аналізу першопричин. На цьому етапі доцільно підключати AI/RCA-компоненти для ранжування гіпотез, але з обов’язковою карткою доказів і експертною перевіркою. Критерієм успіху є не

“модель відповіла”, а скорочення часу збирання доказів, зменшення кількості ручних переходів між інструментами, краща повторюваність діагностики та накопичення бази знань про типові деградації.

Четвертий етап – керування даними й масштабування. Якщо пілот показує практичну користь, платформу можна поширювати на інші сервіси. Але масштабування має відбуватися через стандартизацію шаблонів: шаблон контракту телеметрії, шаблон панелі спостереження, шаблон операційної інструкції, шаблон ADR, реєстр ризиків, шаблон конвеєра постачання. Без шаблонів кожна команда створить власну версію спостережуваності, і централізований AI/RCA втратить сумісність сигналів.

П’ятий етап – оптимізація витрат і зрілості процесів. Після накопичення історії слід переглянути retention, sampling, cardinality, кількість алертів, якість операційні інструкції, частку confirmed RCA та вплив на середній час відновлення (MTTR). Якщо певні сигнали не використовуються для рішень, їх варто зменшити або перемістити в дешевший retention tier. Якщо певні алерти не приводять до дій, їх слід переформулювати або видалити.

Рекомендації щодо впровадження слід формулювати обережно. Для невеликої команди недоцільно починати з комплексної AI/RCA-системи; достатньо створити якісне інструментування, SLO, панелі спостереження й контрольні шлюзи продуктивності. Для середньої команди варто додати стандартизовані картки доказів, ADR і операційні інструкції. Для більш зрілої організації можна розглядати AI/RCA як допоміжний аналітичний шар, але лише за умови наявності історичних даних, ownership і керування даними.

Для кваліфікаційної роботи важливо показати, як саме результати можна перевірити. Мінімальний демонстраційний сценарій може включати два профілі навантаження: базовий рівень і degraded. У degraded-профілі штучно збільшується latency залежності або зменшується resource limit. Платформа має показати порушення SLO, відобразити зміну на дашборді, згенерувати картка доказів, показати відповідний trace/log контекст і повернути рекомендацію, яка не є остаточним verdict, а потребує підтвердження.

Оцінювання ефекту можна здійснювати через порівняння ручного та інструменталізованого процесу. Наприклад, без платформи інженер вручну переходить між панелями спостереження, журналами, інтерфейсом трасування та історією CI. З платформою частина цих переходів автоматизована, а картка доказів збирає ключові посилання й припущення. Показниками можуть бути середній час виявлення (MTTD), час до отримання доказів (time-to-evidence), кількість ручних кроків, частка підтверджених гіпотез, кількість пропущених альтернатив і середня кількість повторних спрацювань.

Загальна рекомендація полягає в тому, щоб будувати аналіз продуктивності на основі ШІ як еволюційний інженерний контур, а не як одноразове впровадження інструменту. Система має починатися з вимог і сценаріїв якості, продовжуватися архітектурним моделюванням, переходити до операційних артефактів і завершуватися циклічним керуванням ефекту. Лише за такої умови AI-компонент має шанс бути корисним: він працює не у вакуумі, а в середовищі якісних даних, визначених SLO, прозорих рішень і контрольованих ризиків. Дорожня карта впровадження платформи наведено в таблиці 3.4.

Таблиця 3.4 – Дорожня карта впровадження платформи

Етап	Зміст робіт	Ключові артефакти	Definition of Done
1. Telemetry foundation	Інструментування 1 критичного сервісу, metrics/траси/logs, correlation ID	Контракт телеметрії, базовий панель спостереження	Сервіс передає обов'язкові атрибути; працює траса propagation
2. SLO and gates	Опис SLI/SLO, профіль навантаженняs, базовий рівень, конвеєр gate	каталог SLO, контрольний шлюз продуктивності report	Pipeline формує пройдено / не пройдено (пройдено / не пройдено (pass/fail)) verdict і зберігає звіт
3. Доказовий аналіз першопричин (доказовий аналіз першопричин (evidence-based RCA))	Підключення ШІ/RCA advisory layer, картки доказів, експертна перевірка	Картка доказів, RCA report, feedback log	Кожна гіпотеза має докази та статус експертна перевірка
4. Керування даними та процесами and scaling	Шаблони для сервісів, ADR, операційні інструкції, access policy	ADR, операційна інструкція library, реєстр ризиків	2-3 сервіси підключені за єдиним шаблоном
5. Optimization	Перегляд період зберігання, вибіркове збирання, alert noise, cost	Cost report, maturity assessment	Зменшено шум і витрати без втрати критичної діагностики

Шаблон картки доказів для RCA наведено в таблиці 3.5.

Таблиця 3.5 – Шаблон картки доказів для RCA

Поле	Зміст заповнення	Приклад для деградації API
Incident / regression	Що саме виявлено і коли	p95-затримка search API зросла з 320 до 780 мс після релізу 1.8.4
Affected scenario	Користувацький або технічний сценарій	Пошук даних у вебзастосунку під навантаженням 300 RPS
SLO impact	Який SLO порушено	p95 ≤ 500 мс порушено протягом 12 хв
Evidence for hypothesis	Метрики, траси, логи, події змін	Trace показує зростання DB span; розгортання event за 8 хв до деградації; CPU без saturation
Alternative explanations	Що ще може пояснювати симптом	Зовнішній трафік, зміна індексу, network затримка
Decision	Що зроблено або рекомендовано	Rollback або перевірка SQL-плану; створити ADR щодо індексації
Residual risk	Що залишається невизначеним	Не перевірено вплив кешу та concurrent writes

Проміжні висновки до розділу 3 полягають у тому, що запропонована архітектура має практичну цінність за умови поетапного впровадження, реалістичного трактування AI/RCA і жорсткої прив’язки до доказових артефактів. ATAM показує, що основні компроміси стосуються вибіркового трасування, кардинальності міток, меж автоматичної RCA, стабільності контрольних шлюзів продуктивності й керування телеметричними даними. SWOT свідчить, що сильна сторона моделі – зв’язність артефактів, а слабка – залежність від процесної дисципліни.

ВИСНОВКИ

У роботі відновлено критичний інженерний фокус теми: аналіз продуктивності програмного забезпечення розглянуто не як опис популярних засобів моніторингу, а як архітектурно зумовлений процес вимірювання, пояснення та покращення поведінки системи. Показано, що latency, throughput, частка помилок (error rate) і resource utilization мають практичний сенс лише в контексті сценаріїв якості, SLO, середовища виконання, профілю навантаження й архітектурних залежностей.

У першому розділі систематизовано методологічну основу роботи. Продуктивність пов’язано зі спостережуваністю, надійністю, масштабованістю, безпекою телеметрії та вартістю. Обґрунтовано, що метрики мають використовуватися як доказові інструменти, а не як декоративні графіки. Запропоновано каталог метрик, який пов’язує SLI/SLO, джерела даних і рішення, що приймаються за результатами вимірювання.

У другому розділі побудовано архітектурну модель платформи аналізу продуктивності на основі ШІ. C4-рівень контексту (Context), контейнерів (Container) і компонентів (Component) доповнено UML-діаграмами прецедентів, послідовності та діяльності, що дозволяє поєднати структурний і поведінковий опис системи. Запропоновано QAW-сценарії якості, контракт телеметрії, контрольні шлюзи продуктивності, картку доказів, ADR і реєстр ризиків як операційні артефакти впровадження.

У третьому розділі виконано оцінювання архітектури через ATAM, SWOT і реєстр ризиків. Визначено точки чутливості, зокрема вибіркове трасування, кардинальність міток, межі автоматичної RCA, стабільність контрольних шлюзів продуктивності та керування телеметричними даними. Показано, що основні переваги моделі пов’язані не з наявністю окремого AI-компонента, а зі зв’язком між метриками, сценаріями якості, доказами, рішеннями та відповідальністю інженера.

Практична цінність роботи полягає в тому, що запропонований підхід може бути використаний як шаблон поетапного впровадження: від

інструментування одного критичного сервісу до SLO, контрольних шлюзів продуктивності, доказового аналізу першопричин і керування даними та процесами. У роботі не стверджується, що AI автоматично розв’язує проблему аналізу першопричин; навпаки, обґрунтовано більш реалістичну роль AI як допоміжного механізму пошуку гіпотез і прискорення збирання доказів.

Подальший розвиток результатів доцільно спрямувати на створення відтворюваного демонстраційного стенда з двома профілями навантаження, контрольованими деградаціями, автоматичним формуванням картки доказів і порівнянням ручного та інструменталізованого RCA-процесу. Це дозволить перейти від архітектурно-аналітичного обґрунтування до експериментальної оцінки ефективності запропонованої платформи за сценаріями, визначеними в роботі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Методичні рекомендації до виконання кваліфікаційних робіт за спеціальністю 121/F2 «Інженерія програмного забезпечення». Черкаси : ЧДБК, 2026. 30 с.

2. ISO/IEC/IEEE 42010:2022. Software, systems and enterprise – Architecture description. Geneva : International Organization for Standardization, 2022.

3. ISO/IEC 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Geneva : ISO, 2011.

4. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley, 2021. 464 p.

5. Software Engineering Institute. The SEI Quality Attribute Workshop. Carnegie Mellon University, 2018. URL: https://www.sei.cmu.edu/library/the-sei-quality-attribute-workshop/

6. Kazman R., Klein M., Clements P. ATAM: Method for Architecture Evaluation. Pittsburgh : Software Engineering Institute, Carnegie Mellon University, 2000. URL: https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/

7. Brown S. The C4 model for visualising software architecture. URL: https://c4model.com/

8. OpenTelemetry Project. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/

9. OpenTelemetry Project. Semantic Conventions. URL: https://opentelemetry.io/docs/concepts/semantic-conventions/

10. Prometheus Authors. Prometheus: Monitoring system and time series database. URL: https://prometheus.io/

11. Grafana Labs. Grafana documentation. URL: https://grafana.com/docs/

12. Jaeger Project. Jaeger documentation. URL: https://www.jaegertracing.io/docs/latest/

13. Elastic. Elastic Observability documentation. URL: https://www.elastic.co/guide/en/observability/current/index.html

14. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol : O'Reilly Media, 2016. URL: https://sre.google/sre-book/table-of-contents/

15. Beyer B., Murphy N. R., Rensin D. K., Kawahara K., Thorne S. The Site Reliability Workbook. Sebastopol : O'Reilly Media, 2018. URL: https://sre.google/workbook/table-of-contents/

16. Sigelman B. H., Barroso L. A., Burrows M., Stephenson P., Plakal M., Beaver D., Jaspan S., Shanbhag C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report, 2010.

17. W3C. Trace Context. W3C Recommendation. 2021. URL: https://www.w3.org/TR/trace-context/

18. Gregg B. Systems Performance: Enterprise and the Cloud. 2nd ed. Boston : Addison-Wesley, 2020. 928 p.

19. Gregg B. The USE Method. URL: https://www.brendangregg.com/usemethod.html

20. Apache Software Foundation. Apache JMeter User Manual. URL: https://jmeter.apache.org/usermanual/

21. Locust. Locust Documentation. URL: https://docs.locust.io/

22. Gatling Corp. Gatling Documentation. URL: https://docs.gatling.io/

23. Wang T., Qi G. A Comprehensive Survey on Root Cause Analysis in (Micro) Services: Methodologies, Challenges, and Trends. arXiv, 2024. URL: https://arxiv.org/abs/2408.00803

24. Cheng Q., Sahoo D., Saha A., Yang W., Liu C., Woo G., Singh M., Savarese S., Hoi S. C. H. AI for IT Operations (AIOps) on Cloud Platforms: Reviews, Opportunities and Challenges. arXiv, 2023. URL: https://arxiv.org/abs/2304.04661

25. Notaro P., Cardoso J., Gerndt M. A Survey of AIOps Methods for Failure Management. ACM Transactions on Intelligent Systems and Technology. 2021.

26. Khan Z. A., et al. Impact of log parsing on deep learning-based anomaly detection. Empirical Software Engineering. 2024.

27. He P., Zhu J., Zheng Z., Lyu M. R. Drain: An Online Log Parsing Approach with Fixed Depth Tree. 2017 IEEE International Conference on Web Services. 2017.

28. National Institute of Standards and Technology. Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 100-1. Gaithersburg : NIST, 2023. URL: https://www.nist.gov/itl/ai-risk-management-framework

29. National Institute of Standards and Technology. Building Secure Microservices-based Applications Using Service-Mesh Architecture. NIST Special Publication 800-204A. Gaithersburg : NIST, 2020.

30. OWASP Foundation. OWASP Top 10: The Ten Most Critical Web Application Security Risks. 2021. URL: https://owasp.org/Top10/

31. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 p.

32. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O'Reilly Media, 2017. 616 p.

33. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley, 2010. 512 p.

34. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps. Portland : IT Revolution Press, 2018. 288 p.

35. Richardson C. Microservices Patterns: With examples in Java. Shelter Island : Manning Publications, 2018. 520 p.

ДОДАТКИ

Додаток А – Мінімальний перелік артефактів для демонстрації практичної частини

Таблиця А.1 – Мінімальний комплект практичних артефактів

Артефакт	Форма подання	Призначення
Контракт телеметрії	Таблиця або YAML-фрагмент	Фіксує обов’язкові сигнали, атрибути й обмеження
каталог SLO	Таблиця SLI/SLO	Пов’язує користувацькі сценарії з метриками
Load profile	JMeter/Locust/Gatling сценарій	Відтворює контрольоване навантаження
Dashboard	Grafana/Kibana mock або експорт	Показує SLO-first спостережуваність
Картка доказів	Шаблон і приклад заповнення	Підтримує доказовий RCA
ADR	Текстовий запис рішення	Фіксує rationale і компромісs
Операційна інструкція	Інструкція дій при алерті	Операціоналізує реакцію на інцидент