Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

Магістерська робота

освітній ступінь – магістр

на тему: «**ФРЕЙМВОРК ТА МЕТОДОЛОГІЯ ДЛЯ ЕФЕКТИВНОГО ТЕСТУВАННЯ І ТРАСУВАННЯ У МІКРОСЕРВІСНИХ СИСТЕМАХ**»

Виконав: студент 2-го року навчання,
освітньо-наукової програми «Інженерія програмного забезпечення», 121
Мазурок Андрій Сергійович
Керівник Глибовець А.М.,
доктор технічних наук, професор

Рецензент _____
(прізвище та ініціали)

Магістерська робота захищена
з оцінкою _____
Секретар ЕК _____
«____» _____ 2026 р.

Київ – 2026

Зміст

АНОТАЦІЯ....4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ....5
ВСТУП....6
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ....9
1.1 Проблематика End-to-End тестування в мікросервісних системах....9
1.2 Концепція спостережуваності.... 11
1.3 Еволюція підходів до логування та трасування розподілених запитів....12
1.4 Огляд та порівняльний аналіз існуючих інструментів....15
1.4.1 Бібліотеки логування в екосистемі Java.... 15
1.4.2 Системи централізованого збору та аналізу логів....17
1.4.3 Аналіз існуючих фреймворків тестування....19
1.5 Висновки до розділу 1....21
РОЗДІЛ 2. РОЗРОБКА МЕТОДОЛОГІЇ ТА ПРОГРАМНОГО ФРЕЙМВОРКУ.... 23
2.1 Архітектурний підхід та методологія Trace-Based Testing.... 23
2.2 Розробка бібліотеки структурованого логування.... 25
2.2.1 Ядро системи: Бібліотека tlog....25
2.2.2 Модуль автоконфігурації – spring-boot-starter-tlog.... 27
2.2.3 Наскрізне передавання та управління життєвим циклом контексту.... 28
2.2.4 Вплив структурованого логування на продуктивність та управління конкурентним доступом....31
2.3 Проєктування модуля автоматизованої валідації....32
2.3.1 Архітектура десеріалізації та парсингу JSONL-потоку....34
2.3.2 Оптимізація продуктивності та безпека управління пам'яттю.... 35
2.4 Висновки до розділу 2....36
РОЗДІЛ 3. АПРОБАЦІЯ ФРЕЙМВОРКУ ТА ОЦІНКА ЙОГО ЕФЕКТИВНОСТІ.38
3.1 Оркестрація інфраструктури тестового середовища....38

3.2 Реалізація наскрізного тестового сценарію.....40
3.3 Аналіз результатів та валідація методології.....41
3.3.1 Поведінка фреймворку при виявленні дефектів.....42
3.3.2 Порівняльний аналіз складності автоматизації тестових сценаріїв..... 42
3.4 Інтеграція інструментарію в конвеєр Continuous Integration.....45
3.5 Висновки до розділу 3.....45
ВИСНОВКИ..... 47
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 50

АНОТАЦІЯ

Магістерська кваліфікаційна робота присвячена розробці фреймворку та методології для ефективного наскрізного тестування мікросервісних систем на основі даних розподіленого трасування. Було реалізовано незалежну бібліотеку логування для генерації структурованих аудит-логів: забезпечено автоматичне передавання єдиного ідентифікатора транзакції за міжнародним стандартом W3C Trace Context [1] через синхронні та асинхронні канали зв'язку. Усі етапи виконання бізнес-логіки фіксуються в єдиному контексті із прив'язкою до семантичних ідентифікаторів операцій.

Під час тестування незалежний клієнт генерує унікальний ідентифікатор та ініціює запит до системи. Розроблений модуль валідації працює за принципом чорного ящика: він асинхронно сканує спільний файл логів і верифікує повний життєвий цикл розподіленого запиту. У системі реалізовано механізм відкладеної перевірки з використанням декларативного програмного інтерфейсу, що дозволяє гнучко визначати очікувані системні події. У разі виникнення помилок чи зависання окремих компонентів система генерує деталізований звіт про перерваний ланцюжок викликів.

Програмне рішення було реалізовано мовою Java для екосистеми Spring Boot. Інтеграція інфраструктури охоплює брокер повідомлень Apache Kafka, протокол віддаленого виклику процедур gRPC та класичні REST API.

Ключові слова: наскрізне тестування, мікросервісна архітектура, розподілене трасування, аудит-логи, Java, Spring Boot, Kafka, gRPC.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

E2E – End to End

JSONL – JSON Lines

UI – User Interface

MDC – Mapped Diagnostic Context

ACID – Atomicity, Consistency, Isolation, Durability

CAP – Consistency, Availability, Partition tolerance

DTO – Data Transfer Object

CI/CD – Continuous Integration/Continuous Deployment

ВСТУП

Актуальність теми

Зі зростанням навантаження та кількості користувачів усе більше систем будуються на мікросервісній архітектурі. Навіть уже наявні монолітні системи часто розбивають на мікросервіси. Мікросервісна архітектура забезпечує високу масштабованість, незалежність розгортання та гнучкість розробки. Проте розподілені системи породжують нові виклики, насамперед у сфері забезпечення якості, моніторингу та інтеграційного тестування.

Мікросервісна архітектура, з одного боку, полегшує тестування, адже дозволяє перевіряти кожен модуль окремо, а з іншого – створює ризик збоїв через проблеми з інтеграцією різних компонентів. У мікросервісному середовищі виконання одного запиту користувача часто ініціює складний ланцюжок синхронних (через REST або gRPC) та асинхронних (через брокери повідомлень, такі як Apache Kafka) викликів між багатьма незалежними компонентами. Традиційні підходи до E2E-тестування, що спираються на синхронні очікування або жорстко прив'язані до конкретних API окремих сервісів, виявляються неефективними. Вони є нестабільними, важко підтримуються та ламаються за найменшого архітектурного рефакторингу, оскільки не здатні надійно відстежувати асинхронні операції.

Для розв’язання цієї проблеми у розподілених системах індустрія активно впроваджує інструменти наскрізного трасування, спираючись на міжнародні стандарти, зокрема W3C Trace Context. Однак на сьогодні ці інструменти здебільшого використовуються лише для пасивного моніторингу та пошуку помилок у робочому середовищі. Використання даних трасування як фундаменту для автоматизованого тестування є відносно новим, але вкрай перспективним концептом, який ще не має широкого інструментального покриття для зручної інтеграції в екосистеми на базі Java.

З огляду на це, розробка спеціалізованого фреймворку та методології тестування, що базується на аналізі наскрізних аудит-логів та ідентифікаторів

транзакцій, є надзвичайно актуальним завданням. Створення інструментарію, який дозволяє відв'язати автотести від фізичної топології мережі та перевіряти систему за фактом виконання семантичних бізнес-операцій (принцип «чорного ящика»), дозволить суттєво підвищити надійність E2E-тестування, зменшити витрати часу на підтримку тестової бази та забезпечити високу якість складних розподілених програмних продуктів.

Мета та завдання магістерської роботи

Мета: Розробити фреймворк та методологію для ефективного автоматизованого E2E-тестування мікросервісних систем на основі даних розподіленого трасування та аналізу структурованих аудит-логів.

Завдання:

1. Проаналізувати сучасні підходи до інтеграційного та E2E-тестування розподілених систем, а також існуючі стандарти наскрізного трасування.
2. Розробити методологію фіксації бізнес-операцій та створити бібліотеку логування для генерації структурованих аудит-логів у середовищі Java.
3. Забезпечити автоматичне передавання єдиного ідентифікатора транзакції через синхронні (REST, gRPC) та асинхронні (Apache Kafka) канали зв'язку.
4. Розробити тестовий модуль валідації життєвого циклу розподіленого запиту на основі механізму відкладеної перевірки.
5. Інтегрувати розроблене рішення в мікросервісне середовище та підтвердити його стійкість до архітектурного рефакторингу під час тестування.

Об’єкт дослідження

Процес забезпечення якості та автоматизованого E2E-тестування в розподілених мікросервісних системах.

Предмет дослідження

Методологія та інструментальні засоби валідації бізнес-операцій на основі даних розподіленого трасування і структурованих аудит-логів під час синхронної та асинхронної взаємодії сервісів.

Методи дослідження

1. Методи наскрізного трасування та логування: використання міжнародного стандарту W3C Trace Context для кореляції розподілених запитів, структуроване логування семантичних бізнес-подій у форматі JSONL [2].
2. Методи автоматизованого тестування програмного забезпечення: E2E-тестування, перевірка систем за принципом «чорного ящика», Trace-Based Testing, метод відкладеної перевірки для валідації асинхронних операцій.
3. Методи системного аналізу та інтеграції: моделювання та аналіз розподіленої міжсервісної взаємодії через синхронні (REST, gRPC) та асинхронні (Apache Kafka) протоколи передачі даних.
4. Методи проєктування програмної архітектури: декларативний підхід до конфігурації тестових сценаріїв, застосування структурних патернів проєктування для розробки модуля валідації.

Наукова новизна роботи полягає в удосконаленні методології E2E-тестування розподілених систем шляхом інтеграції механізмів наскрізного трасування у процес автоматизованої валідації.

Практичне значення результатів роботи полягає у створенні готового до використання програмного інструментарію (фреймворку логування та модуля валідації) для екосистеми Java та Spring Boot.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Проблематика End-to-End тестування в мікросервісних системах

Протягом останнього десятиліття мікросервісна архітектура де-факто стала стандартом для розробки масштабованих корпоративних програмних продуктів. Розбиття великих монолітних систем на невеликі, слабко зв'язані та незалежно розгортувані сервіси дозволяє командам розробників працювати автономно, прискорювати цикл випуску програмного забезпечення та гнучко масштабувати окремі компоненти системи під навантаженням. Проте перехід від локальних викликів функцій у пам'яті до розподіленої взаємодії через мережу породжує різноманітні виклики у сфері забезпечення якості та тестування. У традиційному моноліті бізнес-транзакція зазвичай виконується в межах одного потоку, підпорядковуючись принципам ACID (Атомарність, Узгодженість, Ізольованість, Довговічність). У мікросервісах ці гарантії зникають, поступаючись місцем теоремі CAP (Узгодженість, Доступність, Стійкість). Цей зсув парадигми вимагає кардинального перегляду підходів до тестування.

У класичній піраміді тестування Майка Кона [3] (Mike Cohn) базовим рівнем є модульні (Unit) та інтеграційні тести, які в умовах мікросервісів працюють досить ефективно, оскільки перевіряють ізольовані компоненти або їхню взаємодію із зовнішніми залежностями (базами даних, брокерами повідомлень)

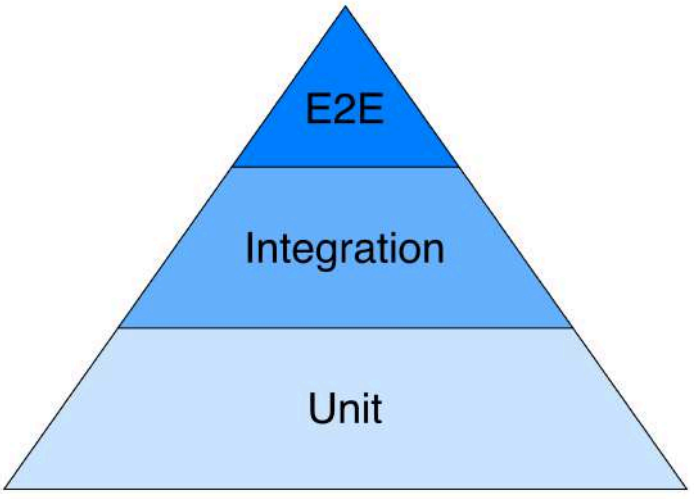

Рисунок 1.1 – Піраміда тестування

Однак найбільш проблемною стає вершина піраміди – наскрізне (E2E або його ще зображують як UI) тестування. Метою E2E-тестування є верифікація коректності роботи всієї системи з точки зору кінцевого користувача, що вимагає розгортання та одночасної роботи декількох, а іноді й декількох десятків мікросервісів.

Складність наскрізного тестування у розподілених системах зумовлена кількома ключовими факторами:

1. **Асинхронна комунікація та відсутність єдиного стану.** Виконання одного бізнес-сценарію (наприклад, оформлення замовлення) часто ініціює каскад подій. Запит, який починається як синхронний HTTP-виклик до API Gateway, далі може трансформуватися у внутрішні RPC-виклики (наприклад, через протокол gRPC) і завершуватися публікацією серії асинхронних подій у брокерах повідомлень (наприклад, Apache Kafka). Традиційні тестові фреймворки орієнтовані на синхронну парадигму «запит-відповідь» і не мають вбудованих механізмів для надійного відстеження завершення фонових асинхронних процесів.
2. **Проблема нестабільності тестів.** Через мережеві затримки та асинхронну обробку повідомлень час виконання бізнес-операції стає недетермінованим. Спроби вирішити цю проблему шляхом додавання статичних затримок або створення складних механізмів циклічного опитування баз даних призводять до того, що тести стають нестабільними. Вони можуть періодично завершуватися помилкою не через дефекти в бізнес-логіці, а через затримки мережі або тимчасове перевантаження брокера повідомлень.
3. **Жорстка зв'язність (Tight Coupling) тестів з інфраструктурою.** Класичні E2E-тести часто пишуться з прив'язкою до конкретних REST-ендпоінтів або структури баз даних окремих мікросервісів. Це порушує принцип інкапсуляції. Якщо під час архітектурного рефакторингу один мікросервіс розбивається на два або змінюється транспортний протокол (наприклад, перехід з REST на gRPC), бізнес-логіка залишається незмінною, але E2E-тести повністю ламаються і потребують переписування.

4. **Складність локалізації помилок.** Коли традиційний наскрізний тест завершується невдачею, він зазвичай повідомляє лише про факт невідповідності фінального результату очікуваному (наприклад, HTTP статус 500 замість 200). При цьому інженерам доводиться вручну збирати та зіставляти розрізнені логи з багатьох сервісів, щоб зрозуміти, на якому саме етапі розподіленого ланцюжка стався збій.

З огляду на вищезазначене, класичні підходи до E2E-тестування розподілених архітектур виявляються неефективними: вони стають «вузьким місцем» процесу розробки, уповільнюють розгортання нових версій та не дають достатнього рівня впевненості у якості продукту. Індустрія потребує переходу до нових парадигм валідації систем, які б базувалися не на жорсткому опитуванні кінцевих точок або UI-інтерфейсів, а на спостереженні за внутрішнім станом розподіленої системи за допомогою механізмів трасування.

1.2 Концепція спостережуваності

З переходом індустрії до мікросервісної архітектури, класичний термін «моніторинг» був витіснений ширшою концепцією «спостережуваності». Моніторинг відповідає на питання «Чи працює система?», опитуючи заздалегідь відомі показники (наприклад, використання процесора). Натомість спостережуваність – це властивість системи, яка дозволяє інженерам зрозуміти її внутрішній стан та причини збоїв виключно на основі зовнішніх вихідних даних (телеметрії), навіть якщо ці збої є новими і непередбачуваними.

В індустрії програмного забезпечення загальноприйнято виділяти три фундаментальні стовпи спостережуваності (Three Pillars of Observability [4]):

1. **Метрики (Metrics)**: Числові дані, агреговані за певні проміжки часу (наприклад, кількість запитів на секунду, відсоток помилок, середній час відповіді). Інструменти на зразок Prometheus та Grafana чудово збирають ці дані. Проте метрики є агрегованими: вони показують загальну тенденцію, але втрачають контекст окремої транзакції. Їх неможливо використати для точного E2E-тестування конкретного користувацького сценарію.

2. **Логи (Logs):** Деталізовані текстові або структуровані записи про події, що відбулися всередині конкретного компонента системи. Логи ідеально підходять для глибокого аналізу та відповіді на питання «Чому сталася помилка?» (наприклад, завдяки наявності стектрейсу винятку у полі message). Однак у розподіленій системі, де запит обробляється п'ятьма різними мікросервісами, розрізнені логи стають марними без механізму їх об'єднання.
3. **Трейси (Traces):** Відображають життєвий цикл єдиного запиту під час його проходження через усі компоненти розподіленої системи. Трейсинг дозволяє зрозуміти шлях виконання та знайти вузьке місце, відповідаючи на питання «Де саме стався збій?».

Методологія тестування, розроблена в цій роботі, базується на унікальному симбіозі другого та третього стовпів. Використовуючи структуровані аудит-логи, які містять обов'язкові поля traceId та spanId, система отримує деталізованість логів і топологічну зв'язність трейсів. Це створює ідеальний фундамент даних (Single Source of Truth), що дозволяє програмно валідувати складні бізнес-сценарії, залишаючи класичні метрики для операційного моніторингу на Production-середовищі.

1.3 Еволюція підходів до логування та трасування розподілених запитів

У традиційних монолітних системах моніторинг та відстеження життєвого циклу запиту були відносно тривіальними задачами. Оскільки весь код виконувався в межах одного процесу, усі лог-записи, згенеровані під час обробки транзакції, записувалися у файл послідовно. Інженеру було достатньо відкрити текстовий файл журналу та прочитати його зверху вниз.

У мікросервісній архітектурі цей підхід повністю втрачає свою ефективність. Коли один бізнес-сценарій ініціює виклики до п'яти різних сервісів, кожен із яких працює на власному сервері і пише логи у свій власний файл, виникає проблема втрати контексту. Особливо важким стає відстеження логів,

коли мікросервіс обробляє десятки, а то й сотні запитів одночасно. Тоді вже буде недостатньо відфільтрувати логи за часом, бо в цей відрізок часу потрапить велика кількість нерелевантних логів. Також цей процес важко автоматизувати через те, що отримати лог з помилкою часто недостатньо для того, щоб виправити її, через те, що контекст є неповним. Розробнику зазвичай потрібно бачити ланцюжок з логів, щоб побачити повну картину і знайти корінь проблеми.

Для розв'язання цієї проблеми індустрія адаптувала концепцію розподіленого трасування (Distributed Tracing), фундаментальні принципи якої були закладені у дослідженні інфраструктури Dapper компанією Google [5].

Система Dapper запровадила два ключових терміни, які сьогодні є стандартом індустрії:

1. **Слід (Trace):** Відображає повний шлях запиту через усю розподілену систему. Кожен трейс має єдиний, глобально унікальний ідентифікатор (Trace ID).
2. **Проміжок (Span):** Базовий блок побудови трейсу. Спан репрезентує одну логічну операцію (наприклад, виконання SQL-запиту, HTTP-виклик до іншого сервісу або обробку повідомлення з черги). Спани утворюють ієрархічну деревоподібну структуру, де кожен дочірній спан посилається на ідентифікатор батьківського (Parent Span ID).

Головна ідея полягає в тому, що на самому початку життєвого циклу запиту (наприклад, на рівні API Gateway або навіть на стороні клієнта) генерується унікальний глобальний ідентифікатор транзакції – **Trace ID**. Цей ідентифікатор автоматично передається між усіма компонентами системи разом із кожним мережевим запитом (HTTP, gRPC) або повідомленням у брокері (Kafka).

Довгий час кожен розробник фреймворків реалізовував передачу цього ідентифікатора по-своєму. Наприклад, екосистема Spring Cloud Sleuth використовувала заголовки формату B3 (*X-B3-TraceId*) [6], системи AWS використовували *X-Amzn-Trace-Id* [7], а інші рішення – *X-Request-Id* [8]. Це породжувало проблеми з інтеграцією різних мікросервісів, адже якщо трейс приходив не в тому форматі в якому його очікували, то контекст втрачався, і

генерувався новий *Trace ID*. Ланцюжок розривався, унеможливлюючи наскрізний аналіз.

Щоб розв'язати цю проблему, консорціум W3C затвердив єдиний міжнародний стандарт – **W3C Trace Context** [1]. Він стандартизує формат HTTP-заголовків:

- *traceparent*: містить загальну інформацію про версію (8 біт), Trace ID (128 біт) – унікальний ідентифікатор для всієї розподіленої транзакції. Він залишається незмінним, поки запит проходить через усі мікросервіси, Span ID (64 біти) – ідентифікатор конкретного запиту всередині транзакції. Він змінюється при кожному переході між сервісами, та прапорець семплювання (8 біт) – вказує, чи повинен цей запит бути записаний системою моніторингу. Приклад: *00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01*.
- *tracestate*: дозволяє вендорам додавати специфічну інформацію для своїх систем моніторингу, не порушуючи при цьому основного ідентифікатора.

Таке рішення гарантує збереження контексту транзакції, яка обробляється декількома мікросервісами, написаними на різних мовах програмування.

Забезпечення передачі *Trace ID* між сервісами – це лише половина завдання. У межах одного мікросервісу цей ідентифікатор потрібно прив'язати до кожного рядка логу, згенерованого під час обробки конкретного запиту. Передавати *Trace ID* як параметр у кожну Java-функцію (через сигнатуру методу) означало б катастрофічне забруднення бізнес-логіки інфраструктурним кодом.

Для вирішення цієї проблеми в екосистемі Java, збереження цього контексту в межах одного мікросервісу зазвичай реалізується через механізм Mapped Diagnostic Context (MDC), що дозволяє автоматично додавати *Trace ID* до кожного запису логу без необхідності явно передавати його у сигнатурах методів. MDC базується на класі *ThreadLocal* і дозволяє зберігати контекстні дані, прив'язані до конкретного системного потоку виконання. Як тільки запит потрапляє до контролера мікросервісу, спеціальний фільтр перехоплює заголовок *traceparent*,

витягує з нього *Trace ID* і кладе його в MDC. Далі цей ідентифікатор автоматично додається до усіх подальших викликів логера у межах цього потоку.

Саме наявність наскрізного ідентифікатора дає можливість перейти від хаотичного текстового виводу до структурованого логування, що є необхідною передумовою для автоматизації тестування.

1.4 Огляд та порівняльний аналіз існуючих інструментів

Для реалізації ефективного E2E-тестування на основі даних трасування необхідно проаналізувати наявні інструменти, що використовуються індустрією. Усі існуючі рішення можна умовно розділити на три категорії: низькорівневі бібліотеки логування, системи централізованого моніторингу та тестові фреймворки.

1.4.1 Бібліотеки логування в екосистемі Java

Стандартом де-факто для логування у світі Java є використання фасаду SLF4J (Simple Logging Facade for Java) [9] у поєднанні з імплементаціями рівня Logback або Apache Log4j2. Ці бібліотеки є надзвичайно оптимізованими, вони підтримують асинхронний запис у файли або мережеві сокети за допомогою кільцевих буферів (наприклад, LMAX Disruptor у Log4j2 [10]), що мінімізує блокування основного потоку виконання додатку. Крім того, ці бібліотеки реалізують механізм MDC, який дозволяє зберігати наскрізний ідентифікатор транзакції.

Проте ці інструменти є лише низькорівневим рівнем логування. Вони відповідають виключно за транспорт і форматування тексту, але не диктують жодних правил щодо семантики бізнес-подій. Використання чистого Logback призводить до того, що розробники форматують повідомлення довільно. Відсутність стандартизації повідомлень та обов'язкової структури (наприклад, у вигляді чітких JSON-об'єктів) робить неможливим їх надійний автоматизований парсинг тестовим фреймворком. Отже, для цілей Trace-Based Testing ці інструменти є необхідним фундаментом, але вимагають створення спеціалізованої

надбудови, яка б стандартизувала формат виводу та інтегрувала його зі специфікацією W3C Trace Context.

Для реалізації ефективного автоматизованого парсингу та аналізу розподілених запитів було проведено порівняння двох основних підходів до форматування виводу: **Plain Text** (традиційний) та **JSONL** (структурований).

1. **Plain Text (Звичайний текст)**

   Це формат за замовчуванням для більшості Java-бібліотек (Logback, Log4j2).

- Переваги: Висока швидкість читання людиною без додаткових інструментів.
- Недоліки для автоматизації:
    - Нестабільність структури: Розробники можуть довільно змінювати повідомлення, що ламає регулярні вирази парсера.
    - Багаторядкові логи: Стектрейси помилок розбиваються на декілька рядків, що ускладнює їх коректне збирання та прив'язку до конкретного запиту.
    - Складність парсингу: Вимагає написання складних регулярних виразів для виокремлення **Trace ID** чи **Timestamp**

2. **JSONL (JSON Lines)**

   Формат, де кожен окремий лог-запис є самостійним та валідним JSON-об'єктом, розміщеним на одному рядку.

- Переваги:
    - Читання програмою: Такий рядок легко прочитати і розпаристи в DTO-об’єкт програмно.
    - Гнучкість: Дозволяє легко додавати нові метадані, не порушуючи роботу існуючих тестів.
- Недоліки: Гірша читабельність файлу для людини через велику кількість додаткової інформації.

Хоча звичайний текст залишається зручним для швидкого налагодження вручну, для цілей Trace-Based Testing він є непридатним через високу ймовірність

помилок при автоматизованому зборі даних. Використання JSONL дозволяє працювати за принципом потокової обробки, де кожен рядок файлу є незалежною одиницею даних.

1.4.2 Системи централізованого збору та аналізу логів

Для роботи з розподіленими логами в робочому середовищі індустрія використовує потужні платформи типу ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, Datadog або спеціалізовані системи розподіленого трасування, такі як Jaeger та Zipkin.

У класичній архітектурі ELK мікросервіси зберігають свої логи локально. Потім проста програма-агент зчитує ці файли та відправляє їх у компонент обробки Logstash, який фільтрує дані та завантажує їх до пошукового рушія Elasticsearch. Після цього інженери використовують графічний інтерфейс Kibana для візуалізації трейсів та пошуку першопричин помилок [11].

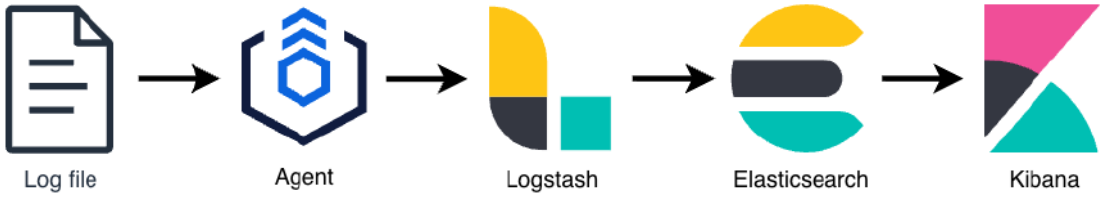

Рисунок 1.2 – Архітектура екосистеми ELK

Splunk є однією з найпотужніших пропрієтарних систем аналітики логів. Її архітектура дещо нагадує ELK, але базується на власних компонентах: на серверах мікросервісів встановлюються агенти *Universal Forwarders*, які збирають логи і відправляють їх до вузлів *Indexers* для збереження. Користувачі взаємодіють із системою через *Search Heads*, використовуючи спеціалізовану мову запитів SPL (Search Processing Language) [12]. Попри високу продуктивність, Splunk є важковаговим комерційним продуктом, розгортання якого для потреб тестів є економічно та технічно недоцільним.

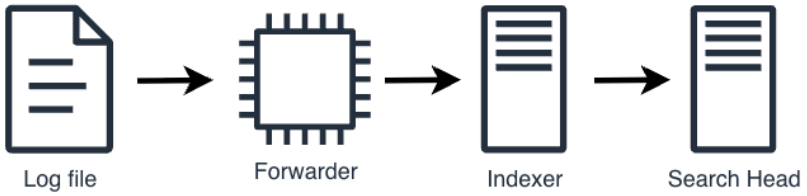

Рисунок 1.3 – Архітектура екосистеми Splunk

Datadog представляє інший підхід – моніторинг як послуга. Замість розгортання власної інфраструктури для збереження логів, компанії встановлюють на своїх серверах єдиний Datadog Agent. Цей агент збирає метрики, логи та наскрізні трейси, після чого асинхронно відправляє їх на хмарні сервери Datadog [13]. Використання такої платформи для автоматизованого тестування має критичний недолік: залежність від зовнішньої мережі. Відправка тестових даних у хмару та наступне їх опитування через зовнішнє API створює значні мережеві затримки та порушує принцип ізольованості тестового середовища.

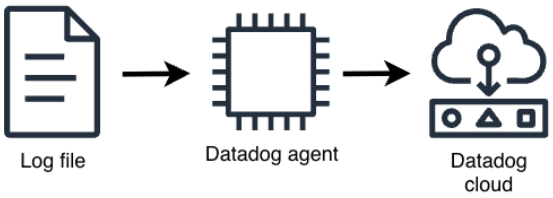

Рисунок 1.4 – Архітектура екосистеми Datadog

На відміну від вищезгаданих інструментів, які історично створювалися для роботи з текстом та метриками, системи Jaeger [14] (розроблена Uber) та Zipkin [15] (розроблена Twitter) від самого початку проєктувалися виключно для розподіленого трасування. Мікросервіси через спеціальні клієнтські бібліотеки відправляють дані про свої операції локальному Agent, який пересилає їх до центрального Collector. Далі дані зберігаються в базі, а графічний інтерфейс будує на їх основі візуальні дерева викликів та діаграми Ганта (Gantt charts) для аналізу таймінгів.

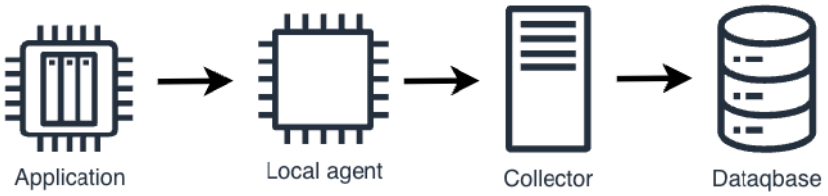

Рисунок 1.5 – Архітектура екосистеми Jaeger/Zipkin

Ці платформи ідеально підходять для візуального пошуку помилок та побудови графіків продуктивності. Проте їх використання в контурі автоматизованого локального тестування (на етапі розробки або в CI/CD пайплайнах) є недоцільним з кількох причин:

- **Надмірне споживання ресурсів:** Розгортання Elasticsearch та інфраструктури збору логів лише для запуску тестів вимагає значних обчислювальних потужностей.
- **Затримка індексації:** Системи типу ELK індексують дані із певною затримкою (від кількох секунд до хвилин). Тестовий фреймворк, який опитуватиме таку систему, зіткнеться з проблемою нестабільності.
- **Відсутність тестових перевірок:** Ці системи орієнтовані на людину-оператора, а не на програмну валідацію очікуваних результатів.

Тому для потреб автотестування необхідний значно легший інструмент, здатний працювати безпосередньо з файловими або стандартними потоками виводу мікросервісів у реальному часі.

1.4.3 Аналіз існуючих фреймворків тестування

В екосистемі Java існує розвинений набір інструментів для тестування мікросервісів. Такі бібліотеки як REST Assured дозволяють зручно тестувати HTTP-інтерфейси, а технологія Testcontainers стала стандартом для ізольованого запуску залежностей (баз даних, брокерів Kafka, інші мікросервіси) у Docker-контейнерах під час виконання тестів. Однак важливо чітко розмежовувати сферу застосування цих інструментів. Ці інструменти ідеально підходять для

інтеграційного тестування – тобто для перевірки роботи одного мікросервісу в ізоляції разом із його безпосередніми залежностями. Проте їх можливостей часто не вистачає для автоматизації повноцінного E2E тестування всієї розподіленої системи.

Проте ці тестові інструменти сфокусовані виключно на граничних інтерфейсах (API) системи. Наприклад, REST Assured може відправити запит до сервісу замовлень і дочекатися HTTP-відповіді 200 OK. Але він не має вбудованих механізмів для того, щоб переконатися, чи дійсно після цього запиту сервіс успішно відправив повідомлення у Kafka і чи успішно це повідомлення згодом обробив інший мікросервіс.

Спроби автоматизувати такі наскрізні сценарії за допомогою наявних інструментів призводять до створення громіздких конструкцій: розробникам доводиться підіймати додаткові сервіси всередині тесту або писати скрипти для прямого опитування баз даних кожного залученого мікросервісу. Через високу вартість розробки таких автотестів та їхню постійну нестабільність, індустрія сьогодні масово змушена покладатися на ручне E2E-тестування.

У більшості компаній фінальна перевірка наскрізних бізнес-процесів виконується QA-інженерами вручну в спеціальних тестових середовищах. Інженери імітують дії користувача (наприклад, через Postman або UI), а потім вручну перевіряють стан логів, баз даних або систем моніторингу, щоб підтвердити, що система досягла кінцевої узгодженості. Такий підхід створює критичне «пляшкове горлечко» у процесі розробки: ручне тестування є повільним, дорогим і фундаментально унеможливлює реалізацію справжньої безперервної доставки.

Таким чином, індустрії бракує спеціалізованої сполучної ланки – фреймворку, який міг би:

1. Збирати структуровані Trace-логи напряму з мікросервісів без необхідності піднімати важкі системи (ELK/Jaeger)
2. Надавати зручний програмний інтерфейс (Assertion API) для валідації повного асинхронного життєвого циклу запиту

3. Дозволити командам перевести наскрізні E2E перевірки з ручного режиму в автоматизований CI/CD контур

1.5 Висновки до розділу 1

Проведений у першому розділі системний аналіз предметної області дозволяє зробити такі висновки:

1. **Криза традиційного E2E-тестування.** Перехід індустрії до мікросервісної архітектури та парадигми кінцевої узгодженості суттєво ускладнив верифікацію наскрізних бізнес-процесів. Традиційні підходи до тестування, орієнтовані на синхронну взаємодію клієнт-сервер, не здатні надійно перевіряти складні асинхронні ланцюжки (наприклад, через брокери повідомлень). Це призводить до експоненційного зростання кількості нестабільних перевірок та змушує інженерні команди покладатися на ручне тестування, що стає головним бар'єром для реалізації процесів безперервної доставки (Continuous Delivery).
2. **Еволюція спостережуваності як технологічний фундамент.** Впровадження індустріальних стандартів розподіленого трасування, зокрема W3C Trace Context, та використання механізмів локального контексту потоку (MDC у Java) розв'язує проблему фрагментації та хаосу в логах. Наскрізний ідентифікатор транзакції (*Trace ID*) створює надійний фундамент для відстеження повного життєвого циклу розподіленого запиту.
3. **Прогалина в існуючому інструментарії.** Порівняльний аналіз наявних рішень виявив, що сучасні платформи централізованого логування та трасування (ELK Stack, Splunk, Datadog, Jaeger) оптимізовані виключно для пасивного моніторингу та візуального аналізу інцидентів людиною-оператором. Вони є занадто ресурсомісткими для локальних тестових середовищ і не мають вбудованих механізмів програмної валідації. Водночас популярні фреймворки (REST Assured, Testcontainers) ефективні лише на рівні інтеграційного тестування окремих компонентів, залишаючись «сліпими» до асинхронних процесів усієї системи.

Тому можна зробити висновок, що на сьогодні в екосистемі Java відсутній стандартизований інструмент, який дозволяв би автоматизувати End-to-End тестування на основі даних наскрізного трасування. Це повністю обґрунтовує актуальність та необхідність розробки спеціалізованої методології та програмного фреймворку (бібліотеки структурованого логування та модуля валідації трейсів).

РОЗДІЛ 2. РОЗРОБКА МЕТОДОЛОГІЇ ТА ПРОГРАМНОГО ФРЕЙМВОРКУ

2.1 Архітектурний підхід та методологія Trace-Based Testing

Розв'язання проблеми нестабільності та складності традиційного End-to-End тестування в розподілених системах вимагає фундаментальної зміни парадигми. Замість того, щоб покладатися на синхронні мережеві запити та ручне опитування баз даних, запропонована методологія базується на концепції Trace-Based Testing (тестування на основі трасування).

Головна ідея цього підходу полягає у використанні структурованих аудит-логів, об'єднаних єдиним ідентифікатором транзакції, як головного джерела правди для валідації стану системи. У межах даної роботи було спроєктовано та реалізовано спеціалізований програмний фреймворк, архітектура якого базується на трьох ключових принципах:

1. **Перехід від топологічної до семантичної валідації**

   Класичні автотести сильно пов'язані з фізичною топологією мережі: вони знають, що для створення замовлення треба викликати *Service A*, який звернеться до *Service B*. Розроблена методологія абстрагує тести від інфраструктури. Тестовий фреймворк розглядає систему як «чорний ящик» і валідує її не за назвами сервісів чи REST-ендпоінтів, а за фактом виконання семантичних бізнес-операцій (наприклад, *SEND_NOTIFCATION, SAVE_TO_DB, SEND_KAFKA_EVENT*). Це робить тести агностичними до архітектурного рефакторингу: якщо моноліт розбивається на мікросервіси, але загальна бізнес-логіка не змінюється, тести продовжують успішно працювати без жодних змін у коді перевірок.

2. **Стандартизована генерація подій**

   Щоб семантична валідація стала можливою, усі компоненти системи повинні спілкуватися однією мовою. Для цього розроблено концепт єдиного формату виводу – структурованого JSONL. Кожна важлива дія в мікросервісі генерує не просто рядок тексту, а машиночитаний об'єкт події, який обов'язково містить *Trace ID* (за стандартом W3C Trace Context) та

ідентифікатор операції. Таким чином створюється єдиний, наскрізний потік подій системи

3. **Асинхронне спостереження та відкладена перевірка**

   Оскільки розподілена система досягає кінцевої узгодженості через певний час, запропонований фреймворк відмовляється від жорстких таймаутів та синхронних блокувань. Натомість реалізовано механізм відкладеної перевірки: незалежний тестовий модуль асинхронно сканує спільний потік подій і очікує появи всієї необхідної послідовності бізнес-операцій для конкретного *Trace ID*. Як тільки всі очікувані події зафіксовані, тест миттєво позначається як успішний, що кардинально зменшує загальний час виконання тестового набору.

Загальний архітектурний флоу фреймворку:

Практична реалізація даної методології складається з двох взаємопов'язаних програмних компонентів, розроблених для екосистеми Java (Spring Boot):

1. Бібліотека для логування **tlog** (Trace Log Library): Проста бібліотека для мікросервісів, яка інкапсулює логіку взаємодії з Mapped Diagnostic Context (MDC), автоматично перехоплює наскрізні ідентифікатори та генерує валідні JSONL-події. Вона забезпечує єдиний стандарт виводу без необхідності розгортання важких систем моніторингу.
2. Автоконфігураційний модуль для Spring Boot **spring-boot-starter-tlog:** Готова до використання бібліотека логування, реалізована за архітектурним патерном Spring Boot Starter, що забезпечує інтеграцію в мікросервіси за принципом «нульової конфігурації». Завдяки вбудованим механізмам автоконфігурації екосистеми Spring, розробнику достатньо лише додати залежність у систему збірки. Бібліотека самостійно ініціалізується та підтягує метадані контексту.
3. Модуль тестування **TraceAsserter:** Незалежний інструмент для тестування. Він ініціюється під час виконання автотесту, приймає на вхід унікальний згенерований клієнтом *Trace ID* та надає декларативний програмний інтерфейс для конфігурації очікуваних результатів. Цей модуль перевіряє, чи

всі необхідні бізнес-операції були виконані, а також чи не було помилок під час виконання.

Такий розподіл відповідальності між генерацією телеметрії всередині мікросервісів та її аналізом ззовні дозволяє легко інтегрувати фреймворк у сучасні CI/CD пайплайни (зокрема при використанні інструментів контейнеризації, таких як Testcontainers), забезпечуючи високу надійність наскрізних перевірок.

2.2 Розробка бібліотеки структурованого логування

Для забезпечення можливості автоматизованої валідації розподілених бізнес-процесів необхідно усунути неоднозначність, притаманну традиційному текстовому виводу. Для цього було розроблено бібліотеку для логування **tlog**, а також розроблено модуль **spring-boot-starter** для простої інтеграції у середовище Spring Boot. Такий підхід дозволяє відокремити бізнес-логіку формування подій від інфраструктурних налаштувань конкретного фреймворку.

2.2.1 Ядро системи: Бібліотека tlog

Бібліотека **tlog** є фреймворконезалежним модулем, який містить ключову логіку формування структурованих повідомлень та серіалізації. Її основним завданням є стандартизація виводу та забезпечення цілісності моделі даних незалежно від середовища виконання. Важливою архітектурною особливістю розробленого модуля є те, що головний клас **TLog** безпосередньо імплементує стандартний інтерфейс **org.slf4j.Logger**. Це означає, що бібліотека діє як повноцінна заміна стандартних логерів.

Завдяки такій імплементації **tlog** повністю зберігає всі контракти SLF4J та надає знайомий усім Java-розробникам програмний інтерфейс (з класичними методами *debug(), info(), warn(), error()* та підтримкою параметризації рядків через фігурні дужки (*{}*). Це забезпечує нульовий поріг входження для інженерних команд: щоб перевести існуючий мікросервіс на новий інструмент логування, розробникам достатньо лише змінити тип об'єкта під час його ініціалізації чи

ін'єкції, тоді як весь існуючий код у бізнес-логіці залишиться повністю працездатним і не потребуватиме переписування.

Фундаментом розширеного функціоналу бібліотеки є її об'єктна модель, яка описує взаємодію між логером та структурою даних події. Архітектура спирається на використання структурного патерну для обгортання стандартного **Logger** та патерну Builder для безпечного створення об'єктів перенесення даних. На рисунку 2.1 наведено діаграму класів ядра фреймворку.

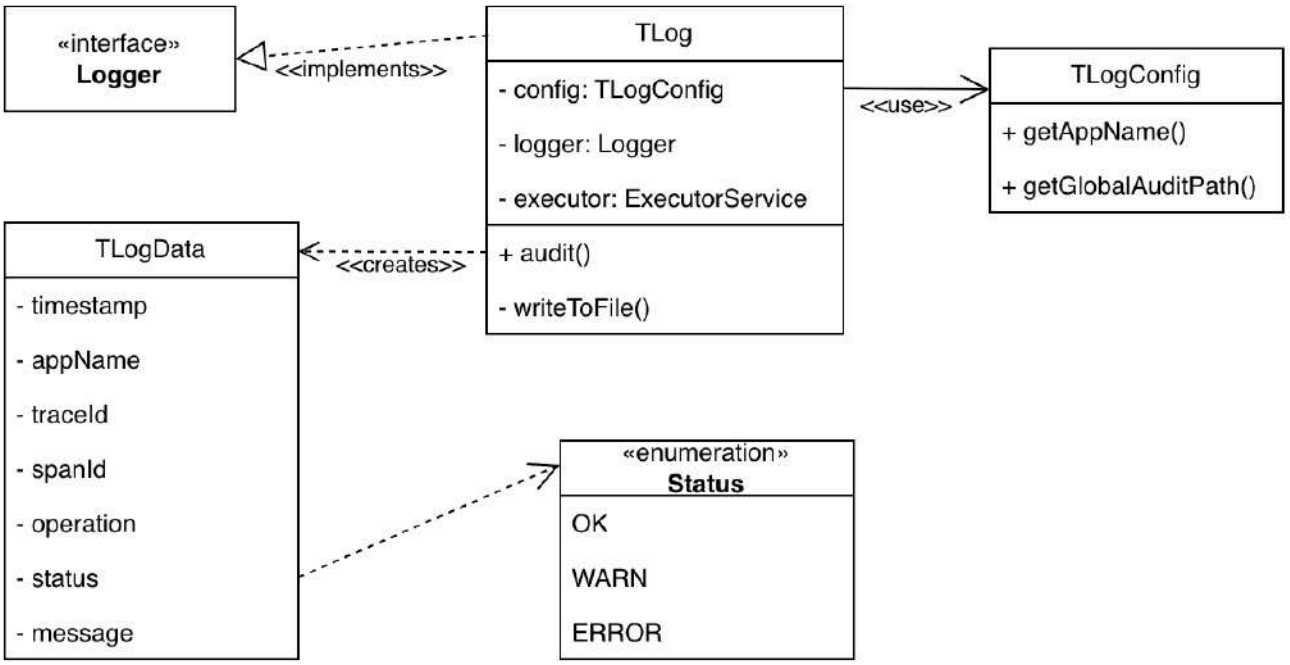

Рисунок 2.1 UML-діаграма класів об'єктної моделі бібліотеки tlog

Центральним елементом моделі є стандартизований клас **TLogData**, що визначає строгий контракт між мікросервісом, який генерує подію, та тестовим модулем, який її аналізує. Для забезпечення максимальної деталізації та сумісності зі стандартами трасування, клас **TLogData** містить такі ключові атрибути:

- **timestamp**: Точний час генерації події (наприклад, *2026-03-26 15:25:31.490*)
- **appName**: Назва мікросервісу, що згенерував подію (наприклад, *diplomaJavaKafka*)
- **traceId**: Унікальний 32-символьний ідентифікатор розподіленого запиту за стандартом W3C, який слугує наскрізним ключем для кореляції

- **spanId**: 16-символьний ідентифікатор конкретного проміжку або локальної операції, що дозволяє будувати ієрархічні дерева викликів
- **operation**: Семантичний ідентифікатор бізнес-операції (наприклад, *ORDER_CREATED, PAYMENT_PROCESSED*). Саме за цим полем тестовий фреймворк розпізнає ключові етапи бізнес-флоу
- **status**: Стан виконання операції (наприклад, *OK* або *ERROR*)
- **source**: Повне ім'я класу, де було згенеровано лог, що критично важливо для локалізації помилок
- **thread**: Назва системного потоку виконання
- **message**: Текстове повідомлення або додатковий контекст

Для серіалізації цих подій ядро tlog використовує формат JSON Lines. Кожен рядок у вихідному потоці є повністю незалежним і валідним JSON-об'єктом. Це робить його ідеальним для потокової обробки.


{"timestamp": "2026-03-07 11:50:38.555", "appName": "Order Service", "traceId": "99995678901234567890123456789012", "spanId": "4842bc85c4f83270", "status": "OK", "source": "edu.ukma.diploma.controller.OrderController", "thread": "http-nio-8080-exec-2", "message": "Order 123 has been created successfully", "operation": "ORDER_CREATED"}

Рисунок 2.2 – Приклад логу у формат JSON Lines

2.2.2 Модуль автоконфігурації – spring-boot-starter-tlog

Щоб максимально спростити інтеграцію ядра **tlog** у сучасні мікросервіси на базі Spring Boot, було розроблено додатковий модуль **spring-boot-starter-tlog**. Він реалізує архітектурний патерн автоконфігурації екосистеми Spring, забезпечуючи підключення за принципом нульової конфігурації.

Під час старту мікросервісу цей Starter автоматично виконує такі дії:

1. **Інтеграція з середовищем:** Зчитує метадані застосунку (значення *spring.application.name* для заповнення поля *appName*) і передає їх у конфігурацію ядра tlog. Встановлює розташування файлу для збереження логів.

2. **Управління життєвим циклом MDC:** Отримані **traceId** та **spanId** безпечно поміщаються у Mapped Diagnostic Context (MDC) системного потоку на початку обробки запиту і гарантовано очищаються після її завершення, запобігаючи витоку пам'яті та перехресному забрудненню контекстів.

Завдяки такому розділенню на ядро та модуль автоконфігурації, для кінцевого розробника використання цих інструментів є максимально прозорим. Бібліотека пропонує чітке розмежування між звичайним інформаційним логуванням та семантичним аудит-логуванням для Trace-Based тестування. Розробник може використовувати класичний вивід для загального контексту (наприклад, ***log.info("Event received: {}", event);***). Але для фіксації критичних бізнес-операцій, які підлягають автоматизованій перевірці, ядро **tlog** надає спеціалізований метод **audit**. Цей метод розширює можливості SLF4J, поєднуючи класичну параметризацію повідомлень із суворою типізацією бізнес-подій (наприклад, ***log.audit("SEND_EMAIL", Status.OK, "Notification for user has been sent");***).

2.2.3 Наскрізне передавання та управління життєвим циклом контексту

Критичним викликом при проєктуванні системи стала необхідність безперервного передавання ідентифікатора **traceId** між мікросервісами, які використовують кардинально різні мережеві протоколи взаємодії (HTTP, gRPC та Apache Kafka). Крім того, важливою вимогою до розробленого фреймворку логування є забезпечення наскрізного трасування без додаткового навантаження для бізнес-логіки. Передача ідентифікаторів **traceId** та **spanId** як аргументів у кожну функцію (через сигнатури методів) є антипатерном, оскільки це порушує принцип єдиної відповідальності (Single Responsibility Principle) та ускладнює підтримку коду.

Для забезпечення цілісності розподіленого трасування було прийнято рішення делегувати управління транспортним контекстом інструментарію Context

Propagation [16] з екосистеми OpenTelemetry, який відповідає за автоматичну ін'єкцію та екстракцію W3C-заголовків **traceparent**. Функції експорту та збереження даних повністю децентралізовані та делеговані бібліотеці **tlog**, що записує події безпосередньо у файлову систему. Управління ж внутрішнім контекстом у межах потоків виконання реалізовано через механізм Mapped Diagnostic Context (MDC) фасаду SLF4J.

Для мінімізації дублювання конфігурацій, модуль **spring-boot-starter-opentelemetry** було інтегровано не в кожен окремий мікросервіс, а інкапсульовано безпосередньо всередину розробленого модуля автоконфігурації **spring-boot-starter-tlog** як транзитивну залежність. Таким чином, мікросервісам-клієнтам достатньо підключити лише єдиний розроблений **tlog** стартер, який автоматично підтягує та ініціалізує всю необхідну інфраструктуру OpenTelemetry. Цей підхід дозволив реалізувати наскрізне передавання контексту за принципом автоінструментації (auto-instrumentation):

1. **HTTP та gRPC взаємодія:** Бібліотека OpenTelemetry автоматично перехоплює вихідні мережеві запити і вшиває у них стандартний заголовок **traceparent** згідно зі специфікацією W3C. На стороні приймача цей заголовок автоматично розпізнається, і контекст відновлюється.
2. **Асинхронна взаємодія через Apache Kafka:** Для передачі контексту через брокер повідомлень OpenTelemetry автоматично використовує механізм заголовків (Kafka Record Headers). Під час відправки події **traceparent** серіалізується у масив байтів і додається до метаданих повідомлення. На стороні споживача (Consumer) слухач **KafkaListener** витягує ці метадані та ініціалізує новий Span, який логічно прив'язаний до батьківського **traceId**.

Після того, як контекст подолав мережевий рівень і потрапив до мікросервісу, необхідно безпечно зберегти його для паралельної обробки. Під капотом механізм MDC реалізований з використанням класу *java.lang.ThreadLocal*. Оскільки класичні веб-фреймворки (на базі Tomcat) обробляють кожен HTTP-запит в окремому системному потоці, використання

ThreadLocal гарантує потокобезпечність і робить **traceId** глобально доступним з будь-якого місця коду, але виключно в межах поточного потоку.

Оскільки розробники не повинні самостійно взаємодіяти з MDC, інтеграція OpenTelemetry бере управління життєвим циклом на себе. Технічно це реалізовано на найнижчому рівні вебконтейнера за допомогою механізму сервлетних фільтрів (*jakarta.servlet.Filter*). Використання саме фільтра є критично важливим: він перехоплює HTTP-запит найпершим, ще до диспетчера Spring MVC, що гарантує збереження контексту на всіх етапах обробки. Життєвий цикл MDC у межах одного запиту складається з трьох фаз:

1. **Екстракція та Ініціалізація (Pre-Filter)**: Коли запит надходить до мікросервісу, фільтр OpenTelemetry аналізує вхідні HTTP-заголовки згідно зі специфікацією W3C Trace Context (**traceparent**). Відбувається екстракція ідентифікаторів traceId та spanId, на основі яких створюється або продовжується об'єкт Span. Після цього вмикається вбудований механізм синхронізації, який автоматично копіює ці значення у Mapped Diagnostic Context (MDC) фасаду SLF4J для поточного потоку.
2. **Використання (Execution)**: Під час виконання бізнес-логіки, коли розробник викликає метод *log.audit(...)*, ядро **TLog** звертається до методу *MDC.get("traceId")*. Завдяки **ThreadLocal**, система миттєво отримує ідентифікатор саме поточного запиту та вбудовує його в DTO-об'єкт **TLogData** перед серіалізацією.
3. **Обов'язкове очищення (After-Completion)**: Це найважливіша фаза з точки зору безпеки використання пам'яті. Контейнери сервлетів використовують механізм пулу потоків (Thread Pool). Коли обробка запиту завершується, потік не знищується операційною системою, а повертається до пулу для обслуговування наступних користувачів. Якби система не очищала MDC, наступний запит, який отримає цей же потік, успадкував би traceId попереднього користувача. Це явище називається «змішуванням контексту» (Context Leak) і призводить до катастрофічного руйнування дерев трасування. Для запобігання цьому, фільтр OpenTelemetry використовує

класичну конструкцію *try-finally*. У блоці *finally*, який гарантовано виконується перед поверненням потоку до пулу, викликається закриття поточного Span та інструкція *MDC.clear()*.

Таким чином, розроблена інфраструктура забезпечує абсолютно прозоре, високопродуктивне та безпечне щодо пам'яті управління контекстом транзакцій, створюючи надійну основу для подальшої агрегації телеметрії.

2.2.4 Вплив структурованого логування на продуктивність та управління конкурентним доступом

Будь-яке розширене логування має свою ціну з точки зору використання обчислювальних ресурсів. Генерація JSON-об'єктів для кожної бізнес-події, серіалізація об'єкта **TLogData** (який містить дев'ять полів: *timestamp, appName, traceId, spanId, operation, status, source, thread, message*) та їхній запис на жорсткий диск можуть стати «пляшковим горлечком» (bottleneck) системи. Синхронні операції вводу-виводу здатні суттєво знизити пропускну здатність мікросервісу під високим навантаженням. Для того щоб розроблена бібліотека **tlog** відповідала суворим вимогам до продуктивності та не уповільнювала роботу системи, в її архітектурі застосовано комбінацію асинхронної обробки та використання низькорівневих можливостей операційної системи.

1. **Оптимізація серіалізації об'єктів**

   Для перетворення об'єктів у формат JSONL використовується бібліотека Jackson (**ObjectMapper**). Щоб уникнути витрат процесорного часу та оперативної пам'яті на створення нових екземплярів серіалізатора для кожного запису, об'єкт **ObjectMapper** ініціалізується як одинак (Singleton) на етапі запуску програми.

2. **Асинхронне виконання запису**

   Замість того, щоб змушувати основний потік виконання (який обслуговує HTTP-запит користувача або повідомлення з Kafka) чекати на завершення дискової операції, процес збереження логів винесено в окремий фоновий пул потоків за допомогою *ExecutorService.newSingleThreadExecutor()*.

Коли розробник викликає метод логування *log.audit(...)*, основний потік лише формує об'єкт TLogData і передає його як завдання на виконання у фоновий потік. Після цього він миттєво повертається до виконання бізнес-логіки. Таким чином, затримка для користувацького запиту, спричинена роботою бібліотеки **tlog**, зводиться практично до нуля.

3. **Вирішення проблеми конкурентного доступу**

   Окремим архітектурним викликом була робота у тестовому середовищі Docker Compose, де чотири незалежні мікросервіси одночасно записують події у єдиний фізичний файл *audit.log* на хост-машині. Традиційне використання довготривалих відкритих потоків (наприклад, BufferedWriter) неминуче призвело б до стану гонитви (Race Condition) та перемішування байтів від різних сервісів. Для елегантного вирішення цієї проблеми механізм запису у **tlog** реалізовано з використанням системного виклику *java.nio.file.Files.write* з константою *StandardOpenOption.APPEND*. На рівні POSIX-сумісних операційних систем ця інструкція транслюється у прапорець *O_APPEND*. Завдяки цьому операційна система на рівні ядра самостійно гарантує атомарність операції запису для кожного окремого рядка JSONL. Система автоматично керує блокуваннями файлу між різними процесами, унеможливлюючи перехресне забруднення логів.

Таким чином, поєднання асинхронного фонового запису (на рівні віртуальної машини Java) та атомарного додавання рядків (на рівні ядра ОС) гарантує, що генерація аудит-логів для E2E-тестування є абсолютно безпечною та ресурсоефективною операцією, яка не призводить до деградації роботи мікросервісів.

2.3 Проєктування модуля автоматизованої валідації

Наявність стандартизованого потоку аудит-логів є необхідною, але недостатньою умовою для автоматизації тестування. Для того, щоб розробники могли ефективно перевіряти систему, було спроєктовано та реалізовано незалежний тестовий модуль – клас **TraceAssertion**. Цей модуль аналізує

згенеровані мікросервісами телеметричні дані та підтверджує (або спростовує) факт коректного виконання розподіленого бізнес-сценарію. Архітектура модуля **TraceAssertion** побудована на чотирьох ключових концепціях:

1. **Патерн Fluent API (Декларативний інтерфейс)**

   Для мінімізації когнітивного навантаження на розробників тестів, конфігурація перевірок реалізована за допомогою патерну Builder з зрозумілим інтерфейсом (Fluent API). Замість написання десятків окремих інструкцій assert, інженер створює єдиний ланцюжок очікувань. Модуль надає набір семантичних методів:

   a. **hasApps()** – перевіряє участь певних мікросервісів у транзакції;
   b. **hasOperations()** – валідує наявність конкретних бізнес-кроків;
   c. **hasJavaClasses()** – підтверджує виконання логіки у вказаних компонентах коду;
   d. **completedWithoutErrors()** – перевірє відсутність помилок
   e. **completedWithoutWarnings()** – гарантує відсутність попереджень

2. **Механізм відкладеної перевірки (Polling)**

   Ще однією архітектурною особливістю модуля є відмова від жорстких затримок (на зразок Thread.sleep(5000) у самому коді тесту), які є головною причиною нестабільності автотестів. Натомість реалізовано алгоритм розумного циклічного опитування (Polling). Під час виклику фінального методу **verify()**, модуль фіксує дедлайн (**deadlineMillis**) на основі заданого таймауту. Далі запускається цикл: фреймворк зчитує лог-файл, перевіряє виконання всіх умов, і якщо вони не виконані – засинає на 200 мілісекунд, після чого повторює спробу. Це забезпечує ідеальну оптимізацію часу виконання: якщо асинхронний процес у системі завершиться за 300 мс, тест миттєво позеленіє і піде далі, не чекаючи закінчення повного 5-секундного таймауту.

3. **Потокове читання та парсинг (Stream Processing)**

   Оскільки загальний лог-файл тестового середовища може швидко збільшуватися у розмірах, TraceAssertion використовує клас java.nio.file.Files

для потокового читання файлу рядок за рядком. Кожен рядок динамічно десеріалізується з формату JSON у DTO-об'єкт **TLogData** за допомогою бібліотеки Jackson (ObjectMapper). Після цього застосовується фільтрація за унікальним **traceId**. Це гарантує, що тестовий модуль аналізує виключно події конкретної транзакції, повністю ігноруючи фоновий шум від інших паралельно працюючих тестів чи мікросервісів.

4. **Деталізована локалізація помилок (Root Cause Analysis)**

   Якщо таймаут вичерпано, а система так і не досягла очікуваного стану, метод **assertAllConditions()** не просто викидає стандартну помилку, а генерує розширений звіт. Алгоритм проходить по кожній невиконаній умові, збирає всі знайдені помилки (наприклад, статуси ERROR з інших сервісів) і формує єдиний виняток AssertionError з детальним описом того, на якому саме етапі обірвався ланцюжок. Це зводить час на пошук першопричини до мінімуму.

2.3.1 Архітектура десеріалізації та парсингу JSONL-потоку

Головним завданням модуля автоматизованої валідації **TraceAsserter** є видобуток подій із загального файлу аудит-логів та перетворення сирих текстових даних (рядків у форматі JSON Lines) у строгі типізовані Java-об'єкти **TLogData** для подальшого семантичного аналізу. Це завдання виконується за допомогою конвеєра десеріалізації.

Перетворення текстових рядків у DTO-об'єкти виконується за допомогою високопродуктивної бібліотеки Jackson (**ObjectMapper**). Враховуючи специфіку розподілених середовищ на базі контейнеризації (наприклад, Docker), алгоритм парсингу спроєктовано з урахуванням стійкості до пошкодження даних.

У реальних умовах, якщо під час запису логу один із мікросервісів був раптово зупинений (наприклад, через примусову зупинку сервіса), у спільному файлі може з'явитися обірваний, синтаксично невалідний JSON-рядок. Класичний підхід до парсингу у такій ситуації неминуче призвів би до викидання **JsonParseException** та повного падіння процесу валідації тесту. Щоб уникнути

цієї вразливості, функція-мапер фреймворку **TraceAsserter** інкапсулює процес десеріалізації кожного окремого рядка у блок безпечного перехоплення помилок (*try-catch*). Якщо алгоритм натрапляє на пошкоджений JSON, помилка логується на рівні **ERROR**, а метод повертає **null**. Це гарантує високу відмовостійкість: злам одного запису не блокує перевірку інших, повністю валідних подій у потоці.

Логіка видобутку цільових подій реалізована як єдиний ланцюжок трансформацій. Конвеєр складається з наступних етапів:

1. **Зчитування:** Отримання сирого текстового рядка з файлу.
2. **Десеріалізація:** Безпечне перетворення рядка на об'єкт **TLogData** (з обробкою можливих винятків).
3. **Первинна фільтрація:** Відсіювання невалідних об'єктів, що виникли внаслідок пошкодження логів.
4. **Семантична фільтрація:** Відсіювання валідних подій, які належать до інших паралельних транзакцій, шляхом суворого порівняння поля **traceId** з ідентифікатором поточного тесту.

Після проходження цього конвеєра система отримує чистий набір подій для конкретного тестового сценарію, який далі передається на етап застосування семантичних перевірок.

2.3.2 Оптимізація продуктивності та безпека управління пам'яттю

Під час розробки модуля **TraceAssertion** особлива увага приділялася ефективному використанню системних ресурсів. У реальних умовах загальний лог-файл тестового середовища може за лічені хвилини зростати до сотень мегабайт, оскільки він агрегує події від усіх сервісів.

Класичний підхід до читання файлів (наприклад, метод Files.readAllLines()) завантажує весь вміст у оперативну пам'ять, що може призвести до винятку OutOfMemoryError у контейнеризованих середовищах CI-серверів з обмеженими ресурсами. Для усунення цієї вразливості клас **TraceAssertion** спроєктовано з використанням концепції відкладених обчислень. Метод потокового читання

реалізовано за допомогою *java.nio.file.Files.lines()*, який створює лінивий потік рядків. Завдяки цьому підходу в пам'яті одночасно знаходиться лише один рядок логу, що парситься і відфільтровується. Складність за пам'яттю для зчитування файлу становить *O(1)*, тоді як у пам'ять записується лише відфільтрований мікро-набір подій, які належать виключно до поточного **traceId** (складність *O(K)*, де *K* – кількість кроків конкретної транзакції). Це дозволяє запускати тестовий фреймворк на машинах із мінімальним обсягом оперативної пам’яті без втрати продуктивності.

2.4 Висновки до розділу 2

У другому розділі кваліфікаційної роботи було запропоновано, архітектурно спроєктовано та програмно реалізовано інструментарій для автоматизації End-to-End тестування мікросервісних систем.

Головним результатом розробки став перехід від традиційної (синхронної та топологічно залежної) моделі перевірок до парадигми Trace-Based Testing – семантичної валідації розподілених бізнес-процесів на основі єдиного потоку телеметрії. Запропоноване рішення розділено на два ключові компоненти, що забезпечують низьку зв'язність системи:

1. **Ядро та механізми генерації подій:** Створено бібліотеку **tlog**, адаптовану для екосистеми Spring Boot через патерн автоконфігурації. Вона функціонує як прозора надбудова над стандартом SLF4J. Завдяки глибокій інтеграції з індустріальним стандартом OpenTelemetry та використанню сервлетних фільтрів, реалізовано безпечне управління життєвим циклом контексту (MDC). Це гарантує безперервне наскрізне трасування при перетині меж різних мережевих протоколів (REST, gRPC, Apache Kafka) без перевантаження бізнес-логіки. Крім того, механізм запису у формат JSON Lines було оптимізовано за допомогою пулу фонових потоків та атомарних дискових операцій, що гарантує відсутність впливу логування на продуктивність мікросервісів.

2. **Модуль асинхронної валідації:** Спроєктовано клас TraceAsserter, що виступає в ролі незалежного арбітра результатів виконання транзакцій. Алгоритм парсингу логів побудовано на основі концепції відкладених обчислень (Lazy Stream Processing) та безпечної десеріалізації. Це забезпечило константну просторову складність O(1) при зчитуванні файлів будь-якого розміру та високу відмовостійкість системи. Використання патерна Builder дозволило створити декларативний API для інженерів, який інкапсулює складність асинхронного опитування (Polling) та створює надійний фундамент для наступного етапу – практичної інтеграції в тестове середовище.

РОЗДІЛ 3. АПРОБАЦІЯ ФРЕЙМВОРКУ ТА ОЦІНКА ЙОГО ЕФЕКТИВНОСТІ

3.1 Оркестрація інфраструктури тестового середовища

Для практичного підтвердження ефективності розробленої методології Trace-Based Testing було спроєктовано та розгорнуто повноцінне розподілене тестове середовище. Головною метою цього етапу було створення інфраструктури, яка б максимально точно імітувала складність реальних корпоративних систем (з використанням синхронних та асинхронних подій), залишаючись при цьому достатньо легкою для запуску на локальній машині розробника або в ізольованому контурі Continuous Integration.

Тестовий стенд розгорнуто за допомогою технології контейнеризації Docker Compose. Така оркестрація гарантує детермінованість середовища: кожен запуск тестів відбувається в ідентичних умовах, усуваючи проблему різного середовища. Інфраструктура складається з брокера повідомлень Apache Kafka (включно з UI-панеллю для візуального контролю топіків) та чотирьох незалежних мікросервісів, що реалізують складний ланцюжок обробки:

1. **root сервіс**, виконує роль вхідної точки. Він приймає запити від користувачів і маршрутизує їх далі.
2. **grpc сервіс**, що імітує високопродуктивну внутрішню взаємодію через бінарний протокол gRPC.
3. **rest сервіс**, допоміжний мікросервіс для обробки класичних HTTP-викликів.
4. **kafka сервіс**, асинхронний споживач, який обробляє події опубліковані іншими сервісами у брокері повідомлень.

Ключовим архітектурним та оптимізаційним рішенням для цього тестового контуру стало використання механізму спільних томів (Shared Volumes). У промисловому середовищі кожен сервіс відправляв би логи по мережі до агента. Проте для потреб швидкого E2E-тестування це створює непотрібні мережеві

затримки. Замість цього через налаштування Docker усі чотири мікросервіси отримали доступ до єдиного фізичного файлу на хост-машині. Завдяки тому, що бібліотека **tlog** пише дані в асинхронному режимі з використанням системних блокувань, вдалося досягти агрегації телеметрії в режимі реального часу без конфліктів доступу до файлу. Цей файл *audit.log* став ідеальним та миттєвим джерелом даних для модуля **TraceAsserter**.

Для перевірки фреймворку було спроєктовано наскрізний бізнес-сценарій, який навмисно поєднує синхронні та асинхронні етапи виконання. Потік взаємодії компонентів наведено на діаграмі послідовності.

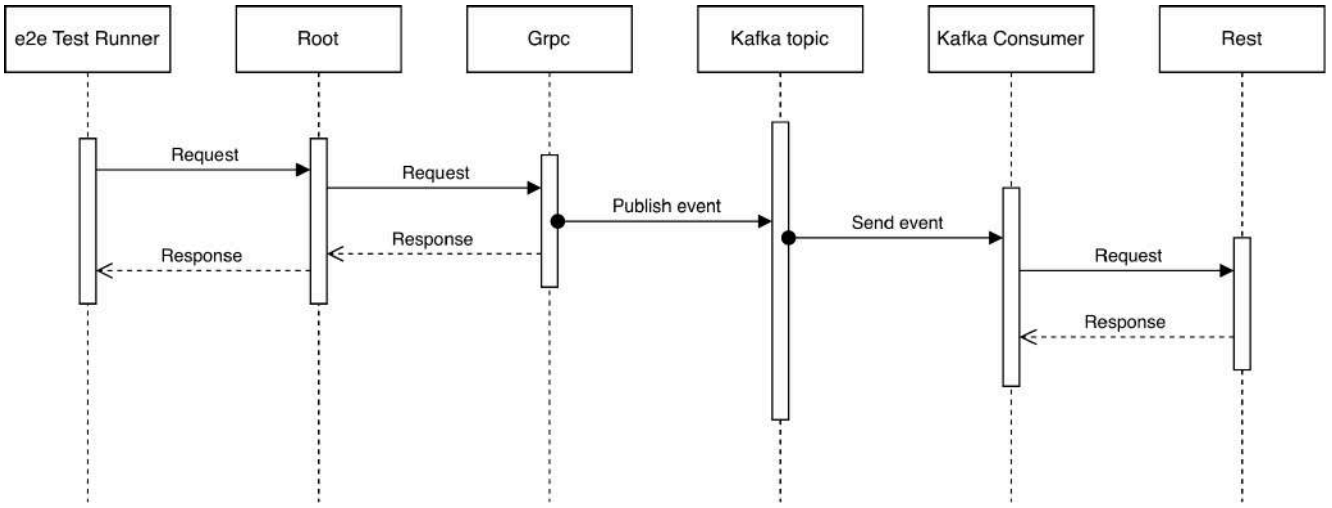

Рисунок 3.1 - Діаграма послідовності виконання E2E-тесту

Як видно з діаграми, життєвий цикл запиту розгалужується:

1. **Синхронна гілка:** E2E Test Runner відправляє запит до **Root**, який викликає **Grpc**. Сервіс **Grpc** публікує подію у **Kafka topic** та відразу повертає відповідь назад по ланцюжку, і тестовий раннер отримує HTTP Response.
2. **Асинхронна гілка: Kafka Consumer** вичитує подію з **Kafka topic** і робить HTTP-запит до сервісу **Rest**.

Фундаментальна проблема класичного тестування, яку яскраво демонструє ця діаграма, полягає в тому, що у момент отримання HTTP-відповіді тестовим раннером, асинхронна гілка (обробка в Kafka та виклик Rest) ще не завершена.

3.2 Реалізація наскрізного тестового сценарію

Щоб надійно верифікувати такий гібридний сценарій було розроблено окремий незалежний застосунок **E2eTestApp**. Процес тестування був розділений на три логічні фази:

Фаза 1: Ініціалізація та ін'єкція контексту. Щоб мати можливість дістати потрібні логи саме для тестового запиту, тестовий клієнт бере управління контекстом на себе. Він генерує унікальний *traceId* (через *UUID.randomUUID()*) та вручну формує стандартний HTTP-заголовок W3C Trace Context.

```
String traceId = UUID.randomUUID().toString().replace( target: "-",  replacement: "");
try (HttpClient client = HttpClient.newHttpClient()) {
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://localhost:8081/start/validation"))
            .header( name: "Traceparent", "00-%s-1111111111111111-01".formatted(traceId))
```

Рисунок 3.2 - Ініціалізація тестового запиту

Формат *00-{traceId}-1111111111111111-01* чітко відповідає специфікації консорціуму W3C, де *00* – версія протоколу, фіксований спан *1111111111111111* імітує батьківський виклик з боку клієнта, а прапорець *01* вказує на обов'язковість збереження цього трейсу всіма подальшими сервісами.

Фаза 2: Виконання граничної перевірки. Клієнт відправляє HTTP POST запит із корисним навантаженням і перевіряє отримання первинної синхронної відповіді від **root** сервісу (**200 OK**). Цей крок є критичним, але недостатнім: статус 200 підтверджує лише факт прийняття повідомлення до обробки, але система не дає жодних гарантій щодо того, чи не впаде Kafka-споживач через мілісекунду після цього.

Фаза 3: Відкладена семантична валідація асинхронної гілки. Після отримання HTTP-відповіді управління передається модулю **TraceAsserter**. Розпочинається процес циклічного опитування спільного лог-файлу. Фреймворк чекає, поки асинхронна частина діаграми (Kafka Consumer → Rest) повністю відобразиться у вигляді JSONL-подій:

```
traceAsserter.awaitForTrace(traceId, Duration.ofSeconds(5))
        .hasApps(
                "diplomaJavaMzrkRoot",
                "diplomaJavaRest",
                "diplomaJavaGrpc",
                "diplomaJavaKafka"
        )
        .hasJavaClasses(
                "edu.ukma.diplomajavakafka.service.KafkaConsumerService"
        )
        .hasOperations(
                "ROOT_VALIDATION_CALLED",
                "ROOT_VALIDATION_COMPLETED",
                "GRPC_VALIDATE_CALLED",
                "GRPC_VALIDATE_COMPLETED",
                "DIPLOMA_REST_CALLED",
                "DIPLOMA_REST_COMPLETED",
                "KAFKA_VALIDATION_CALLED",
                "KAFKA_VALIDATION_COMPLETED")
        .completedWithoutErrors()
        .completedWithoutWarnings()
        .verify();
```

Рисунок 3.3 - Приклад використання тестового модуля на тестовому стенді

3.3 Аналіз результатів та валідація методології

Після успішної реалізації та запуску наскрізних тестових сценаріїв на базі розгорнутої мікросервісної інфраструктури, наступним етапом дослідження є комплексний аналіз отриманих результатів.

Валідація розробленого програмного рішення здійснювалася за кількома ключовими критеріями: здатність фреймворку коректно відстежувати успішні асинхронні транзакції, стійкість до мережевих затримок, реакція на збої в окремих компонентах системи, а також загальне зниження складності розробки та підтримки тестової кодової бази.

Аналіз загального потоку телеметрії під час виконання успішних сценаріїв підтвердив головну гіпотезу дослідження: використання агрегованих аудит-логів дозволяє тестовому модулю динамічно підлаштовуватися під реальну швидкодію

мікросервісів. Тест успішно завершувався в момент досягнення системою кінцевої узгодженості, повністю нівелюючи проблему жорстко заданих таймаутів. Для більш глибокого аналізу надійності фреймворку було досліджено його поведінку в нештатних ситуаціях.

3.3.1 Поведінка фреймворку при виявленні дефектів

Особливу увагу під час апробації було приділено здатності фреймворку локалізувати помилки. Було проведено серію експериментів із симуляцією збоїв (наприклад, відключення брокера Kafka). У традиційних тестах це призвело б до неінформативного повідомлення «Timeout Exceeded». Натомість **TraceAsserter**, завдяки агрегації логів, миттєво перериває очікування при виявленні подій зі статусом *ERROR* (якщо налаштовано *completedWithoutErrors()*) і виводить у консоль точне місце збою: назву сервісу, клас та зміст винятку. Це зводить час пошуку першопричини (Root Cause Analysis) до мінімуму

У результаті запуск стенду підтвердив, що **TraceAsserter** коректно зчитує лог-файл, ідентифікує події з різних джерел та успішно завершує виконання тесту в момент досягнення кінцевої узгодженості (Eventual Consistency), зазвичай за 200-400 мс, не чекаючи завершення жорстко заданого таймауту. Це повністю вирішує проблему нестабільності тестів та доводить готовність розробленого рішення до використання у виробничих пайплайнах.

3.3.2 Порівняльний аналіз складності автоматизації тестових сценаріїв

Для об'єктивної оцінки ефективності розробленого фреймворку необхідно порівняти його з традиційними підходами до E2E-тестування. Розглянемо наскрізний гібридний сценарій, проілюстрований раніше на діаграмі послідовності (Рисунок 3.1): початковий запит надходить через REST до сервісу **root**, далі виконується синхронний RPC-виклик до **grpc**, після чого генерується асинхронна подія в топіку **Kafka**, яку зрештою вичитує інший мікросервіс і здійснює фінальний HTTP-виклик до сервісу **rest**.

Алгоритм традиційного підходу (Класичне E2E-тестування)

Щоб автоматизувати цей сценарій за допомогою стандартних інструментів, інженеру з якості доводиться писати складний, інфраструктурно залежний алгоритм перевірки:

1. **Ініціалізація інфраструктури в тесті:** Необхідно зробити окремий тестовий Kafka Consumer та підключитися до брокера повідомлень, щоб мати змогу перехоплювати проміжні події і перевіряти їх обробку. А також налаштувати доступ до баз даних всіх мікросервісів, щоб перевірити, чи запит був успішно оброблений, або ці сервіси мають мати додатковий ендпоінт, щоб можна було запитати результат виконання.
2. **Запуск тесту:** Тестовий застосунок відправляє початковий запит і перевіряє отримання успішної HTTP-відповіді.
3. **Валідація кінцевого стану:** Оскільки у нас є декілька сервісів, які виконуються, нам необхідно перевірити, що кожен з них завершив свою роботу успішно. Тому спершу нам потрібно перевірити, що подія була опублікована в брокері повідомлень. Потім перевірити, що решта сервісів успішно виконала свої задачі. Оскільки фінальний крок виконується асинхронно іншим сервісом, тест запускає цикл опитування – безпосередньо звертаючись до бази даних або до спеціально створеного тестового API, щоб переконатися, що запит успішно оброблений.

Недоліки такого підходу очевидні:

1. **Надмірна зв'язність:** Тест знає занадто багато деталей реалізації (назви топіків Kafka, структуру бази даних). Будь-який рефакторинг інфраструктури гарантовано ламає тестові сценарії.
2. **Складність впровадження:** Для такого тестування потрібно додатково готувати інфраструктуру. Доступи до баз даних, брокера повідомлень, чи створення тестових ендпоінтів. Все це створює додаткове навантаження на команди з розробки.

Алгоритм розробленого підходу (Trace-Based Testing)

На противагу традиційному методу, розроблений фреймворк абстрагує інженера від фізичної топології мережі. Алгоритм тестування зводиться до кількох логічних кроків:

1. **Збір логів:** Необхідно зібрати логи усіх сервісів в одному місці, щоб тестовий модуль мав до них доступ. Зазвичай цей механізм уже налаштований, наприклад, за допомогою ELK.
2. **Ініціалізація тестового модуля:** Тест ініціалізує модуль **TraceAsserter**, вказуючи йому шлях до спільного файлу аудит-логів. Жодних підключень до баз даних чи брокерів повідомлень не вимагається.
3. **Ін'єкція контексту:** Тест генерує унікальний **traceId** і вставляє його у стандартний HTTP-заголовок W3C Trace Context початкового запиту.
4. **Запуск тесту:** Відправка запиту та перевірка первинної відповіді (статус 200).
5. **Декларативна семантична перевірка:** Замість написання циклів опитування баз даних, тест передає модулю валідації **traceId** та масив семантичних назв очікуваних бізнес-операцій (наприклад, *SEND_NOTIFICATON*, *PAYMENT_PROCESSED*).
6. **Асинхронний аналіз:** Модуль **TraceAsserter** бере на себе всю складність очікування. Він асинхронно сканує єдиний потік подій, поки не зафіксує виконання всього замовленого ланцюжка або поки не виявить помилку.

Ключові переваги розробленої методології:

- **Інкапсуляція:** Тест діє як справжній чорний ящик. Він оперує виключно бізнес-термінами (назвами операцій), ігноруючи транспортний рівень.
- **Повне покриття:** Фреймворк гарантовано валідує всі етапи транзакції, включаючи глибокі внутрішні RPC-виклики.
- **Зниження витрат на підтримку:** Відмова від написання зайвого коду для налаштування тестових брокерів та конекторів баз даних значно зменшує

обсяг тестової бази, підвищує її читабельність і кардинально знижує поріг входження для нових спеціалістів.

3.4 Інтеграція інструментарію в конвеєр Continuous Integration

Розроблений фреймворк спроєктований не лише для локального запуску, а й для повної автоматизації в системах CI/CD (таких як GitHub Actions, GitLab CI або Jenkins).

Оскільки TraceAsserter є стандартним Java-застосунком, його інтеграція у пайплайн є безшовною. Під час збірки проєкту CI-сервер піднімає Docker Compose інфраструктуру, після чого запускається контейнер із модулем E2E-тестів.

Критично важливою архітектурною рисою методу *verify()* є те, що при невиконанні умов він викидає стандартний виняток **AssertionError**. У контексті CI-пайплайну це автоматично конвертується у ненульовий код завершення процесу. Це означає, що якщо будь-яка гілка асинхронної транзакції відпрацювала з помилкою (або взагалі не виконалася), система безперервної інтеграції миттєво блокує розгортання зламаного артефакту, гарантуючи, що на Production-середовище потрапить лише перевірений та надійно узгоджений код.

3.5 Висновки до розділу 3

У третьому розділі було проведено комплексну практичну апробацію розробленої методології Trace-Based Testing та програмного фреймворку в умовах, максимально наближених до реального промислового середовища.

Розгортання розподіленого тестового стенду на базі оркестратора Docker Compose дозволило змоделювати складний наскрізний бізнес-сценарій, який поєднує синхронні (REST, gRPC) та асинхронні (Apache Kafka) канали комунікації мікросервісів. Практична перевірка підтвердила життєздатність запропонованих архітектурних рішень та виявила такі ключові переваги розробленого інструментарію:

1. **Нівелювання проблеми нестабільних тестів:** Доведено, що використання модуля TraceAsserter ефективно відстежує життєвий цикл складних транзакцій. Завдяки механізму відкладеного семантичного аналізу агрегованих аудит-логів, фреймворк динамічно підлаштовується під швидкодію системи, успішно завершуючи перевірку в момент досягнення кінцевої узгодженості і повністю усуваючи негативний вплив мережевих затримок без використання жорстко заданих таймаутів.
2. **Ефективна локалізація дефектів:** Аналіз поведінки системи під час симуляції збоїв довів, що фреймворк забезпечує миттєву локалізацію помилок на рівні конкретних класів у розподілених модулях, що значно пришвидшує процес відлагодження.
3. **Зниження складності тестової кодової бази:** Проведений порівняльний аналіз продемонстрував, що розроблений підхід дозволяє суттєво скоротити обсяг коду E2E-тестів порівняно з традиційними інструментами. Це досягається завдяки усуненню необхідності налаштування допоміжних інфраструктурних об'єктів (наприклад, тестових Kafka-консьюмерів) та переходу до декларативного Fluent API.

Таким чином, результати апробації повністю підтверджують ефективність створеного рішення. Розроблений інструментарій готовий до безшовної інтеграції у процеси безперервної інтеграції та розгортання (CI/CD) у якості надійного блокувальника дефектних релізів. Він дозволяє командам інженерів замінити складне та повільне ручне тестування гібридних потоків на швидкі, стабільні та топологічно незалежні автоматизовані перевірки.

ВИСНОВКИ

У магістерській кваліфікаційній роботі розв’язано актуальне науково-прикладне завдання: розроблено методологію та фреймворк для надійного, швидкого та незалежного від топології End-to-End (E2E) тестування розподілених мікросервісних систем на основі даних наскрізного трасування (Trace-Based Testing). У результаті проведеного дослідження та практичної розробки отримано такі основні результати:

1. **Проаналізовано проблематику та обмеження класичних підходів до тестування.** Доведено, що перехід індустрії до мікросервісної архітектури та широке використання асинхронних патернів взаємодії (Event-Driven Architecture) роблять традиційні інструменти (орієнтовані на синхронні REST-виклики) неефективними. Виявлено, що використання жорстких затримок та ручного опитування баз даних призводить до появи нестабільних тестів, створює надмірну зв'язність тестового коду з фізичною інфраструктурою та унеможливлює повноцінну автоматизацію процесів безперервної доставки (Continuous Delivery).
2. **Запропоновано методологію Trace-Based Testing (семантичної валідації).** Обґрунтовано перехід від тестування через виклики API до тестування через аналіз телеметрії. Запропонована методологія базується на використанні єдиного машиночитаного потоку подій (JSON Lines), де кожна бізнес-операція однозначно ідентифікується завдяки впровадженню міжнародного стандарту наскрізного трасування W3C Trace Context. Це дозволило абстрагувати тестові сценарії від мережевих протоколів та перевіряти систему як «чорний ящик» за фактом виконання семантичних бізнес-правил.
3. **Розроблено архітектуру та реалізовано бібліотеку структурованого логування (tlog).** Створено фреймворк-незалежне ядро **tlog** та модуль автоконфігурації **spring-boot-starter-tlog** для екосистеми Java/Spring Boot. Бібліотека діє як прозора надбудова над стандартом SLF4J і забезпечує сувору типізацію бізнес-подій. Завдяки поєднанню з індустріальним

стандартом OpenTelemetry реалізовано безшовне перехоплення та розповсюдження ідентифікаторів транзакцій (**traceId**, **spanId**) через гетерогенні середовища: синхронні HTTP-виклики, бінарний протокол gRPC та метадані брокера повідомлень Apache Kafka.

4. **Забезпечено високу продуктивність та безпеку конкурентного доступу.** Для усунення проблеми «пляшкового горлечка» під час генерації розширених аудит-логів, розроблено механізм асинхронного неблокуючого запису. Використання виділеного пулу потоків (*ExecutorService*) та системних можливостей атомарного додавання дозволило декільком незалежним мікросервісам безпечно і без конфліктів записувати телеметрію у спільний файл у середовищі Docker, не впливаючи на пропускну здатність основної бізнес-логіки.
5. **Спроєктовано модуль автоматизованої валідації (TraceAsserter).** Створено незалежний інструмент для проведення тестування, який усуває проблему нестабільності асинхронних перевірок. Використання патерну Builder дозволило створити інтуїтивно зрозумілий декларативний інтерфейс для написання тестів. Реалізація механізму розумного відкладеного опитування (Polling) із відкладеним потоковим читанням (Lazy Stream Processing) файлу логів гарантує оптимальне використання оперативної пам'яті та миттєве завершення тесту одразу після досягнення системою кінцевої узгодженості (Eventual Consistency).
6. **Проведено успішне практичне застосування в гібридному середовищі.** Розроблений фреймворк було інтегровано у спеціально створений розподілений тестовий стенд на базі Docker Compose, що імітує реальну корпоративну систему. Інтеграція підтвердила, що застосування **TraceAsserter** дозволяє суттєво скоротити обсяг тестового коду, усунути необхідність написання інфраструктурних Mock-компонентів для брокерів повідомлень і гарантувати стовідсоткове покриття як зовнішніх, так і прихованих внутрішніх викликів. Забезпечено миттєву локалізацію

першопричин збоїв, що робить інструмент готовим до інтеграції у CI/CD-конвеєри як надійний механізм блокування дефектних релізів.

Практичне значення отриманих результатів полягає у тому, що створений програмний комплекс є завершеним, готовим до використання інструментарієм, який може бути інтегрований у будь-який сучасний Java-проєкт. Розроблений фреймворк становить особливу цінність для фінтех-продуктів, платформ електронної комерції, великих логістичних систем та будь-яких рішень із високим ступенем розподіленості, де цілісність наскрізних бізнес-процесів (таких як обробка транзакцій чи замовлень) є критичною, а тривале ручне тестування стає бар'єром для швидкого випуску нових релізів.

Перспективи подальших досліджень у цьому напрямку охоплюють розширення розробленого інструментарію на інші мови програмування (наприклад, Go, Python, Node.js) для забезпечення наскрізного тестування різномовних мікросервісних систем. Крім того, перспективною є розробка графічного інтерфейсу для візуального конструювання тестових сценаріїв та зручного аналізу результатів валідації, додавання нативної підтримки інших популярних брокерів повідомлень (таких як RabbitMQ чи Redis), а також створення власного легковагового агента для децентралізованого збору та агрегації аудит-логів, що дозволить ще легше масштабувати фреймворк у складних кластерних інфраструктурах.

Таким чином, мету магістерської роботи досягнуто повною мірою, а всі поставлені завдання успішно виконано. Розроблена методологія Trace-Based Testing та програмний комплекс для автоматизованої валідації розподілених запитів довели свою високу ефективність, стабільність роботи з асинхронними процесами та повну готовність до практичного застосування у сучасних конвеєрах безперервної доставки (CI/CD) промислових систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. W3C Trace Context, https://www.w3.org/TR/trace-context/
2. JSONL, https://jsonlines.org/
3. Test Automation Pyramid, https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid
4. Three pillars of observability, https://www.ibm.com/think/insights/observability-pillars
5. Dapper, a Large-Scale Distributed Systems Tracing Infrastructureб https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/
6. X-B3-TraceId, https://http.dev/x-b3-traceid
7. X-Amzn-Trace-Id, https://http.dev/x-amzn-trace-id
8. X-Request-ID, https://http.dev/x-request-id
9. SLF4J, https://www.slf4j.org/manual.html
10. Log4j Asynchronous loggers, https://logging.apache.org/log4j/2.x/manual/async.html
11. Elastic Stack, https://www.elastic.co/docs/get-started/the-stack
12. Splunk Architecture, https://www.edureka.co/blog/splunk-architecture/
13. Datadog Architecture, https://docs.datadoghq.com/cloudprem/introduction/architecture/
14. Jaeger Architecture, https://www.jaegertracing.io/docs/2.17/architecture/
15. Zipkin Architecture, https://zipkin.io/pages/architecture.html
16. Context propagation in Opentelemetry, https://opentelemetry.io/docs/concepts/context-propagation/