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

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ЖУРНАЛЮВАННЯ ТА ОБРОБКА ТЕЛЕМЕТРІЇ ЗАЛЕЖНО ВІД КЛАСУ ПРОГРАМНИХ СИСТЕМ

Виконав: студент групи 1П-22
Спеціальності
121 Інженерія програмного забезпечення
Ігор НІКОНОВ
Керівник:
Станіслав МАРЧЕНКО

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 СТУДЕНТУ

Ніконову Ігору Ростиславовичу

1. Тема кваліфікаційної роботи Журналювання та обробка телеметрії залежно від класу програмних систем

Керівник роботи Марченко Станіслав Віталійович, викладач першої 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: мова програмування Python; вебфреймворк FastAPI; локальна база даних SQLite; фреймворк автоматизованого тестування pytest; HTML, CSS і JavaScript для реалізації вебдашборда; PlantUML для побудови UML- та C4-діаграм; Git/GitHub для керування версіями програмного коду.

4. Зміст кваліфікаційної роботи: теоретичні основи журналювання та обробки телеметрії (характеристика предметної області; інструменти журналювання та обробки телеметрії; засоби програмної обробки та інтеграції спостережуваних даних; постановка задачі на розроблення); проєктування та реалізація програмного забезпечення (інженерна постановка вимог та профілі телеметрії; попередня архітектура програмного продукту; модель даних, синтетичне забезпечення та сценарії генерації телеметрії; реалізація серверного застосунку, конвеєра та політик обробки); тестування та верифікація програмного забезпечення (вибір стратегій тестування та критерії якості; формування тест-

плану та тестових даних; організація тестування, приклади запуску та результати; аналіз результатів, обмеження й напрями посилення верифікації)

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. Теоретичні основи журналювання та обробки телеметрії	09.12.2025	
3	Розділ 2. Проєктування та реалізація програмного забезпечення	10.03.2026	
4	Розділ 3. Тестування та верифікація програмного забезпечення	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 _____ Ігор НІКОНОВ
(підпис)

Керівник роботи _____ Станіслав МАРЧЕНКО
(підпис)

АНОТАЦІЯ

Кваліфікаційну роботу присвячено дослідженню підходів до журналювання, обробки та візуального подання телеметричних даних у програмних системах різних класів. У роботі розглянуто відмінності між телеметрією вбудованих, IoT/edge, корпоративних і хмарно-орієнтованих систем, зокрема з позиції ресурсних обмежень, мережевої доступності, критичності подій, потреби в кореляційному контексті та вимог до безпеки даних.

У теоретичній частині проаналізовано сучасні засоби журналювання, збору метрик, трасування, централізованого зберігання подій і формування спостережуваності. Обґрунтовано, що ефективна робота з телеметрією потребує не лише накопичення журналів, а й нормалізації, фільтрації, маскування чутливих атрибутів, збагачення контекстом і подальшого аналітичного використання.

Практичним результатом роботи є програмний прототип системи обробки телеметрії. Він містить серверний API, генератор синтетичних подій, профільні політики, конвеєр обробки, SQLite-сховище, модуль обчислення метрик, Prometheus-сумісний експорт, генерацію графіків і вебдашборд. Прототип демонструє зміну стратегії обробки залежно від профілю програмної системи та властивостей джерела сигналу.

Для перевірки працездатності розробленого програмного забезпечення сформовано тест-план і реалізовано набір автоматизованих pytest-перевірок. Тестування охоплює API-рівень, моделі даних, профільні політики, адаптацію джерел, конвеєр обробки, сховище, синтетичні сценарії, метрики, графіки, Prometheus-експорт і консольні команди. Додатковим засобом аналізу результатів є дашборд із графіками, що відображають обсяг подій, розподіл критичності, частку відкидання, затримки, кореляційний статус і адаптивні політики.

Ключові слова: ЖУРНАЛЮВАННЯ, ТЕЛЕМЕТРІЯ, СПОСТЕРЕЖУВАНІСТЬ, МЕТРИКИ, ДАШБОРД, ПРОГРАМНІ СИСТЕМИ.

ABSTRACT

The qualification work focuses on approaches to logging, telemetry processing, and visual representation of telemetry data in different classes of software systems. The study considers the differences between telemetry in embedded, IoT/edge, enterprise, and cloud-oriented systems, taking into account resource constraints, network availability, event criticality, correlation context, and data security requirements.

The theoretical part analyzes modern tools and practices for logging, metric collection, tracing, centralized event storage, and observability. The work substantiates that effective telemetry processing is not limited to storing log records. It also requires normalization, filtering, sensitive data masking, contextual enrichment, and further analytical use of the collected data.

The practical result of the work is a software prototype for telemetry processing. It includes a server API, a synthetic event generator, profile-based policies, a telemetry processing pipeline, SQLite storage, a metrics calculation module, Prometheus-compatible export, chart generation, and a web dashboard. The prototype demonstrates how the processing strategy changes depending on the software system profile and the properties of the signal source.

To verify the developed software, a test plan was prepared and a set of automated pytest checks was implemented. The testing covers the API layer, data models, profile policies, source adaptation, processing pipeline, storage, synthetic scenarios, metrics, charts, Prometheus export, and command-line operations. The dashboard with visual charts serves as an additional analysis tool, showing event volume, severity distribution, drop rate, latency, correlation status, and adaptive policies.

Keywords: LOGGING, TELEMETRY, OBSERVABILITY, METRICS, DASHBOARD, SOFTWARE SYSTEMS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

AIOps	Artificial Intelligence for IT Operations — використання методів штучного інтелекту для аналізу IT-операцій, виявлення аномалій, групування інцидентів і підтримки експлуатації програмних систем.
CSV	Comma-Separated Values — текстовий формат подання табличних даних, у якому значення розділяються комами або іншими роздільниками.
Edge	Периферійний рівень обчислень, розташований ближче до джерела даних; використовується для попередньої обробки, фільтрації та зменшення затримок передавання.
endpoint	Окрема адреса або маршрут API, через який виконується конкретна операція: приймання події, запуск симуляції, отримання метрик або графіків.
FastAPI	Вебфреймворк мови Python, використаний для реалізації серверного API програмного прототипу.
IoT/edge	Поєднання пристроїв інтернету речей і периферійних обчислень; у роботі використовується як профіль систем із нестабільним зв’язком, локальним контекстом і потребою у фільтрації даних.
Loki	Система централізованого зберігання та пошуку журналів, орієнтована на роботу з мітками та інтеграцію з Grafana.
OpenSearch	Платформа пошуку, індексації та аналізу журналів і подій; може використовуватися як зовнішнє сховище спостережуваних даних.
OpenTele-metry	Відкрита специфікація та набір засобів для формування, збирання, обробки й експорту телеметрії: журналів, метрик і трас.
OTLP	OpenTelemetry Protocol – протокол передавання телеметричних даних у межах екосистеми OpenTelemetry.
payload	Корисне навантаження події; набір основних даних або атрибутів, що передаються разом із телеметричним записом.

p50	50-й перцентиль затримки; значення, нижче якого перебуває половина виміряних затримок обробки.
p95	95-й перцентиль затримки; показник, що характеризує верхню межу затримок для більшості подій і допомагає оцінити граничну поведінку системи.
p99	99-й перцентиль затримки; показник для аналізу рідкісних, але важливих випадків підвищеної затримки.
Prometheus	Система збирання та зберігання метрик у вигляді часових рядів; у роботі підтримується Prometheus-сумісний текстовий експорт.
pytest	Фреймворк автоматизованого тестування для Python; використовується для перевірки API, конвеєра обробки, метрик, графіків і CLI-сценаріїв.
SIEM	Security Information and Event Management – клас систем для централізованого збирання, аналізу та кореляції подій інформаційної безпеки.
SQLite	Локальна вбудована реляційна база даних; використовується у прототипі для зберігання нормалізованих телеметричних подій.
trace	Траса виконання; послідовність пов’язаних операцій або викликів, що дозволяє простежити шлях запиту через компоненти системи.
span	Окрема ділянка виконання всередині траси; описує конкретну операцію, її тривалість, атрибути та зв’язок з іншими операціями.

ВСТУП

Актуальність теми зумовлена зростанням складності сучасних програмних систем і підвищенням вимог до контролю їхнього фактичного стану під час експлуатації. Програмне забезпечення дедалі частіше працює не як ізольований застосунок, а як сукупність взаємопов’язаних компонентів, сервісів, пристроїв, мережевих вузлів, баз даних, черг повідомлень і зовнішніх інтеграцій. У таких умовах звичайного фіксування повідомлень про помилки недостатньо: для виявлення причин збоїв, оцінювання продуктивності, контролю безпеки та аналізу якості роботи системи потрібні структуровані телеметричні дані.

Особливу складність становить те, що різні класи програмних систем потребують різних підходів до журналювання та обробки телеметрії. У вбудованих системах головними обмеженнями є пам’ять, енергоспоживання, обчислювальні ресурси й час реакції. В IoT- та edge-середовищах важливими стають нестабільність зв’язку, локальне накопичення даних і вибіркове передавання подій. У корпоративних системах телеметрія пов’язана з аудитом, безпекою, контролем доступу та збереженням контексту користувацьких дій. У хмарно-орієнтованих і розподілених системах ключового значення набувають кореляція подій, метрик і трасувань, контроль обсягу даних, вартість зберігання та можливість швидкого аналізу інцидентів.

Через це універсальна стратегія журналювання для всіх типів систем є недостатньо ефективною. Надмірне збирання даних може створювати зайве навантаження, збільшувати витрати на зберігання й ускладнювати пошук важливих подій. Недостатня деталізація, навпаки, призводить до втрати діагностичного контексту, неможливості відтворити послідовність дій і складності виявлення причин відмов. Додатковою проблемою є ризик потрапляння чутливих даних у журнали, зокрема токенів, паролів, службових ключів, персональних або конфігураційних відомостей. Отже, актуальним є розроблення підходу, який поєднує нормалізацію, фільтрацію, збагачення, маскування, зберігання, обчислення метрик і візуальне подання телеметрії з урахуванням класу програмної системи.

Об’єктом розроблення є процес збирання, структурування, передавання та аналізу спостережуваних даних у програмних системах.

Предметом розроблення виступають програмні засоби нормалізації, фільтрації, збагачення, зберігання й візуального подання телеметричних даних з урахуванням ресурсних, часових, мережевих та експлуатаційних обмежень.

Метою розроблення є створення модульного програмного прототипу системи журналювання та обробки телеметрії, який забезпечує приймання подій із різних типів джерел, приведення їх до єдиної структури, фільтрацію, збагачення контекстом, збереження, базовий аналіз і демонстрацію відмінностей між стратегіями телеметрії для програмних систем різних класів. Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати предметну область журналювання, телеметрії та спостережуваності програмних систем, визначити особливості формування й обробки телеметричних даних для вбудованих, IoT/edge, корпоративних і хмарно-орієнтованих систем;

– розглянути сучасні інструменти журналювання, збирання метрик, трасування, централізованого зберігання, візуалізації та аналізу спостережуваних даних;

– сформулювати вимоги до програмного прототипу системи журналювання та обробки телеметрії;

– спроєктувати архітектуру програмного забезпечення, структуру даних, сценарії генерації подій і логіку обробки телеметрії;

– реалізувати серверний API, генератор синтетичних подій, конвеєр нормалізації, фільтрації, маскування й збагачення телеметричних записів;

– розробити тест-план і перевірити працездатність програмного прототипу за допомогою автоматизованих тестів і контрольних сценаріїв, проаналізувати результати тестування, визначити обмеження реалізації та напрями подальшого розвитку системи.

Практична цінність розроблення полягає в можливості моделювати поведінку різних класів програмних систем без підключення реальних промислових джерел телеметрії. Завдяки генератору синтетичних подій можна відтворювати контрольовані сценарії для вбудованих, IoT/edge, корпоративних і хмарно-орієнтованих профілів, аналізувати частку прийнятих і відкинутих подій, перевіряти покриття кореляційним контекстом, оцінювати затримки обробки та виявляти вплив профільних політик на якість телеметричного потоку.

Результати роботи можуть бути використані під час навчання з дисциплін, пов’язаних з інженерією програмного забезпечення, архітектурою програмних систем, тестуванням, DevOps-практиками, моніторингом і спостережуваністю. Крім того, запропонована структура може бути розширена для підключення реальних джерел журналів, інтеграції з OpenTelemetry Collector, Prometheus, Loki, OpenSearch або Grafana, а також для впровадження складніших методів аналізу подій і виявлення аномалій.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ЖУРНАЛЮВАННЯ ТА ОБРОБКИ ТЕЛЕМЕТРІЇ

1.1 Характеристика предметної області

Журналювання та обробка телеметрії в сучасних програмних системах не можуть розглядатися як універсальна допоміжна функція, однаково придатна для будь-якого середовища виконання. Характер таких даних визначається апаратною платформою, архітектурою програмного забезпечення, критичністю функцій, режимом мережевої взаємодії, вимогами до безпеки та способом подальшого використання зібраної інформації. У малоресурсному пристрої один додатковий запис може впливати на часову поведінку або ресурс пам’яті, тоді як у хмарній платформі головною проблемою стає не дефіцит окремого вузла, а надмірний обсяг подій, вартість зберігання та складність встановлення причинно-наслідкових зв’язків між компонентами. Тому предметна область цієї роботи охоплює не лише засоби запису повідомлень, а ширшу систему інженерних рішень щодо формування, передавання, збереження, нормалізації, кореляції та аналізу спостережуваних даних.

У межах життєвого циклу програмного забезпечення журнали й телеметрія виконують кілька взаємопов’язаних ролей. Під час розроблення вони підтримують пошук дефектів, перевірку припущень розробника та аналіз граничних станів. На етапі тестування такі дані використовуються для відтворення помилок, оцінювання продуктивності, порівняння версій і перевірки поведінки системи під навантаженням. Після впровадження вони стають підставою для технічної підтримки, розслідування інцидентів, аудиту, контролю безпеки та планування модернізації. Такий погляд узгоджується з ISO/IEC/IEEE 12207:2017, де програмне забезпечення описується через процеси створення, експлуатації, супроводу та вдосконалення [1]. Архітектурний аспект також має принципове значення: стандарт ISO/IEC/IEEE 42010:2022 пов’язує опис архітектури з зацікавленими сторонами, їхніми занепокоєннями, архітектурними поглядами та моделями [2]. Для телеметрії це означає, що одна й та сама подія

може мати різну цінність для розробника, адміністратора, фахівця з безпеки, оператора обладнання або керівника сервісу.

Журналювання у вузькому значенні охоплює фіксацію подій, попереджень, помилок, змін стану, дій користувачів і службових повідомлень. Телеметрія має ширший зміст: вона включає журнальні записи, числові метрики, траси виконання, події інфраструктури, сенсорні дані, повідомлення безпеки, аварійні знімки стану та службові сигнали середовища виконання. У сучасних розподілених системах ці категорії дедалі частіше об’єднуються поняттям спостережуваності, тобто здатності робити висновки про внутрішній стан системи за зовнішніми сигналами. Специфікація OpenTelemetry описує стандартизований підхід до формування, оброблення та передавання телеметрії, а оглядові матеріали проєкту виділяють траси, метрики й журнали як базові сигнали спостереження [3; 4]. Водночас наявність великої кількості сигналів ще не гарантує розуміння поведінки системи: між збором даних і корисним інженерним висновком існує складний ланцюг обробки.

Наукові дослідження підтверджують, що проблема журналювання не обмежується вибором бібліотеки або формату повідомлення. Автори [5] у систематичному дослідженні проаналізували праці, присвячені моніторингу програмного забезпечення на основі журналів, і показали, що журнали та дані середовища виконання залишаються важливим джерелом для виявлення небажаної поведінки, але перетворення великої кількості записів на практично корисні висновки залишається складною задачею. Kratzke [6], аналізуючи хмарно-орієнтовані системи, звертає увагу на фрагментацію традиційних каналів спостереження – журналів, метрик і трасувань – та обґрунтовує переваги структурованого й уніфікованого журналювання. Отже, предметна область має подвійний характер: з одного боку, потрібно зафіксувати достатню кількість фактів про роботу системи, з іншого – не створити некерований потік даних, який важко зберігати, захищати й інтерпретувати.

Першим ключовим аспектом аналізу є ресурсні обмеження. У мікроконтролерних і вбудованих системах телеметрія конкурує з основною

програмною логікою за оперативну пам’ять, постійну пам’ять, обчислювальний час, енергію та пропускну здатність інтерфейсів. Повні текстові журнали в такому середовищі часто є неприйнятними, оскільки вони займають багато місця, потребують форматування рядків, створюють операції введення-виведення та можуть прискорювати знос енергонезалежної пам’яті. Замість них використовують короткі коди подій, лічильники, кільцеві буфери, бітові маски станів, аварійні знімки пам’яті або передавання діагностичних даних через спеціальні канали налагодження. Тому цінність телеметрії у малоресурсній системі визначається не обсягом повідомлень, а здатністю компактно зберегти контекст помилки.

Другою віссю є часові обмеження. Для систем реального часу якість роботи залежить не лише від правильності обчислень, а й від своєчасності реакції. Якщо програмний компонент повинен обробити подію протягом обмеженого проміжку часу, то журналювання не може блокувати критичний шлях виконання. Небезпека полягає в тому, що службовий запис, який у звичайній інформаційній системі сприймається як безпечний, у керувальному контурі може спричинити затримку, втрату події або порушення послідовності дій. Через це в таких системах застосовують попередньо виділені буфери, асинхронне накопичення, обмеження частоти записів, розподіл подій за критичністю та заборону повільних операцій у критичних секціях. Журнал тут має бути передбачуваним за тривалістю операцій, а не лише інформативним.

Третій фактор пов’язаний з мережевою доступністю. Пристрої інтернету речей, автономні сенсорні вузли та мобільні клієнти часто працюють у режимі нестабільного зв’язку. Для них телеметрія охоплює не лише помилки застосунку, а й рівень заряду, якість сигналу, час останньої синхронізації, стан локальної черги, версію прошивки, температуру, параметри сенсорів і мережеві відмови. У таких умовах повне передавання всіх подій може бути технічно або економічно невиправданим. Потрібні локальне накопичення, стискання, пакетне передавання, подієва передача, вибіркова деталізація й повторне надсилання після відновлення зв’язку. Дослідження Log2Evt [7] показує, що для систем

інтернету речей перспективним є перехід від низькорівневих журналів до високорівневих подій шляхом кореляції журналів із контекстом виконання прошивки та стеком викликів. Це демонструє зсув від простого збереження рядків до побудови змістовнішої подієвої картини.

Четверта вісь – зв’язок програмної поведінки з фізичним процесом. У кіберфізичних, робототехнічних, медичних, транспортних і промислових системах програмні події не можна інтерпретувати без даних про стан об’єкта керування. Помилка алгоритму, затримка команди або збій сенсора можуть мати фізичні наслідки, тому журнальний запис повинен пов’язуватися з часом, режимом роботи, показниками датчиків, діями оператора та станом виконавчих механізмів. NIST SP 800-82 [8] визначає операційну технологію як програмовані системи й пристрої, що взаємодіють із фізичним середовищем або керують такими пристроями, і підкреслює їхні особливі вимоги до продуктивності, надійності та безпеки. Для цієї групи систем телеметрія має не лише діагностичне, а й безпекове значення: вона допомагає виявляти відхилення, небажані команди, зміну режимів, аварійні стани та можливі кіберінциденти.

П’ятий аспект стосується розподіленості та багаторівневої архітектури. Периферійні обчислення переносять частину обробки ближче до джерела даних, зменшуючи затримку й обсяг передавання до центру. Проте такий підхід ускладнює спостереження за системою: події виникають на пристроях, периферійних вузлах, мережевих проміжних рівнях і центральних сервісах. Сучасний огляд моніторингу та спостережуваності периферійних обчислювальних систем підкреслює наявність компромісів між накладними витратами, масштабованістю, мобільністю та повнотою телеметрії [9]. Через це для периферійних систем потрібні узгоджені часові мітки, локальна фільтрація, нормалізація форматів, збереження контексту подій, повторна доставка й механізми відновлення повної картини після розривів мережі.

Шостий фактор пов’язаний з корпоративними та хмарно-орієнтованими системами. Тут обмеження окремого пристрою поступаються місцем проблемам масштабу, різнорідності й вартості. Один користувацький запит може пройти

через шлюз, службу автентифікації, кілька прикладних сервісів, базу даних, кеш, чергу повідомлень і зовнішній програмний інтерфейс. Якщо кожен компонент формує події окремо, але не передає спільний ідентифікатор запиту, то розслідування інциденту перетворюється на зіставлення фрагментів без гарантованого причинного зв'язку. Тому хмарно-орієнтовані системи потребують не тільки централізованого збирання, а й кореляції журналів, метрик і трасувань через спільний контекст [4; 6].

Надмірність телеметрії є окремою проблемою. У великих системах журнали можуть накопичуватися швидше, ніж їх здатні аналізувати оператори або автоматизовані платформи. Частина записів є критичною, частина має тимчасову цінність, а частина створює інформаційний шум. Галузевий звіт Grafana Labs [10] показує, що організації використовують багато різнорідних засобів спостереження, а вартість, простота використання та взаємодія з наявними інструментами належать до важливих критеріїв вибору рішень. Це підтверджує, що в хмарних і корпоративних середовищах основною інженерною задачею стає не максимальне накопичення даних, а керування їхньою цінністю: фільтрація, вибіркове зберігання, зменшення шуму, агрегування та збереження тих сигналів, які справді потрібні для пояснення поведінки системи.

Сьома вісь – безпека та конфіденційність. Журнали можуть містити імена користувачів, мережеві адреси, параметри запитів, фрагменти персональних даних, службові токени, конфігураційні відомості або дані про режими роботи обладнання. У промисловому середовищі вони можуть розкривати структуру об'єкта, технологічні параметри або слабкі місця мережевої сегментації. У хмарному середовищі централізоване сховище телеметрії стає привабливою ціллю для зловмисників. Тому журналювання повинно передбачати маскування чутливих полів, контроль доступу, шифрування каналів передавання, розмежування ролей, політики строків зберігання й перевірку того, що діагностична користь не створює неприйнятного ризику витоку інформації.

Восьмий аспект стосується аналітичної обробки. Традиційно журнали переглядалися після відмови, коли користувач уже повідомив про проблему або

система зупинилася. Сучасна практика рухається до проактивного аналізу: виявлення зростання затримок, накопичення черг, нетипових послідовностей подій, аномальної активності, деградації продуктивності або прихованих помилок конфігурації. Систематичний огляд AIOps [11] для виявлення аномалій у журналах у добу великих мовних моделей наголошує, що сучасні IT-системи генерують великі обсяги журнальних даних, а традиційні методи часто потребують значної підготовки ознак і важко пристосовуються до динамічних середовищ. Звідси випливає потреба в нормалізації, шаблонізації, виділенні подій, часовому аналізі, виявленні відхилень і поєднанні правил з методами статистики та машинного навчання.

Загальну логіку предметної області можна подати як систему взаємозалежних чинників, де апаратне середовище, обмеження, критичність, типи спостережуваних даних, способи обробки й інженерні суперечності утворюють єдину модель (рис. 1.1). Така модель показує, що клас програмної системи впливає не лише на вибір інструменту журналювання, а й на рівень деталізації, місце попередньої обробки, політику зберігання, структуру подій і спосіб подальшої аналітики. Вона також пояснює, чому рішення, прийнятне для серверного застосунку, може бути неприйнятним для контролера або автономного сенсорного вузла. Порівняльне узагальнення основних інженерних суперечностей подано в табл. 1.1.

Таблиця 1.1 – Інженерні компроміси журналювання та обробки телеметрії

Інженерна суперечність	Найбільш характерні середовища	Наслідок для організації телеметрії
Повнота даних ↔ обсяг пам’яті	мікроконтролери, вбудовані системи	короткі коди подій, кільцеві буфери, обмеження рівнів деталізації
Деталізація ↔ час реакції	системи реального часу, кіберфізичні системи	асинхронний запис, попередньо виділені буфери, відсутність блокування критичних задач
Передавання всіх подій ↔ енергоощадність	інтернет речей, автономні пристрої	локальна агрегація, пакетне передавання, зміна деталізації залежно від режиму

Продовження таблиці 1.1

Централізація ↔ автономність	периферійні та розподілені системи	локальна обробка, буферизація, повторна доставка після відновлення зв'язку
Інформативність ↔ безпека процесу	промислові OT/ICS-системи	пасивний збір, мінімальне навантаження, пріоритет безперервності фізичного процесу
Повнота спостереження ↔ вартість	корпоративні та хмарно-орієнтовані системи	фільтрація, вибіркове зберігання, керування строками збереження
Діагностика ↔ конфіденційність	мобільні, корпоративні, промислові та хмарні системи	маскування чутливих полів, розмежування доступу, контроль строків зберігання

Рисунок 1.1 – Ментальна модель залежності журналювання та телеметрії від класу програмної системи

Телеметрія формується не на одному рівні системи (рис. 1.2). Нижче прикладного коду існують фізичні сигнали, апаратні події, події операційного середовища, мережеві взаємодії та службові стани інфраструктури. Вище прикладного рівня розташовані засоби спостереження, що збирають журнали, метрики, трасування й події безпеки, а ще вище – аналітичні механізми пошуку, кореляції, сповіщення та виявлення відхилень. Якщо ці рівні ізольовані, система має багато даних, але мало пояснювальної сили. Якщо вони пов'язані спільним

часом, контекстом, ідентифікаторами операцій і структурованими схемами подій, телеметрія стає основою для розслідування складних інцидентів.

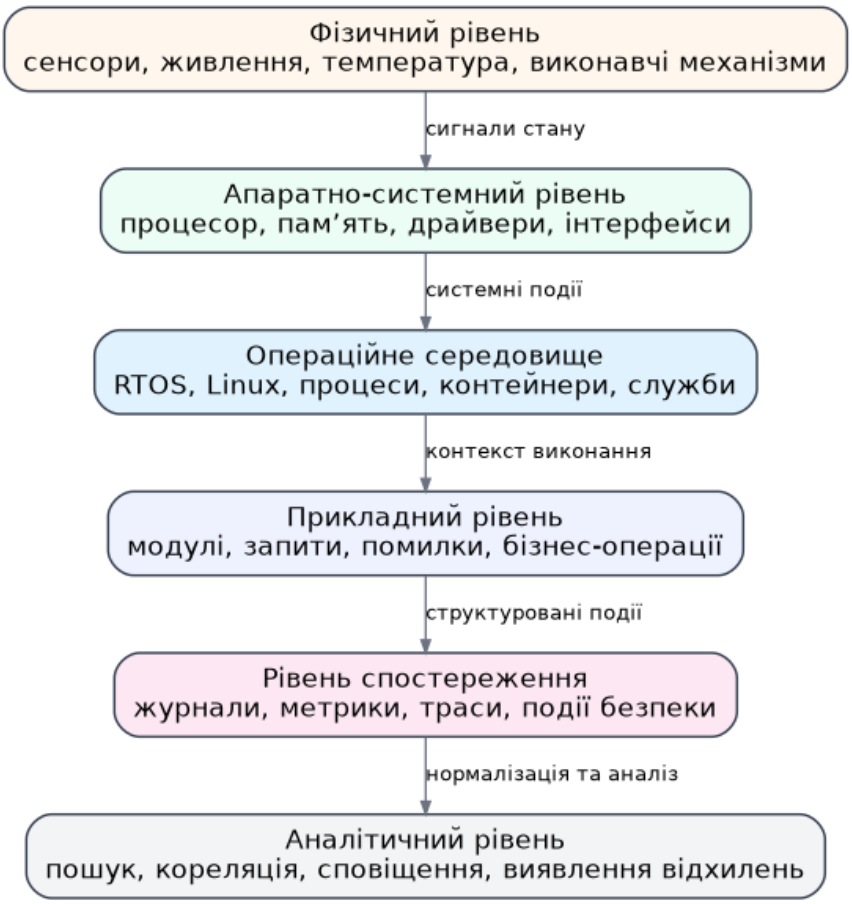

Рисунок 1.2 – Рівні формування, збирання та аналітичної обробки телеметрії

Критичний аналіз предметної області показує, що розроблення системи журналювання починається з визначення цільової моделі спостереження. Для малоресурсного пристрою такою моделлю може бути мінімальна локальна телеметрія з аварійними знімками стану. Для системи інтернету речей – подієво орієнтоване передавання з локальним накопиченням і зміною деталізації. Для промислової системи – контрольоване спостереження, що не порушує фізичний процес. Для периферійної інфраструктури – попередня обробка та узгодження подій між пристроями й центральними сервісами. Для корпоративної або хмарної системи – централізована спостережуваність із кореляцією журналів, метрик і трасувань.

Звідси, журналювання та обробка телеметрії залежать від апаратного середовища, архітектури, критичності, мережевих умов, вимог до безпеки й очікуваного способу аналізу. Універсальне рішення, яке з однаковою ефективністю працює для всіх класів систем, є малоймовірним. Раціональнішим є підхід, за якого для кожного класу системи визначається припустимий обсяг спостереження, місце попередньої обробки, рівень деталізації, формат подій, політика зберігання та набір аналітичних операцій. Така постановка створює основу для подальшого огляду інструментів журналювання й обробки телеметрії, а також для формулювання вимог до прототипу, який демонструватиме різні стратегії роботи зі спостережуваними даними залежно від класу програмної системи.

1.2 Інструменти журналювання та обробки телеметрії

Інструменти журналювання та обробки телеметрії не утворюють однорідного класу програмних засобів. Частина з них працює всередині прикладного коду й відповідає за формування записів; частина діє на рівні операційної системи, контейнера або вузла; окремі засоби виконують приймання, фільтрацію, нормалізацію, перетворення та маршрутизацію даних; інші забезпечують зберігання, пошук, побудову панелей стану, сповіщення або виявлення відхилень. Через це інструментарій треба оцінювати за місцем у телеметричному ланцюгу, типом сигналів, які він підтримує, впливом на систему, вимогами до ресурсів і здатністю зберігати причинно-наслідковий контекст подій [3; 4; 15].

Перший рівень такого ланцюга становлять засоби інструментування програмного коду. Вони формують записи журналів, метрики, події й трасування безпосередньо в місці виникнення програмної дії. На цьому рівні визначається, які події будуть видимими для подальшого аналізу, які атрибути отримає запис, чи буде в ньому ідентифікатор запиту, користувача, пристрою, сесії або версії компонента. Помилка проєктування на цьому етапі майже не компенсується

подальшими інструментами: сховище може швидко шукати записи, проте воно не відновить контекст, який не був зафіксований у момент створення події [5;13].

Класичні бібліотеки журналювання залишаються потрібними, оскільки більшість систем починає спостереження з прикладних повідомлень. Їхня сильна сторона криється в простоті впровадження, підтримці рівнів важливості, можливості виводити записи у файл, консоль або зовнішній приймач. Проблема полягає в тому, що неструктуровані текстові повідомлення швидко втрачають аналітичну цінність у великих системах. Повідомлення, написане для читання людиною, погано піддається групуванню, автоматичній класифікації та зіставленню з метриками або трасами. Тому сучасна практика переходить до структурованого журналювання, де запис має сталі поля: час, джерело, рівень, код події, компонент, середовище, ідентифікатор операції, повідомлення й набір доменних атрибутів [6; 20].

Структуроване журналювання також має обмеження. Якщо структура визначається стихійно, різні команди починають використовувати різні назви полів, різні формати часу, несумісні рівні важливості та неоднакове трактування помилок. У результаті з’являється формальна структурованість без семантичної узгодженості. Для подолання цієї проблеми застосовують спільні схеми подій. Наприклад, Elastic Common Schema [25] задає узгоджений набір полів для зберігання подієвих даних, зокрема журналів і метрик, що полегшує пошук, фільтрацію й повторне використання правил обробки. У системах, орієнтованих на відкриті стандарти, подібну роль відіграють семантичні угоди OpenTelemetry, які допомагають однаково описувати типові атрибути сервісів, запитів, помилок і ресурсів [3; 4].

Другий рівень інструментарію формують агенти збирання та пересилання даних. Вони читають файли журналів, системні журнали, події контейнерів, повідомлення з мережевих сокетів, метрики процесів або потоки подій і передають їх до подальших компонентів. Для цього класу засобів критичними є мале споживання ресурсів, стійкість до перебоїв, підтримка локального буфера, повторна доставка, фільтрація на вузлі та здатність працювати в неоднорідних

середовищах. Fluent Bit [18] позиціонується як швидкий і легкий агент для журналів, метрик і трас із наголосом на продуктивність і використання в різних операційних системах. Vector [19] описується як високопродуктивний потік обробки спостережуваних даних, який збирає, перетворює та маршрутизує журнали, метрики й траси до різних приймачів.

Агенти можуть змінювати якість даних: відкидати записи, додавати атрибути середовища, змінювати часові мітки, групувати події, виконувати перетворення форматів або маскувати чутливі поля. Це створює важливий інженерний компроміс. З одного боку, обробка на вузлі зменшує трафік і вартість зберігання. З іншого боку, надто агресивна фільтрація може знищити деталі, потрібні для розслідування складного інциденту. У вбудованих, мобільних та периферійних системах ця проблема особливо помітна, бо локальний буфер обмежений, а зв’язок може бути нестабільним [7; 9; 18].

Третій рівень пов’язаний із колекторами та проміжними потоками обробки. OpenTelemetry Collector є показовим прикладом такого підходу: він приймає телеметрію, обробляє її й експортує до різних сховищ або платформ, використовуючи приймачі, обробники, експортери та з’єднувачі [16; 17]. Його значення полягає не тільки в підтримці кількох протоколів. Колектор відокремлює прикладне інструментування від конкретного постачальника сховища або платформи аналізу (рис. 1.3). Це знижує ризик технологічної залежності й дає змогу змінювати маршрути, правила фільтрації та політики зберігання без масового переписування прикладного коду [15; 16].

Попри переваги, колекторна модель має власні ризики. Додавання проміжного шару збільшує складність експлуатації, потребує моніторингу самого колектора, налаштування черг, обмеження пам’яті, політик повторної відправки й захисту каналів передавання. Якщо колектор стає єдиною точкою приймання телеметрії, його відмова може призвести до втрати службових даних або до затримки виявлення інцидентів. У великих середовищах колектори розгортають ієрархічно: локальні екземпляри працюють на вузлах або в кластерах, а центральні рівні виконують агрегування й маршрутизацію. Така

архітектура підходить для хмарних та периферійних систем, проте потребує чіткої політики буферизації й обмеження обсягів [16; 17; 23].

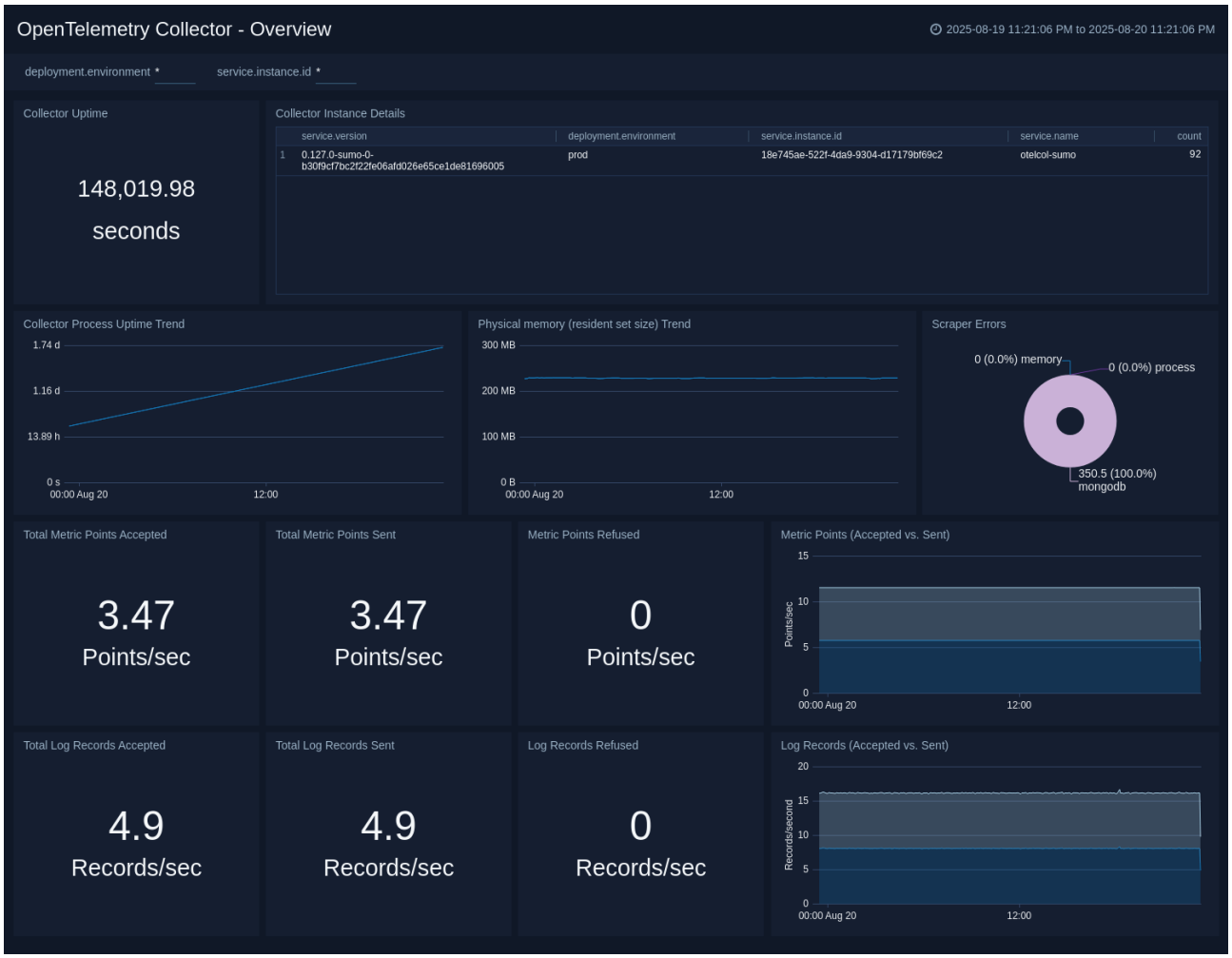

Рисунок 1.3 – Моніторинг колектора даних

Четвертий напрям становлять системи збирання й аналізу метрик. Метрика відрізняється від журнального запису тим, що вона призначена для кількісного спостереження за станом системи в часі. Prometheus зберігає метрики як часові ряди, де кожне значення має часову мітку й набір міток-атрибутів [21]. Такий підхід добре підходить для контролю навантаження, затримок, кількості помилок, використання пам’яті, стану черг, доступності сервісів і побудови сповіщень. У порівнянні з журналами метрики компактніші, краще агрегуються й швидше дають відповідь на питання про загальний стан системи.

Водночас метрики не замінюють журнали. Вони відображають, що показник змінився, але не завжди пояснюють причину. Висока затримка може бути наслідком перевантаження бази даних, помилки мережі, блокування черги, невдалої зміни конфігурації або залежності від зовнішнього сервісу. Крім того, метрики чутливі до проблеми високої кардинальності: якщо в мітки потрапляють ідентифікатори користувачів, пристроїв, запитів або транзакцій, кількість часових рядів різко зростає. Це підвищує вимоги до пам’яті, індексації та обчислень. Тому системи метрик потребують дисципліни в проєктуванні назв і міток не меншою мірою, ніж журнали потребують дисципліни в проєктуванні структури запису [21; 22].

П’ятий напрям охоплює системи централізованого зберігання та пошуку журналів. Рішення цього класу можна умовно поділити за тим, що вони індексують: повний вміст записів або переважно метадані. Grafana Loki побудований навколо ідеї індексації міток, тоді як самі журнальні дані стискаються й зберігаються блоками. Це зменшує вартість індексації та зближує модель журналів із моделлю Prometheus, але потребує обережного вибору міток. Якщо мітки надто загальні, пошук стає менш точним; якщо вони надто деталізовані, система стикається з проблемою високої кардинальності [22].

Платформи на основі Elasticsearch або OpenSearch, навпаки, орієнтовані на потужний пошук, індексацію, аналітику та взаємодію з великими масивами подій. Elastic Observability описується як засіб об’єднання журналів, метрик, трасувань, даних користувацького досвіду та інших сигналів в інтегрованій платформі [23]. OpenSearch також розвиває можливості аналізу журналів, метрик і трас у межах робочих просторів спостереження [24]. Перевага таких підходів полягає в гнучкому пошуку та багатих аналітичних можливостях, проте плата за це – складніша інфраструктура, вищі вимоги до сховища, індексації, керування схемами та безпеки доступу.

Шостий напрям становить розподілене трасування. Якщо журнали описують події, а метрики – агрегований стан, то траси відтворюють шлях операції через компоненти системи. Для мікросервісних і хмарно-орієнтованих

архітектур це критично, бо користувацький запит може проходити через багато сервісів, черг і зовнішніх залежностей. W3C Trace Context стандартизує передавання ідентичності траси через заголовки traceparent і tracestate, що дає різним системам змогу зберігати зв’язок між частинами розподіленої операції [26]. Jaeger [27] описує розподілене трасування як засіб простеження запитів між сервісами для виявлення затримок, помилок і вузьких місць.

Трасування також має обмеження. Повний запис усіх трасувань у високонавантаженій системі може бути надмірно дорогим, тому часто застосовують вибіркове трасування. Проте вибірка створює ризик втрати рідкісних, але важливих інцидентів. Складні асинхронні сценарії, черги повідомлень, фонові задачі та події пристроїв можуть розривати контекст, якщо він не передається послідовно через усі межі системи. Тому трасування не є автоматичним розв’язанням проблеми причинності. Воно потребує дисципліни передачі контексту, узгоджених бібліотек, єдиних правил і перевірки того, що контекст не губиться на транспортних або організаційних межах [26; 27].

Ще один напрям пов’язаний із виявленням відхилень, інтелектуальною обробкою журналів та AIOps. Через зростання обсягу телеметрії ручний перегляд записів стає непридатним для великих систем. Систематичний огляд [11] показує, що сучасні IT-системи генерують великі обсяги журналів, які ускладнюють своєчасне виявлення відхилень, а традиційні методи часто потребують трудомісткого виділення ознак і гірше пристосовуються до динамічних середовищ. У цьому напрямі застосовують кластеризацію подій, пошук шаблонів, статистичні методи, моделі послідовностей, автоенкодери, мовні моделі та методи пояснення інцидентів.

Критичний аспект інтелектуальної обробки полягає в тому, що якість результатів залежить від якості телеметрії. Якщо журнали неструктуровані, часові мітки неточні, контекст відсутній, а різні компоненти по-різному називають однакові події, модель виявлення відхилень працює з шумними ознаками. У таких умовах штучний інтелект не усуває проблеми проєктування телеметрії, а лише переносить її на етап навчання й інтерпретації результатів.

Через це автоматичне виявлення аномалій повинно поєднуватися з нормалізацією подій, контролем схем, перевіркою якості даних і людською валідацією висновків [7; 11].

Окрему групу становлять інструменти для малоресурсних і вбудованих систем. Вони не завжди мають вигляд універсальних платформ. Часто це спеціалізовані кільцеві буфери, бінарні формати подій, аварійні дампи, апаратні канали трасування, журнали завантажувача або обмежені телеметричні повідомлення, що передаються через послідовний інтерфейс, радіоканал чи службовий протокол. Їхня цінність полягає в здатності працювати без порушення часових меж і без значного споживання пам’яті. Для таких систем централізовані платформи можуть використовуватися лише після вивантаження або агрегування даних на шлюзі, периферійному вузлі чи стенді випробувань [7; 9].

У промислових та операційно-технологічних системах інструменти журналювання мають узгоджуватися з вимогами безперервності й безпеки. NIST SP 800-82 [8] наголошує, що операційні технології мають особливі вимоги до продуктивності, надійності та захисту, оскільки взаємодіють із фізичним середовищем. Через це активне опитування пристроїв, глибоке сканування або інтенсивний збір подій можуть бути небажаними. У таких середовищах перевагу часто мають пасивний збір мережевих подій, контроль змін конфігурації, фіксація команд оператора, журналювання аварійних станів і обмежена передача даних до центрів моніторингу.

Для корпоративних систем особливу роль відіграють інструменти безпекового моніторингу, аудиту та керування інцидентами. Журнали тут містять не тільки технічні помилки, а й сліди доступу до даних, зміни прав, автентифікаційні події, операції з фінансовими або персональними відомостями. Інструмент повинен підтримувати контроль доступу, незмінність або захищеність записів, політики зберігання, маскування чутливих полів і пошук за атрибутами. Парадокс полягає в тому, що журнали, створені для підвищення безпеки, самі можуть стати джерелом витоку, якщо містять токени, паролі, надмірні параметри запитів або персональні дані без захисту.

Вибір інструментів залежить від того, який тип сигналу вважається провідним. Якщо головна мета – оперативно бачити стан сервісу, першочерговими є метрики, часові ряди та сповіщення. Якщо потрібно відновлювати конкретні інциденти, центральну роль відіграють структуровані журнали й пошук. Якщо система є розподіленою, важливими стають траси та передавання контексту. Якщо джерелом є фізичне середовище або пристрій, потрібні буферизація, локальна агрегація й захист від втрати подій. У зрілих системах ці підходи поєднуються в багаторівневу модель спостереження [3-4; 6].

Загальну архітектуру інструментів журналювання та обробки телеметрії можна подати як послідовність взаємопов'язаних рівнів (рис. 1.4): джерела даних, інструментування, агенти, колектори, потік обробки, сховища й аналітичні засоби. Така схема не означає, що кожна система повинна мати всі рівні. Для мікроконтролера достатнім може бути локальний буфер і періодичне вивантаження. Для IoT-рішення потрібні шлюз, буферизація та обмеження трафіку. Для хмарної платформи потрібні колектори, сховища часових рядів, пошукові індекси, трасування й панелі аналізу. Різниця полягає не в назвах інструментів, а в конфігурації телеметричного ланцюга.

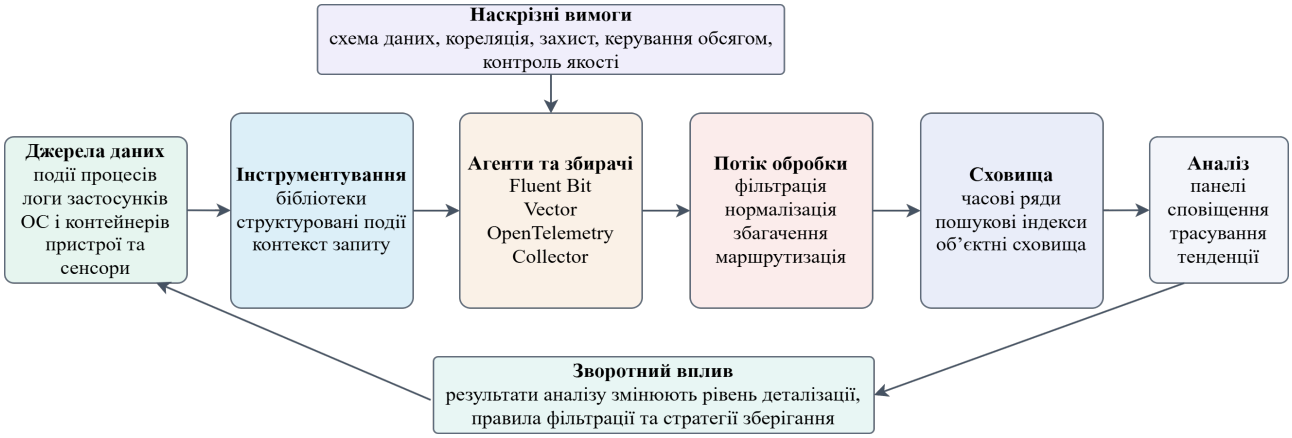

Рисунок 1.4 – Узагальнена архітектура інструментів журналювання та обробки телеметрії

Порівняння в табл. 1.2 показує, що жоден клас інструментів не покриває всю проблематику. Бібліотека журналювання не замінює сховище, сховище не

замінює коректне інструментування, а система трасування не усуває потреби в метриках.

Таблиця 1.2 – Порівняння основних класів інструментів журналювання та телеметрії

Клас інструментів	Основна функція	Сильні сторони	Обмеження та ризики
Бібліотеки журналювання	Формування подій у прикладному коді	простота впровадження; контроль рівнів; близькість до доменної логіки	ризик неструктурованості; різні формати між командами; можливе потрапляння чутливих даних
Схеми та моделі подій	Уніфікація полів і семантики записів	полегшують пошук, кореляцію та повторне використання правил	потребують дисципліни проєктування; надмірна схема може бути важкою для малих систем
Агенти й пересилачі	Збирання даних із вузлів, файлів, контейнерів і служб	працюють без зміни прикладного коду; підтримують буферизацію та фільтрацію	можуть втрачати дані при переповненні; фільтрація здатна знищити контекст
Колектори та проміжні потоки	Приймання, обробка й маршрутизація телеметрії	зменшують залежність від постачальника; дозволяють централізувати правила	ускладнюють експлуатацію; самі потребують моніторингу й резервування
Системи метрик	Зберігання числових показників у часі	ефективні для стану, тенденцій і сповіщень	не пояснюють причину без журналів і трас; ризик високої кардинальності
Сховища журналів	Пошук і аналіз подієвих записів	дають детальний контекст інциденту; підтримують аудит	вартість індексації; шум; складність політик зберігання
Системи трасування	Відтворення шляху операції між сервісами	виявляють затримки й вузькі місця в розподілених системах	потребують передачі контексту; вибірка може приховати рідкісні помилки
Інструменти AIOps	Автоматичне виявлення відхилень і групування інцидентів	зменшують ручну роботу; працюють із великими обсягами даних	залежать від якості даних; складність пояснення результатів; ризик хибних спрацьовувань

Помилка багатьох проєктів полягає в намаганні розв’язати організаційно-архітектурну проблему одним засобом. Наприклад, упровадження централізованого сховища журналів без уніфікації полів лише переносить хаос з окремих файлів у спільний індекс. Упровадження трасування без передачі контексту через черги й фонові задачі створює фрагменти трас замість повної картини. Масове збирання метрик без контролю кардинальності призводить до технічного й фінансового перевантаження.

Для прототипу, що розробляється в межах кваліфікаційної роботи, важливо відтворити принципову архітектуру їхньої взаємодії. Мінімальна демонстраційна модель повинна показувати формування структурованої події, додавання контексту, локальну буферизацію, передавання до проміжного обробника, фільтрацію або нормалізацію, збереження та аналітичне використання. Додатково потрібна можливість змінювати поведінку телеметрії залежно від класу системи: для малоресурсного режиму – зменшувати деталізацію; для мережево нестабільного режиму – накопичувати дані; для хмарного режиму – дод авати кореляційні поля й імітувати обробку значного потоку.

Отже, інструменти журналювання та телеметрії треба розглядати як багаторівневу інженерну екосистему. Її нижній рівень відповідає за народження якісної події, середній – за доставку, очищення, нормалізацію й маршрутизацію, верхній – за зберігання, пошук, кореляцію, візуалізацію та виявлення відхилень. Практична проблема полягає в узгодженні цих рівнів із класом програмної системи. Для одних систем найбільш цінною є мінімальна й надійна подія, для інших – повний розподілений контекст, для третіх – доказовість і захищеність журналу, для четвертих – керування витратами на зберігання й обробку. Тому подальший розгляд програмної обробки спостережуваних даних має перейти від огляду інструментів до аналізу способів їх інтеграції, нормалізації та використання в єдиному інформаційному контурі системи.

1.3 Засоби програмної обробки та інтеграції спостережуваних даних

Засоби програмної обробки та інтеграції спостережуваних даних формують проміжний шар між джерелами телеметрії та засобами її аналітичного використання. Якщо інструменти журналювання відповідають за народження повідомлення, метрики або трасування, то програмна обробка визначає, чи перетвориться цей первинний сигнал на придатну для діагностики, аудиту й управлінського рішення інформацію. Первинний запис без контексту має обмежену цінність: він може показати факт помилки, але не пояснити її зв’язок із версією застосунку, вузлом виконання, користувацькою операцією, станом пристрою, зміною конфігурації або зовнішньою залежністю. Через це інтеграція спостережуваних даних охоплює інструментування, приймання, фільтрацію, нормалізацію, збагачення, маршрутизацію, кореляцію, зберігання, захист і подальшу аналітичну інтерпретацію.

У сучасних програмних системах телеметрія рідко надходить з одного джерела. Вона формується у прикладному коді, бібліотеках, операційній системі, контейнерному середовищі, базах даних, чергах повідомлень, мережевих вузлах, промислових контролерах, сенсорах, шлюзах інтернету речей і системах безпеки. Ці дані відрізняються частотою появи, структурою, точністю часових міток, рівнем критичності, строком зберігання та придатністю до автоматизованого аналізу. Специфікація OpenTelemetry [3] описує уніфіковані вимоги до телеметрії, що генерується, збирається й експортується різними мовами програмування та середовищами виконання. Така стандартизація важлива, бо без неї система спостереження швидко перетворюється на сукупність несумісних журналів і метрик.

Для систем різних апаратних класів місце обробки телеметрії змінюється. У мікроконтролерних пристроях обробка обмежується короткими кодами подій, лічильниками, аварійними знімками та локальним буфером. В інтернеті речей частина логіки переноситься на шлюз або периферійний вузол, де можна агрегувати вимірювання, відкидати повтори, стискати дані й додавати контекст. У промислових системах обробка повинна не втручатися в технологічний контур

і не створювати додаткового ризику для безперервності процесу [8]. У хмарних системах більше можливостей для централізованого аналізу, але зростають обсяг, вартість і складність кореляції. Отже, програмна обробка не може мати єдину універсальну топологію для всіх класів систем.

Загальну логіку програмної обробки можна подати як конвеєр (рис. 1.5), у якому дані проходять від джерел до аналітичного використання, а наскрізне керування якістю контролює схеми, кореляцію, втрати, захист і строки зберігання. Така схема не означає жорсткої лінійності: в реальних системах окремі етапи можуть виконуватися локально, на периферійному вузлі, у хмарі або в кількох місцях одночасно. Проте конвеєрна модель допомагає побачити, що кожен етап додає цінність і водночас створює нові ризики.

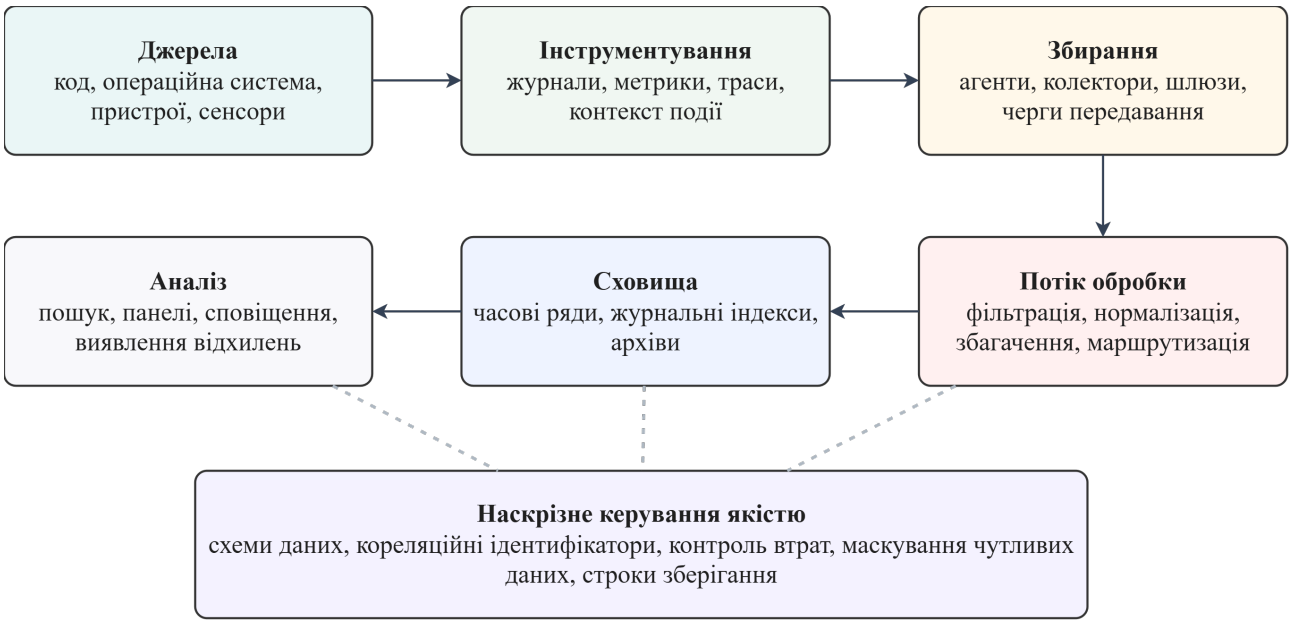

Рисунок 1.5 – Конвеєр програмної обробки та інтеграції спостережуваних даних

Рисунок 1.5 підкреслює, що обробка телеметрії має не тільки технічний, а й керівний вимір. Якщо джерела даних не узгоджені зі схемою, інструментування не має предметного контексту, збирачі не контролюють втрати, а сховища не мають політик строків зберігання, аналітичний рівень отримує неповну або викривлену картину. Зворотний зв’язок також важливий:

результати аналізу інцидентів повинні змінювати правила фільтрації, рівні деталізації, перелік обов’язкових атрибутів і строки збереження даних. Без такого зворотного зв’язку система спостереження залишається статичною, тоді як сама програмна система постійно змінюється.

Порівняння основних етапів програмної обробки подано в таблиці 1.3. Вона показує, що кожен етап має окреме призначення та власний клас ризиків. Помилка на ранньому етапі часто переноситься далі й посилюється: некоректне інструментування породжує неповні записи, неповні записи погано нормалізуються, відсутність кореляційних ідентифікаторів унеможливлює трасування, а погано обрані мітки збільшують вартість зберігання метрик.

Таблиця 1.3 – Етапи програмної обробки телеметрії

Етап	Призначення	Критичні ризики	Контрольні вимоги
Інструментування	Формування журналів, метрик, трас і предметного контексту.	Формальні записи без діагностичної цінності; різні назви полів.	Спільна схема подій; мінімальний набір обов’язкових атрибутів.
Приймання та передавання	Збір даних з вузлів, контейнерів, пристроїв і сервісів.	Втрата записів; перевантаження агента; помилки експорту.	Моніторинг черг, затримок, відкинутих записів і повторних спроб.
Фільтрація	Зменшення шуму, обсягу та вартості подальшої обробки.	Приховування слабких сигналів відмов; втрата рідкісних аварійних подій.	Правила за критичністю, режимом роботи та строком цінності даних.
Нормалізація	Приведення різних записів до узгодженої структури.	Втрата предметної специфіки або змішування несумісних значень.	Семантичні угоди, внутрішній словник полів, перевірка схеми.
Збагачення	Додавання середовища, версії, вузла, ролі компонента та контексту.	Зростання обсягу; потрапляння чутливих даних у телеметрію.	Маскування, обмеження атрибутів, контроль доступу.
Маршрутизація	Спрямування потоків до сховищ, систем сповіщення або архівів.	Неправильний приймач; дублювання; втрата даних через конфігурацію.	Версіонування і тестування правил конвеєра.
Кореляція	Зв’язування журналів, метрик, трас, запитів, пристроїв і сеансів.	Фрагментарна картина інциденту; ручне зіставлення подій.	Ідентифікатори трас, запитів, пристроїв, узгоджені часові мітки.
Зберігання та аналітика	Пошук, панелі, сповіщення, аналіз відхилень і першопричин.	Висока вартість; хибні спрацювання; залежність від якості даних.	Політики строків зберігання, контроль якості даних, перевірність висновків.

Також видно, що програмна обробка телеметрії змінює зміст, ціну й доступність спостережуваних сигналів. Фільтрація може як зменшити шум, так і приховати ранні ознаки відмови. Збагачення може як підвищити пояснювальну цінність запису, так і створити ризик витоку даних. Кореляція може як відновити причинний ланцюг інциденту, так і виявитися неможливою через відсутність ідентифікаторів на етапі інструментування. Тому програмний конвеєр телеметрії потребує проєктування, тестування й експлуатаційного контролю так само, як інші критичні компоненти системи.

Отже, засоби програмної обробки та інтеграції спостережуваних даних виконують роль архітектурного мосту між подією в системі та інженерним висновком про її стан. Їхня ефективність визначається не кількістю зібраних повідомлень, а здатністю зберегти контекст, зменшити шум, забезпечити кореляцію, захистити чутливі дані та надати основу для перевірного аналізу. Для систем різного апаратного класу цей міст має різну конфігурацію, але його базові етапи залишаються спільними: формування, приймання, обробка, зберігання, аналіз і зворотне вдосконалення політик телеметрії.

1.4 Постановка задачі на розроблення

Проведений огляд показав, що журналювання та обробка телеметрії не зводяться до вибору бібліотеки запису повідомлень або централізованого сховища. У різних класах програмних систем змінюються цілі спостереження, допустимий обсяг службових даних, вимоги до затримки, режими мережевої взаємодії, строк зберігання та рівень критичності подій. Для малоресурсних пристроїв першочерговими є компактність і мінімальний вплив на виконання прикладної логіки. Для систем інтернету речей – стійкість до втрати зв’язку, локальна буферизація та передавання лише значущих подій. Для корпоративних і хмарних платформ головними стають кореляція журналів, метрик і трас, контроль вартості зберігання, керування надмірністю та забезпечення аудиторської цінності даних.

З урахуванням цього, практична частина роботи має бути спрямована на створення програмного прототипу, який демонструє адаптивну обробку телеметрії залежно від класу програмної системи. Прототип не повинен імітувати промислову платформу повного циклу спостережуваності. Його призначення полягає в експериментальному відтворенні основних інженерних режимів: мінімалістичного журналювання, контекстного збагачення подій, фільтрації надлишкових записів, буферизації під час нестабільного зв'язку, збереження для аудиту та порівняння обсягу телеметрії між профілями систем.

Для прототипу, що розробляється в межах кваліфікаційної роботи, з попереднього випливає кілька вимог. По-перше, прототип має демонструвати не лише запис повідомлень, а повний шлях даних від джерела до аналітичного подання. По-друге, він має підтримувати різні режими деталізації для систем різного класу: мінімальний режим для малоресурсних пристроїв, контекстний режим для кіберфізичних і промислових систем, централізований режим для серверних і хмарних застосунків. По-третє, прототип має показувати різницю між необробленими й нормалізованими даними, а також наслідки фільтрації, збагачення та кореляції. По-четверте, потрібно передбачити контроль якості телеметрії: відсутні поля, затримки, дублікати, втрати, чутливі значення та записи без ідентифікаторів. Попередні функціональні вимоги зібрано в табл. 1.4.

Таблиця 1.4 – Попередні функціональні вимоги до прототипу

№	Вимога	Зміст
FR-1	Профілі систем	Вибір класу системи та пов'язаних правил збору телеметрії.
FR-2	Приймання подій	Отримання подій з генераторів, файлів або HTTP-запитів.
FR-3	Уніфікація структури	Перетворення різнорідних записів до спільної моделі.
FR-4	Фільтрація	Відкидання, зниження пріоритету або агрегування надлишкових подій.
FR-5	Збагачення	Додавання даних про середовище, джерело, профіль і кореляційний контекст.
FR-6	Збереження	Запис оброблених подій і результатів сценаріїв у локальне сховище.
FR-7	Аналітика	Розрахунок показників обсягу, критичності, втрат і якості телеметрії.
FR-8	Експорт	Формування файлу з результатами або короткого звіту.

У межах прототипу передбачено чотири профілі систем: малоресурсний або вбудований, інтернет речей або периферійний вузол, корпоративна інформаційна система, хмарна або розподілена система. Кожен профіль задає власні правила формування подій, рівні деталізації, правила відкидання або агрегування записів, набір контекстних полів і строк зберігання. Такий поділ дає змогу порівняти не інструменти як такі, а наслідки різних стратегій спостереження.

Очікуваним результатом є програмний прототип, який показує, що ефективність журналювання визначається відповідністю стратегії збору даних класу системи, критичності подій і способу подальшого аналізу. Обмеженнями роботи є локальний режим виконання, демонстраційний характер частини джерел подій, спрощена модель аналітики та відсутність інтеграції з промисловими сховищами повного циклу. Водночас архітектура має залишатися розширюваною: до неї можна додати реальні джерела журналів, зовнішні збирачі телеметрії, спеціалізовані сховища часових рядів і модулі виявлення відхилень.

Нефункціональні вимоги охоплюють модульність, розширюваність, відтворюваність демонстраційних сценаріїв, контрольовану затримку обробки, захист чутливих полів, прозорість правил фільтрації та технологічну простоту локального запуску. Ці вимоги важливі для навчального прототипу, оскільки він має не лише працювати, а й пояснювати логіку вибору стратегії телеметрії.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Інженерна постановка вимог та профілі телеметрії

Журналювання та телеметрія не є універсальним допоміжним механізмом, який однаково переноситься між мікроконтролерними, периферійними, корпоративними і хмарними системами. Практична частина роботи будується як відповідь на цю проблему: замість створення абстрактного журналу подій розробляється прототип, у якому стратегія обробки змінюється залежно від профілю системи. Такий підхід надає можливість перевірити як працездатність коду, так і коректність інженерної ідеї: одна й та ж первинна подія може бути прийнята, скорочена, збагачена, замаскована або відкинута залежно від умов експлуатації.

Прототип «Telemetry Observer Prototype» буде реалізовано як локальний веборієнтований застосунок мовою Python із серверною частиною FastAPI, моделями валідації Pydantic, вбудованим SQLite-сховищем, простим HTML/JavaScript-інтерфейсом та набором автоматизованих тестів. Фреймворк FastAPI використовується як сучасний інструмент створення прикладних програмних інтерфейсів на базі стандартних анотацій типів Python [31], а Pydantic – як механізм суворої перевірки структури вхідних даних і перетворення їх у типізовані об’єкти [32]. У межах кваліфікаційної роботи цей стек не потребує складної інфраструктури, але дозволяє показати ключові принципи промислових конвеєрів телеметрії.

Вимоги до системи було сформовано як набір інженерних властивостей. Прототип повинен приймати події різних класів, перетворювати їх у спільну модель, застосовувати профільні політики, захищати чутливі дані, зберігати результат і надавати аналітичне зведення.

На рівні предметної логіки виділено чотири профілі: embedded, iot_edge, enterprise та cloud. Профіль embedded моделює вбудовану або малоресурсну систему, де надмірне журналювання небажане через обмеження пам’яті, енергії

та часу реакції. Профіль iot_edge моделює периферійний або IoT-вузол, для якого критичними є нестабільний зв’язок, буферизація та локальне скорочення даних. Профіль enterprise відображає корпоративну інформаційну систему, де важливими є аудит, контроль доступу і наявність ідентифікатора операції. Профіль cloud описує розподілену систему, у якій основною умовою аналізу є кореляція подій між компонентами. Функціональні вимоги було розділено на вимоги до приймання подій, вимоги до конвеєра обробки, вимоги до сховища, вимоги до аналітики та вимоги до демонстраційного режиму (табл. 2.1). Нефункціональні вимоги (табл. 2.2) стосуються модульності, відтворюваності запуску, тестопридатності, простоти локального розгортання, безпечної обробки чутливих полів і пояснюваності причин відкидання подій.

Таблиця 2.1 – Функціональні вимоги до програмного прототипу

Код	Група вимог	Інженерний зміст	Критерій приймання
FR-1	Профілі систем	Система підтримує профілі embedded, iot_edge, enterprise, cloud із різними правилами обробки.	Користувач може запустити демонстрацію для кожного профілю.
FR-2	Приймання подій	API приймає одиничні події та пакет подій у JSON-форматі.	Валідний запит повертає нормалізовану подію зі статусом обробки.
FR-3	Нормалізація	Первинні записи перетворюються на єдину структуру NormalizedTelemetryEvent.	У результаті є event_id, UTC-час, profile, severity, payload_size.
FR-4	Профільна політика	Для кожного профілю застосовується поріг критичності, обмеження payload і правила кореляції.	Події нижче порога отримують статус dropped із поясненням.
FR-5	Маскування	Чутливі поля та шаблони в payload приховуються до запису в сховище.	token, password, card, email не зберігаються у відкритому вигляді.
FR-6	Зберігання	Оброблені записи зберігаються у SQLite разом зі статусом і причиною відкидання.	Після демонстрації /api/events повертає записи з бази даних.
FR-7	Фільтрація	Список подій можна обмежити за профілем, критичністю і статусом.	Запит /api/events?status=dropped повертає лише відкинуті події.
FR-8	Аналітика	Система формує summary за профілями, статусами, рівнями та якістю.	/api/summary повертає accepted_ratio і correlation_coverage.
FR-9	Демонстрації	Синтетичні сценарії імітують характерні події різних класів систем.	/api/demo/{profile} створює контрольований пакет подій.
FR-10	Відтворюва-ність	Генератор подій працює з фіксованим зерном для контрольованих тестів.	Повторний запуск формує однакову структуру тестового набору.

Особливістю постановки вимог є включення синтетичного забезпечення даними вже на етапі проєктування. Для навчального прототипу неможливо покладатися лише на випадкові ручні приклади, оскільки вони не покривають граничні стани: відсутність кореляційного ідентифікатора, надмірний payload, вкладені секрети, debug-події, різні часові формати, дублювання, помилки мережевого середовища. Тому вимоги до даних розглядаються як частина вимог до системи, а не як допоміжна дія перед тестуванням (табл. 2.3).

Таблиця 2.2 – Нефункціональні вимоги та способи їх перевірки

Код	Нефункц. вимога	Пояснення для проєкту	Перевірка
NFR-1	Модульність	API, схеми, профілі, конвеєр, сховище і симулятор винесено в окремі модулі.	Статичний перегляд структури проєкту й імпорт модулів у тестах.
NFR-2	Тестопридатність	Ключові функції не залежать від HTTP-шару та можуть перевірятися напряму.	Модульні тести pipeline.py і profiles.py.
NFR-3	Локальне розгортання	Система запускається без зовнішньої СУБД, брокера або хмарного сервісу.	Запуск uvicorn і створення SQLite-файлу.
NFR-4	Безпека журналів	Чутливі поля маскуються до збереження, а SQL-запити параметризовані.	Тести маскування token, email, card; перегляд storage.py.
NFR-5	Пояснюваність	Кожна відкинута подія має drop_reason.	Перевірка відповідей API і тестів enterprise/cloud без correlation_id.
NFR-6	Керованість обсягу	Повідомлення й payload обмежуються політикою профілю.	Тести truncation і payload key limit.
NFR-7	Простота супроводу	Профільні правила задаються як дані у POLICIES, а не розкидані по коду.	Додавання нового профілю не потребує зміни UI-логіки.
NFR-8	Спостережуваність прототипу	Система рахує не лише події, а й якість самої телеметрії.	summary.quality містить accepted_ratio, correlation_coverage, masked_payload_ratio.

Додатково вимоги до даних формалізовано через перелік мінімальних атрибутів, які повинні бути присутніми в кожній події. Такий перелік потрібний не лише для програмування, а й для перевірки якості телеметрії. Якщо подія не містить часу, джерела, профілю або рівня критичності, її складно інтерпретувати навіть тоді, коли саме повідомлення виглядає змістовним. Якщо в корпоративній

або хмарній події немає correlation_id, вона втрачає зв’язок із бізнес-операцією або трасою виконання. Якщо payload не обмежується і не маскується, журнал може стати каналом витоку службових або персональних даних.

Таблиця 2.3 – Мінімальна інформаційна модель телеметричної події

Атрибут	Обов’язковість	Призначення	Типові помилки даних
profile	обов’язковий	визначає політику обробки	невідомий профіль або неправильна назва
source	обов’язковий	ідентифікує модуль, пристрій або сервіс	порожнє поле, надто загальна назва
event_type	опційний із типовим значенням	класифікує лог, метрику, сенсорну подію або trace	плутанина між log і audit
severity	обов’язковий	визначає критичність і проходження порога	завищення або заниження рівня важливості
message	обов’язковий	людиночитний опис події	надто довгий текст або службові секрети
payload	обов’язковий як dict	зберігає структурований контекст	надлишкові ключі, вкладені секрети, великі списки
correlation_id	умовно обов’язковий	пов’язує події однієї операції	відсутній у enterprise/cloud-сценаріях
timestamp	опційний	фіксує час виникнення події	наївна часова мітка без timezone
device_id/service_name	контекстний	уточнює джерело в IoT або cloud	відсутність стабільної ідентифікації джерела

У вимогах свідомо закладено різницю між прийнятою і збереженою подією. У промислових системах відкинуті записи часто не зберігаються з міркувань економії. У програмному прототипі вони залишаються в базі разом із причиною відкидання, оскільки це дає змогу продемонструвати дію політики і провести тестування. Без збереження dropped-подій перевіряючий бачив би лише те, що події немає, але не міг би встановити, чи це результат правильної політики, помилки валідації або втрати даних.

2.2 Попередня архітектура програмного продукту

Архітектура прототипу побудована за принципом розділення відповідальностей. На верхньому рівні користувач працює з простою

вебсторінкою, яка виконує роль демонстраційного інтерфейсу. Серверний рівень реалізує маршрути API, перевірку запитів, виклик доменного конвеєра, запис у сховище та формування агрегованих відповідей. Доменний рівень містить профілі телеметрії, правила обробки та алгоритми маскування. Рівень даних відповідає за SQLite-сховище, індекси, список подій і зведення.

З архітектурного погляду прототип відображає спрощену, але інженерно корисну модель промислової системи спостережуваності. У промисловому середовищі замість локального HTML-інтерфейсу може бути окрема панель моніторингу, замість SQLite – спеціалізоване сховище журналів або часових рядів, а замість вбудованого генератора подій – реальні агенти Fluent Bit, Vector або OpenTelemetry Collector. Проте головний принцип залишається тим же: джерела не повинні напряму нав’язувати сховищу власну структуру даних; між ними має бути керований шар нормалізації та політик.

Для опису системи використано C4-логіку як спосіб ієрархічного подання контейнерів та UML 2.5.1 як формальну нотацію для компонентних, класових, послідовних і активнісних діаграм [36; 37]. У документі C4-стилізація не підміняє UML-нотацію, а застосовується лише як спосіб візуального групування контейнерів. Компонентна діаграма використовує коректні залежності та lollipop-нотацію інтерфейсів, діаграма послідовності показує синхронні виклики й повернення, а діаграма класів відображає логічні типи та зв’язки даних.

На рис. 2.1 показано контейнерний рівень системи. Він підкреслює, що прототип є локальним, але не монолітним за змістом: користувацький інтерфейс, серверний застосунок, синтетичні джерела, сховище і тестовий контур мають різні ролі. Рішення не потребує зовнішнього брокера повідомлень, проте структура коду залишає можливість замінити генератор подій на реальний приймач телеметрії.

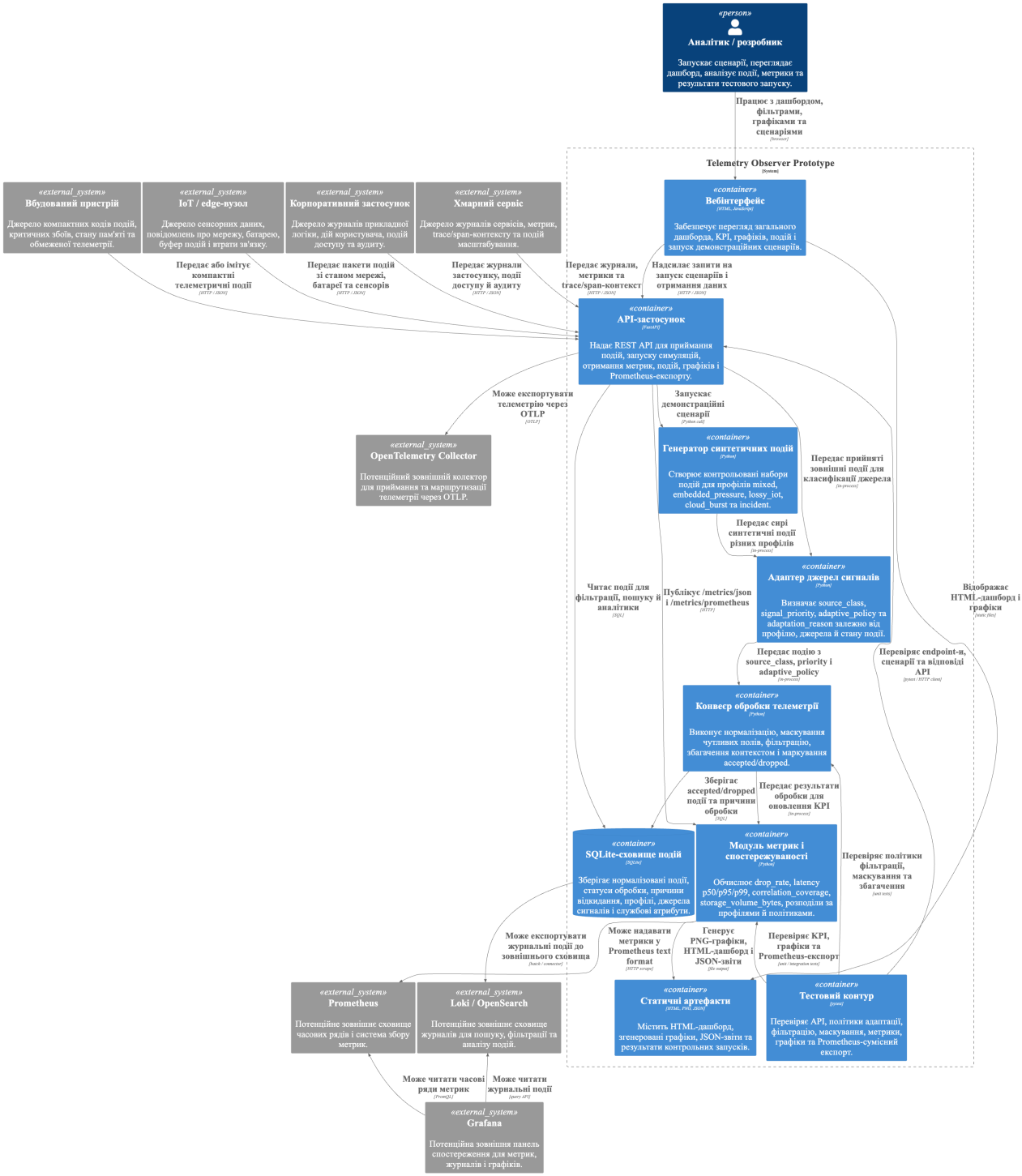

Рисунок 2.1 – C4-подання контейнерів прототипу системи

На рис. 2.2 подано послідовність взаємодії контейнерів під час запуску демонстраційного сценарію та обробки телеметричних подій. Діаграма відображає як приймання й збереження подій, так і адаптивний вибір політики залежно від джерела сигналу, нормалізацію, маскування чутливих даних,

фільтрацію, оновлення метрик, генерацію графіків і публікацію Prometheus-сумісного експорту.

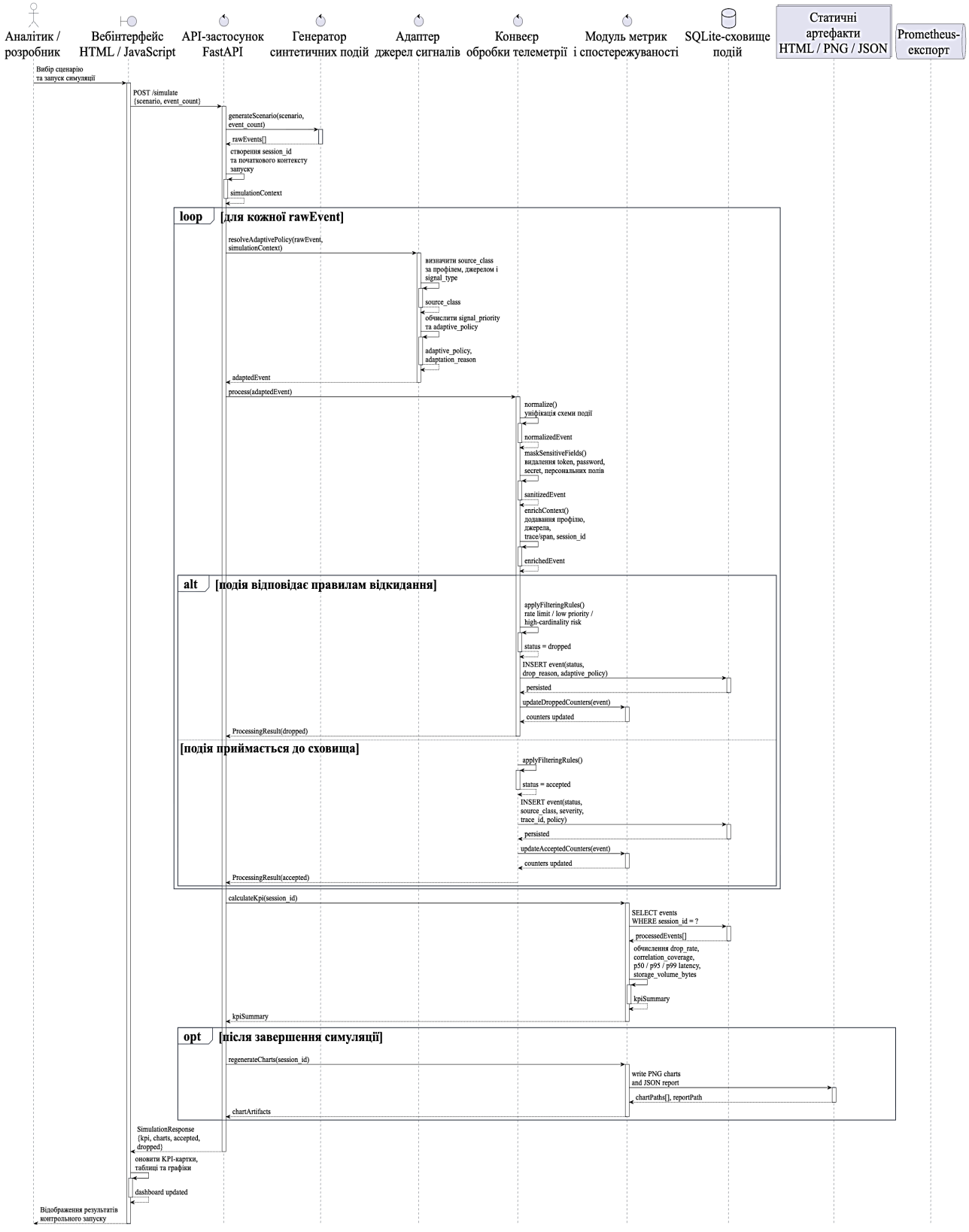

Рисунок 2.2 – UML-діаграма послідовності обробки телеметричної події

Схема демонструє те, що HTTP-запит не записується у сховище безпосередньо. Спочатку він проходить валідацію, потім нормалізацію, потім профільну політику, після чого результат зберігається разом із поясненням. Така організація потрібна для відтворюваності: якщо подія була відкинута, у базі зберігається причина, а не лише факт відсутності події в основному потоці.

На рис. 2.3 подано UML-діаграму пакетів програмного прототипу системи журналювання та обробки телеметрії. Діаграма відображає фактичну організацію програмного коду за каталогами, модулями та артефактами виконання. Центральним пакетом є app, у якому розміщено програмне ядро системи. Окремо виділено пакети static, data, charts, reports і службові файли проєкту.

Пакет static містить файли користувацького інтерфейсу. Файл index.html виконує роль стартової сторінки або переходу до основної панелі, а dashboard.html забезпечує перегляд дашборда, KPI, графіків, результатів симуляцій і збережених телеметричних подій. Цей пакет не містить логіки обробки телеметрії, а взаємодіє з програмним ядром через HTTP-запити до API, реалізованого в модулі app/main.py.

Пакет app є основним програмним пакетом прототипу. Модуль main.py створює FastAPI-застосунок і визначає endpoint-и для приймання телеметричних подій, перегляду збережених записів, запуску синтетичних сценаріїв, отримання метрик у JSON-форматі, формування Prometheus-сумісного експорту та повторної генерації графіків.

Модуль schemas.py містить узгоджені моделі даних, які використовуються іншими частинами програми. У ньому описуються початкові телеметричні події, нормалізовані події, результати проходження конвеєра, адаптивні рішення та знімки метрик. Це забезпечує єдину структуру даних для API, конвеєра обробки, сховища, метрик і графіків.

Модуль profiles.py задає профілі програмних систем і політики обробки телеметрії. У ньому визначаються правила для вбудованих, IoT/edge, корпоративних і хмарно-орієнтованих систем. Цей модуль пов’язує ідею

залежності телеметрії від класу програмної системи з конкретними параметрами програмної реалізації.

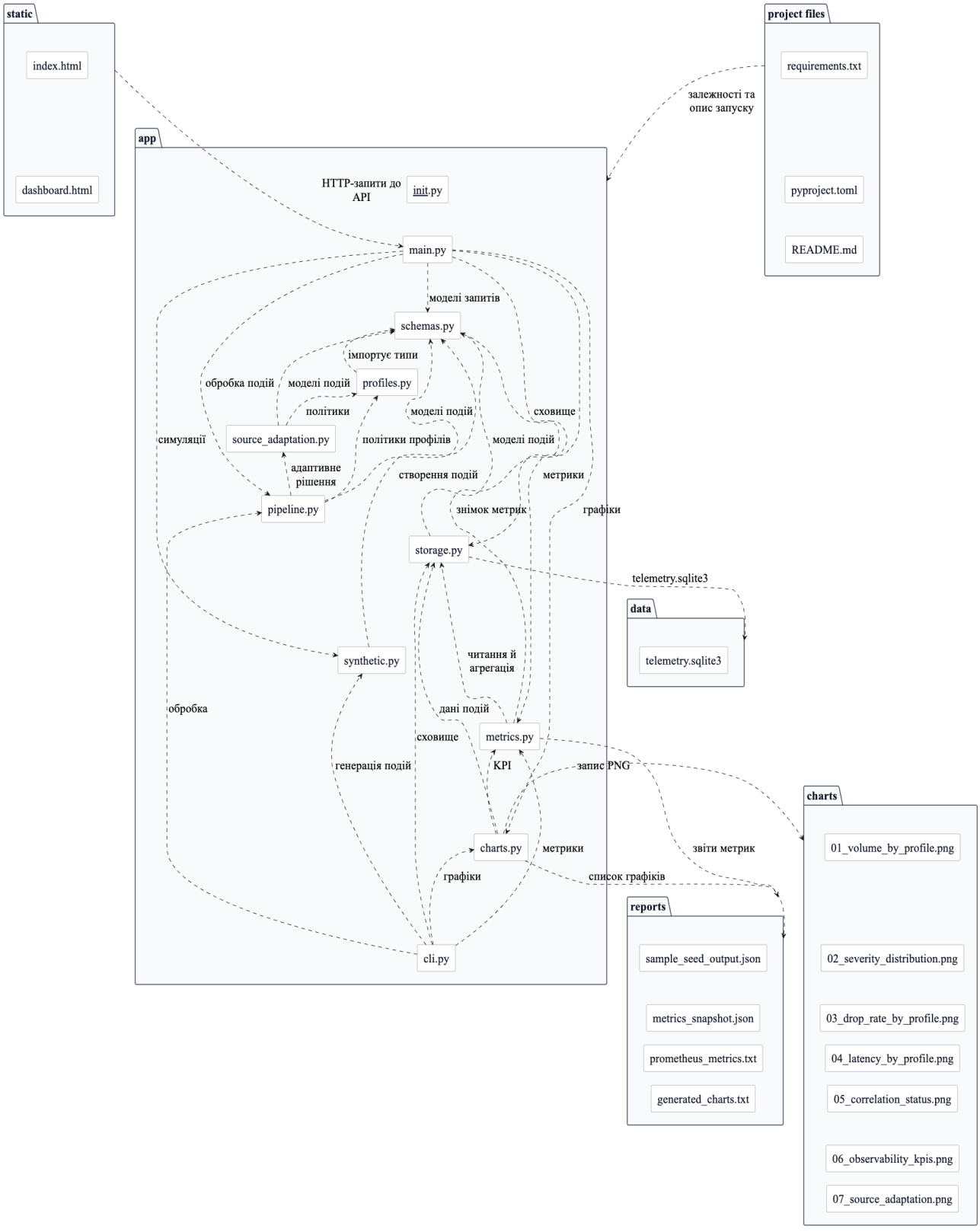

Рисунок 2.3 – Діаграма пакетів програмного прототипу

Модуль source_adaptation.py реалізує адаптацію під джерела сигналів. Він використовує моделі з schemas.py і політики з profiles.py, щоб визначати клас джерела, пріоритет сигналу, адаптивну політику та причину прийнятого рішення. У такий спосіб система враховує не лише зміст події, а й її походження, критичність, кореляційний контекст і умови роботи джерела.

Модуль pipeline.py реалізує основний конвеєр обробки телеметрії. Він відповідає за нормалізацію подій, маскування чутливих атрибутів, визначення кореляційного статусу, вибір рівня зберігання, фільтрацію та формування результату обробки. На цьому етапі події отримують статус прийнятих або відкинутих і службові ознаки для подальшого аналізу.

Модуль storage.py інкапсулює роботу з локальним SQLite-сховищем. Він забезпечує запис нормалізованих подій, читання даних, очищення сховища та агрегування записів для подальшого розрахунку метрик. Фізичним результатом його роботи є база telemetry.sqlite3, розміщена в каталозі data.

Модуль synthetic.py відповідає за формування синтетичних телеметричних подій. Він створює контрольовані набори даних для різних сценаріїв, зокрема змішаного навантаження, втрат зв’язку в IoT-середовищі, хмарного сплеску подій та інциденту. Це дозволяє перевіряти поведінку прототипу без підключення реальних зовнішніх джерел.

Модуль metrics.py реалізує обчислення показників спостережуваності. Він формує агреговані значення: кількість подій, частку відкидання, покриття кореляційним контекстом, затримки p50, p95 і p99, обсяг збережених даних, розподіли за профілями, критичністю, джерелами й адаптивними політиками. Також цей модуль формує текстовий експорт метрик у форматі, сумісному з Prometheus.

Модуль charts.py відповідає за побудову графічних артефактів. Він використовує дані зі сховища та агреговані метрики для формування PNG-графіків: обсягу подій за профілями, розподілу критичності, частки відкидання, затримок, кореляційного статусу, загальних KPI та адаптації під джерела

сигналів. Згенеровані файли зберігаються в каталозі charts і використовуються на дашборді та в експериментальному аналізі.

Модуль cli.py забезпечує консольний спосіб роботи з прототипом. Через нього можна запускати генерацію синтетичних подій, обробку, розрахунок метрик і побудову графіків без використання вебінтерфейсу. Це спрощує контрольні запуски, повторювані експерименти та автоматизоване відтворення результатів.

Каталог data містить локальне сховище telemetry.sqlite3, у якому зберігаються нормалізовані події, статуси їх обробки, профілі, джерела сигналів, причини відкидання та службові атрибути. Каталог charts містить PNG-графіки, згенеровані модулем app.charts. Каталог reports містить результати контрольних запусків, знімки метрик, Prometheus-сумісний експорт і перелік згенерованих графіків.

Службові файли requirements.txt, pyproject.toml і README.md описують залежності, параметри проєкту та порядок запуску. Вони не беруть участі безпосередньо в обробці телеметрії, але забезпечують відтворюваність середовища, налаштування інструментів і можливість повторного розгортання прототипу.

Отже, діаграма пакетів демонструє фактичну структуру програмного проєкту без включення тестового коду. Основне програмне ядро зосереджено в пакеті app, інтерфейс користувача винесено в static, а результати роботи системи зберігаються в каталогах data, charts і reports. Така структура спрощує супровід, повторне використання модулів і подальше розширення системи..

2.3 Модель даних, синтетичне забезпечення та сценарії генерації телеметрії

Модель даних реалізовано в модулі schemas.py. У ньому визначено перелік профілів систем SystemProfile, рівнів критичності Severity, типів сигналів SignalType, а також моделі RawTelemetryEvent, NormalizedTelemetryEvent, PipelineResult і MetricsSnapshot (табл. 2.4). Такий поділ відокремлює сирий

вхідний сигнал від нормалізованої події, що пройшла адаптацію, маскування, збагачення, фільтрацію і підготовку до зберігання.

RawTelemetryEvent описує подію до обробки. Вона містить джерело, профіль системи, тип сигналу, рівень критичності, повідомлення, код події, часову мітку, trace/span-контекст, ідентифікатори пристрою або сервісу, операцію, числове значення, одиницю вимірювання та словник атрибутів. NormalizedTelemetryEvent доповнює ці дані службовими полями: event_id, ingest_timestamp, enriched, payload_size_bytes, processing_latency_ms, correlation_status, retention_tier, was_sampled, was_dropped, drop_reason, source_class, adaptive_policy, adaptation_reason і signal_priority.

Таблиця 2.4 – Моделі даних і перелічення, реалізовані в app.schemas

Елемент коду	Тип	Призначення
SystemProfile	Enum	Задає класи систем: embedded, iot_edge, enterprise, cloud_native.
Severity	Enum	Описує рівні критичності: DEBUG, INFO, WARN, ERROR, CRITICAL.
SignalType	Enum	Описує тип сигналу: log, metric, trace, sensor, security, health.
RawTelemetryEvent	Pydantic-модель	Описує подію до обробки та валідує вхідні поля API або генератора.
NormalizedTelemetryEvent	Pydantic-модель	Описує результат нормалізації, адаптації, фільтрації й збагачення.
PipelineResult	Pydantic-модель	Групує прийняті та відкинуті події й коротке зведення роботи конвеєра.
MetricsSnapshot	Pydantic-модель	Фіксує агреговані показники спостережуваності та якість синтетичних даних.

Синтетичне забезпечення реалізовано в модулі synthetic.py через функцію generate_synthetic_events(count, scenario, seed, start). На відміну від попередньої демонстраційної логіки, генератор не обмежується кількома фіксованими записами. Він формує послідовності подій з керованою випадковістю, різними профілями систем, рівнями критичності, типами сигналів, джерелами, trace/span-контекстом, атрибутами мережі, батареї, контейнерного середовища, аудиту й високої кардинальності. Параметр seed забезпечує повторюваність експериментів.

Сценарії mixed, embedded_pressure, lossy_iot, cloud_burst та incident задають різні умови перевірки. Сценарій mixed поєднує всі профілі, embedded_pressure створює події вбудованих систем, lossy_iot імітує нестабільний зв’язок і буферизацію на периферії, cloud_burst підсилює хмарний потік із trace-контекстом і висококардинальними мітками, а incident збільшує частку помилок і критичних станів. Завдяки цьому прототип має дані не лише для перевірки приймання подій, а й для перевірки адаптивних політик, метрик і графіків.

2.4 Реалізація серверного застосунку, конвеєра та політик обробки

Серверний застосунок реалізовано в модулі main.py на основі FastAPI. У цьому модулі створюється об’єкт app, підключаються каталоги /static і /charts, а також визначаються маршрути /, /dashboard, /events, /simulate, /metrics/json, /metrics/prometheus і /charts/regenerate. Кореневий маршрут і маршрут /dashboard перенаправляють користувача на static/dashboard.html, тому загальний дашборд є основною точкою взаємодії з прототипом.

Маршрут POST /events приймає об’єкт RawTelemetryEvent, передає його до process_events() і записує як прийняті, так і відкинуті події через insert_events(). Це відповідає логіці спостережуваності: відкинута подія не зникає повністю, а зберігається з ознакою was_dropped і причиною drop_reason. Маршрут GET /events читає дані зі сховища через query_events() і підтримує фільтри profile, severity та limit.

Маршрут POST /simulate запускає повний інтеграційний сценарій. Якщо параметр clear має значення true, сховище очищується через reset(). Далі generate_synthetic_events() створює пакет сирих подій, process_events() виконує адаптацію й нормалізацію, insert_events() записує результати, generate_all_charts() будує графіки, а compute_metrics() повертає актуальний знімок показників. Отже, один endpoint демонструє повний шлях від синтетичного джерела до метрик і графічних результатів.

Адаптивність під джерела сигналів реалізовано в модулі source_adaptation.py. Основна функція resolve_adaptive_policy(raw) поєднує базову політику профілю з таксономією джерел SOURCE_CLASSES, пріоритетами типів сигналів SIGNAL_PRIORITY і контекстними атрибутами події. Рішення повертається як AdaptiveDecision і містить політику, клас джерела, назву адаптивної політики, числовий пріоритет сигналу та текстову причину адаптації.

Конвеєр обробки реалізовано в модулі pipeline.py. Функція process_events() послідовно викликає адаптацію, нормалізацію, перевірку профільної політики, семплювання, обмеження payload, визначення статусу зберігання і формування списків accepted та dropped. Функція normalize() створює стабільний event_id, обчислює розмір payload, додає ingest_timestamp, формує event_code, збагачує подію службовими атрибутами й переносить результат адаптації в поля source_class, adaptive_policy, adaptation_reason і signal_priority.

Безпечність журналювання підтримується функцією mask_attributes(), яка рекурсивно обробляє словник атрибутів і маскує ключі password, token, secret, api_key, authorization і card_number. Функція correlation_status() класифікує подію як full, partial або missing залежно від наявності trace/span-контексту, ідентифікатора пристрою або сервісу. Функція retention_tier() задає рівень зберігання hot, warm, cold або discard з урахуванням профілю, критичності та типу сигналу.

Профільні правила зосереджено в модулі profiles.py. Структура TelemetryPolicy задає мінімальний рівень критичності, максимальний розмір payload, коефіцієнти семплювання для DEBUG та INFO, правила збереження метрик і трас, вимогу кореляційного контексту для хмарного профілю та строки гарячого і теплого зберігання. Словник POLICIES містить політики для embedded, iot_edge, enterprise і cloud_native.

Сховище реалізовано в модулі storage.py. Функція connect() створює SQLite-підключення, виконує DDL-скрипт таблиці telemetry_events (табл. 2.5) і перевіряє наявність нових колонок, пов’язаних з адаптивністю: source_class,

adaptive_policy, adaptation_reason, signal_priority. Таблиця має індекси за профілем і часом, критичністю і часом, trace-контекстом, класом джерела та адаптивною політикою.

Таблиця 2.5 – Ключові поля таблиці telemetry_events

Група полів	Поля	Призначення
Ідентифікація події	event_id, event_code, source, profile, signal_type, severity	Унікальність, класифікація та пошук події.
Час і контекст	timestamp, ingest_timestamp, trace_id, span_id, device_id, service_name, operation	Відтворення часу виникнення, приймання та кореляційного контексту.
Дані сигналу	message, value, unit, attributes, enriched	Зберігання повідомлення, числового значення, JSON-атрибутів і службового збагачення.
Показники обробки	payload_size_bytes, processing_latency_ms, correlation_status, retention_tier	Оцінка обсягу, затримки, повноти контексту та рівня зберігання.
Рішення конвеєра	was_sampled, was_dropped, drop_reason	Фіксація прийняття, семплювання або відкидання події.
Адаптивність	source_class, adaptive_policy, adaptation_reason, signal_priority	Збереження результату адаптації під джерело сигналу та тип події.

Спостережуваність реалізовано у модулях metrics.py і charts.py. Функція compute_metrics() читає всі події зі сховища та формує MetricsSnapshot: кількість подій, кількість прийнятих і відкинутих, частку відкидання, кількість критичних і помилкових подій, покриття кореляційним контекстом, p50/p95/p99 затримки, середній розмір payload, обсяг збережених даних, розподіли за профілями, критичністю, типами сигналів, адаптивними політиками та класами джерел. Функція prometheus_text() перетворює цей знімок на Prometheus-сумісний текстовий формат.

Модуль charts.py будує сім графіків: обсяг подій за профілями, розподіл критичності, частку відкидання за профілями, затримки обробки, стан кореляційного контексту, узагальнені KPI спостережуваності та адаптацію за класами джерел сигналів. Така реалізація виправляє обмеження простого журналу подій: прототип не лише зберігає записи, а й оцінює якість телеметричного потоку.

Загалом, у другому розділі було виконано проєктування програмного прототипу системи журналювання та обробки телеметрії залежно від класу програмних систем. На основі сформульованих вимог визначено призначення системи, її основні функції, обмеження та очікувані результати роботи. Прототип орієнтовано не лише на приймання й збереження телеметричних подій, а й на демонстрацію відмінностей у стратегіях обробки для вбудованих, IoT/edge, корпоративних і хмарно-орієнтованих систем.

Запропонована архітектура побудована як модульна система з чітким поділом відповідальності між користувацьким інтерфейсом, API-рівнем, доменною логікою, сховищем, засобами генерації синтетичних подій, обчисленням метрик і побудовою графіків. Розроблена структура дозволяє приймати телеметричні події, приводити їх до єдиного формату, застосовувати політики адаптації залежно від джерела сигналу, виконувати фільтрацію й маскування чутливих атрибутів, зберігати результати в локальному сховищі, формувати показники спостережуваності та будувати графічні артефакти для аналізу. Це створює передумови для експериментального дослідження роботи системи, перевірки її поведінки на синтетичних даних і порівняння стратегій телеметрії для різних класів програмних систем.

РОЗДІЛ 3 ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір стратегій тестування та критерії якості

Тестування Telemetry Observer Prototype спрямоване на перевірку коректності запуску та інженерної поведінки конвеєра. Для системи телеметрії критичною є відповідність обробки подій обраній політиці, відсутність витоку чутливих даних, збереження причин відкидання, стабільність агрегованих показників і відтворюваність синтетичних сценаріїв. Тому стратегія тестування поєднує модульний, інтеграційний, функціональний, негативний, безпековий, регресійний і данісно-орієнтований рівні.

Загальна логіка тестування узгоджується з підходами стандарту ISO/IEC/IEEE 29119, де тестування розглядається як керований процес планування, проєктування, виконання та оцінки результатів перевірки програмної системи [35]. Для цього прототипу тестування має бути трасованим: кожний клас тестів повинен відповідати певній вимозі або архітектурному ризику. Наприклад, тести маскування відповідають вимозі безпечного журналювання, тести correlation_id – вимозі кореляції, а тести embedded-профілю – вимозі мінімізації телеметрії.

Модульне тестування перевіряє функції normalize(), apply_policy(), mask_value(), limit_payload(), stable_event_hash() і is_allowed_severity(). Інтеграційне тестування перевіряє маршрути API, зв’язок із TelemetryStore, вставку в SQLite та повернення summary. Функціональне тестування перевіряє демонстраційні сценарії з погляду користувача. Негативне тестування перевіряє некоректні вхідні структури, порожні поля, відсутні кореляційні ідентифікатори та дані нижче порогу. Безпекове тестування зосереджене на маскуванні секретів і персональних шаблонів.

Діаграму активності для опису стратегії тестування подано на рис. 3.1. Вона відображає послідовність: спочатку визначаються критерії приймання,

потім проєктуються синтетичні дані, після цього виконуються модульні й інтеграційні тести, а завершальний етап полягає в аналізі результатів і дефектів.

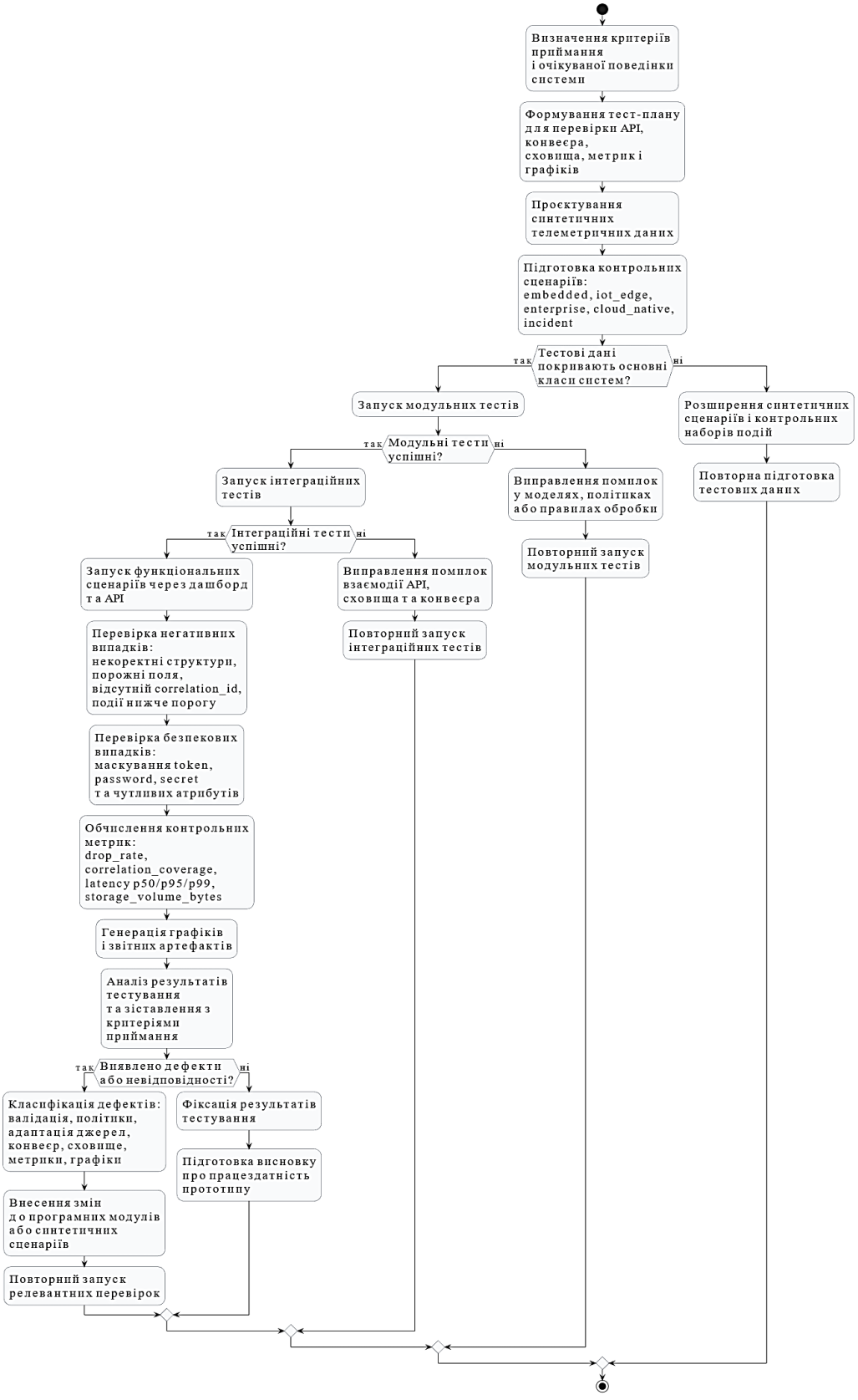

Рисунок 3.1 – UML-діаграма активності стратегії тестування

Таблиця 3.1 узагальнює обґрунтування вибору стратегій тестування програмного прототипу. Оскільки розроблена система поєднує API-рівень, конвеєр обробки телеметрії, профільні політики, локальне сховище, механізми формування метрик і графічних артефактів, її перевірка не може обмежуватися лише ручним запуском дашборда або перевіркою окремого endpoint-а. Для підтвердження працездатності необхідно поєднати кілька рівнів тестування, кожен із яких виявляє окремий клас можливих дефектів.

Таблиця 3.1 – Обґрунтування вибору стратегій тестування

Рівень тестування	Об’єкт перевірки	Типові дефекти	Інструменти
Модульний	pipeline.py, profiles.py, schemas.py	хибний поріг severity, помилка маскування, неправильний hash	pytest, параметризовані тести
Інтеграційний	API + storage + pipeline	помилки маршрутизації, невідповідність JSON-відповіді, втрата записів	TestClient, SQLite tmp_path
Функціональ-ний	демонстраційні сценарії	UI/API не відтворює очікувану поведінку профілю	ручний запуск, Swagger UI
Негативний	некоректні запити та неповні події	відсутність 422, прийняття неповної події	pytest, HTTP-кейси
Безпековий	payload із секретами	витік token, email, card, authorization	тести маскування, перегляд БД
Регресійний	зафіксовані сценарії	неочікувана зміна політик після рефакторингу	автоматичний прогін 46 тестів
Даніcно-орієнтований	синтетичні набори	нерепрезентативні дані, відсутність межових випадків	генератор demo_events, seed

Модульне тестування обрано як базовий рівень перевірки, оскільки значна частина логіки системи зосереджена в окремих Python-модулях і може перевірятися без запуску вебсервера. До таких модулів належать schemas.py, profiles.py, source_adaptation.py і pipeline.py. На цьому рівні перевіряється коректність моделей даних, профільних правил, адаптації під джерело сигналу, нормалізації подій, фільтрації, маскування чутливих атрибутів і визначення причин відкидання подій. Типовими дефектами, які має виявляти модульне тестування, є неправильний поріг критичності, помилка в політиці профілю,

збереження чутливого значення без маскування або некоректне визначення статусу accepted / dropped.

Інтеграційне тестування потрібне для перевірки взаємодії між API, конвеєром обробки та сховищем. Навіть якщо окремі функції працюють коректно, помилки можуть виникати на межі модулів: під час приймання JSON-запиту, перетворення його на внутрішню модель, запису в SQLite або формування відповіді клієнтові. Для такого тестування доцільно використовувати TestClient і тимчасове SQLite-сховище, щоб не залежати від попередніх запусків системи.

Функціональне тестування орієнтоване на перевірку системи з позиції користувацьких сценаріїв. До таких сценаріїв належать запуск симуляції для різних профілів, перегляд збережених подій, отримання підсумкових метрик, генерація графіків і аналіз результатів через дашборд. Функціональні перевірки особливо важливі для демонстраційної системи, оскільки саме вони підтверджують, що користувач може відтворити експериментальний сценарій без прямого втручання в код.

Негативне тестування включено до стратегії через те, що система працює з вхідними телеметричними подіями, які потенційно можуть бути неповними, надлишковими, некоректними або небезпечними з погляду подальшого збереження. Перевіряються ситуації з відсутніми обов’язковими полями, неправильними типами значень, порожнім correlation_id, низькою критичністю події, перевищенням допустимого розміру payload або невідповідністю профілю. Такий рівень тестування дає змогу переконатися, що система не приймає некоректні події без пояснення, а формує контрольований результат обробки.

Безпекове тестування в межах прототипу зосереджене не на повному аудиті захищеності, а на перевірці безпечності журналювання. Для систем телеметрії це принципово важливо, оскільки журнали часто містять службові токени, ідентифікатори користувачів, адреси електронної пошти, номери карток або інші чутливі атрибути. Тому в таблиці 3.1 виділено перевірки маскування таких значень до моменту збереження. Очікуваним результатом є те, що

небезпечні поля не потрапляють у SQLite-сховище та звітні артефакти у відкритому вигляді.

Регресійне тестування потрібне для підтвердження стабільності системи після змін у профільних політиках, конвеєрі або механізмах формування метрик. Оскільки логіка обробки подій залежить від класу програмної системи, навіть невелика зміна порогу або правила фільтрації може вплинути на частку прийнятих подій, покриття кореляційним контекстом або обсяг збережених даних. Тому повторний запуск тестів після змін у коді є необхідною умовою збереження передбачуваної поведінки прототипу.

3.2 Формування тест-плану та тестових даних

Тест-план було сформовано на основі вимог, модулів системи та профілів телеметрії. Особлива увага приділена не кількості однотипних перевірок, а покриттю інженерних ризиків. Для embedded-профілю перевіряється, що система не зберігає низькорівневий інформаційний шум. Для enterprise і cloud-профілів перевіряється обов’язкова кореляція. Для всіх профілів перевіряється маскування чутливих значень. Для сховища перевіряється, що accepted і dropped-події зберігаються разом із поясненням, а summary коректно агрегує результат.

Підготовка тестових даних виконана у трьох формах. Перша форма – ручні фікстури у тестах, які дозволяють точно задати профіль, severity, payload і correlation_id. Друга форма – генератор demo_events(), який створює сценарні пакети для кожного профілю. Третя форма – мутаційний підхід у тестах, коли коректна подія змінюється шляхом видалення correlation_id, додавання token, збільшення payload або зміни рівня критичності. Такий підхід дозволяє перевірити не лише основні сценарії, а й граничні стани.

Для майбутнього розширення тестування варто передбачити окремий модуль synthetic_factory.py. Він міг би генерувати події за схемою: профіль → тип сценарію → рівень шуму → частка чутливих полів → частка відсутніх кореляцій → обсяг payload → часовий розподіл. Такий генератор дозволив би створювати великі набори для перевірки продуктивності, стабільності

підсумкового звіту, ефективності індексів і витрат на збереження. Окремим режимом може бути replay-набір, де зафіксована послідовність подій повторюється після кожної зміни коду.

На рис. 3.2 показано організацію синтетичних даних. Вона відокремлює генератор профілю, нормальні події, аномальні події, мутації та тестові набори. Такий поділ важливий, оскільки одна й та сама функція обробки повинна перевірятися не лише на «правильному» вході, а й на пошкодженому, надмірному або небезпечному вході.

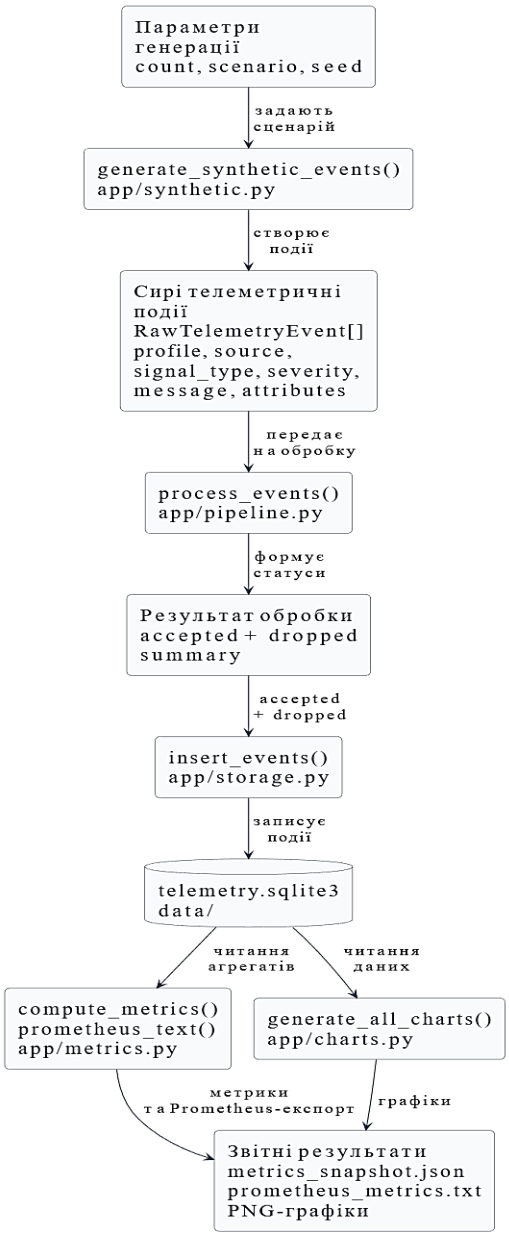

Рисунок 3.2 – Організація синтетичних даних для тестування

Проєктування фікстур виконано так, щоб кожен тест мав мінімальний, але достатній набір полів. Наприклад, допоміжна функція make_event() задає стандартну enterprise-подію з correlation_id, а конкретний тест змінює лише ту ознаку, яка перевіряється. Це зменшує дублювання в тестах і робить причину помилки очевидною. Якщо тест embedded_drops_info_event падає, проблема, найімовірніше, пов’язана з порогом severity, а не з випадковими даними payload. Фікстури та джерела тестових даних зібрано в табл. 3.2.

Таблиця 3.2 – Фікстури та джерела тестових даних

Фікстура / джерело	Де використовується	Що контролює	Перевага
make_event()	pipeline tests	мінімальну валідну подію	зменшує дублювання й випадковість
demo_events(profile)	simulation/API tests	типові сценарії профілів	перевіряє реалістичніші пакети
tmp_path SQLite	storage tests	ізоляцію бази даних	немає залежності від попередніх запусків
TestClient(app)	API tests	HTTP-рівень без реального сервера	швидке інтеграційне тестування
payload з token/password	security tests	маскування секретів	перевіряє безпечне журналювання
event без correlation_id	policy tests	обов’язкову трасованість	перевіряє enterprise/ cloud-політики

Формування тестових даних має враховувати не тільки валідність JSON-структури, а й семантичну правдоподібність подій (табл. 3.3). Наприклад, для embedded-профілю реалістичними є короткі повідомлення, обмежений payload, коди стану драйверів, watchdog-події та показники сенсорів. Для IoT/edge-профілю характерні стан радіоканалу, буферизація, рівень батареї, ідентифікатор пристрою та затримка передавання. Для enterprise-профілю важливі аудит дій, доступ користувача, бізнес-операція і кореляція запиту. Для cloud-профілю необхідні trace/span-події, назви сервісів, метрики черг, timeout-и та ідентифікатори трас.

Через це тестовий набір не може бути складений лише з абстрактних подій виду message="test". Такі записи перевіряють роботу типів, але не перевіряють предметну логіку. У роботі використано змішаний підхід: частина тестів

застосовує мінімальні фікстури, щоб точно перевірити одну властивість, а частина спирається на demo_events(), щоб зберегти зв’язок із реалістичними профілями. Для подальшого розвитку доцільним є створення каталогу JSON-фікстур, у якому кожен файл міститиме вхідні події та очікуваний результат обробки (табл. 3.4).

Таблиця 3.3 – Семантичні вимоги до тестових подій різних профілів

Профіль	Семантично обов’язкові поля	Бажані поля	Небажані або ризикові поля
embedded	profile, source, severity, message	event_code, driver, sensor, deadline_ms	великі масиви raw-даних, персональні дані
iot_edge	profile, source, severity, device_id	battery, rssi, buffered_events, channel	відкриті token, email оператора, необмежені списки
enterprise	profile, source, severity, correlation_id	actor, operation_id, business_object, audit_type	паролі, Bearer-token, повний експорт персональних даних
cloud	profile, source, severity, correlation_id, service_name	span, latency_ms, provider, queue_depth	номер картки, trace без root-id, високоваріативні label
security	profile, source, severity, event_type	ip, actor, rule_id, action	секрети доступу, повні ключі API, необроблені заголовки

Таблиця 3.4 – Каталог мутацій для синтетичного тестування

Тип мутації	Приклад	Очікувана реакція	Тестова цінність
Видалення поля	відсутній correlation_id у cloud	dropped: missing_correlation_id	перевірка обов’язкової кореляції
Зниження severity	embedded info/debug	dropped: severity_below_warning	перевірка мінімального журналювання
Надмірний payload	10-50 ключів у embedded	Скорочення і _truncated_keys	контроль ресурсоємності
Секрет у ключі	token="secret"	Значення ***	захист від витоку
Секрет у рядку	email або номер картки	Маскування шаблону	захист від неочевидних витоків
Довге повідомлення	200-500 символів	Обрізання (truncation)	контроль обсягу логів
Невідомий enum	profile="desktop"	HTTP 422	валідація контракту
Порожній source	source=""	HTTP 422	якість ідентифікації джерела
Дублювання	дві однакові події	однаковий hash	перевірка стабільності event_id
Час без timezone	datetime без tzinfo	UTC-нормалізація	узгодженість часових міток

3.3 Організація тестування, приклади запуску та результати

Практична перевірка прототипу передбачає локальний запуск без зовнішньої інфраструктури. Мінімальний сценарій складається зі встановлення залежностей з requirements.txt, запуску FastAPI-застосунку через uvicorn app.main:app --reload, відкриття /dashboard або /static/dashboard.html і виконання симуляції через дашборд або endpoint POST /simulate. Оскільки в проєкті вже наявні синтетичний генератор, локальне сховище, графіки і звітні файли, демонстрація не залежить від реальних журналів або промислових агентів.

Під час запуску користувач може сформувати телеметричний потік різного типу: змішаний, embedded-навантаження, IoT зі втратами зв'язку, хмарний сплеск або інцидент. Після виконання симуляції в каталозі data оновлюється база telemetry.sqlite3, у каталозі charts формуються PNG-графіки, а в каталозі reports зберігаються знімки метрик і результати контрольних запусків. Таким чином прототип має не лише інтерактивний, а й відтворюваний експериментальний контур.

Загальний дашборд відображається через static/dashboard.html (рис. 3.3). Він використовує API для запуску симуляцій, отримання JSON-метрик, перегляду подій і оновлення графіків. Окремий маршрут /metrics/prometheus дозволяє переглянути ті самі метрики в текстовому форматі, сумісному з Prometheus. Це демонструє, що система може працювати як локальний навчальний прототип і водночас має зрозумілу точку розширення до промислової системи моніторингу.

Консольний модуль cli.py дублює основні операції без вебінтерфейсу. Команда seed генерує синтетичні події, запускає конвеєр і записує результат; команда metrics виводить агрегований знімок метрик або Prometheus-сумісний текст; команда charts будує графіки. Це корисно для повторюваних експериментів, підготовки демонстраційних даних і перевірки працездатності проєкту після змін у коді.

Демонстрація має показувати не лише факт приймання подій, а й відмінність політик для різних класів систем. Для embedded-профілю важливими

є компактність і відкидання малозначущих повідомлень. Для IoT/edge – реакція на слабку мережу, низький заряд і обмеження payload. Для enterprise – збереження аудиторського контексту та маскування службових маркерів. Для cloud_native – trace/span-контекст, контроль високої кардинальності та Prometheus-сумісні показники. Саме ці відмінності відображаються в полях source_class, adaptive_policy, adaptation_reason, signal_priority та в метриках. Основні сценарії роботи з розробленим програмним прототипом зібрано в табл. 3.5.

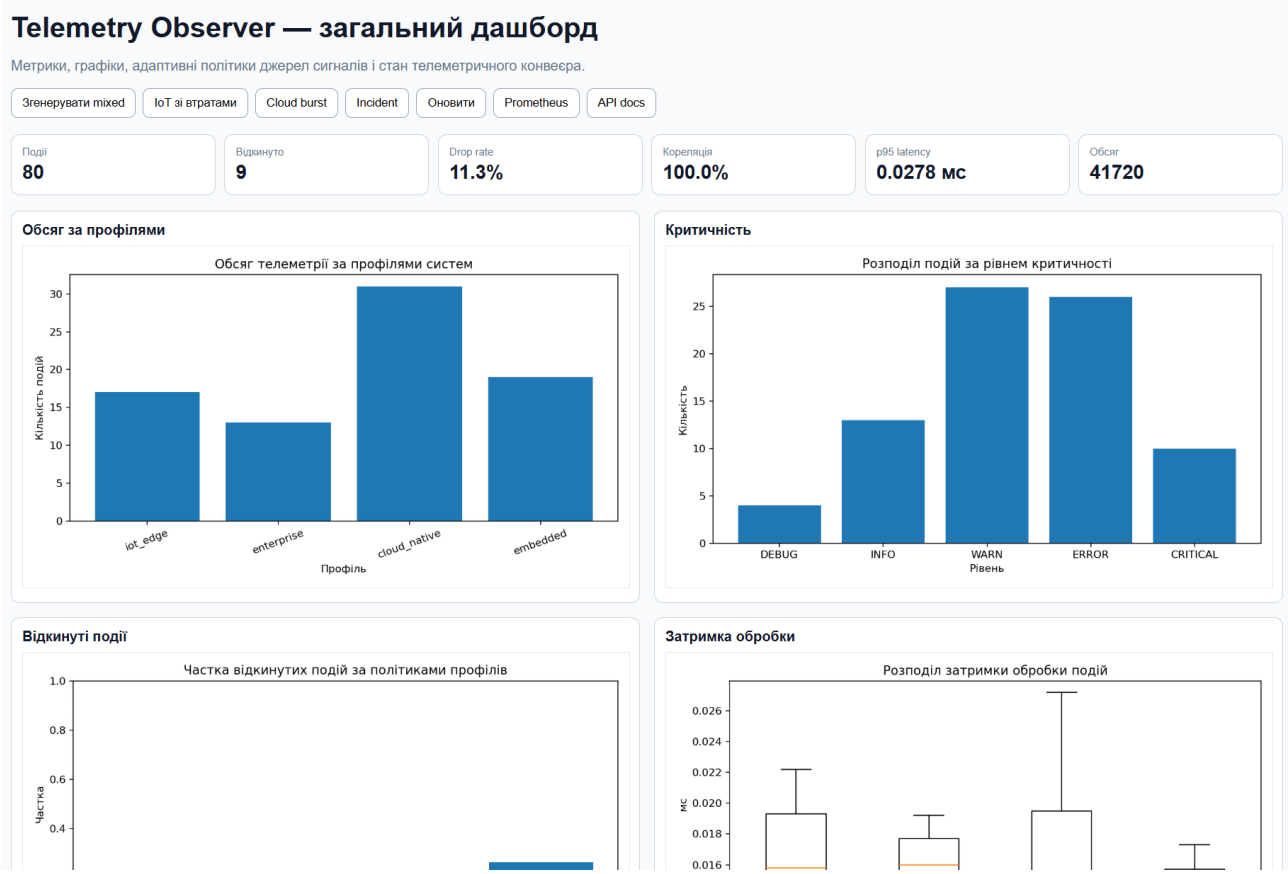

Рисунок 3.3 – Загальний вигляд дашборду

Автоматизоване тестування організовано засобами pytest [31]. Тести поділено на три файли: test_pipeline.py, test_api.py і test_storage.py. Така структура відповідає архітектурі системи: доменна обробка перевіряється окремо від HTTP-рівня, API перевіряється через FastAPI TestClient, а сховище перевіряється з тимчасовою SQLite-базою у tmp_path. Це запобігає взаємному

впливу тестів і дозволяє запускати їх повторно без ручного очищення глобального стану.

Таблиця 3.5 – Основні сценарії практичної роботи з прототипом

Сценарій	Спосіб запуску	Результат
Відкрити дашборд	GET /dashboard або GET /static/dashboard.html	Відображення HTML-панелі, KPI, графіків і елементів запуску сценаріїв.
Прийняти одну подію	POST /events	Обробка RawTelemetryEvent, запис accepted/dropped результату, повернення summary.
Запустити симуляцію	POST /simulate?scenario=...&count=...&seed=...	Генерація подій, конвеєрна обробка, запис у SQLite, оновлення графіків і метрик.
Переглянути події	GET /events?profile=...&severity=...&limit=...	Отримання списку нормалізованих подій із фільтрами.
Отримати JSON-метрики	GET /metrics/json	Повернення MetricsSnapshot у структурованому форматі.
Отримати Prometheus-експорт	GET /metrics/prometheus	Повернення текстового формату для систем збору метрик.
Регенерувати графіки	POST /charts/regenerate або CLI charts	Побудова всіх PNG-графіків у каталозі charts.

Модульні тести конвеєра охоплюють нормалізацію часу, пороги важливості, скорочення сенсорного payload, маскування token, password, email і номерів карток, обмеження кількості ключів, стабільність хешування та генерацію demo-подій. API-тести перевіряють доступність профілів, приймання валідної події, відхилення невалідного payload, пакетну обробку, демонстраційні ендпоінти, список подій, фільтр dropped і очищення сховища. Тести сховища перевіряють insert_many(), list_events(), clear(), фільтрацію, summary і метрики якості.

Тести запускалися в локальному середовищі Python 3.13. Результат автоматичного прогону становив 46 успішних перевірок (рис. 3.3). Інформацію по відповідних тестових випадках систематизовано в додатку А. Цей результат не означає, що система є промислово завершеною, але підтверджує коректність реалізованого навчального ядра: профільні політики працюють, невалідні запити відхиляються, чутливі дані маскуються, а сховище повертає коректні агрегати.

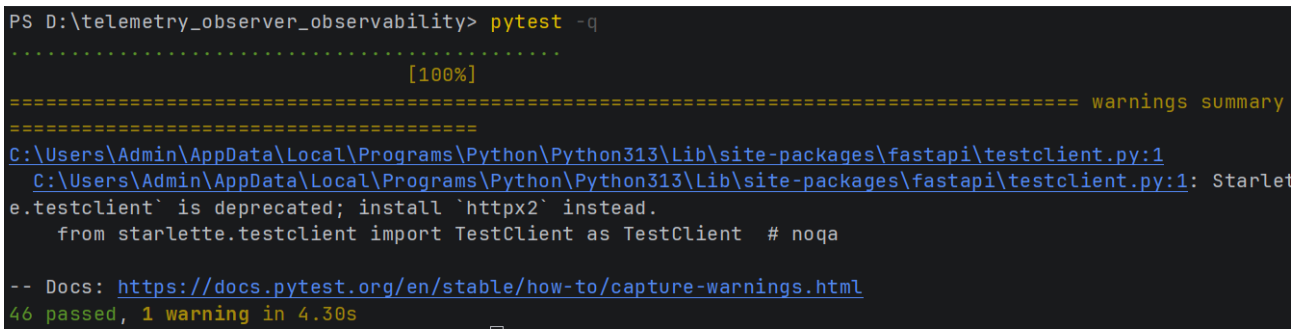

Рисунок 3.3 – Зразок результату запуску автоматизованих тестів

Окремо було виконано ручну перевірку запуску сервера та інтерфейсу. Після команди uvicorn app.main:app --reload головна сторінка відкривається в браузері, кнопки запуску профілів створюють записи, таблиця подій оновлюється, а summary показує кількість прийнятих і відкинутих записів. Ручна перевірка не замінює автоматизовані тести, але підтверджує, що API, сховище та інтерфейс працюють як єдиний сценарій.

Таблиця 3.7 є RTM-матрицею, яка встановлює відповідність між вимогами, визначеними в попередньому розділі, та тестовими випадками, відпрацьовуваними в ході тестування. Отримане зведення показує достатнє покриття вимог тестами в межах кваліфікаційної роботи.

3.4 Аналіз результатів та обмеження програмного прототипу

Аналіз результатів за групами тестів показав, що найбільш критичні сценарії покриті автоматично. Зокрема, тести конвеєра перевіряють, чи не змішується доменна логіка з API-рівнем. Якщо змінити правило мінімального severity для embedded-профілю, тест test_embedded_severity_threshold одразу покаже зміну очікуваної поведінки. Якщо прибрати рекурсивне маскування, впадуть тести token/password або вкладених значень. Це дає можливість виконувати рефакторинг без втрати контролю над профільними політиками.

Інтеграційні тести API мають іншу цінність: вони перевіряють не самі функції, а зв’язок між HTTP-контрактом, Pydantic-моделями, конвеєром і сховищем. Наприклад, тест пакетної обробки перевіряє, що одна прийнята й одна

відкинута подія повертаються як узгоджений результат, а не як набір незалежних відповідей. Тест фільтрації dropped показує, що причина відкидання збережена й доступна через API. Це важливо для практичного використання, оскільки аналітик має бачити не лише результат, а й пояснення.

Таблиця 3.7 – Матриця трасованості вимог і тест-кейсів

Вимога	Тест-кейси	Ступінь покриття	Коментар
FR-1 Профілі систем	TC-16, TC-21, TC-25, TC-37-40	високий	Перевірено на рівні генератора, API та сценаріїв.
FR-2 Приймання подій	TC-22, TC-23, TC-24	високий	Перевірено валідний і невалідний JSON.
FR-3 Нормалізація	TC-01, TC-19, TC-20	середній	Покрито час і hash, можна додати форматні мутації.
FR-4 Профільна політика	TC-02-09, TC-14-15, TC-37	високий	Покрито пороги, debug, payload і кореляцію.
FR-5 Маскування	TC-10-13, TC-39, TC-45	високий	Покрито секрети, email, card, вкладені значення.
FR-6 Збереження	TC-31-36	високий	Покрито вставку, очищення, фільтри і summary.
FR-7 Пошук і перегляд	TC-27, TC-28, TC-33, TC-34	середній	Покрито API-фільтри, без перевірки пагінації.
FR-8 Аналітика	TC-26, TC-35, TC-36, TC-46	середній	Покрито quality, потрібні більші набори даних.
FR-9 Демонстрації	TC-25, TC-37-40	високий	Покрито всі профілі.
FR-10 Відтворюваність	TC-16, TC-44	середній	Є контроль seed, потрібен окремий regression-fixture файл.

Тести сховища підтверджують, що локальна база підтримує фільтрацію за профілем і статусом, очищення, підрахунок accepted/dropped та метрики якості. Для майбутньої промислової версії ці тести можуть бути збережені як контракт поведінки при переході з SQLite на PostgreSQL або інше сховище. Тоді заміна технології не повинна змінити доменний зміст операцій list_events() і summary().

Під час тестування не було виявлено аварійних помилок, які блокують запуск або основні сценарії. Водночас за результатами аналізу тест-плану виділено кілька контрольованих обмежень, які не є дефектами поточної реалізації, але мають бути зафіксовані для наступної ітерації. Найбільш помітне обмеження – відсутність великого навантажувального набору. Друге

обмеження – відсутність окремого сховища еталонних JSON-фікстур. Третє – відсутність тестів на конкурентний доступ до SQLite.

Окремим результатом тестування програмного прототипу є формування графічних матеріалів, що відображаються на дашборді системи. На відміну від звичайного журналу подій, дашборд дає змогу швидко оцінити якість телеметричного потоку, співвідношення прийнятих і відкинутих подій, розподіл критичності, затримки обробки, наявність кореляційного контексту та роботу адаптивних політик для різних джерел сигналів.

Графіки формуються модулем app/charts.py на основі даних, збережених у SQLite-сховищі telemetry.sqlite3, та агрегованих показників, які обчислюються модулем app/metrics.py. Після запуску симуляції або повторної генерації графіків результати зберігаються у каталозі charts у форматі PNG і відображаються на сторінці dashboard.html. Такий підхід дозволяє поєднати тестування програмної логіки з візуальним контролем результатів обробки телеметрії.

Графік 01_volume_by_profile.png відображає кількість подій за профілями програмних систем (рис. 3.4). Він використовується для перевірки того, чи синтетичний сценарій справді формує події для різних класів систем: embedded, iot_edge, enterprise і cloud_native. Цей графік важливий для контролю репрезентативності тестового набору. Якщо певний профіль відсутній або представлений занадто малою кількістю подій, результати тестування не можна вважати повними для порівняння стратегій обробки телеметрії.

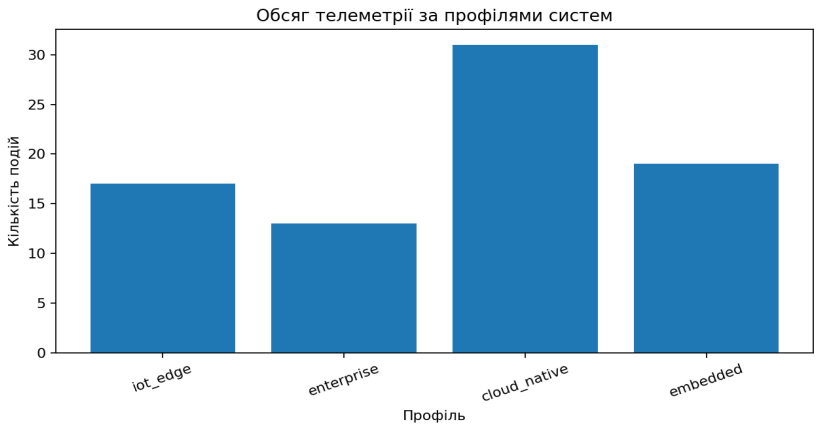

Рисунок 3.4 – Обсяг телеметрії за профілями систем

Графік 02_severity_distribution.png показує розподіл подій за рівнями критичності (рис. 3.5). Він дає змогу оцінити, чи містить тестовий набір достатню кількість інформаційних, попереджувальних, помилкових і критичних подій. Для системи журналювання це суттєво, оскільки політики фільтрації залежать від рівня важливості події. Наприклад, для вбудованого профілю низькокритичні повідомлення можуть відкидатися як інформаційний шум, тоді як для корпоративного або хмарного профілю частина таких подій може зберігатися для аудиту чи подальшої кореляції.

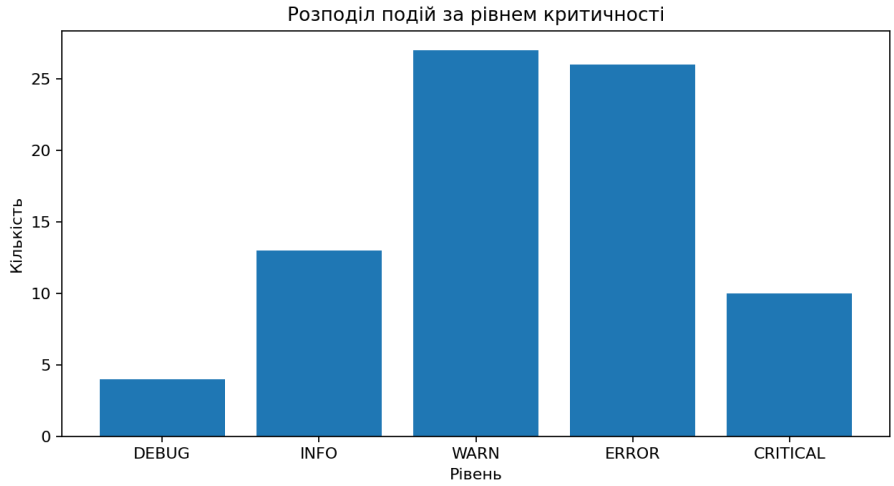

Рисунок 3.5 – Розподіл подій за рівнем критичності

Графік 03_drop_rate_by_profile.png відображає частку відкинутих подій за профілями (рис. 3.6). Він є одним із ключових індикаторів адаптивності системи, оскільки показує, що правила фільтрації не застосовуються однаково до всіх джерел. Вищий рівень відкидання для малоресурсного або вбудованого профілю може бути очікуваним результатом, якщо система обмежує обсяг службових даних. Водночас надмірне відкидання подій у корпоративному або хмарному профілі може свідчити про занадто жорсткі правила фільтрації або про дефекти у вхідному наборі даних.

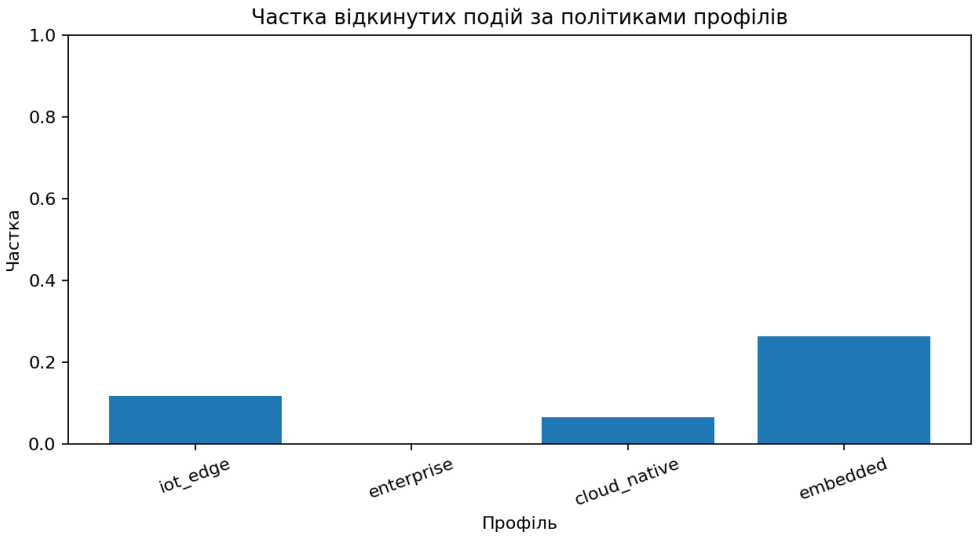

Рисунок 3.6 – Частка відкинутих подій за політиками профілів

Графік 04_latency_by_profile.png показує затримки обробки подій для різних профілів (рис. 3.7). Його призначення полягає в оцінюванні того, як конвеєр обробки працює з різними типами телеметричних записів. Показники затримки використовуються разом із метриками p50, p95 і p99, що дає змогу оцінити не лише середню поведінку системи, а й граничні випадки. Якщо затримка для певного профілю помітно вища, це може бути пов’язано з більшим обсягом атрибутів, складнішою адаптацією джерела або додатковими перевірками під час нормалізації та маскування.

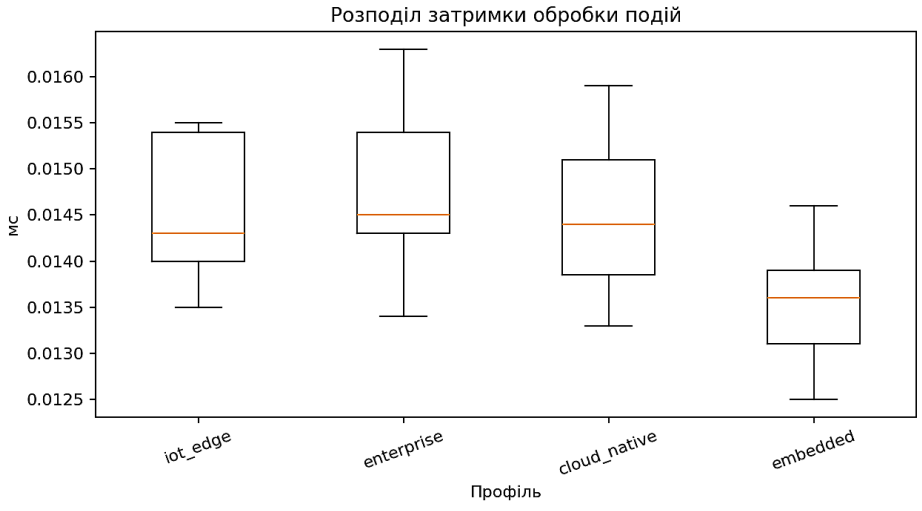

Рисунок 3.7 – Розподіл затримки обробки подій

Графік 05_correlation_status.png відображає стан кореляційного контексту подій. Він показує, яка частина записів має достатній зв’язок із запитом, сесією, trace/span-контекстом або іншим ідентифікатором виконання. Для систем спостережуваності ця характеристика є критичною, оскільки ізольована подія без кореляційного ідентифікатора має значно меншу діагностичну цінність. Якщо графік показує низьке покриття кореляційним контекстом, це означає, що подальший аналіз інцидентів буде ускладнений, навіть якщо самих журнальних записів достатньо.

Графік 06_observability_kpis.png агрегує основні показники спостережуваності (рис. 3.8). До них належать частка відкидання подій, покриття кореляційним контекстом, обсяг збережених даних, затримки обробки та інші узагальнені метрики. Його роль полягає в тому, щоб надати швидку оцінку загального стану телеметричного потоку після виконання сценарію. На відміну від окремих графіків, які деталізують певний аспект, цей графік використовується як підсумкова панель якості обробки.

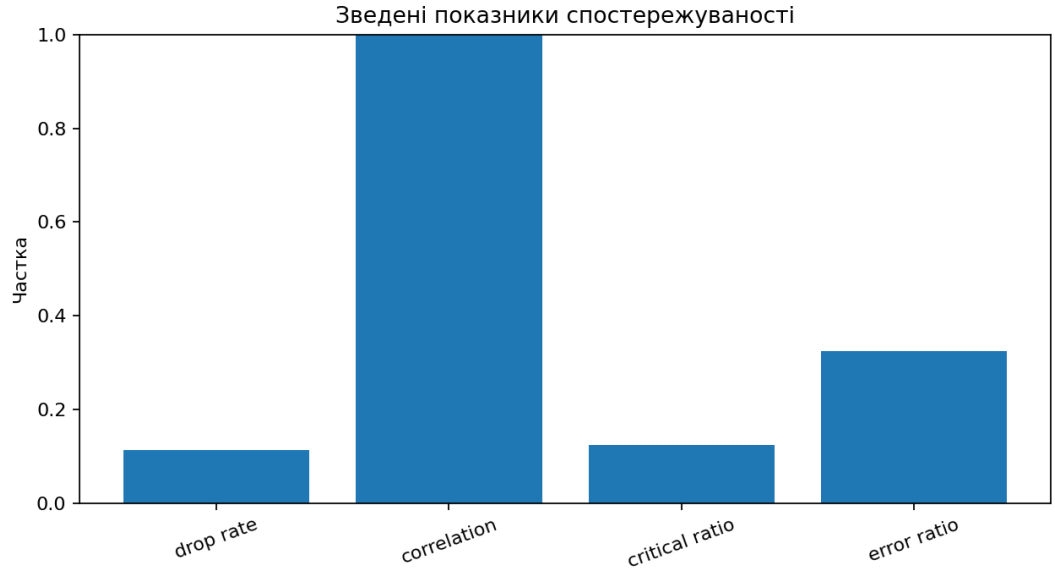

Рисунок 3.8 – Зведені показники спостережуваності

Результати тестування підтвердили, що прототип виконує основну інженерну задачу: він змінює поведінку обробки телеметрії залежно від профілю

системи. Embedded-події низької важливості не накопичуються, enterprise і cloud-події без кореляційного ідентифікатора не приймаються, чутливі значення не зберігаються відкрито, а summary дозволяє оцінити якість потоку. Це відповідає висновкам теоретичного огляду, де телеметрію було розглянуто як контрольований потік даних, а не як довільне накопичення журналів.

Разом із тим прототип має обмеження. По-перше, він працює локально й не перевіряє мережеву доставку в умовах реальних затримок, повторів і втрат пакетів. По-друге, SQLite-сховище придатне для навчального зразка, але не демонструє поведінку великомасштабного журналювання. По-третє, синтетичні дані поки мають сценарний характер і не моделюють повноцінні часові ряди, пікове навантаження або довгі ланцюги кореляції. По-четверте, інтерфейс виконує демонстраційну функцію і не містить розширеної візуальної аналітики.

Найважливішим напрямом посилення є розширення синтетичного генератора. Для повноцінного тестування потрібно створити фабрику подій із параметрами: кількість джерел, частка помилок, частка секретів, частка відсутніх correlation_id, розподіл severity, довжина повідомлень, частка повторів, часовий інтервал і профіль навантаження. Такий генератор даватиме можливість перевіряти не лише правильність одиничної події, а й стабільність системи на потоці, де виникають накопичення черг, повтори, перекоси розподілу та шум.

Другим напрямом є інтеграція тестів продуктивності. Для прототипу можна запровадити мікробенчмарки process_batch() для 100, 1000 і 10000 подій, а також вимірювання часу вставки в SQLite. Це дозволить порівняти, як змінюється вартість обробки для embedded, enterprise і cloud-політик. Третім напрямом є перевірка стійкості до помилок сховища: імітація заблокованої бази, неправильного шляху, пошкодженого JSON або переривання під час batch-вставки.

Четвертим напрямом є перевірка семантичної якості телеметрії. Навіть якщо всі тести проходять, система може накопичувати малокорисні записи. Тому варто ввести метрики якості: частка подій без кореляції, частка відкинутих записів за причинами, середній payload_size, частка замаскованих подій,

кількість унікальних джерел, частка повторів. Ці показники можуть стати основою для автоматичного попередження про деградацію самої телеметрії.

Загалом тестування показує, що розроблений прототип має достатню внутрішню цілісність для кваліфікаційної роботи. Його сильна сторона полягає в прозорому зв’язку між предметною проблемою, архітектурою, профільними політиками, синтетичними даними й тестами. Подальше розширення має бути спрямоване не на ускладнення інтерфейсу, а на посилення генератора даних, навантажувальні сценарії, підтримку реальних джерел телеметрії та глибшу перевірку якості спостережуваних даних.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано задачу дослідження, проєктування, реалізації та перевірки програмного прототипу системи журналювання й обробки телеметрії залежно від класу програмних систем. У роботі показано, що телеметрія не може розглядатися як однаковий набір службових повідомлень для всіх середовищ виконання. Її склад, деталізація, спосіб передавання, правила зберігання та аналітична цінність залежать від апаратних ресурсів, часових обмежень, мережевої доступності, критичності програмних функцій, вимог до безпеки та очікуваного способу подальшого аналізу.

У першому розділі проаналізовано предметну область журналювання, телеметрії та спостережуваності програмних систем. У межах теоретичного аналізу розглянуто основні класи інструментів журналювання та обробки телеметрії: бібліотеки формування журналів, схеми структурованих подій, агенти збирання даних, колектори, системи метрик, сховища журналів, засоби трасування, платформи візуалізації та інструменти інтелектуального аналізу. Виявлено, що жоден окремий інструмент не вирішує всієї проблеми спостережуваності. Якість телеметрії формується на всьому ланцюгу: від створення події в коді до її нормалізації, маршрутизації, зберігання, візуалізації та використання для аналізу. Тому в роботі обґрунтовано необхідність модульного підходу, у якому стратегія обробки залежить від класу системи та властивостей джерела сигналу.

У другому розділі виконано проєктування й реалізацію програмного прототипу. Визначено інженерні вимоги до системи, профілі телеметрії, структуру вхідних і нормалізованих подій, сценарії генерації синтетичних даних, логіку адаптації під джерела сигналів і правила обробки. Програмний прототип реалізовано як локальний модульний застосунок із серверним API, сховищем подій, генератором синтетичної телеметрії, конвеєром обробки, модулем метрик, побудовою графіків і користувацьким дашбордом. Реалізований прототип демонструє повний керований шлях телеметричної події: від створення або

приймання до нормалізації, фільтрації, збагачення, маскування, збереження, вимірювання та візуального аналізу.

Особливу увагу приділено адаптивності обробки телеметрії. У прототипі подія аналізується з урахуванням профілю системи, джерела сигналу, типу повідомлення, критичності, наявності кореляційного контексту та інших службових ознак. На цій основі формується рішення щодо класу джерела, пріоритету сигналу, політики обробки та причини прийнятого рішення. Це дозволяє продемонструвати відмінність між стратегіями телеметрії для малоресурсних, периферійних, корпоративних і хмарних систем. Реалізований конвеєр обробки виконує нормалізацію структури подій, застосування профільних правил, маскування чутливих атрибутів, визначення кореляційного статусу, приймання або відкидання подій і фіксацію причини такого рішення.

У третьому розділі виконано тестування та верифікацію програмного прототипу. Тестування було спрямоване не лише на підтвердження працездатності окремих функцій, а й на перевірку інженерних ризиків: втрати важливих подій через фільтрацію, витоку чутливих атрибутів у журналах, некоректного обчислення метрик, відсутності кореляційного контексту, помилок запису у сховище та невідповідності графіків фактичним даним. Для забезпечення відтворюваності перевірок використано синтетичні телеметричні події, які дали змогу контрольовано моделювати різні профілі програмних систем, рівні критичності, ситуації з неповним контекстом, події з чутливими атрибутами, різні сценарії приймання й відкидання записів. Контрольний запуск завершився успішним проходженням усіх перевірок, що підтверджує працездатність основних сценаріїв прототипу.

Розроблений прототип має контрольовані обмеження. Він працює в локальному режимі, використовує синтетичні або демонстраційні джерела подій, не інтегрований із промисловими платформами повного циклу спостережуваності та не реалізує повномасштабну систему виявлення відхилень. Водночас, модульна структура залишає можливість подальшого розвитку: підключення реальних агентів збирання журналів, інтеграції з OpenTelemetry

Collector, використання PostgreSQL, Loki, OpenSearch або сховищ часових рядів, додавання потокової обробки, розширення дашбордів і впровадження алгоритмів виявлення аномалій.

Отже, поставлену мету роботи досягнуто. Проведено теоретичний аналіз предметної області, обґрунтовано залежність журналювання та телеметрії від класу програмної системи, спроєктовано архітектуру програмного прототипу, реалізовано основні модулі обробки, створено механізми синтетичного тестування, метрики й графічне подання результатів, а також виконано автоматизовану перевірку працездатності. Отримані результати підтверджують, що адаптивна обробка телеметрії є практично важливою для програмних систем різних класів, оскільки дозволяє узгоджувати повноту спостереження, ресурсні витрати, безпеку, діагностичну цінність і можливість подальшого аналізу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes. Geneva : International Organization for Standardization, 2017. URL: https://www.iso.org/standard/63712.html (дата звернення: 04.06.2026).
2. ISO/IEC/IEEE 42010:2022. Software, systems and enterprise — Architecture description. Geneva : International Organization for Standardization, 2022. URL: https://www.iso.org/standard/74393.html (дата звернення: 04.06.2026).
3. OpenTelemetry. OpenTelemetry Specification. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/specs/otel/ (дата звернення: 06.06.2026).
4. OpenTelemetry. Signals. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/concepts/signals/ (дата звернення: 04.06.2026).
5. Cândido J., Aniche M., van Deursen A. Log-based software monitoring: a systematic mapping study. PeerJ Computer Science. 2021. Vol. 7. Article e489. DOI: 10.7717/peerj-cs.489.
6. Kratzke N. Cloud-Native Observability: The Many-Faceted Benefits of Structured and Unified Logging — A Multi-Case Study. Future Internet. 2022. Vol. 14, No. 10. Article 274. DOI: 10.3390/fi14100274.
7. Li T. et al. Log2Evt: Constructing high-level events for IoT Systems by correlating logs and runtime contexts. Journal of Systems Architecture. 2025. URL: https://www.sciencedirect.com/science/article/abs/pii/S1383762125002504 (дата звернення: 04.06.2026).
8. Stouffer K., Pease M., Tang C. et al. Guide to Operational Technology (OT) Security : NIST Special Publication 800-82 Revision 3. Gaithersburg : National Institute of Standards and Technology, 2023. URL: https://csrc.nist.gov/pubs/sp/800/82/r3/final (дата звернення: 04.06.2026).
9. Monitoring and Observability in Edge Computing Systems. Computers, Materials & Continua. 2026. URL: https://www.techscience.com/cmc/online/detail/26962 (дата звернення: 04.06.2026).

10. Grafana Labs. Observability Survey Report 2025: Key findings. Grafana Labs, 2025. URL: https://grafana.com/observability-survey/2025/ (дата звернення: 04.06.2026).
11. De la Cruz Cabello M., Prince Sales T., Machado M. R. AIOps for log anomaly detection in the era of LLMs: A systematic literature review. Intelligent Systems with Applications. 2025. Article 200608. DOI: 10.1016/j.iswa.2025.200608.
12. New Relic. 2024 Observability Forecast Report. New Relic, 2024. URL: https://newrelic.com/sites/default/files/2024-10/new-relic-2024-observability-forecast-report.pdf (дата звернення: 04.06.2026).
13. OpenTelemetry. Logs Data Model. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/specs/otel/logs/data-model/ (дата звернення: 04.06.2026).
14. IoT Analytics. Number of connected IoT devices growing 14% to 21.1 billion globally. 2025. URL: https://iot-analytics.com/number-connected-iot-devices/ (дата звернення: 04.06.2026).
15. OpenTelemetry. Logs API. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/specs/otel/logs/api/ (дата звернення: 04.06.2026).
16. OpenTelemetry. Collector Architecture. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/collector/architecture/ (дата звернення: 06.06.2026).
17. OpenTelemetry. Collector Configuration. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/collector/configuration/ (дата звернення: 04.06.2026).
18. Fluent Bit. Fluent Bit Documentation: High Performance Telemetry Agent for Logs, Metrics and Traces. URL: https://docs.fluentbit.io/manual (дата звернення: 04.06.2026).
19. Vector. Introduction: Observability Data Pipeline. URL: https://vector.dev/docs/introduction/ (дата звернення: 06.06.2026).

20. Elastic. ECS formatted application logs. Elastic Docs. URL: https://www.elastic.co/docs/solutions/observability/logs/ecs-formatted-application-logs (дата звернення: 04.06.2026).
21. Prometheus. Overview. Prometheus Documentation. URL: https://prometheus.io/docs/introduction/overview/ (дата звернення: 04.06.2026).
22. Grafana Labs. Grafana Loki documentation. URL: https://grafana.com/docs/loki/latest/ (дата звернення: 01.06.2026).
23. Elastic. Elastic Observability solution overview. Elastic Docs. URL: https://www.elastic.co/docs/solutions/observability (дата звернення: 01.06.2026).
24. OpenSearch. Observability. OpenSearch Documentation. URL: https://docs.opensearch.org/latest/observing-your-data/ (дата звернення: 01.06.2026).
25. Elastic. Elastic Common Schema (ECS) reference. Elastic Docs. URL: https://www.elastic.co/docs/reference/ecs (дата звернення: 01.06.2026).
26. W3C. Trace Context. W3C Recommendation. 2021. URL: https://www.w3.org/TR/trace-context/ (дата звернення: 01.06.2026).
27. Jaeger. Jaeger Tracing Documentation. URL: https://www.jaegertracing.io/ (дата звернення: 01.06.2026).
28. OpenTelemetry. Semantic Conventions. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/concepts/semantic-conventions/ (дата звернення: 01.06.2026).
29. OpenTelemetry. Collector Processors. OpenTelemetry Documentation. URL: https://opentelemetry.io/docs/collector/components/processor/ (дата звернення: 01.06.2026).
30. Prometheus. Metric and label naming. Prometheus Documentation. URL: https://prometheus.io/docs/practices/naming/ (дата звернення: 04.06.2026).
31. FastAPI. FastAPI Documentation. 2026. URL: https://fastapi.tiangolo.com/ (дата звернення: 04.06.2026).

32. Pydantic. Pydantic Documentation. 2026. URL: https://docs.pydantic.dev/ (дата звернення: 04.06.2026).
33. SQLite. SQLite Documentation. 2026. URL: https://www.sqlite.org/docs.html (дата звернення: 04.06.2026).
34. pytest. pytest Documentation. 2026. URL: https://docs.pytest.org/ (дата звернення: 04.06.2026).
35. ISO/IEC/IEEE 29119-1:2022. Software and systems engineering - Software testing - Part 1: General concepts. Geneva : International Organization for Standardization, 2022.
36. Brown S. The C4 model for visualising software architecture. URL: https://c4model.com/ (дата звернення: 04.06.2026).
37. Object Management Group. OMG Unified Modeling Language Version 2.5.1. 2017. URL: https://www.omg.org/spec/UML/2.5.1/ (дата звернення: 04.06.2026).
38. OWASP Foundation. Logging Cheat Sheet. OWASP Cheat Sheet Series. 2026. URL: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html (дата звернення: 04.06.2026).
39. OWASP Foundation. Secrets Management Cheat Sheet. OWASP Cheat Sheet Series. 2026. URL: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html (дата звернення: 04.06.2026).
40. Python Software Foundation. logging - Logging facility for Python. Python Documentation. 2026. URL: https://docs.python.org/3/library/logging.html (дата звернення: 04.06.2026).
41. OpenTelemetry. OpenTelemetry Collector Architecture. OpenTelemetry Documentation. 2026. URL: https://opentelemetry.io/docs/collector/architecture/ (дата звернення: 04.06.2026).
42. Grafana Labs. Grafana Loki Documentation. 2026. URL: https://grafana.com/docs/loki/latest/ (дата звернення: 04.06.2026).

ДОДАТКИ

Додаток А – Розгорнутий тест-план програмного прототипу

Таблиця А.1 – Опис тестових випадків

ID	Область	Назва тесту	Вхідні умови	Очікуваний результат
TC-01	normalize	Наївна часова мітка	Подія з datetime без timezone	timestamp отримує UTC
TC-02	embedded	Info-подія embedded	severity=info	status=dropped, severity_below_warning
TC-03	embedded	Warning-подія embedded	severity=warning	status=accepted
TC-04	embedded	Сенсорний payload	event_type=sensor	payload_reduced=True
TC-05	enterprise	Відсутній correlation_id	correlation_id=None	status=dropped
TC-06	enterprise	Корельована подія	correlation_id=req-42	status=accepted
TC-07	cloud	Debug із trace	severity=debug, correlation_id=trace	status=accepted
TC-08	cloud	Debug без trace	severity=debug, correlation_id=None	status=dropped
TC-09	policy	Довге повідомлення	message=200 символів	довжина обмежена
TC-10	security	Token і password	payload із token/ password	значення замасковані
TC-11	security	Email у payload	person@example.com	***@***
TC-12	security	Номер картки	4111 1111 1111 1111	****-****-****-****
TC-13	security	Числовий ідентифікатор	device-123456789	часткове маскування
TC-14	embedded	Обмеження кількості ключів	payload 10 ключів	_truncated_keys у payload
TC-15	cloud	Cloud без обмеження payload	payload 20 ключів	без _truncated_keys
TC-16	simulation	Генератор кожного профілю	profile in SystemProfile	4 події
TC-17	simulation	Обробка demo batch	demo_events(profile)	без винятків
TC-18	severity	Поріг embedded debug	severity=debug	False
TC-19	hash	Стабільний hash	дві однакові події	однаковий event_id
TC-20	hash	Зміна hash	зміна message	різний event_id
TC-21	API	Список профілів	GET /api/profiles	4 профілі
TC-22	API	Валідна enterprise-подія	POST /api/events	status=accepted
TC-23	API	Невалідний payload	неповний JSON	HTTP 422
TC-24	API	Пакет із 2 подій	POST /api/batch	processed=2, accepted=1
TC-25	API	Embedded demo	POST /api/demo/embedded	generated=4
TC-26	API	Summary після demo	POST demo + GET summary	total=4
TC-27	API	Список подій	GET /api/events?limit=2	2 записи
TC-28	API	Фільтр dropped	GET /api/events?status=dropped	є missing_correlation_id
TC-29	API	Очищення сховища	DELETE /api/events	total=0

Продовження таблиці А.1

TC-30	API	Головна сторінка	GET /	HTML містить назву системи
TC-31	Storage	Вставка і список	insert_many + list_events	1 запис
TC-32	Storage	Очищення	clear()	list_events=[]
TC-33	Storage	Фільтр profile	profile=embedded	1 embedded-запис
TC-34	Storage	Фільтр status	status=dropped	1 dropped-запис
TC-35	Storage	Підрахунок статусів	accepted+dropped	summary accepted=1 dropped=1
TC-36	Storage	Метрики якості	payload із token	masked_payload_ratio=1
TC-37	Synthetic	Embedded шум	debug/info серія	події нижче warning відкидаються
TC-38	Synthetic	IoT втрата мережі	buffered_events збільшується	warning зберігається
TC-39	Synthetic	Enterprise аудит	audit event з authorization	authorization замасковано
TC-40	Synthetic	Cloud trace	кілька сервісів trace-1	події групуються за correlation_id
TC-41	Negative	Порожній source	source=""	HTTP 422
TC-42	Negative	Невідомий profile	profile=desktop	HTTP 422
TC-43	Load	Пакет 100 подій	batch з повторюваними подіями	без помилки вставки
TC-44	Regression	Повторний seed	однаковий seed	однакова структура пакета
TC-45	Security	Вкладений secret у списку	payload зі списком словників	secret замасковано
TC-46	Quality	Події без кореляції	мікс enterprise/ cloud	correlation_coverage зменшується

Додаток Б – Посилання на репозиторій

Код програмного забезпечення, розробленого в межах кваліфікаційної роботи, розташований за адресою https://github.com/nikonovir/qualification-work-.