# титульний аркуш

Monitoring and observability of distributed systems in the context of preventive safety

# завдання на бакалаврську роботу

TODO:

* реквестнути в керівника

# Анотація

Моніторинг та оглядовість включають в себе збір, аналіз та візуалізацію різноманітної динамічної інформації стосовно певного процесу. У сучасних розподілених системах вже доступна велика кількість напрацювань, аби полегшити ці три нетривіальні завдання, проте моніторинг досі одна з найскладніших та найважливіших частин моделювання програмних продуктів у production середовищах. Робота розкриває причини такого широкого поширення розподілених систем серед розробників та тих, хто моделює системи, наголошуючи на доступності таких практик як DevOps, SRE, тощо.

Робота також визначає характеристики розподілених систем, що роблять їх моніторинг таким нетривіальним. Ці характеристики використані задля розуміння методології моніторингу та порівняння доступних досліджень на цю тематику.

Сама робота має на меті дослідити важливість моніторингу та оглядовісті в сучасних розподілених системах, зважаючи на доменну область системи в якій застосовані, а також дослідити збір, аналіз, та презентацію операційних даних в розподілених системах задля запобігання критичним ситуаціям. Окрім технічних аспектів моніторингу, робота також оцінює моніторинг з точки зору менеджера проєктів, описуючи підходи до SLA, культури моніторингу, інцидентів, тощо. Так само, робота досліджує інциденти, що стаються та методологію відновлення від них, з наголосом на те, як це впливає на запобігання наступним інцидентам. Так само, робота розкриває ціну моніторингу на різних етапах життєвого циклу програмного продукту, з наголосом на ціну моніторингу саме в production середовищах.

Кожна частина цього дослідження буде розглянута як концептуально, так і на практиці, підтверджуючи теорію прикладами створених систем та їх оглядовості: метрик, сповіщень, логування, графіків, повідомлень, відлагодження, трасування. Ці приклади виконані з урахуванням доступних сучасних інструментів таких як Prometheus, .NET, Kibana, ElasticSearch, Logstash, Zipkin, OpenTelemetry, та інші.

Інформація подана в цій роботі базується на досвіді дослідника стосовно розробки та проектування розподілених систем, а також на досвіді підтримки таких систем в production середовищах з фокусом на стабільність та безпеку.

Ключові слова: MONITORING, OBSERVABILITY, DISTRIBUTED SYSTEMS, SOFTWARE SAFETY, SOFTWARE ENGINEERING, PREVENTIVE SAFETY IN SOFTWARE

# Анотація англійською.

Monitoring and observability for distributed systems include collecting, analyzing, and presenting various dynamic data regarding some particular processes running in the distributed system. In a modern distributed systems environment, many works and practices are available for making these non-trivial tasks more manageable, even though monitoring remains one of the most complicated aspects to tackle while modeling production-ready software. The work exposes reasons behind such wide adoption of distributed systems for modern industry workloads, focusing on such practices as DevOps and SRE, making it so accessible for a broad audience.

The work also defines characteristics of distributed systems that make monitoring so complicated. These characteristics are used for understanding the methodology of monitoring and comparing related papers.

The main goal of the work is to examine the importance of monitoring and observability in the modern distributed system, taking into account the domain where applied. Also, to investigate collecting, analyzing, and presenting operational data in distributed systems for preventing critical situations, i.e., incidents. Besides technical challenges, the works also evaluate the monitoring from a product ownership point of view, describing the approaches to SLAs, monitoring culture mindset, incidents, et cetera. Also, incidents are discussed in work, focusing on recovering and learning from them, most importantly preventing them in the future. Then, the price of monitoring is discussed at various stages of the software product lifecycle, especially in production environments.

Also, the software product is built based on this study which shows the usage of modern monitoring best practices. All of the aspects are covered both conceptually and practically: metrics, alerts, logging, dashboards, messages, debugging, tracing. The software product is built for the reader's better understanding of the concepts described. Technologies used to create this sample include but are not limited to .NET Core, Prometheus, Docker, Docker-Compose, Kibana, Logstash, Zipkin, OpenTelemetry.

The information given in this thesis is based on the researcher's experience regarding building and architecting distributed systems and running and supporting such systems in production environments with a strong focus on reliability, stability, and safety.

Keywords: MONITORING, OBSERVABILITY, DISTRIBUTED SYSTEMS, SOFTWARE SAFETY, SOFTWARE ENGINEERING, PREVENTIVE SAFETY IN SOFTWARE

# Зміст

# Перелік умовних позначень, символів, одиниць, скорочень і термінів

Production система

serverless функції

SLA (37 сре бук) AOP

Docker Prometheus

Happy-path сценарій –

JSON

Аспектно Орієнтовне Програмування -

CPU - Кешування - Apache License 2.0 - Cloud Native Computing Foundation -

# Вступ

Ця робота організована наступним чином: розділ 1 описує область дослідження - розподілені системи, а також знайомить читачів з поняттями моніторингу. У розділі 3 описано підходи до створення програмних продуктів, чому сучасні системи будують як розподілені, та які основні проблеми мають такі системи. Моніторинг для розподілених систем презентований у розділі 4. Превентивна інструменти моніторингу висвітлені у розділі 5. Розділ 6 описує економічну частину роботи. Розділ 7 присвячений висновкам.

## Дотичні дослідження

Дослідження проблем навколо оглядовості розподілених систем ведуться ще з ранніх часів паралельних обчислень [1]. Згідно з IEEE, термінологія моніторингу та оглядовості бере свої початки в теорії контролю систем [3] та вимірює рівень відносоно якого за системою можна спостерігати, іншими словами - як внутрішній стан системи може бути визначеним через зовнішній стан.

Більшість таких класичних досліджень були сфокусуваними на створенні власних моніторингових систем, що були покликані забезпечити розуміння стабільності розподіленої системи. [2]

Окрім стандартних підходів до моніторингу, дуже важливим є передбачати продуктивність розподілених систем.  
Johng et al. [11] досліджуючи тематику, пропонує два підходи до проблеми моніторингу - бенчмаркінг та симуляція роботи розподілених систем. Ці підходи виявляються оптимальними та раціональними, швидкими, дешевими та надійними. Так само, Lin et. al [14] пропонує новий шлях дослідження та знаходження причин інцидентів у мікросервісних архітектурах за допомогою візуалізації.

В хмарних середовищах, оглядовість визначається за таким же принципом, проте зовнішнім станом можуть бути логи, метрики, трейси, тощо. Саме вони використовуються для визначення рівня оглядовості.

Існують також сучасні роботи, що розглядають підходи для моніторингу виключно розподілених систем [10]. Базуючись на цьому [10], Mostafа та Bonakdarpour  [11] адаптують підходи для розпізнавання аномалій за певними властивостями, що можна вважати некоректною поведінкою.

Є також сучасні дослідження, які розкривають аспекти моніторингу, що є прикладними до сьогодення, описуючи інструменти та підходи, що вже використовуються серед розробників систем. Гарним прикладом такого дослідження є "On Observability and Monitoring of Distributed Systems – An Industry Interview Study" від Sina Niedermaier, Falko Koetter, Andreas Freymann, Stefan Wagner [12]. Такі дослідження менше фокусуються на фундаментальних концепціях, натякаючи, що читачі вже знайомі з усією термінологією, а основою дослідження є прикладні історії та висновки з індустрії.

Превентивний аспект моніторингу розподілених систем не був досі широко висвітленим серед доступних досліджень, тому й ця робота покликана заповнити цю прогалину, аналізуючи існуючі дотичні роботи, та осмислюючи як моніторинг може запобігати інцидентам та різноманітним неочікуваним проблемам.

## Розподілені системи

Областю дослідження в цій роботі однозначно є розподілені системи. Перед тим, як ми заглибимось в підходи, термінологію, інструменти, важливим є окреслити сучасну суть розподілених систем, та як вони еволюціонували з часом, адже поняття вже є сталим в індустрії ще з 1980-х років [4]. Тим не менше, можна стверджувати, що визначення розподілених систем з 1980-х цілком може підійти й для сьогодення. Розподілена система складається з певної кількості процесів, що працюють на різних серверах та виконують спільну роботу. Очевидно, що ця спільна робота є розподіленою між серверами задля оптимальних сценаріїв виконання. Ці процеси мусять координуватись між собою, комунікуючи за допомогою протоколів комунікації (напр. HTTP, AMQP, gRPC, тощо) [13;14;15]. Системи є розподіленими за своїм дизайном ще здавна, адже наприклад багатопотічна програма є також розподіленою системою, хоч і робота в ній розподілена поміж потоками та ядрами, а не між серверами.

В рамках останнього десятиліття (2010-2020) розподілені системи стали трендом в розробці високонаватажених систем. Варто зазначити, що сучасні розподілені системи є найбільш розподіленими, яких лише бачив світ, адже вони захоплюють різні сервери, хмарні провайдери, розташовані в різних країнах, контрольовані різними компаніями, виконують різноманітні програми, що базуються на різних платформах, тощо. Розподілені системи мають певний список переваг перед традиційними системами:

* зменшена вартість інфраструктури
* покращена надійність та доступність
* легкість виокремлення модулів та їх компонування.
* гнучкість в конфігурації
* можливість поступового росту по покращень системи

Навіть коли кожна окрема підсистема є тривіальною та простою, інтеграція цих компонентів разом може створювати неймовірну складність. Це саме по собі створює низку проблем, яких не було в традиційних системах, що працювали в рамках одного сервера. Кількість задіяних частин збільшує статистичну ймовірність помилки.

Ці проблеми включають в себе, але не обмежуються:

* Різноманітні компоненти можуть неочікувано вимикатися незалежно один від одного, непередбачувано реагувати на проблеми в інших частинах системи
* Комунікація між частинами системи може бути провальною, в тому числі й через тимчасові проблеми мережі. В таких середовищах, якість мережі та її стабільність відіграють важливу роль.
* Якщо мережеві проблеми стаються, то починаються повторення запитів / повідомлень, що може спричиняти неочікуваний та некоректний стан системи у кінці-кінців
* Гетерогенність систем, адже вони можуть складатися з кількох рівнів - код, що виконує бізнес-сценарії, інфраструктурні технології такі як Docker контейнери, віртуальні машини, або навіть serverless функції. Більше того, команди для таких систем також можуть бути гетерогенними. В той час, коли таке середовище може культивувати іновації, дозволяючи різним командам вибирати різні технології для своїх частин системи, з іншого боку це ж і створює проблематику для консистентного підходу до моніторингу таких систем.

### Розподілені системи та моніторинг

Відповідно, відслідковування трендів, та огляд розподілених обчислень й систем є важливою проблемою ще з ранніх часів розподілених систем. [1].  
Ще в 1987 Joyce [7] ідентифікував п'ять аспектів в моніториннгу розподілених систем, що зараз можна вважати ранньою спробою виділити характеристики основних проблем, що відрізняють моніторинг монолітних систем від розподілених систем. Ось ці аспекти:

* Розподілені системи мають набагато більше точок, де можна контролювати їх. В деяких ситуаціях, настільки багато, що це видається заледве реалістичним охопити усі.
* Існування недетерміністичної затримки у комунікації між частинами розподіленої системи, адже це унеможливлює детерміністичні твердження про стан системи в даний момент часу
* Унаслідуваний нон-детермінізм від визначення розподілених та асинхронних систем
* Моніторинг розподілених систем змінює їхню поведінку. Подібне явище описане у контексті фізики - ефект спостерігача [5].
* Складність інтеракції між системою та розробниками системи.

Розглядаючи питання доступності розподілених систем та їхньої популярності, варто згадати про усі інструменти, що стають доступними з технічним прогресом. Одним з фундаментальних інструментів є Docker, що дозволяє ізолювати роботу програмного продукту, що, в свою чергу, забезпечує більше детермінізму при переносі Docker image (що є програмним артефактом) у інше середовище. Так само, варто згадати постійно-ростучу популярність хмарних рішень, які значно полегшують старт програмних продуктів. За своєю суттю, хмарні рішення є презентацією розподіленої системи, адже рішення є створеними за допомогою комбінування окремих компонентів. Якщо до ранніх 2010-х років хмарні обчислення були занадто недоступними через ціну, то зараз вони є настільки дешевими, що є набагато вигіднішими, аніж власні дата-центри. Тим не менше, ростуча популярність хмарних обчислень, великих даних, оркестраторів змусила переосмилювати моделювання моніторингу розподілених систем, адже самі розподілені системи змінились. Основною зміною було розділяти систему на набагато менші, більш конкретні підсистеми. Така зміна має прямий вплив на те, що треба моніторити, адже збільшує кількість компонентів, що варто моніторити. В кінці кінців, така кількість компонентів може лише створювати великі об'єми інформаційного шуму замість цінних даних.

З тим як розподілені системи стають все більш поширеними серед розробників та тих, хто проектує системи, постала логічна потреба в моніторингу таких систем, адже кожен розробник та менеджер хочуть мати якомога більше розуміння та контролю над такою системою задля покращення її якостей. Тому, за такими системами необхідно слідкувати у всіх стадіях їхнього життєвого циклу - від розробки до щоденної підтримки production системи. А із тим як змінюється область застосування моніторингу мусять змінюватись й інструменти, адже їхні цілі лишаються незмінними - зрозуміти, де знаходяться проблеми з продуктивністю, звідки беруться помилки, вчасно ідентифікувати інцидент, сповістити розробників розподіленої системи, тощо.

### Запобігаючи інцидентам у розподілених системах

Дана робота розглядає моніторинг як інструмент задля запобігання інцидентам у розподілених системах. Основним фокусом роботи є розкрити різні аспекти моніторингу в якості інструмента забезпечення стабільності розподілених систем. У роботі є розглянуті такі теми:

* першочергові проблеми в програмних продуктах, що змушують замислитись про моніторинг
* основні компоненти моніторингу, що є в наявності задля запобігання інцидентам
* приклад реалізації даних компонентів задля ілюстративності
* ціна моніторингу у розподілених системах
* превентивний аспект моніторингу та інструменти збільшення відсотку інцидентів, яких вдалось запобігти превентивно

Описуючи дану проблематику, робота осмислює важливість моніторингу в розподілених системах задля запобігання інцидентам, а також пропонує наступні кроки, що можуть бути ідеями для подальших досліджень в даному напрямку.

# Основна частина

### Проблеми з програмними продуктами на стадії підтримки

Можна виділити кілька типів проблем із програмними продуктами на стадії підтримки - операційні проблеми, програмні проблеми. До операційних проблем ми можемо віднести мережеві проблеми (у всій країні вимкнули інтернет), перегрів CPU через спеку, проблеми з hardware - диск перестав працювати, бо на нього впав камінь в серверній - це все проблеми, які варто вирішити за допомогою зовнішнього впливу. Цим зовнішнім впливом можуть стати такі прості речі як переставити сервер в прохолодний підвал, або замінити привід диску.

Є ще один тип проблем - це програмні проблеми. Простіше кажучи, це проблеми спричинені тими чи іншими розробниками програм. Це проблеми всередині логіки програми. Аби вирішити такі проблеми, розробники мусять до інциденту зібрати якомога більше інформації стосовно того, що відбувалось в системі, і це дозволить їм сформувати ланцюжок причинно-наслідкових зв'язків, які допоможуть їм знайти джерело проблеми та виправити його. В таких ситуаціях моніторингові дані - це критична необхідність, а не додатковий аспект системи. Даними можуть бути різноманітні метрики, логи, трейси, тощо.

Описуючи проблеми з програмними продуктами, необхідно зазначити, що природа цих проблем не є бінарною, тобто не лише існує дві опції "працює" та "не працює". Навпаки, складні розподілені системи можуть працювати в такому стані, де продуктивність є зниженою через ті чи інші фактори. Такі стани зменшеної продуктивності дуже часто можуть створювати проблеми, які важко відлагодити. І саме моніторингові дані є єдиним ключем до відлагодження таких проблем у розподілених системах.

Парадоксально, проте чим більшу інвестицію проєкт робить в оглядовість, тим менше часу доведеться за ним наглядати, тим більше часу можна буде виділити на розробку проекту.

Одним з важливих завдань в моделюванні оглядовості для розподілених систем є створення середовища, де розробники могли б приймати рішення базуючись на інформації (метрики, сповіщення, тощо) про стан системи. Інакше, якщо при моделюванні цей аспект врахований не буде, то розробникам в певний момент будуть очевидними проблеми із системою (скарги користувачів, витоки даних, некоректно відпрацьовані сценарії, тощо), проте це вже буде занадто пізно - шкода буде зроблена для користувачів та зацікавлених осіб, що може бути критичним. Саме в цьому й полягає основне завдання оглядовості розподілених систем - попередити критичну ситуацію, аби розробники могли прийняти рішення і не дати цьому статися. Саме це й можна вважати превентивною безпекою в даному випадку.

##### Моніторинг та його компоненти

Деякі з описаних вище проблем можна було б вирішити за допомогою сучасних підходів до моніторингу.

Цей розділ описує 3 основні етапи з яких і складається моніторинг розподілених систем - збір даних, обробка та презентація.

**Створення та збір даних**: Релевантні дані з системи мусять бути сформовані в зрозумілий формат для системи збору цих даних. Наприклад, Prometheus збирає дані з сервісу і репрезентує їх через endpoint у наступному форматі:

|  |
| --- |
| # HELP process\_private\_memory\_bytes Process private memory size |
| # TYPE process\_private\_memory\_bytes gauge |
| process\_private\_memory\_bytes 74604544 |
| # HELP process\_virtual\_memory\_bytes Virtual memory size in bytes. |
| # TYPE process\_virtual\_memory\_bytes gauge |
| process\_virtual\_memory\_bytes 2223070081024 |
| # HELP process\_start\_time\_seconds Start time of the process since unix epoch in seconds. |
| # TYPE process\_start\_time\_seconds gauge |
| process\_start\_time\_seconds 1576244939.1073897 |
| # HELP dotnet\_total\_memory\_bytes Total known allocated memory |
| # TYPE dotnet\_total\_memory\_bytes gauge |
| dotnet\_total\_memory\_bytes 3013928 |
| # HELP process\_cpu\_seconds\_total Total user and system CPU time spent in seconds. |
| # TYPE process\_cpu\_seconds\_total counter |
| process\_cpu\_seconds\_total 0.796875 |
| # HELP dotnet\_collection\_count\_total GC collection count |
| # TYPE dotnet\_collection\_count\_total counter |
| dotnet\_collection\_count\_total{generation="1"} 0 |
| dotnet\_collection\_count\_total{generation="2"} 0 |
| dotnet\_collection\_count\_total{generation="0"} 0 |
| # HELP process\_working\_set\_bytes Process working set |
| # TYPE process\_working\_set\_bytes gauge |
| process\_working\_set\_bytes 56242176 |
| # HELP process\_num\_threads Total number of threads |
| # TYPE process\_num\_threads gauge |
| process\_num\_threads 35 |
| # HELP process\_open\_handles Number of open handles |
| # TYPE process\_open\_handles gauge |
| process\_open\_handles 566 |

*Рисунок 1. Приклад даних в форматі для Prometheus*

Відповідно, ці дані вже відповідають очікуваному формату, і можуть бути зібрані. Так само, варто згадати, що збір даних має забирати якомога менше ресурсів у production системи та не заважати її нормальному функціонуванню.

**Обробка**:

В деяких випадках даних може бути більше, аніж достатньо, але великого інформаційного навантаження вони не несуть - напр. з певних сервісів немає сенсу збирати щосекунди інформацію про навантаженість їхнього CPU. В таких випадках, може бути застосовний такий підхід як адаптивний семплінг. Це зменшить кількість даних, які зберігаються на рівні сховища даних, проте це не буде критичним для розуміння стану системи. Простіше кажучи, деякими даними моніторинг система готова жертвувати, адже вони не несуть нової інформації для користувачів, а тому не такими, що достойні займати ресурси. Такі зміни впливають на кількість підсистем, які може моніторити та обслуговувати моніторинг система.

Дані зберігаються в сховищі даних, яке є оптимізованим для такого типу навантаження - великого об'єму даних, де багато даних додається щосекунди, проте набагато менше читається. Тим не менше, дані мають бути оптимізовані таким чином, аби їх можна було легко презентувати з урахуванням певних фільтрів. Це означає, що необхідно індексувати дані та підтримувати відповідні індекси при кожному оновленні бази даних. Прикладом сховища даних може бути ElasticSearch в такому випадку.

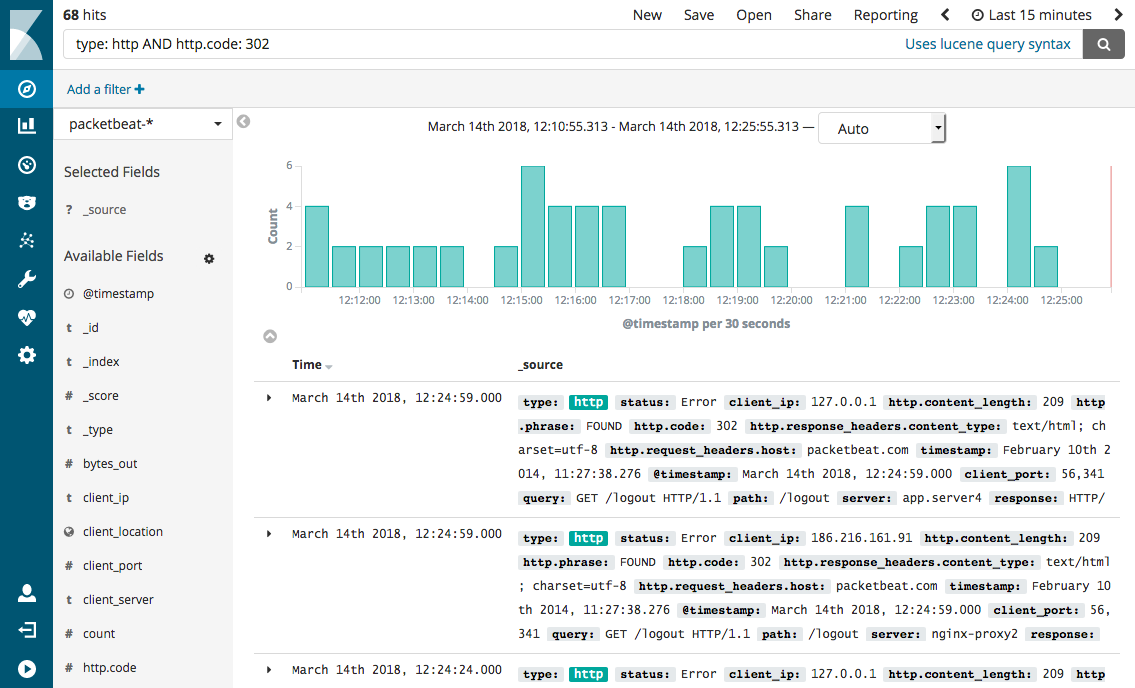
Graphical user interface, text, application

Description automatically generated

*Рисунок 2. Приклад зберігання даних в ElasticSearch.*

Презентація: Це етап, де дані потрапляють до користувача моніторингової системи. Зібрані та проаналізовані дані представляються користувачам в формі, яка є доречною для кожного випадку. Зважаючи на те, що розподілена система може створювати величезні обсяги інформації для аналізу та презентації, користувач моніторинг системи не в стані справитися з таким когнітивним навантаженням. Тому існує необхідність оптимізовувати ці дані до такого вигляду, що буде релевантним користувачу. Саме для цього існує фільтрування даних. Фільтрування даних - це процес мінімізації даних таким чином, що користувач моніторингової системи зможе отримати лише необхідні дані для його рівня сприйняття (напр. отримати інформацію про все, що відбувалось на конкретному сервері за останні 15 хвилин). Варто зазначити, що дані досі лишаються в сховищі даних, проте користувацький інтерфейс показує лише те, що користувач робить запит.

Так само, дуже важливо мати інструменти візуалізації, що допоможуть зменшити когнітивне навантаження на користувача, не зменшуючи кількість інформації, яку він може оглянути. Більше того, при коректній візуалізації, користувач має можливість ознайомитись з даними за довгі періоди часу, які адаптовані для легшого сприйняття в стислі рамки часу. Приклади імплементації візуалізації моніторингових даних можна побачити в розділі ...



*Рисунок х. Приклад фільтрації даних (логів) в Kibana. Фільтрація за типом запиту http та кодом відповіді 302.*

Ознайомившись з етапами моніторингу, обов'язковим є наголосити на важливості даних. В розподілених системах виділяють три основні типи даних, що роблять можливим моніторинг такої системи:

* Логи
* Метрики
* Трейси

Також, цю трійку компонентів називають the three pillars of observability [6]. Детальніше про кожен з типів даних є описано в наступних підрозділах.

###### Логи

Лог - це незмінний запис в моніторинговій системі, що описує факт, який стався в системі. Важливо зазначити, що цей запис має часову позначку, що презентує момент, коли цей факт стався.

Логи можна поділити на два типи: текстові та структуровані. Текстові є найбільш поширеними; це лінійка тексту, що репрезентує якийсь факт. Структуровані є менш поширеними, але набирають все більше популярності, адже дозволяють структурований пошук та фільтрування по полях лог запису, який презентований в форматі JSON. Обидва є реалізовані в даній роботі, та приклади їх наведені в розділі ...

Стосовно ролі логування варто зазначити, що вони є ключовими інструментами у відлагодженні систем. Особливо, це є важливим в рамках розподілених систем, адже події та логи можуть відбуватись одночасно на рівні кількох підсистем, проте система моніторингу може агрегувати такі логи та показувати їх в легкому для користувацького сприйняття порядку. Так само, дуже рідко в розподілених системах стаються інциденти через проблему в одному специфічному компоненті. У більшості випадків, це один або кілька компонентів, що не змогли відпрацювати коректно, а згодом прокомунікували некоректні результати своєї роботи по ланцюжку. Саме в таких випадках дуже доречним є екстенсивне логування, де будь-яке місце, де або приймається рішення, або дані проходять через рівень комунікації, логується. Відповідно, у системах, де логуються вхідні та вихідні параметри, вони можуть бути використані для повторного виконання таких запитів та відлагодження реації системи на вхідні параметри в ізольованому середовищі. У арсеналі сучасних підходів до логування так само існує Аспектно Орієнтовне Програмування [19], що дозволяє робити логування вхідних параметрів та результуючих значень автоматично [18], приховано від очей того, хто досліджує кодову базу, аби зменшити когнітивне навантаження на нього.

###### Метрики

Метрики є цифровою репрезентацією даних, які були виміряні за певний період часу. Вони дозволяють застосовувати математичне моделювання та статистику, аби передбачувати поведінку системи, що дає додаткову інформацію, яка впливає на рішення розробників, які підтримують систему.

Зважаючи на те, що метрики є вже певною мірою опрацьованими даними, то це дозволяє легко будувати дашборди, базуючись на них, адже ці дашборди будуть репрезентативними для розуміння історичних трендів, іншими словами - того що ставалося із системою напротязі певного періоду часу.

Гарним прикладом метрик можуть бути завантаженість CPU, кількість пам'яті, що виділяється на систему, або кількість активних потоків. Приклад метрик, що збираються, наведено в розділах ... та ..., де в останньому додатково описується імплементація підходу, що дозволяє збирати метрики.

###### Трейси

Одним з найбільш складних аспектів в розподілених системах, де існує велика кількість компонентів, є розуміння того, як усі компоненти співпрацюють один з одним. Особливо важливим це стає при відлагодженні системи, адже, базуючись на існуючих даних (логах, метриках) необхідно ізолювати частину системи, що є джерелом інциденту. Зважаючи на те, що кожен запит може проходити через деяку кількість підсистем, аби виконати свою роботу, це є когнітивно складно для людини розуміти весь ланцюжок залежностей. Аби зняти це когнітивне навантаження з розробників та користувачів моніторингових систем, цей процес був автоматизований. Результатом такої автоматизації є розподілене трасування (Distributed Tracing). Розподілене трасування це підхід до інструментації підсистем таким чином, що запит, проходячи через різні підсистеми, наповнюється метаданими, що репрезентують його шлях між ними. Кожен запит має унікальний ідентифікатор, який зчитується в різних підсистемах, а також доповнюється таким чином, що може показати шлях запиту та кількість часу, що він провів в кожній підсистемі. Кінцевим результатом, що отримується за рахунок такого підходу є візуалізація, що показана на рисунку ...

Diagram

Description automatically generated*Рисунок х. Трейс презентований у вигляді спанів, які показують, що за частини системи займали найбільше часу для виконання.*

Така графічна візуалізація робить проблему ідентифікації та ізоляції проблеми тривіальною, адже є очевидним, де запит проводить найбільше часу, та де це не очікується. Так само, така графічна візуалізація легко показує зв'язок між залежностями, що є частиною життєвого циклу запиту у розподіленій системі. Це гарно презентовано на рисинку ...

Diagram

Description automatically generated*Рисунок х. Автоматично згенерована з трейсів карта системи (Application Map) в Microsoft Azure.*

###### Реалізовуючи компоненти моніторингу

Кожна з описаних вище компонентів необхідна, аби розподілена система стала оглядовою. Це було зроблено в розробленій нами програмі, яка покликана показати практичні приклади використання моніторингу в розподілених системах в контексті превентивної безпеки. В розробленій програмі було використано такі технології як .NET Core, Docker, Docker-Compose, Prometheus, Kibana, ElasticSearch та Logstash. Кожен з цих компонентів відповідає за певну частину моніторингу. .NET Core є платформою, яка дозволяє розмістити інтеграцію з цими додатками в програмному коді. Програмний код, що представлений на рисинку х, агрегує та виставляє через API endpoint метрики, аби вони були доступними для зовнішніх систем. Прикладом такої системи є Prometheus scrapper.

A picture containing graphical user interface

Description automatically generated

Рисунок х. Програмний код для збору та агрегації метрик для Prometheus.

Docker-Compose використано для агрегації необхідних моніторингових компонентів в один елемент, що може бути розгорнутим за допомогою команди:

**docker-compose up -d**

Graphical user interface, text, application

Description automatically generated

*Рисунок х. Docker-Compose файл, що відповідає за розгортання моніторингових компонентів.*

Docker-Compose файл виконано за допомогою мови YAML, яка декларативним способом описує необхідні ресурси для розгортання. В даному файлі можна побачити розгортання таких сервісів як Prometheus на порті 9090, ElasticSearch на портах 9200 та 9300, а також Logstash. Також файл описує необхідні конфігураційні аспекти такі як визначення співвідношення віртуальних сховищ даних та реальних сховищ даних.

Prometheus - це моніторингове рішення з відкритим кодом, що широко застосовується для збору, обробки та презентації даних пов'язаних з метриками. Першочергово, це рішення було розробленим в компанії SoundCloud, а згодом його зробили доступним для загального користування за ліцензією Apache License 2.0. Після його створення в 2012 багато компаній та організацій почали ним користуватись, адже проєкт мав велику спільноту навколо нього, що означало підтримку. Так само, Prometheus було додано до Cloud Native Computing Foundation у 2016, що підтверджує його статус стандарту в індустрії. Для того щоб розгорнути Prometheus у відповідному середовищі було використано Docker-Compose як найоптимальніший спосіб для швидкого розгортання, що не залежить від стану середовища.  
У наступних додатках можна побачити конфігураційні файли для Prometheus, що виконані на мові YAML. Ці конфігураційні файли впливають на те, як буде розгорнутим Prometheus та якою буде його поведінка у середовищі.

Text, timeline

Description automatically generated with medium confidence

Рисунок х. Файл Prometheus.yaml, що визначає конфігурацію сервісу Prometheus.

Graphical user interface, text, application

Description automatically generated

Рисунок х. Файл alert.yml визначає, які сповіщення Prometheus мусить відслідковувати і повідомляти користувачам, якщо такі стануться.

Агрегуючи усі наявні метрики, наш програмний продукт відкриває їх через API endpoint "/metrics". Зробивши запит до цього API endpoint, можна побачити наступну відповідь:

|  |
| --- |
| # HELP process\_open\_handles Number of open handles |
| # TYPE process\_open\_handles gauge |
| process\_open\_handles 461 |
| # HELP process\_working\_set\_bytes Process working set |
| # TYPE process\_working\_set\_bytes gauge |
| process\_working\_set\_bytes 51826688 |
| # HELP process\_num\_threads Total number of threads |
| # TYPE process\_num\_threads gauge |
| process\_num\_threads 25 |
| # HELP prom\_warning This fields indicates the warning count. |
| # TYPE prom\_warning counter |
| prom\_warning 3 |
| # HELP process\_cpu\_seconds\_total Total user and system CPU time spent in seconds. |
| # TYPE process\_cpu\_seconds\_total counter |
| process\_cpu\_seconds\_total 2.453125 |
| # HELP process\_private\_memory\_bytes Process private memory size |
| # TYPE process\_private\_memory\_bytes gauge |
| process\_private\_memory\_bytes 34926592 |
| # HELP process\_start\_time\_seconds Start time of the process since unix epoch in seconds. |
| # TYPE process\_start\_time\_seconds gauge |
| process\_start\_time\_seconds 1621152828.4403536 |
| # HELP dotnet\_collection\_count\_total GC collection count |
| # TYPE dotnet\_collection\_count\_total counter |
| dotnet\_collection\_count\_total{generation="1"} 0 |
| dotnet\_collection\_count\_total{generation="2"} 0 |
| dotnet\_collection\_count\_total{generation="0"} 0 |
| # HELP prom\_exception This fields indicates the exception count. |
| # TYPE prom\_exception counter |
| prom\_exception 2 |
| # HELP prom\_ok This fields indicates the transactions that were processed correctly. |
| # TYPE prom\_ok counter |
| prom\_ok 131 |
| # HELP process\_virtual\_memory\_bytes Virtual memory size in bytes. |
| # TYPE process\_virtual\_memory\_bytes gauge |
| process\_virtual\_memory\_bytes 2203929231360 |
| # HELP dotnet\_total\_memory\_bytes Total known allocated memory |
| # TYPE dotnet\_total\_memory\_bytes gauge |
| dotnet\_total\_memory\_bytes 4218768 |

*Таблиця X. Відповідь API endpoint, що дає розроблений програмний продукт.*

###### Візуалізовуючи результати моніторингу

Prometheus відображдає продуктивність системи у вигляді графу, що дозволяє користувачам у доступному їм форматі легко зрозуміти стан системи. Інформація подана в форматі графіків дозволяє легко порівнювати продуктивність системи з різних перспектив, а також оцінювати її відносно часу. Задля інтерактивної візуалізації використовується Prometheus Expression Browser, що надає ефективний спосіб відображати великі об'єми метрик зібраних за певний час. Ця частина системи є доступною за шляхом "<http://localhost:9000/new/graph>". За допомогою Prometheus Expression Browser можна дослідити дані, вибираючи конкретні метрики, що користувач хоче побачити на графіку.

Graphical user interface

Description automatically generated with low confidence

*Рисунок Х. Приклад користування Prometheus.*

A picture containing chart

Description automatically generated

*Рисунок Х. Приклад користування Prometheus.*

Chart, histogram

Description automatically generated

*Рисунок х. Приклад візуалізації метрик з розробленого нами програмного продукту.*

Дослідивши даний графік, вже можна зробити висновки, що програма не є стабільною, адже очевидними є падіння в графіку, що репрезентують вимкнення програми. Так само, варто наголосити на тому, що графік показує лінійний ріст вимірюваної метрики - "prom\_ok".

##### Ціна моніторингу як інструменту забезпечення стабільності

Це звучить логічно, що покращувати стабільність системи має сенс для користувачів системи та для бізнесу. Тим не менше, це парадоксально - в деяких випадках стає лише гірше!

Максимальна стабільність має свою ціну у вигляді довшого часу на розробку нових можливостей системи, довший час на доставку системи до користувачів. Очевидно, ця ціна напряму корелює з бюджетними обмеженнями, відповідно, менше можливостей системи буде імплементовано.

Порівнюючи такі SLA як 99%, 99.99% та 99.999% для певної системи, можна стверджувати, що користувач зазвичай не помітить різниці, адже його користувацький досвід стосовно стабільності буде більше сфокусований на менш стабільних факторах таких як сучасний смартфон, що забезпечує 99% SLA.

|  |  |  |
| --- | --- | --- |
| Дозволена недоступність на місяць | Дозволена недоступність на рік | SLA |
| 72 годин | 36.5 днів | 90% ("одна дев’ятка") |
| 7.20 годин | 3.65 днів | 99% ("дві дев’ятки") |
| 43.8 хвилин | 8.76 годин | 99.9% ("три дев’ятки") |
| 4.38 хвилин | 52.56 хвилин | 99.99% ("чотири дев’ятки") |
| 25.9 секунд | 5.26 хвилин | 99.999% ("п’ять дев’яток") |

*Таблиця х. Ілюстрація SLA у вигляді часу, коли система не доступна.*

Розуміючи ці фактори, при моделюванні систем ми мусимо глибоко розуміти доменну область і бізнес-модель проєкту, адже для соціальної мережі обміну картинками котиків абсолютно не обов'язково інвестувати в 99.999% SLA, проте для системи обробки запитів до невідкладної допомоги 9 годин відсутності можливості приймати запити є критичним та напряму впливає на життя та здоров'я користувачів системи. Тому, при виборі SLA, ми мусимо балансувати між ризиком недоступності системи (і шкодою, яка може бути нанесена цією недоступністю) та інноваційним розвитком системи, що створює нові можливості для своїх користувачів. Для того, аби змоделювати систему прийняття такого рішення, розглянемо наступний приклад, де кожен запит приносить однакову цінність:

|  |  |
| --- | --- |
| Пропозиція стосовно покращення SLA | 99.9% > 99.99% |
| Пропоноване покращення в стабільності | 0.09% |
| Прибуток, який приносить система | 1 млн. дол. |
| Фінансова цінність покращення | 1 млн. дол. \* 0.0009 = 900$ |

*Таблиця х. Відношення ціни до результату для моделювання рішення про інвестицію в покращення SLA.*

В цьому випадку, якщо вартість створення запропонованого покращення буде менша за 900$, то це має сенс імплементувати. Якщо вартість покращення буде вищою за 900$, то вартість покращення буде вищою за інвестицію, відповідно прибуток не збільшиться. Зазвичай, вартість такого покращення буде надзвичайно високою, адже покращення саме на верхній межі є найдорожчими з точки зору ресурсів необхідних на імплементацію.

Так само, розглядаючи ціну моніторингу, варто згадати, що моніторинг вимагає впровадження, розробку, конфігурацію та підтримку додаткових систем. Окрім цього, це створює додаткове когнітивне навантаження на розробників, що займаються розробкою та підтримкою розподіленої системи. Відповідно, необхідно більше ресурсів витрачати на те, аби документувати підходи та навчити розробників користуватись такою моніторинговою системою.

Тим не менше, інвестицію в навчання працівників можна оптимізувати таким чином, аби не витрачати на це час щоразу, коли з'являється новий працівник. Гарним рішенням подібної проблеми може бути уніфікація моніторингової інфраструктури та моніторингових вимог від частин розподіленої системи. У більшості випадків, моніторинг застосовується до різних платформ таких як .NET, Java, Go, тощо. Створивши документ, який описує найкращі практики до застосування моніторингових інструментів, з'являється можливість легкого навчання працівників, а також перевикористання цих найкращих практик у частинах розподіленої системи. Це стосується не лише концептуальних підходів, а й вже готових рішень для перевикористання: конфігурацій метрик, списку залежностей, тощо. Так само, окрім документації таких підходів, ще варто приділити увагу створенню базового прикладу сервісу, що береться за відправну точку для кожного нового сервісу в розподіленій системі. Такий базовий приклад включає в себе всі найкращі практики, які дозволяють вже бути готовими до використання в production середовищах з першого дня існування. Варто наголосити, що основна причина для такої документації - це оптимізувати використання часу в командах продуктових розробників, аби вони могли більше фокусуватися на продукті, а не на інструментації, моніторингу, чи налаштуванні інфраструктури.

Так само, осмислюючи моніторинг, варто врахувати, що деякий відсоток обчислювальних ресурсів системи буде витрачатись на моніторинг. Відповідно, це буде відображено у загальній продуктивності системи. Деякі розподілені системи можуть сягати тисяч серверів, тому очевидно, що ресурси витрачені на моніторинг та оглядовість таких систем можуть мати великий вплив на продуктивність розподіленої системи. Аби обслуговувати такі розподілені системи, моніторингова система має споживати якомога менше ресурсів, наскільки це можливо, але не менше, аби не спричинити втрати даних або власну недоступність.

Ще один фактор, що є частиною ціни і вартий того, аби його розглянути в даній роботі, це великий та постійно ростучий розмір артефакту, який створений моніторинговою системою. Очевидно, що обсяг даних, що збирається з розподіленої системи, є таким, що важко опрацювати, тому критично важливим є оптимізація таких даних для сприйняття користувачем. Так само, цей масив даних необхідно зберігати. В деяких системах є допустими видаляти такі дані після певного періоду зберігання.

Кілька тез у якості висновку:

* Надійність та стабільність систем має багато чого спільного з вимірюванням ризиків і прийманням обдуманих рішень. Аби вимірювати ризики, необхідно розуміти систему, продукт, а також вплив на користувачів. Вимірювання ризиків зазвичай є дорогим інструментом саме по собі.
* Рівень доступності системи має бути підібраний відповідно до того, що бізнес може собі дозволити - з точки зору ціни та впливу помилок на користувачів. Більшість систем не потребують SLA 99.999% та більше.

##### Превентивна безпека за допомогою моніторингу

Ненадійні системи можуть нівельовувати усі позитивні можливості продукту, адже користувачам буде важко ними скористатися. Відповідно, моделюючи систему, ми хочемо мінімізувати ризик проблем з системою, балансуючи з усіма іншими факторами. Тим не менше, що ми намагаємося запобігти? Говорячи про безпеку, що є небезпекою у цьому контексті?

* незадоволення користувачів
* шкода користувачам або їх власності (напр. даним)
* втрата довіри
* прямі або непрямі втрати прибутків
* негативний вплив на бренд та репутацію
* небажані згадування в засобах масової інформації
* неоптимальне використання обчислювальних ресурсів

Більшість з цих пунктів є прикладними з точки зору бізнесу, адже бізнес є тим на кого будь-що, що стається із системою, має прямий вплив. Так само, більшість з цих пунктів важко вимірювати. Врешті-решт, кожен очікує, що система буде працювати стабільно, без втрат, без проблем, і буде приносити дохід компанії, а користувачам цінність. Як каже практика - такого не буває.

Загальним підходом до проблеми стабільності є моніторинг, який було розглянуто в попередніх розділах. Тим не менше, більшість інструментів в моніторингу сфокусовані на дослідження та вирішення проблеми, коли інцидент відбувається прямо зараз, або вже стався. Напротивагу, ця робота розглядає інциденти задовго до того як вони стаються.

Відповідно, деякі підходи до превентивної безпеки є схожими до підходів вирішення інцидентів в production системах.

Моніторинг дає можливість інформувати розробників систем про виявлені негативні тренди в роботі системи, що, у свою чергу, дозволяє їм приймати осмислені рішення для реагування на ці тренди. В контексті превентивної безпеки однозначно нема сенсу терміново реагувати на зміни в трендах. Навпаки, є більше сенсу присвятити час аналізу, та лише потім, якщо аналіз показав джерело проблем, то пріорітизувати вирішення таких проблем. Так само, задля фокусу на превентивній безпеці, при розробці розподіленої системи необхідно особливу увагу приділяти застереженням. Задля цього можна доповнювати кодову базу різноманітним логуванням, додатковими метриками, що мали б під собою одну мету - визначити гранулярність того, як працює система. Іншими словами, необхідно зрозуміти ситуації, які не є помилкою, проте одночасно й не є happypath сценарієм. Найбільше нас цікавлять застереження (warnings) та їх рівень (warnings threshold).

Базуючись на даних з моніторингу, які презентовані у вигляді дашбордів та трендів, розробники систем та власники продукту можуть робити осмислені рішення, які спровоковані даними. Презентовані метрики на дашбордах можуть допомогти ідентифікувати частини розподіленої системи, які утилізовані більше, аніж це очікувано, або навпаки - утилізовані менше, аніж це очікувано. В обох випадках є спектр доступних рішень, що можна прийняти - чи то масштабування частин системи, переосмислення рішень моделювання системи, відключення частин системи, тощо.

|  |  |
| --- | --- |
| Аналітика даних | Рішення |
| CPU повільно зростає протягом останнього місяця | Розробники приймають рішення масштабувати сервіс, аби він мав більший об'єм CPU доступним, що, в свою чергу, зменшує ризик інцидентів. |
| Потік запитів зменшився, сатурація зменшилась | Розробники приймають рішення використати менш потужний сервер для цього сервісу і оптимізовують ресурси. |
| База даних занадто навантажена, 2% запитів закінчуються помилкою | Розробники приймають рішення змінити архітектуру розподіленої системи, додавши кешування даних. |

*Таблиця х. Цикл зворотнього зв’язку при наявності моніторингу.*

Так само, важлива річ, аби згадати, це те, що тренди можуть змінюватись базуючись на змінах в програмному продукті, що ми оглядаємо. Очевидно, що будь-яка зміна в кодовій базі може відобразитися на метриках. Тому, дуже цінним є мати можливість відслідкувати який конкретно артефакт (наприклад Docker образ) продукував такі метрики, аби могти відтворити схожу поведінку вже в іншому середовищі, а також змогти зкорелювати зміну в програмному коді, яка сталась, і зміну в трендах метрик. Чим більше даних ми маємо, аби підтвердити цю кореляцію, тим ліпше, тим змістовнішим є таке рішення.

##### Превентивна безпека з точки зору менеджера проєкту

Враховуючи, що ця робота фокусується на моделюванні та інженерній складовій надійності системи, тим не менше, не можна не долучити також менеджерський погляд на цю проблему, а якщо точніше, то погляд того, хто визначає бізнес-цінність продукту.

Відповідаючи на питання "Що є важливим з точки зору бізнесу?", відразу є очевидними кілька цілей:

* Продукт має нарощувати необхідні можливості, аби створювати додану вартість для користувачів якомога швидше
* Продукт має бути стабільним та доступним для користувачів

Балансуючи між цими цілями й створюється продукт. Інструментами для досягення цих цілей є команда, що ще іноді називають ресурсом [17]. Задля створення нових можливостей програмного продукту, очевидно, необхідними є зусилля команди. Тим не менше, їхні зусилля так само необхідні для підтримки системи та роботи над моніторинговою частиною.

Однією з критично важливих задач власника продукту стосовно надійності є визначити толерантність системи до помилок. Очевидно, що необхідно глибоко розуміти бізнес-цінність продукту та ризики кожної помилки, що може статися. З точки зору власника продукту, необхідно сформулювати бізнес цілі та висловити їх інженерній команді. Особливо це важливо в контексті систем, які мають велику кількість кінцевих користувачів, адже в таких системах є прямий зв'язок між успішністю продукту, та надійністю й продуктивністю системи.

Намагаючись визначити толерантність системи до помилок, варто задати та знайти відповіді на наступні питання:

* Який рівень доступності є необхідним? 99%, 50%, 99.999%? Відповідаючи на це питання, очевидно, що постає необхідність вимірювати чи SLA враховується в роботі продукту, тому необхідно вже на цьому етапі запитання робити інвестицію в моніторинг.
* Як правильно виміряти рівень доступності для даної системи? Чи є це зкорельованим із кількістю успішно виконаних запитів, чи радше з часом, коли система працює і відповідає хоч якось на запити?
* Чи можна виділити типи помилок, де деякі є більш критичними для продукту за інші?
* Які метрики варто враховувати досліджуючи поведінку цієї системи?
* На що впливають помилки з цієї системи? Який ризик? Що під цим ризиком - наші дані, дані користувачів, прибуток, прибуток користувачів, тощо?
* Якщо є схожі системи доступні на ринку, то які рівні доступності вони показують?
* Якщо проблема стається, то що ліпше - не приймати зовсім ніякі запити, або приймати, але обслуговувати частину з них?

Тим не менше, інвестиція в стабільність продукту мусить бути осмисленим рішення, що покликане створити баланс в парі нових можливостей та стабільності продукту, адже кількість ресурсів є обмеженою. Ще один фактор, аби розглянути, що відноситься до роботи з командою, який впливає на превентивну безпеку, це - культура моніторингу. Згідно з дослідженнями (референс на інтерв'ю), культура моніторингу є фундаментальною частиною превентивної безпеки в розподілених системах.  
Основою цієї культивації такої поведінки є проведення просвітницької роботи в командах, що працюють над продуктом, аби підвищити їхній рівень розуміння важливості моніторингу, зменшити кількість ізольованих нестандартних рішень, що не є оптимальними задля моніторингу системи. Моніторинг культура пропонує один з варіантів вирішення проблеми моніторингу розподіленої системи - збільшення важливості нефункційних вимог до програмного продукту (напр. продуктивність, доступність), які можна перевірити та контролювати за допомогою SLA [16].

Отже, розглядаючи менеджерський погляд на моніторинг, ми зрозуміли, що основним завданням є успіх продукту, що є результатом балансу між створенням нових можливостей продукту та його стабільністю. Також, були розглянуті способи досягти цього балансу, з наголосом на один з найрелевантніших у рамках сучасних розподілених систем - створення моніторингової культури.

##### Postmortem культура як частина превентивної безпеки

Корегуючі та превентивні дії (Corrective and Predictive Actions) - це загально-відома концепція для покращення надійності, яка фокусується на систематичному дослідженні причин ідентифікованих проблем та ризиків задля того, аби запобігти їхньому повторенню. Одним з таких CAPA інструментів в моніторингу розподілених систем є postmortem культура [16]. Це одна з найкращих практик для роботи з інцидентами. Вона полягає в тому, аби дослідити інцидент та навчитись від нього. А сам Postmortem - це документ про інцидент, що відповідає на наступні питання:

* Що сталось?
* Як система та розробники відреагували?
* Що можна було б зробити інакше наступного разу?
* Що можна зробити, аби не допустити цього ще раз?

У контексті превентивної безпеки, дуже важливо розуміти відповіді на останні два запитання, адже саме вони є тим, що може запобігти небезпеці - чи то уникнувши її, чи ефективніше поборовши. Основною ціллю створення такого документу є можливість засвідчити, що інцидент є задокументований, зрозумілий і очевидні його причини, а також, особливо важливим є те, що створений план превентивних дій, які не дозволять такому інциденту повторитися.

Postmortem обов'язково мають бути досліджені не лише в рамках команди, яка опрацювала інцидент, а й серед інших інженерних команд, аби всі мали можливість отримати розуміння того, що сталось. Таким чином, знання поширюватимуться по організації та запобігатимуть появі нових інцидентів. В кінці кінців, описані в postmortem інциденти мають потенціал зміцнити розподілену систему, а також команду розробників, що працює над нею.

# економічна частина

## Економічна характеристика проектного рішення (програмного продукту)

мету бакалаврської кваліфікаційної роботи, зокрема навести економічну оцінку доцільності розроблення програмного продукту з точки зору його окупності, маркетингову оцінку – з точки зору його прийняття споживачами на ринку інформаційних технологій.

## Інформаційне забезпечення та формування гіпотези щодо потреби розроблення проектного рішення (програмного продукту)

## Оцінювання та аналізування факторів зовнішнього та внутрішнього середовищ

## Формування стратегічних альтернатив

## Бюджетування

## Вибір стратегії

# Висновки

Моніторинг - це незамінна, необхідна частина сучасних розподілених систем для візуалізації, тестування, відлагодження та розробки. Без моніторингу та оглядовості сучасна розподілена система має мінімум шансів на успіх та стабільність. Зважаючи на це та інші дослідження, було ідентифіковано, що моніторинг та оглядовість розподілених систем вже далеко не технічна проблема. Навпаки, це стає темою, що пронизує весь життєвий цикл програмного продукту, а також є критичним для його успіху.

Дана робота описала розподілені системи як область дослідження, а також моніторинг таких систем, що дозволяє зрозуміти проблематику. Так само, робота розкрила питання моніторингу, описуючи три основні частини моніторингу - логи, метрики та трейси. Ці частини були презентовані як концептуально, так і практично за допомогою програмного продукту, що був створений для ілюстративності і розкриття концепцій. Велика частина цієї роботи є присвяченою превентивній безпеці як ключовій темі в рамкахх дослідження. Було зроблено наголос на ціну моніторингу, що складається не лише з технічних факторів, а й з організаційних. Так само, розкрите питання превентивної безпеки як цілі, та моніторингу як інструменту досягання цієї цілі. Осмислюючи моніторинг та превентивну безпеку також була розглянута менеджерська точка зору на дане питання. Завершуючи превентивну частину, було також розкрито вплив такого підходу як postmortem на запобігання інцидентам.

рекомендації щодо використання результатів бакалаврської роботи в навчальному процесі університету. зв’язок виконаної роботи з науково-дослідними розробками кафедри САП;

Можливі майбутні праці в тій тематиці є презентованими в наступному розділі.

## Майбутні роботи

Одним з найперспективніших напрямків дослідження ми вважаємо використання штучного інтелекту задля розпізнавання аномалій. Таке застосування може автоматизувати велику кількість процесів, які пов'язані з визначенням нормального рівня здоров'я системи. Наприклад, штучний інтелект можна застосувати для автоматичного визначення предикатів для сповіщень про стан розподіленої системи. Так само, можна використати штучний інтелект для виявлення аномалій, які не є залучені в систему сповіщень, але, тим не менше, є релеватними для дослідження розробниками системи, адже можуть впливати на стабільність системи. Зрозумівши природу цих аномалій, можна запобігти тим чи іншим інцидентам. Ще одним можливим застосуванням штучного інтелекту в моніторингових системах може бути автоматична мінімізація даних доступних користувачу для дослідження, аби зменшити когнітивне навантаження на користувача. Відповідно, деякими даними можна пожертвувати. Так само, одним з найцікавіших варіантів застосування штучного інтелекту та машинного навчання при інциденті є передбачення причин інциденту, базуючись на наявних даних.

Вплив на роботу та продуктивність розподіленої системи не було досліджено у цій роботі, а лише згадано. Тим не менше, це є важливою складовою для прийняття рішення про створення моніторингу, тому є можливість дослідити цю тематику також. Особливо, зважаючи на швидкі зміни в технологічному плані в моніторингових системах, очевидно, що є ризик зменшення продуктивності системи проти якої застосовано моніторинг. Дослідити вплив моніторингу на перформанс системи

В цій роботі ми фокусувались на сучасних підходах моніторингу стосовно превентивної безпеки в розподілених системах. Очевидно, що сучасний світ є таким, що швидко еволюціонує та змінюється, особливо в галузі інформаційних технологій. Тому однією з майбутніх робіт могло б бути дослідження змін, які відбуваються в превентивній безпекі за допомогою моніторингу в розподілених системах.

# Список використаних джерел

1. Robert Cooper and Keith Marzullo. 1991. Consistent detection of global predicates. SIGPLAN Not. 26, 12 (Dec. 1991), 167–174. DOI: <https://doi.org/10.1145/127695.122774>
2. Mansouri-Samani, Masoud & Sloman, Morris. (1993). Monitoring distributed systems. Network, IEEE. 7. 20 - 30. 10.1109/65.244791.
3. IEEE: Ieee standard glossary of software engineering terminology (1990), <https://ieeexplore.ieee.org/document/159342>
4. Manzo, M. & Frisiani, Arrigo & Vernazza, T.. (1982). A monitoring distributed system. Microprocessing and Microprogramming. 10. 19–24. 10.1016/0165-6074(82)90118-1.
5. Observer effect (physics), Wikipedia - <https://en.wikipedia.org/wiki/Observer_effect_(physics)>
6. Distributed Systems Observability by Cindy Sridharan, [https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html#:~:text=Logs%2C%20metrics%2C%20and%20traces%20are,the%20three%20pillars%20of%20observability](https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html%23:~:text=Logs%2C%20metrics%2C%20and%20traces%20are,the%20three%20pillars%20of%20observability)
7. Joyce, Jeffrey & Lomow, Greg & Slind, Konrad & Unger, Brian. (1987). Monitoring Distributed Systems.. ACM Trans. Comput. Syst.. 5. 121-150. 10.1145/13677.22723.
8. Johng, H., Kim, D., Hill, T., Chung, L.: Estimating the performance of cloud-based systems using benchmarking and simulation in a complementary manner. In: Intl Conference on Service-Oriented Computing. pp. 576–591. Springer (2018)
9. Lin, J., Chen, P., Zheng, Z.: Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Service-Oriented Computing. pp. 3–20. Springer International Publishing, Cham (2018)
10. Pinal V Chauhan, 2012, Cloud Computing In Distributed System, INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume 01, Issue 10 (December 2012),
11. Mostafa, Menna & Bonakdarpour, Borzoo. (2015). Decentralized Runtime Verification of LTL Specifications in Distributed Systems. 494-503. 10.1109/IPDPS.2015.95.
12. Niedermaier, S., Koetter, F., Freymann, A., & Wagner, S. (2019). On Observability and Monitoring of Distributed Systems – An Industry Interview Study. Lecture Notes in Computer Science, 36–52.
13. George Coulouris. Distributed Systems: Concepts and Design. Addison-Wesley, 2011
14. Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In Proc. of CCS’11, pages 151–162, 2011
15. Hagit Attiya and Jennifer L. Welch. Distributed computing: fundamentals, simulations and advanced topics. Wiley, 2004
16. Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. Site Reliability Engineering: How Google Runs Production Systems (1st. ed.). O'Reilly Media, Inc.
17. Resource (Project management), Wikipedia - <https://en.wikipedia.org/wiki/Resource_(project_management)>
18. Logging - the AOP way, OREN EINI <https://ayende.com/blog/3474/logging-the-aop-way>
19. Aspect-oriented programming, Wikipedia [https://en.wikipedia.org/wiki/Aspect-oriented\_programming#:~:text=In%20computing%2C%20aspect%2Doriented%20programming,separation%20of%20cross%2Dcutting%20concerns.](https://en.wikipedia.org/wiki/Aspect-oriented_programming%23:~:text=In%20computing%2C%20aspect%2Doriented%20programming,separation%20of%20cross%2Dcutting%20concerns.)

# додатки

весь код, таблиці, картинки, тощо.

# Подяки

Хотілося б подякувати усім людям, що супроводжували на шляху отримання релевантного досвіду для написання цієї роботи: усім, хто надихав, радив літературу, показував власні приклади, описував історії, тощо. Все це допомогло створити дану роботу саме такою, як вона є.