МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ХАРЧОВИХ ТЕХНОЛОГІЙ

Інститут (факультет) Автоматизації та комп'ютерних систем
Кафедра Інформаційних технологій, штучного інтелекту і кібербезпеки

«До захисту в ЕК»	**«До захисту допущено»**
Декан факультету АКС	Завідувач кафедри
________ Андрій ФОРСЮК	________ Сергій ГРИБКОВ
(підпис) (ім'я та прізвище)	(підпис) (ім'я та прізвище)
«03» лютого 2024 р.	«03» лютого 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

зі спеціальності 122 "Комп'ютерні науки"
(код і назва спеціальності)

освітньо-професійної програми Інформаційні управляючі системи та технології

на тему: Дослідження та розроблення методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення

Виконав: здобувач 2 курсу, групи ІС-2-3М
Драгомерецький Дмитро Сергійович ________
(прізвище, ім'я, по батькові повністю) (підпис)

Керівник Грибков Сергій Віталійович ________
(прізвище , ім'я та по батькові повністю) (підпис)

Консультанти ________ ________
(ім'я та прізвище) (підпис)

________ ________
(ім'я та прізвище) (підпис)

________ ________
(ім'я та прізвище) (підпис)

Рецензент Ярослав СМІТЮХ ________
(ім'я та прізвище) (підпис)

Я як здобувач(ка) Національного університету харчових технологій розумію і підтримую політику університету з академічної доброчесності. Я не надавав(-ла) і не одержував(-ла) недозволеної допомоги під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Здобувач________
(підпис)

Київ — 2024р.

2

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ХАРЧОВИХ ТЕХНОЛОГІЙ

Інститут (факультет) Автоматизації та комп’ютерних систем
Кафедра Інформаційних технологій, штучного інтелекту і кібербезпеки
Освітній ступінь магістр
Спеціальність 122 “Комп’ютерні науки”
(код і назва)
Освітньо-професійна програма Інформаційні управляючі системи і технології
(назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій, штучного інтелекту і кібербезпеки
____________ Сергій ГРИБКОВ
«19» грудня 2023 року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

Драгомерецького Дмитра Сергійовича
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження та розроблення методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення,
керівник роботи проф. Грибков Сергій Віталійович,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від «19» грудня 2023 р. № 1006-кс
2. Строк подання здобувачем роботи: 22.01.2024
3. Вихідні дані до роботи: Результати аналізу надійності, статистика та наслідки відмов розподіленого програмного забезпечення одержанні під час проходження переддипломної практики. Статистичні дані з системи моніторингу та звіти аналізу причин відмов
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити): Проаналізувати проблеми створення монолітного програмного забезпечення, проаналізувати проблеми створення розподіленого програмного забезпечення, проаналізувати методи оцінки готовності до введення в експлуатацію програмного забезпечення, розробити нову методологію перевірки готовності до введення в експлуатацію програмного забезпечення.

5. Перелік графічного матеріалу:
1) Відношення між підрозділами розробки та підтримки програмного забезпечення
2) Взаємодія каталогу сервісів з іншими частинами системи моніторингу

3) Процес введення в експлуатацію компоненту розподіленого програмного забезпечення

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Грибков С.В., професор		
2	Грибков С.В., професор		
3	Грибков С.В., професор		

7. Дата видачі завдання: «19» грудня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів виконання кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Робота над розділом 1. Загальна інформація та дослідження проблем керування надійністю розподіленого програмного забезпечення	20.11.2023	виконано
2	Робота над розділом 2. Дослідження процесів розробки розподіленого програмного забезпечення	01.12.2023	виконано
3	Робота над розділом 3. Розробка методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення	01.01.2024	виконано
4	Оформлення пояснювальної записки	15.01.2024	виконано
5	Створення автореферату та презентації	21.01.2024	виконано

Здобувач ____________ Драгомерецьки Д.С.
(підпис) (прізвище та ініціали)

Керівник роботи ____________ Грибков С.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Дослідження та розроблення методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення» розроблена здобувачем ступеня магістр за освітньо-професійною програмою «Інформаційні управляючі системи і технології» спеціальності 122 “Комп’ютерні науки” Драгомерецьким Д.С. та складається з 84 сторінки, 3 розділи, 3 рисунки, 2 додатки та 13 літературних джерел.

Метою роботи є дослідження та розроблення методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення яка буде застосовуватися в компанії Sabertech Systems S.L.

Проведено аналіз існуючих підходів до вирішення проблеми. Обґрунтовано недоцільність використання існуючих методів.

Результатом виконання магістерської роботи є методологія оцінки готовності до введення в експлуатацію компонентів розподіленої системи, форма самооцінки для підготовки до оцінки.

КЛЮЧОВІ СЛОВА: РОЗПОДІЛЕНІ СИСТЕМИ, НАДІЙНІСТЬ, ВВЕДЕННЯ В ЕКСПЛУАТАЦІЮ, МЕТРИКИ НАДІЙНОСТІ.

SUMMARY

The qualification work on the topic “Research and development of methodology for acceptance into service of components of distributed systems” was developed by Dragomeretskyi D.S., a master’s degree holder in the educational program “Information control systems and technologies”, specialty 122 "Computer Science" and consist of 84 pages, 3 chapters, 3 pictures, 2 appendices and 13 literary sources.

The goal of the master's work is to create methodology for acceptance into service of components of distributed systems that will be used by Sabertech Systems S.L.

In the master's thesis: Investigated problems of distributed systems during development and acceptance into service. Performed analysis on existing solutions for solve such type of problems. Provided arguments why existing methods not suitable to use it in now days.

The result of the master's work is methodology for acceptance into service of components of distributed systems and self-assessment checklist.

KEY WORDS: DISTRIBUTED SYSTEMS, RELIABILITY, ACCEPTANCE INTO SERVICE, RELIABILITY METRICS.

ЗМІСТ

РОЗДІЛ 1. ЗАГАЛЬНА ІНФОРМАЦІЯ ТА ДОСЛІДЖЕННЯ ПРОБЛЕМ КЕРУВАННЯ НАДІЙНІСТЮ РОЗПОДІЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. 12

1.1 Дослідження проблем створення монолітного програмного забезпечення 12

1.2 Дослідження процесу введення в експлуатацію монолітного програмного забезпечення 14

1.3 Постановка задачі на дослідження та розробку методології 16

1.4 Висновки по розділу 1 17

РОЗДІЛ 2. ДОСЛІДЖЕННЯ ПРОЦЕСІВ РОЗРОБКИ РОЗПОДІЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 18

2.1 Місце проведення досліджень 18

2.2 Структура підрозділів розробки 19

2.3 Керування продуктивністю 19

2.4 Керування якістю 20

2.5 Наслідки неналежного введення в експлуатацію 20

2.6 Аудит та аналіз існуючих сервісів 22

2.7 Проблема впровадження практик DevOps 31

2.8 Існуючі методи оцінки готовності до введення в експлуатацію 33

2.9 Висновки до розділу 2 33

РОЗДІЛ 3. РОЗРОБКА МЕТОДОЛОГІЇ ОЦІНКИ ГОТОВНОСТІ ДО ВВЕДЕННЯ В ЕКСПЛУАТАЦІЮ РОЗПОДІЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 34

3.1 Збір та структурування документації про компоненти розподіленого програмного забезпечення 34

3.2 Каталог сервісів 34

3.3 Зміна практики з DevOps на Dev 35

3.4 Метрики оцінки надійності 36

3.5 Баланс між NPS та eNPS 39

3.6 Форма самоперевірки 39

3.7 Методологія перевірки готовності до введення в експлуатацію 63
3.8 Результати впровадження пілотного проекту 67
3.9 Результати по розділу 3 70
ВИСНОВКИ 72
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 73
ДОДАТКИ 75
ДОДАТОК А. ПРИКЛАД НАЛАШТУВАННЯ SLO ЗА ДОПОМОГОЮ TERRAFORM В СИСТЕМІ МОНІТОРИНГУ DATADOG 75
Додаток Б. Фрагмент програми створення приватного індексу для LLM моделі 80

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

SOLID - це абревіатура, яка представляє собою п'ять базових принципів об'єктно-орієнтованого програмування та дизайну. Ці принципи допомагають створювати гнучкі, розширювані та обслуговувані програми.

DevOps - низка практик, призначених для покращення взаємодії розробників із фахівцями інформаційно-технологічного обслуговування та зближення їхніх робочих процесів одне з одним.

Mean Time to Recovery (MTTR) – показник вимірює середній час, який потрібен для відновлення системи після виникнення збою чи відмови.

ВСТУП

Актуальність теми. В процесі розвитку програмного забезпечення розробники стикаються з проблемами масштабування та зменшенням швидкості розробки при використанні монолітної архітектури. Для вирішення цих проблем існує рішення запропоноване великими технологічними компаніями та полягає в тому щоб перейти від монолітної архітектури до розподіленої (мікросервісної архітектури).

Проте брак попередньої експертизи в командах розробки та відсутність можливості збільшити кількість співробітників стає перешкодою для більшості компаній в збережені надійності розподіленої системи в цілому, що призводить до великих фінансових втрат, де мотивації команд розробки та припинення ініціативи впровадження рішення в цілому.

Вирішенням проблеми може буди методологія керуючись якою компанії зможуть ефективно впроваджувати рішення на базі мікросервісної архітектури та адаптувати саму методологію під потреби бізнесу.

Зв’язок роботи з науковими програмами, планами, темами кафедри, університету. Наукова робота виконувалась згідно з науково-дослідною роботою на кафедрі інформаційних технологій, штучного інтелекту і кібербезпеки «Дослідження та використання сучасних інформаційних технологій для виконання функцій та завдань виробничого і організаційного управління підприємств харчової галузі» (0120U105386 2020–2025 рр.) Національного університету харчових технологій.

Метою дослідження. Метою кваліфікаційної роботи є удосконалення ефективності оцінки готовності до введення в експлуатацію компонентів розподіленого програмного забезпечення за рахунок розроблення нової методології оцінки.

Завдання дослідження. Для поставленої мети необхідно розв’язати наступні задачі:

- проаналізувати проблеми створення монолітного програмного забезпечення;

- проаналізувати проблеми створення розподіленого програмного забезпечення, проаналізувати методи оцінки готовності до введення в експлуатацію програмного забезпечення;
- розробити нову методологію перевірки готовності до введення в експлуатацію програмного забезпечення.

Методи дослідження. Основними методами дослідження в цій роботі є дотримання загальнонаукових методологій і принципів системного підходу. Для розв'язання поставлених задач використовувались такі методи:

- методи математичної статистики для обробки та оцінки ефекту від неналежного введення в експлуатацію компонентів розподіленого програмного забезпечення;
- сумаризація за допомогою великих мовнних моделей для аналізу великої кількості звітів про причини виникнення проблем з підтримкою розподіленого програмного забезпечення та рішень що застосовувалися для їх усунення з метою отримання статистичних даних;
- методи об’єктно-орієнтованого проектування, використані для створення інформаційної технології та елементів системи;
- теорія реляційних та багатовимірних моделей даних при проектуванні баз знань, вітрин та сховища даних;
- методи, засоби і технології сучасного прикладного програмування для побудови практичних реалізацій.

Таким чином, кваліфікаційна робота припускає використання як теоретичного, так і емпіричного методу дослідження. Теоретичною основою роботи є наукові роботи провідних зарубіжних учених в області програмної інженерії, проектування систем. Емпіричною основою роботи стали методи порівняння, вимірювання, спостерігання, експерименту, аналізу, направлені на створення методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення.

Об'єктом дослідження є процес введення в експлуатацію компонентів розподіленого програмного забезпечення.

Предметом дослідження є технології та методи які застосовуються при введення в експлуатацію компонентів розподіленого програмного забезпечення.

Наукова новизна. Розроблена нова методологія оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення.

Практичне значення одержаних результатів. Одержані результати досліджень дозволять значно скоротити витрати, підвищити швидкість на розробку програмного забезпечення, та покращити надійність.

Особистий внесок здобувача. Досліджено існуючі методи оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення. Розроблено методологію оцінки готовності сервісу. Розроблено форму самоперевірки для підготовки до оцінки готовності сервісу до введення в експлуатацію.

РОЗДІЛ 1. ЗАГАЛЬНА ІНФОРМАЦІЯ ТА ДОСЛІДЖЕННЯ ПРОБЛЕМ КЕРУВАННЯ НАДІЙНІСТЮ РОЗПОДІЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

1.1 Дослідження проблем створення монолітного програмного забезпечення

Сучасне програмне забезпечення шо розраховане на широку аудиторію зробляться з використанням web технологій, що в свою чергу значно здешевлює вартість розробки, підтримки та розгортання нових версій.

В переважній більшості випадків даний тип програмного забезпечення має монолітну архітектуру.

Монолітна архітектура - це підхід до розробки програмного забезпечення, при якому весь функціонал програми або системи об'єднаний в одному кодові базі та запускається як єдиний процес.

Основні переваги монолітної архітектури:

- Одномоментність розгортання - усі частини програми (серверні та клієнтські) розгортаються разом, що полегшує розгортання та управління версіями.
- Простота розробки та тестування - всі компоненти програми розташовані в одній кодові базі, що спрощує розробку, тестування та пошук проблем.
- Зменшення складності конфігурації - оскільки все розташовано в одному місці, конфігурація та управління компонентами є менш складною.
- Простота моніторингу та ведення журналів помилок - логування та моніторинг зазвичай є простішими, оскільки весь стек додатку є єдиною системою.
- Оптимізована продуктивність - взаємодія між компонентами є ефективнішою, оскільки вони спілкуються безпосередньо в межах одного процесу.
- Простота масштабування – монолітні програми досить легко масштабувати як вертикально так і горизонтально.

Монолітна архітектура програмно забезпечення є оптимальним вибором для початку розробки нового програмного продукту невеликою кількістю розробників, що дозволяє досить швидко розвивати необхідну функціональність та поступово збільшувати чисельність розробників. Проте з плином часу ситуація змінюється в негативному напрямку, та виникають наступні проблеми:

1. Із збільшенням кількості функцій програмного забезпечення також збільшується кількість модульних на інтеграційних тестів, на виконання яких необхідно більше часу, тому збільшується час необхідний на реліз нового функціоналу або виправлення помилок (MTTR).
2. Час запуску програмного забезпечення збільшується шо призводить до сповільнення швидкості розгорнення нових версій, що в свою чергу збільшує час необхідний на відновлення після збоїв та сповільнює розробку загалом. Проблему можливо вирішити збільшенням кількості серверів шо оновлюються паралельно але це відповідно потребує збільшення кількості серверів які будуть задіяні лише під час розгортання програмного забезпечення. Частково це можливо оптимізувати використанням Kubernetes в хмарному середовищі, проте це дозволить оптимізувати лише вартість ресурсів, а час розгортання не зменшиться.
3. Сповільнення швидкості розробки та розгортання призводить до того що починається розробка функціональності програмного забезпечення що дозволяє змінювати налаштування без розгортання нової версії продукту. Це в свою чергу збільшує кодову базу та погіршує стан речей описаних в першому на другому пунктах.
4. Збереження налаштувань програмного забезпечення за межами вихідних кодів, наприклад в базі даних ускладнює процес розробки та підтримки.
5. Розростання кодової бази програмного продукту призводить до появи зон відповідальності за окремими частинам, та зон сумісної відповідальності яка

зазвичай ігнорується всіма членами команди та призводить до конфліктів в колективі.

6. Через появу зон відповідальності та зменшення швидкості тестування та розгортання виникає черга з релізів яку важко контролювати, наприклад команди що займаються зовнішнім виглядом продукту та маркетинговими дослідженнями хочуть виконувати часті розгортання з мінімальними змінами, проте вони вимушені чекати завершення довгого процесу розгортання, тим самим витісняють інші команди з черги що також призводить до конфліктів в колективі.

Одним із варіантів вирішення проблем пов’язаних з розробкою програмного забезпечення за монолітною архітектурою є відокремлення логічно ізольованих частин в розподіленні програмі модулі що взаємодіють між собою через мережу. Даний підхід називається мікро сервісна архітектура, на в останні роки набрав величезну популярність в компаніях середнього та великого бізнесу, оскільки дозволяє вирішити усі проблеми які виникають при використанні монолітної архітектури.

1.2 Дослідження процесу введення в експлуатацію монолітного програмного забезпечення

Програмне забезпечення розроблене з використанням монолітної архітектури вводиться в експлуатацію поетапно з залученням підрозділів розробки, кібербезпеки та інфраструктури компанії, в декілька етапів та займає великий обсяг часу, та складається з наступних етапів описаних у роботі [1], що повторюються з певною періодичністю:

1. Тестування навантаження та масштабування
2. Тестування кібербезпеки
3. Тестування стійкості до відмов
4. Тестування розгортання з резервних копій

5. Впровадження та навчання команди підтримки

Данні процедури можливо застосовувати і для введення в експлуатацію розподіленого програмного забезпечення оскільки кожен компонент системи по своїй природі є маленьким сервісом з монолітною архітектурою. Проте такий підхід не масштабується, і це призводить до перевантаження підрозділів кібербезпеки та інфраструктури компанії, а збільшення кількості співробітників в цих підрозділах є економічно недоцільними.

На даний момент міжнародні технологічні компанії такі як Google, Amazon, Netflix працюють використовуючи підхід з самообслуговуванням описаний в роботі [2], коли команди розробки самостійно вводять сервіси в експлуатацію. Даний досвід є цікавим для запозичення але важким в реалізації так як наведені компанії мають високий рівень зрілості та автоматизації бізнес процесів.

Враховуючи відсутність спеціалізованого досвіду введення в експлуатацію програмного забезпечення у команд розробки та зосередженість на певній предметній галузі призводить до виникнення проблем в наступних напрямках:

1. Керування конфігурацією – зазвичай команди розробки не мають достатнього досвіду в побудові надійної інфраструктури, системи розгортання програмного забезпечення.

2. Моніторинг та ведення журналів помилок – розподілена система потребує використання уніфікованих назв середовищ, використання Mapped Diagnostic Context (MDC), що описано в праці [3] для відстеження розподілених транзакцій, встановлення лімітів на використання ресурсів системи моніторингу та бюджетів помилок.

3. Резервне копіювання та безперебійність роботи – при розробці розподіленої системи відповідальність за резервне копіювання та впровадження політик забезпечення безперебійної роботи перекладається на команду розробки в якої зазвичай відсутній попередній досвід в цій галузі.

4. Мережеві проблеми – розподілена система по замовчуванню повинна буди стійкою до мережевих проблем, обривів зв’язку, та відповідно обробляти та звітувати про даний тип проблем, описано в праці [1].

5. Кібербезпека – в розподіленій системі набагато важче керувати встановленням оновлень бібліотек та компонентів, відслідковуванням появи нових вразливостей, впровадженням політик безпеки, через те що кожна підсистема має свій цикл релізів. Тому вирішення даного типу проблем потребує впровадження додаткових перевірок на етапі модульного тестування, компіляції проектів та системи моніторингу.

6. Відсутність обмінну знаннями між командами розробки – ви переході з розробки монолітного програмного забезпечення до розподіленого підрозділи розробки в короткі терміни починають фокусуватися на певних галузях перестають обмінюватися знаннями з колегами. Це призводить до виникнення так званих “бункерів знань” (knowledge silos) [4] та повторення одних й тих самих помилок та роботи над однаковими рішеннями різними командами.

1.3 Постановка задачі на дослідження та розробку методології

Для дослідження та розробки методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення необхідно розв’язати наступні задачі:

1. проаналізувати проблеми створення монолітного програмного забезпечення;
2. проаналізувати проблеми створення розподіленого програмного забезпечення,
3. проаналізувати методи оцінки готовності до введення в експлуатацію програмного забезпечення;
4. розробити нову методологію перевірки готовності до введення в експлуатацію програмного забезпечення.

1.4 Висновки по розділу 1

Розробка програмного забезпечення з використанням монолітної архітектури є оптимальним рішенням для невеликих команд або для програмних продуктів з низькою частотою релізів. З появою необхідності збільшити частоту релізів, кількість розробників та отримати можливість масштабування частин програмного забезпечення є доцільним перехід на розподілену архітектуру. Проте цей підхід потребує високого рівня зрілості від команд розробки та компанії в цілому.

Для дослідження та розробки методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення було сформульовано 4 задачі

РОЗДІЛ 2. ДОСЛІДЖЕННЯ ПРОЦЕСІВ РОЗРОБКИ РОЗПОДІЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Місце проведення досліджень

Дослідження проводяться в міжнародній організації що займається розробкою програмного забезпечення що початково мало монолітну архітектуру. Організація розпочала розробку 14 років тому невеликою командою інженерів та із зростанням кодової бази та кількості співробітників стикнулася з вже відомими проблемами підтримки такого типу рішень. Для вирішення проблем продуктивністю та гнучкістю розробки в 2020 році було прийнято рішення про перехід до розробки розподіленої системи без зупинки підтримки існуючого монолітного рішення. В 2022 році було розроблено значну кількість компонентів розподіленої системи та з’явилося розуміння необхідності впровадження нових практик контрою надійності через те що швидкість розробки сповільнилась і більшість команд витрачає час на стабілізацію роботи сервісів. Постійні перебої в роботі системи створюють також значне навантаження на команду підтримки користувачів.

Характеристики компанії:

- Кількість співробітників – 600
- Кількість розробників програмного забезпечення – 120
- Провайдер хмарних послуг – Amazon AWS
- Кількість підрозділів – 15
- Кількість програмних продуктів (сервісів) - 140
- Країни в яких знаходяться підрозділи розробки:
    - Іспанія
    - Німеччина
    - Україна
    - Словаччина

2.2 Структура підрозділів розробки

Компанія має ієрархічну структуру що складається з керівника розробки (Head of Delivery) та структурних підрозділів розробки що мають відповідну спеціалізацію у різних галузях такий як наприклад електронні платежі, взаємодія з державними структурами, взаємодія з системами корпоративного управління та обліку, мобільні додатки.

Керівник розробки – контролює роботу структурних підрозділів та видає дозволи на введення в експлуатацію розроблених ними програмних комплексів та сервісів.

Структура підрозділу розробки:

- Керівник підрозділу (Engineering Manager) – підпорядковується керівнику розробки та виконує контроль роботи свого структурного підрозділу;
- Головний архітектор (Domain Architect) – підпорядковується керівнику підрозділу, займається розробкою архітектури програмного забезпечення та навчанням інженерів;
- Інженер (Software Engineer) – підпорядковується керівнику підрозділу, займається розробкою програмного забезпечення;

Для забезпечення високої продуктивності роботи структурні підрозділи мають великий рівень автономності у виборі методології та керування життєвим циклом розробки програмного забезпечення. В зв’язку з унікальністю структурних підрозділів вони виконують підтримку розробленого програмного забезпечення самостійно.

2.3 Керування продуктивністю

Продуктивність роботи структурних підрозділів відслідковується за допомогою DORA метрик (DevOps Research and Assessment), що включають в себе чотири ключових показники по досягненню яких організація вважається високопродуктивною:

- Частота розгортання нових версій програмного забезпечення (Deploy frequency)

- Час розробки нових функцій (Lead time)
- Відсоток невдалих змін в програмному забезпеченні що призвели до проблем (Change failure rate)
- Середній час відновлення після виникнення проблем (Mean time to recovery)

2.4 Керування якістю

Для забезпечення якості програмного забезпечення корпоративною політикою затверджено використання автоматизовано неперервного інтеграційного тестування (Continuous Integration) та нереперного розгортання (Continuous Delivery). Заборонено ручне налаштування хмарних середовищ, дозволено лише використання підходу опису інфраструктури за допомогою вихідних кодів (Infrastructure as a Code).

Для керування проектами використовується Atlassian Jira де для кожного структурного підрозділу заводиться по два проекти:

- Проект керування розробкою
- Проект керування підтримкою

Для кожної зміни вихідних кодів програмного забезпечення реєструється відповідна задача. Номер задачі використовується при збереженні змін в систему контролю версій, що дозволяє відслідковувати та аналізувати які саме зміни призвели до проблем в роботі програмного забезпечення.

2.5 Наслідки неналежного введення в експлуатацію

Для підтвердження актуальності дослідження проблем в надійності було прийнято рішення про необхідність кількісної оцінки наслідків.

Оцінку наслідків неналежного введення в експлуатацію компонентів розподіленої було проаналізовано використовуючи статистичні дані з системи керування проектами Atlassian Jira. В якості вибірки було взято задачі за останні 3 роки що були пов’язані з порушенням надійності сервісів та проаналізовано звіти про причини виникнення проблем (Root Cause Analysis). Результати було згруповано за напрямком, вказано причину проблеми на час протягом якого система не працювала наведені в таблиці 2.1.

Таблиця 2.1. Порушення надійності на час простою системи

Напрямок	Проблема	Час простою (г)
Хмарна інфраструктура	Не спрацювало автоматичне масштабування кластеру Kubernetes	6
	Розмір кластеру AWS ECS було зменшено до мінімуму під час розгортання нової версії програмного забезпечення	1
Бізнес процеси	Нескоординовані дій під час проблем з базою даних збільшили час простою з 1 години до 4	4
	В результаті ручного видалення індексів в таблицях бази даних індекси були перебудовані в результаті роботи ORM шару що призвело до блокування таблиць на 4 години	4
Кібербезпека	В наслідок повідомленої за програмою BugBoundy вразливості робота сервісу була призупинена зупинена на 4 години для усунення проблеми	4

Середня вартість 1 години простою IT системи, що надано керівництвом організації складає близько 20 тисяч доларів США. Тобто загальні втрати за період склали 19*20.000 = 380,000 доларів США, не враховуючи втрат рекламного бюджету та потенційної недоодержаної вигоди. Проте в організації розробляються нові програмні продукти тому майбутні втрати можуть збільшитися.

Середній час необхідний для виправлення помилок знайдених при аудитах буде враховано для оцінки вартості впровадження методології, порівняння вартості з втратами що понесла організація в минулому через присутність даних помилок. Тобто

це дозволить нам порахувати період повернення інвестицій (ROI) якщо він матиме місце.

2.6 Аудит та аналіз існуючих сервісів

В зв’язку з унікальністю підрозділів розробки виникає проблема коректної оцінки керівником розробки, готовності до введення в експлуатацію програмних комплексів та сервісів.

Для вирішення проблеми пропонується провести аудит програмного забезпечення розробленого різними підрозділами для виявлення порушень в наступних напрямках:

- Інженерія програмного забезпечення
- Хмарна інфраструктура
- Бізнес процеси
- Кібербезпека

В результаті проведення аудиту 4 програмних продуктів було виявлено значну кількість проблем та підтверджено що проблеми є загальними для більшості структурних підрозділів. Знайдені проблеми представленні в таблиці 2.2 що містить в собі напрямок аудиту, знайдені проблеми, кількість сервісів з 4 що пройшли аудит в яких була знайдена проблема та опис ризиків пов’язаних з нею.

Таблиця 2.2. Результати аудиту

Напрямок	Проблема	Кількі-сть постраж-далих компо-нентів	Ризики
Програмне забезпечення	Налаштування профілю Spring Boot	3	В разі запуску програмного забезпечення в середовищі

Напрямок	**Проблема**	**Кількість постраждалих компонентів**	**Ризики**
	не містить конфігурації обробки системного сигналу SIGTERM		кластерів Kubernetes або AWS ECS при автоматичному масштабуванні можливе завершення процесу без закриття транзакцій що призведе до не коректної роботи.
	Реєстрація помилок в системному журналі виконується з некоректним рівнем важливості	2	Реєстрація помилок в системному журналі з некоректними рівнем важливості не дає змогу оцінювати стан роботи програмного забезпечення та налаштувати автоматичне сповіщення про проблеми в експлуатації
	Час запуску програмного забезпечення не оптимізовано	2	В разі запуску програмного забезпечення в середовищі кластерів Kubernetes або AWS ECS при автоматичному масштабуванні великий час

Напрямок	**Проблема**	**Кількі-сть постраж-далих компо-нентів**	**Ризики**
			запуску призведе до не коректної роботи системи та не опрацювання запитів користувачів
	Програмне забезпечення запускається від імені користувача що має адміністративні привілеї	2	В разі наявності вразливостей RCE (Remote Core Execution) в програмному забезпеченні або в сторонніх бібліотеках призведе до можливості встановлення на серверах шкідливого програмного забезпечення та вірусів

Напрямок	Проблема	Кількість постраждалих компонентів	Ризики
	Використовуються абсолютні http посилання на Rest API в React застосунках	2	Використання абсолютних http посилань може призвести до ситуації що тестова версія програмного забезпечення буде використовувати продуктивну версію серверного програмного забезпечення і навпаки. На практиці це призводить до того що користувачі не можуть використовувати систему тому що на тестових серверах відсутні данні та необхідні налаштування
Хмарна інфраструктура	Не налаштовано створення резервних копій S3 сховища стану Terraform	2	Відсутність резервних копій S3 сховища стану Terraform може призводити до ситуацій коли файли стану Terraform пошкоджені і при внесенні змін в налаштування вся інфраструктура буде видалена

Напрямок	**Проблема**	**Кількість постраждалих компонентів**	**Ризики**
	Helm chart не містить налаштувань requested/limit для пам’яті виділеної під Pod	2	Відсутність налаштувань requested/limit для пам’яті виділеної під Pod може призводити до ситуацій коли функція OOM Killer ядра операційної системи буде завершувати процес виконання програмного забезпечення у раз недостатньої кількості вільної пам’яті замість процедури масштабування кластеру Kubernetes
	Виконується розгортання нових версій програмного забезпечення в кластер AWS ECS за допомогою Terraform що призводить до зміни desire_count	2	При розгортанні нових версій програмного забезпечення в кластері AWS ECS за допомогою Terraform призводить до зміни desire_count (кількості запущених задач) до значень вказаних в Terraform. Це призводить до проблем пов’язаних з тим що на момент

Напрямок	**Проблема**	**Кількість постраждалих компонентів**	**Ризики**
			розгортання кластер може буди масштабованим для обслуговування великої кількості користувачів, і різке зменшення розміри приведе до відмови в обслуговуванні.
	Відсутнє налаштування автоматичного очищення застарілих Docker образів з реєстру AWS ECR	4	Відсутнє налаштування автоматичного очищення застарілих Docker образів з реєстру AWS ECR призводить до значного дорожчання вартості хмарної інфраструктури
	Використовується в якості Healthcheck контролер що завжди повертає HTTP 200	2	Використання якості Healthcheck контролерів що завжди повертають HTTP 200 призводить до некоректної оцінки готовності до роботи сервісів кластерами Kubernetes або AWS ECS та відмови в обслуговуванні.

Напрямок	**Проблема**	**Кількість постраждалих компонентів**	**Ризики**
Бізнес процеси	Відсутній DRP документ	2	Відсутній DRP документ може призвести до непередбачуваних наслідків в процесі обслуговування програмного забезпечення в умовах збоїв в роботі. Що може призвести до повної зупинки бізнесу
	Відсутній процес тестування відновлення з резервних копій	4	Відсутній процес тестування відновлення з резервних копій з певною періодичністю може призвести до ситуації відсутності резервних копій в процесі обслуговування програмного забезпечення в умовах збоїв в роботі. Що може призвести до повної зупинки бізнесу
Кібербезпека	Налаштування профілю Spring Boot містять	2	Налаштування профілю Spring Boot містять незашифровані паролі доступу до бази даних

Напрямок	Проблема	Кількість постраждалих компонентів	Ризики
	незашифровані паролі доступу до бази даних		призводить до ризику несанкціонованого доступу до баз даних та необхідності зміни всіх паролів при звільнені співробітників. Необхідно перейти на використання хмарного менеджеру паролів (KMS)
	Відсутня перевірка вразливих залежностей на етапі компіляції Spring Boot проекту	3	Відсутня перевірка вразливих залежностей на етапі компіляції Spring Boot проекту може призводити до появи вразливостей в програмному забезпечені. Необхідно впровадити автоматичну перевірку за допомогою maven-dependency-check
	Відсутня перевірка вразливих залежностей на етапі	3	Відсутня перевірка вразливих залежностей на етапі компіляції React проекту може призводити до появи вразливостей в

Напрямок	Проблема	Кількі-сть постраж-далих компо-нентів	Ризики
	компіляції React проекту		програмному забезпечені. Необхідно впровадити автоматичну перевірку за допомогою npm audit
	Базовий образ Docker з Java JDK береться не з офіційного реєстру образів	2	Використання базових образів Docker з Java JDK що беруться не з офіційного реєстру образів несе ризики встановлення шкідливого програмного забезпечення на сервери компанії в разі атаки на розробника даного образу
	Відсутня перевірка вразливих версій програмного забезпечення в Docker образі	2	Відсутня перевірка вразливих версій програмного забезпечення в Docker образі призводить до ризиків появи вразливих компонентів на серверах компанії
	Використовуються одні і тій самі паролі	2	Використовуються одні і тій самі паролі до бази даних в різних сервісах призводить до

Напрямок	Проблема	Кількість постраждалих компонентів	Ризики
	до бази даних в різних сервісах		ризиків несанкціонованого доступу до баз даних та відмови в обслуговуванні при спробах змінити паролі тому що вони мають буди замінені всюди одночасно.
	Ненадійна перевірка авторизації доступу до методів Rest Api	1	Ненадійна перевірка авторизації доступу до методів Rest Api розповсюджена проблема при налаштуванні шару безпеки Spring Security в Spring проекті. Для уникнення даних проблем перевірка доступу має буди описана в автоматизованих інтеграційних тестах всіх контролерів

2.7 Проблема впровадження практик DevOps

DevOps - низка практик, призначених для пожвавлення взаємодії розробників із фахівцями інформаційно-технологічного обслуговування та зближення їхніх робочих процесів одне з одним [6]. Практика DevOps покращує взаємозалежність між розробкою та використанням програмного забезпечення і має на меті допомогти

організаціям швидше створювати та оновлювати програмні продукти та послуги. Перші спроби впровадження приктик DevOps з’явилися у 2008 році, і цей напрямок на сьогоднішній день набрав величезної популярності.

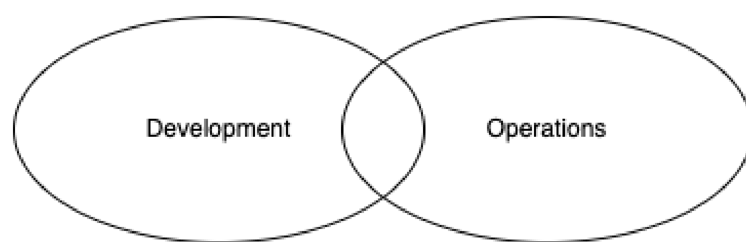

Рисунок 2.1 – Відношення між підрозділами розробки та підтримки програмного забезпечення

Проте через не вірне розуміння концепцій нової культури це призвело до зміни назви позиції системного адміністратора на DevOps інженера, що не наблизило індустрію розробки програмного забезпечення до вирішення проблем, а призвело до погіршення стану речей:

- Розробники програмного забезпечення намагаються застосовувати до керування інфраструктурою, яка за своєю природою має декларативних стиль опису, підходи з об’єктно орієнтованого програмування. Це призводить до розробки складної кодової бази керування інфраструктурою або в особливих випадках нових фреймворків.
- Системні адміністратори в свою чергу через відсутність досвіду розробки програмного забезпечення та знань принципів SOLID створюють рішення що призводять до витоків інфраструктурного коду на рівень бізнес логіки. Це призводить до проблем з підтримкою програмного забезпечення та складності в розробці автоматизованих модульних та інтеграційних тестів.
- Практика дозволяє покращити співпрацю між командами але не усунути існування “бункерів знань”.

2.8 Існуючі методи оцінки готовності до введення в експлуатацію

На ринку існують стандарти сімейства ISO такі як ISO/IEC/IEEE 29119 які описують процеси введення в експлуатацію програмного забезпечення та їх можливо застосовувати для розподілених систем. Проте їм можливо вважати "скомпрометованими" через те що багато компаній на ринку намагаються одержати сертифікацію ISO, цим процесом у більшості випадків успішно займаються керівники підрозділів без залучення розробників. Тому в спільноті розробників існує супротив при намаганні безпосередньо застосувати стандарти ISO для керування розробкою, та негативно впливає настрої в команді.

Експерименти з впровадження можливо проводити у випадку бізнес необхідності або якщо компанія має високі значення індексу eNPS.

2.9 Висновки до розділу 2

Результати аудиту підтверджують наявність в компанії проблем пов’язаних з визначенням готовності сервісів до введення в експлуатацію, а однакові помилки допущені різними підрозділами розробки свідчить про відсутність обміну знаннями. Одержані результати підтверджують доцільність розробки методології оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення.

РОЗДІЛ 3. РОЗРОБКА МЕТОДОЛОГІЇ ОЦІНКИ ГОТОВНОСТІ ДО ВВЕДЕННЯ В ЕКСПЛУАТАЦІЮ РОЗПОДІЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Збір та структурування документації про компоненти розподіленого програмного забезпечення

Розробка та підтримка розподіленого програмного забезпечення потребує використання автоматизованих систем керування документацією про компоненти, через те що розробка ведеться незалежними командами, та оновлення документації використовуючи класичні підходи такі як використання корпоративних баз знань та UML діаграм є неефективною та потребує значних людських ресурсів.

Документування API компонентів можливо виконувати безпосередньо в кодовій базі використовуючи специфікацію OpenAPI та автоматично генерувати інтерактивну документацію.

Оскільки важливою складовою розподіленого програмного забезпечення є система моніторингу, що автоматично наповнюється інформацією про компоненти, їх назви, середовища виконання, використання ресурсів, помилки, продуктивність та взаємодію між собою ми можемо використовувати її як систему керування документацією додавши лише базову інформацію таку як опис та посилання на додаткові джерела інформації.

3.2 Каталог сервісів

Найпростішим але надзвичайно дієвим інструментом структуризації інформації про розподілену систему є так званий каталог сервісів (Service Catalog). Каталог зазвичай містить список доступних сервісів з із описом, власником, посиланнями на документацію, продуктивні та тестові середовища, репозиторії з вихідними кодами та відповідні розділи в системі моніторингу.

Сучасні системи моніторингу мають вбудовану підтримку каталогу сервісів з автоматичним заповненням базуючись на інформації що надходить від сервісів які мають інтеграцію з системою моніторингу. На практиці це звільняє команди

розробників від ручного оновлення каталогу, та дозволяє отримувати вичерпну інформацію про кількість сервісів, із власників, середовища в яких вони запущені, версії програмного забезпечення, стан та наявність помилок.

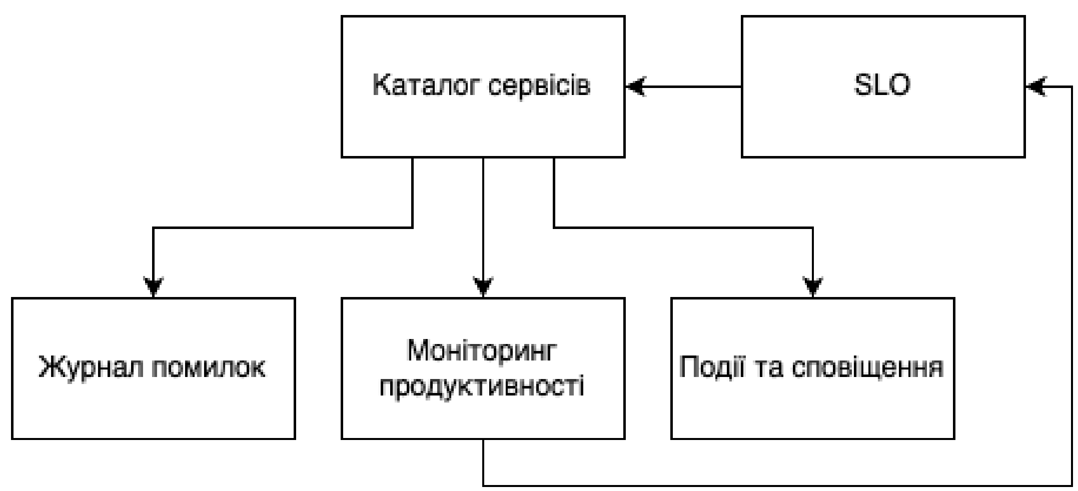

Рисунок 3.1 Взаємодія каталогу сервісів з іншими частинами системи моніторингу

Впровадження каталогу сервісів є відправною точкою в створені процесу керування надійністю розподілених систем та при інтеграції метрик оцінки надійності може використовуватися в якості глобального звіту про стан розподіленої системи.

3.3 Зміна практики з DevOps на Dev

Оскільки використання практики DevOps не дає повноцінного вирішення проблеми ми можемо скористатися іншим підходом на спробувати об’єднати експертизу різник команд в одну.

При спробі долучити системних адміністраторів до розробки програмного забезпечення ми стикнемся з проблемою відсутності досвіду розробки комерційного програмного забезпечення та знань принципів SOLID [3], на отримання який знадобиться багато часу. Іншою проблемою буде бункер інфраструктурних знань, який прийдеться вирішувати шляхом додавання в кожну команду по 2 інженера що є економічно невигідним.

Іншим рішенням є делегування задач з керування інфраструктурою на розробників програмного забезпечення. В даному випадку розробники вже мають відповідний досвід роботи з усіма видами сервісів які надають команди системних адміністраторів, наприклад бази даних, черги та тощо. Винятком є лише відсутність досвіду роботи з мережами, резервним копіюванням та системами захисту. Проте в сучасному хмарному середовищі налаштування мережі відбувається не дуче часто адже а резервного копіювання та системи захисту налаштовуються за допомогою наприклад Terraform. Тобто надавши команді розробки приклади налаштування того чи іншого хмарного сервісу, вимог та методів перевірки ми зможемо в короткі строки закрити прогалини в знаннях та повністю делегувати інфраструктурні задачі.

3.4 Метрики оцінки надійності

Метрика надійності (Reliability) кількісно відображає ймовірність активної роботи інфраструктури за призначенням без перебоїв або помилок, вимірюватись може як тривалість, так і у вигляді ймовірностей чи відсотків. Reliability корелює з іншою метрикою — MTBF (Mean Time Between Failures):

$$\text{MTBF} = \frac{Total\ uptime}{Number\ of\ breakdowns}$$

Проте використання MTBF не дає очікуваного результату оскільки сервіс може бути в частково не робочому стані, наприклад не опрацьовувати запити користувачів в певному стані, або не впливати на користувачів з повільним з’єднанням до мережі інтернет.

Сучасні системи моніторингу мають вбудовану підтримку моніторингу продуктивності програмного забезпечення (Application Performance Monitoring, APM) яка використовуються для моніторингу та управління продуктивністю, ефективністю та доступністю програмного забезпечення у реальному часі.

Основні функції APM включають в себе:

- Моніторинг продуктивності - вимірювання швидкодії та ефективності додатків, включаючи час відповіді, завантаження сервера, час виконання запитів та інші показники.
- Відстеження та аналіз транзакцій - визначення проблем у вихідному коді, які можуть впливати на продуктивність, та аналіз патернів викликів функцій у ході виконання транзакцій.
- Моніторинг ресурсів - спостереження за використанням оперативної пам'яті, процесорного часу, мережевими ресурсами та іншими системними ресурсами.
- Спостереження за викликами до бази даних - визначення ефективності запитів до баз даних, виявлення повільних запитів та оптимізація викликів.
- Моніторинг взаємодії з зовнішніми сервісами - спостереження за взаємодією з іншими веб-сервісами чи зовнішніми компонентами та визначення впливу цих взаємодій на продуктивність.
- Виявлення та аналіз помилок - визначення помилок, виключень та несправностей у коді для швидкого реагування та усунення проблем.
- Аналіз користувацького досвіду - моніторинг взаємодії користувачів з додатком для виявлення проблем та можливостей оптимізації.

Застосування APM дозволяє розробникам відслідковувати та оптимізувати продуктивність додатків, підтримувати високу доступність та швидко реагувати на проблеми в реальному часі. APM-інструменти допомагають вдосконалювати якість та продуктивність програмного забезпечення, що є ключовим для забезпечення задоволення користувачів та успіху бізнесу.

Використовуючи статистичні дані по кількість запитів до сервісу, час відповіді та помилки зібрані за допомогою APM-інструменту ми можемо зрозумілі метрики оцінки надійності що можливо використовувати як KPI для підрозділів розробки. В якості системи метрик можемо використати запропоновані компанією Google в книзі Site Reliability Engineering [6]:

- SLI (Service Level Indicator) - визначає конкретний показник чи параметр, який використовується для вимірювання роботи системи. Це може бути, наприклад, час відгуку веб-сайту, швидкість виконання запитів до бази даних, чи інший вимір, який важливий для функціонування системи.
- SLO (Service Level Objective) - є цільовим значенням чи допустимим діапазоном для SLI. Це конкретна мета, яку команда чи організація ставить для певного SLI. Наприклад, "99% запитів повинні мати час відгуку менше 200 мілісекунд".
- SLA (Service Level Agreement) - це формалізована угода між постачальником послуги та користувачем чи клієнтом щодо рівня обслуговування. Він включає у себе SLO та визначає відповідальність за досягнення цих цілей. Якщо рівень обслуговування не відповідає SLO, постачальник може взяти на себе відповідальність за невиконання угоди.

Для визначення стану сервісу нам буде достатньо визначення двох SLI на базі яки ми будуємо SLO з цільовим значенням 99.99% та додаємо їх в каталог сервісів:

- Latency - середній час відповіді на запити
- Failure rate – відсоток запитів що завершилися з помилкою

При розрахунку показників ми повинні виключити з розрахунку статистику про роботу так званих Health та Liveness методів API наших сервісів які використовуються системами Kubernetes, AWS ECS та Docker Swarm для визначення стану сервісу та його готовності до опрацювання запитів. Це пов’язано з тим що дані методи API мають мінімальний час відповіді та викликаються дуже часто навіть при умові шо сервіс не опрацьовує запити, що призводить до викривлення статистики.

За допомогою Latency SLO та Failure rate SLO ми отримуємо інформацію про те які з сервісів розподіленої системи мають проблеми з продуктивністю та надійністю та можемо приймати рішення про поглиблений аналіз технічних проблем того чи іншого сервісу. Також SLO є зручним інструментом KPI якими можуть користуватися керівництво компанії. Приклад налаштування SLO в системі моніторингу DataDog за допомогою Terraform наведено в додатку 1.

3.5 Баланс між NPS та eNPS

Net Promoter Score (NPS) або індекс споживчої лояльності – це показник репутації компанії, який допомагає зрозуміти, наскільки все добре чи погано. Дослідження спрямовується на оцінювання лояльності клієнтів до компанії або покупців до будь-якого її товару чи послуги. Вважається, що індекс NPS тісно корелює з доходами, адже лояльність клієнтів до компанії змушує їх бажати купувати більше та рекомендувати робити це своїм знайомим та друзям.

Employee Net Promoter Score (eNPS) - це похідний від NPS індекс, який дозволяє компаніям виміряти лояльність працівників. Під лояльністю у даному випадку ми розуміємо прихильність до компанії, задоволеність роботою в неї настільки, щоб рекомендувати її як місце роботи своїм друзям та знайомим.

В зв’язку з недостачею висококваліфікованих кадрів на ринку розробки програмного забезпечення компанії в даному сегменті змушенні балансувати між фокусом на клієнтах та збільшенню продажів та фокусом на створенні комфортних умов праці для співробітників.

Дана проблема накладає обмеження на можливості керівників впливати на впровадження методів підвищення надійності програмного забезпечення. Оскільки високий рівень бюрократичності та суворі процедури призводять до зниження показнику індексу eNPS та відтоку кадрів, що в свою чергу веде до зниження продуктивності розробки в цілому та збільшенню витрат на пошук нових співробітників. Тому при розробці підходів необхідно це враховувати, та залучати команди до процесу створення та намагатися зібрати відгуки про нових процес на етапах проектування.

3.6 Форма самоперевірки

Підготовка до введення в експлуатацію програмного це складний процес що потребує додаткового залучення спеціалістів з кібербезпеки та системних адміністраторів. Їх залучають зазвичай на фінальних етапах розробки програмного забезпечення де вони проводять аудит та готують інфраструктуру до розгортання.

Проте масовий перехід на використання хмарних технологій та підходів Infrastructure as Code (IaC) довів що команди розробки можуть самостійно вирішувати питання побудови інфраструктури та забезпечення захисту від кібер загроз, і виконують цю роботу ефективніше ніж спеціалісти з кібербезпеки та системні адміністратори тому що знають особливості, помилки та план розвитку власного програмного забезпечення.

Також практично доведено що планування та розгортання інфраструктури на ранніх етапах розробки програмного забезпечення дозволяти розробити надійний сервіс та залучити всіх членів команди до вивчення всього стеку технологій тим самим зменшити прогалини в знаннях всієї команди в цілому. Від компанії потрібно отримати чіткі вимоги до сервісів та надати рекомендації.

Для цього можливо запозичити підхід шо використовується платіжними системами при підключені нових мерчантів. Як відомо для того щоб отримати дозвіл на опрацювання даних платіжних карток необхідно мати сертифікацію Payment Card Industry (PCI) Data Security Standard яку можливо отримати після аудиту спеціалізованими аудиторськими організаціями. Проте для мерчантів що обробляють менше 20 тисяч транзакцій на рік можливо пройти внутрішню перевірку Payment Card Industry (PCI) Data Security Standard Self-Assessment Questionnaire A and Attestation of Compliance. Підхід з самоперевіркою можливо використовувати і для підготовки сервісів до введення в експлуатацію, для потрібно відгодувати форму самоперевірки.

Для підготовки списку питань будо використано статистичні дані про запитання та проблеми шо надійшли до команди системних адміністраторів, відділу кібер безпеки та команди підтримки шляхом аналізу звітів про причини виникнення проблем (Root Cause Analysis, RCA). Було використано дані за період в 4 роки, що були експоровані з Jira в окрему базу даних, ручну обробку, фільтрацію та сумаризацію з використанням LLM. Фрагмент програми створення GPT індексу наведено в додатку 2. Після формування списку вимог вони були перевірені командою системних архітекторів AWS компанії Amazon.

Форма самоперевірки приведена в таблиці №3.1, та містить вимоги з коментарем і важливість цих вимог за шкалою від 1 до 10 де 1 – це бажана запровадити а 10 – абсолютна необхідність.

Таблиця 3.1. Форма самоперевірки

Напрямок	Вимога	Важл.	Коментар
1. Кодова база	1. Кодова база не повинна містити в собі паролі та ключі доступу	10	Наявність паролів та ключів доступу в кодові базі несе за собою великі ризики безпеки та потребує зміни паролів при звільненні когось з співробітників. Перевірку можливо автоматизувати за допомогою плагіну *spotbugs*.
	2. Релізна гілка репозиторію вихідних кодів повинна бути захищеною від прямих змін	10	Для унеможливлення автоматичного розгортання продуктивного середовища не протестованою версією програмного забезпечення. Релізна гілка репозиторію повинна приймати зміни лише через merge запити.

Напрямок	Вимога	Важл.	Коментар
	3. Зміни в репозиторій вихідних кодів повинні прийматися при вказуванні ідентифікатора задачі	10	Репозиторій вихідних кодів повинен містити hook що перевіряє наявність в коментарі до коміту ідентифікатора задачі. Ще необхідно для автоматичного формування списку задач що увійшли до релізу та полегшення пошуку проблем.
2. Програмне забезпечення	1. Програмне забезпечення повинно запускатися від імені не привелігійованого користувача	10	Запуск програмного забезпечення від імені користувача з адміністративними повноваженнями може мати негативні наслідки в разі наявності вразливостей в програмному забезпеченні та надає зловмиснику можливість встановлення додаткового програмного забезпечення

Напрямок	Вимога	Важл.	Коментар
	2. Програмне забезпечення повинно використовувати рекомендовану версію runtime	8	Ідеальний випадок це використання однієї версії runtime для всіх компонентів системи. На практиці цього досягти складно, тому шо новіші сервіси будуть розроблятися з використанням останніх версій бібліотек та фреймворків тому рекомендується обрати одного постачальника runtime. Наприклад Azul JDK. Для Spring Boot 2 проектів 11 версію а для Spring Boot 3 – 17+. Це дозволить відслідковувати релізи лише цього виробника та спростить пошук проблем
	3. Програмне забезпечення повинно бути оптимізовано для швидкого старту	10	Швидкість запуску програмного забезпечення безпосередньо впливає на швидкість розгортання нових версій компонентів

Напрямок	Вимога	Важл.	Коментар
			або виправлення критичних помилок, тобто на MTTR, а також на швидкість автоматичного масштабування кластерів Kubernetes або AWS ECS. Тобто програмне забезпечення що стартує повільно буде не в змозі швидко масштабуватися щоб опрацьовувати запити користувачів, що призведе до простою в роботі.
	4. Програмне забезпечення повинно опрацьовувати сигнал операційної системи SIGTERM [8]	5	Kubernetes або AWS ECS використають SIGTERM для сповіщення задач про зменшення розміру кластеру або про відмову від деякий серверів в пулі в разі використання spot серверів або необхідності міграції на інший сервер для проведення обслуговування обладнання. Програмне

Напрямок	Вимога	Важл.	Коментар
			забезпечення повинно опрацьовувати SIGTERM за коректно завершувати задачі та транзакції
	5. Docker Image повинен мати мінімально можливий розмір	3	Розмір Docker Image безпосередньо впливає на швидкість розгортання та масштабування компоненту. Необхідно контролювати розмір та переконатися що не створюються непотрібні шари файлової системи під час компіляції
	6. Встановлення додаткового програмного забезпечення під час компіляції забороняється [8]	6	Встановлення додаткового програмного забезпечення на етапі компіляції збільшу час всього CI/CD процесу та може завершуватися невдачею в разі проблем на стороні постачальника дистрибутива Linux, що негативно впливає на MTTR під час розгортання виправлень

Напрямок	Вимога	Важл.	Коментар
			критичних помилок к компонені
	7. Програмне забезпечення повинно підтримувати rolling updates	8	При розгортанні нових версій програмного забезпечення паралельно працюють 2 версії. Тому важливо розробляти компоненти з використанням зворотної сумісності та покривати такі частини модульними та інтеграційними тестами. Наприклад не змінювати DTO повідомлень Kafka а розширювати їх а застарілі поля видаляти в наступному релізі.
	8. Програмне забезпечення повинно містити Health та Liveness методи API [8]	9	Kubernetes або AWS ECS потребують наявності Health та Liveness методів API які будуть використовуватися для визначення стану задачі під час розгортання. Health – повинен

Напрямок	Вимога	Важл.	Коментар
			перевіряти що програмне забезпечення має доступ до сторонніх сервісів (база даних, kafka) Liveness – повинен перевіряти шо кеш та службові дані на сервісному рівні програмного забезпечення завантаженні.
	9. Програмне забезпечення повинно містити налаштування Cors	5	Розподілене програмне забезпечення в якості frontend дуже часто використовує JavaScript застосунки шо розміщуються на іншому домені. Для їх коректної роботи потрібно увімкнути підтримку CORS
	10. Програмне забезпечення повинно виконувати 3 стадійне	8	В разі використання реляційних баз даних постає потреба використання міграцій

Напрямок	Вимога	Важл.	Коментар
	керування схемою бази даних [1]		для керування схемою даних. При розгортанні нових версій програмного забезпечення паралельно працюють 2 версії. Тому важливо розробляти компоненти з використанням зворотної бази даних та покривати такі частини модульними та інтеграційними тестами. Тобто якщо нам потрібно змінити існуючу схему ми робимо це в 3 етами: 1) Додаємо нові поля та виконуємо розгортання 2) Додаємо функціональність шо використовує нове поле та перестає використовувати стане 3) Видаляємо старі поля та робимо розгортання
3. Кібер безпека	1. Базові Docker Image повинні	10	Docker Image від індивідуальних

Напрямок	Вимога	Важл.	Коментар
	бути лише від офіційних постачальників або власні [6]		постачальників дуже часто знаходяться в неналежному стані, містять вразливості та pull запити з шкідливим програмним забезпеченням. Необхідно використовувати лише перевірених постачальників або в разі відчутності необхідного image його необхідно створити самостійно та надавати йому підтримку на рівні з іншими компонентами.
	2. Під час компіляції необхідно виконувати статичний аналіз вихідних кодів	10	З метою уникнення стандартних помилок пов’язаних з кібербезпекою необхідно виконувати статичний аналіз вихідних кодів за допомогою сканеру SonarCube та модуля розширення maven – spotbugs.

Напрямок	Вимога	Важл.	Коментар
			CI/CD повинен блокувати реліз до усунення проблем
	3. Під час компіляції необхідно виконувати перевірку наявності вразливостей в сторонніх бібліотеках	10	Під час компіляції необхідно виконувати перевірку наявності вразливостей в сторонніх бібліотеках за допомогою модуля розширення maven-dependency-check та npm autit. CI/CD повинен блокувати реліз до усунення проблем
	4. Повідомлення про помилки не повинні містити чутливої інформації	10	Повідомлення про помилки не повинні містити чутливої інформації такої як паролі, дані кредитних карток або персональні дані користувачів.
	5. У стандартному профілі налаштувань програмного забезпечення	7	

Напрямок	Вимога	Важл.	Коментар
	повинен бути вимкнений режим DEBUG		
	6. Програмне забезпечення повинно містити модульні тести контролерів шо перевіряють авторизацію та аутентифікацію	10	Найчастішою проблемою пов’язаною з кібербезпекою та витоками персональних даних є неналежна перевірка авторизації та аутентифікації. Для зменшення ризиків програмне забезпечення повинно містити модульні тести контролерів шо перевіряють авторизацію та аутентифікацію
	7. Модифіковані базові Docker Image повинні буди виконані у вигляді окремих репозиторіїв з CI/CD процесом що містить перевірку на вразливості [8]	8	В раз створення власних Docker Image повинні буди виконані у вигляді окремих репозиторіїв з CI/CD процесом що містить перевірку на вразливості за допомогою наприклад trivy. CI/CD повинен блокувати

Напрямок	Вимога	Важл.	Коментар
			реліз до усунення проблем
	8. Компоненти не повинні використовувати паролі та ключі доступу інших сервісів	9	Використання паролів та ключів доступу інших сервісів несе за собою ризики витоку інформації та призводить до проблем надійності під час процесу змани паролів
4. Моніторинг	1. Програмне забезпечення повинно буди під'єднано до централізованої системи моніторингу	10	
	2. Програмне забезпечення повинно буди під'єднано до AMP	10	
	3. Повідомлення повинні бути в форматі JSON	10	Відправка помилок в форматі JSON спрощує налаштування системи моніторингу та дозволяє використовувати MDC

Напрямок	**Вимога**	**Важл.**	**Коментар**
	4. Повідомлення про помилки сторонніх бібліотек повинні бути обмежені до рівня WARN	4	
	5. Повідомлення про помилки мають використовувати наступні рівні: - info – службові повідомлення які використовуються для відстеження процесу роботи програмного забезпечення - warn – повідомлення про помилки які не впливають на роботу програмного забезпечення або мають вбудований механізм	10	

Напрямок	Вимога	Важл.	Коментар
	повторних спроб - error - повідомлення про помилки які впливають на роботу програмного забезпечення Для повідомлень типу error повинні бути налаштовані сповіщення [1]		
	6. Для кожного компоненту необхідно створити запис в каталозі сервісів що містить наступну інформацію: - назва - опис - посилання на репозиторії - команда - контакті дані	10	Каталог сервісів ще основний інструмент керівника та команди підтримки тому він мусить містити вичерпну інформацію про кожен компонент систему для швидкого пошуку проблем та сповіщення.

Напрямок	Вимога	Важл.	Коментар
	7. Для кожного компоненту необхідно створити метрики SLO [6] з оповіщенням команд розробки та підтримки: - Latency SLO - Failure Rate SLO	10	SLO це основний інструмент визначення стану компоненту та KPI команди розробки
	8. Вразі необхідності моніторингу статистики внутрішнього стану компоненту необхідно використовувати Custom Metrics	5	
	9. Кожен компонент повинен моніторитись мінімальною кількістю метрик	10	Створення великої кількості метрик та сповіщень як не завжди будуть вірно визначати проблему зменшує сфокусованніcь команд та

Напрямок	Вимога	Важл.	Коментар
			з часом на них перестають реагувати
	10. Повинно буди створено Dashboard що відображає використання ресурсів системи моніторингу - кількість повідомлень - кількість custom metrics - Об'єм повідомлень	8	Система моніторингу при необдуманому використані може витрачати великі кошти. Необхідно виконувати контроль використання ресурсів системи моніторингу на рівні підрозділів розробки та компонентів
	11. Налаштування системи моніторингу повинно буди частиною IaC репозиторію	7	
	12. Компоненти повинні використовувати Mapped Diagnostic Context (MDC) [3]	9	

Напрямок	Вимога	Важл.	Коментар
5. Інфраструктура	1. Репозиторій вихідних кодів повинен містити CI/CD процес	10	Процес CI/CD повинен виконувати наступні кроки - аналіз коду - компіляція - тестування - пакування - розгортання
	2. Інфраструктурні параметри повинні бути максимально уніфікованими для усіх компонентів щоб спростити розгортання (порти, health та readiness api, змінні оточення)	7	
	3. Специфічні налаштування інфраструктури компонента повинні бити винесенні в		

Напрямок	Вимога	Важл.	Коментар
	репозиторій компоненту		
	4. При розгортанні нових версій компоненту повинна зберігатися значення кількості запушених задач	10	При розгортанні нових версій компонентів необхідно зберігати значення кількості реплік задачі. В іншому випадку це призводитиме до відмови в обслуговуванні через зменшення кількості реплік до значення за замовчуванням. Наприклад: значення кількості реплік задачі дорівнює 10 але на час запуску розгортання нової версії компонент був під навантаженням а автоматично масштабований до 30 реплік. Під час розгортання кількість зменшиться до 10 чого буде не достатньо для опрацювання всіх запитів. Компонент буде

Напрямок	Вимога	Важл.	Коментар
			перевантажено запитами і від перестане відповідати до тих пір поки кластер не масштабує його знову.
	5. При розгортанні повинно буди вказано кількість ресурсів (CPU, RAM) необхідних для запуску однієї задачі	10	Вказування кількість ресурсів (CPU, RAM) необхідних для запуску однієї задачі є необхідною передумовою для коректної роботи автоматичного масштабування кластерів Kubernetes або AWS ECS. Також за відсутності вимог ресурсів ядро операційної системи Linux застосовує процес OOM Killer для зупинки процесів що використовують багато ресурсів. Це призводить до непередбачуваної зупинки компонентів та нестабільної роботи системи в цілому

Напрямок	Вимога	Важл.	Коментар
	6. При розгортанні компонентів необхідно дотримуватися конвенції іменування компонентів та середовищ	10	Компоненти повинні використовувати одне ім’я для всіх оточень та різні назви оточень. Це дозволить згрупувати компоненти в каталозі сервісів та зменшити вартість ліцензій APM оскільки вони прив’язуються до імені компоненту
	7. Доступ до публічно доступних компонентів повинен виконуватись через Web Application Firewall (WAF) та систему захисту від DDoS атак [6]	8	
	8. CI/CD процес повинен генерувати Release Notes та	10	

Напрямок	Вимога	Важл.	Коментар
	відправляти повідомлення в корпоративну базу знань [7]		
6. DRP	1. Для компоненту необхідно налаштувати резервне копіювання	10	
	2. Для компоненту необхідно створити процедуру тестування розгортання з резервних копій [6]	10	
	3. Для компоненту необхідно створити процедуру дій в надзвичайних випадках [6]	10	
	4. Необхідно розробити On Call процес та	10	

Напрямок	Вимога	Важл.	Коментар
	повідомити про нього команді підтримки		
7. Введення в експлуатацію	1. Перед введенням в експлуатацію необхідно пройти аудит	10	
	2. Перед введенням в експлуатацію необхідно провести комунікацію	10	Перед введенням в експлуатацію необхідно підготувати та провести комунікацію про новий компонент. Комунікація повинна бути спрямована на інші команди розробки, системних адміністраторів, кібербезпеки на технічної підтимки
	3. Після введенням в експлуатацію необхідно проводити аудити 2 рази на рік [13]	10	

3.7 Методологія перевірки готовності до введення в експлуатацію

Підготовка до введення в експлуатацію повинна виконуватися разом з початком роботи над новим компонентом розподіленої системи. Це дає можливість мати достатньо часу щоб усі члени команди отримали відповідні знання та навички. Робота проводитися відповідно наступному процесу, activity діаграма якого представлена на рисунку 3.2:

1. На початку розробки нового компоненту розподіленої системи команда розробки повинна підготувати загальну інформацію та вимоги надійності:
    a. Назва компоненту
    b. Опис компоненту
    c. Вимоги надійності
        i. Latency SLO
        ii. Failure Rate SLO
    d. Орієнтовний бюджет інфраструктури
    e. Орієнтовний бюджет системи моніторингу
2. Команда розпочинає розробку компоненту та процесів його розгортання в тестовому та продуктивному середовищі
3. Кожна ітерація розробки повинна містити в собі задачі з списку самоперевірки
    a. Якщо при роботі над пунктом з списку самоперевірки відсутні приклади реалізації іншими командами, команді необхідно підготувати рішення, оновити список посиланням на приклад, та провести нараду з презентацією рішення.
    b. В разі реалізації рішення команда розробки бере на себе відповідальність надавати консультації по даному питанню
4. Після завершення процесу розробки готується звіт що містить в собі:
    a. Заповнену форму “Форма компоненту” з вичерпною інформацією про компонент розміщену в корпоративній базі знань
    b. Заповнену копію “Форму самоперевірки”

5. Керівник підрозділу перевіряє звіт та в разі часових обмежень переконується що виконані всі пункти з рівнем важливості більше 7
    a. Всі інші пункти необхідно виконати протягом 2 наступних ітерацій
6. Керівник підрозділу надає або відхиляє дозвіл про введення в експлуатацію
    a. В разі дозволу про новий компонент проводиться комунікація
        i. Інформується відділ технічної підтримки та моніторингу
        ii. Інформується відділ інформаційної безпеки
        iii. Інформуються інші команди розробки
        iv. В разі роботи з персональними даними інформується номінальний офіцер з даних
7. Компонент вводиться в експлуатацію
8. У випадку виникненням проблем з компонентом в процесі експлуатації проводиться аналіз причин виникнення проблеми
    a. У разі виявлення проблеми що описана в формі самоперевірки
        i. Команді розробки забороняються наступні розгортання нових версій
        ii. Проводиться повний аудит на відповідність формі самоперевірки
        iii. Розгортання нових версій дозволяється після усунення всіх проблем знайдених під час аудиту
9. Існуючі компоненти повинні пройти такий же самий процес аудиту

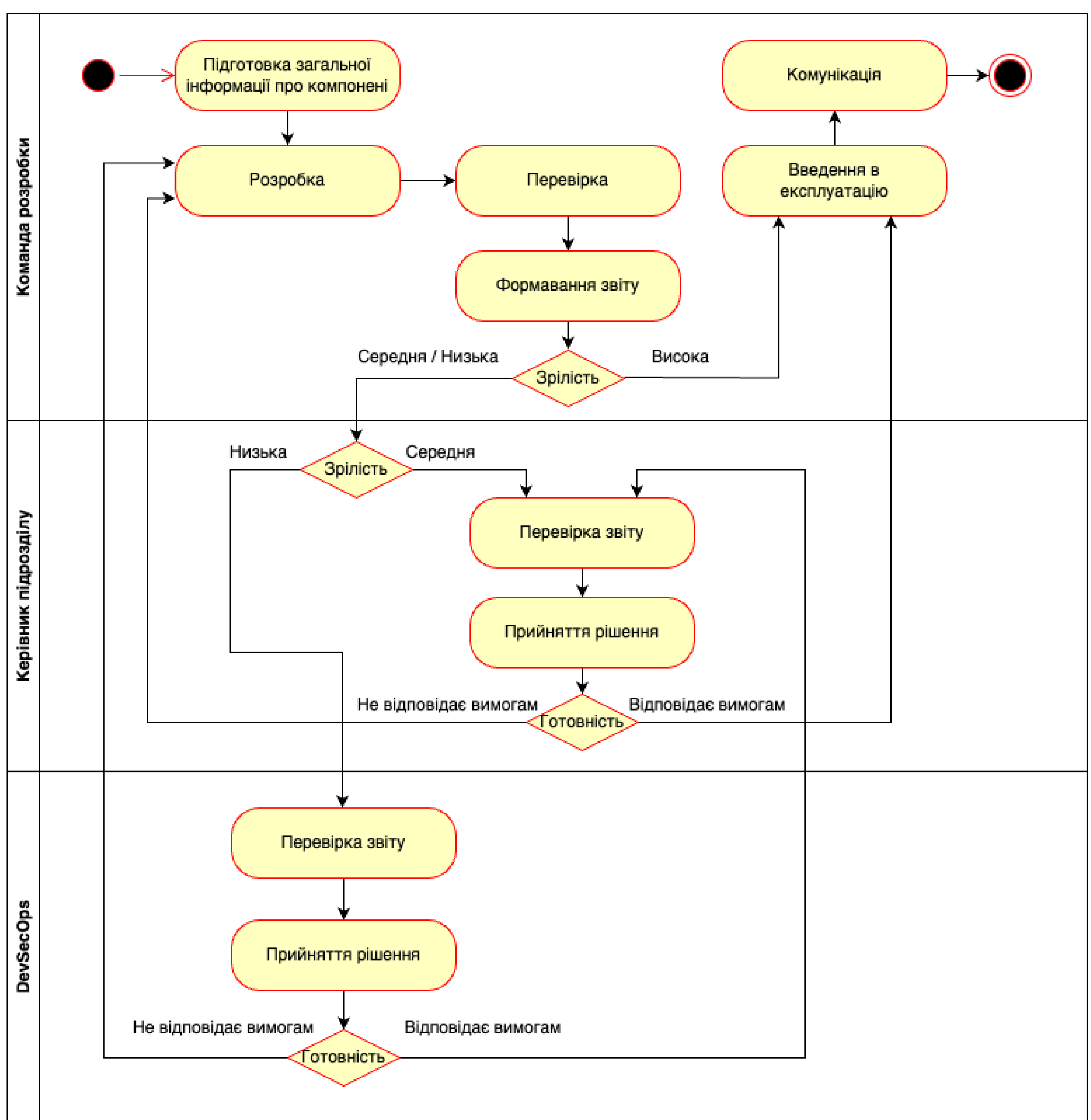

Рисунок 3.2. – Activity діаграма процесу введення в експлуатацію компоненту розподіленого програмного забезпечення

Для оптимізації процесу роботи та витрат часу вводяться рівні зрілості команд розробки:

- Високий – команда вправно виконує вимоги щодо підтримки рівня надійності власних компонентів. Команда може самостійно проводити внутрішні аудити та вводити в експлуатацію компоненти за спрощеною процедурою.

- Середній - команда не завжди вправно виконує вимоги щодо підтримки рівня надійності власних компонентів. Команда може самостійно проводити внутрішні аудити та вводити в експлуатацію компоненти за спрощеною процедурою з додатковими повними аудитами сервісів обраних випадковим методом.
- Низький - команда не виконує вимоги щодо підтримки рівня надійності власних компонентів. Дозвіл на введення в експлуатацію надається після повного аудиту.

Таблиця 3.2. Форма компоненту з прикладом заповнення

Назва компоненту	usersearch-api
Опис компоненту	Компонент індексує базу даних користувачів використовуючи Apache Solr та надає Rest API пошуку для компоненту KYC
Команда	Core
Репозиторій	https://gitlab.company.com/usersearch/api
Репозиторій Terraform	https://gitlab.company.com/usersearch/terraform
Репозиторій Helm Chart	https://gitlab.company.com/usersearch/chart
Репозиторії Docker Images	https://gitlab.company.com/usersearch/docker-solr
Середовище запуску	Kubernetes
Latency SLO	99.9%
Failure Rate SLO	99.9%
Вартість інфраструктури	$1600
Вартість системи моніторингу	$1250
Документація	https://wiki.company.com/services/usersearch/info
DRP Документація	https://wiki.company.com/services/usersearch/drp
On Call графік	https://wiki.company.com/services/usersearch/oncall
Дата	15.01.2024

На початку роботи усім командам розробки присвоюється високий рівень зрілості, який може знижуватися в процесі роботи над компонентами в разі виникнення проблем. Рівень зрілості розраховуються пропорційно до кількості підтримуваних компонентів. В таблиці 3.2 представлено список вхідних даних з прикладом заповнення, що надають відомості компонент розподіленого програмного забезпечення що потрібно підготувати для початку проходження оцінки про готовність до введення в експлуатацію.

3.8 Результати впровадження пілотного проекту

Методологія була впроваджена в пілотному режимі в компанії Sabertech Systems S.L, що займається розробкою та підтримкою розподіленого програмного забезпечення в галузі електронної комерції. Впровадження було виконано для 15 команд розробки та отримала схвальні відгуки від керівників структурних підрозділів та розробників програмного забезпечення. Відмічається простота та зрозумілість процесу роботи з методологією, зручність та корисність форми самоперевірки. Позитивні відгуки про методологію може свідчити про те що її впровадження не призведе до зниження індексу eNPS.

Пілотний проект розпочався з впровадження каталогу сервісів, створення SLO та бюджетів системи моніторингу. Під налаштування виявлено та усунуто проблеми описані в таблиці 3.3.

Таблиця 3.3. Список виявлених та усунутих проблем

Було	Стало	Коментар
Кількість компонентів 140	Кількість компонентів 90	Команди розробки звітували керівництву що розподілена система використовує 140 компонентів. Після запровадження каталогу сервісів кількість компонентів скоротилась до 90. Інші 50 компонентів копіями що

Було	Стало	Коментар
		розгорнуті в тих чи інших середовищах під іншим іменем. Впровадження уніфікованого іменування дозволило впорядкувати інформацію, безпечно видалити копії компонентів що не використовуються та знизити вартість хмарної інфраструктури на 15%, модуля APM системи моніторингу на 10%, модуля керування журналом помилок на 10%, та модуля Custom Metrics на 7%. Впровадження каталогу сервісів дало чітке розуміння про кількість компонентів та середовищ в яких вони виконуються.
Відсутні метрики SLO	Запровадженні метрики SLO	Після запровадження метрик Latency SLO та Failure Rate SLO для усіх 90 компонентів розподіленої системи було виявлено шо 4 компонента не дотримуються Latency SLO та 9 Failure Rate SLO. Для усунення проблем було проведено поглиблений аналіз та виявлено компоненти з невідповідним Latency SLO мають проблеми з продуктивністю пов’язанні з неефективними запитами до бази даних. Оптимізація запитів дозволила зменшити використання хмарних ресурсів, зменшити час відповіді компонентів та вартість модуля системи моніторину Custom Metrics на 7%.

Було	Стало	Коментар
Відсутній бюджет системи моніторингу	Впроваджено бюджет системи моніторингу (на даному етапі однаковий для всіх команд)	Впровадження бюджетів системи моніторингу показало що 90 компонентів використовують майже рівномірно APM та Custom Metrics але модуль керування журналами помилок що є cost driver, 80% витрат приходиться на 5 компонентів. Проблема була детально проаналізована та проведена оптимізація логування компонентів, що призвело до зниження вартості системи моніторингу в цілому майже на 47%

За час проведення тестування пілотному режимі було введено в експлуатацію 4 нових компоненти:

1. Компонент інтеграції с платіжною системою;
2. Компонент пошуку в базі даних кленів що використовується в панелі керування підсистеми Know Your Client (KYC);
3. Компонент підсистеми маркетингу для керування акціями;
4. Компонент двох-факторної аутентифікації;

3 компоненти пройшли повні перевірки для визначення середнього часу що необхідний для виконання процесу, щоб надати продуктовій команді значення часу які необхідно закладати про плануванні нового компоненту. Середній час склав 1 день.

Для введення в експлуатацію використовувалась форма самоперевірки наведена в таблиці 3.1. В результаті перевірки було виявлено недотримання рекомендацій з розділів “Програмне забезпечення” (2) та “Інфраструктура” (5), невиконання яких призвели б до проблем в разі введення компонентів в експлуатацію:

- 2.3 Програмне забезпечення повинно бути оптимізовано для швидкого старту
- 5.5 При розгортанні повинно бути вказано кількість ресурсів (CPU, RAM) необхідних для запуску однієї задачі
- 5.7 Доступ до публічно доступних компонентів повинен виконуватись через Web Application Firewall (WAF) та систему захисту від DDoS атак

Етап підготовки до застосування методології дозволив впорядкувати інформацію про компоненти та отримати інтерактивний звіт про стан розподіленої систем вцілому, впровадити метрики SLO які будуть використовуватися в якості KPI для підрозділів розробки, а також мав позитивний економічний ефект у вигляді зниження вартості системи моніторингу на 47% та хмарної інфраструктури на 15%.

В процесі роботи з формою самоперевірки команди розробки отримали бачення що необхідно впровадити в компонентах для успішного введення в експлуатацію, та оцінили потенційні прогалини в знаннях предметної галузі. Результатом чого став запит керівників структурних підрозділів про проведення семінару про особливості роботи модуля автоматичного масштабування систем керування кластерами Kubernetes та AWS ECS, також принцип роботи системи захисту CloudFlare та її налаштування за допомогою Terraform. Даний запит підтверджує присутність «бункерів знань» та необхідність роботи над їх усуненням шляхом обміну досвідом між командами розробки, проведенням семінарів підрозділами кібербезпеки та інформаційно-технологічного обслуговування.

Наступними кроками будуть запровадження бюджетів інфраструктури та проведення перевірок компонентів, що вже знаходяться в експлуатації з метою виявлення та усунення потенційних проблем.

3.9 Результати по розділу 3

Оцінку готовності до введення в експлуатації компонентів розподіленого програмного забезпечення можливо виконувати ефективно та з використанням наявних ресурсів в компанії, проте цей процес потребує клопіткої підготовки,

введення звітів та системи метрик що дозволяють відстежувати ефективність роботи команд розробки та якості програмного забезпечення що вони розробляють.

ВИСНОВКИ

У кваліфікаційній роботі на здобуття ступеню «магістр» проаналізовано проблеми створення монолітного та розподіленого програмного забезпечення, методи оцінки готовності до введення в експлуатацію. На основі цього розроблено методологію оцінки готовності до введення в експлуатацію розподіленого програмного забезпечення.

Розроблена методологія була впроваджена в тестовому режимі в компанії Sabertech Systems S.L та надала можливість командам розробки та керівникам структурних підрозділів ефективно проводити оцінку готовності до введення компонентів системи електронної комерції що розроблена з використанням архітектури розподілених систем. Додатково впровадження дозволило отримати зрозумілі метрики надійності що використовуються в якості KPI та позитивний економічний ефект у вигляді значного зниження загальної вартості володіння системою.

Проблеми з надійністю системи електронної комерції що виникали в компанії в наслідок неналежної оцінки готовності до введення в експлуатацію компонентів було усунуто.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Building Secure and Reliable Systems [Електронний ресурс] / [H. Adkins, B. Beyer, P. Blankinship та ін.]. – 2020. – Режим доступу до ресурсу: https://google.github.io/building-secure-and-reliable-systems/
2. Bryane D. Full Cycle Developers at Netflix: from Mindsets to Self-Service Tooling [Електронний ресурс] / Daniel Bryane. – 2018. – Режим доступу до ресурсу: https://www.infoq.com/news/2018/06/netflix-full-cycle-developers/.
3. Burns B. Kubernetes: Up and Running: Dive into the Future of Infrastructure / B. Burns, J. Beda, K. Hightower., 2022.
4. Harrison N. Pattern Languages of Program Design 3 / Neil Harrison., 1997.
5. Hoffman A. Web Application Security: Exploitation and Countermeasures for Modern Web Applications / Andrew Hoffman., 2020.
6. Shimon I. Deploy Containers on AWS: With EC2, ECS, and EKS / Ifrah Shimon., 2019.
7. Merode H. Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines / Henry Merode., 2023
8. Petoff J. Site Reliability Engineering: How Google Runs Production Systems / J. Petoff, N. Murphy, B. Beyer., 2016.
9. Rice L. Container Security: Fundamental Technology Concepts that Protect Containerized Applications / Liz Rice., 2020.
10. Sachania E. Breaking Knowledge Silos with Knowledge Management [Електронний ресурс] / Ekta Sachania. – 2023. – Режим доступу до ресурсу: https://www.kminstitute.org/blog/breaking-knowledge-silos-knowledge-management.
11. Siocon G. Employee Net Promoter Score (eNPS): The Ultimate 2024 Guide [Електронний ресурс] / Gem Siocon. – 2024. – Режим доступу до ресурсу: https://www.aihr.com/blog/employee-net-promoter-score-enps/.
12. Vehent J. Securing DevOps: Security in the Cloud / Julien Vehent., 2018.

13. Whitman M. Principles of Incident Response and Disaster Recovery / Michael Whitman., 2021.

ДОДАТКИ

ДОДАТОК А. ПРИКЛАД НАЛАШТУВАННЯ SLO ЗА ДОПОМОГОЮ TERRAFORM В СИСТЕМІ МОНІТОРИНГУ DATADOG

```
gateway-redirector  = {
        team = "payments",
        latency_crit = 4.0,
        latency_warn = 1.8,
        latency_slo_crit = 99.80,
        latency_slo_warn = 99.85,
        avail_slo_crit = 99.80,
        avail_slo_warn = 99.85
},

resource "datadog_service_level_objective" "availability_slo" {
  for_each = var.service_list

  name        = "Availability SLO for ${each.key} service"
  type        = "metric"
  description = "This SLO tracks the availability of the ${each.key} service. Availability is measured as the number of successful requests divided by the number of total requests for the service"
  query {
    numerator   = "sum:trace.servlet.request.hits{env:prod,service:${each.key}}.as_count() - sum:trace.servlet.request.errors{env:prod,service:${each.key}}.as_count()"
    denominator = "sum:trace.servlet.request.hits{env:prod,service:${each.key}}.as_count()"
  }
```

```
  thresholds {
    timeframe = "7d"
    target    = each.value.avail_slo_crit
    warning   = each.value.avail_slo_warn
  }

  tags = ["managed-by:terraform", "service:${each.key}", "env:prod",
"team:${each.value.team}"]
}

resource "datadog_monitor" "availability_alert" {
  for_each = var.service_list

  name              = "Availability ${each.key} > ${each.value.avail_slo_crit}%"
  type              = "slo alert"
  enable_logs_sample = true
  message           = "Service ${each.key} availability is less ${each.value.avail_slo_crit}%.
${var.slack_alert_channel}"
  priority          = 3

  query = <<EOT

error_budget("${datadog_service_level_objective.availability_slo[each.key].id}").over("7
d") > ${each.value.avail_slo_crit}
  EOT

  monitor_thresholds {
```

```
    critical = each.value.avail_slo_crit
  }
  tags = ["managed-by:terraform", "service:${each.key}", "env:prod",
"team:${each.value.team}"]
}

resource "datadog_monitor" "latency_alert" {
  for_each = var.service_list

  name     = "Latency ${each.key} > ${each.value.latency_crit} sec"
  type     = "query alert"
  message  = "Service ${each.key} latency is high. There is no alert slack channel here.
Alert is sent only by SLO, when it goes down less than SLO threshold."
  priority = 3
  query    = <<EOT
    avg(last_6h):median_9(trace.servlet.request.duration{env:prod,service:${each.key}}) >
${each.value.latency_crit}
  EOT

  monitor_thresholds {
    critical = each.value.latency_crit
    warning  = each.value.latency_warn
  }
  tags = ["managed-by:terraform", "service:${each.key}", "env:prod",
"team:${each.value.team}"]
}

resource "datadog_service_level_objective" "latency_slo" {
```

```
  for_each = var.service_list

  name        = "Latency SLO for ${each.key} service"
  type        = "monitor"
  description = "This SLO tracks the latency of the ${each.key} service."
  monitor_ids = [datadog_monitor.latency_alert[each.key].id]

  thresholds {
    timeframe = "7d"
    target    = each.value.latency_slo_crit
    warning   = each.value.latency_slo_warn
  }

  tags = ["managed-by:terraform", "service:${each.key}", "env:prod",
"team:${each.value.team}"]
}

resource "datadog_monitor" "latency_slo_alert" {
  for_each = var.service_list

  name               = "Latency ${each.key} alert"
  type               = "slo alert"
  enable_logs_sample = true
  message            = "Service ${each.key} latency is high. "
  priority           = 3
  query              = <<EOT
```

```
error_budget("${datadog_service_level_objective.latency_slo[each.key].id}").over("7d")
> ${each.value.latency_slo_crit}
  EOT

  monitor_thresholds {
    critical = each.value.latency_slo_crit
  }

  tags = ["managed-by:terraform", "service:${each.key}", "env:prod",
"team:${each.value.team}"]
}
```

ДОДАТОК Б. ФРАГМЕНТ ПРОГРАМИ СТВОРЕННЯ ПРИВАТНОГО ІНДЕКСУ ДЛЯ LLM МОДЕЛІ

```
import gradio as gr
import os
import sys
import uuid

from gpt_index.readers.schema.base import Document
from gpt_index import GPTListIndex, GPTSimpleVectorIndex, LLMPredictor,
PromptHelper
from langchain import OpenAI
from sqlalchemy.dialects.mysql import LONGTEXT

from app import db

class CustomModel(db.Model):
    __tablename__ = 'custom_models'
    __table_args__ = {
        'mysql_engine': 'InnoDB',
        'mysql_charset': 'utf8',
        'mysql_collate': 'utf8_general_ci'
    }

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)

    name = db.Column(db.String(255))
```

```
index = db.Column(db.String(1024), nullable=False, default=generate_index_name)
list_index = db.Column(db.String(1024), nullable=False, default=generate_index_name)

model_name = db.Column(db.String(255), nullable=False, default="text-davinci-003")

max_input_size = db.Column(db.Integer, index=True, nullable=False, default=4096)
num_outputs = db.Column(db.Integer, index=True, nullable=False, default=512)
max_chunk_overlap = db.Column(db.Integer, index=True, nullable=False, default=20)
chunk_size_limit = db.Column(db.Integer, index=True, nullable=False, default=600)
temperature = db.Column(db.Float, index=True, nullable=False, default=0.7)

created_on = db.Column(db.DateTime, default=func.now())
updated_on = db.Column(db.DateTime, default=func.now(), onupdate=func.now())

def save(self):
    db.session.add(self)
    db.session.commit()

def as_dict(self):
    return {
        c.name: getattr(self, c.name) for c in self.__table__.columns
    }

@classmethod
def get_model_by_name(cls, name):
    return cls.query.filter_by(name = name).first()

def gpt_query(self, text, response_mode):
```

```
    if not os.path.exists(self.index):
        raise Exception("Model not found")

    index = GPTSimpleVectorIndex.load_from_disk(self.index)
    response = index.query(text, response_mode = response_mode)

    return response

def gpt_list_query(self, text, response_mode):
    if not os.path.exists(self.list_index):
        raise Exception("Model not found")

    index = GPTListIndex.load_from_disk(self.list_index)
    response = index.query(text, response_mode = response_mode)

    return response

def gpt_list_train(self):
    documents = []
    for r in CustomModelDocument.get_documents_by_custom_model_id(self.id):
        documents.append(Document(r.body))

    index = GPTListIndex(documents)

    print("build gpt_list_train %s", self.list_index)

    if self.list_index == '':
        self.list_index = generate_index_name()
```

```
            db.session.commit()

        index.save_to_disk(self.list_index)

    def gpt_train(self):
        prompt_helper = PromptHelper(
            self.max_input_size, self.num_outputs, self.max_chunk_overlap,
self.chunk_size_limit)

        llm_predictor = LLMPredictor(
            llm = OpenAI(
                temperature = 0.7,
                model_name = self.model_name,
                max_tokens = self.num_outputs
            )
        )

        documents = []
        for r in CustomModelDocument.get_documents_by_custom_model_id(self.id):
            documents.append(Document(r.body))

        print("build gpt_train %s", self.index)

        index = GPTSimpleVectorIndex(
            documents,
            llm_predictor = llm_predictor,
            prompt_helper = prompt_helper)
        index.save_to_disk(self.index)
```