Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Соколянського Дмитра Васильовича*
	(ПІБ)
академічної групи	*122м-22з-1*
	(шифр)
спеціальності	*122 Комп'ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«122 Комп'ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження відмовостійкого середовища на базі хмарної платформи Azure*

_____*Д.В. Соколянський*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*доц. Спірінцев В.В.*			
економічний				
Рецензент				
Нормоконтролер	*проф. Лактіонов І.С.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122м-22з-1* *Соколянському Дмитру Васильовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження відмовостійкого середовища на базі хмарної платформи Azure*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1228 -с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес розробки відмовостійкої структури програми на базі хмарних сервісів Azure.

Предмет досліджень – архітектурні моделі, що забезпечують безвідмовність роботи типового інтернет магазину на базі сервісів Azure.

Мета НДР – підвищення ефективності хмарної платформи Azure на основі розробки та дослідження середовища, яке буде максимально безвідмовне і незалежне від зовнішніх чинників.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження, методологія розробки додатку і аналіз інфраструктури.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає в тому, що дістали подальшого розвитку інноваційні механізми відновлення працездатності та надійності роботи застосунку при виникненні випадків простою та збоїв, за рахунок оптимізації налаштувань сервісів Azure, що сприяє забезпеченню ефективного використання ресурсів і мінімізації виробничих витрат.

Практична цінність результатів полягає у підвищенні надійності програми та зниженні ймовірності простоїв, що важливо для забезпечення стабільності бізнес-процесів та задоволення потреб користувачів.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє безвідмовність роботи типового інтернет магазину на базі сервісів Azure. В результаті роботи повинен бути розроблений програмний комплекс для розробки та аналізу середовища, яке буде максимально безвідмовне і незалежне від зовнішніх чинників.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	10.09.2023-30.09.2023
Побудова нечіткої моделі представлення даних для вирішення задачі ідентифікаційної експертизи	01.10.2023-31.10.2023
Створення автоматизованої системи для вирішення задачі ідентифікаційної експертизи бензинів	01.11.2023-04.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки скороченню капітальних витрат на локальну інфраструктуру.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки підвищенню надійності програм, розміщених у хмарному середовищі та зниження ймовірності їх простоїв.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ *Спірінцев В.В.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Соколянський Д.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *09.10.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *04.12.2023*

РЕФЕРАТ

Пояснювальна записка: 94 стор., 34 рис., 3 таблиці, 4 додатка, 19 джерел.

Об'єкт дослідження: процес розробки відмовостійкої структури програми на базі хмарних сервісів Azure.

Предмет дослідження: архітектурні моделі, що забезпечують безвідмовність роботи типового інтернет магазину на базі сервісів Azure.

Мета роботи: підвищення ефективності хмарної платформи Azure на основі розробки та дослідження середовища, яке буде максимально безвідмовне і незалежне від зовнішніх чинників.

Методи дослідження. Для вирішення поставлених задач використані методи: аналізу режиму відмов, аналітики даних для прогнозування та виявлення проблем, які можуть призвести до відмов, теорії баз даних, об'єктно-орієнтоване програмування.

Новизна полягає в тому, що дістали подальшого розвитку інноваційні механізми відновлення працездатності та надійності роботи застосунку при виникненні випадків простою та збоїв, за рахунок оптимізації налаштувань сервісів Azure, що сприяє забезпеченню ефективного використання ресурсів і мінімізації виробничих витрат.

Практична цінність результатів полягає у підвищенні надійності додатку та зниженні ймовірності його простоїв, що важливо для забезпечення стабільності бізнес-процесів та задоволення потреб користувачів.

Область застосування. Розроблена інформаційна система може застосовуватися на замовлення бізнесу, для забезпечення надійності та стабільності роботи додатків та сервісів.

Значення роботи та висновки. Удосконалена методика дозволяє проектувати інформаційні системи зі значним скороченням як капітальних витрат, так і операційних, що підтверджується розробленим програмним продуктом в даній роботі.

Прогнози щодо розвитку досліджень. Покращити інформаційну систему, додавши метод контейнеризації з використання Azure Kubernetes Service, з метою оптимізації роботи додатків та підвищення їх відмовостійкості.

Список ключових слів: додаток, MERN Stack, Node.js, Express.js, Mongo DB, Azure, хмара, Cosmos DB, інформаційна система.

ABSTRACT

Explanatory note: 94 pages, 34 figures, 3 tables, 4 applications, 19 sources.

Object of research: the process of developing a fault-tolerant program structure based on Azure cloud services.

Subject of research: architectural models that ensure the uninterrupted operation of a typical online store based on Azure services.

Purpose of Master's thesis: increasing the efficiency of the Azure cloud platform based on the development and research of an environment that will be as error-free and independent of external factors as possible.

Research methods. To solve the problems, the following methods were used: failure mode analysis, data analytics for forecasting and identifying problems that can lead to failures, database theory, object-oriented programming.

Originality of research is associated with the further development of innovative mechanisms for restoring the performance and reliability of the application in the event of downtime and failures, due to the optimization of Azure service settings, which helps to ensure the efficient use of resources and minimize production costs.

Practical value of the results consists of increasing the reliability of the application and reducing the probability of its downtime, which is important for ensuring the stability of business processes and meeting the needs of users.

Scope of application. The developed information system can be applied to business orders to ensure the reliability and stability of applications and services.

The value of the work and conclusions. The improved technique allows designing information systems with a significant reduction of both capital costs and operating costs, which is confirmed by the software product developed in this work.

Research forecast and development. To improve the information system by adding a containerization method using Azure Kubernetes Service, in order to optimize the operation of applications and increase their fault tolerance.

Keywords: application, MERN Stack, Node.js, Express.js, Mongo DB, Azure, cloud, Cosmos DB, information system.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних;

IaaC – Infrastructure as a Code;

CI/CD – Continuous Integration and Continuous Delivery;

RBAC – Role-based access control;

ACL – Access-control list;

SLA – Service-level agreement;

FMA – Failure mode analysis;

TCO – Total Cost of Ownership;

CAPEX – CAPital EXpenditure;

OPEX – Operating expenses;

ЗМІСТ

ВСТУП

Актуальність теми. У сучасну цифрову епоху створення веб-додатків стало невід'ємною частиною зростання та розширення бізнесу. Сучасні технології, такі як аналітика великих даних, штучний інтелект, розміщення веб- та мобільних додатків потребують великої обчислювальної потужності. Хмарні обчислення та хмарні платформи пропонують підприємствам альтернативу побудові внутрішньої інфраструктури. Завдяки хмарним обчисленням будь-хто, хто має доступ до Інтернету, може користуватися перевагами масштабованих обчислювальних потужностей за принципом plug and play. Оскільки це позбавляє організації від необхідності інвестувати та підтримувати дорогу інфраструктуру, воно стало дуже популярним рішенням. Є багато компаній, які пропонують хмарні платформи для розробки, керування та розгортання програм.

Кожен веб-додаток повинен підпадати під певні критерії, для того, щоб клієнти були задоволені використанням продукту:

- збереження даних у захищеному сховищі;
- відмовостійкість;
- динамічна масштабованість;
- захист від зовнішніх загроз;

Для виконання цих критеріїв необхідно розмістити застосунок на захищеній інфраструктурі, яка надасть змогу працювати 365 днів на рік без перебоїв. Розміщення на власних ресурсах чи в орендованому дата-центрі може коштувати дуже великих грошей та часу на налаштування, а також необхідність постійної підтримки, що буде тільки затримувати розвиток бізнесу. Тому найбільш підходящим варіантом буде розміщення на хмарних ресурсах, що не тільки зменшить витрати на створення початкової інфраструктури, а і дозволить розгорнути застосунок в будь-якій точці світу за лічені секунди.

Мета кваліфікаційної роботи: формування, розробка та аналіз середовища, яке буде максимально безвідмовне і незалежне від зовнішніх чинників.

Для вирішення поставленої мети необхідно вирішити наступні задачі:

- розглянути особливості масштабування, балансування навантаження, використання Azure Cosmos DB бази даних, моніторингу та журналування, резервного копіювання та відновлення;
- визначити основні вимоги, які повинні бути враховані при розробці
- програмного продукту. Ці вимоги охоплюють функціональні характеристики, інформаційну безпеку, склад і параметри технічних засобів, інформаційну та програмну сумісність, а також зміст інформаційного наповнення системи;
- розробити веб-орієнтований додаток та описати його структуру, наповнення та алгоритми взаємодії;

Об'єктом дослідження є процес розробки відмовостійкої структури програми на базі хмарних сервісів Azure.

Методи дослідження. Для вирішення поставлених задач використані методи: аналізу режиму відмов, аналітики даних для прогнозування та виявлення проблем, які можуть призвести до відмов, теорії баз даних, об'єктно-орієнтоване програмування.

Предметом дослідження є архітектурні моделі, що забезпечують безвідмовність роботи типового інтернет магазину на базі сервісів Azure.

Наукова новизна роботи полягає в тому, що дістали подальшого розвитку інноваційні механізми відновлення працездатності та надійності роботи застосунку при виникненні випадків простою та збоїв, за рахунок оптимізації налаштувань сервісів Azure, що сприяє забезпеченню ефективного використання ресурсів і мінімізації виробничих витрат.

Практичне значення отриманих в роботі результатів полягає у:

- підвищенні надійності програми та зниженні ймовірності простоїв, що важливо для забезпечення стабільності бізнес-процесів та задоволення потреб користувачів, оптимізації використання обчислювальних ресурсів завдяки автоматизованому масштабуванню та управлінню ресурсами в

залежності від навантаження, що може заощадити витрати на IT-інфраструктуру;

- скороченні часу відновлення після збоїв, що допомагає мінімізувати втрати прибутку та надає більш надійні послуги;
- можливості легкого масштабування програми в залежності від потреб бізнесу, що змінюються, що забезпечує гнучкість у розвитку;
- впровадженні сучасних методів безпеки та відповідності, що забезпечує захист даних та відповідність нормативним вимогам;
- зниженні ризиків, пов'язаних з потенційними збоями у роботі системи, завдяки стійкості до відмови архітектурі;
- скороченні витрат на підтримку високої доступності і відмовостійкості за допомогою ефективного використання хмарних ресурсів;
- створенні бази для розширення бізнесу та надання можливості для запуску нових продуктів та послуг.

Особистий внесок автора. Було створено додаток типового інтернет магазину, використовуючи MERN стек і розміщено його у хмарній платформі Azure. Також було здійснено дослідження відмовостійкості отриманого середовища.

Структура та обсяг кваліфікаційної роботи. Відповідно до мети, завдань і предмета дослідження кваліфікаційна робота складається з реферату, вступу, трьох основних розділів і висновків, списку використаних джерел та 4 додатків. Загальний обсяг роботи містить 94 сторінки друкованого тексту, із них основної частини - 63 сторінок з 34 рис., спеціальної – 19 сторінок, списку використаних джерел з 18 найменувань на 2 сторінках, 4 додатках на 15 сторінках.

РОЗДІЛ 1
АНАЛІЗ ТЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Хмарні платформи

1.1.1. Аналіз і характеристика послуг хмарних обчислень

Використання хмарних ресурсів значно полегшує початкове розгортання та запуск додатку. Використовуючи інструментарій провайдера хмарних послуг, можна створити ізольовану мережу і налаштувати відмовостійке середовище, що дозволить забезпечити максимально надійну роботу додатку. Найчастіше провайдери хмарних послуг мають сервіси, які дозволяють використовувати обчислювальні рішення, які необхідні виходячи із існуючих потреб.

Виділяють 3 основних категорії послуг хмарних обчислень [12, 17]:

1. Infrastructure as a Service (IaaS) – «інфраструктура як послуга». IaaS-провайдери надають замовнику обчислювальну інфраструктуру (сервери, сховища даних, операційні системи та мережеві ресурси) для розгортання та запуску власних програмних рішень. Варіант підійде компаніям, потреба яких у ресурсах не однакова в різні моменти часу - бувають сплески потреб, але вони поступово спадають (або організація швидко зростає, і виникає проблема постійного масштабування інфраструктури). Також IaaS буде оптимальним рішенням для невеликих кампаній, стартапів, коли компанія не має достатньо коштів на створення власної інфраструктури.

2. Platform as a Service (PaaS) – «платформа як послуга». Провайдер хмарних сервісів надає замовнику готове програмне середовище та інструменти для його налаштування. Елементами PaaS є апаратне забезпечення, операційна система, СУБД, проміжне ПЗ, інструменти тестування та розробки. Таку платформу клієнт може налаштувати під свої потреби, зробивши з неї майданчик для тестування або систему для автоматизації системи управління.

Такий вид сервісу користується особливою популярністю у розробників програмного забезпечення.

3. Software as a Service (SaaS) – «програмне забезпечення як послуга». Розробник програмної платформи надає віддалений доступ до неї клієнту. Наприклад, саме за моделлю SaaS корпорація Microsoft забезпечує клієнтам користування MS Office Suite (Office Web Apps) поряд із SharePoint Server, Exchange Server та іншими сервісами та додатками. Також за моделлю SaaS надається веб-електронна пошта, як-от Gmail або Outlook. Коли ви користуєтеся веб-службою електронної пошти, ваші повідомлення та вкладення зберігаються на серверах постачальника електронної пошти, а не на вашому комп'ютері.

Іншим прикладом хмарних обчислень є служби онлайн-зберігання документів і спільної роботи, такі як Google Docs або Microsoft Office 365. Ці служби дають змогу створювати, редагувати та ділитися документами з іншими через Інтернет. Ваші документи зберігаються на серверах сервісу, а не на вашому комп'ютері.

Останім часом також Functions as a Service (FaaS) - «функція як послуга», яку також називають «безсерверною» (serverless), набуває величезної популярності. Оскільки компаніям не потрібно інвестувати у великі сервери, вони можуть вибрати послугу, яка розширить їхні вимоги до серверів відповідно до потреб їхніх програм. Очевидно, він не буде справді безсерверним — сервери все одно будуть присутні, але користувачам не доведеться втручатися в технічні нюанси та конфігурації. Це допоможе зробити інновації більш доступними для компаній і створити новий досвід для користувачів.

На сьогоднішній день у світі існує велика кількість хмарних хостингів, які надають широкий спектр можливостей для масштабування та розробки on-line додатків. Кожен провайдер має свої особливості, а також ресурси, які можуть задовольнити потреби бізнесу. Клієнти можуть використовувати 4 типи побудови хмарної інфраструктури, використовуючи різні типи хмар [12] :

– Публічна хмара (Public) - побудова інфраструктури використовуючи ресурси провайдера хмарних послуг. Цей продукт пропонується клієнтам

хмарними провайдерами, а ресурси доступні через загальнодоступний Інтернет. Про все, що стосується інфраструктури, дбають провайдери.

- Приватна хмара (Private) - побудова інфраструктури з використанням власних ресурсів. Організація оплачує і керує інфраструктурою та персоналом і користується звичайними перевагами хмарних обчислень, такими як масштабованість і спільне використання ресурсів за допомогою віртуалізації.
- Гібридна хмара (Hybrid) - поєднує публічну та приватну моделі, з'єднуючи їх через Інтернет та віртуальні приватні мережі. Гібридна модель ідеально підходить для підприємств, яким потрібна зовнішня віртуальна резервна копія для пом'якшення наслідків стихійного лиха або якщо організація використала всі власні ресурси та потребує додаткової обчислювальної потужності. Гібрид особливо добре працює, якщо дані, які потрібно швидко доставити користувачам і до яких часто звертаються, можна зберігати в публічній загальнодоступній хмарі, таким чином звільняючи місце для зберігання приватних і конфіденційних даних у приватній хмарі. Використання кількох послуг від різних постачальників може ускладнити роботу. Налаштування гібридної хмари допоможе спростити складну природу та оптимізувати роботу користувача.
- Мульти хмара (Multi) - побудова інфраструктури з використанням декількох хмар від різних провайдерів.

На сьогоднішній день побудова інфраструктури для розгортання додатків на базі хмарних ресурсів, є одним із основних напрямків веб розробки та є дуже привабливою для розвитку та автоматизації процесів бізнесу.

Зазвичай, провайдери хмарних послуг мають різні сервіси, які розрізняються показниками відмовостійкості, ефективності роботи та вартості використання. Azure має багато суміжних сервісів, які мають схожий функціонал, але краще підходять для розв'язання певних задач.

1.1.2. Роль хмарних технологій в IT індустрії

Хмара кардинально змінила правила розвитку індустрії технологій, і її вплив лише зростає. Є багато причин, чому хмара така важлива.

По-перше, це дуже зручно. Ви можете отримати доступ до своїх даних і програм з будь-якого комп'ютера або мобільного пристрою, підключеного до Інтернету.

По-друге, це ефективно. Хмара може зберігати багато даних, і її легко збільшувати або зменшувати відповідно до змін ваших потреб.

По-третє, вона гнучка. Ви можете використовувати хмару для різних цілей, від зберігання до обчислювальної потужності для програмного забезпечення як послуги (SaaS).

По-четверте, і, мабуть, найважливіше, хмара безпечна. Ваші дані зберігаються в захищених центрах обробки даних і резервні копії створюються в кількох місцях. Це значно зменшує ймовірність втрати чи зламу ваших даних.

1.1.3. Основні елементи хмарних обчислень

Ключовими елементами хмарних обчислень є:

- Еластичність: можливість збільшення чи зменшення масштабу за потреби, щоб задовольнити попит.
- Оплата за використання: ви платите лише за ресурси, які використовуєте, коли ви їх використовуєте.
- Самообслуговування: можливість надавати власні ресурси та керувати ними без необхідності проходити тривалий процес затвердження.
- Спільні ресурси: можливість ділитися ресурсами з іншими користувачами, щоб підвищити ефективність і зменшити витрати.
- Багатокористувацький доступ: можливість мати кілька користувачів на одній платформі, при цьому кожен користувач не має власних виділених ресурсів.

1.1.4. Переваги хмарних платформ

1. Економія: одна з головних переваг хмарних обчислень полягає в тому, що вони можуть допомогти зменшити витрати. Наприклад, підприємствам більше не потрібно інвестувати в дороге локальне обладнання та програмне забезпечення. Натомість вони можуть отримати доступ до хмарних додатків і послуг на основі оплати за використання.

2. Масштабованість: хмарні обчислення дуже масштабовані. Це означає, що підприємства можуть легко збільшувати або зменшувати використання хмарних ресурсів у міру зміни потреб.

3. Гнучкість: ще одна перевага хмарних обчислень полягає в тому, що вони пропонують більшу гнучкість, ніж традиційна локальна IT-інфраструктура. Наприклад, підприємства можуть швидко надавати нові ресурси, коли вони їм потрібні, а також можуть легко вивільняти їх, коли вони їм більше не потрібні.

4. Гнучкість: хмарні обчислення можуть допомогти підприємствам досягти більшої гнучкості. Це означає, що вони можуть краще реагувати на зміни ринкових умов і можуть швидко розгортати нові програми та послуги.

5. Покращена безпека: хмарні обчислення можуть запропонувати покращену безпеку порівняно з традиційною локальною IT-інфраструктурою. Це пояснюється тим, що постачальники хмарних послуг мають досвід у сфері безпеки та можуть запропонувати різноманітні функції безпеки, наприклад шифрування даних і виявлення вторгнень.

6. Розширена співпраця: хмарні обчислення можуть допомогти покращити співпрацю між співробітниками. Наприклад, вони можуть простіше ділитися файлами та документами та отримувати доступ до програм і служб з будь-якого місця.

7. Підвищення продуктивності: хмарні обчислення можуть допомогти співробітникам бути більш продуктивними. Наприклад, вони можуть отримати доступ до хмарних програм і служб з будь-якого місця та використовувати їх на будь-якому пристрої.

8. Аварійне відновлення: Хмарні обчислення можуть допомогти підприємствам швидше відновлюватися після аварій. Це тому, що вони можуть використовувати хмарні служби резервного копіювання та аварійного відновлення.

9. Переваги для навколишнього середовища: хмарні обчислення можуть допомогти підприємствам зменшити свій вуглецевий слід. Це тому, що хмарні провайдери використовують енергоефективні центри обробки даних і відновлювані джерела енергії для забезпечення своїх операцій.

10. Покращення взаємодії з клієнтами: хмарні обчислення можуть допомогти підприємствам покращити взаємодію з клієнтами. Це пояснюється тим, що вони можуть використовувати хмарні додатки та служби, щоб надавати клієнтам кращий досвід.

1.1.5. Недоліки хмарних платформ

Якби хмарні обчислення були ідеальними, усі б використовували їх. І хоча хмарні обчислення є поширеною платформою, яка постійно розвивається, вона має певні недоліки, які заважають її загальному прийняттю.

1. Вартість. Подібно до того, як оренда житлового приміщення не обов'язково дешевша, ніж пряма покупка будинку, платформи хмарних обчислень не завжди є вигідною угодою. Вартість зводиться до унікальних потреб і ситуацій бізнесу. Цілком можливо, що дешевше мати невеликий власний центр обробки даних, який щомісяця запускає одні й ті самі програми, як годинник.

2. Міграція також може коштувати дорого. У деяких ситуаціях перехід із внутрішньої системи на хмару може спричинити надмірні витрати та створити значні труднощі.

3. Є ще проблеми з довірою. Деякі компанії не сприймають ідею, що їхня конфіденційна інформація зберігається на тих самих серверах, що й дані їхніх конкурентів, що потенційно підриває конкурентну перевагу.

1.2. Висновки до першого розділу

У першому розділі, виходячи з проведеного аналізу, можна стверджувати, що з очікуваним продовженням зростання хмарних обчислень підприємства все більше підприємств переходитимуть до хмари. Це створить підвищений попит на стійкі до відмови рішення, оскільки бізнес-критичні додатки залежатимуть від надійності хмарних інфраструктур.

Також набудуть подальшого розвитку технології автоматизації, включаючи DevOps-практики, контейнеризацію та управління інфраструктурою як код (IaaC), сприятиме більш ефективної реалізації стійких до відмови рішень на платформі Azure.

Хмарні обчислення вирішують безліч проблем стійкості до відмови, таких як мережеві збої, проблеми з обладнанням, кібератаки та інші. Географічний розподіл ресурсів у хмарі відіграє ключову роль у забезпеченні стійкості до відмови. Використання кількох регіонів дозволяє усунути збої в окремих локаціях та підвищити доступність сервісів.

Кібератаки становлять серйозну загрозу, і безпека має бути у центрі стратегії відмовостійкості. Ефективні заходи безпеки включають контроль доступу, моніторинг подій та шифрування даних.

Системи моніторингу та управління подіями є важливими інструментами для швидкого виявлення та реагування на збої. Регулярний моніторинг допомагає попереджати проблеми до їхніх серйозних наслідків.

Процеси резервного копіювання та відновлення даних є невід'ємною частиною стійкості до відмови. Регулярне тестування процедур відновлення наголошує на їх ефективності.

Оптимізація використання ресурсів, таких як балансування навантаження та автоматичне масштабування, є ключовим фактором для забезпечення ефективної стійкості до відмови при мінімізації витрат.

Таким чином успішна стратегія відмовостійкості вимагає комплексного та всебічного підходу.

РОЗДІЛ 2
ТЕХНОЛОГІЇ СТВОРЕННЯ ІНФРАСТРУКТУРИ

2.1. Опис використаних технологій та мов програмування

Стек технологій, використаний при розробці:

Серверна частина:

- збереження Node.js;
- Express.js;
- MongoDB -NoSQL база даних и Mongoose як ODM;

Фронтенд частина:

- бібліотека React;
- Styled Component

Таким чином бачимо , що це становить так званий MERN стек.

MERN стек — це комбінація чотирьох технологій: Mongo DB, Express JS, React JS і Node JS [7]. Це один із найактуальніших стеків у світі, який швидко зростає щодня, залучаючи багатьох розробників по всьому світу у величезну спільноту. Головна перевага цього полягає в тому, що використовується єдина мова програмування Javascript. Розглянемо що означає кожна літера цього акроніму:

- Mongo DB: База даних із відкритим вихідним кодом на основі документів, яка забезпечує масштабованість і гнучкість;
- Express JS: Структурована база, призначена для розробки веб-додатків і API;
- React JS: Бібліотека Javascript для створення інтерфейсів користувача. Підтримується Facebook;
- Node JS: Середа виконання JavaScript, побудована на движку Chrome V8 JS.

Архітектура MERN дозволяє легко побудувати трирівневу архітектуру (front end, back end, база даних), повністю використовуючи JavaScript і JSON.

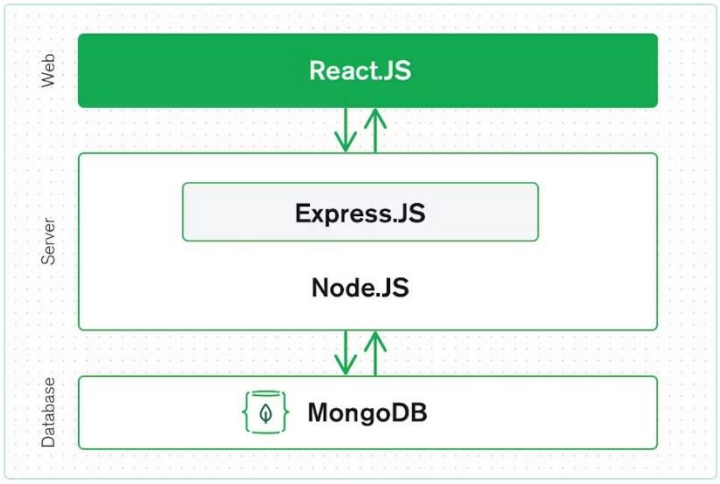

Рис.2.1. Трирівнева архітектура MERN

React.js front end.

Найвищим рівнем стеку MERN є React.js, декларативна структура JavaScript для створення динамічних клієнтських програм у HTML. React дозволяє створювати складні інтерфейси за допомогою простих компонентів, підключати їх до даних на сервері та відтворювати їх як HTML.

Сильна сторона React — це обробка інтерфейсів із збереженням стану, керованих даними, з мінімальною кількістю коду та мінімальними труднощами, а також має всі переваги, які ви очікуєте від сучасної веб-платформи: чудова підтримка форм, обробки помилок, подій, списків тощо.

Express.js і Node.js серверний рівень

Наступним рівнем є серверна структура Express.js, яка працює на сервері Node.js. Express.js називає себе «швидким, непереборним, мінімалістичним веб-фреймворком для Node.js», і це справді саме те, що він є. Express.js має потужні моделі для маршрутизації URL-адрес (зіставлення вхідної URL-адреси з функцією сервера) і обробки HTTP-запитів і відповідей.

Роблячи XML HTTP-запити (XHR) або GET або POST із вашого інтерфейсу React.js, ви можете підключитися до функцій Express.js, які забезпечують роботу вашої програми. Ці функції, у свою чергу, використовують драйвери MongoDB Node.js, або через зворотні виклики (callbacks), або за допомогою promises, для доступу та оновлення даних у вашій базі даних MongoDB.

Express.js (працює на Node.js) і React.js роблять програму JavaScript/JSON MERN повним стеком (full stack). Express.js — це серверний фреймворк додатків, який обгортає HTTP-запити та відповіді та дозволяє легко зіставляти URL-адреси з функціями на стороні сервера. React.js — це зовнішній фреймворк JavaScript для побудови інтерактивних інтерфейсів користувача в HTML і зв'язку з віддаленим сервером.

Ця комбінація означає, що дані JSON природним чином переміщаються з frontend в backend, що робить їх швидким для створення та досить простим для налагодження. Крім того, потрібно знати лише одну мову програмування та структуру документа JSON, щоб зрозуміти всю систему!

MongoDB рівень бази даних

Оскільки наша програма зберігає будь-які дані (user profiles, content, comments, uploads, events, etc.), то знадобиться база даних, з якою легко працюватумуть React, Express, and Node.

Тут на допомогу приходить MongoDB: документи JSON, створені у інтерфейсі React.js, можна надіслати на сервер Express.js, де їх можна обробити та (якщо вони дійсні) зберегти безпосередньо в MongoDB для подальшого отримання. MongoDB було розроблено для зберігання даних JSON нативно (технічно вона використовує двійкову версію JSON під назвою BSON), і все, від інтерфейсу командного рядка до мови запитів (MQL або MongoDB Query Language) побудовано на JSON і JavaScript.

MongoDB надзвичайно добре працює з Node.js і неймовірно спрощує зберігання, обробку та представлення даних JSON на кожному рівні вашої програми. Для хмарних додатків MongoDB Atlas робить це ще простіше,

надаючи вам автоматичне масштабування кластера MongoDB у хмарному провайдеру за вашим вибором, так само просто, як кілька натискань кнопки.

Як і в будь-якому веб-стеку, у MERN ви можете створювати все, що завгодно, хоча він ідеально підходить для випадків, які пов'язані з JSON, є рідними для хмари та мають динамічні веб-інтерфейси.

2.2. Методологія додатку

Для того щоб додаток був як можливо безболісно розгорнутий в будь-якому хмарному середовищі, тобто він був cloud-native (хмарно-орієнтований), його архітектура і структура повинні дотримуватися певних принципів. Для цього побудуємо додаток, який дотримується наступної методології:

1. Одна кодова база, один додаток.

Додаток повинен мати єдину кодову базу, яка відстежується в системі контролю версій (VCS), такій як Git. Це гарантує наявність єдиного джерела правди для коду програми. Це також дає вам можливість здійснювати стільки розгортань або конвеєрів доставки, скільки потрібно, не турбуючись про несподівані відмінності між різними джерелами. Очевидно, що це найпростіший фактор, який не потребує додаткових пояснень.

2. Управління залежностями.

Залежності програми мають бути явно оголошені та ізольовані від коду програми. У Node.js це досягається за допомогою менеджера пакетів, наприклад npm або yarn, який може керувати залежностями програми та встановлювати їх, а також виконувати інші корисні дії.

3. Проектуйте, збирайте, випускайте та запускайте.

Для керування етапами життєвого циклу додатку: створення, випуск і запуск, будемо використовувати Webpack для збірки, Babel, tslib для компіляції коду програми, а також Docker для упаковки та розгортання програми. Дотримання цього принципу також дозволяє забезпечити зручний, автоматизований і передбачуваний спосіб отримання робочого та

стандартизованого артефакту програми та її розгортання без або з мінімальним ручним втручанням у процес. Це один із найважливіших факторів, який можна правильно застосувати, щоб уникнути проблем, які завжди виникають під час розгортання програми вручну.

4. Конфігурація, облікові дані та код.

Параметри конфігурації програми Node.js слід зберігати у змінних середовища, а не жорстко закодувати в коді програми. Наприклад, замість того, щоб зберігати (hard-codding) URL-адресу бази даних у коді програми, її слід зберігати як змінну середовища, яку можна легко змінити, не змінюючи код. У Node.js є багато інструментів, таких як dotenv або config, для керування конфігурацією за допомогою змінних середовища.

5. Логування.

Програма буде реєструвати всі свої події та помилки, щоб їх можна було аналізувати та контролювати. Для реалізації цього фактору буде використовуватися журналювання Bunyan. Щоб зібрати та проаналізувати дані журналів, їх слід надіслати до централізованої системи керування журналами, наприклад ELK або Splunk.

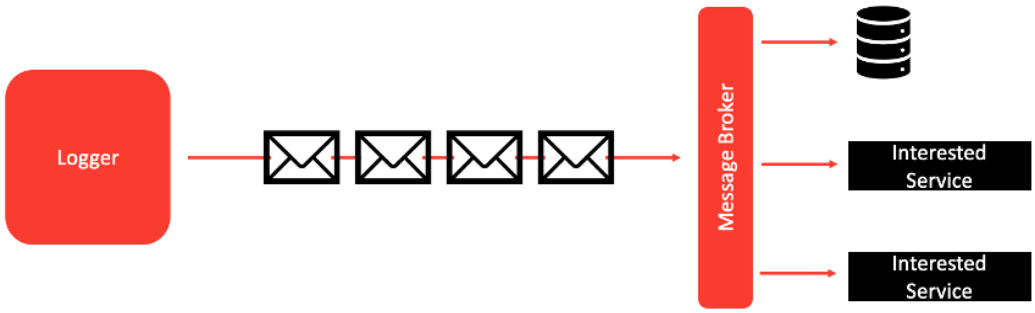

Рис. 2.2. Логування

6. Одноразовість.

Додаток буде одноразовим, щоб його можна було швидко та легко запускати та зупиняти. Для цього налаштуємо можливості автоматичного масштабування в хмарі для автоматичного запуску та завершення екземплярів

програми залежно від завантаження програми. Програма буде готова до швидкого запуску та плавного завершення без переривання існуючих з'єднань і пошкодження даних програми. Тому у нашій програмі буде реалізовано плавне (graceful shutdown) завершення роботи.

7. Супровідні послуги

Додаток розглядає всі зовнішні служби (такі як бази даних, кеші та брокери повідомлень) як прикріплені ресурси, до яких можна отримати доступ через URL-адресу або інший строку підключення. Це гарантує, що програма відокремлена від конкретних деталей реалізації послуг, які вона використовує. Також він зберігає деталі підключення в середовищі та абстрагує їх у своєму коді за допомогою бібліотеки mongoose для MongoDB. Це також стосується зовнішніх API. Тому наш додаток також керує даними до зовнішніх інтерфейсів API та з них, щоб забезпечити слабкий зв'язок між службами.

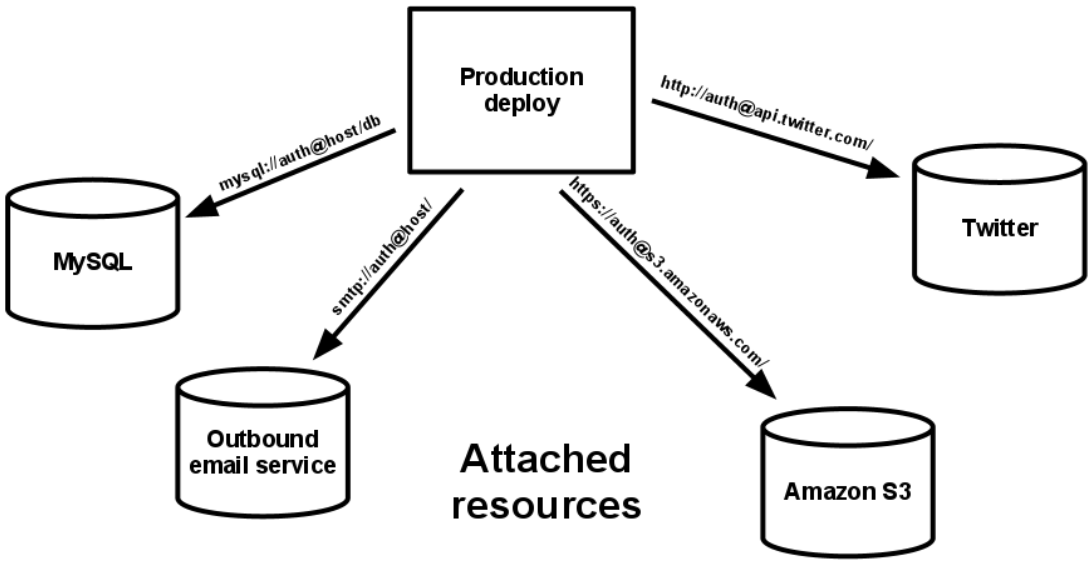

Рис. 2.3. Взаємодія з зовнішніми службами

8. Паритет середовища.

Додаток буде мати паритет між середовищами розробки (development) та виробництва (production), щоб ту саму кодову базу можна було запускати в обох середовищах без змін. Цього можна досягти за допомогою змінних середовища для налаштування програми та використання конвеєра безперервної інтеграції та

розгортання (CI/CD) для автоматизації процесу розгортання. Це також означає, що development, staging та production однакові з точки зору конфігурації обладнання. У реальному світі мати кілька середовищ (особливо development), що працюють на апаратному забезпеченні, подібному до production, може бути досить дорогим. У будь-якому випадку, вигідно мати хоча б одне середовище, схоже на виробниче. Для процесу безперервної інтеграції та розгортання (CI/CD) будемо користуватися засобами Azure.

9. Процеси адміністрування.

У додатку надані адміністративні інтерфейси для таких завдань, як міграція баз даних і моніторинг системи.

10. Прив'язка порту.

Додаток прив’яжемо до порту, визначеного змінною середовища, а не жорстко закодований певний номер порту у програмі. Це дозволяє програмі запускатися на будь-якому номері порту без зміни коду. Певною мірою цей фактор виглядає дуже схожим на фактор конфігурації, але він зосереджується лише на номері порту програми, оскільки важливо запустити програму на певному або доступному порту, щоб розпочати обробку запиту. Якщо з якоїсь причини використовується порт, вам потрібно мати можливість змінити цю конфігурацію без зміни коду та додаткового циклу розгортання.

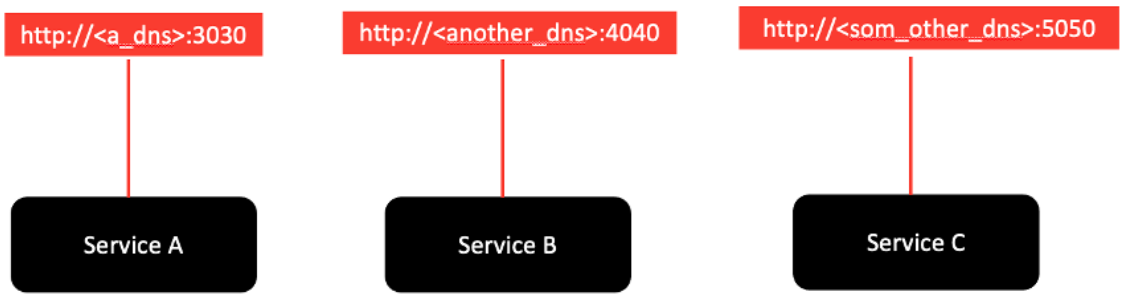

Рис. 2.4. Прив'язка порту

11. Процеси без стану.

Розробимо масштабовану програму Node.js без стану, щоб її можна було запускати на кількох екземплярах для обробки вхідних запитів (іншими словами, програма буде горизонтально масштабованою). Будемо використовувати балансир навантаження (load balancer) для розподілу трафіку між кількома екземплярами в межах одного обчислювального екземпляра. Кожен екземпляр буде самостійно обробляти вхідний запит і не залежати від свого внутрішнього/кешованого стану.

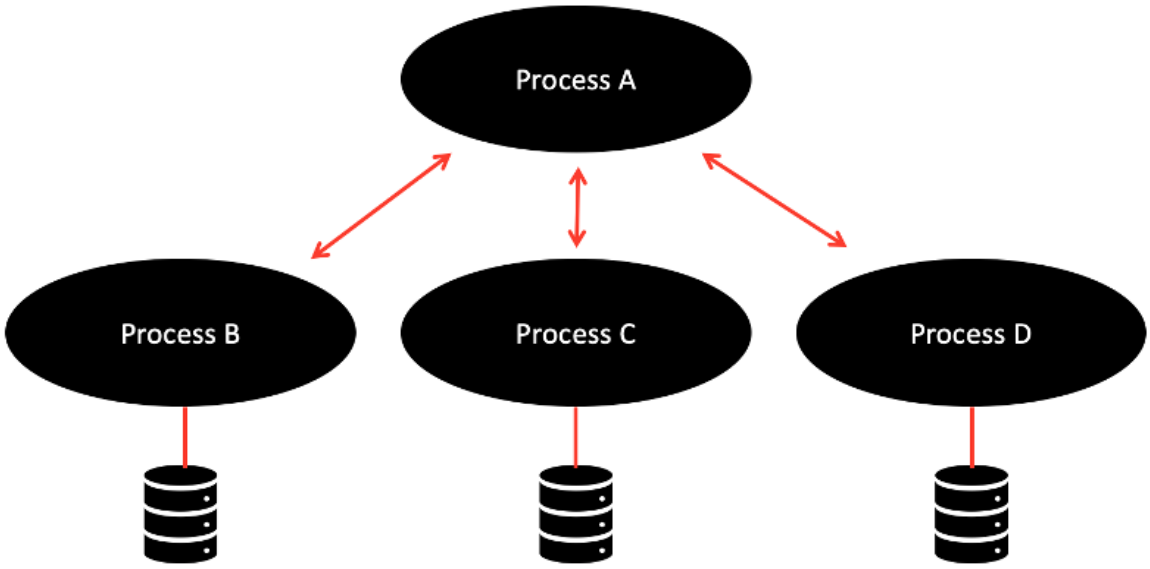

Рис. 2.5. Процеси без стану

12. Паралелізм.

Архітектура додатку буде одночасно обробляти запити і уникати операцій блокування. В додатку використовуються асинхронні функції та зворотні виклики. У хмарних середовищах є багато способів досягнення паралелізму: запуск кількох екземплярів додатків і розподіл трафіку за допомогою балансувальника навантаження, використання інструментів «функція як послуга» (FaaS), використання роз'єднаних або керованих подіями (event-driven) архітектур і пов'язаних інструментів тощо.

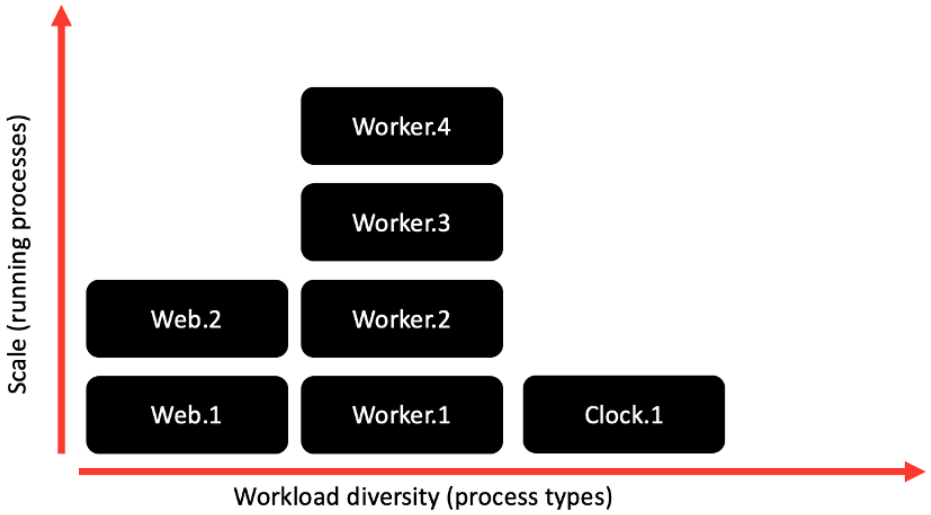

Рис. 2.6. Паралелізм

Впроваджуючи ці принципи ми тим самим розробляємо, а потім створюємо програми, які легше розгортати, контролювати та підтримувати. Крім того, це дає нам високий рівень гнучкості, щоб адаптувати свої рішення до вимог, що змінюються з часом.

2.3. Проектування і розробка API

Програми, розроблені для хмари, зазвичай є учасниками екосистеми розподілених сервісів, і якщо API не визначено чітко, це може призвести до кошмару збоїв інтеграції. Ключові кроки під час реалізації принципу «перш за все API» в Node.js:

- розробка API: починаємо із розробки кінцевих точок API, структур запитів/відповідей і загальної функціональності. Визначаємо ресурси та операції, які надаватиме API. Для розробки та документування API будемо використовувати такі інструменти, як OpenAPI Specification (OAS) або API Blueprint;
- створення контракту API: коли дизайн API буде готовий, створимо контракт, який визначає очікувану поведінку API. Цей контракт буде у

формі документа OpenAPI, який визначає кінцеві точки, вхідні параметри, формати відповідей та інші деталі;

- створення серверного коду. Використовуємо такі інструменти, як Swagger Codegen або OpenAPI Generator, щоб створити структуру коду на стороні сервера на основі контракту API. Ці інструменти можуть генерувати код Node.js, який налаштовує базову структуру API, включаючи маршрутизацію, обробку запитів і керування помилками;
- впровадження кінцевих точок API: використовуючи згенерований код як відправну точку, реалізуємо кінцеві точки API, додавши необхідну бізнес-логіку та рівні доступу до даних. Це передбачає написання коду Node.js для обробки вхідних запитів, обробки даних і створення відповідних відповідей;
- перевіряємо та тестуємо API: переконаємося, що реалізація API відповідає визначеному контракту API. Перевіряємо запити та відповіді на відповідність очікуваним структурам і виконуємо ретельне тестування, щоб переконатися, що API працює належним чином. Для тестування кінцевих точок API використовуємо такі інструменти, як Jest, Mocha або Supertest;
- документування API: документування API має вирішальне значення для його ефективного використання. Створюємо документацію API автоматично з контракту API або використовуємо такі інструменти, як Swagger UI (є багато спеціальних пакетів для фреймворку, наприклад, nest-swagger) або Redoc для створення інтерактивної документації API. Ця документація допомагає розробникам зрозуміти, як взаємодіяти з API та його доступними кінцевими точками;
- увімкнення розробки клієнта: після впровадження та документування API клієнти можуть почати використовувати його для створення програм. Клієнти можуть бути розроблені на будь-якій мові програмування або фреймворку, який підтримує виконання запитів HTTP. Зосереджуючись

на принципі «перш за все API», ми надаємо чіткий і послідовний інтерфейс для взаємодії розробників клієнтів.

Виконавши ці кроки, ми зможете застосувати принцип «перш за все API» в Node.js і забезпечити добре розроблений, задокументований і масштабований API для проекту.

2.4. Реалізація телеметрії

Телеметрія є ще одним важливим доповненням інформаційної системи. Телеметрія потрібна як окремий фактор на додаток до логування, який уже був включений в загальну методологію програми. Хоча ведення журналів / логування є важливим елементом створення хмарних додатків, зазвичай це інструмент, який використовується під час розробки для діагностики помилок і потоків коду. Логування зазвичай орієнтується на внутрішню структуру додатка, а не на відображення реального використання клієнтами. Телеметрія, з іншого боку, зосереджена на зборі даних після випуску програми. Телеметрія та моніторинг додатків у режимі реального часу дозволяють розробникам відстежувати продуктивність, працездатність і ключові показники своїх додатків у цьому складному та дуже розподіленому середовищі.

Принципи реалізації телеметрії:

1. Інструментарій. Інструментування – це процес додавання коду до програми для збору відповідних даних. Це може включати показники / метрики, журнали / логи та трасування. У Node.js можливо використовувати різні бібліотеки та інструменти для інструментування, такі як Prometheus, StatsD або вбудований модуль консолі.

2. Визначення метрик: визначаємо метрики, важливі для моніторингу та розуміння програми. Ці показники можуть включати використання процесора, споживання пам’яті, час запиту/відповіді, рівень помилок та будь-які інші відповідні дані. Визначаємо, яка інформація допоможе оцінити продуктивність, стабільність і використання нашої програми.

3. Інтегрування інструментів моніторингу: виберемо та інтегруємо інструменти моніторингу, сумісні з Node.js. Серед популярних варіантів – Prometheus, Grafana, New Relic, Datadog або Elastic Stack (Elasticsearch, Logstash і Kibana). Ці інструменти надають функції для збору даних, візуалізації, сповіщень та аналізу.

4. Збирання та зберігання даних: налаштуємо свої бібліотеки інструментів та інструменти для збору визначених показників і зберігання даних у відповідному сервері, наприклад базі даних часових рядів або сховищі логів. Це дозволяє зберігати історичні дані та виконувати аналіз протягом тривалого часу.

5. Візуалізування та аналізування даних: використовуємо можливості інструментів моніторингу для візуалізації та аналізу зібраних даних. Створюємо інформаційні панелі та візуалізації, які відображають показники значущим чином. Це допоможе отримати уявлення про продуктивність нашої програми, визначити тенденції та виявити аномалії.

6. Налаштування сповіщень: визначаємо порогові значення та правила для активації попереджень, коли певні показники перевищують попередньо визначені порогові значення або коли відбуваються певні події. Це дає змогу завчасно реагувати на критичні проблеми або зниження продуктивності у нашому додатку.

7. Постійне вдосконалення: регулярно переглядаємо та аналізуємо телеметричні дані, щоб визначити області, які потребують покращення. Використовуємо інформацію, отриману за допомогою телеметрії, щоб оптимізувати продуктивність, виявити вузькі місця, виправити помилки та покращити взаємодію з користувачем. Постійно удосконалюємо додаток на основі інформації, зібраної за допомогою телеметрії.

Шляхом реалізації принципу телеметрії, отримуємо цінну інформацію про поведінку та продуктивність нашої програми. Це дозволяє приймати рішення на основі даних, покращувати загальну якість нашої програми та забезпечувати кращий досвід для ваших користувачів.

2.5. Роль автентифікації

Автентифікація — це процес перевірки доступу користувача або пристрою до системи або ресурсів. Іншими словами, коли користувач намагається отримати доступ до інформації в мережі, він повинен надати секретні облікові дані, щоб підтвердити свою особу.

Найпоширенішим методом аутентифікації є аутентифікація за допомогою логіна та пароля. Але зі збільшенням загроз кібербезпеці в останні роки більшість організацій використовують і рекомендують додаткові фактори автентифікації для багаторівневої безпеки.

Зараз аутентифікація є звичайною практикою. Більшість людей використовували автентифікацію для доступу до своєї особистої інформації та пристроїв вдома та на роботі:

- вхід в соціальну мережу за допомогою логіна та пароля;
- відкриття телефону за допомогою TouchID, FaceID або унікального PIN-коду.

Звичайно, нові методи автентифікації набирають обертів, щоб краще захистити особисті, ділові та державні ресурси від несанкціонованого доступу.

Існує три основні категорії облікових даних, які використовуються для автентифікації або підтвердження особи:

- фактор знань (найпоширеніший фактор): перевіряється особа шляхом підтвердження користувачів за допомогою конфіденційної інформації, якою вони користуються, наприклад логіна та пароля;
- фактор володіння: перевіряється особа за допомогою унікального об'єкта, такого як секретний ключ;
- фактор приналежності: перевіряється особа за допомогою властивих біометричних характеристик користувача, таких як відбиток пальця, TouchID, FaceID тощо.

Інші типи інформації, такі як дані про місцезнаходження або ідентифікаційні дані пристрою, також можуть використовуватися як фактори автентифікації.

Будемо використовувати Bearer автентифікацію.

Bearer автентифікація (також відома як маркерна / токен автентифікація) — це схема автентифікації, вбудована в протокол HTTP і зазвичай реалізована на стороні веб-сервера. Під час використання Bearer автентифікації клієнт включає заголовок авторизації кожного запиту, який він робить. Цей заголовок має починатися зі слова Bearer, за яким слідує пробіл і рядок у кодуванні base64 ім’я користувача: пароль.

Як приклад можна розглянути користувача з логіном testuser і паролем testpassword. Щоб зробити HTTP-запит із базовою автентифікацією, нам потрібно закодувати рядок testuser:testpassword у base64 і додати його до заголовка авторизації таким чином:

```
Authorization: Bearer dGVzdHVzZXI6dGVzdHBhc3N3b3Jk
```

Коли сервер отримує запит, має бути функція, відповідальна за доступ для перевірки. Ця функція витягне маркер авторизації, декодує його, витягне користувача та пароль і перевірить, чи наданий користувач має точно такий самий пароль, збережений у базі даних. Якщо перевірка не вдається, сервер надішле відповідь із кодом HTTP 401.

2.6. Роль авторизації

Будемо використовувати Role-based access control (RBAC) Authorization стратегію. У цій стратегії кожному користувачеві призначається одна або кілька попередньо визначених ролей, і кожна роль має певний набір дозволів. Щоб зробити наш магазин безпечним, нам потрібно захистити доступ до конфіденційних даних, таких як продукти, особисті дані користувачів тощо. Для цього ми створимо кілька ролей, таких як клієнт, адміністратор, менеджер.

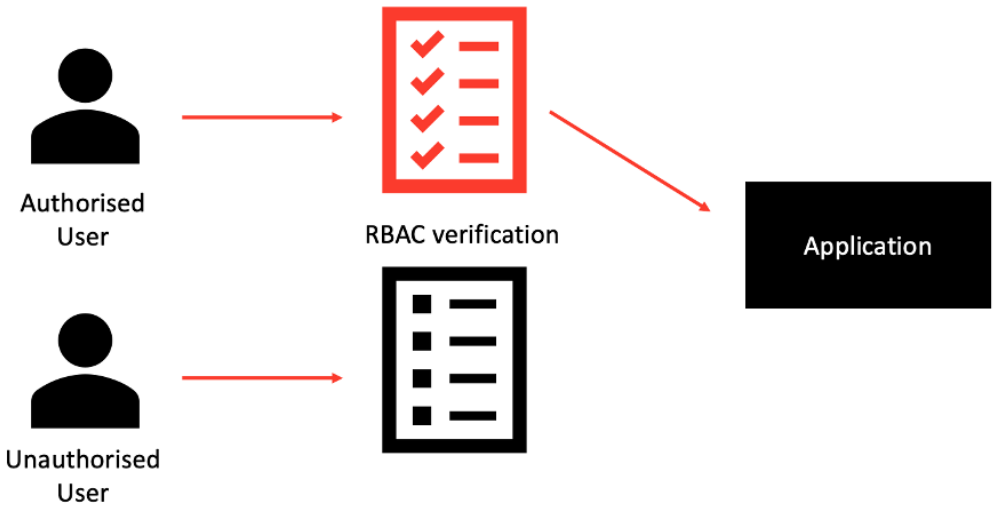

Рис. 2.7. RBAC Authorization

Тепер ми можемо створювати проміжне програмне забезпечення для захисту наших даних і, наприклад, надавати доступ до модифікації продуктів лише менеджерам і адміністраторам.

```
    app.post(`/products`, (req, res) => {
  const user = req.currentUser;

  if (user.role !== Roles.ADMIN || user.role !== Roles.MANAGER) {
      res.status(401).send("User don't have permissions to create
products");
  }
  // Logic
  ...
});
```

Крім цієї стратегії, існують ще:

– контроль доступу на основі атрибутів (ABAC);

Щоб впоратися з нерозв'язними проблемами в RBAC, було розроблено інше рішення на основі атрибутів. Ця стратегія використовує атрибути ресурсу та користувача, як-от місто користувача, країна, день тижня чи власник ресурсу, замість ролі користувача, щоб дозволити або відхилити доступ до ресурсу.

– список контролю доступу (ACL);

У цій стратегії використовується список правил, які дозволяють або забороняють доступ до ресурсу.

Додавання фактору автентифікації та авторизації додає важливий акцент на безпеку для хмарних програм. Розгортання програм у хмарному середовищі означає, що програми можна транспортувати через багато центрів обробки даних по всьому світу, виконувати в кількох контейнерах і мати доступ до них майже необмежена кількість клієнтів. Отже, життєво важливо, щоб безпека не була запізнілою думкою для хмарних програм, а була дуже важливим фактором, який слід враховувати.

2.7. Сервіси Azure

На даний момент ми спостерігаємо трансформацію IT-галузі, спричинену впровадженням хмарних обчислень (Cloud computing). IT-департаменти компаній змінюють свою роль і коло відповідальності завдяки використанню інфраструктури як сервісу (Infrastructure as a Service, IaaS) та програмних сервісів (Software as a Service, SaaS). Концепція платформи додатків як сервісу (Platform as a Service, PaaS) змінює підходи до розробки програмного забезпечення. Змінюється робоче оточення (runtime environment) додатків, ключові функції переносяться з локальних компонентів на програмні послуги, змінюються моделі розгортання, процеси тестування та діагностики.

Microsoft Azure є платформою для розробки додатків і, відповідно до моделі PaaS, надає інструменти розробки, операційне середовище (runtime environment) та набір хмарних сервісів, які надають готові рішення для типових завдань при побудові додатків. Microsoft Azure також надає прикладні інтерфейси та інструменти розміщення ресурсів та управління. Як операційне оточення виступають сервіси Azure Virtual Machines, Azure Cloud Services та Azure App Service, які відрізняються рівнем контролю, легкістю розгортання та гнучкістю масштабування:

– сервіс Azure VM надає абстракцію апаратного забезпечення та ліцензію на операційну систему (іноді – з встановленим програмним забезпеченням). Розробник та адміністратор отримують повний доступ до всіх функцій операційної системи, однак повинні самі вирішувати питання розгортання програми, безпеки, підтримки, стійкості до відмов та масштабування;
– сервіс Azure Cloud Services надає операційну систему для запуску програми, включаючи оновлення, захист від вірусів, підтримку горизонтального масштабування (scale out) та стійкості до відмов. Сервіс обмежує можливості розробника та адміністратора, так як будь-які зміни коду програми та більшість змін конфігурації вимагають повторного розгортання програми;
– сервіс Azure App Service надає абстракцію операційного середовища, т.зв. пісочницю (sandbox) для запуску веб-застосунків. Сервіс дозволяє легко розміщувати програми, а також масштабувати як вертикально (scale up), так і горизонтально (scale out). Розробник або адміністратор отримує повний доступ до файлів програми та конфігурації веб-сервера.

Хмарні сервіси, що входять до складу Microsoft Azure, мають вирішувати типові завдання, які постають перед розробниками додатків. У таблиці нижче наведено завдання, приклади їх вирішення у локальному середовищі (on premise), а також сервіси Microsoft Azure, які надають необхідну функціональність.

Таблиця 2.1

Функціональні сервіси Microsoft Azure сервісів

Функція	Приклади локальної реалізації	Сервіси від Microsoft Azure
1	2	3
Операційне оточення	Windows Server, HyperV	Azure App Service, Azure Cloud Services, Azure VM
Реляційна база даних	Microsoft SQL Server, Oracle Database	Azure SQL

Продовження Таблиці 2.1

1	2	3
NoSQL база даних	MongoDB, CouchDB	Azure Table Storage, Azure DocumentDB
Файлове сховище	Windows File Server	Azure Blob Storage, Azure File Storage
Виконання фонових завдань	Windows Services	Web Jobs
Індексування та пошук даних	Solr, ElasticSearch	Azure Search Service
Зберігання секретів/паролів	CryptoAPI, Microsoft Active Directory	Azure Key Vault
Черги повідомлень, обробка подій, пакетна обробка	MSMQ, RabbitMQ	Azure Storage Queues, Azure Service Bus, Azure Event Hubs
Ідентифікація користувачів	Microsoft Active Directory	Azure Active Directory
Діагностика та телеметрія	Windows Event Log, Windows Performance Counters	Application Insights, Windows Azure Diagnostics

Крім функціональних сервісів Microsoft Azure надає інфраструктурні сервіси, що дозволяють реалізувати глобальні високопродуктивні програми:

Таблиця 2.2

Інфраструктурні сервіси Microsoft Azure сервісів

Функція	Сервіси від Microsoft Azure
Глобальний розподіл трафіку	Azure Traffic Manager
Мережа доставки контенту (CDN)	Azure CDN
Розподіл навантаження та маршрутизація веб-трафіку	Azure Application Gateway

Для керування службами та виділенням ресурсів Microsoft Azure надає як графічний інтерфейс portal.azure.com, так і набори інтерфейсів Azure Service Management, Azure Resource Manager, модуль Azure PowerShell та інструмент Azure CLI.

2.8. Висновки до другого розділу

У другому розділі в ході проведеного аналізу, були розглянуті такі аспекти, як: архітектура, принцип роботи та внутрішня структура MERN стеку, з'ясували, що він ідеально підходить для випадків, які пов'язані з JSON, є рідними для хмари та мають динамічні веб-інтерфейси.

Саме на основі цього стеку будемо будувати наш додаток типового інтернет магазину, який згодом розмістимо у хмарі для подальшого моніторингу і аналізу стійкості отриманої інфраструктури до відмов.

Визначили методологію, дотримуючись якої наш додаток перетворюється в cloud-native, тобто хмарно-сумісний і його стає легше розгортати, контролювати та підтримувати. Впровадження цих принципів дає високий рівень гнучкості, щоб адаптувати свої рішення до вимог, що змінюються з часом.

Крім того, були розглянуті хмарні сервіси Microsoft Azure, їх функціональне навантаження в залежності від завдання, інтерфейси для керування службами та виділенням ресурсів.

РОЗДІЛ 3
РОЗРОБКА ТА ДОСЛІДЖЕННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Для практичного прикладу буде використовуватись розроблений на Node.js типовий інтернет магазин, для якого потрібно створити середовище, яке буде максимально безвідмовне і незалежне від зовнішніх чинників. Лістинг програми міститься в Додатку А.

У хмарі Azure є розширені рішення, які дозволяють створити конвеєр безперервної інтеграції та безперервної доставки (Continuous Integration and Continuous Delivery CI/CD) у хмарі. Про важливість CI/CD в надійності, відмовостійкості інформаційної системи говорилося в попередньому розділі

Azure пропонує два основні керовані інструменти CI/CD — Azure DevOps і Azure Pipelines. Azure DevOps дозволяє керувати процесами CI/CD, визначаючи такі сутності, як артефакти, дошки та плани тестування. Azure Pipelines забезпечує контроль версій, сервер збірки та системи розгортання, які можуть використовувати CI/CD на практиці. У нашій роботі будемо використовувати Azure DevOps.

3.1. Налаштування допоміжних інструментів

Встановимо та налаштуємо наведені нижче інструменти, щоб створити надійний і автоматизований робочий процес розробки, покращуючи якість і ефективність процесу розробки. Крім того, такі інструменти створюють міцну основу для встановлення конвеєрів CI для будь-якої програми та її середовища.

- Webpack - збірник модулів, який компілює кілька модулів в один файл. Це допомагає зменшити розмір програми, прискорити її завантаження та підвищити продуктивність;
- Jest - структура тестування, яка спрощує написання та виконання тестів. Це автоматизує процес тестування, полегшуючи виявлення та усунення проблем. існує багато альтернатив і додаткових пакетів для Jest;

- ESLint - інструмент аналізу коду, який перевіряє типові помилки кодування та забезпечує дотримання стандартів кодування. Це допомагає забезпечити узгодженість і придатність кодової бази;
- Husky - менеджер підключень Git, який дозволяє запускати власні сценарії в певних точках робочого процесу Git. Його можна використовувати для запуску хуків перед фіксацією, забезпечуючи якість і послідовність коду. Зазвичай ви встановлюєте такі хуки: commit-msg для запуску commitlint і перевірки повідомлення коміту; попередній запуск linter; попереднє натискання для запуску модульних тестів; інші хуки, засновані на умовностях і потребах команди. Як ви можете помітити, за допомогою git-хуків ви запускаєте ті самі інструменти якості коду, що й конвеєр CI. Важливо виконувати всі перевірки якості коду та виправляти проблеми на ранніх етапах. Це також економить гроші на обчислювальних ресурсах і відповідно час. Зрештою, можна сказати, що CI починається у локальному середовищі;
- Prettier - засіб форматування коду, який автоматично форматує код відповідно до попередньо визначених правил. Це допомагає забезпечити послідовний стиль кодування та читабельність;
- SonarQube та інші постачальники статичних SaaS/PaaS. Він може працювати незалежно, але найбільша користь, якщо поєднати його з іншими інструментами, такими як ESLint, Jest. Він може використовувати конфігурацію eslint, а також звіт про тестове покриття, створений Jest. Потім він створює звіти про якість коду та дає гарну візуалізацію. Цікава перевага SonarQube полягає в тому, що можна запустити свій власний сервер/екземпляр (локально або в хмарі) з образу докера.

3.2. Налаштування Azure сервісів

Побудова відмовостійкої інфраструктури в Azure включає в себе використання різних сервісів і підходів, що надаються платформою. Для налаштування сервісів Azure будемо використовувати портал Azure як простий у використанні веб-інтерфейс користувача, який допомагає створювати, керувати, розгортати та ефективно використовувати ресурси Azure. Портал Azure — це уніфікована веб-консоль, яка є альтернативою інструментам командного рядка. За допомогою порталу Azure ви можете керувати своєю підпискою на Azure за допомогою графічного інтерфейсу користувача. Ви можете створювати, керувати та контролювати все, від простих веб-програм до складних хмарних розгортань на порталі.

3.2.1. Створення екземпляру та вибір характеристик

Для хостингу додатків будемо використовувати Azure App Service - потужну та гнучку платформу. Сервіс орієнтований на легкість розгортання (deployment), конфігурації, діагностики та масштабування додатків. Azure App Service — це пропозиція платформи як послуги (PaaS), яка дозволяє розробникам створювати, розміщувати та масштабувати веб-додатки та API без керування основною інфраструктурою. Він підтримує різні мови програмування, фреймворки та інструменти, що робить його чудовим вибором для розробників із різними технологічними перевагами. За допомогою App Service Plans ми можемо легко керувати нашим додатком і масштабувати його відповідно до потреб користувачів. App Service Plans, з іншого боку, використовуються для визначення ресурсів і можливостей, доступних для розміщених в них веб-програм. Вони визначають такі фактори, як потужність сервера, параметри автоматичного масштабування та географічне розташування.

Таким чином, перш за все налаштуємо App Service Plan. У взаємодії з Azure будемо використовувати Azure Portal як візуальний та найбільш інтуїтивно зрозумілий інтерфейс.

Спочатку створимо Resource Group. Resource Group допомагають організовувати та керувати пов’язаними ресурсами Azure. Створення спеціальної групи ресурсів для нашої програми та App Service Plan є хорошою практикою.

Увійдемо на портал Azure і виконаємо ці кроки, щоб створити свої ресурси Azure App Service:

Крок 1. На порталі Azure:

- ввести « web app database » у рядок пошуку у верхній частині порталу;
- вибрати елемент із позначкою Web App + Database під заголовком Marketplace;

Крок 2. На сторінці «Create Web App + Database» заповнити наведену нижче форму.

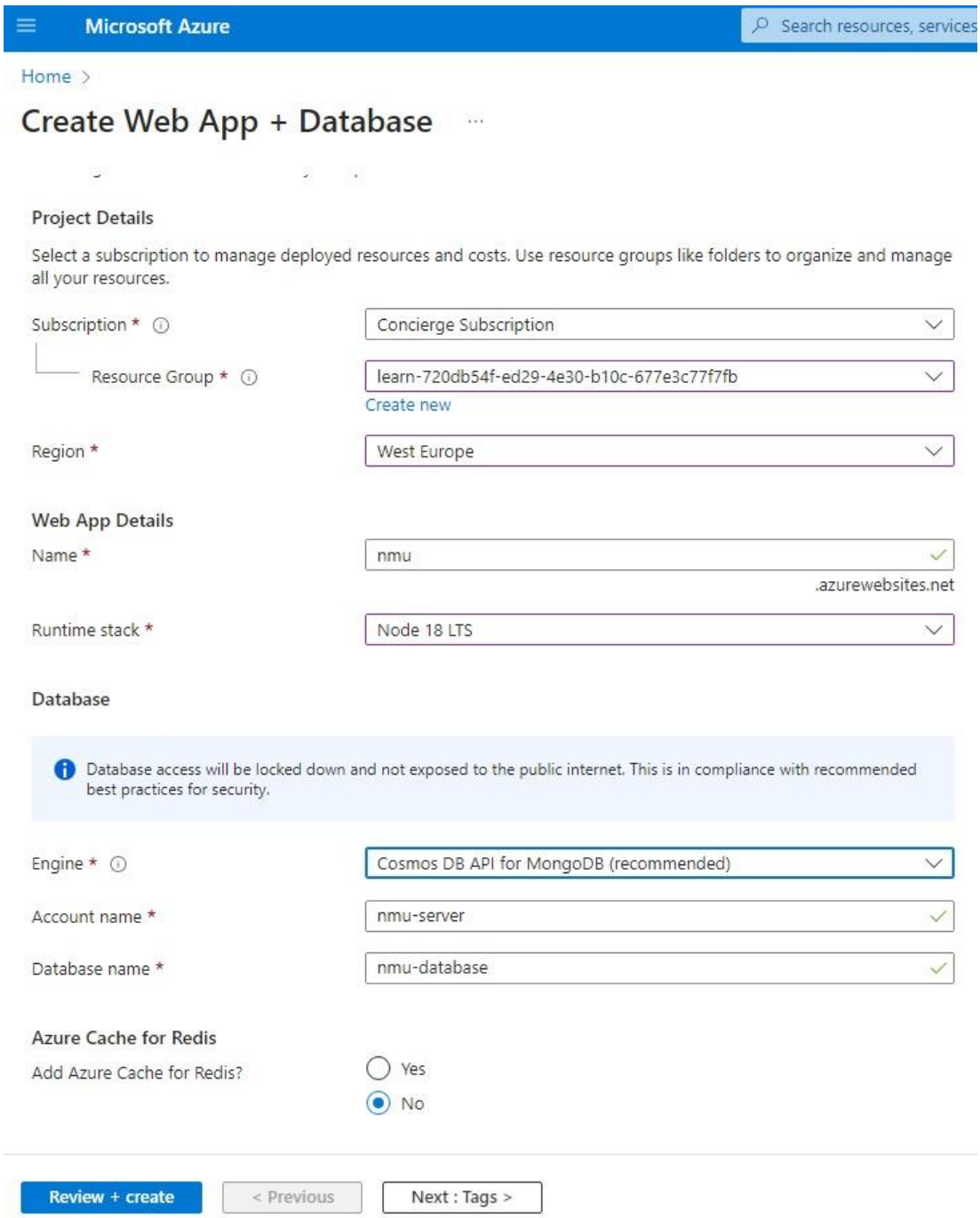

Рис. 3.1. Створення екземпляру

Пояснення:

Name - Це ім’я, яке використовується як частина імені DNS для нашої програми у формі https://<назва-програми>.azurewebsites.net.

Region - Регіон для фізичного запуску програми у світі. Вибираємо Регіон, як можна ближче до цільової аудиторії. Припустимо наш інтернет магазин буде націлений на покупців зі Східної Європи.

Runtime stack - Стек середовища виконання для програми. Тут ми вибираємо версію Node для свого додатка.

Hosting plan - План хостингу для програми. Це рівень ціноутворення, який включає набір функцій і можливості масштабування для нашої програми.

Resource Group - Група ресурсів для програми. Група ресурсів дає змогу групувати (у логічному контейнері) усі ресурси Azure, необхідні для програми.

Крок 3. Розгортання займає кілька хвилин. Після завершення розгортання натиснемо кнопку Go to resource. Перейдемо безпосередньо до App Service, і переконаємось, що створено такі ресурси:

- Група ресурсів (Resource group) → Контейнер для всіх створених ресурсів;
- App Service plan → Визначає обчислювальні ресурси для служби програми. Створено план Linux на стандартному рівні;
- App Service → Представляє вашу програму та працює в плані App Service;
- Віртуальна мережа (Virtual network) → Інтегрована з програмою App Service та ізолює внутрішній мережевий трафік;
- Приватна кінцева точка (Private endpoint) → Кінцева точка доступу до ресурсу бази даних у віртуальній мережі;
- Мережевий інтерфейс (Network interface) → Представляє приватну IP-адресу для приватної кінцевої точки;
- Azure Cosmos DB для MongoDB → доступ лише за приватною кінцевою точкою. На сервері створюється база даних і користувач;
- Приватна зона DNS (Private DNS zone) → Вмикає розпізнавання DNS сервера Azure Cosmos DB у віртуальній мережі.

Для підвищення відмовостійкості встановимо багатокористувацьку зонно-надлишкову службу додатків (multi-tenant zone-redundant App Service). Для цього ввімкнемо опцію резервування зони під час «Create Web App» або «Create App Service Plan

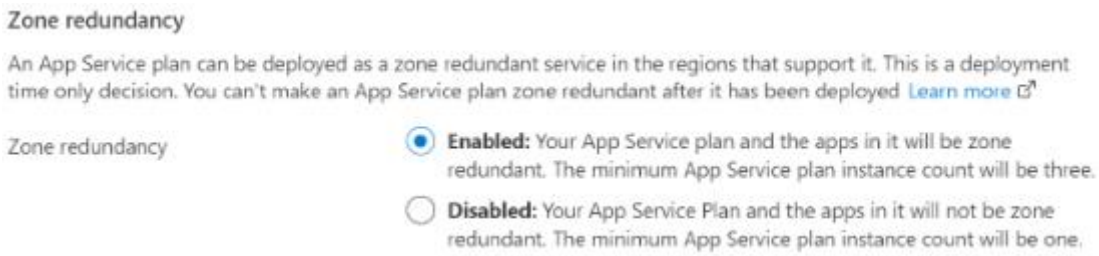

Рис. 3.2. Встановлення zone redundancy

Потужність / кількість екземплярів можна змінити після створення App Service Plan перейшовши до налаштувань Scale out (App Service plan).

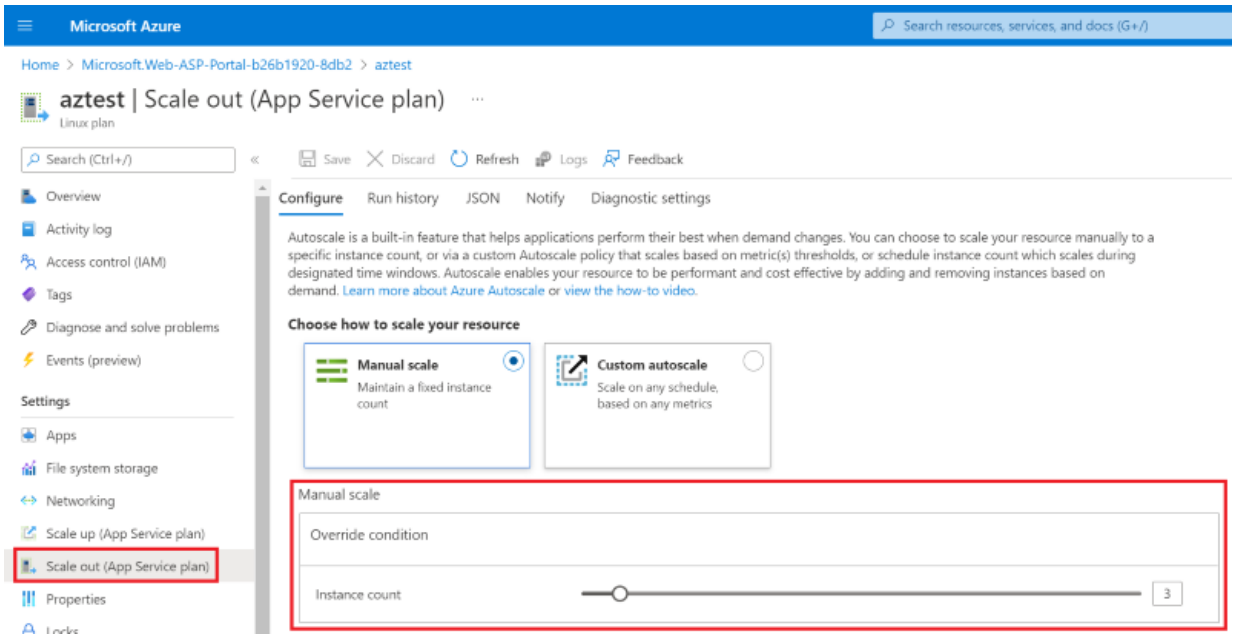

Рис. 3.3. Встановлення масштабування (scale out)

Трафік направляється до всіх доступних екземплярів App Service. У випадку, коли зона вимикається, платформа App Service виявляє втрачені екземпляри та автоматично намагається знайти нові екземпляри для заміни та за потреби розподіляє трафік. Коли при вже налаштованому автомасштабуванні стане зрозуміло, що потрібні додаткові екземпляри, автомасштабування також надішле запит до App Service, щоб додати більше екземплярів.

3.2.2 Налаштування підключення до бази даних

Майстер створення вже згенерував URI MongoDB, але для програми ще потрібні змінні DATABASE_URL і DATABASE_NAME. На цьому кроці створимо налаштування програми у потрібному форматі.

Крок 1. На сторінці App Service у меню ліворуч виберемо Configuration

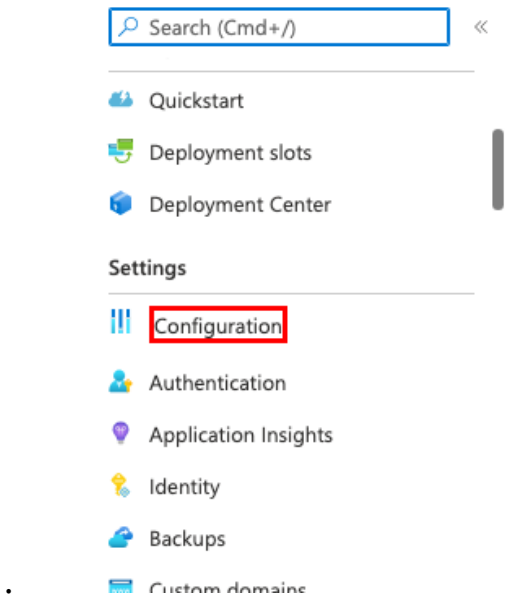

Рис. 3.4. Вибір конфігурації

Крок 2. На вкладці «Application settings» на сторінці «Configuration» створимо налаштування DATABASE_NAME:

- вибрати «New application setting»;
- у полі Name ввести DATABASE_NAME;
- у полі Value ввести автоматично згенероване ім’я бази даних за допомогою майстра створення, яке виглядає як msdocs-expressjs-mongodb-XYZ-database;
- підтвердити вибір, натиснувши кнопку OK.

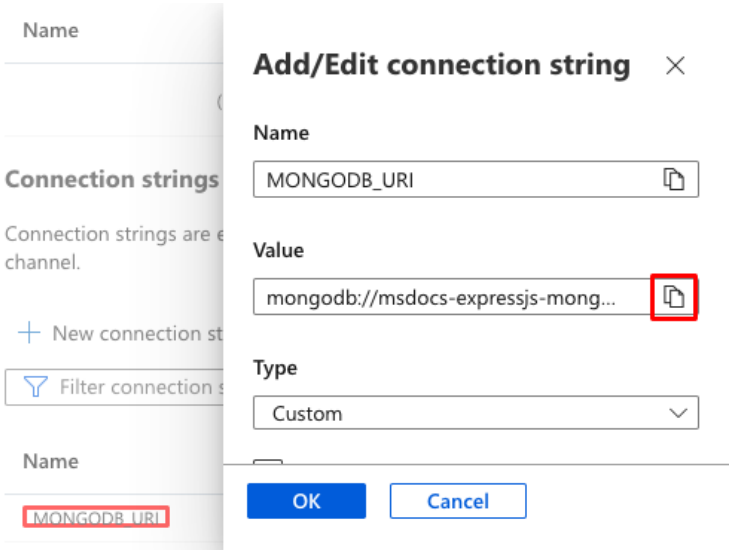

Рис. 3.5. Налаштування підключення бази даних

Крок 3.

- вибрати строку підключення MONGODB_URI, яка була створена майстром створення;
- скопіювати із поля Value значення в текстовий файл для наступного кроку. Це у форматі URI строки підключення MongoDB;
- вибрати «Скасувати».

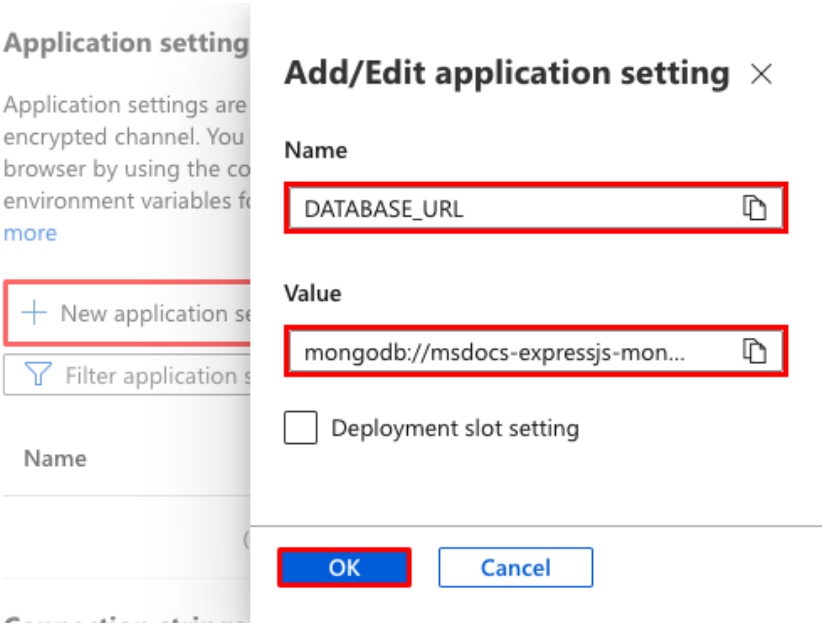

Рис. 3.6. Налаштування підключення бази даних

Крок 4.

– виконуючи ті самі кроки, що й у кроці 2, створимо параметр програми з іменем DATABASE_URL і встановимо значення, яке скопіювали з строки підключення MONGODB_URI (тобто mongodb://...);
– на панелі меню вгорі вибрати «Save»;
– коли буде запропоновано, вибрати Continue.

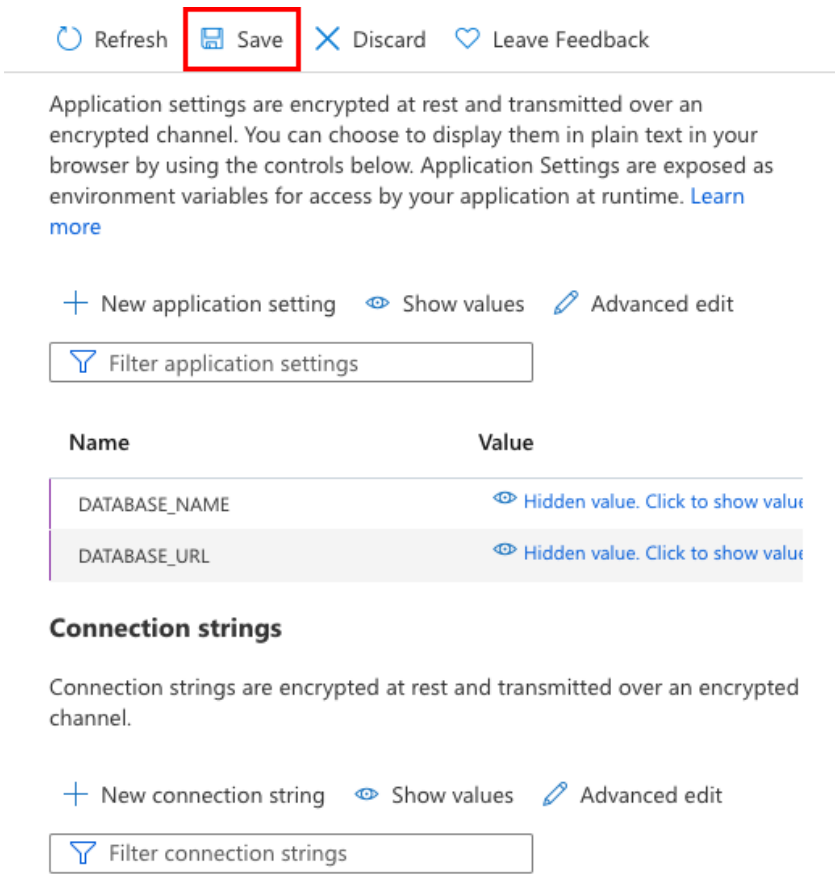

Рис. 3.7. Збереження налаштувань бази даних

Що стосується резервного копіювання в Azure Cosmos DB, у нас є два варіанти. Ми можемо покладатися на автоматичне резервне копіювання, яке постачається з Azure Cosmos DB, або можемо керувати власними резервними копіями. Для потреб нашого додатка достатньо автоматичного резервного копіювання.

3.3. Розгортання додатка

Портал Azure забезпечує готову безперервну інтеграцію та розгортання за допомогою служб Azure DevOps, GitHub, Bitbucket, FTP або локального сховища Git на машині розробки. Можна підключіть веб-програму до будь-якого з вищезазначених джерел, а App Service зробить усе інше шляхом автоматичної синхронізації коду та будь-яких майбутніх змін коду у програмі.

Azure DevOps є чудовим інструментом для реалізації CI/CD, оскільки він надає такі функції, як підтримка будь-якої мови/платформи, сумісність проектів із відкритим вихідним кодом або одночасно розгортає різні типи обчислювальних ресурсів/цілей, надає кінцевому користувачеві послідовний і якісний код. доступний, і, що найважливіше, він дає більше видимості того, що ми на ньому реалізуємо, допомагає зменшити ручну роботу під час створення та розгортання коду, що зменшує людські помилки.

3.3.1 Налаштування безперервної інтеграції

Створимо Pipeline / конвеєр.

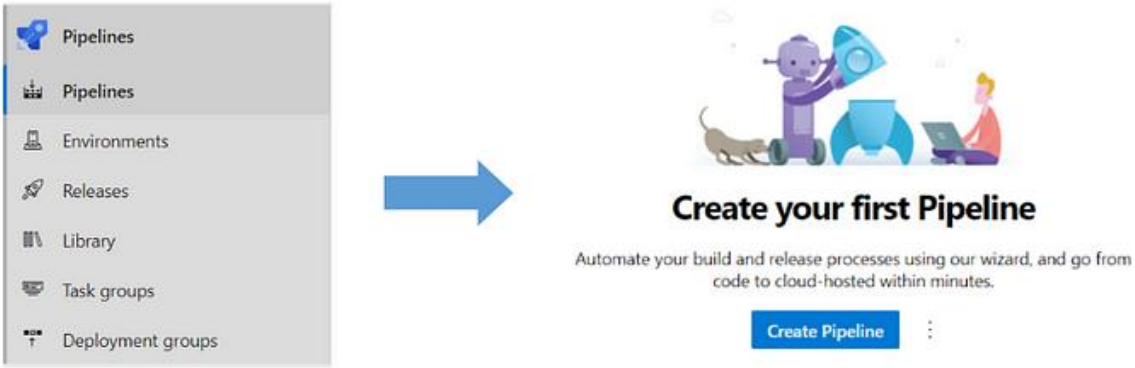

Рис. 3.8. Створення конвеєру

Виберемо опцію «Використовувати редактор / Use the Editor'».

Виберемо підключення до вихідного сховища та створимо відповідне службове підключення для авторизації до сховища.

Виберемо відповідне сховище та гілку та продовжити.

Почнемо з порожньої job.

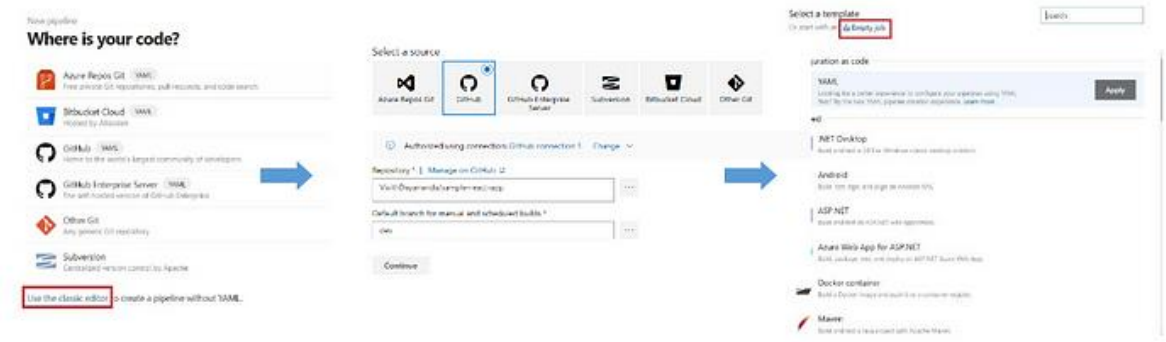

Рис. 3.9. Підключення до сховища

Виберемо необхідний пул агентів і додамо необхідні кроки, натиснувши «+» на job агента. Тут пул агентів — Azure Pipelines, а специфікація агента — остання версія Windows.

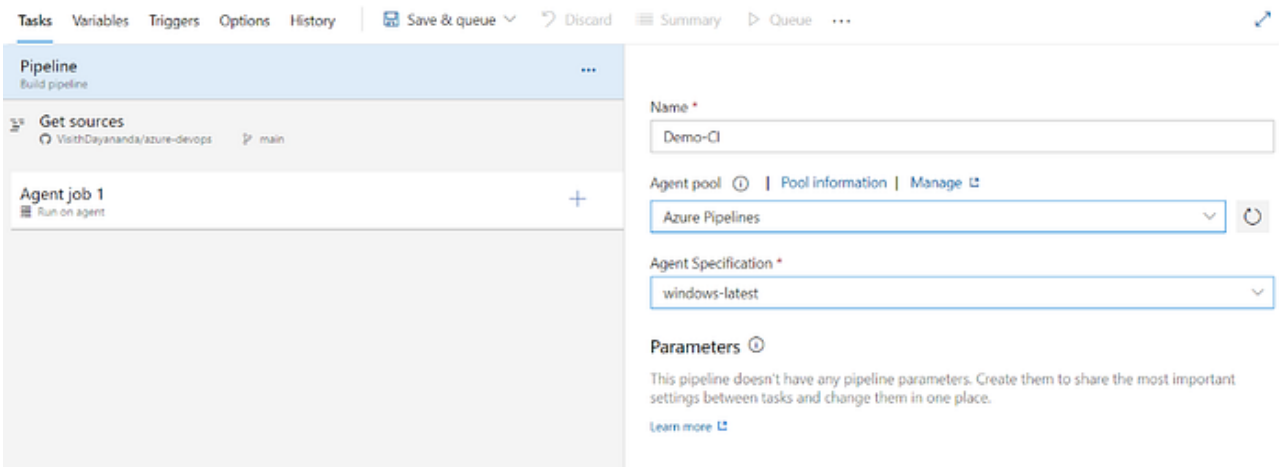

Рис. 3.10. Налаштування пула агентів

Натиснемо «+» і додамо необхідні завдання, наприклад завдання npm для збірки та завдання Publish Artifact для публікації артефакту програми.

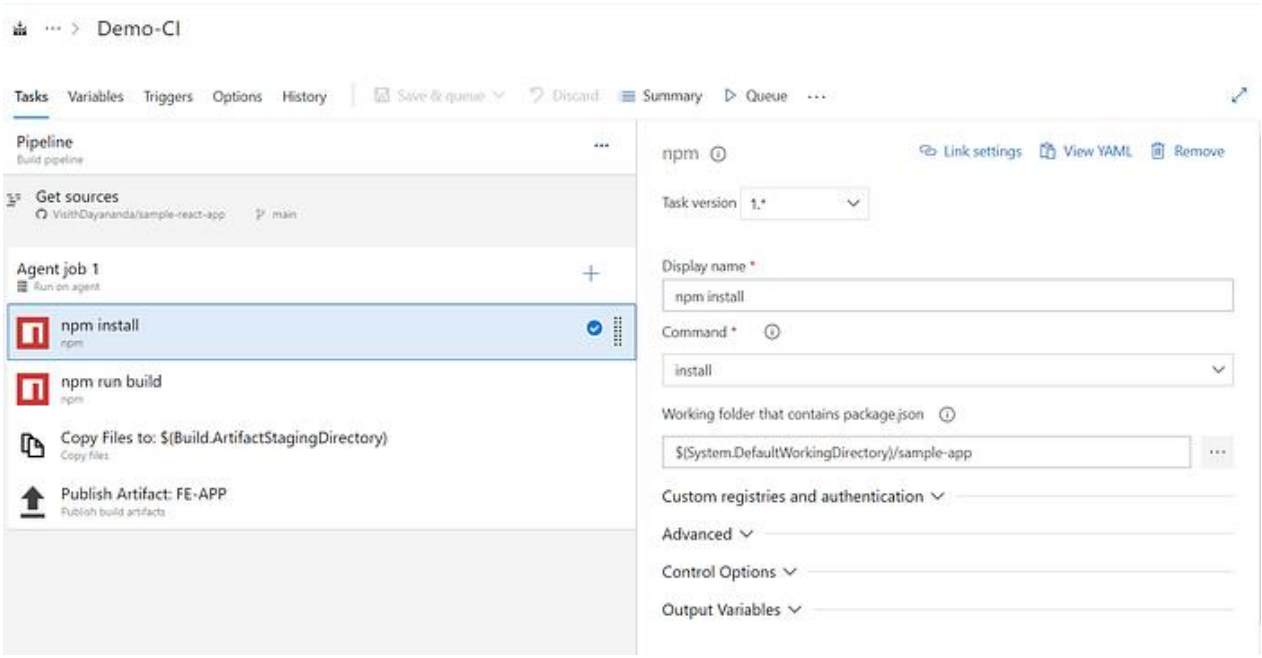

Рис. 3.11. Додавання завдань

Запустимо конвеєр, вибравши «Queue».

Після завершення прогону ми можемо перевірити журнали, вибравши відповідний прогін конвеєра.

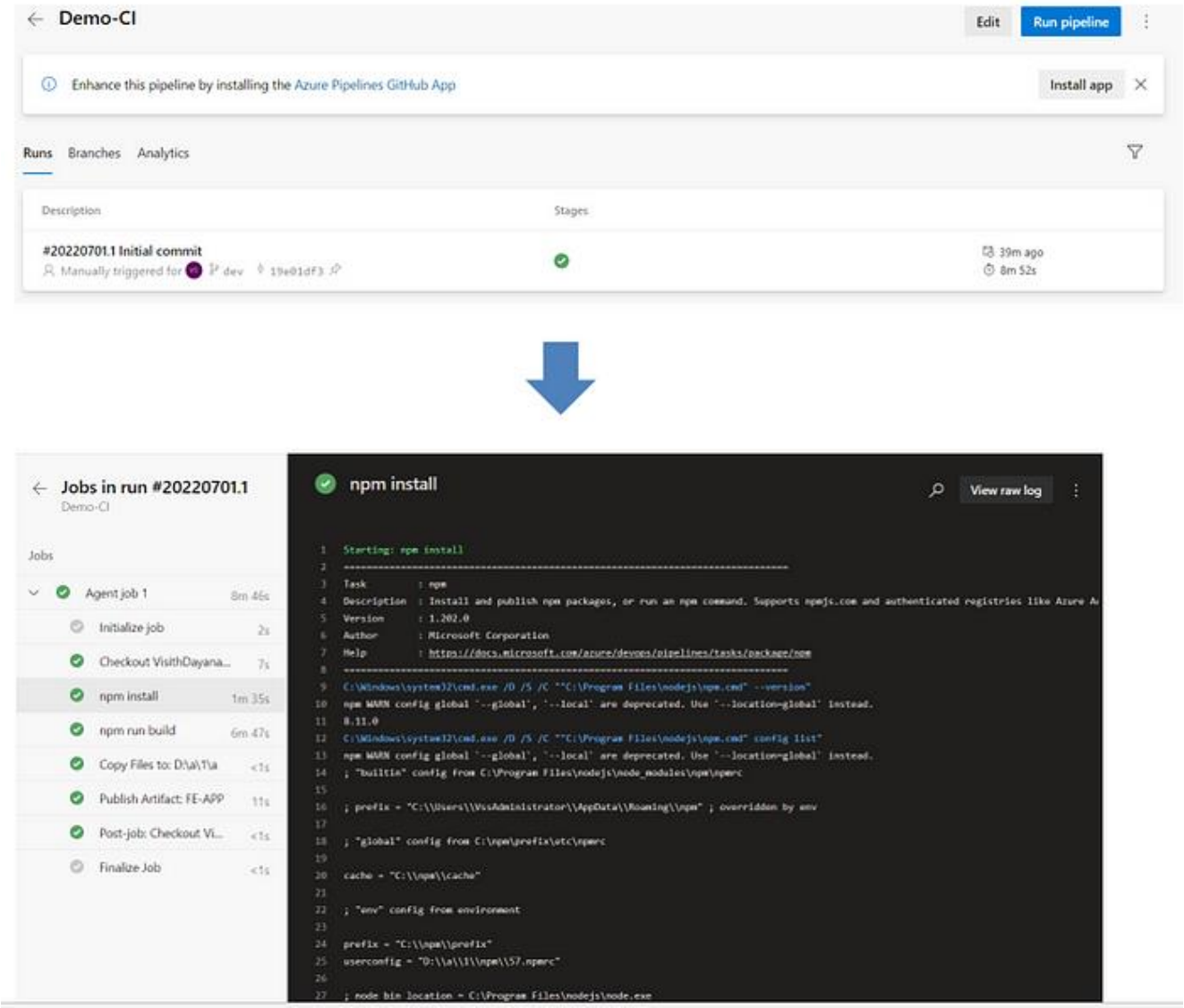

Рис. 3.12. Робота конвеєра

З цього запуску бачимо, що 1 артефакт опубліковано, і цей артефакт буде використано під час розгортання.

Рис. 3.12. Артефакт

3.3.2 Налаштування безперервної доставки

Виберемо «Releases» в розділі «Azure Pipelines» і потім виберемо «New Pipeline».

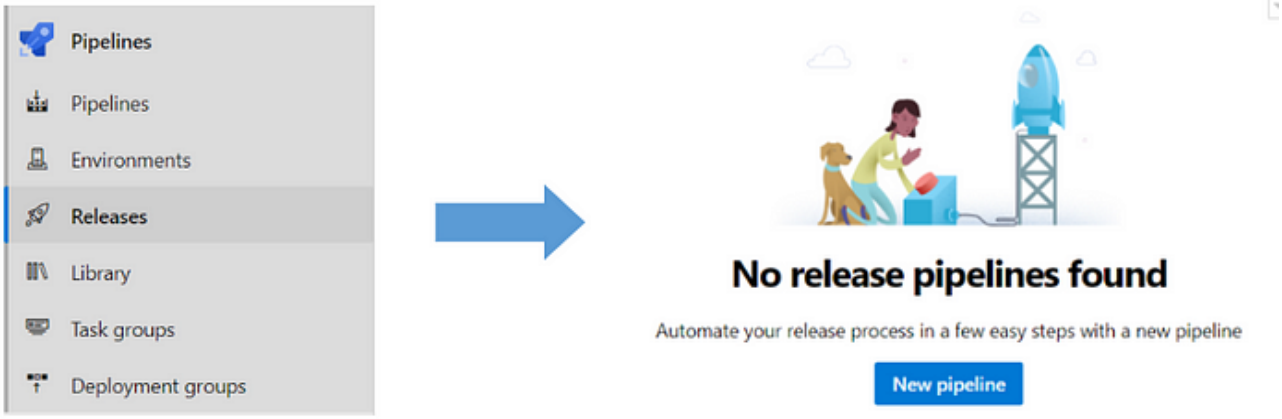

Рис. 3.12. Вибір Release

Виберемо Deploy node.js app to Azure App Service

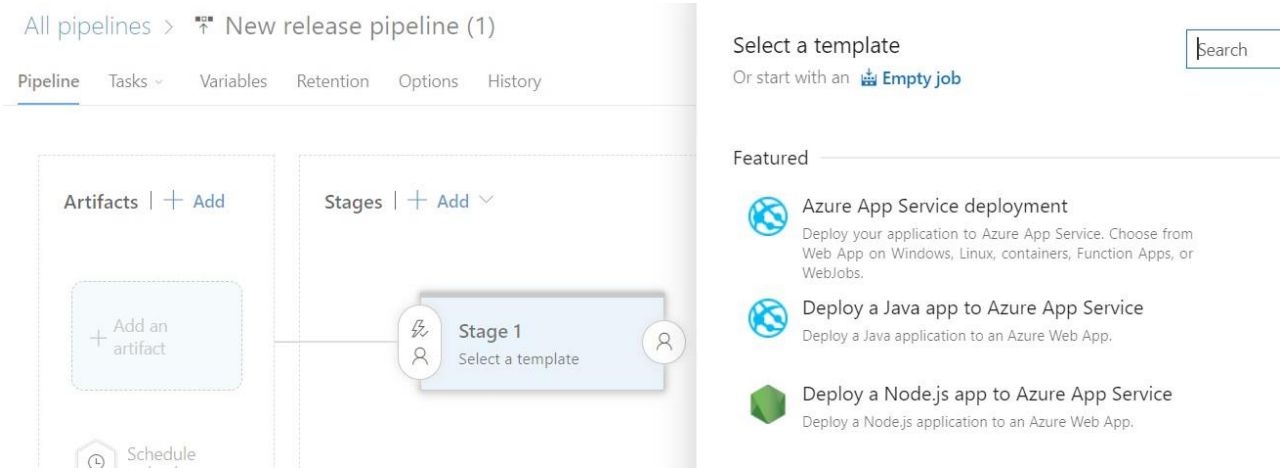

Рис. 3.13. Налаштування Release

Перейменуємо етап / stage на «Development».

Додамо артефакт, вибравши Source як конвеєр CI, який ми створили раніше.

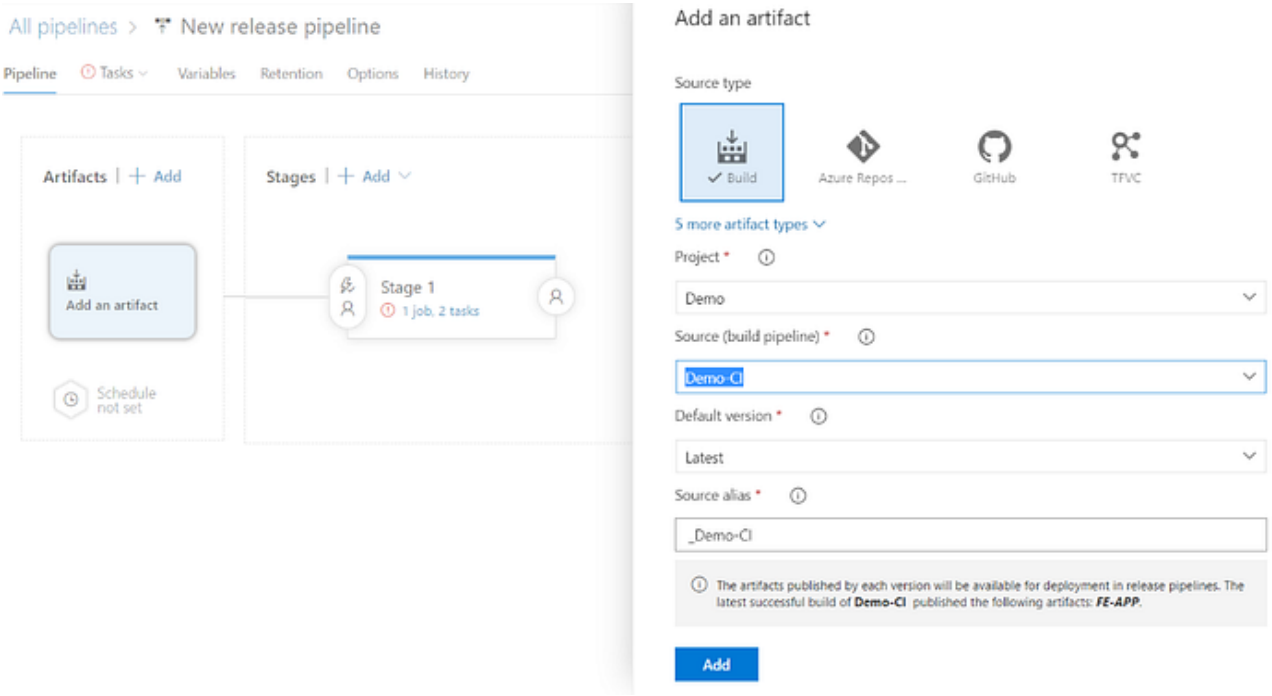

Рис. 3.14. Додавання артефакту

Перейдемо до етапу Development stage, переглянувши завдання етапу. Для цього нам потрібна група розгортання, яка має агентів, де слід розгорнути артефакт.

Створимо групу розгортання:

Azure Pipelines → Deployment Group → Add a deployment group → Create → Run the script on the respective machine

Потім цей агент розгортання реєструється в групі розгортання.

Для розгортань у середовищі клонуємо наявний етап, перейменовуємо його за назвою середовища та міняємо агента розгортання з вибраної групи розгортання.

Нарешті, наш конвеєр безперервної доставки виглядає так:

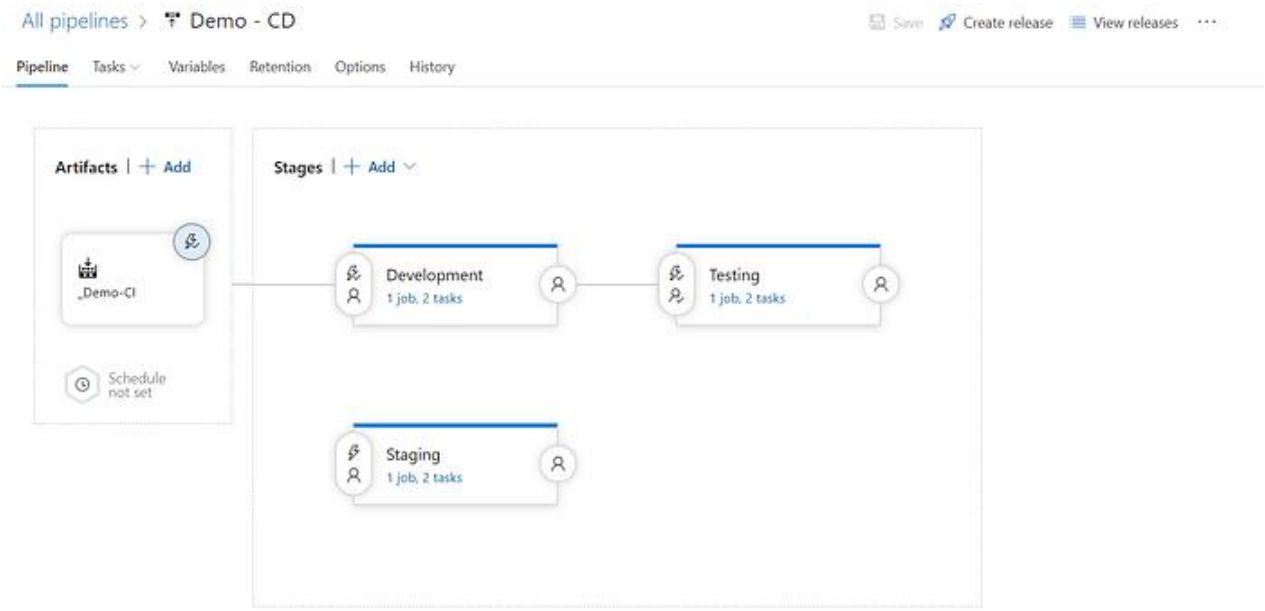

Рис. 3.14. Конвеєр безперервної доставки

3.4. Аналіз відмовостійкості Azure App Service

Після розгортання додатку, важливо, щоб ресурси були доступні, коли це необхідно. Висока доступність зосереджена на забезпеченні максимальної доступності, незалежно від збоїв або подій, які можуть статися.

Коли розробляється архітектура рішення, потрібно враховувати гарантії доступності послуг. Azure — це високодоступне хмарне середовище з гарантіями безвідмовної роботи залежно від служби. Ці гарантії є частиною угод про рівень обслуговування (SLA).

Azure App Service додає програмі можливості Microsoft Azure, наприклад:

- безпека;
- балансування навантаження;
- автомасштабування;
- автоматизоване управління.

Додаток має масштабованість, що означає здатність налаштовувати ресурси відповідно до попиту. Якщо раптово виникне пік трафіку та системи

стануть перевантажені, можливість масштабування означає, що можна додати більше ресурсів, щоб краще справлятися зі збільшеним попитом.

Ще одна перевага масштабованості полягає в тому, що ми не переплачуємо за послуги. Оскільки хмара — це модель, заснована на споживанні, ми платимо лише за те, що використовуємо. Якщо попит впаде, ми можемо скоротити свої ресурси та тим самим зменшити витрати.

Масштабування зазвичай буває двох видів: вертикальне та горизонтальне. Вертикальне масштабування орієнтоване на збільшення або зменшення можливостей ресурсів. Горизонтальне масштабування - це додавання або віднімання кількості ресурсів.

Увімкнення зон доступності не вимагає додаткових витрат. Ціноутворення для служби додатків із резервуванням зони таке ж, як і для служби додатків для однієї зони. З нас стягуватиметься плата на основі SKU нашого тарифного плану App Service, указаної нами ємності та будь-яких екземплярів, які ми масштабуємо відповідно до критеріїв автоматичного масштабування. Якщо ми ввімкнете зони доступності, але вкажемо ємність менше трьох, платформа запровадить мінімальну кількість екземплярів у три та стягне плату за ці три екземпляри.

3.5. Аналіз відмовостійкості Cosmos DB

Azure Cosmos DB автоматично створює резервні копії даних кожні 4 години, не впливаючи на продуктивність або доступність. Ці резервні копії зберігаються в Azure Blob Storage у тому самому регіоні, що й ваш регіон запису Cosmos DB (якщо для облікового запису Cosmos DB налаштовано кілька головних копій, резервні копії зберігаються в одному з місць запису).

Ці резервні копії створюються кожні 4 години, і зберігаються дві останні резервні копії. Якщо видалите контейнер або базу даних у своєму обліковому записі, наявні знімки цих контейнерів або баз даних зберігаються протягом 30 днів.

На щастя, ці знімки створюються без використання будь-якої пропускної здатності вашого облікового запису, тому вам не потрібно надавати будь-яку додаткову пропускну здатність для баз даних або контейнерів Cosmos DB для створення резервних копій.

Azure Cosmos DB може відновити дані в будь-якій із наведених нижче ситуацій:

- весь обліковий запис буде видалено;
- одну або кілька баз даних видалено;
- один або кілька контейнерів видалено;
- елементи всередині контейнера видаляються або змінюються;
- контейнер пропозицій спільної пропускної здатності в спільній базі даних пропозицій видалено або пошкоджено.

Згідно з документацією, новий обліковий запис Cosmos DB буде створено для зберігання відновлених даних. Якщо ви перебуваєте на порталі в цей час, ви побачите обліковий запис Cosmos DB із такою назвою:

<Azure_Cosmos_Account_Name>-restored1

остання цифра покаже кількість спроб відновлення, які були зроблені.

Azure Cosmos DB забезпечує:

- забезпечує глобальне поширення в 30+ регіонах світу;
- забезпечує 99,99% доступність для одного регіону, і 99,999% доступність для читання в облікових записах бази даних кількох регіонів;
- забезпечує незалежне масштабування сховища та пропускної здатності;
- забезпечує 99% читання за < 10 мс і запису за < 15 мс;
- завжди ввімкнена та пропонує плани на випадок непередбачених простоїв.

3.6. Налаштування моніторингу

Для того, щоб розуміти чи все працює правильно і стабільно, чи достатньо ресурсів для роботи додатку нам потрібно налаштувати моніторинг системних ресурсів, зберігати данні використання ресурсів та алертинг для оперативного усунення проблем або превентивного виконання дій для попередження можливих проблем.

Хмарний моніторинг — це набір методів, які використовуються для відстеження продуктивності, безпеки та доступності хмарних служб і програм. Моніторинг допомагає оптимізувати інфраструктуру, виявляти збої в роботі служб і виявляти вразливі місця або загрози в обліковому записі. Є три основні причини, чому хмарний моніторинг корисний, а саме:

- Моніторинг продуктивності дозволяє відстежувати доступність ресурсів і служб, забезпечуючи відповідність очікуванням користувачів або вимогам щодо робочого навантаження. Крім того, це допомагає виявити вузькі місця або недостатні ресурси, дає змогу оптимізувати конфігурації та усувати помилки програми;
- Моніторинг безпеки гарантує, що доступ до даних, облікових записів і програм є безпечним, а інфраструктура та програми постійно оновлюються. Встановлення найвищих стандартів безпеки та постійного моніторингу для забезпечення відповідності стандартам, виявлення кіберзагроз і відстеження шкідливих дій;
- Моніторинг витрат дає змогу відстежувати всі ресурси, що використовуються ефективно або недостатньо, і навіть ті, які забуті та переміщені до невикористаних потужностей. Таким чином стає легко оптимізувати та змінювати вартість використовуваних ресурсів, які запущені та переведені в режим простою або зупинені.

У цій справі нам допоможе Azure Monitor. Він збирає дані для нас у фоновому режимі. Azure Monitor збирає показники та логи, пов’язані з нашими ресурсами Azure, і використовує дані для створення сповіщень, моніторингу продуктивності, усунення проблем і розробки інформаційних панелей, щоб забезпечити повну видимість майна Azure для прийняття рішень, особливо коли виникають проблеми.

Є кілька способів використання зібраних показників або даних, як-от виявлення та діагностика проблем у програмах і залежностях за допомогою статистичних даних програм. Для усунення несправностей і глибокої діагностики детально ознайомимося з даними моніторингу за допомогою Log Analytics. Той самий набір даних використовується для створення візуалізацій за допомогою інформаційних панелей і робочих книг Azure для більш спрощеного доступу до стану програми. Це також дає змогу співвідносити проблеми інфраструктури з аналітичними даними віртуальних машин і контейнерів.

Потім Azure Monitor організовує дані, які він збирає для моніторингу, за двома типами: показники та журнали / логи.

- Показники / метрики: Логи метрик надають візуалізовані показники на основі груп ресурсів підписки та типу ресурсу та представляють продуктивність і стан системи у зручних числових значеннях, які можна переглянути на графіках. Можна відстежувати та перевіряти продуктивність додатків і баз даних, починаючи від використання ЦП і закінчуючи середнім відсотком використання тощо;
- Логи: Azure зі своїм інструментом моніторингу надає детальну інформацію про події логів в логах активності або перегляді подій у операційній системі. Події логів складаються зі змін ресурсів у групах або підписках, наприклад додавання або видалення, а також дії конкретних користувачів. Водночас у логах активності перераховуються будь-які дії, наприклад перебої в мережі.

Показники корисні, оскільки вони представляють деякі аспекти вашої програми або системи в певний час для історичної довідки. Вони використовуються для досить швидкого налаштування сповіщень. У той же час логи – це події телеметрії, організовані в текстові записи, які надають додатковий контекст, наприклад, коли певний ресурс було створено або змінено востаннє або коли було виявлено помилку в програмі. У поєднанні з показниками це допомагає проаналізувати та визначити повну картину для глибшого аналізу та визначення тенденцій у часі.

Важливо відстежувати та аналізувати ресурси, коли це необхідно, а також інформувати та реагувати на проблеми, виявлені в зібраних даних. Для того, щоб випереджати потенційні виклики, потрібно мати способи підсумовувати та подавати дані моніторингу в доступний та ефективний спосіб. Azure Monitor надає низку можливостей, які покращують здатність бізнес-команд та IT-команд активно та продуктивно вирішувати проблеми завчасно. Є кілька функцій, які можна використовувати на порталі Azure, щоб максимально використати ресурси Azure, зокрема:

- Універсальна платформа моніторингу: з єдиної інформаційної панелі ви отримуєте видимість і контроль над своїми програмами, інфраструктурою та мережею. Це дозволяє відстежувати та оптимізувати ваші ресурси з урахуванням бізнес-вимог, таких як доступність, продуктивність і вартість.
- Повна інтеграція сторонніх розробників: Azure Monitor підтримує всі популярні мови програмування та фреймворки, включаючи .NET, Java та Node.js, і інтегрується з процесами та інструментами DevOps, такими як Azure DevOps, PagerDuty та Jira. За допомогою уніфікованої інформаційної панелі можна відстежувати живі метрики, запити, час відповіді та події.
- Працездатність служби: логи та метрики є двома потужними інструментами Azure Monitor, але є ще один розділ потужності, про який слід згадати: справність. Розділ справності на порталі Azure розкрийте

інформацію про справність Azure в цілому. Розміщуючи проблеми з обслуговуванням Azure, заплановане технічне обслуговування та історію працездатності разом із показниками нашого облікового запису, Azure неймовірно легко визначати як локальні, так і глобальні проблеми з інфраструктурою додатків.

- Power BI: за допомогою Power BI Azure Monitor забезпечує самообслуговування бізнес-аналітики, як-от більш глибокі та інтерактивні візуалізації багатьох джерел даних. Таким чином, ви отримуєте інформаційні панелі та звіти, багаті даними, за кілька кліків.
- Сповіщення. Однією з найцікавіших і найважливіших функцій Azure Monitor є сповіщення. Інструмент має вбудовані зміни, які сповіщають користувачів про потенційні або поточні проблеми, відфільтровані за минулими годинами, як-от 6 годин або останні 24 години. Сповіщення можна автоматизувати або обробляти вручну. Тому ви можете налаштувати автоматичний процес для вжиття коригувальних дій, коли виникають проблеми та отримуєте сповіщення про діяльність. Правила сповіщень встановлюються на основі метрик, отже надаючи сповіщення, наближені до режиму реального часу.

Розділ Monitor на порталі Azure надає візуальний інтерфейс, який надає доступ до даних, зібраних для ресурсів Azure, і простий спосіб отримати доступ до інструментів, аналітичних даних і візуалізацій у Azure Monitor.

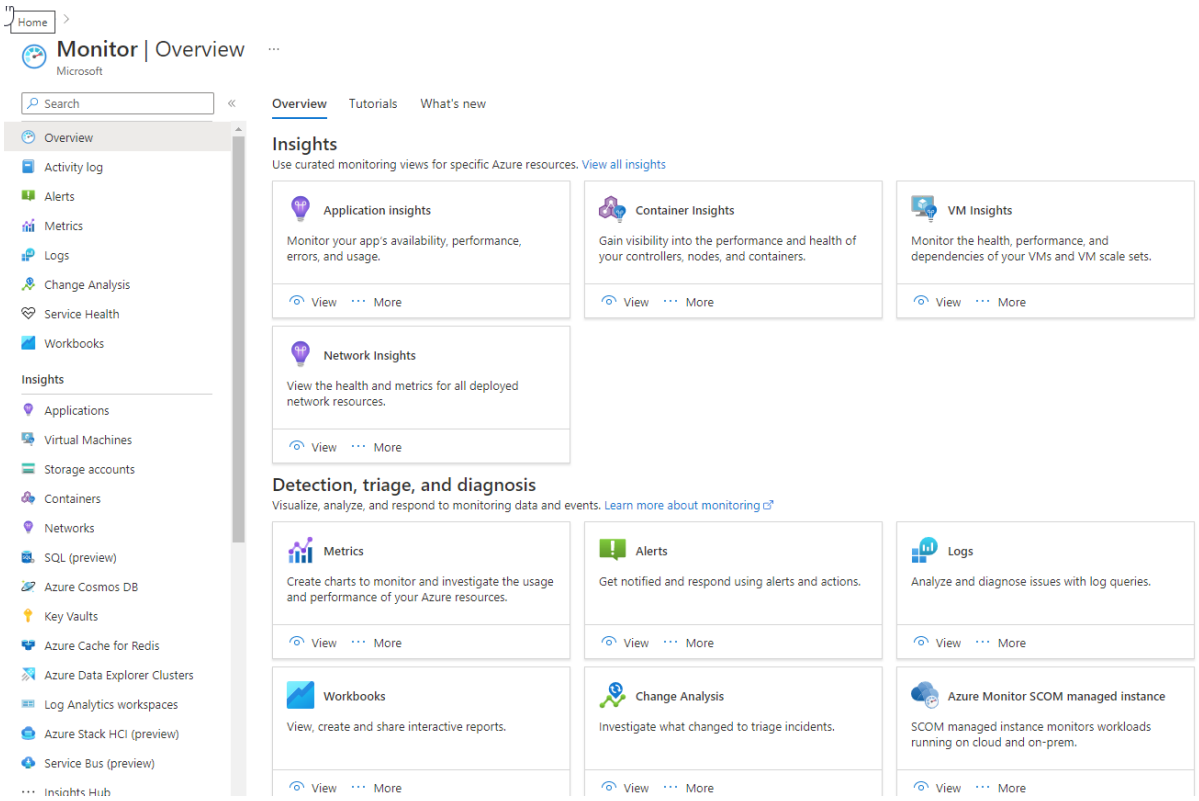

Рис. 3.15. Розділ Monitor

Розглянемо ці вбудовані інструменти для аналізу даних моніторингу.

Metrics Explorer (дослідник метрик) – для дослідження справності і використання ресурсів. Дослідник метрик допомагає будувати діаграми, візуально співвідносити тенденції та досліджувати стрибки та спади значень показників. Провідник показників містить функції для застосування параметрів і фільтрації, а також для налаштування діаграм. Ці функції допомагають аналізувати саме ті дані, які нам потрібні, візуально інтуїтивно зрозумілим способом.

Log Analytics. Інтерфейс користувача Log Analytics на порталі Azure допомагає запитувати дані логів, зібрані Azure Monitor, щоб ми могли швидко отримувати, консолідувати та аналізувати зібрані дані. Після створення тестових запитів ми можемо безпосередньо проаналізувати дані за допомогою інструментів Azure Monitor або зберегти запити для використання з візуалізаціями чи правилами сповіщень. Робочі області Log Analytics базуються на Azure Data Explorer із використанням потужного механізму аналізу та

розширеної мови запитів Kusto (KQL). Azure Monitor Logs використовує версію мови запитів Kusto, яка підходить для простих запитів логів, і розширені функції, такі як агрегації, об’єднання і інтелектуальна аналітика.

Change Analysis (Аналіз змін) — це постачальник ресурсів Azure на рівні підписки, який перевіряє зміни ресурсів у підписці та надає дані для інструментів діагностики, щоб допомогти користувачам зрозуміти, які зміни могли спричинити проблеми. Інтерфейс користувача Change Analysis на порталі Azure дає нам уявлення про причини проблем з активним сайтом, збоїв або збоїв компонентів. Аналіз змін використовує Azure Resource Graph для виявлення різних типів змін, від рівня інфраструктури до розгортання програми.

Ефективне рішення для моніторингу проактивно реагує на критичні події, не потребуючи, щоб окрема особа чи команда помічали проблему. Відповіддю може бути текстове повідомлення або електронний лист адміністратору, або автоматизований процес, який намагається виправити помилку.

Штучний інтелект для IT-операцій (AIOps) підвищує якість і надійність послуг за допомогою машинного навчання для обробки та автоматичного реагування на дані, які збираються з програм, служб і IT-ресурсів, у Azure Monitor. Він автоматизує завдання, керовані даними, прогнозує використання ємності, виявляє проблеми з продуктивністю та виявляє аномалії в програмах, службах та IT-ресурсах. Ці функції спрощують IT-моніторинг і операції, не вимагаючи досвіду машинного навчання.

Сповіщення Azure Monitor сповіщають про критичні умови та можуть вжити заходів для виправлення. Правила сповіщень базуються на даних метрик або логів.

- Правила сповіщень про метрики забезпечують сповіщення майже в реальному часі на основі зібраних показників.
- Правила сповіщень логів, засновані на логах, дозволяють використовувати складну логіку для даних із багатьох джерел.

Правила сповіщень використовують групи дій, які можуть виконувати такі дії, як надсилання сповіщень електронною поштою або SMS. Групи дій

можуть надсилати сповіщення за допомогою веб-хуків для запуску зовнішніх процесів або інтеграції з інструментами керування IT-послугами. Групи дій, дії та набори одержувачів можуть використовуватися в кількох правилах.

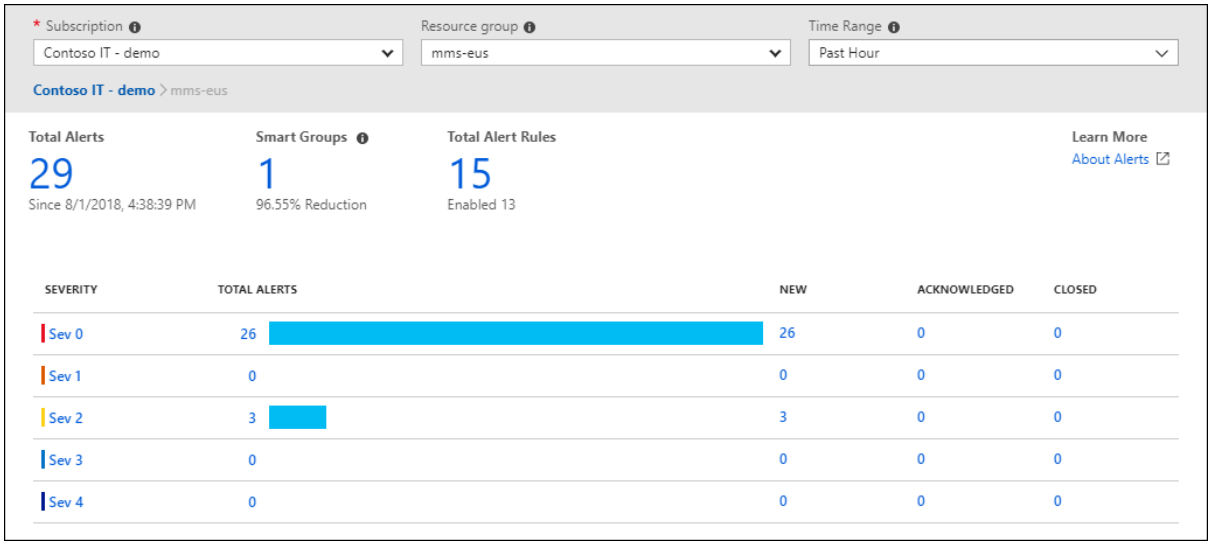

Рис. 3.16. Налаштування групи дій

Автоматичне масштабування дозволяє динамічно контролювати кількість запущених ресурсів для обробки навантаження на програму. Ми можемо створити правила, які використовують метрики Azure Monitor, щоб визначати, коли автоматично додавати ресурси, коли навантаження зростає, або видаляти ресурси, які простоюють. Ми можемо вказати мінімальну та максимальну кількість екземплярів, а також логіку збільшення або зменшення ресурсів, щоб заощадити гроші та підвищити продуктивність.

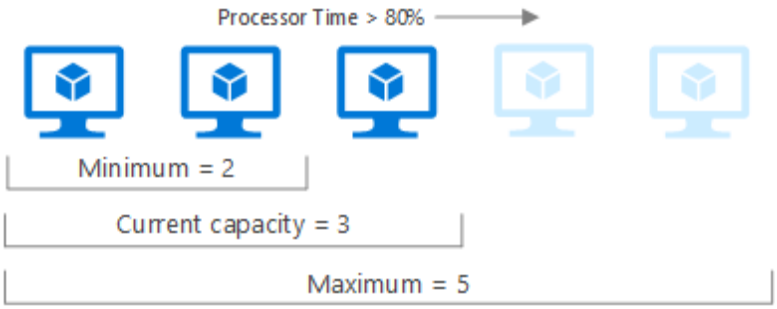

Рис. 3.17. Автоматичне масштабування

Щоб створити нове сповіщення, натиснемо опцію Alert (Сповіщення), а потім “+ New alert rule”.

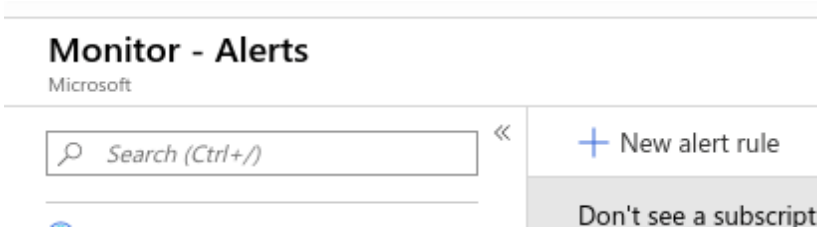

Рис. 3.18. Створення сповіщення

Виберемо ресурси для моніторингу. Панель, що з’явиться, дозволяє фільтрувати за типом ресурсу та розташуванням. Можна створити сповіщення лише для ресурсів одного типу та в одному місці. Якщо є ресурси в інших місцях, доведеться створити додаткові правила (ймовірно, з однаковими налаштуваннями) для кожного іншого місця.

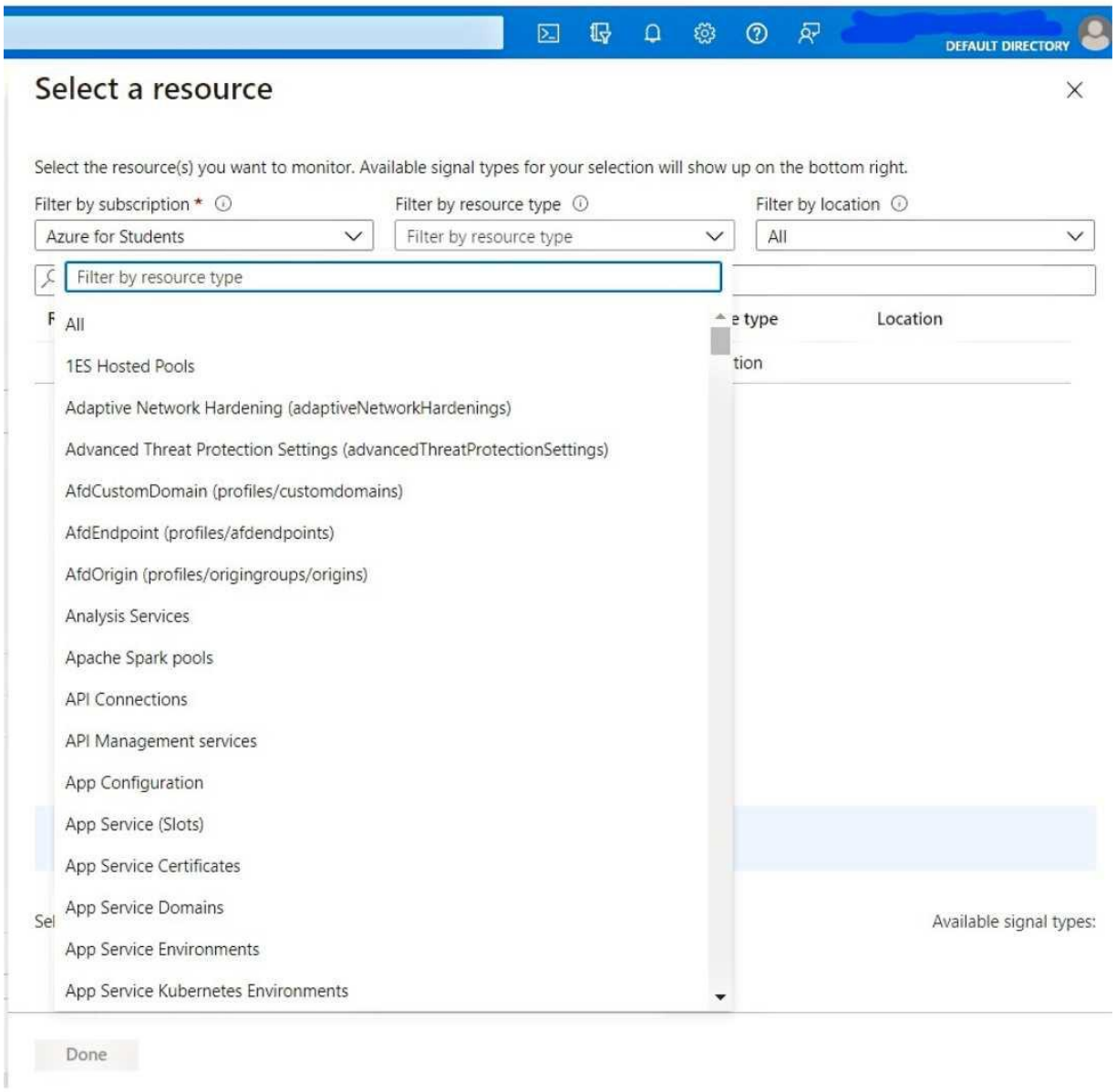

Рис. 3.19. Вибір ресурсів для сповіщення

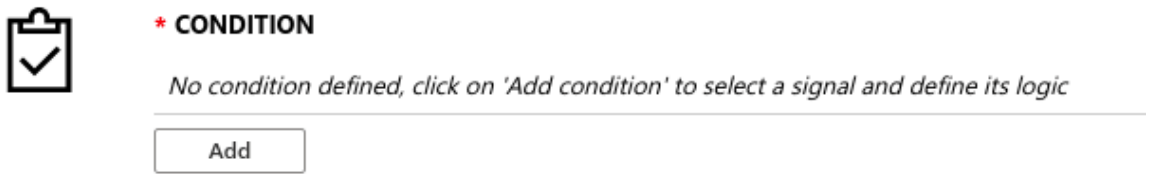

Рис. 3.20. Додавання умови для ініціювання сповіщення

Для цього почнемо із вибору показника, який хочемо контролювати.

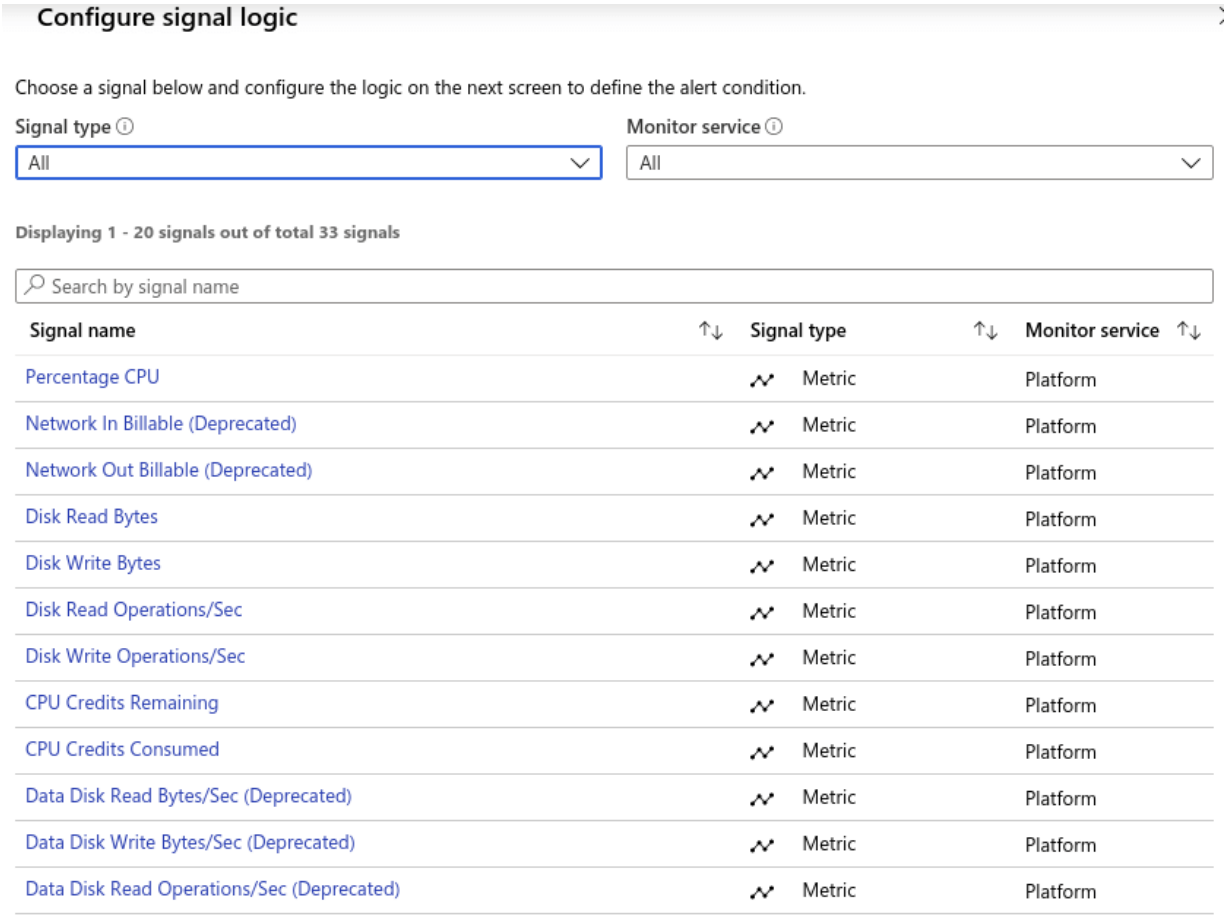

Рис. 3.21. Вибору показника для контролю

Встановимо поріг, про який хочемо сповістити. Якщо знаємо конкретне значення метрики, про яке хочемо сповіщати, створимо статичне сповіщення з цим порогом. В іншому випадку створимо динамічний поріг. Це дозволить Azure визначити порогове значення на основі історичного значення нашого показника, і його можна налаштувати на високу, середню або низьку чутливість. Також потрібно буде вибрати деталізацію показників і частоту оцінки для сповіщення.

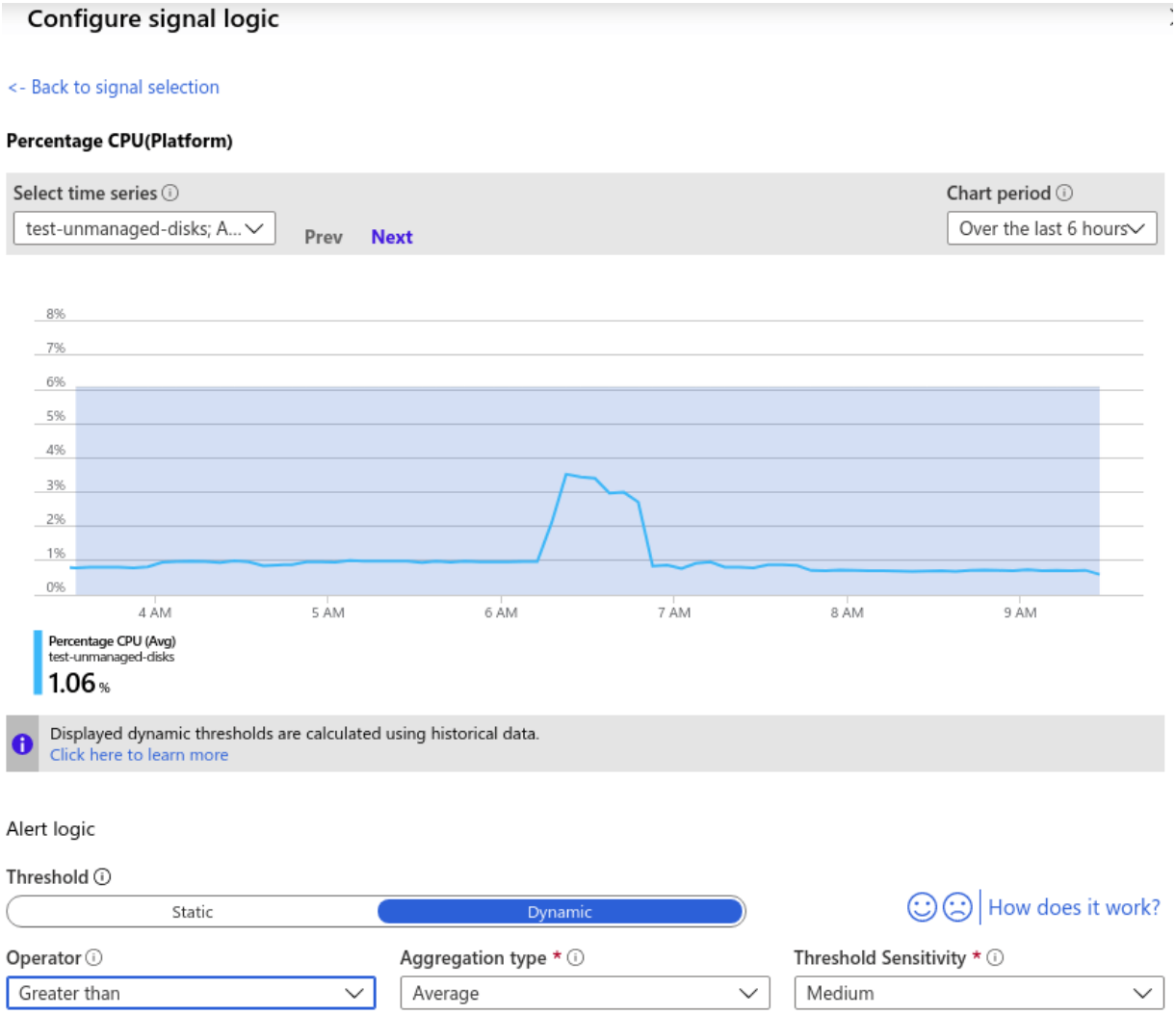

Рис. 3.22. Налаштування порога спрацьовування

Додамо групу дій до сповіщення. Це вкаже, що робити, коли спрацьовує сповіщення. Azure дозволяє виконувати багато дій, зокрема надсилати сповіщення, запускати функції Azure або викликати вебхук.

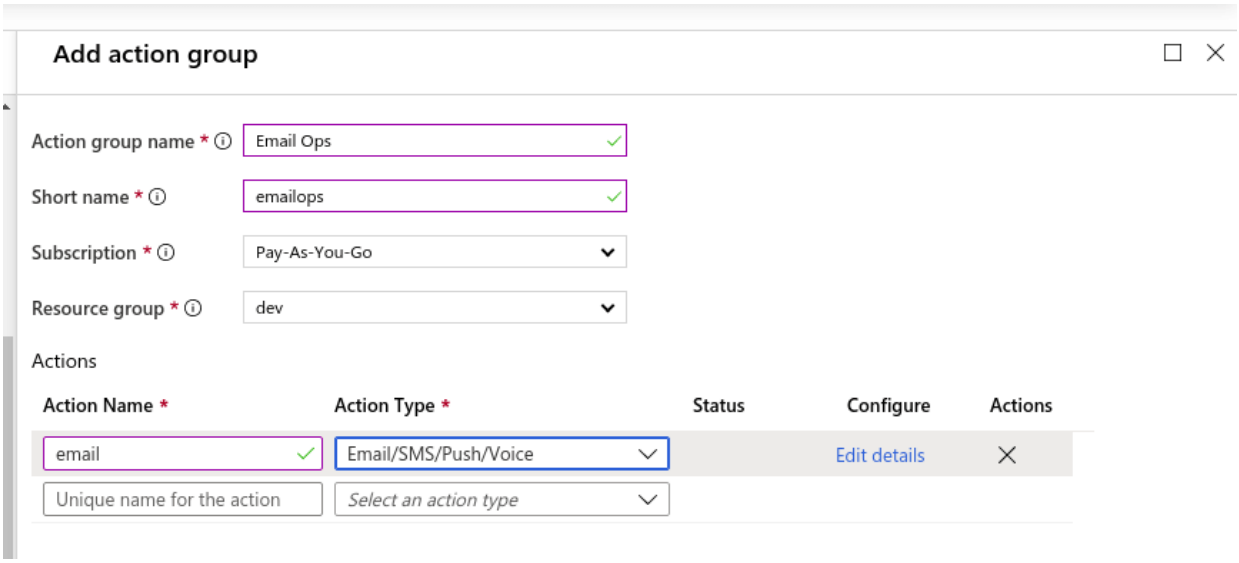

Рис. 3.23. Додавання групи дій до сповіщення

Нарешті введемо деталі сповіщення та збережемо нове сповіщення. Це включає встановлення рівня серйозності для сповіщення.

Рис. 3.24. Налаштування деталей сповіщення

3.7. Налаштування Application Insights

Application Insights допомагає контролювати, діагностувати та покращувати продуктивність і доступність веб-програм і служб. Він надає уявлення про використання, продуктивність, доступність і поведінку користувачів нашої програми, дозволяючи виявляти та виправляти проблеми, перш ніж вони вплинуть на наших клієнтів.

Крок1: Створимо ресурс Application Insights

- на порталі Azure виберемо «Створити ресурс» у меню ліворуч;
- знайдемо «Application Insights» і виберемо його зі списку служб;
- натиснемо «Створити» та заповнимо необхідну інформацію для ресурсу, таку як ім’я, підписка, група ресурсів і розташування;
- натиснемо «Переглянути + створити», а потім «Створити», щоб створити ресурс.

Крок 2. Увімкнемо Application Insights для служби додатків Azure

- перейдемо до служби додатків Azure, яку хочемо контролювати за допомогою Application Insights;
- натиснемо «Application Insights» у розділі «Моніторинг» у меню ліворуч;
- у розкривному меню виберемо ресурс Application Insights, який ми створили на кроці 1;
- натиснемо «Увімкнути», щоб увімкнути Application Insights для служби додатків

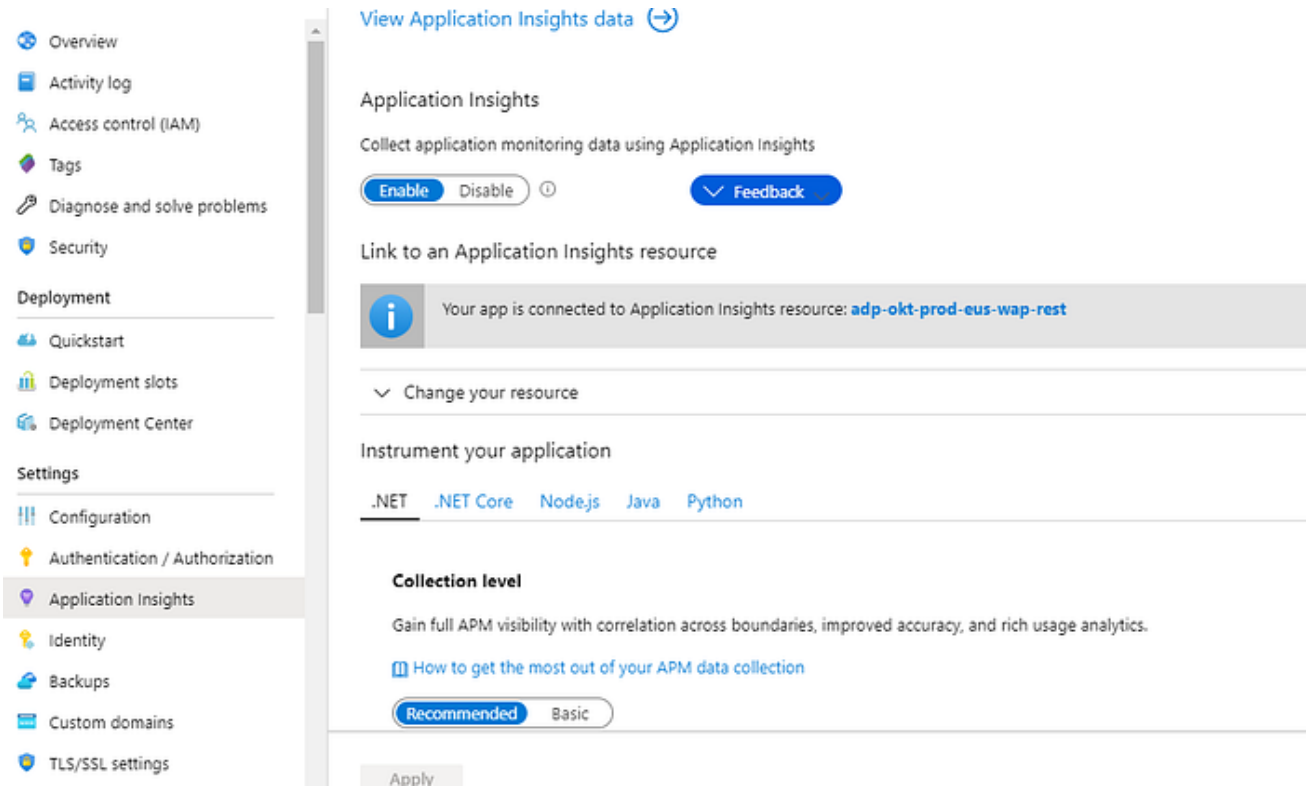

Рис. 3.25. Вмикання Application Insights

Крок 3. Налаштуємо Application Insights

– повернемось до свого ресурсу Application Insights на порталі Azure;
– натиснемо «Налаштувати» в меню зліва;
– виберемо вкладку «Налаштування програми»;
– прокрутимо вниз до розділу «Ключ інструментів» і скопіюємо значення «Ключ інструментів»;
– повернемось до служби додатків на порталі Azure;
– натиснемо «Конфігурація» в розділі «Параметри» меню ліворуч;
– додаємо нове налаштування з назвою «APPINSIGHTS_INSTRUMENTATIONKEY» та вставимо значення інструментального ключа, скопійоване на кроці 4, як значення;
– натиснемо «Зберегти», щоб зберегти зміни конфігурації.

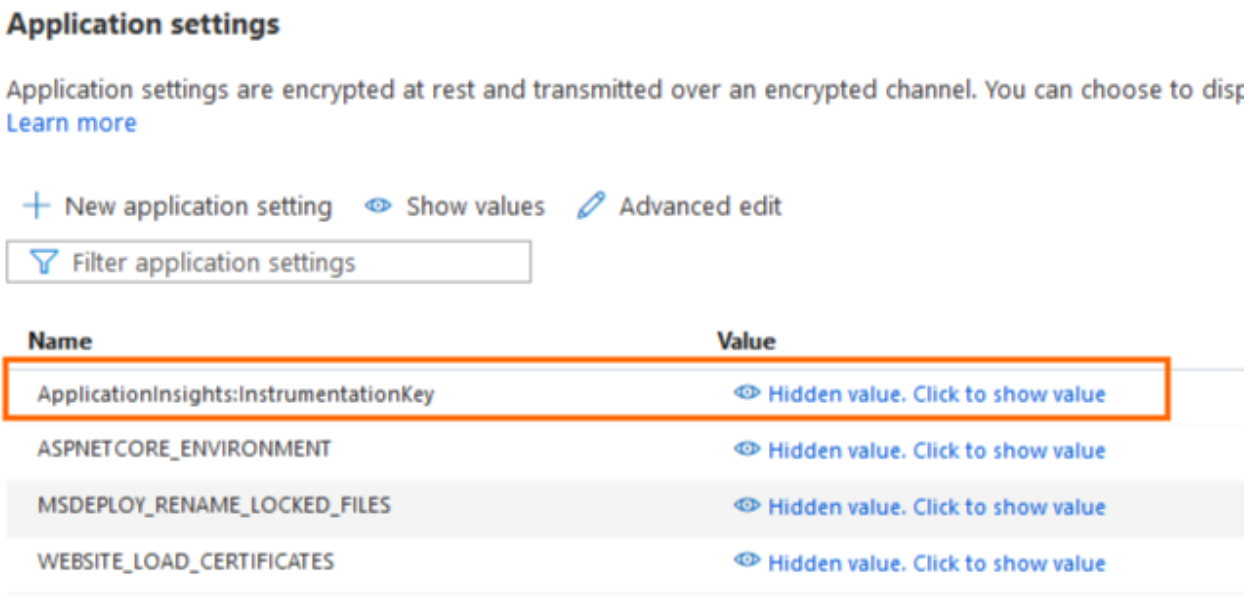

Рис. 3.26. Налаштування Application Insights

Крок 4. Переглянемо дані Application Insights

- перейдемо до свого ресурсу Application Insights на порталі Azure;
- натиснемо «Огляд» у меню ліворуч;
- виберемо вкладку «Пряма трансляція показників», щоб переглянути дані про службу додатків у реальному часі;
- виберемо вкладку «Карта програми», щоб переглянути високорівневе подання компонентів служби програми;
- виберемо вкладку «Продуктивність», щоб переглянути детальну інформацію про продуктивність служби додатків.

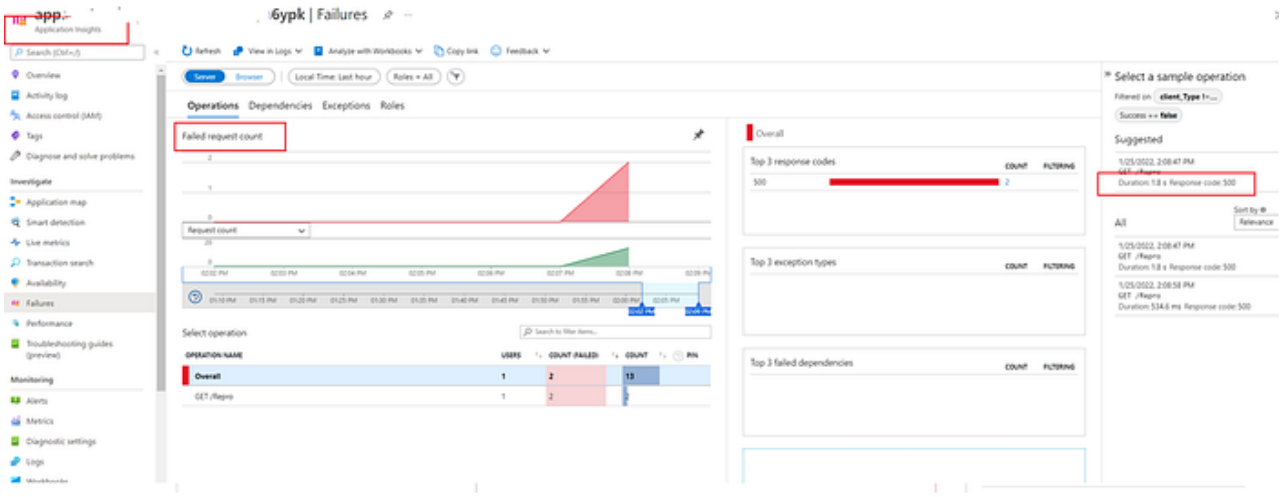

Рис. 3.27. Дані Application Insights

3.8. Аналіз режиму відмови

Для виявлення та визначення пріоритетів потенційних збоїв у компонентах нашого рішення будем використовувати аналіз режиму відмови (FMA). Виконаємо FMA, щоб допомогти оцінити ризик і наслідки кожного режиму відмови. Визначимо, як робоче навантаження реагує та відновлюється. Проаналізуємо компоненти нашої інформаційної системи.

FMA — це практика виявлення потенційних точок збою у робочому навантаженні та пов’язаних з ним потоках і відповідно планування дій із зменшення наслідків. На кожному кроці процесу визначається радіус вибуху кількох типів відмов, що допомагає розробити нове робоче навантаження або змінити існуюче робоче навантаження, щоб мінімізувати поширений вплив відмов.

Ключовий принцип FMA полягає в тому, що збої трапляються незалежно від того, скільки рівнів стійкості ви застосовуєте. Більш складні середовища піддаються більшій кількості відмов. Враховуючи цю реальність, FMA дає змогу спроектувати робоче навантаження таким чином, щоб воно витримало більшість типів збоїв і плавно відновлювалось у разі виникнення збою.

Якщо взагалі пропустити FMA або виконати неповний аналіз, робоче навантаження ризикує статися непередбаченою поведінкою та потенційними збоями, спричиненими неоптимальним дизайном.

У наведеній нижче таблиці наведено FMA для нашого веб-сайту електронної комерції, який розміщено на екземплярах служби додатків Azure з базою даних Cosmos DB. Потік користувача: вхід користувача, пошук продукту та взаємодія з кошиком для покупок.

Таблиця 3.3

Аналіз режиму відмови

Компонент	**Риск**	**Ймовірність**	**Ефект/пом'якшення/примітка**	**Відключення**
Microsoft Entra ID	Збій служби	Низька	Повний збій робочого навантаження. Виправлення залежить від Microsoft.	Повне
Microsoft Entra ID	Неправильна конфігурація	Середня	Користувачі не можуть увійти. Немає ефекту downstream. Служба підтримки повідомляє про проблему конфігурації команді ідентифікації.	Немає
Cosmos DB	Збій служби	Низька	Повний збій робочого навантаження. Виправлення залежить від Microsoft.	Повне
Cosmos DB	Регіональний збій	Дуже низька	Група автоматичного перемикання (Auto-failover group) після відмови переходить до додаткового регіону. Потенційний збій під час відновлення після відмови. Цільовий час відновлення (Recovery time objectives, RTO) і цілі точки відновлення (recovery point objectives, RPO), які визначаються під час тестування надійності.	Потенційно повне
Cosmos DB	Збій зони доступності	Низька	Жодного ефекту	Немає
Cosmos DB	Зловмисна атака (ін'єкція)	Середня	Мінімальний ризик. Усі екземпляри Azure Cosmos DB прив’язані до віртуальної мережі через приватні кінцеві точки (private endpoints), а групи безпеки мережі (network security groups, NSG) додають додатковий захист у віртуальній мережі.	Потенційний низький ризик
App Service	Збій служби	Низька	Повний збій робочого навантаження. Виправлення залежить від Microsoft.	Повне
App Service	Регіональний збій	Дуже низька	Мінімальний ефект. Затримка для користувачів у постраждалих регіонах. Azure Front Door автоматично спрямовує трафік до незачеплених регіонів.	Немає
App Service	Збій зони доступності	Низька	Жодного ефекту. Служби додатків розгорнуто як резервні зони. Без резервування зон є потенціал для ефекту.	Немає
App Service	DDoS attack	Середня	Мінімальний ефект. Вхідний трафік захищений Azure Front Door і Well-Architected Framework.	Немає

3.9. Висновки до третього розділу

В третьому розділі було розроблено додаток і розміщено його у хмарній платформі Azure за допомогою таких сервісів як App Service і Cosmos DB, виконано безперервну інтеграції і безперервну доставку, яка допомагає зменшити ризики появи багів і помилок у кодовій базі, покращує якість коду та збільшує швидкість доставки. Це також допомагає переконатися, що програма завжди актуальна та доступна для кінцевих користувачів.

У цьому розділі дістали подальшого розвитку механізми відновлення працездатності та надійності роботи застосунку при виникненні випадків простою та збоїв, за рахунок оптимізації налаштувань сервісів Azure, що ефективно використовується для підтримки операційних процесів. Це дозволяє максимально знизити вплив негативних ситуацій на продуктивність системи та забезпечити безперебійну роботу застосунку. Інноваційні механізми відновлення працездатності та надійності, що впроваджені, дозволяють автоматизувати процеси відновлення після виникнення простою чи збою, реагуючи на події в реальному часі та миттєво адаптуючи конфігурації для забезпечення оптимальної продуктивності. Оптимізація налаштувань сервісів Azure сприяє забезпеченню ефективного використання ресурсів та зменшенню часу відновлення послуги, що в свою чергу підвищує загальну ефективність та конкурентоспроможність розглянутого застосунку.

Важливим елементом роботи була розробка та тестування механізмів відновлення після збоїв, включаючи автоматизовані процеси та засоби моніторингу.

Окрім цього, було налаштовано моніторинг і логування, проведено комплексній аналіз відмовостійкості інфраструктури, намічені подальші шляхи підвищення надійності та стійкості до збоїв.

ВИСНОВКИ

В рамках кваліфікаційної роботи було проведено комплексне дослідження та аналіз відмовостійкого середовища на базі хмарної платформи Azure. В роботі було розглянуто та проаналізовано основні принципи та підходи до побудови відмовостійких архітектур за допомогою послуг Azure.

У процесі дослідження було досліджено та проаналізовано основні компоненти та послуги Azure, які використовуються для побудови відмовостійких систем. Було описано основні методики та стратегії резервного копіювання, реплікації та відновлення даних з використанням Azure Backup.

Також було досліджено та проаналізовано можливості автоматичного масштабування та балансування навантаження в середовищі Azure, що дозволяє реагувати на зміну обсягу роботи та забезпечувати стабільну працездатність системи навіть в умовах великого навантаження.

У процесі експериментального дослідження було побудовано та перевірено відмовостійкі архітектури на базі Azure для різних сценаріїв навантаження та відмов. Були проведені тести на відновлення системи та перевірено її стійкість до відмов. Отримані результати свідчать про ефективність та надійність використання Azure у побудові відмовостійких середовищ.

Новизна запропонованих рішень полягає в тому, що дістали подальшого розвитку інноваційні механізми відновлення працездатності та надійності роботи застосунку при виникненні випадків простою та збоїв, за рахунок оптимізації налаштувань сервісів Azure, що ефективно використовується для підтримки операційних процесів. Це дозволяє максимально знизити вплив негативних ситуацій на продуктивність системи та забезпечити безперебійну роботу застосунку. Інноваційні механізми відновлення працездатності та надійності, що впроваджені, дозволяють автоматизувати процеси відновлення після виникнення простою чи збою, реагуючи на події в реальному часі та миттєво адаптуючи конфігурації для забезпечення оптимальної продуктивності. Оптимізація налаштувань сервісів Azure сприяє забезпеченню ефективного

використання ресурсів та зменшенню часу відновлення послуги, що в свою чергу підвищує загальну ефективність та конкурентоспроможність розглянутого застосунку.

На основі проведеного дослідження можна зробити висновок, що Azure є потужним і надійним інструментом для побудови відмовостійких систем. Він надає велику кількість гнучких та масштабованих послуг, які дозволяють забезпечити високу надійність та продуктивність системи навіть в умовах великого навантаження. Використання Azure дозволяє знизити вартість розробки та управління відмовостійкими середовищами, а також забезпечити швидке та ефективне відновлення системи у разі відмови.

Отже, розроблені архітектури та проведене експериментальне дослідження підтверджують можливості та переваги використання хмарної платформи Azure для побудови відмовостійких середовищ. Результати дослідження можуть бути використані для подальшого розроблення та вдосконалення відмовостійких систем на базі Azure.

Виходячи з проведеного аналізу, можна стверджувати, що з очікуваним продовженням зростання хмарних обчислень підприємства все більше переходитимуть до хмари. Це створить підвищений попит на стійкі до відмови рішення, оскільки бізнес-критичні додатки залежатимуть від надійності хмарних інфраструктур.

Також набудуть подальшого розвитку технології автоматизації, включаючи DevOps-практики, контейнеризацію та управління інфраструктурою як код (IaaC), що сприятиме більш ефективної реалізації стійких до відмови рішень на платформі Azure.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке хмарні технології та їхня користь для бізнесу. [Електронний ресурс] – Режим доступу: https://vps.ua/blog/ukr/cloud-technologies-for-business/ (дата звернення 15.10.2023).
2. Azure Developer Documentation [Electronic resource] – Режим доступу: https://learn.microsoft.com/en-us/azure/developer/ (дата звернення 15.10.2023).
3. Azure Storage documentation [Electronic resource] – Режим доступу: https://learn.microsoft.com/en-us/azure/storage/
4. Azure Cosmos DB documentation [Electronic resource] – Режим доступу: https://learn.microsoft.com/en-us/azure/cosmos-db/ (дата звернення 15.10.2023).
5. App Service Documentation [Electronic resource] – Режим доступу: https://learn.microsoft.com/en-us/azure/app-service/overview (дата звернення 15.10.2023).
6. App Service Plan Documentation [Electronic resource] – Режим доступу: https://learn.microsoft.com/en-us/azure/app-service/overview-hosting-plans (дата звернення 15.10.2023).
7. Greg Lim, Beginning MERN Stack: Build and Deploy a Full Stack MongoDB, Express, React, Node.js App./ G.Lim – San Francisco: No Starch Press, 2021. - 159p.
8. Greg Lim, Beginning Node.js, Express & MongoDB Development./ G.Lim – San Francisco: No Starch Press, 2019, 154 p.
9. Kevin Hoffman, Beyond the Twelve-Factor App, - Kiev: O'Reilly Media, Inc. 2016, 180p.
10. Judith S. Hurwitz, Cloud Computing For Dummies 2nd Edition./ Daniel Kirsch - Boston: Addison-Welly, 2020, 320p.
11. Timothy L. Warner, Microsoft Azure For Dummies 1st Edition 2020, 368p.
12. John Savill, Microsoft Azure Infrastructure Services for Architects: Designing Cloud Solutions 1st Edition, 2019, 448p.
13. Jack A. Hyman, Microsoft Azure For Dummies 2nd Edition, 2023, 328p.

14. Ethan Brown, Web Development with Node and Express: Leveraging the JavaScript Stack, 2019, 355p.
15. Tim Oxley, Node.js in Action 2nd Edition, Nathan Rajlich, TJ Holowaychuk , Alex Young, 2017, 215p.
16. Thomas Hunter, Distributed Systems with Node.js: Building Enterprise-Ready Backend Services 1st Edition, 2020, 219p.
17. Nishant Singh, Cloud Native Infrastructure with Azure: Building and Managing Cloud Native Applications, Michael Kehoe, 2022, 460p.
18. Henry van Merode, Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines, - New York: Apress, 2023, 423p.

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

```
// src/config/db.ts
export default function connectDB() {
  const { MONGO_URI } = process.env;

  if (!MONGO_URI) {
    console.log("Please provide DataBase URI to connect. exiting now...");
    process.exit(1);
  }

  try {
    mongoose.connect(MONGO_URI);
  } catch (err: any) {
    console.error(err.message);
    process.exit(1);
  }
  const dbConnection = mongoose.connection;
  dbConnection.once("open", (_) => {
    console.log(`Database connected: ${MONGO_URI}`);
  });

  dbConnection.on("error", (err) => {
    console.error(`connection error: ${err}`);
  });

  return;
}

// src/controllers/auth.controller.ts
import Router from 'express-promise-router';
import bcrypt from "bcryptjs";
import * as jwt from "jsonwebtoken";
import { createUser, getUserByEmail } from '../models/users.repository';

export const router = Router();

router.post('/register', async (req, res) => {
    try {
        const { first_name, last_name, email, password, isAdmin } =
req.body;

        if (!(email && password && first_name && last_name)) {
            res.status(400).send("All input is required");
        }

        const oldUser = await getUserByEmail(email);

        if (oldUser) {
            return res.status(409).send("User Already Exist. Please
Login");
        }

        const encryptedPassword = await bcrypt.hash(password, 10);

        await createUser({
            first_name,
            last_name,
            email: email.toLowerCase(),
```

```
            password: encryptedPassword,
            role: isAdmin === "true" ? "admin" : "user"
        });

        res.status(201).send("User successfully registered");
    } catch (err) {
        console.error(err);
        res.status(500).send("Internal Server Error");
    }
});

router.post('/login', async (req, res) => {
    try {
        const { email, password } = req.body;

        if (!(email && password)) {
            res.status(400).send("All input is required");
        }

        const user = await getUserByEmail(email);

        if (user && (await bcrypt.compare(password, user.password))) {
            const token = jwt.sign(
                { user_id: user._id, email, role: user.role },
                process.env.TOKEN_KEY!,
                {
                    expiresIn: "2h",
                }
            );

            return res.status(200).json({ token });
        }
        res.status(400).send("Invalid Credentials");
    } catch (err) {
        console.log(err);
        res.status(500).send("Internal Server Error");
    }
});

// src/controllers/cart.controller.ts
import Router from 'express-promise-router';
import { deleteCart } from '../models/carts.repository';
import { getUserCart, updateUserCart } from '../services/cart.service';
import { createUserOrder } from '../services/order.service';
import { isAdmin } from '../middleware';

export const router = Router();

router.get('/', async (req, res) => {
    const userId = req.user.user_id;
    const cart = await getUserCart(userId as string);
    res.send({ data: { cart }, error: null });
});

router.put('/', async (req, res) => {
    const userId = req.user.user_id;
    const data = req.body;
    const cart = await updateUserCart(userId as string, data);
    res.send({ data: { cart }, error: null });
});

router.delete('/', isAdmin, async (req, res) => {
    const userId = req.user.user_id;
    const data = await deleteCart(userId as string);
```

```
        res.send({ data, error: null });
    });

    router.post('/checkout', async (req, res) => {
        const userId = req.user.user_id;
        const order = await createUserOrder(userId as string);
        res.send({ data: { order }, error: null });
    });

    export const CartController = router;

    // src/controllers/product.controller.ts
    import Router from 'express-promise-router';
    import { getProductById, getProducts } from
'../models/products.repository';

    export const router = Router();

    router.get('/', async (req, res) => {
        const products = await getProducts();
        res.send({ data: products, error: null });
    });

    router.get('/:productId', async (req, res) => {
        const id = req.params.productId;
        const product = await getProductById(id);
        res.send({ data: product, error: null });
    });

    export const ProductController = router;

    // src/entities/cart.ts
    import mongoose, { Schema, Types } from "mongoose";
    import { IProduct, Product } from "./product";

    export interface CartItemEntity {
        product: IProduct;
        count: number;
    }

    export interface ICart {
        _id: Types.ObjectId;
        isDeleted: boolean;
        user: Types.ObjectId;
        items: CartItemEntity[];
        _doc: {
            isDeleted: boolean;
            user: Types.ObjectId;
            items: CartItemEntity[];
        };
    }

    const cartSchema = new Schema<ICart>({
        isDeleted: {
            type: Boolean,
            required: true
        },
        user: {
            type: Schema.Types.ObjectId,
            ref: 'User'
        },
        items: [{
            product: {
                    type: Schema.Types.ObjectId,
```

```
                ref: Product
            },
        count: {
            type: Number,
            required: true,
            min: 1
        }
    }],
})

export const Cart = mongoose.model<ICart>("Cart", cartSchema);

// src/entities/user.ts
import { Types, Schema, model  } from "mongoose";


export interface IUser {
    _id?: string,
    first_name: string,
    last_name: string,
    email: string,
    password: string,
    role: string
}

export const userSchema = new Schema<IUser>({
    first_name: { type: String, default: null },
    last_name: { type: String, default: null },
    email: { type: String, unique: true, required: true },
    password: { type: String, required: true },
    role: {type: String}
});

export const User = model('User', userSchema);

// src/entities/order.ts
import mongoose, { Schema, Types } from "mongoose";
import { CartItemEntity } from "./cart";

type ORDER_STATUS = 'created' | 'completed';

export interface IOrder {
    user: Types.ObjectId;
    cart: Types.ObjectId;
    items: CartItemEntity[];
    payment: {
        type: string,
        address?: any,
        creditCard?: any,
    };
    delivery: {
        type: string,
        address: any,
    };
    comments?: string;
    status: ORDER_STATUS;
    total: number;
}

const orderSchema = new Schema({
    user: {
        type: Schema.Types.ObjectId,
        ref: 'User'
    },
```

```
    cart: {
        type: Schema.Types.ObjectId,
        ref: 'Cart'
    },
    items: [{
        product: {
            title: {
                type: String,
                required: true,
            },
            description: {
                type: String,
                required: true,
            },
            price: {
                type: Number,
                required: true,
                min: 0,
            },
        },
        count: {
            type: Number,
            required: true,
            min: 0,
        }
    }],
    payment: {
        type: {
            type: String,
            required: true,
        },
        address: {
            type: String,
        },
        creditCard: {
            type: String,
        },
    },
    delivery: {
        type: {
            type: String,
            required: true,
        },
        address: {
            type: String,
            required: true,
        },
    },
    comments: {type: String},
    status: {type: String, required: true,},
    total: {
        type: Number,
        required: true,
        min: 0,
    },
})

// src/errors/index.ts
export enum Errors {
    BAD_REQUEST = 'BadRequestError',
    AUTH = 'AuthError',
    FORBIDDEN = 'ForbiddenError',
    NOT_FOUND = 'NotFoundError',
    INTERNAL_SERVER = 'InternalServerError',
```

```
}

export class HttpError extends Error {
    statusCode: number;
    constructor(message: string) {
        super(message);
        this.statusCode = 400;
    }
}

export class BadRequestError extends HttpError {
    constructor(message: string) {
        super(message);
        this.name = Errors.BAD_REQUEST;
        this.statusCode = 400;
    }
}

export class AuthError extends HttpError {
    constructor(message: string = 'User is not authorized') {
        super(message);
        this.name = Errors.AUTH;
        this.statusCode = 401;
    }
}

export class ForbiddenError extends HttpError {
    constructor(message: string = 'You must be authorized user') {
        super(message);
        this.name = Errors.FORBIDDEN;
        this.statusCode = 403;
    }
}

export class NotFoundError extends HttpError {
    constructor(message: string = 'Not Found') {
        super(message);
        this.name = Errors.NOT_FOUND;
        this.statusCode = 404;
    }
}

export class InternalServerError extends HttpError {
    constructor(message: string = 'Internal Server error') {
        super(message);
        this.name = Errors.INTERNAL_SERVER;
        this.statusCode = 500;
    }
}

// src/middleware/auth.ts
import { NextFunction, Request, Response } from "express";
import * as jwt from "jsonwebtoken";

export interface CurrentUser {
    user_id: string,
    email: string,
    role: string
}

export async function auth(req: Request, res: Response, next: NextFunction)
{
    const authHeader = req.headers.authorization;
```

```
        if (!authHeader) {
            return res.status(401).send("Token is required");
        }

        const [tokenType, token] = authHeader.split(' ');

        if (tokenType !== 'Bearer') {
            return res.status(403).send("Invalid Token");
        }

        try {
            const user = jwt.verify(token, process.env.TOKEN_KEY!) as
CurrentUser;

            req.user = user;
        } catch (err) {
            return res.status(401).send("Invalid Token");
        }
        return next();
    }

    // src/middleware/error.ts
    import { NextFunction, Request, Response } from "express";
    import { HttpError } from "../errors";

    export const errorHandler = (err: HttpError, req: Request, res: Response,
next: NextFunction) =>
        res.status(err.statusCode).json({ data: null, message: err.message });

    // src/middleware/isAdmin.ts
    import { Request, Response, NextFunction } from "express";
    import { ForbiddenError } from "../errors";

    export function isAdmin(req: Request, res: Response, next:NextFunction) {
        const currentUser = req.user;

        if (currentUser.role !== 'admin') {
            throw new ForbiddenError('Only admins can delete carts!');
        }
        next();
    }

    // src/models/carts.repository.ts

    import { InternalServerError, NotFoundError } from '../errors';
    import { ICart } from '../entities/cart';
    import { Cart } from '../entities/cart';

    export function transformCartData(cart: ICart) {
        const items = [];
        let total = 0;
        for (const item of cart.items) {
             const product = item.product;
            items.push({ product, count: item.count });
            total += product.price * item.count;
        }

        return { ...cart._doc, items, total };
    }

    export async function getCart(userId: string) {
        let cart;
```

```
        cart     =     await     Cart.findOne({user:     userId,     isDeleted:
false}).populate({path: 'items', populate: 'product'});
        if (!cart) {
           cart = await Cart.create({isDeleted: false, user: userId});
        }

        return cart;
    }

    export async function updateCart(userId: string, { product, count }: any) {
        let cart;

        try {
            cart = await Cart.findOne({user: userId, isDeleted: false, items:
{$elemMatch: {product}}})
            if (cart) {
                count ?
                await   cart.updateOne({$set:   {"items.$[t].count":   count}},
{arrayFilters: [{"t.product": product.id}, ]})
                :
                await cart.updateOne({$pull: {items: {product} }})
            } else {
                if (count) {
                    cart   =   await   Cart.findOneAndUpdate({user:   userId,
isDeleted: false},
                        {$push: {items: {product: product.id, count}}});
                }
            }
        } catch (err: any) {
            throw new InternalServerError(err.message);
        }

        return getCart(userId);
    }

    export async function deleteCart(userId: string) {
        const cart = await Cart.findOne({user: userId, isDeleted: false});
        if (!cart) {
            throw new NotFoundError("Cart was NOT found!");
        }

        await cart.updateOne({$set: {isDeleted: true}});

        return { success: true }
    }

    // src/models/users.repository.ts
    import { IUser, User } from '../entities/user';

    export async function getUserByEmail(email: string): Promise<IUser | null>
{
        return await User.findOne({email});
    }

    export async function createUser(user: IUser) {
        return await User.create(user);
    }

    // src/models/products.repository.ts
    import  {Product } from '../entities/product';

    export async function getProducts() {
        return await Product.find({});
    }
```

```
export async function getProductById(productId: string) {
    return await Product.findById(productId);
}

// src/seeders/database.seeder.ts
import mongoose from 'mongoose';
import connectDB from '../config/db';
import { Product } from '../entities/product';
import { User } from '../entities/user';
import { Cart } from '../entities/cart';

const products = [
    {title: 'test', description: 'Lorem Ipsum ...', price: 4},
    {title: 'Qwerty', description: 'Qsvsddbdb gbfgnfn', price: 9}
];

connectDB();

const seedDB = async () => {
    await Product.deleteMany({});
    await Product.insertMany(products);

    await User.deleteMany({})
    const user = new User();
    await user.save();

    await Cart.deleteMany({});
}

seedDB().then(() => mongoose.connection.close());

// src/services/cart.service.ts
import Joi from 'joi';
import { getCart, transformCartData, updateCart } from
"../models/carts.repository";
import { BadRequestError, InternalServerError, NotFoundError } from
'../errors';
import { getProductById } from '../models/products.repository';

const schema = Joi.object({
    productId: Joi.string().required(),
    count: Joi.number().integer().required()
});

export async function getUserCart(userId: string) {
    let cart;
    try {
        cart = await getCart(userId);
    } catch {
        throw new InternalServerError();
    }

    return transformCartData(cart);
}

export async function updateUserCart(userId: string, data: { productId:
string; count: number }) {
    const { error, value } = schema.validate(data);

    if (error) {
        const message = error.details.map(err => err.message).join(', ')
        throw new BadRequestError(message)
    }
```

```
        const { productId, count} = data;

        let product;
        try {
            product = await getProductById(productId);
        } catch {
            throw new NotFoundError(`Product with id ${productId} was NOT
found!`);
        }

        const cart = await updateCart(userId, { product, count });

        return transformCartData(cart);
    }

    // src/services/order.service.ts
    import { IOrder } from "../entities/order";
    import { BadRequestError } from "../errors";
    import { deleteCart, getCart, transformCartData } from
"../models/carts.repository";
    import { createOrder } from "../models/order.repository";

    const orderDetails: Omit<IOrder, 'user' | 'cart' | 'items' | 'total'> = {
        payment: {
            type: 'paypal',
            address: 'London',
            creditCard: '1234-1234-1234-1234'
        },
        delivery: {
            type: 'post',
            address: 'London'
        },
        comments: '',
        status: 'created'
    }

    export async function createUserOrder(userId: string) {
        const cart = await getCart(userId);
        const { items, total } = transformCartData(cart);

        if (!total) {
            throw new BadRequestError('Cart is empty');
        }

        const order = await createOrder({ user: cart.user, cart: cart._id,
items, total, ...orderDetails });
        deleteCart(userId);

        return order;
    }

    // src/server.ts
    import * as dotenv from 'dotenv';
    import express from 'express';

    import connectDB from './config/db';
    import { auth, errorHandler, CurrentUser } from './middleware';
    import { AuthController, CartController, ProductController } from
'./controllers';

    declare global {
        namespace Express {
            interface Request {
```

```
                user: CurrentUser
            }
        }
    }

    dotenv.config();

    export const app = express();
    const { API_PORT } = process.env;
    const port = API_PORT || 3001;

    connectDB();

    app.use(express.json());

    app.use('/', AuthController);
    app.use('/api', auth);

    app.get('/', (req, res) => res.json({ message: 'Welcome to Nodejs express
TS app!' }));
    app.use('/api/profile/cart', CartController);
    app.use('/api/product', ProductController);
    app.use('/api', errorHandler);

    const server = app.listen(port, () => {
        console.log(`Server running at http://localhost:${port}`);
    });

    // Graceful shutdown
    let connections: any[] = [];

    server.on('connection', (connection) => {
      // register connections
      connections.push(connection);

      // remove/filter closed connections
      connection.on('close', () => {
        connections      =      connections.filter((currentConnection)      =>
currentConnection !== connection);
      });
    });

    function shutdown() {
      console.log('Received kill signal, shutting down gracefully');

      server.close(() => {
        console.log('Closed out remaining connections');
        process.exit(0);
      });

      setTimeout(() => {
        console.error('Could not close connections in time, forcefully shutting
down');
        process.exit(1);
      }, 20000);

      // end current connections
      connections.forEach((connection) => connection.end());

      // then destroy connections
      setTimeout(() => {
        connections.forEach((connection) => connection.destroy());
      }, 10000);
    }
```

```
process.on('SIGTERM', shutdown);
process.on('SIGINT', shutdown);

// .env
export API_PORT=3001

export MONGO_URI=mongodb://127.0.0.1/node-gmp-db

TOKEN_KEY=someverysecterstring
```

ДОДАТОК Б

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Спірінцева В'ячеслава Васильовича, доц. каф. ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

На кваліфікаційну роботу
студента Соколянського Дмитра Васильовича
(прізвище, ім'я, по батькові)

курсу II групи 122м-22з-1
спеціальності 122 Комп'ютерні науки
на тему Дослідження відмовостійкого середовища
на базі хмарної платформи Azure

Актуальність теми Представлена магістерська кваліфікаційна робота присвячена дослідженню відмовостійкого середовища на базі хмарної платформи Azure. З очікуваним продовженням зростання хмарних обчислень підприємства все більше переходитимуть до хмари. Це створить підвищений попит на стійкі до відмови рішення, оскільки бізнес-критичні додатки залежать від надійності хмарних інфраструктур. Таким чином дослідження щодо відмовостійкості стають ключовими для забезпечення безперервності бізнес-процесів.

Мета досліджень Полягає в підвищенні ефективності хмарної платформи Azure на основі розробки та дослідження середовища, яке буде максимально безвідмовне і незалежне від зовнішніх чинників.

Коротка характеристика розділів роботи Перший розділ роботи присвячений огляду проблеми, що вирішується в ході виконання роботи, а також представлені можливі шляхи вирішення цієї проблеми. Другий розділ містить аналітичний огляд на архітектуру, композицію та методологію додатку, що

розробляється. В третьому розділі був розроблений додаток і розміщений у хмарній платформі Azure, для подальшого аналізу відмовостійкості.

Практичне значення роботи полягає в підвищенні надійності програми та зниженні ймовірності простоїв, що важливо для забезпечення стабільності бізнес-процесів та задоволення потреб користувачів, оптимізації використання обчислювальних ресурсів завдяки автоматизованому масштабуванню та управлінню ресурсами в залежності від навантаження, що заощаджує витрати на IT-інфраструктуру.

Зауваження та недоліки недостатньо глибокий аналіз сервісів хмарної платформи, їх функціонального навантаження у розрізі розв'язуваних завдань. Однак це жодною мірою не впливає на загальну схвальну оцінку роботи.

Висновки та оцінка Магістром було проведено аналіз та порівняння можливих методів розв’язання поставленої задачі та обрано оптимальний варіант. Під час виконання магістерської кваліфікаційної роботи студент Соколянський Д.В. проявив себе грамотним, кваліфікованим спеціалістом здатним приймати самостійно складні технічні рішення.
Вважаю, що магістерська кваліфікаційна робота заслуговує оцінку 90 «відміно», а Соколянський Д.В. – присвоєння кваліфікації «магістра» з комп’ютерних наук.

Науковий керівник Спірінцев В.В., ктн, доц. каф. ПЗКС
(прізвище, ім’я, по батькові, посада, місце роботи)

«___»_____20__ р. _____
(підпис)

ДОДАТОК В

РЕЦЕНЗІЯ
на кваліфікаційну роботу

студента Соколянського Дмитра Васильовича
(прізвище, ім'я, по батькові)

курсу II групи 122м-22з-1
кафедри програмного забезпечення комп'ютерних систем
спеціальності 122 Комп'ютерні науки
Тема роботи Дослідження відмовостійкого середовища на базі хмарної платформи Azure

Стисла характеристика розділів роботи Перший розділ роботи присвячений огляду проблеми, що вирішується в ході виконання роботи, а також представлені можливі шляхи вирішення цієї проблеми. Другий розділ містить аналітичний огляд на архітектуру, композицію та методологію додатку, що розробляється. В третьому розділі був розроблений додаток і розміщений у хмарній платформі Azure, для подальшого аналізу відмовостійкості

Пропозиції, внесені студентом, рівень їх наукового обґрунтування В даній кваліфікаційній роботі студентом надано декілька пропозицій щодо вирішення поставлених задач. Кожна з пропозицій була обґрунтована та підкріплена науковими даними

Практичне значення роботи полягає в підвищенні надійності програми та зниженні ймовірності простоїв, що важливо для забезпечення стабільності бізнес-процесів та задоволення потреб користувачів, оптимізації використання обчислювальних ресурсів завдяки автоматизованому масштабуванню та управлінню ресурсами в залежності від навантаження, що заощаджує витрати на IT-інфраструктуру.

Недоліки в роботі недостатньо глибокий аналіз проблематики та її впливу на подальшу міграцію бізнес-критичних застосунків у хмару. Проте вказаний недолік не впливає на позитивне враження від роботи.

Загальний висновок Магістерська кваліфікаційна робота виконана у
(підготовленість студента до самостійної роботи як спеціаліста)
відповідності з завданням із дотриманням всіх вимог.
Оцінка магістерської роботи Робота заслуговує оцінки «відміно», а студент Соколянський Д.В. – присвоєння кваліфікації «магістра» з комп'ютерних наук.

Рецензент _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»_____20__ р. _____
(підпис)

ДОДАТОК Г

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Соколянський.docx	Пояснювальна записка роботи. Документ Word.
Диплом_ Соколянський.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми
Презентація	
Презентація_Соколянський.ppt	Презентація роботи