Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук і штучного інтелекту
Кафедра інтелектуальних програмних систем і технологій

Рекомендовано до захисту:
протокол засідання кафедри
№ ___ від ____.____.2025 р.
Завідувач кафедри к. т. н
Олешко О.І.
(підпис)

Пояснювальна записка
до кваліфікаційної роботи бакалавра
на тему "Методи аналізу параметрів шаблонів стійкості для підвищення доступності мікросервісів при високому навантаженні"

Захищено на засіданні ЕК № _____
протокол № ____ від ____.____.2025 р.
Оцінка _____ / _____
Голова ЕК

_____ _____
(підпис) (прізвище та ініціали)

Виконав:
студент 4 курсу, групи КС-41
Спеціальності:
122 Комп'ютерні науки
Кафтанатій Іван Сергійович
Керівник доцент, кандидат технічних наук Гамзаєв Р. О.
(підпис)
Рецензент кандидат технічних наук Марчук А.В.
(підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра інтелектуальних програмних систем і технологій
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Спеціальність F3 Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

"__11__" ___січня___2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ
Кафтанатій Іван Сергійович
(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Методи аналізу параметрів шаблонів стійкості для підвищення доступності мікросервісів при високому навантаженні»

керівник роботи Гамзаєв Рустам Олександрович, кандидат технічних наук, доцент кафедри інтелектуальних програмних систем і технологій
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "16" квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи_____

3. Перелік питань, які потрібно розробити:

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).

2	Пошук та аналіз інформаційних джерел за темою КР.
3	Дослідити та класифікувати основні шаблони стійкості (resilience patterns), такі як: Circuit Breaker, Retry, Bulkhead, Timeout, Rate Limiter.
4	Визначити ключові параметри налаштування зазначених шаблонів та їх вплив на стабільність системи.
5	Реалізувати прототип системи, в якій можна варіювати параметри шаблонів стійкості та фіксувати їхній вплив на доступність сервісів.
6	Провести серію експериментів із використанням інструментів навантажувального тестування (наприклад, Gatling, k6, JMeter). Проаналізувати результати тестування, зокрема в контексті часу відгуку, кількості помилок, кількості відмов. Надати рекомендації щодо оптимальних налаштувань шаблонів стійкості залежно від типу навантаження.
7	Підсумок отриманих результатів, проведення тестування, аналіз отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ШІ.

5. Дата видачі завдання 11 січня 2025

Студент _ __ І. О. Кафтанатій
(підпис) (ініціали, прізвище)

Керівник роботи _____ Р. О. Гамзаєв
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Дана робота складається з 52 сторінок, містить 14 ілюстрацій, 1 таблиці і складається з трьох розділів. В даній роботі використано 17 джерел та посилань. Метою даної роботи є аналіз параметрів шаблонів стійкості та вивчення методології їх аналізу, дослідження впливу даних патернів і їх параметрів на доступність сервісів при їх динамічному налаштуванні. У роботі розглянуто найбільш популярні патерни стійкості, розроблено і впроваджено методологію динамічного налаштування, розглянуто теоретичну і прикладну сторони проблеми, зроблено висновки щодо розглянутих аспектів проблеми. Для дослідження було створено застосунок мовою Java з використанням бібліотеки Resilience4j. Дана робота несе в собі наукову новизну, оскільки надає комплексний аналіз найбільш використовуваних патернів і впливу їх динамічно налаштовуваних параметрів на стійкість і доступність мікросервісів. Результати даної роботи можуть бути використані для впровадження динамічного налаштування шаблонів стійкості в прикладних застосунках відповідно до вказаних в роботі висновків. Значущість роботи полягає в систематизації знань щодо впливу динамічності налаштування параметрів патернів і порівняльному аналізі. Подальші дослідження можуть включати дослідження менш популярних патернів і ще ширшого вивчення параметрів.

Ключові слова: ШАБЛОН СТІЙКОСТІ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ДИНАМІЧНЕ НАЛАШТУВАННЯ ПАРАМЕТРІВ

ABSTRACT

This work consists of 52 pages, contains 14 illustrations, 1 table and consists of three sections. This work uses 17 sources and references. The purpose of this work is to analyze the parameters of resilience patterns and study the methodology for their analysis, to study the impact of these patterns and their parameters on the availability of services during their dynamic configuration. The work considers the most popular resilience patterns, develops and implements a dynamic configuration methodology, and considers the theoretical and applied aspects of the problem, draws conclusions on the aspects of the problem considered. For the study, an application was created in Java using the Resilience4j library. This work is scientifically novel, as it provides a comprehensive analysis of the most commonly used patterns and the impact of their dynamically configured parameters on the stability and availability of microservices. The results of this work can be used to implement dynamic configuration of resilience patterns in applications in accordance with the conclusions specified in the work. The significance of the work lies in the systematization of knowledge about the impact of dynamic parameter tuning of patterns and comparative analysis. Further research may include the study of less popular patterns and an even broader study of parameters.

Keywords: RESILIENCE PATTERN, MICROSERVICE ARCHITECTURE, DYNAMIC PARAMETER TUNING

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

Шаблони стійкості – те ж, що і патерни стабільності, струкутурні рішення, що допомагають мікросервісам ефективно реагувати на відмови сервісів, зберігаючи доступність основних сервісів і їх функціональність.

БД – база даних, сукупність даних, організованих відповідно до концепції, яка описує характеристику цих даних і взаємозв'язки між їх елементами.

Downtime – час, у який сервіс недоступний, виведений з ладу.

Кінцева точка (endpoint) – веб-частина застосунку, що відповідає за прийняття чи надання інформації як відповідь на запити.

ВСТУП

Монолітна архітектура застосунків, популярна в минулому, все більше стає рідкістю через використання розподілених систем і їх комунікацію через мережу. На заміну їй все частіше розробники використовують мікросервіну архітектуру, що забезпечує значну масштабованість та незалежність сервісів. На відміну від монолітної, мікросервісна архітектура дозволяє кожному сервісу виконувати свою окрему бізнес-функцію, відповідно і відмова одного сервісу не одразу призведе до збою в системі в цілому. Таке архітектурне рішення не тільки добре імплементується в розподілені системи, а й дозволяє пришвидшити розробку застосунку завдяки розподіленню відповідальності команд, що будуть розробляти окремі сервіси автономно від інших учасників розробки, реалізуючи загальний план розробки.

Попри ці переваги, перед розробниками мікросервісних архітектур постає низка викликів, серед яких найбільшої актуальності набуває питання забезпечення стабільності таких систем і обробки нестандартних ситуацій (збоїв), зокрема за великого навантаження. Як і в монолітній архітектурі, у застосунку іноді виникають відмови та помилки: перевищення затримок очікування відповіді через затримки в мережі, надмірне навантаження на апаратну частину, що призводить до перезавантаження, помилки в коді і т.д. З розвитком мікросервісної архітектури виявилося, що відмови окремих сервісів можуть призвести до каскадної відмови застосунку, тобто втрачається перевага ізольованості застосунку. До того ж надмірне очікування, що не призводить до коректної відповіді застосунку, негативно впливає на продуктивність системи, бо займає її ресурси. Для забезпечення стійкості застосунку і зменшення таких ризиків і були розроблені шаблони стійкості, які дозволяють керувати відмовами і забезпечувати неперервність роботи системи.

На сьогодні існує кілька основних патернів стійкості, серед яких Circuit Breaker, Retry, Time Limiter, Bulkhead, Rate Limiter , Fallback та інші. Кожен із них має свої особливості та застосовується в різних сценаріях. Однак наявність

декількох різних патернів ставить питання щодо доцільності та ефективності їх використання в конкретних умовах. Окремо також постає питання щодо правильності налаштування пантеонів і зміні їх параметрів динамічно. Вибір оптимального патерну і його параметрів залежить від характеру навантаження, типу сервісів та вимог до продуктивності.

Варто відмітити, що дана тема є доволі великою і дослідити усі аспекти без наявного досвіду роботи з цими патернами в реальних застосунках буде доволі складно, що може призвести до неповних висновків. Хоча коло досліджень навколо мікросервісної архітектури і шаблонів стійкості зокрема є доволі суттєвим, аналогічні чи близькі за змістом дослідження зустрічаються дуже рідко і зазвичай містять лише дослідження одного з патернів, що зменшує кількість даних, на які можна було б спиратися.

Актуальність зумовлена використанням патернів стійкості в мікросервісній архітектурі і необхідністю систематизації знань про шаблони стійкості разом з аналізом їх ефективності при динамічній конфігурації в умовах високого навантаження.

Метою дослідження є дослідження впливу параметрів на доступність мікросервісів при високому навантаженні.

Об’єкт дослідження: методи забезпечення стійкості мікросервісних систем

Предмет дослідження: динамічне налаштування шаблонів стійкості.

Перед собою в даному дослідженні поставлено задачу з розробки та впровадження методології адаптивної конфігурації шаблонів стійкості в мікросервісній архітектурі

РОЗДІЛ 1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ДОСЛІДЖЕНЬ ПО ТЕМАТИЦІ РОБОТИ

1.1 Шаблони стійкості та особливості їх використання

Для поглибленого аналізу зазначеної теми було проведено дослідження літератури за суміжними темами. Зокрема було знайдено сучасні наукові роботи, що висвітлюють проблеми мікросервісної архітектури, впровадження шаблонів стійкості для мінімізації ризиків та динамічного їх налаштування. Стан дослідження даної області можна назвати задовільним, хоча здебільшого ці дослідження стосуються більш широкого спектру завдань і питань.

За останнє десятиліття активний розвиток мікросервісної архітектури, що обумовлений потребами масштабованості і гнучкості, призвів до появи спеціалізованих інструментів, спрямованих на спрощення розробки і підтримки таких систем. Разом з тим все частіше розробники стикаються з проблемами, що виникають під час використання даної архітектури. Так, згідно з [1], мікросервісні архітектури схильні до збоїв через мережеві, апаратні чи програмні проблеми, тому важливо впроваджувати механізми, які допомагають мінімізувати наслідки відмов. Це і залишається однією з ключових проблем забезпечення відмовостійкості і доступності сервісів. Варто зазначити, що природа цих збоїв може бути різноманітною і бути як на стороні сервісу, так і проміжних провайдерів: затримки в мережі, надмірне навантаження на сервер, вихід з ладу апаратного забезпечення, помилки в коді застосунку. Відтак і час на відновлення роботи застосунку, його «downtime» буде різним: іноді це просто відмова якогось елементу в мережі, що виправляється автоматично, а іноді це відмова сервісу як такого, в результаті чого потрібно втручання зі сторони розробників. Такий комплекс проблем вимагає впровадження стратегій, спрямованих на мінімізацію негативного впливу збоїв.

Говорячи про мікросервісну архітектуру, варто згадати основні роботи, що розкривають сутність даної теми, що важливо для розуміння подальшої

роботи. Зокрема в роботі [2] вказано один з ключових принципів розробки мікросервісної архітектури, який і надає її основну перевагу перед монолітною – декомпозиція, тобто розділення системи на сервіси з чіткими завданнями. Окрім того зазначаються також з інших джерел можна дізнатися, що зазвичай така архітектура виконує комунікацію через легші протоколи, характеризується незалежністю, гетерогенністю сервісів і технологій їх виконання. Попри це дослідники згадують також складність управління і комунікації між сервісами, збільшення затримок через потребу в комунікації, наявність безпекових проблем та інше. Вказані проблеми знову відсилають нас до потреби впровадження шаблонів стійкості та інших рішень. Можна сказати, що монолітна архітектура простіша на початку, при невеликому застосунку, проте з часом її складність збільшується. Варто згадати, що в роботах часто використовують Spring Framework, що має багато інструментів для реалізації мікросервісної архітектури, через що і було вирішено використати його.

Крім того варто згадати також про базові поняття, зокрема про роботу Майкла Ніґарда, аналіз якої надано в джерелі [11]. В даній роботі описані класичні патерни, зокерма Circuit Breaker, Retry, Bulkhead. В тому числі там згадано важливість динамічного налаштування параметрів, таких як час переходу в Circuit Breaker, задля адаптації до різних ситуацій у системі. Аналіз же проблематики мікросервісної та монолітної архітектур надано в багатьох роботах як вступний матеріал, зокрема в вітчизняних дослідженнях [3] та [17]. Зокрема в останній згадано основну причину популярності мікросервісної архітектури останнім часом: порівняно з монолітною, вона забезпечує з часом набагато більшу продуктивність і більшу масштабованість.

Продовжити детальніше заглиблення в тему хотілося б з огляду літератури про шаблони стійкості як такі. Насправді, існує велика кількість наукових та науково-популярних статей за мікросервісами, зокрема у великих IT-корпорацій, як Microsoft та Amazon, що дозволяють отримати загальну інформацію про той чи інший патерн, його основні характеристики і реалізації, зокрема з використанням популярних бібліотек. Наводити їх інформацію тут не

було б доцільно, оскільки сама сутність патернів буде розглянута в першому розділі з посиланнями на джерела. Тут лиш зазначимо, що в процесі дослідження було досліджено декілька статей для отримання загального уявлення про роботу цих патернів і реалізацію їх у бібліотеці Resilience4j.

Наведемо також приклад порівняльного аналізу даних архітектур. Такий проведено у кваліфікаційній роботі [3] за різними показниками, серед яких толерантність до відмов і масштабованість. Очікувано, застосування шаблонів стійкості дозволяє досягти кращої відмовостійкості, однак, що важливо для даного дослідження, правильне налаштування цих шаблонів суттєво впливає на ефективність впровадженого рішення. Зокрема в роботі [3] описуються різні інструменти для тестування і моніторингу для проведення аналізу. Серед них є методи навантажувального тестування, моделювання відмов і аналіз метрик продуктивності. Окремо варто відмітити що дуже часто зустрічається стек технологій Spring, як найбільш популярне рішення для такої архітектури мовою Java. У роботах часто згадано бібліотеку Resilience4j, що містить реалізацію найпопулярніших шаблонів стійкості мовою Java з можливістю їх налаштування. У зв’язку зі зручністю та гнучкістю бібліотеки було вирішено використати її для програмної реалізації моделювання. Дана бібліотека в згаданій роботі визнана актуальною і зручною, відповідно написання власної реалізації визнано недоцільною.

Однак існують і більш наближені до теми роботи наукові досягнення, тобто такі, що висвітлюють не лише проблему масштабованості, а й виклики у вигляді адаптації систем до змінних умов експлуатації. У сучасних роботах все частіше згадуються автоматизація регулювання параметрів стійкості задля оптимізації балансування продуктивності, зменшення часу простою і так далі. Зокрема роботи [14] [9] розглянуто впровадження динамічного налаштування параметрів конкретних шаблонів стійкості і розглянуто вигоди від впровадження такого елементу системи як динамічний конфігуратор. Зокрема, в роботі [3] в процесі замірів вийшло визначити, що час простою завдяки адаптації до умов системи і навантаження зменшився вдвічі (з 54 секунд до 28),

що є суттєвим для системи. В роботі [9] кількість збоїв, порівняно з прогнозованою, також була суттєво нижче за прогнозовані відмови. Дослідження [14] показало, що покращення від запровадження динамічного налаштування моделей можуть сягати трикратного приросту, порівняно зі статичним. “Auto-tuned Rate-limiter”, запропонований в роботі [13], адаптується до навантаження за алгоритмом мультиплікативного зменшення/збільшення, що дозволяє автоматично регулювати ліміти запитів у межах вказаного коридору, що в результаті також призвело до трикратного зменшення часу відгуку з 3,6 до 1,2 секунд при 15 потоках. Як зазначено в роботі, при використанні такого патерну зз асинхронною комунікацією можливо зменшити ймовірність каскадних відмов до 50%. Отже, при зростанні кількості помилок через перевантаження, ліміт знижується за експоненційною функцією, що призводить до запобігання каскадній відмові застосунку. такий підхід зокрема можна використати не лише до відносно простих патернів, а й до більш складних. З цих робіт можна зробити висновок, що, як і очікувалося, впровадження динамічного налаштування шаблонів стійкості може надавати дуже суттєву перевагу в часі очікування, порівнювану навіть з перевагою від покращення апаратного забезпечення. Дані з цих робіт стануть нам в нагоді при налаштуванні власного алгоритму налаштування шаблонів стійкості.

Однак попри великі виграші від вказаних патернів стабільності варто не забувати, що необдуманий підхід чи неправильне налаштування у випадку зі статичною реалізацією може призвести до негативного впливу на систему, оскільки обмеження ресурсів чи запитів не завжди є доцільним чи виправданим. До прикладу, в роботі [12] проведено експерименти із комбінацією Retry та Circuit Breaker, результати яких показали, що при використанні шаблону Retry з 1 спробою та обмеженням черги 1 для Circuit Breaker час відгуку збільшується на 30% порівняно з випадком без Retry, що також є суттєвим погіршенням. В той же час, збільшення дозволених повторних спроб до 10 збільшує кількість успішних запитів до 15% при збільшенні навантаження на мережу. Виходячи з цього можна переконатися, що баланс між

обмеженням ресурсів задля стабільності і забезпеченням ефективності роботи системи є дуже важливим при застосуванні шаблонів, особливо при використанні комбінації двох патернів, де параметри в обох патернах можуть конфліктувати чи робити сервіс абсолютно недосяжним навіть за відсутності перевантаження. Разом з цим збільшується важливість тестування і вивчення найкращої комбінації параметрів, розроблення моделей для тестування навантаження і збору даних про наявні навантаження та аномалії в системі.

Окремо хочемо виділити математично моделювання у роботі [10], де проведено глибинне дослідження з використанням марковських ланцюгів. В ході даного дослідження дослідники вивчено шаблони Retry та Circuit Breaker, і на основі їх даних можна визначити, що для Retry оптимальна кількість спроб залежить від середнього часу між відмовами, тобто якщо час між відмовами 10 сек, то кількість спроб 3 забезпечить 99% успішних запитів, тобто досягнуто баланс у питанні забезпечення стійкості і швидкості відповіді. Для Circuit Breaker параметр допустимих відмов у 60% і час очікування в відкритому стані в 10 сек мінімізують час недоступності сервісу Ця робота є цікавою нам, оскільки використовує не стандартні метрики, як-от час відповіді, кількість запитів і відсоток відмов, але й час між відмовами і час відновлення, що розширює горизонт досліджень і дозволяє отримати свіжі ідеї для будування методології динамічного налаштування шаблонів стійкості.

Відзначимо також той факт, що вже з’являються комерційні рішення для застосування гнучких політик налаштування, зокрема в патерні Retry у сервісі Temporal, що вказує на збільшення зацікавленості з боку бізнес-структур до даної теми. При цьому варто враховувати також і той факт, що це лише відкрита інформація, а більшість фактів використання залишаться без уваги дослідників через політику захисту коду і нерозголошення даних про систему. У вказаній системі вказано про експоненційне збільшення інтервалу з коефіцієнтом відступу 2.

Окрім використання традиційних схем тестування на навантаження та моделювання відмов, новітні роботи пропонують застосування цифрових

двійників (digital twins) для симуляції експлуатаційних режимів мікросервісних систем. Цей підхід дозволяє детально вивчити поведінку системи у критичних ситуаціях та розробити стратегії оперативного відновлення після збоїв. Таке інтегроване моделювання сприяє не лише підвищенню технічної надійності, а й оптимізації економічних показників експлуатації, що є важливим фактором для впровадження масштабованих рішень у промислових умовах [10]. Однак використання таких систем нами розглядатися не буде через високу складність впровадження і невідповідність темі роботи.

Водночас варто зазначити, що впровадження такого рішення залишається досить складною через відсутність чітких методології визначення порогових значень, визначення методології налаштувань і реакції на аномальні стани. Аналіз сучасних досліджень вказує на необхідність розробки комплексних підходів, що враховують специфіку розподілених систем, в тому числі різницю у підході і вимогах до стабільності різних сервісів всередині однієї архітектури. Очевидно, що впровадження шаблонів стабільності в першу чергу стосуватиметься критичних розділів застосунку, які найчастіше піддаються навантаженням і відмова яких стане критичною для застосунку. Окрім того можливо використання комбінованого динамічно-статичного підходу для економії ресурсів, оскільки саме налаштування патернів також займає ресурси застосунку і може призвести до збоїв. Однак вигода від таких рішень все ще залишається високою, що підтверджується вказаними вище роботами.

Підсумовуючи інформацію, отриману з оглянутих джерел, можемо дійти висновку, що впровадження в мікросервісні архітектури систем з динамічним налаштувань параметрів шаблонів стійкості цілком ймовірно стане трендом в найбільші часи, якщо вже не стало таким. Зі зростанням популярності мікросервісної архітектури дедалі більше команд розробників стикатимуться з необхідністю забезпечити стійкість елементів системи, що приведе їх до впровадження того чи іншого шаблону стійкості і, зрештою, до динамічного їх налаштування з метою покращення ефективності роботи застосунку. З часом все більше досліджень будуть спрямовані на покращення методології такого

налаштування, розроблення бібліотек, розширюючих існуючі варіанти шаблонів стійкості, що відкриває нові горизонти для підвищення якості й надійності мікросервісних рішень.

Варто згадати також саму бібліотеку Resilience4j, що є досить популярною і була використана в даній роботі. Її було обрано саме за легкість і простоту впровадження. Вона містить усі патерни, які б хотілося дослідити, є модульною і дозволяє обирати різні патерни для використання і налаштування. В ході дослідження і використання її було використано офіційний репозиторій даної бібліотеки та на [6], як перевірене джерело навчального матеріалу неакадемічної спрямованості. Дана бібліотека була зокрема використана у вже згаданих роботах [10] [3]. Інші рішення або є застарілими або не варті сил, витрачених на їх вивчення, оскільки не відрізняються принципово від даної і не пропонують функціонал, що був би застосований в даній дослідницькій роботі.

1.2 Обґрунтування актуальності та доцільності дослідження

Виходячи з вищевказаного, можемо відмітити, що сучасні програмні системи з мікросервісною архітектурою розвиваються стрімко. Однак, із зростанням популярності даного підходу з'являються нові виклики, серед яких – забезпечення високої доступності та відмовостійкості сервісів у умовах високого навантаження. Відповідно, актуальність даного дослідження зумовлена необхідністю розробки методології аналізу параметрів шаблонів стійкості та впровадження адаптивної конфігурації для підвищення продуктивності мікросервісних застосунків, оцінки ефективності такого підходу. Отже, результати такої роботи будуть корисними і можуть бути використані в реальних програмних продуктах для оптимізації їх роботи. Це зумовлює доцільність даного дослідження і його прикладну цінність.

Висновок до розділу І

У першому розділі наведено загальний аналіз наявної академічної літератури за темою, що зокрема включає в себе загальну інформацію про

мікросервісну архітектуру, дані про шаблони стійкості, аналіз робіт з динамічної конфігурації шаблонів стійкості різного виду. Визначено, що мікросервісна архітектура є сучасним типом архітектури і активно розвивалася останнє десятиліття. Попри свої переваги у вигляді гнучкості, ізольованості і масштабованості, з даною архітектурою постають і виклики, такі як мережева нестабільність, відмова частини застосунку та ін. Дані проблеми і призвели до зростання популярності шаблонів стійкості, що є критично важливими для покращення ефективності застосунку. Вони спрямовані на запобігання відмовам у системах з великою кількістю залежностей, проте через статичні параметри можуть виявитися недостатньо ефективними чи обмежувати продуктивність системи надмірно. В ході наведеного вище аналізу також відмічено роботи, що є близькими до даної за темою і розкривають динамічні налаштування параметрів шаблонів стійкості, проте зазвичай містять один чи два таких шаблони. Разом з тим такі роботи зазвичай показують значні прирости ефективності, що сягають близько трикратного покращення, порівняно з звичайним, статичним варіантом реалізації згаданих в роботах шаблонів. В розділі також наведено аналіз деяких ключових для шаблонів стійкості параметрів і згадки значень у відповідних роботах.

Отже, нами було отримано достатнє інформаційне підґрунтя для подальшого власного дослідження, отримані знання будуть використані в наступних розділах.

РОЗДІЛ 2 ТЕОРЕТИЧНІ ОСНОВИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Загальні відомості про архітектуру і шаблони стійкості

Мікросервісна архітектура є основою багатьох сучасних веб-застосунків. Як відомо, сутність даної архітектури полягає в створенні незалежних ізольованих сервісів шляхом розбиття великої системи на невеликі сервіси, кожний з яких відповідає за окрему бізнес-функцію. Це забезпечує застосунку:

- Масштабованість: окремі компоненти чи сервіси легко окремо масштабуються в залежності від навантаження.
- Незалежність розгортання: зміни в одному сервісі не впливають на інші, що скорочує ризик впровадження оновлень.
- Полегшене тестування та обслуговування: завдяки розподілу відповідальності можна ізолювати помилки й швидко реагувати на збої.
- Гнучкість впровадження нових технологій: кожен сервіс може використовувати найкращі практики чи навіть різні технологічні стекі залежно від специфіки завдань.

В сучасних умовах дані переваги є суттєвими, через що популярність даної архітектури активно зростала останніми роками. Це зумовило також стрімкий розвиток технологій, в першу чергу комерційних, що намагалися задовольнити попит на спрощення розробки застосунків з використанням даної архітектури.

Тим не менш, вказані переваги існують поряд з тим, що розподілена природа мікросервісних систем створює й нові виклики, зокрема нестабільність мережевого з’єднання чи затримки в ньому, перевантаження компонентів системи може значно вплинути на продуктивність всього програмного рішення[10]. І хоча сервіси ізольовані і відмова одного компоненту не впливає на роботу іншого, допускати необроблені помилки і великий час очікування відповіді є не найкращим рішенням.

Отже, зі збільшенням кількості користувачів і, відповідно, навантаження, вимоги до надійності і продуктивності системи підвищуються. Власне, через це в таких архітектурах все більш важливим стає впровадження шаблонів стійкості, що гарантують покращення продуктивності і стійкості, а динамічна їх конфігурація має покращити вказаний ефект.

2.2 Шаблони стійкості

Шаблони стійкості – це струкутурні рішення, що допомагають мікросервісам ефективно реагувати на відмови сервісів, зберігаючи доступність основних сервісів і їх функціональність. Це дозволяє зменшувати кількість збоїв у системі в цілому і дає допоміжному сервісу час на відновлення або час команді на його обслуговування.[7] Однак для різних ситуацій той чи інший шаблон стійкості буде більш ефективним, тому варто розглянути кожен з них окремо.

2.2.1 Circuit Breaker

Circuit Breaker – один з найпопулярніших з шаблонів стійкості. Даний шаблон відповідає за моніторинг стану відповідей на запити між сервісами. Він запобігає повторним запитам до сервісу базуючих на даних про відповідь сервісу, що повертав повідомлення про відмову. Його назва походить від електричного запобіжника: при збільшенні кількості помилок він переходить зі стану Closed(закритий), де він пропускає всі запити до сервісу, в стан Open(відкритий), тобто “розриває” ланцюг і не пропускає нові запити далі. Це дає сервісу час на відновлення, дозволяє системі “охолодитися” і не допускає каскадної відмови через надмірні запити. Через вказаний проміжок часу він переходить, як правило, у стан Half-Opened(напіввідкритий), тобто намагається отримати відповідь від сервісу, припускаючи, що він за цей час мав би відновитися.[8] За успішного результату він переходить до початкового стану, а у випадку помилки знов у відкритий стан. Відповідно, такий шаблон має такі переваги:

- запобігає накопиченню збоїв через завчасне переривання ланцюгу запитів
- полегшує відновлення сервісу через відсутність навантаження
- обмежує використання ресурсів на непрацюючий сервіс

Недоліками ж такого підходу є:

- складність налаштування порогових значень може призвести до надмірного навантаження/надто раннього блокування

Вірогідно, даний шаблон стійкості підійде для критично важливих елементів системи, що мають бути максимально доступними, і при збільшенні навантаження краще перекрити до них доступ взагалі, давши їм час на відновлення. Загалом даний патерн видається найбільш часто використовуваним, оскільки пропонує більш інтелектуальний підхід до вирішення проблем доступності.

2.2.2 Bulkhead

Bulkhead – патерн, заснований на аналогії з конструкцією корабля (де перегородки, від яких і походить його назва, ізолюють конкретні відсіки). Тобто порушення в одному відсіку не мають впливати на загальну “плавучість”. В цьому випадку даний патерн ізолює критичні ресурси(потоки виконання, запитів до БД, сокети, тощо) для запобігання каскадної відмови застосунку [5]. Для цього вони (ресурси) поділяються на “комірки”(cells), кожна з яких і отримує свою кількість ресурсу, окремий пул ресурсів, отже не може надмірно навантажити компоненти системи і не впливає на інші комірки[13]. Тобто при вичерпанні ресурсів сервіс перестає отримувати нові запити. Такий підхід має переваги:

- запобігає ланцюговим реакціям при відмові через ізоляцію ресурсів
- оптимізує використання ресурсів розподіляючи їх

Однак він також створює обмеження:

- обмеженість ресурсів комірки встановленими заздалегідь межами може виявитися критичною

- розподіл ресурсів також потребує ресурсів
- впровадження потребує ретельного планування обмежень

Даний патерн скоріше підійде застосункам, де є велике навантаження, сфера його застосування дуже близька до вищезгаданого шаблону стійкості.

2.2.3 Time Limiter

Time Limiter – доволі простий шаблон стійкості, який використовує часові обмеження очікування відповіді на запити до сервісу. Якщо запит не оброблено у встановлений часовий проміжок, система скасовує операцію, що запобігає накопиченню “завислих” запитів. Таким чином запобігається найчастіша проблема з мережевими проблемами – надто довгі очікування. Вони зазвичай витрачають значні ресурси системи, що знижує її ефективність і реактивність на вказаний період. Завдяки цьому шаблону існує можливість встановити максимальний час очікування відповіді, за спливання якого запит повертає помилку або виконує альтернативну дію(повертає власний код, кешовані дані, тощо)[9]. Перевагами такого шаблону є:

- уникнення блокування ресурсів на тривалий час
- краща загальна доступність і гнучкість системи

Можна однак виділити такі недоліки:

- непродумані обмеження можуть призвести до передчасного завершення запитів без можливості отримати інформацію навіть за стандартного часу відповіді
- ризик втрати даних за відсутності fallback-механізму

Даний шаблон є доволі простим в налаштуванні, що робить його дуже привабливим для застосування у другорядних сервісах, що мають давати швидку відповідь і, зазвичай, не потребують складних обчислень

2.2.4 Rate Limiter

Rate Limiter – шаблон, що схожий на попередній, однак обмежує не час запиту, а їх кількість, а точніше частоту вхідних запитів. В такий спосіб він

запобігає перевантаженню системи і покращує її стабільність. Це доволі прямолінійний спосіб позбавити сервіс надмірного навантаження – обмежити кількість запитів, що можуть бути оброблені ним за певний час. Існує декілька основних підходів до встановлення лімітів: фіксовані, змінні, токен-бакет (запит забирає токени, які з часом наповнюються), лічильник (фіксована швидкість обробки запитів)[14]. Такий підхід має наступні переваги:

- запобігає перевантаженню в дуже простий спосіб
- має доволі тривіальне і гнучке налаштування

Недоліками можна виділити:

- ризик блокування через занадто суворі обмеження
- неефективність сервісу у піковому навантаженні і простому обмеженні
- необхідність припасування лімітів при зміні апаратного та програмного аспектів системи

Цей патерн дозволяє захистити сервіс від надмірного навантаження, його сфера застосування може бути близькою до Time Limiter

2.2.5 Retry

Retry – шаблон стійкості, заснований на повторному виконанні запитів при відмові. Він, попри свою простоту, забезпечує доволі ефективне рішення проблеми, якщо збої трапляються не часто і є одиничними. Як і попередній, шаблон автоматичного повторення має декілька варіантів реалізації: фіксований (повторення з однаковим часовим проміжком), експоненційний (збільшення часу між викликами з кожною наступною спробою), додавання випадкових затримок (для уникнення синхронного повторення запитів клієнтів), повторення з обмеженням (для запобігання нескінченному повторенню)[15]. З переваг хотілося б відмітити:

- прості імплементація і налаштування
- зменшення впливу на систему короткочасних проблем
- при використанні комбінованого підходу є достатньо ефективним

Однак такий підхід містить також і недоліки:

- при неналежному налаштуванні сервіс отримує шторм запитів і не відновиться
- визначення оптимальних параметрів все ще потребує дослідження
- якщо відмова пов’язана з серйозними проблемами лише збільшать навантаження і створюють ілюзію можливості отримання відповіді

Цей шаблон може бути застосований всюди, оскільки дозволяє просто вирішити проблему стабільності, однак при використанні на сервісі, який вже відмовив і не відновиться, він може погіршити ситуацію. Тому краще використовувати його там, де можливі короткочасні і випадкові відмови.

2.3 Інтеграція і реалізація шаблонів в мікросервісному середовищі

Оскільки всі вказані шаблони стійкості мають свої переваги і недоліки, логічно було припустити, що в реальних застосунках різні сервіси матимуть і різні шаблони стійкості залежно від специфіки виконуваної задачі. Наприклад, Circuit Breaker, який реагує на кількість відмов сервісу, а не на час його відповіді, може бути застосований до критичних елементів системи, що не обов’язково мають давати швидку відповідь, однак мають забезпечувати стабільний функціонал. У той час як на інші сервіси, що можуть надавати контент для перегляду, звичайного Time Limiter буде достатньо, щоб не змушувати користувача чекати відповіді надто довго. Не виключено також використання комбінацій шаблонів, наприклад, використання Circuit Breaker разом із Retry дозволяє відновлювати роботу після короткочасних збоїв, тоді як Bulkhead обмежує вплив перевантаження одного сервісу на решту системи. Однак, як вже було зазначено, найбільш ефективним видається саме імплементація динамічної конфігурації даних патернів, що завдяки застосуванню систем моніторингу, що дозволяють відстежувати важливі для прийняття рішень метрики, допомагає коригувати налаштування шаблонів в реальному часі.

В рамках даного застосунку не буде існувати різниці між шаблонами стійкості і їх сервіси будуть однаковими, оскільки така різниця призведе до некоректності результатів. Шаблони стійкості будуть використовуватися як обгортка до сервісів, тобто запити ззовні системи будуть надходити у вказаний сервіс через шаблон стійкості, що буде перевіряти внутрішні параметри системи і пропускати чи відмовляти запиту в обробці. Як було зазначено, було вирішено зосередитися не на написанні власної реалізації даних шаблонів стійкості, а на аналізі вже існуючих рішень, оскільки існує достатньо гідних уваги рішень.

2.4 Аналіз наявних інструментів

Перед початком розробки і проектування даного застосунку, який буде використано для моделювання використання і конфігурації даних патернів, розглянемо технології, які знадобляться в даному застосунку. Для імплементації вказаної ідеї в тестовому стенді було обрано такі інструменти, які найбільше підходять для даної задачі, попередньо зробивши аналіз аналогічних рішень і інструментів, що могли б бути використані.

Перше що варто згадати, було обрано мову Java та фреймворк Spring Boot. Це популярний стартер, що спрощує налаштування і інтеграцію залежностей Він забезпечує основу для створення застосунків на основі мікросервісів і вже є доволі добре налаштованим і високопродуктивним. Зокрема Java Spring Boot Actuator допоможе нам у моніторингу мікросервісів і зборі метрик продуктивності.

Для реалізації самих патернів у коді було вирішено використати готову бібліотеку для вказаного фреймворку Resilience4j. Вона була згадана в інших академічних дослідженнях і дозволяє легко конфігурувати вказані шаблоні, зокрема і змінювати їх конфігурацію під час роботи.

З метою збору метрик і їх зберігання задля подальшого використання було вирішено використати Prometheus. Даний інструмент надає високопродуктивне рішення для вказаних задач, має можливість використання

його в ізольованому контейнері і добре інтегрується з уже вказаними технологіями реалізації мікросервісної архітектури. Відповідно, для його запуску використали Docker як інструмент контейнеризації.

Таблиця 2.4 – Характеристика обраних інструментів та їх аналогів

Інструменти	Причини використання	Аналоги	Переваги	Недоліки
Java + Spring Boot	1. Забезпечує швидкий початок роботи 2. Підтримує інтеграцію з іншими інструментами 3. Є нам знайомим і містить всі необхідні інструменти	Python +Flask	Міні - фреймворк, отже ще простіший в розгортанні	Має меншу функціональність
		Ruby on Rails	Велика кількість вбудованих можливостей	Менша продуктивність і масштабованість
Resilience4j		Hystrix (Netflix)	добре реалізує Circuit Breaker	не оновлюється, отже менш перспективна
		Failsafe	легка бібліотека для шаблону стійкості повторних спроб	відсутня реалізація інших шаблонів стійкості
Prometheus	1. Забезпечує систему збору і зберігання інформації 2. Забезпечує контейнеризацію та ізоляцію	InfluxDB + Telegraf + Chronograf	повноцінний стек, гкучкість	більш складний, менш застосовуваний

2.5 Формулювання гіпотези

Перш ніж розробляти застосунок, зазначимо початкову гіпотезу, яку протягом даної роботи спробуємо підтвердити: використання динамічно налаштовуваних патернів на основі попередньо зібраних метрик покращує ефективність системи і запобігає відмовам. Відповідно, щоб перевірити дане твердження, буде розроблено застосунок, що буде реалізувати даний принцип і збирати метрики ефективності системи. Зокрема, для досягнення цієї мети, нам потрібно буде пройти три етапи: розробка даної системи, реалізація у програмному коді, тестування та аналіз зібраних даних.

2.6 Розробка архітектури застосунку

Очевидно, що дане програмне рішення міститиме в собі класи, що будуть реалізувати і задавати ініціалізаційні дані у вказаних патернах з використанням вже вказаних інструментів. Їх використано для перевірки трафіку, що буде приходити на сервіс. Крім цього знадобиться використати також тестер, що буде імітувати навантаження на сервіс і буде з певною періодичністю робити до нього запити через вказані патерни. В його налаштуваннях міститиметься кількість потоків і динаміка таких тестувань. Він робитиме запити за допомогою HTTP-протоколу, тож потрібно буде створити контролери з кінцеві точки , які будуть цей запит отримувати і надалі передавати до сервісів. Використання декількох кінцевих точок покращить сутність моделювання.

Окрім цього, нам треба буде мати сам сервіс, що буде моделювати обробку запитів і відповідати, і відповідно його будуть наслідувати сервіси, що будуть реалізовувати патерни. Це дозволить нам уникнути дублювання коду. Тут варто зупинитися детальніше, адже ця частина застосунку є критичною і напряму вплине на результати дослідження, оскільки для правильності експерименту має бути наближеним до реальної роботи застосунку. Виходячи з цього, просто додати затримку в даному сервісі, імітуючи розрахунки, буде недостатньо, оскільки в реальних сервісах дані процеси не є лінійними і час обробки не є ідеально однаковим. Відповідно, для кращого моделювання

потрібно буде додавати випадкові затримки до обробки кожного запиту. Додатково треба запровадити наявні аномальні реакції, імітуючи якісь критичні затримки всередині системи. Для простоти системи можна встановити ці аномалії з однаковими часовими проміжками, оскільки такий аналіз не потребуватиме аналізу часу аномалії і проміжку між ними, що було б доволі складною задачею, беручи до уваги те, що нам не вдалося знайти відповідну метрику в Prometheus.

Говорячи про сервіс збирання метрик, нам також потрібно буде реалізувати клас, що буде відповідати за налаштування даного сервісу для відправлення і зберігання цих метрик. Згідно з даними самого сервісу, його буде розгорнуто у контейнері незалежно від основної система, яка, скоріш за все, докеризуватися не буде. Важливими метриками для аналізу стануть середній час відповіді, 90 та 50 перцентилі та ін. Наведемо архітектурну діаграму застосунку:

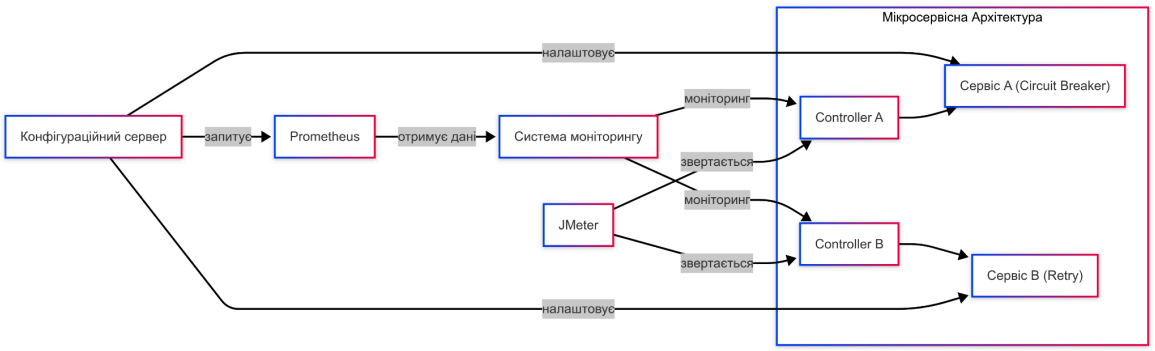

Рисунок 2.1. – Архітектурна діаграма застосунку

За даною схемою бачимо, що JMeter робить одночасні виклики до контролерів кожного з сервісів. Ті в свою чергу за допомогою власних сервісів, незалежно генерують, згідно з зазначеними в наступному розділі розрахунками, затримку і надсилають відповідь назад. Перевагою використання вказаної схеми є те, що вона дозволяє агрегувати дані про кількість відмов та час відповіді і побудувати відповідні візуалізації вказаних даних різного виду за допомогою вбудованих в JMeter метрик. Варто зазначити, що можливо також і тестування інтегрованого застосунку, де сервіси спілкуються між собою. Така варіація моделі може покращити результати експерименту, наблизивши їх до реального

застосунку, де сервіси дуже часто комунікують між собою. Водночас так можна отримати неточні дані, оскільки навантаження на сервіси вийде нерівномірним і чим далі по ланцюгу буде переходити запит від JMeter, тим більше шанс, що його відкине якийсь з патернів. Запропонована ж архітектура дозволяє у простий спосіб дослідити вказану теорію.

Наведемо також діаграму послідовності застосунку:

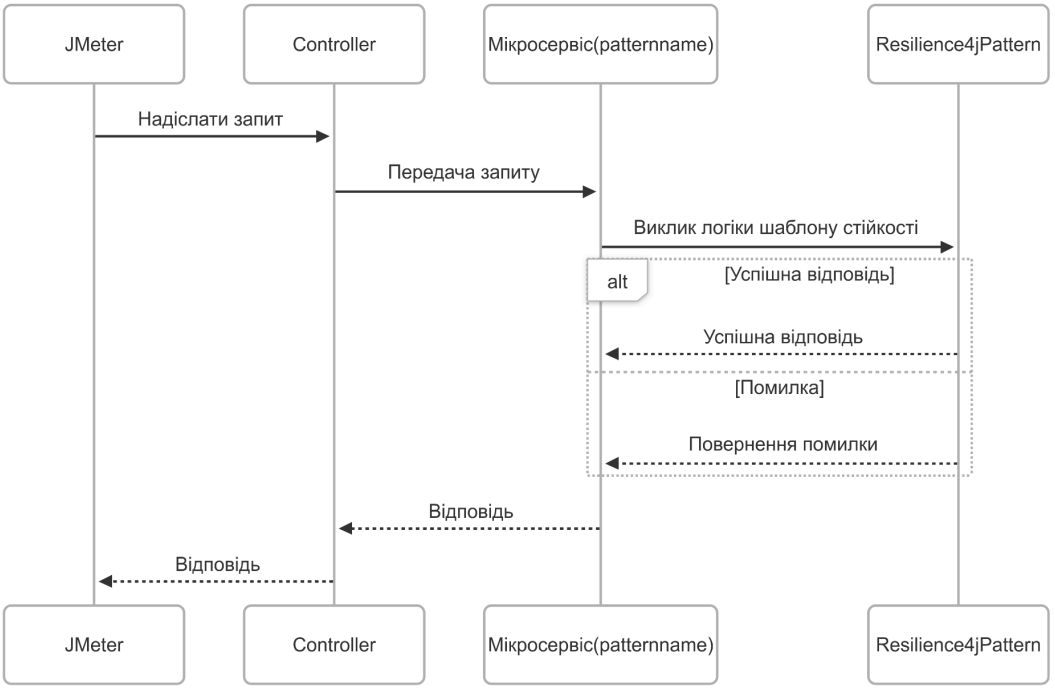

Рисунок 2.2 – Sequence-diagram запитів до сервісів

На даній діаграмі більш детально проілюстровано поведінку застосунку. Як вже було сказано, комунікація через HTTP-протокол відбувається між навантажувальним сервісом і контролером, який викликає відповідний сервіс і використовує для обмеження трафіку шаблони стійкості. Тут показано лише “бізнес” частину застосунку.

Відмітимо також значні обмеження, особливості і умови, що значно вплинуть на процес дослідження. По-перше, це відсутність даних з реальних прикладних застосунків. Не вдалося знайти ані даних про використовувані значення параметрів, ані про використовувані шаблони стійкості чи їх динамічну конфігурацію. Отже, в аналізі основою стануть лише на дані з інших досліджень і на власний емпіричний досвід. По-друге неідеальність моделі.

Незалежно від того, яку архітектуру та форму вона матиме, базуватися вона буде на теоретичних знаннях про мікросервісну архітектуру і висновках та порадах інших дослідників. Відтак, модель не може бути повною і осягнути весь масштаб проблеми чи абсолютно вірно моделювати поведінку реального застосунку.

Однак в даній роботі і не було поставлено перед собою таку мету, адже поставлена в роботі мета – перевірити твердження про ефективність динамічної конфігурації для шаблонів стійкості і дослідити ефекти такого налаштування.

Висновки до розділу II

В даному розділі подано аналіз шаблонів стійкості і їх параметрів, мікросервісної архітектури як такої і наведено декомпозицію поставленої задачі. Виявлено переваги і недоліки вказаних шаблонів стійкості, мікросервісної архітектури, описано виклики, з якими стикаються розробники. Вказаний аналіз знадобиться в наступному розділі при імплементація вказаних рішень у коді.

В ході дослідження було запропоновано архітектуру майбутнього тестового середовища, що включає в себе контролери і сервіси. Дана архітектура, попри свою простоту, забезпечує виконання поставлених цілей і дозволяє дослідити швидкодію застосунку і шаблонів стійкості в статичній та динамічній варіації. В розділі вказано, що основними метриками, які стануть основою подальшої методології динамічної конфігурації шаблонів стійкості стануть експоненційно згладжений середній час відгуку, середній час відгуку, 90 перцентиль, 50 перцентиль. Визначено, що для комунікації навантажувальника і мікросервісів буде використовуватися протокол HTTP і звернення до відповідних контролерів, що матимуть кінцеві точкивідповідно до своєї назви

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА

3.1 Реалізація моделі і динамічної конфігурації

Окрему увагу варто приділити методології дослідження. Зважаючи на представлений в інших дослідженнях аналіз, видається логічним застосувати при моделюванні динамічне налаштування, враховуючи середній час відгуку і зокрема 90 перцентиль як одні із головних метрик для аналізу ефективності мікросервісного застосунку. Тут однак постає питання про створення самого алгоритму і визначення порогових значень, оскільки за відсутності даних з реальних бізнес-застосунків, доводиться спиратися на інші дослідження в питаннях відсотків відмов сервісу, його навантаження і відповідних навантаженню налаштуваннях шаблонів стійкості. Вірогідно, що під час підготовки до моделювання доведеться провести кілька тестових моделювань для розуміння адекватності моделі і правильності налаштування параметрів. Цільовою задачею в плані продуктивності для нас стане сама наявність покращення в динамічному варіанті, порівняно з статичним варіантом, враховуючи наявність декількох шаблонів стійкості і, відповідно, складності їх правильного налаштування. Однак перш за все очікується покращення в районі 10-30% сукупно в середньому часі очікування.

В основі динамічного налаштування шаблонів стабільності лежить цикл “виконуй - аналізуй - коригуй”. Це означає, що модуль динамічної конфігурації спочатку збирає дані про важливі метрики застосунку, аналізує і імплементує на їх основі налаштування шаблонів стійкості і повторює даний цикл для коригування регулярно. В застосунку кожний сервіс має свій шаблон стійкості і налаштування відповідного патерну відбувається через API бібліотеки в реальному часі, що дозволяє підвищити гнучкість механізмів забезпечення стабільності.

Вирішено використати один контролер на патерн, щоб зібрані прометеусом метрики були більш інформативними. Оскільки він має змогу збирати дані про запити з різних кінцевих точок, дана інформація допоможе

краще зрозуміти особливості застосування того чи іншого патерну та його налаштування. Однак для аналізу все одно будуть в тому числі використовуватися консолідовані дані з усіх сервісів, враховуючи, що система в нас одна і, відповідно, ресурси системи поділяються між усіма сервісами.

Звертаючись до досвіду інших досліджень, у дослідженні [9] зазначено, що для клауд-застосунків з мікросервісною архітектурою отриманий відсоток відмов становить як 17%, хоча з посиланням на інше джерело вказано на очікування в 30%, що насправді є доволі великим у випадку з критично важливими сервісами. З інших джерел [16] можемо визначити, що насправді критичними помилками завершуються приблизно 1% запитів, в той час як наявність некритичних помилок у відповідях на запити може сягати більш значних цифр, що зазначені вище. Однак враховуючи динамічний характер роботи застосунку, ініціалізуючі дані можуть бути дуже швидко змінені, а правила налаштування будуть спиратися на ці дані.

Більше уваги було приділено вивченню часу затримок у застосунках. Однак однозначної відповіді про стандартне значення часу відгуку нам знайти не вдалося. Це пов’язано з тим, що час відгуку дуже сильно залежить від складності запиту, відстані до серверу, поточного стану мережі, наявності високого навантаження, тощо. У зв’язку з цим було вирішено використати прості, проте разом з тим виважені числа для початкового та динамічного налаштування.

3.2 Структура моделі

Для емпіричного дослідження вказаних в теоретичній частині тверджень та теорій було розроблено тестове середовище для моделювання мікросервісної архітектури з використанням 5 найбільш застосовуваних патернів: Circuit Breaker, Bulkhead, Rate Limiter, Time Limiter, Retry. Усі ці патерни будуть тестуватися на сервісах паралельно окремими застосунками, що будуть симулювати запити клієнтів.

Окрім цього для виклику всіх вказаних сервісів, що їх обслуговують патерни, було створено контролери, які приймає запити через HTTP-запит і викликають відповідні методи у сервісах. Варто зазначити, що такий вибір комунікації не є випадковим і служить не лише імітації взаємодії у реальному застосунку: для відслідковування ефективності роботи застосунку, часу запиту і використаних ресурсів системи було вирішено використовувати систему Prometheus, яка, як відомо, моніторить інтернет-трафік до системи. Отже, без виконання запитів неможливо було б використовувати цю систему для збирання і візуалізації даних. Наведемо один з таких контролерів для загального розуміння взаємодії з застосунком:

```
@RestController  CS31IvanK *
@RequestMapping("/bulkhead")
public class BulkheadController {

    private final BulkheadService bulkheadService;  2 usages

    @Autowired  CS31IvanK
    public BulkheadController(BulkheadService bulkheadService) {
        this.bulkheadService = bulkheadService;
    }

    @GetMapping  CS31IvanK *
    public ResponseEntity<String> processRequest() {
        try {
            String response = bulkheadService.sendRequest();
            return ResponseEntity.ok(response);
        } catch (Throwable t) {
            System.out.println("BE");

            return ResponseEntity.status(500)
                    .body("BE: " + t.getMessage());
        }
    }
}
```

Рисунок 3.1 – Приклад контролеру мікросервісу

Згадуючи сервіс метрик, варто згадати і метод збирання даних з цього сервісу. Для цього було створено окремий клас, що буде робити запити до контейнера і запитувати необхідну метрику. Методи були розроблені з можливістю зміни часового проміжку збору інформації. Наприклад, для отримання середньої відповіді та перцентилю клас має наступний метод:

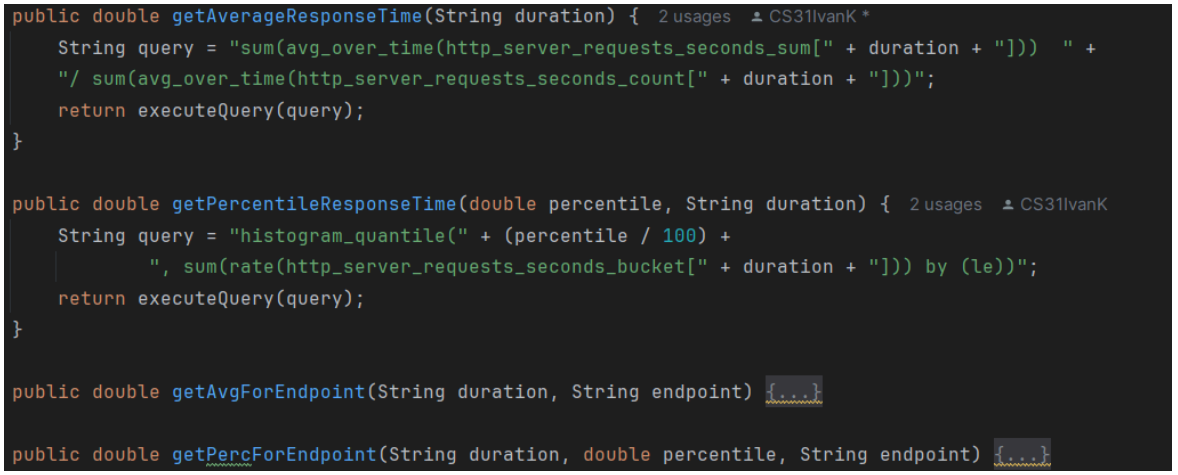

Рисунок 3.2 – Методи для отримання середньої відповіді та перцентилів

Перейдемо до огляду налаштувань самої моделі. Сервіси наслідують абстрактний клас, що задає усім час затримки і формат її обчислення. Нами було обрано наступне налаштування: базова затримка становить 300 мілісекунд, крім неї додається ще випадкова затримка для симулювання різних станів системи. З вірогідністю 50% система буде відповідати від 100 до 701 мілісекунди довше. Дана формула пояснюється тим, що не всі запити є однаковими по складності, тому і час на їх обробку буде відрізнятися. Крім цього варто відмітити, що до цього будуть додаватися й інші, природні затримки на обробку і передачу запитів, на що вплинути немає можливості і що не вплине значно на результат експериментів. Для візуалізації наведемо сам абстрактний клас:

```java
public abstract class RequestService {  5 usages  5 inheritors  CS31IvanK
    protected String simulateDelay(String patternName) {  5 usages  CS31IvanK
        long delay = 300;


        if (ThreadLocalRandom.current().nextDouble() < 0.5) {
            long extraDelay = ThreadLocalRandom.current().nextLong( origin: 100,  bound: 701);
            delay += extraDelay;
        }

        try {
            Thread.sleep(delay);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return "[" + patternName + "] Помилка: " + e.getMessage();
        }
        return "[" + patternName + "] Response after " + delay + "ms";
    }
}
```

Рисунок 3.3 – Створений базовий клас для сервісів

Сервіси ж, що розширюють цей клас, не вирізняються особливою функціональністю, адже їх основна задача – використати обгортку патерну для виконання методу симуляції затримки. Основне, що варто відмітити – кожний патерн має свій сервіс, хоча вони й схожі. Наведемо типовий приклад:

```java
@Getter  CS31IvanK
@Service
public class RetryService extends RequestService {
    private final Retry retry;

    @Autowired  CS31IvanK
    public RetryService(RetryImplementation retryImplementation) { this.retry = retryImplementation.getRetry(); }

    public String sendRequest() throws Throwable {  1 usage  CS31IvanK
        return retry.executeCheckedSupplier(() ->
                simulateDelay( patternName: "Retry")
        );
    }
}
```

Рисунок 3.4 – Приклад створеного сервісу

Розглянемо також налаштування JMeter. Даний тестувальник навантаження зазвичай надає можливість лише тестування за допомогою стабільного навантаження однаковою кількістю користувачів, однак за допомогою плагіна отримано змогу налаштувати Stepping User Group – групу користувачів змінного розміру. Згідно з налаштуваннями, цей варіант тестування дозволяє задати початкове значення користувачів, яке буде

збільшуватися на задану кількість користувачів з певною періодичністю, після чого починає поступово знижувати навантаження. Це дозволяє протестувати застосунок в найбільш критичний момент його існування – збільшення навантаження до критичного, де час очікування і стабільність є найважливішими. Отже, тестування проводилося в піковому навантаженні потоком з 270 користувачів, оскільки за більшого значення система відмовляла всім потокам за внутрішніми алгоритмами фреймворку. Навантаження починалося з запуску 100 потоків, після чого кожні 3 секунди додавалося ще 10 потоків з ramp-up періодом у 3 секунди. Після досягнення цільових 270 потоків, JMeter утримував навантаження протягом 20 секунд. Зрештою, він починав завершувати 5 потоків кожні 3 секунди, імітуючи завершення напливу користувачів, до завершення всіх потоків. Для порівняння двох станів налаштування патернів – мануального та динамічного – нам знадобилося створити дві абсолютно ідентичні групи користувачів. Єдина різниця між ними полягає в тому, що друга група починає свою роботу за 300 секунд від запуску тестування, тобто тоді, коли перша група вже закінчила свою роботу. Відповідно, конфігурація патернів також не вмикається перші 5 хвилин роботи застосунку, щоб мати можливість отримати дані про роботу застосунку з мануально налаштованими патернами. Оскільки в нас 5 кінцевих точок з шаблонами стійкості, то кожна група користувачів буде робити запити до них. Тобто навантаження має бути на всі кінцеві точки однаковим. Також використали вбудовані методи збору та візуалізації даних. Наведемо також графік зміни навантаження для кращої візуалізації:

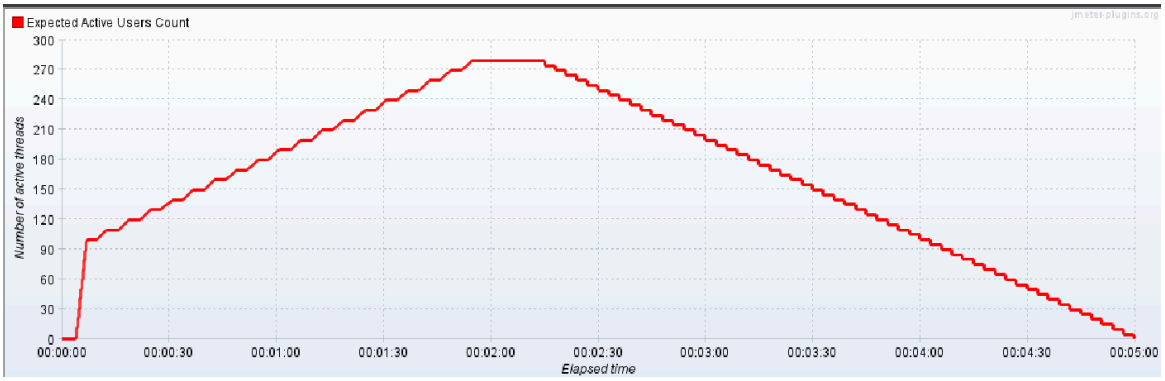

Рисунок 3.5 – Схема зміни навантаження першої групи на застосунок

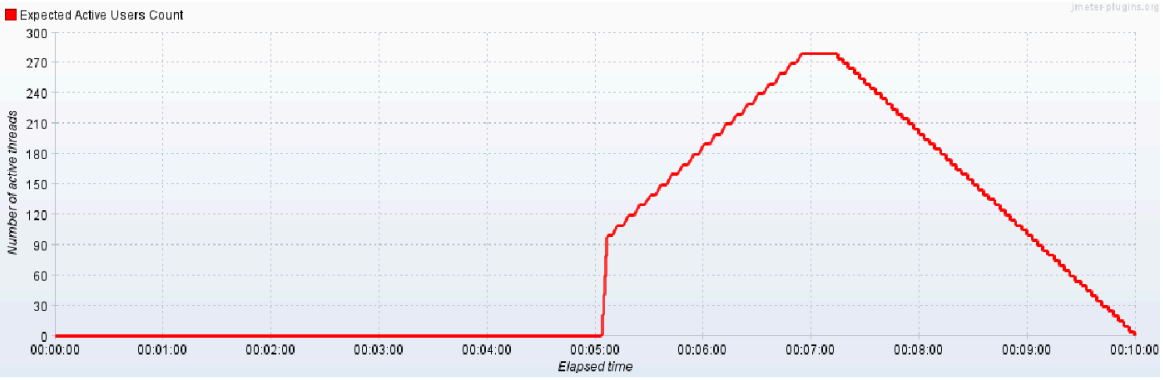

Рисунок 3.6 – Схема зміни навантаження на застосунок другою групою

3.3 Аналіз параметрів шаблонів та налаштування

Перш за все варто зазначити, що під час ініціалізації параметрів шаблонів стійкості основою було те, що розробнику, який буде проводити аналогічну операцію в реальному застосунку, скоріш за все буде відомо про середній час затримки у відповідях. Відповідно, налаштовувати статичну версію застосунку він буде базуючись на вказаних метриках. При налаштуванні алгоритму динамічного налаштування було враховано висновки з вищевказаних робіт і власні розмірковування, що буде детально описано нижче, при вивченні кожного окремого шаблону стійкості. Основними метриками, що їх було використано для аналізу, були середній час відповіді за 1 хвилину та за 5 хвилин, стан завантаженості процесора та пам’яті як показники завантаженості апаратної частини, перцентилі завантаження. Зазначимо, що розгляд середнього часу відповіді за останні 10 хвилин не був би доцільним, враховуючи що було

вирішено проводити динамічну конфігурацію кожну хвилину для кращої реакції на зміни в застосунку. Теоретично час реакції можна було б встановити і меншим, однак таке рішення могло б призвести до викривлення результатів, в той час як збільшення метрик і додавання ще більшого діапазону даних для аналітики видається більш логічним рішенням. Через це було вирішено використовувати згладжене значення часу відгуку, що враховує історичні та нові дані у застосунку. Розглянути логіку налаштувань шаблонів стійкості варто окремо для кожного з них.

3.3.1 Аналіз параметрів шаблону стійкості Circuit Breaker

Даний шаблон стійкості має безліч параметрів, проте буде використано лише основні, оскільки збільшення використовуваних параметрів пропорційно збільшить і складність їх підбору і динамічної конфігурації. Отже, нами були обрані такі параметри:

- failureRateThreshold – рівень відмов сервісу, після якого сервіс перейде в стан “відкритий”, перестане приймати нові виклики
- waitDurationInOpenState – час очікування в “відкритому” стані, перед переходом в напіввідкритий стан

Перший параметр було залишено незмінним з моменту ініціалізації на рівні 0,5, тобто приблизно відповідно до очікувань в вищезгаданих роботах. Значення було низьким, оскільки в ході досліджень було виявлено, що дуже мало запитів завершуються саме відмовою сервісу, а не обмеженнями фреймворку. Другий же параметр було вирішено змінювати під час динамічної конфігурації.

Для визначення було використано експоненційне згладжування, щоб уникати надмірного впливу короткострокових коливань від випадковостей. Згідно з даною метрикою, визначається коефіцієнт альфа, що буде використовуватися для збільшення чи зменшення впливу поточних даних, порівняно з історичними. Нове значення очікування у відкритому стані визначалося залежно від експоненційно згладженого часу середньої відповіді:

якщо він більший за час середньої відповіді сервісу за останні 5 хвилин, то час очікування прирівнювався до різниці двох вказаних метрик, поділене на 2, якщо ж ні – до середнього часу відповіді сервісу. Таким чином час очікування сервісу в відкритому стані був мінімізований і сервіс мав би теоретично відновитися.

3.3.2 Аналіз параметрів шаблону стійкості Time Limiter

Шаблон обмеження часу є набагато простішим і містить два істотних параметри, що, звісно, спрощує його конфігурацію. Один з них відповідає за fallback-метод, а другий було використано. Його опис подано нижче:

- timeoutDuration – максимальний час очікування виконання запиту, перш ніж запит буде відкинуто.

Виходячи з того, що затримка може становити 1300 мс., було вирішено для статичної варіації обрати цей показник на рівні 1 секунди, щоб пропускати найбільш довгі запити і забезпечити прийнятну доступність сервісу.

Разом з тим для динамічної конфігурації цього параметру було використано також доволі простий підхід: згадане вже згладжене значення середньої відповіді сервісів множиться на 1,25, враховуючи, що мінімально даний параметр має бути 500 мс., інакше більшість запитів не дістануться шуканого сервісу, навіть попри його низьке навантаження. Простіше кажучи, використовуються дані про середній час відповіді і додаються 25% на випадок збільшення часу очікування. Попри простоту підходу, це забезпечує доволі гарну продуктивність.

3.3.3 Аналіз параметрів шаблону стійкості Bulkhead

Цей шаблон містить декілька параметрів, з яких було обрано два основних:

- maxConcurrentCalls – кількість одночасних дозволених викликів сервісу
- maxWaitDuration – максимальний час очікування запиту в черзі

Виходячи з того, що даний застосунок матиме 270 одночасних викликів, використано 200 одночасних потоків з часом очікування у 1 секунду. Це дуже рідко призводило до виключень викликів через відповідність параметрам, тож в динамічній конфігурації їх було зменшено.

Для динамічної конфігурації використали той самий підхід: зменшуємо, якщо час відповіді за останню хвилину збільшився, порівняно з останніми 5-ма хвилинами. Кількість викликів зменшено до 150, а час очікування встановлено відповідно до часу очікування за останні 5 хв. з додатковими 100 мс., що забезпечило доволі високу продуктивність системи в більшості випадків. Якщо ж навантаження зменшилося, кількість дозволених потоків залишалася на рівні 200.

3.3.4 Аналіз параметрів шаблону стійкості Rate Limiter

Обмежувач частоти запитів має трохи більше параметрів і було обрано ті, що будуть найбільш придатні до вказаного застосунку, а саме:

- limitRefreshPeriod – визначає період оновлення лімітів запитів
- limitForPeriod – встановлює максимальну кількість запитів у період
- timeoutDuration – час очікування в черзі на обробку

У статичній версії даного патерну використано час оновлення лімітів 1 секунду, ліміт у період в 10 запитів і час очікування в черзі як 800 мс. На практиці дані ліміти виявилися доволі великими, тому знову довелося звужувати їх у динамічній конфігурації.

При аналізі метрик і переналаштуванні шаблону стійкості було використано, аналогічно попередньому шаблону, згладжений час відгуку за останню хвилину і час відгуку за останні 5 хвилин, і, відповідно до порівняння, або звужували ліміт до 5, або збільшували до 20. Час очікування виставлено в час відгуку за п’ять хвилин і додано до цього ще 100 мілісекунд. Конфігурація під час моделювання нас влаштувала.

3.3.5 Аналіз параметрів шаблону стійкості Retry

Даний шаблон стійкості містить напрочуд багато параметрів, як для такого доволі простого патерну, однак нам з них знадобляться тільки 2, оскільки більшість спрямовані на забезпечення функціональності шаблону стійкості при використанні в прикладних застосунках. Отже, цими параметрами є:

- maxAttempts – максимальна кількість повторних спроб запиту
- waitDuration – максимальний час очікування на відповідь

В даних параметрах було дуже просто визначити ініціалізаційні дані: спроб було 3, час очікування – 800 мілісекунд. Такі дані є доволі логічними і забезпечують невелике навантаження на систему.

Що ж до динамічної варіації, то тут було вирішено застосувати інший підхід. Для перевизначення параметрів було використано 90-й та 50-й перцентилі, тобто час відповіді 90 та 50 відсотків запитів відповідно. Їх використано з наступною логікою: якщо значення 90-го перцентилю вище за 50й на певну значну позначку, то існує значна кількість запитів, що достатньо довго очікує на відповідь, а це означає, що треба зменшувати кількість запитів до сервісу. Отже, кількість спроб зменшувалася до 2 чи збільшувалася до 4 залежно від стану застосунку, а також встановлювали час очікування у 400 мілісекунд. Таким чином обмежено навантаження на застосунок, хоча попри це сервіс стабільно отримував нові запити.

3.3.6 Підведення підсумків

Можемо зробити висновок, що порівняно зі статичними, динамічні варіації шаблонів стійкості матимуть більш жорсткі вимоги до трафіку і, відповідно, мають зменшити час відповіді. Наведемо частину класу з реалізацією динамічного налаштування деяких шаблонів стійкості:

```
long newTimeoutMillis = (long) (historicalResponseTime1m * 1250);
if (newTimeoutMillis < 500) {
    newTimeoutMillis = 500;
}
Duration newTimeLimiterTimeout = Duration.ofMillis(newTimeoutMillis);
TimeLimiterConfig newTimeLimiterConfig = TimeLimiterConfig.custom()
        .timeoutDuration(newTimeLimiterTimeout)
        .build();
timeLimiterImplementation.updateConfig(newTimeLimiterConfig);

//alternative
double cbavg =  prometheusClientService.getAvgForEndpoint( duration: "5m",  endpoint: "/circuitbreaker");
int newDuration = (int) ((historicalResponseTime1m > cbavg) ? (historicalResponseTime1m * 2) : cbavg)+1;
CircuitBreakerConfig newCircuitBreakerConfig = CircuitBreakerConfig.custom()
        .waitDurationInOpenState(Duration.ofSeconds(newDuration))
        .build();
circuitBreakerImplementation.updateConfig(newCircuitBreakerConfig);

double rlavg = prometheusClientService.getAvgForEndpoint( duration: "5m",  endpoint: "/ratelimiter");

int rateLim = (historicalResponseTime1m > rlavg) ? 5 : 20;
RateLimiterConfig newRateLimiterConfig = RateLimiterConfig.custom()
        .limitForPeriod(rateLim)
        .timeoutDuration(Duration.ofSeconds((long) ( rlavg+0.1)))
        .build();
rateLimiterImplementation.updateConfig(newRateLimiterConfig);

double bavg = prometheusClientService.getAvgForEndpoint( duration: "5m",  endpoint: "/bulkhead");
int newMaxConcurrentCalls = (historicalResponseTime1m > bavg) ? 150 : 200;
BulkheadConfig newBulkheadConfig = BulkheadConfig.custom()
        .maxConcurrentCalls(newMaxConcurrentCalls)
```

Рисунок 3.7 – Динамічне налаштування шаблонів стійкості в коді

3.4 Налаштування застосунку і інструментів

Окрім створення відповідних класів і реалізації функціоналу, перед нами постала задача конфігурації сервісу Prometheus та самого застосунку. Для першого було обрано наступні налаштування:

```yaml
global:
  scrape_interval: 5s  # metric rate
  evaluation_interval: 5s

scrape_configs:
  - job_name: 'spring-boot-app'
    scrape_interval: 5s
    metrics_path: '/actuator/prometheus'  # metric path
    static_configs:
      - targets: ['host.docker.internal:8081']  # app

  - job_name: 'node-exporter'  # server monitoring (CPU, RAM)
    static_configs:
      - targets: ['localhost:9100']

  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

alerting:
  alertmanagers:
    - static_configs:
        - targets: ['localhost:9093']

rule_files:
  - "alert.rules.yml"
```

Рисунок 3.8 – Налаштування у файлі prometheus.yml

Як видно з вказаного на рисунку, збір даних відбувається кожні 5 секунд, що забезпечує безперервне оновлення даних і доволі точні графіки. Можемо також побачити кінцеву точку, де зберігатимуться метрики – /actuator/prometheus. туди застосунок надсилатиме дані, які вказаний сервіс буде зберігати. При конфігурації враховано ізоляцію сервісу в контейнері. Окрім того на рисунку видно й саму адресу інструменту, до якої можна звертатися для мануального моніторингу застосунку і дослідження різних метрик для подальшого аналізу. Варто відмітити, що дані в Prometheus групуються за кінцеві точки, що дозволяє збирати індивідуальні метрики по кожному сервісу для подальшого аналізу, чим ми, звісно, скористалися. Окрім цього існує можливість для консолідованого звіту, у випадку потреби комплексного аналізу всієї мікросервісної архітектури.

3.5 Умови моделювання та проведення експерименту

Моделювання проводилося протягом 10 хв. і без додаткового навантаження на систему. Як і було згадано, тестування проводилося по суті в два етапи: перший етап без динамічного конфігурування, другий з його використанням.

Після проведення моделювання нами були отримані наступні результати з часу відгуку сервісів у JMeter:

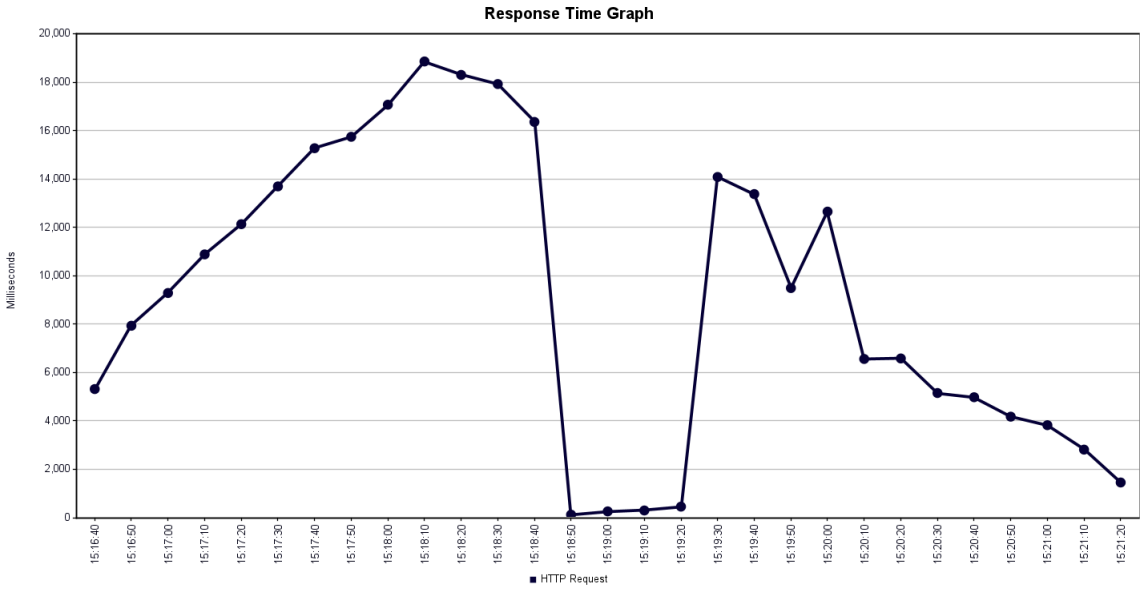

Рисунок 3.9 – Час відгуку сервісів протягом перших 5 хв.

Як бачимо, час очікування тут доволі високий і значно перевищує очікувані значення. Очікувано, час зростає з часом, оскільки збільшується навантаження на сервіси. В той же час можна бачити різке падіння всередині тесту, коли навантаження досягає піку і система блокує нові запити автоматично. Після цього час відповіді різко зростає і поступово спадає, разом зі зменшенням навантаження.

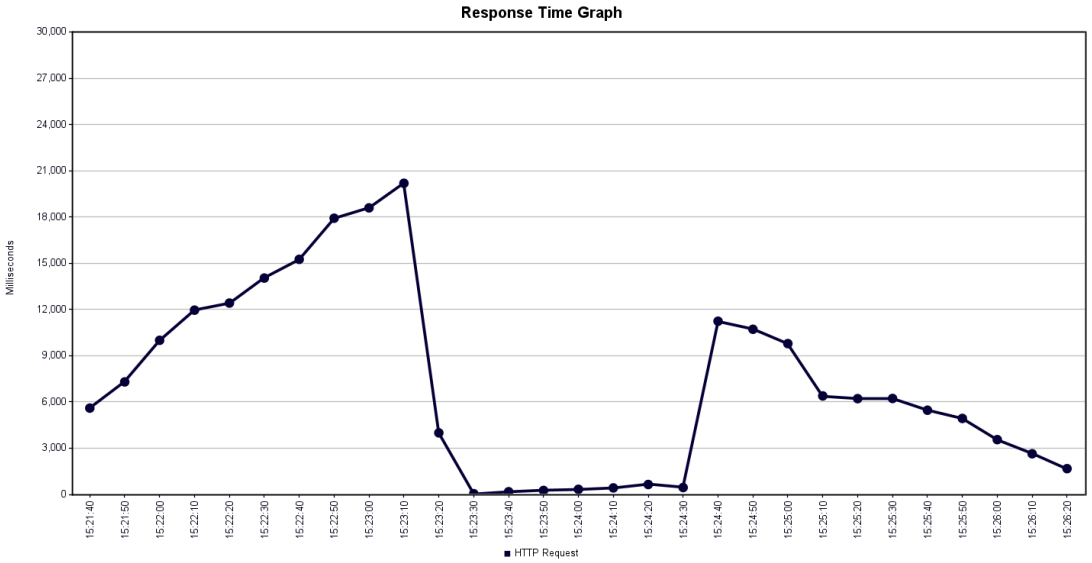

Рисунок 3.10 – Час відгуку протягом останніх 5 хв.

На графіку часу відповіді після початку динамічного налаштування бачимо, що час очікування все ще є суттєвим, проте він все ж зменшився. Разом з тим збільшилася прогалина посередині. Відмітимо загальне зменшення часу відповіді, що може бути проілюстроване наступними графіками.

Окрім графіку часу відповіді також створено графіки медіанного часу відповіді і дані про перцентилі:

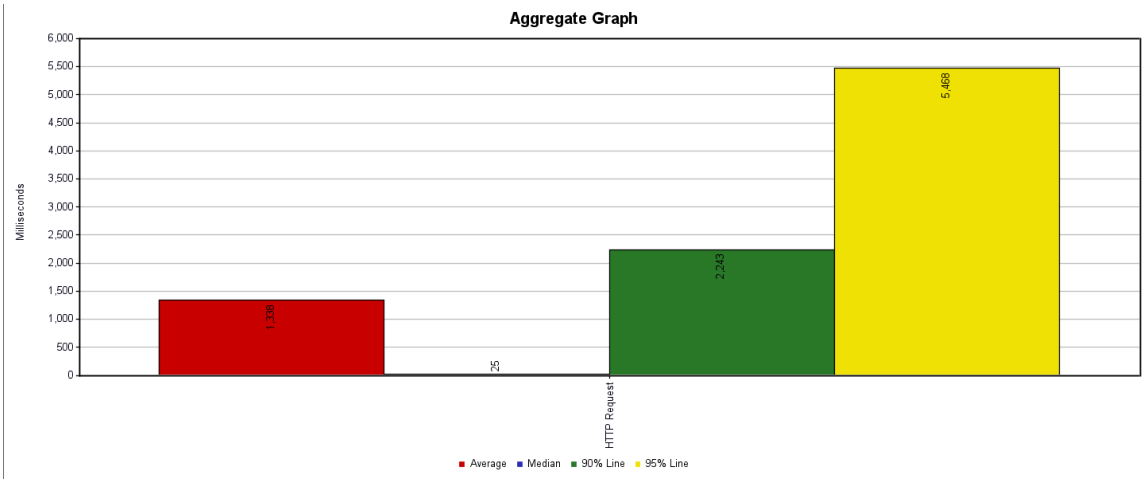

Рисунок 3.11 – Дані про відповіді до конфігурації

Тут можна побачити дані про перцентилі, медіанні та середні значення. Медіанне значення є низьким, оскільки в піковому навантаженні всі запити завершувалися дуже швидко. водночас 90 перцентиль на 95 відрізняються дуже

сильно, що каже нам про наявність невеликої кількості запитів, що завершуються з дуже великим часом очікування.

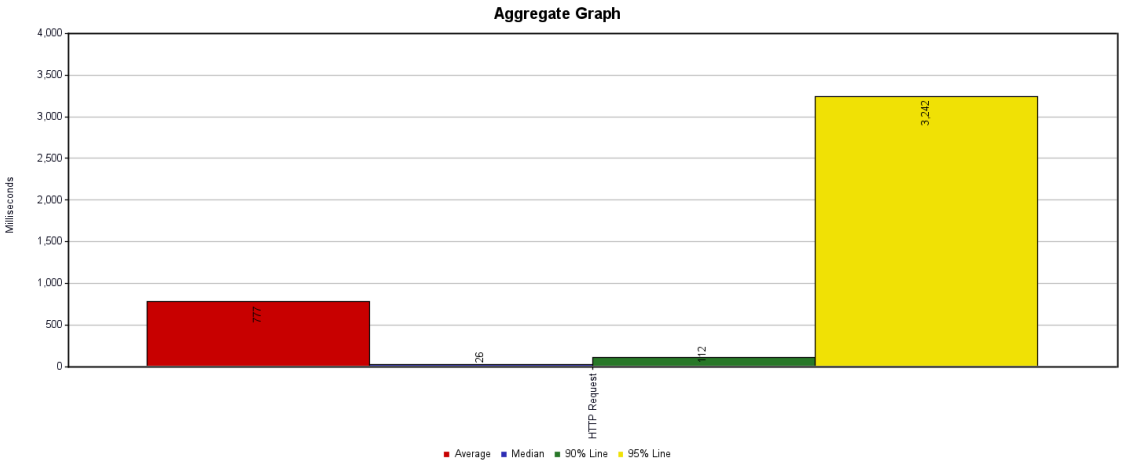

Рисунок 3.12 – Дані про відповіді після конфігурації

Після конфігурації 90 перцентиль став навіть нижче за середній час очікування, що можна вважати дуже гарним результатом. Виходячи з отриманих даних, можна зробити висновок про значне покращення в часі очікування і, відповідно, ефективності розробленого застосунку. Водночас треба відмітити, що відсоток помилок збільшився з 88% до 93%, що може свідчити про роботу патернів, що блокували надмірні запити. Окрім того, збільшився показник Throughput.

3.6 Аналіз результатів моделювання

Виходячи з отриманих даних, можемо відмітити, що динамічна конфігурація все ж є ефективним інструментом для покращення ефективності роботи застосунку. Так, середній час відгуку зменшився з 1388 мс до 777 мс (на 72%), 95 перцентиль зменшився з 5468 до 3242 (на 68%). Ці дані свідчать про значне зменшення часу очікування, що призводить до покращення ефективності роботи застосунку. В той же час збільшення відмов на 5% свідчить про те, що встановлені ліміти були доволі жорсткими, що і призвело до завершення більшого відсотку запитів. Проте не варто відкидати і вплив самої моделі на результати експерименту. Отже, залишається ще простір для покращення методології налаштування.

Тим не менш, виграш в часі очікування є надто суттєвим, щоб можна було робити висновок про неуспішність такого налаштування. Даний результат перевершив очікування від результату та підтверджує висунуту на початку дослідження гіпотезу. Зазначимо однак, що дані результати не можна вважати остаточною відповіддю щодо ефективності патернів, оскільки не було протестовано інтегровану систему з прикладними задачами. Також хотілося б відмітити негативний фактор: в даному експерименті більшу частину часу шаблони стійкості не працювали, оскільки не створювалися умови для блокування ними вхідних запитів (наприклад, Circuit Breaker блокував запити занадто рідко, адже відмов від самого сервісу майже не надходило).

Зрештою, важливим було не стільки дізнатися результати застосування шаблону стійкості як такого, а подивитися на порівняльний результат статичного і динамічного налаштування, і результати порівняння можна назвати значущими. Це може спонукати розробників частіше звертатися до такого методу покращення стабільності і ефективності мікросервісного застосунку, особливо в умовах нестачі інших інструментів покращення продуктивності. Відмітимо зокрема, що нами було проведено ще декілька тестів поза рамками роботи для дослідження різних параметрів і виключення вірогідності впливу випадкового фактору і в більшості з них динамічна конфігурація була значно краща за статичний варіант мікросервісного застосунку.

Висновки до розділу III

В ході проведення експериментів в даному розділі було проведено моделювання навантаження на мікросервісний застосунок зі статичною та динамічною варіацією налаштування шаблонів стійкості застосунку. Перед проведенням експерименту наведено основні елементи коду класів застосунку і конфігураційних файлів. Наведено також налаштування інструментів, використаних для створення навантаження на мікросервісну архітектуру. Також було наведено ініціалізаційні дані шаблонів стійкості та методологія їх динамічної конфігурації.

В ході моделювання виконувалося навантаження застосунку до 270 користувачів зі змінною кількістю протягом 5 хвилин. Результатом даного моделювання стало значне покращення ефективності застосунку, а саме зменшення часу очікування відповіді застосунку більше ніж на 30%. Дані результати є значними і можуть вплинути на вибір технологій і методів в нових програмних застосунках. Дану методологію можна рекомендувати до використання в мікросервісних застосунках, де використовуються шаблони стійкості. Запропонована в даній роботі методологія може бути корисною в мікросервісних застосунках, де виникає потреба зменшення часу очікування і покращення стабільності застосунку.

Разом з тим варто відмітити, що не всі аспекти моделі вийшли такими, як очікувалося. Зокрема відзначимо низький відсоток відмов через роботу деяких патернів, що свідчить про недостатньо якісну конфігурацію та внутрішні обмеження фреймворку, що не дозволив збільшити навантаження на застосунок для тестування відмови мікросервісу. Також залишається великий простір для покращення застосунку, від покращення моделювання обробки запитів, до покращення взаємодії сервісів між собою. Варто також відмітити можливість дослідження даного підходу до організації шаблонів стійкості в мікросервісних застосунках з використанням інших технологій.

ВИСНОВКИ

В ході даної роботи було досліджено шаблони стійкості в мікросервісній архітектурі, проаналізувано вплив параметрів і їх динамічної конфігурації на доступність сервісів. В першому розділі проведено дослідження великого об’єму наявної за темою літератури, наведено комплексний її аналіз. Наведено приклади інших схожих робіт та їх основні висновки, основних праць у вказаному напрямку академічних досліджень і робіт, присвяченим базовим знанням у визначеній області досліджень. В другому розділі наведено базові теоретичні знання з теми і наведено опис принципу роботи використаних в роботі шаблонів стійкості. В третьому розділі було наведено основний процес моделювання і розробки застосунку, а також експеримент і його результати.

В ході дослідження було підтверджено поставлену на початку дослідження гіпотезу, яка полягала в тому, що ефективність застосунку з мікросервісною архітектурою покращюється з використанням динамічної конфігурації параметрів шаблонів стійкості, застосованих до мікросервісів. Проведений аналіз демонструє, що інтеграція патернів стійкості, що вказані в роботі, значно покращує продуктивність мікросервісних систем. Дані з моніторингових систем свідчать про ефективність адаптивного керування параметрами залежно від поточного стану системи, а отже таке рішення може бути використане в прикладних програмах. З огляду на перспективи розвитку, доцільно передбачити використання алгоритмів машинного навчання для автоматичної конфігурації, що дозволить системі ще більш гнучко реагувати на змінювані умови навколишнього середовища.

Результатом роботи стало покращення середнього та медіанного часу відгуку більше ніж на 50%, що є значним досягненням порівняно з витраченими зусиллями. Подальші дослідження можуть включати в себе дослідження застосунків з мікросервісною архітектурою з більшою інтеграцією, тестування запропонованих рішень у прикладних застосунках для більш наближеного до реального результату, розширення кількості шаблонів стійкості чи їх

параметрів, тестування більшого навантаження чи більш продуктивних системах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Терещенко О.І., Інтелектуальні сервіс-орієнтовані розподілені обчислювання. КПІ ім. Ігоря Сікорського, 2022
2. Аношин О.О., Дослідження мікросервісної архітектури. Києво-Могилянська академія , 2023
3. Новак В. В., Порівняння монолітної та мікросервісної архітектур на прикладі корпоративного застосунку. Запорізький національний університет, 2022
4. Шевченко А.Р., Дослідження шаблонів стійкості до відмов у мікросервісних архітектурах та імплементація динамічного налаштування параметрів на основі ретроспективних даних. Харківський національний університет ім. Каразіна, 2024
5. Bulkhead pattern - Azure Architecture Center | Microsoft Learn. Learn Microsoft, https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead. Дата звернення: 23.05.2025.
6. Crusoveanu, Loredana. “Guide to Resilience4j.” Baeldung, https://www.baeldung.com/resilience4j. Дата звернення: 26.05.2025.
7. “Guide to Microservices Resilience Patterns.” JRebel, https://www.jrebel.com/blog/microservices-resilience-patterns. Дата звернення: 21.05.2025.
8. Jain, Sandeep. “What is Circuit Breaker Pattern in Microservices?” GeeksforGeeks, https://www.geeksforgeeks.org/what-is-circuit-breaker-pattern-in-microservices/. Дата звернення: 23.05.2025.
9. Malki, Amine, et al. Impact of API Rate Limit on Reliability of Microservices-Based Architectures. Universität Wien
10. Mendonça, Nabor, et al. Model-based analysis of microservice resiliency patterns. University of York, 2020.

11. Nadel, Ben. Release It! Design And Deploy Production-Ready Software By Michael T. Nygard. Ben Nadel, https://www.bennadel.com/blog/3162-release-it-design-and-deploy-production-ready-software-by-michael-t-nygard.htm. Дата звернення: 20.05.2025
12. Sedghpour, Saleh. AnEmpirical Study of Service Mesh Traffic Management Policies for Microservices. Diva, 2022.
13. Selvaraj, Vinoth. “Resilient Microservice Design – Bulkhead Pattern.” DZone, https://dzone.com/articles/resilient-microservices-pattern-bulkhead-pattern. Дата звернення: 23.05.2025.
14. Shah, Samarth. Dynamic Rate Limiting in Microservices: A User Behavior Aware Approach. International Journal of Research and Analytical Reviews (IJRAR.ORG).
15. Song, Alan. Characterizing Retry Policies for Microservice Applications. MIT Mathematic, 2024.
16. Tiwari, Abhishek. The Hidden Cost of Success: Understanding Non-Fatal Errors in Microservices.. Abhishek Tiwari, 2024.
17. Bondar O., Lisovychenko O. DATA ANALYSIS USING MICROSERVICES TO SOLVE FORECASTING PROBLEM, Міжвідомчийнауково-технічнийзбірник«Адаптивнісистемиавтоматичного управління» No2’ (39). 2021