МАГІСТЕРСЬКА РОБОТА

МР. ІПмз - 02.00.00.000 ПЗ

Група ІПмз-24-1

Зварич Артур

2025

Івано-Франківський національний технічний університет нафти і газу

Інститут післядипломної освіти

Кафедра інженерії програмного забезпечення

Зварич Артур Вікторович

(прізвище, ім'я, по батькові)

УДК 004.9

(індекс)

МАГІСТЕРСЬКА РОБОТА

Методи підвищення ефективності Java-систем шляхом оптимізації ресурсів, процесів та загальної масштабованості

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Зварич А.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Шекета Василь Іванович, д.т.н., професор

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

асист. Ваврик Т.О.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут післядипломної освіти

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**

доц. В.В. Бандура

“ 04 ” вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Зваричу Артуру Вікторовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Методи підвищення ефективності Java-систем шляхом оптимізації ресурсів, процесів та загальної масштабованості ”

керівник проекту (роботи) Шекета В.І., професор

затверджені наказом закладу вищої освіти від “ 03 ” листопада 2025 р. № 198/12

2. Строк подання студентом проекту (роботи) 15 грудня 2025 р.

3. Вихідні дані до проекту (роботи) Формальні моделі і методи побудови Java-систем шляхом оптимізації ресурсів, процесів та масштабованості

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз предметної області підвищення ефективності java-систем шляхом оптимізації ресурсів
2. Представлення методології тестування ефективності та продуктивності java-додатків
3. Архітектура та реалізація фреймворку тестування продуктивності java-додатків
4. Імплементація методів оптимізації ресурсів для підвищення ефективності java-систем

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Приклад генераційної купи (Generational Heap) (рис. 1.1)
2. Приклад балансування навантаження (рис. 1.2)
3. Класифікація методів прогнозування (рис. 1.3)
4. Приклад ілюстрації ковзного середнього (рис. 1.4)
5. Приклад експоненційного згладжування (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата

7. Дата видачі завдання 04 вересня 2025 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі дослідження	19.09.2025	виконано
2	Аналіз предметної області підвищення ефективності java-систем шляхом оптимізації ресурсів	01.10.2025	виконано
3	Представлення методології тестування ефективності та продуктивності java-додатків	17.10.2025	виконано
4	Архітектура та реалізація фреймворку тестування продуктивності java-додатків	02.11.2025	виконано
5	Імплементація методів оптимізації ресурсів для підвищення ефективності java-систем	19.11.2025	виконано
6	Розробка методології побудови фреймворку тестування ефективності та продуктивності java-додатків	30.11.2025	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	16.12.2025	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 78 с., 23 рис., 4 табл., 38 джерел.

Тема: Методи підвищення ефективності Java-систем шляхом оптимізації ресурсів, процесів та загальної масштабованості

Метою магістерської роботи є розробка та обґрунтування методів підвищення ефективності Java-систем шляхом оптимізації використання ресурсів, процесів діагностики та забезпечення масштабованості на основі адаптивних і багатоагентних підходів.

Об'єктом дослідження є процеси функціонування та забезпечення продуктивності розподілених і кластерних Java-систем на базі віртуальної машини Java.

Предметом дослідження є методи, моделі та програмні засоби оптимізації ресурсів, діагностики продуктивності й масштабованості Java-систем у високонавантажених середовищах.

Результати дослідження

В роботі рапропонована концептуальна структура фреймворку передбачає використання багатоагентної архітектури, що забезпечує децентралізований збір, агрегацію та аналіз продуктивнісних метрик

Висновок

Досліджено проблеме забезпечення високої продуктивності та ефективного використання обчислювальних ресурсів у сучасних розподілених Java-системах, що функціонують у середовищах з динамічним навантаженням та підвищеними вимогами до масштабованості й надійності.

JAVA-СИСТЕМИ, ПРОДУКТИВНІСТЬ, ОПТИМІЗАЦІЯ РЕСУРСІВ, JVM, ЗБІР СМІТТЯ, РОЗПОДІЛЕНІ СИСТЕМИ, КЛАСТЕРНІ ОБЧИСЛЕННЯ, МАСШТАБОВАНІСТЬ, ДІАГНОСТИКА ПРОДУКТИВНОСТІ, ФРЕЙМВОРК ТЕСТУВАННЯ.

ABSTRACT

Master Thesis: 78 pp., 23 fig., 4 tab., 38 sources.

Topic: Methods for increasing the efficiency of Java systems by optimizing resources, processes and overall scalability

The purpose of the master's thesis is to develop and substantiate methods for increasing the efficiency of Java systems by optimizing the use of resources, diagnostic processes and ensuring scalability based on adaptive and multi-agent approaches.

The object of the study is the processes of functioning and ensuring the productivity of distributed and clustered Java systems based on the Java virtual machine.

The subject of the study is methods, models and software tools for optimizing resources, diagnosing the productivity and scalability of Java systems in high-load environments.

Research results

The conceptual structure of the framework proposed in the work involves the use of a multi-agent architecture that provides decentralized collection, aggregation and analysis of performance metrics

Conclusion

The problem of ensuring high productivity and effective use of computing resources in modern distributed Java systems operating in environments with dynamic load and increased requirements for scalability and reliability has been studied.

JAVA SYSTEMS, PRODUCTIVITY, RESOURCE OPTIMIZATION, JVM, GARBAGE COLLECTION, DISTRIBUTED SYSTEMS, CLUSTER COMPUTING, SCALABILITY, PERFORMANCE DIAGNOSTICS, TESTING FRAMEWORK.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 10
ВСТУП 11

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ JAVA-СИСТЕМ ШЛЯХОМ ОПТИМІЗАЦІЇ РЕСУРСІВ 15
1.1. Комплексний підхід до оптимізації продуктивності кластерних систем на базі JVM 15
1.2. Виклики діагностики та балансування навантаження в контексті продуктивність розподілених систем 16
1.2.1. Обмеження сучасних інструментів діагностики 17
1.2.2. Балансування навантаження та управління пам’яттю 18
1.3. Фундаментальні аспекти віртуальної машини Java та механізмів збору сміття для інженерії продуктивності 19
1.3.1. Віртуальна машина Java (JVM) 19
1.3.2. Функціональність збору сміття (Garbage Collection) 20
1.3.3 Стратегії збору сміття 22
1.4. Методологічний інструментарій для кластерних систем: балансування навантаження та кількісні прогностичні моделі 24
1.4.1. Балансування навантаження 24
1.4.2. Прогностичні техніки 25
1.4.3. Автономні обчислення 29
Висновки до розділу 31

РОЗДІЛ 2. ПРЕДСТАВЛЕННЯ МЕТОДОЛОГІЇ ТЕСТУВАННЯ ЕФЕКТИВНОСТІ ТА ПРОДУКТИВНОСТІ JAVA-ДОДАТКІВ 33
2.1. Дослідження інструментарію діагностики продуктивності та бенчмарки 33

2.1.1. Інструменти діагностики продуктивності Java 33
2.1.2. Java-бенчмарки 34
2.2. Представлення методології архітектури фреймворку тестування ефективності та продуктивності Java-додатків 37
2.2.1. Опис архітектури фреймворку 37
2.2.2. Концептуальна структура фреймворку 40
2.3. Основний процес функціонування фреймворку 42
2.4. Архітектура та реалізація фреймворку тестування ефективності та продуктивності java-додатків 45
2.4.1. Багатоагентна архітектура 45
2.4.2. Внутрішня структура компонентів 46
2.4.3. Протокол взаємодії агентів 48
Висновки до розділу 49

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА ПОЛІТИК ОПТИМІЗАЦІЇ РЕСУРСІВ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ JAVA-СИСТЕМ 50
3.1. Методика автоматизованого налаштування інтервалу вибірки на основі впливу на продуктивність 50
3.2. Політика розподілу ресурсів інструменту діагностики через адаптивне регулювання інтервалу завантаження 52
3.3. Політика багаторівневої консолідації діагностичних звітів для кластерних систем 55
3.4. Аналітичні допоміжні засоби даних фреймворку тестування ефективності та продуктивності java-додатків 58
3.4.1. Оцінки подібності 58
3.4.2. Стилі серйозності для виявлення проблеми продуктивності 59
3.4.3. Оцінки Go No-Go 60
3.5. Експериментальна оцінка фреймворку на основі компромісу точності 61

3.6. Розробка методології побудови фреймворку тестування ефективності та продуктивності java-додатків 67
Висновки до розділу 70

ВИСНОВКИ 72
ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 75

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ФТЕПД - фреймворк тестування ефективності та продуктивності Java-додатків

MAPE-K - Monitor, Analyze, Plan, Execute – Knowledge

SI - Sampling Interval

UI - Uploading Interval

RT - Response Time

RT_{AVG} - Average Response Time

RT_{BL} - Baseline Response Time

$RT_{Threshold}$ - Response Time Threshold

U_{MAX} - Maximum Utilization Target

RES_{AVG} - Average Resource Utilization

$DRES_{AVG}$ - Average Resource Utilization Duration

CV - Coefficient of Variation

TPS - Transactions Per Second

EMAT - Eclipse Memory Analyser Tool (або Analyser)

GCLITE - IBM Garbage Collection Lite

GCMV - IBM Garbage Collection and Memory Visualiser

HC - IBM Health Center

WAIT - IBM Whole Analysis Idle Time

ВСТУП

Актуальність теми.

Стрімкий розвиток інформаційних технологій, зростання обсягів оброблюваних даних та поширення розподілених і хмарних архітектур зумовлюють підвищені вимоги до продуктивності, масштабованості та надійності програмних систем. Значна частина корпоративних і науково-технічних рішень реалізується з використанням платформи Java, яка завдяки універсальності, розвиненій екосистемі та підтримці багатопоточності залишається одним із домінуючих інструментів для побудови серверних і кластерних застосунків. Водночас складність внутрішніх механізмів віртуальної машини Java, особливості управління пам’яттю та процесами виконання, а також динамічний характер навантаження в сучасних системах ускладнюють забезпечення стабільної та прогнозованої продуктивності.

У контексті високонавантажених і масштабованих Java-систем особливої актуальності набувають задачі оптимізації використання обчислювальних ресурсів, ефективної діагностики продуктивності та балансування навантаження. Традиційні підходи до моніторингу й тестування часто супроводжуються значними накладними витратами та не враховують специфіку кластерних і розподілених середовищ. Це зумовлює необхідність розробки нових методів і інструментів, здатних адаптивно реагувати на зміни стану системи та забезпечувати компроміс між точністю діагностики й ефективністю виконання.

Дана магістерська робота присвячена дослідженню та розробці методів підвищення ефективності Java-систем шляхом оптимізації ресурсів, процесів та загальної масштабованості. У роботі поєднано теоретичний аналіз фундаментальних аспектів JVM і практичну реалізацію фреймворку тестування продуктивності, що орієнтований на використання в сучасних розподілених середовищах.

Актуальність даного дослідження зумовлена широким використанням Java-систем у критично важливих інформаційних інфраструктурах, де навіть незначні втрати продуктивності можуть призводити до істотних економічних і функціональних наслідків. Умови експлуатації сучасних програмних систем характеризуються нерівномірним і динамічним навантаженням, що ускладнює статичне налаштування параметрів виконання та вимагає застосування адаптивних підходів до управління ресурсами.

Складність внутрішніх механізмів JVM, зокрема стратегій збору сміття та управління потоками, створює додаткові труднощі для інженерії продуктивності та діагностики вузьких місць. Існуючі інструменти моніторингу часто орієнтовані на окремі вузли та не забезпечують повноцінної підтримки кластерних середовищ, що обмежує можливості комплексного аналізу. Крім того, надмірний обсяг діагностичних даних і накладні витрати на їхній збір можуть негативно впливати на продуктивність системи, що досліджується.

У зв’язку з цим виникає потреба в розробці методів оптимізації, які поєднують ефективне управління ресурсами, адаптивну діагностику та аналітичну підтримку прийняття рішень. Запропоновані в роботі підходи спрямовані на подолання зазначених обмежень і є актуальними як з наукової, так і з практичної точки зору.

Метою магістерської роботи є розробка та обґрунтування методів підвищення ефективності Java-систем шляхом оптимізації використання ресурсів, процесів діагностики та забезпечення масштабованості на основі адаптивних і багатоагентних підходів.

Об’єктом дослідження є процеси функціонування та забезпечення продуктивності розподілених і кластерних Java-систем на базі віртуальної машини Java.

Предметом дослідження є методи, моделі та програмні засоби оптимізації ресурсів, діагностики продуктивності й масштабованості Java-систем у високонавантажених середовищах.

Завдання дослідження

Для досягнення поставленої мети у роботі вирішено такі завдання:

1. Проаналізувати сучасні підходи до оптимізації продуктивності Java-систем і кластерних архітектур.

2. Дослідити внутрішні механізми JVM та стратегії збору сміття з позиції їхнього впливу на продуктивність.

3. Оцінити можливості й обмеження наявних інструментів діагностики та бенчмаркінгу Java-додатків.

4. Розробити архітектуру фреймворку тестування ефективності та продуктивності Java-систем.

5. Реалізувати політики консолідації діагностичних даних у кластерних середовищах.

6. Провести експериментальну оцінку ефективності запропонованих методів і фреймворку.

Методи дослідження

У роботі використано методи системного аналізу для дослідження предметної області, методи порівняльного аналізу для оцінювання інструментів діагностики, методи моделювання та проєктування програмних систем для розробки архітектури фреймворку. Застосовано експериментальні методи для оцінювання продуктивності та ефективності запропонованих рішень, а також елементи статистичного аналізу для інтерпретації отриманих результатів.

Наукова новизна отриманих результатів

Наукова новизна роботи полягає у розробці адаптивних методів оптимізації діагностики продуктивності Java-систем, що враховують динамічний вплив інструментів моніторингу на продуктивність. Запропоновано багатоагентний підхід до організації фреймворку тестування ефективності, який забезпечує масштабованість і децентралізований аналіз у кластерних середовищах. Удосконалено методи консолідації діагностичних

даних шляхом багаторівневої агрегації та використання аналітичних оцінок для підтримки прийняття рішень.

Практичне застосування результатів

Практична цінність отриманих результатів полягає в можливості використання розробленого фреймворку та запропонованих методів під час проєктування, тестування й експлуатації високонавантажених Java-додатків. Результати роботи можуть бути використані фахівцями з інженерії продуктивності, DevOps та архітекторами програмних систем для підвищення ефективності використання ресурсів і забезпечення стабільної роботи кластерних Java-систем.

Структура магістерської роботи. Представлена магістерська робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 78 сторінок, і містить 23 рисунки, 4 таблиці, перелік використаних джерел із 38 найменувань.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ JAVA-СИСТЕМ ШЛЯХОМ ОПТИМІЗАЦІЇ РЕСУРСІВ

1.1. Комплексний підхід до оптимізації продуктивності кластерних систем на базі JVM

Кластерні архітектури широко імплементуються в корпоративних інформаційних системах з метою досягнення підвищеної швидкодії та збільшення пропускної здатності порівняно з монолітними однокористувацькими конфігураціями. Однак перехід від монолітної парадигми до розподіленої суттєво ускладнює процеси, пов'язані з оптимізацією продуктивності таких кластерних середовищ. Отже, виникає потреба в автоматизованих методиках для полегшення цих завдань з підвищення ефективності, які в іншому випадку є схильними до помилок та вимагають значних трудових ресурсів.

Ця робота присвячена внеску у сферу оптимізації продуктивності кластерних систем, побудованих на базі Java (домінуючої технології в корпоративному сегменті), зокрема у великомасштабних середовищах. У дисертації представлено дві оригінальні методики, спрямовані на вирішення критичних проблем: ефективне виявлення проблем продуктивності, залежних від навантаження, та мінімізація впливу великомасштабного збору сміття (Major Garbage Collection) на загальну ефективність. Ці проблеми є типовими для кластерних Java-систем, які функціонують у середовищах з високим рівнем масштабування.

Зокрема, в роботі пропонується адаптивна архітектура (каркас) для автоматизації застосування діагностичних інструментів продуктивності під час тестування кластерних систем. Мета цього підходу полягає у спрощенні процесу ідентифікації проблем продуктивності шляхом зниження необхідних зусиль та рівня експертних знань, необхідних для ефективного використання

таких інструментів. Крім того, розроблено адаптивну стратегію балансування навантаження, чутливу до збору сміття (GC-aware adaptive load balancing strategy), яка використовує прогнозування великомасштабного збору сміття для прийняття обґрунтованих рішень щодо оптимального розподілу навантаження між доступними вузлами. Ця стратегія спрямована на покращення загальної продуктивності кластерної системи шляхом уникнення негативного впливу на ефективність кластера, спричиненого великомасштабним збором сміття, що відбувається на окремих вузлах.

Експериментальні результати, отримані шляхом застосування цих методик до набору реальних додатків, представлені в роботі, демонструючи значні переваги, які ці підходи забезпечують для кластерних Java-систем.

1.2. Виклики діагностики та балансування навантаження в контексті продуктивність розподілених систем

Продуктивність є критичною метрикою якості та одним з основних пріоритетів у розробці будь-якого програмного забезпечення. Незважаючи на це, проблеми з продуктивністю часто виникають і можуть призводити до значних негативних наслідків, таких як простої в робочих середовищах або навіть припинення програмних проєктів. Наприклад, дослідження серед керівників IT-галузі [5] показало, що 50% респондентів стикалися з проблемами продуктивності щонайменше у 20% розгорнутих додатків. Така ситуація частково пояснюється всеосяжним характером продуктивності, оцінка якої є складною, оскільки вона залежить від практично кожного аспекту архітектури, програмного коду та середовища виконання додатка.

Останніми роками кластерні обчислення набули широкого визнання як потужне та економічно ефективне рішення для паралельної та розподіленої обробки даних [3]. Отже, використання кластерів стало повсюдним: сучасні високодоступні системи та корпоративні додатки, які вимагають як низької затримки, так і високої пропускної здатності на постійній основі, часто

розгортаються в кластерних конфігураціях для задоволення цих суворих вимог до продуктивності. Проте цей перехід від монолітної архітектури до розподіленої також призвів до збільшення складності додатків, що додатково ускладнює всі види діяльності, пов’язані з оптимізацією продуктивності.

1.2.1. Обмеження сучасних інструментів діагностики

Особливою задокументованою проблемою [6] є значна залежність сучасних інструментів діагностики продуктивності від людських експертів для їх коректного налаштування та інтерпретації вихідних даних. Крім того, діагностика проблем продуктивності, особливо у високорозподілених середовищах, часто вимагає консолідації інформації з багатьох джерел. Наприклад, у середовищі Java тестувальнику необхідно аналізувати та співвідносити дампи потоків, журнали збору сміття (GC), дампи купи, а також показники використання центрального процесора (CPU) та оперативної пам’яті. Ця мультиджерельна вимога підвищує рівень експертних знань, необхідних для аналізу, якими зазвичай володіє лише обмежена кількість фахівців в організації. Як наслідок, це може призводити до "вузьких місць", коли певні критичні дії можуть виконуватися лише цими експертами, що негативно впливає на ефективність тестових команд.

Для спрощення процесів аналізу та діагностики продуктивності дослідники розробили інструменти з вбудованою експертизою. Однак ці діагностичні інструменти мають обмеження, які перешкоджають їх ефективному застосуванню у високорозподілених середовищах.

Ручна конфігурація та точність. По-перше, ці інструменти все ще потребують ручного налаштування чутливих параметрів. Неправильна конфігурація може негативно вплинути на точність результатів, що може призвести до значних часових витрат через неможливість отримання бажаних діагностичних даних.

Складність збору даних. По-друге, використання цих інструментів вимагає ручного збору даних. У кластерному середовищі, де необхідно

контролювати та координувати численні вузли, цей ручний процес є вкрай трудомістким та схильним до помилок через великий обсяг даних, які потребують збору та консолідації. У сценаріях довготривалого тестування продуктивності таке ручне використання діагностичних інструментів стає ще складнішим через необхідність періодичного збору даних.

Надмірний обсяг вихідних даних, що генеруються інструментами, може перевантажувати тестувальника через час, необхідний для кореляції та аналізу результатів. Ця проблема виникає внаслідок генерації кількох звітів для кожного контрольованого вузла додатка, інформація з яких потребує ручної кореляції та аналізу.

1.2.2. Балансування навантаження та управління пам'яттю

Навіть після усунення найкритичніших дефектів продуктивності перед випуском додатка в робоче середовище, інші фактори можуть погіршувати його продуктивність. Серед них, ефективний розподіл навантаження між доступними кластерними екземплярами є ключовою проблемою в кластерних обчисленнях (оскільки незбалансоване навантаження може призвести до неефективності обробки). Для вирішення цього виклику численні дослідження зосереджувалися на розробці більш ефективних алгоритмів та стратегій балансування навантаження, заснованих на різних критеріях та евристиках.

В контексті корпоративного сегменту, де домінує технологія Java, особливим джерелом занепокоєння щодо продуктивності є збір сміття (Garbage Collection, GC) [7]. Хоча GC є фундаментальною функцією Java, що автоматизує більшість завдань управління пам'яттю, він має свою ціну: при активації GC впливає на продуктивність системи, призупиняючи задіяні додатки. Хоча паузи тривалістю в кілька мілісекунд зазвичай є прийнятними, більш тривалі паузи GC можуть серйозно вплинути на продуктивність системи, негативно впливаючи на бізнес-функції та загальний досвід користувача. Це особливо критично для додатків, які вимагають швидкої

реакції або високої пропускної здатності. Крім того, ця проблема загострюється у випадку основного збору сміття (Major Garbage Collection, MaGC), який, як правило, спричиняє найтриваліші паузи GC.

1.3. Фундаментальні аспекти віртуальної машини Java та механізмів збору сміття для інженерії продуктивності

Цей підрозділ представляє ключові функціональні можливості та характеристики технології Java, концепції автономних обчислень (хоча вона не була розгорнута в повному обсязі, доречно включити її, якщо вона критична для автоматизованих технік), а також типові процеси балансування навантаження та тестування продуктивності. Ці елементи є фундаментальними для коректного розуміння подальших розділів. Додатково, у цьому підрозділі описуються прогностичні методики, інструменти діагностики продуктивності та набори Java-бенчмарків, використані в цьому дослідженні.

1.3.1. Віртуальна машина Java (JVM)

Віртуальна машина Java (JVM) є середовищем виконання, на якому функціонують додатки, розроблені на Java. Вона створює ізольоване середовище, яке імітує фізичний комп'ютер. Це дозволяє програмам працювати однаково, незалежно від апаратної архітектури та операційної системи хост-машини. Її основна роль полягає в інтерпретації бінарного формату (байт-коду), в який компілюється Java-програма.

Завдяки доступності JVM для більшості сучасних операційних систем, скомпільована Java-програма демонструє високий рівень портативності. Крім того, JVM дозволяє конфігурацію під час запуску через численні параметри, включаючи ті, що стосуються управління пам'яттю.

У таблиці 1.1 наведено опис деяких найбільш поширених параметрів управління пам'яттю JVM.

Таблиця 1.1.

Параметри JVM, пов’язані з конфігурацією пам’яті.

Параметр JVM	**Опис**
`-XX:+DisableExplicitGC`	Вимкнення явних запитів на виконання збору сміття (GC).
`-XX:PermSize`	Встановлення початкового розміру постійної генерації (PermGen).
`-XX:MaxPermSize`	Визначення максимального розміру постійної генерації (PermGen).
`-Xms`	Встановлення початкового розміру купи (heap).
`-Xmx`	Визначення максимального розміру купи (heap).

1.3.2. Функціональність збору сміття (Garbage Collection)

Збір сміття (GC) є ключовою функціональністю Java. Ця форма автоматичного управління пам’яттю забезпечує значні переваги в інженерії програмного забезпечення порівняно з явним управлінням пам’яттю. Зокрема, вона звільняє розробників від необхідності ручного управління пам'яттю, запобігаючи поширеним джерелам перезапису пам'яті та витоків [6], а також підвищуючи продуктивність розробників [9]. Незважаючи на ці переваги, GC загальновизнано має накладні витрати на продуктивність. Наприклад, розробка специфікації реального часу для Java (RTSJ) [10], яка дозволяє програмам повністю оминати GC для забезпечення можливостей реального часу (за рахунок значної модифікації вихідного коду та втрати переваг автоматичного управління пам'ятю), є якісним підтвердженням цього впливу.

Слід зазначити, що програмно неможливо примусово ініціювати виконання GC [11]. Найближчою дією, доступною розробнику, є виклик методу `Runtime.getRuntime().gc()` (або еквівалентного `System.gc()`), який лише пропонує JVM виконати основний збір сміття (MaGC). Однак JVM не зобов’язана виконувати цей запит і може його ігнорувати. Використання цих методів не рекомендується постачальниками JVM [14], оскільки JVM

зазвичай краще вирішує оптимальний час для виконання GC, що є однією з основних мотивацій для використання автоматичного управління пам’яттю.

Область пам’яті Java, відома як купа (heap). Сьогодні одним з найбільш поширених типів купи є поколіннєва купа (generational heap) [15], де об’єкти сегрегуються за їхнім віком у області пам’яті, відомі як покоління (зазвичай обмежені двома або трьома).

Нові об’єкти створюються в наймолодшому поколінні (Young Generation), оскільки рівень виживання молодших об'єктів зазвичай нижчий, ніж старших. Це означає, що молодші покоління частіше містять сміття, що дозволяє частіше виконувати збір сміття в них. Збір сміття в молодших поколіннях називається lрібний GC (Minor GC, MiGC). Він, як правило, недорогий і рідко становить серйозну проблему для продуктивності. MiGC також відповідає за переміщення (підвищення, promotion) живих об’єктів, які досягли достатнього віку, до старшого покоління. Таким чином, MiGC відіграє ключову роль у розподілі пам’яті старших поколінь. Збір сміття в старших поколіннях відомий як Основний GC (Major GC, MaGC) і загальновизнаний як найбільш дорогий тип GC через його значний вплив на продуктивність [17]. Нестача вільної пам’яті в будь-якому поколінні ініціює відповідну подію GC. Приклад такої структури купи зображений на рисунку 1.1, який ілюструє поколіннєву купу Oracle JVM HotSpot 7.

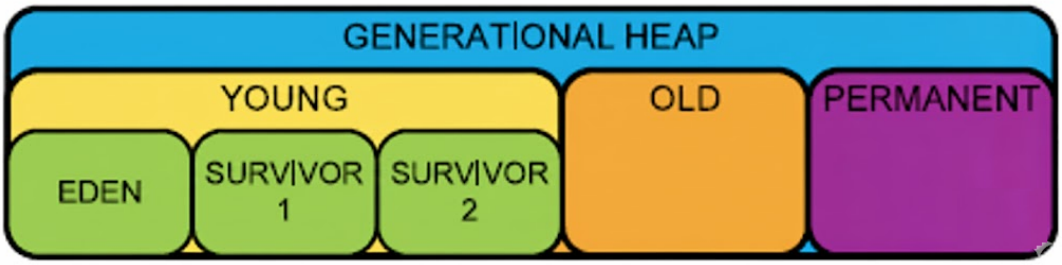

Рис. 1.1. Приклад генераційної купи (Generational Heap)

У цій конфігурації молодше покоління (YoungGen) складається з трьох підобластей: одна область Eden, де створюються нові об’єкти, і дві області

виживання (Survivor), куди MiGC ітеративно переміщує живі об’єкти, доки вони не досягнуть віку, необхідного для переміщення до старшого покоління (OldGen). Нарешті, постійне покоління (PermGen) — це спеціалізована область, де JVM зберігає внутрішні дані, такі як визначення класів. Вичерпання пам'яті в PermGen також може спричинити MaGC.

1.3.3 Стратегії збору сміття

Купа керується певною стратегією GC, обраною під час запуску JVM. Їхня доступність, як правило, пов'язана з типом купи. Наприклад, три найбільш широко використовувані стратегії GC у промисловості [20] функціонують виключно на поколіннєвих купах:

1. Послідовний GC (Serial GC): Виконує всю роботу одним потоком. Рекомендований для клієнтських JVM.

2. Паралельний GC (Parallel GC): Використовує кілька потоків. Рекомендований для серверних JVM, де загальна пропускна здатність має пріоритет над часом відгуку.

3. Конкурентний GC (Concurrent GC): Виконує більшу частину роботи паралельно з потоками додатка. Рекомендований для серверних JVM, де час відгуку має пріоритет над загальною пропускною здатністю.

Кожна стратегія GC включає два алгоритми: один застосовується до молодшого покоління, а інший — до старшого покоління. Таблиця 1.2 підсумовує ці алгоритми.

Таблиця 1.2.

Алгоритми GC за стратегіями

Стратегія	**Алгоритм для YoungGen**	**Алгоритм для OldGen**
Послідовний (Serial)	Копіювання	Позначення-Очищення-Ущільнення
Паралельний (Parallel)	PS Scavenge	PS Mark Sweep
Конкурентний (Concurrent)	PS Scavenge	Concurrent Mark Sweep

Послідовний GC використовує алгоритм копіювання для YoungGen і позначення-очищення-ущільнення (Mark Sweep Compact) для OldGen [17].

Алгоритм копіювання працює, перевіряючи вік живих об’єктів. Об'єкти, які досягли необхідного віку, копіюються до OldGen. Інші живі об'єкти копіюються до невикористаного простору виживання. Якщо цей простір заповнюється, об'єкти, які перевищили його ємність, також копіюються до OldGen.

Алгоритм Позначення-Очищення-Ущільнення: Складається з трьох фаз:

- Позначення (Mark) - визначаються живі об’єкти.
- Очищення (Sweep) - визначається сміття.
- Ущільнення (Compact) - алгоритм зсуває живі об’єкти до початку простору OldGen, консолідуючи вільний простір в один суцільний блок для майбутніх розміщень.

Паралельний GC використовує алгоритм PS Scavenge для YoungGen та PS Mark Sweep для OldGen [17]. Ці алгоритми є паралельними версіями алгоритмів, що застосовуються послідовним GC. Вони залишаються алгоритмами типу "стоп-світ" (stop-the-world, STW), але виконують більшість своїх операцій паралельно, використовуючи кілька ядер CPU.

Конкурентний GC використовує алгоритм PS Scavenge у YoungGen (аналогічно паралельному GC) та Concurrent Mark Sweep (CMS) для OldGen [17]. Алгоритм CMS виконує більшу частину своєї роботи конкурентно з виконанням додатка. Його робота поділяється на фази:

1. Початкове позначення - коротка STW-пауза для визначення набору живих об’єктів, безпосередньо доступних з кореневих посилань додатка.
2. Конкурентне позначення - позначаються всі живі об’єкти, транзитивно доступні з початкового набору. Ця фаза виконується одночасно з роботою додатка.
3. Повторне позначення - друга STW-пауза для фіналізації позначення шляхом повторного відвідування об’єктів, які були змінені додатком під час

конкурентного позначення. Це необхідно для забезпечення коректного позначення всіх живих об'єктів.

4. Конкурентне очищення - остання фаза, яка звільняє виявлене сміття, також виконуючись конкурентно.

1.4. Методологічний інструментарій для кластерних систем: балансування навантаження та кількісні прогностичні моделі

Цей підрозділ деталізує основні функції та характеристики балансування навантаження, прогностичних методик, а також концептуальні основи автономних обчислень. Розуміння цих концепцій є необхідним для сприйняття внесків, представлених у подальших розділах дисертації.

1.4.1. Балансування навантаження

Метою стратегії балансування навантаження є оптимізація продуктивності додатка, розгорнутого в кластері, який складається з набору вузлів додатків (зазвичай розміщених у центрі обробки даних) з ідентичними образами коду. Кластерний додаток повинен бути розділений на атомарні завдання меншого розміру (наприклад, операції веб-додатка, такі як автентифікація, пошук, вихід тощо). Цей типовий сценарій проілюстровано на рисунку 1.2.

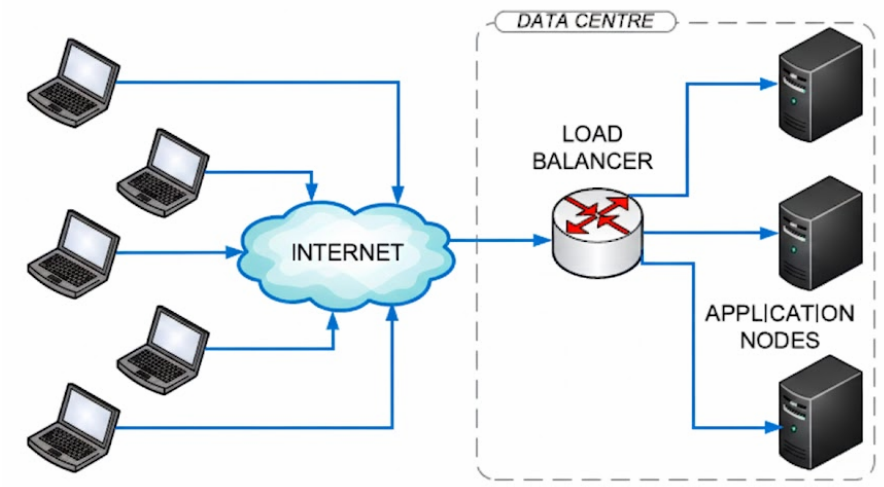

Рис. 1.2. Приклад балансування навантаження

Діапазон існуючих алгоритмів балансування навантаження є широким. В промисловості часто використовуються чотири основні алгоритми:

- Round Robin (циклічний): Вузли вибираються ітеративно, що забезпечує рівномірний розподіл навантаження між доступними вузлами.

- Випадковий (Random): Кожен вузол вибирається випадковим чином серед доступних.

- Зважений Round Robin (Weighted Round Robin): Частота вибору вузла (відповідно до логіки Round Robin) коригується за допомогою вагового коефіцієнта, присвоєного кожному вузлу.

- Зважений Випадковий (Weighted Random): Частота випадкового вибору вузла коригується за допомогою попередньо визначеного вагового коефіцієнта.

1.4.2. Прогностичні техніки

Прогностична техніка (forecasting technique) має на меті формування прогнозів щодо майбутньої події на основі аналізу доступних історичних даних та їхніх трендів. Для досягнення надійності прогнозування необхідно, щоб фактори, пов'язані з передбачуваною подією, були відомі та добре зрозумілі.

Існуючі прогностичні методики є різноманітними і класифікуються на дві основні категорії: якісні та кількісні (як показано на рисунку 1.3).

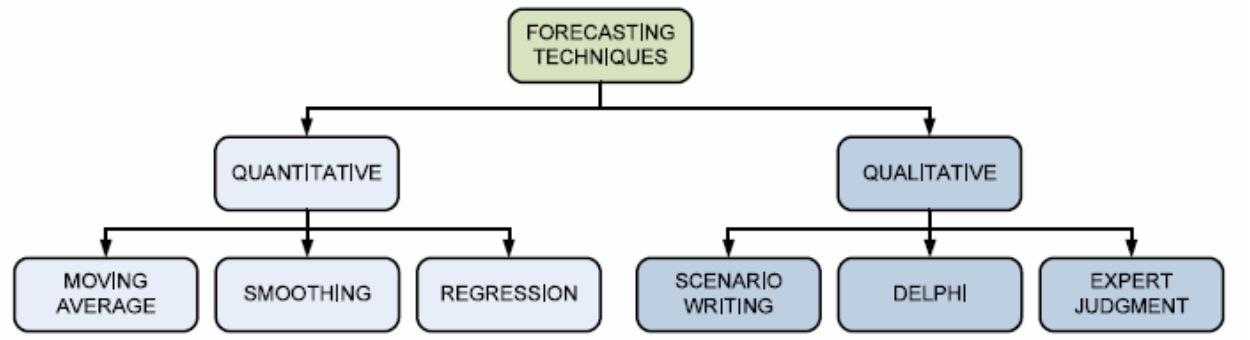

Рис. 1.3. Класифікація методів прогнозування

Якісні техніки є суб'єктивними, оскільки вони базуються на судженнях експертів у певній галузі. Ці техніки є доцільними у випадках, коли відсутні історичні числові дані. Прикладом якісної техніки є метод Дельфі, який використовує комбінацію експертних суджень та історичної інформації для генерації індивідуальних оцінок, які потім порівнюються серед учасників панелі. Цей процес ітеративно повторюється до досягнення консенсусу.

Кількісні прогностичні техніки використовуються для прогнозування майбутніх даних як функції минулих. Це досягається шляхом формального захоплення взаємозв'язків між задіяними змінними за допомогою математичних рівнянь. Ці техніки є придатними, коли наявні минулі числові дані та коли можна припустити, що закономірності у даних збережуться в майбутньому. Три найбільш поширені кількісні техніки: ковзне середнє, експоненційне згладжування та регресія.

Ковзне середнє (Moving Average) [18] - ця техніка передбачає розрахунок середнього значення часового ряду (послідовності спостережень, рівномірно розподілених у часі) за кілька послідовних періодів. Воно називається "ковзним", оскільки постійно перераховується з появою нових даних: найдавніше значення видаляється, а найновіше значення додається.

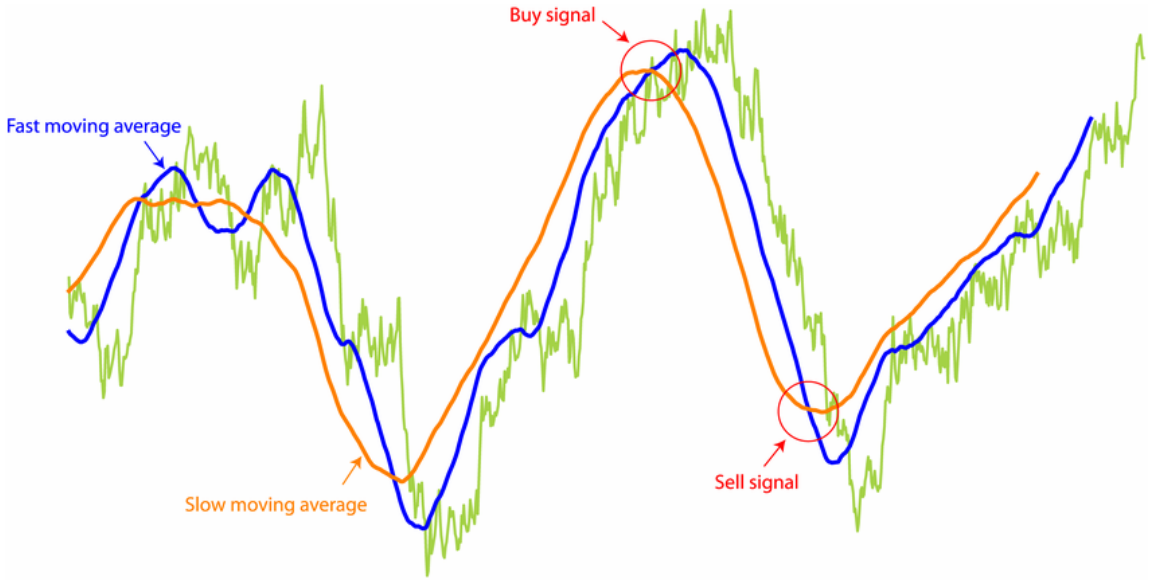

Рис. 1.4. Приклад ілюстрації ковзного середнього

Поширені варіації:

- Просте ковзне середнє (Simple Moving Average): Значення для заданого періоду часу замінюється середнім значенням, розрахованим на основі значень для певної кількості попередніх періодів.

- Наївне ковзне середнє (Naïve Moving Average): Особливий випадок простого ковзного середнього, де кількість періодів, що використовуються для згладжування, дорівнює одиниці.

- Зважене ковзне середнє (Weighted Moving Average): Значення для заданого періоду часу замінюється зваженим середнім, розрахованим на основі значень для певної кількості попередніх періодів.

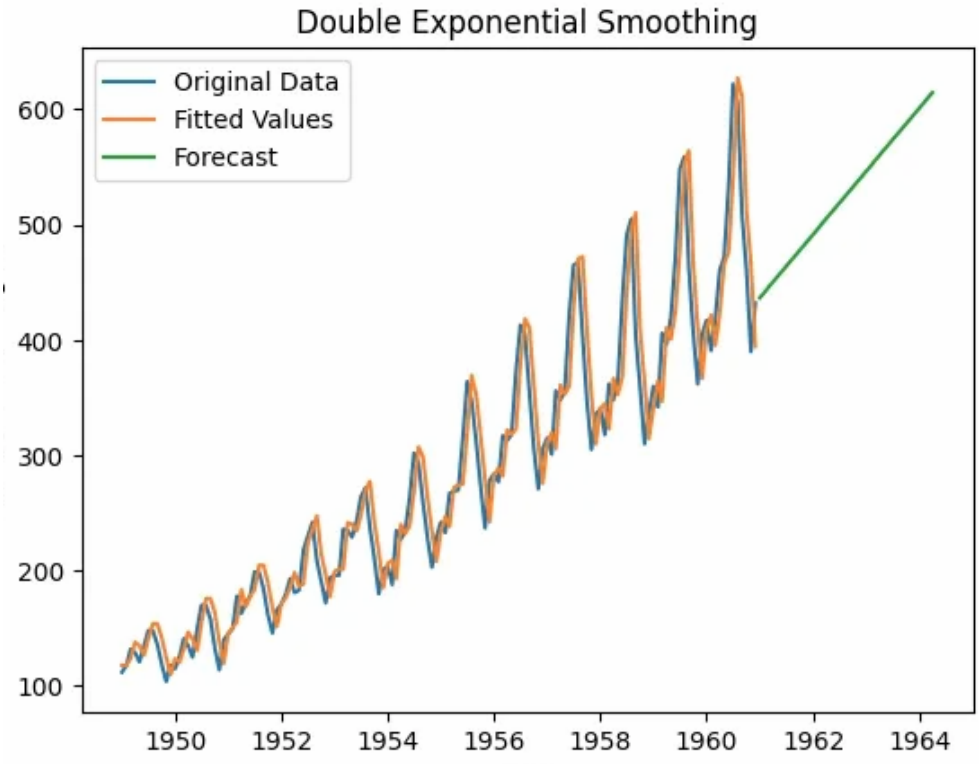

Рис. 1.5. Приклад експоненційного згладжування

Експоненційне згладжування (Exponential Smoothing) [18] - ця техніка здатна виявляти значні зміни в даних, ігноруючи нерелевантні коливання. На відміну від ковзного середнього, старим даним надається прогресивно менша відносна вага (важливість), тоді як новим даним надається прогресивно більша вага.

Поширені варіації:

Просте експоненційне згладжування (Simple Exponential Smoothing) надає минулим спостереженням експоненційно зменшувані ваги для прогнозування майбутніх значень.

Подвійне експоненційне згладжування (Double Exponential Smoothing) [18] (також відоме як метод Голта) - удосконалення простої техніки, яке додає додатковий компонент для врахування будь-яких трендів у даних.

Потрійне експоненційне згладжування (Triple Exponential Smoothing) (також відоме як метод Вінтерса) - удосконалення подвійного експоненційного згладжування, яке додає додатковий компонент для врахування будь-якої сезонності (або періодичності) у даних.

Регресія - це техніка, що допомагає оцінити значення залежної змінної на основі доступних історичних даних про набір незалежних змінних. Це досягається шляхом розробки регресійної моделі, яка складається з незалежних змінних, залежної змінної та будь-яких невідомих (постійних) параметрів.

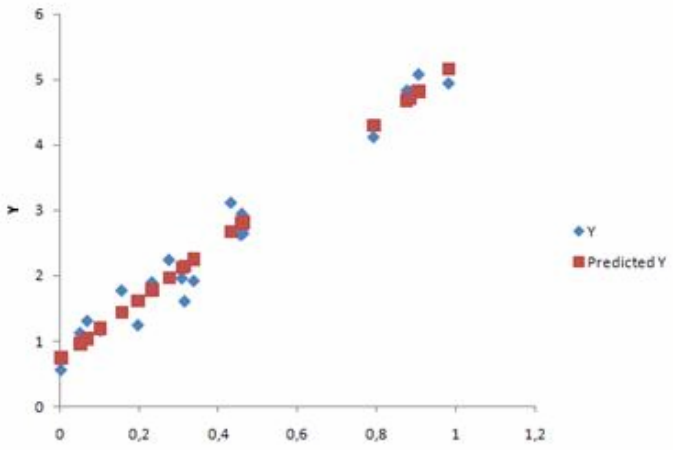

Рис. 1.6. Приклад регресії

Поширені варіації:

- Проста лінійна регресія (Simple Linear Regression) - передбачає одну незалежну змінну. Модель підбирає пряму лінію через набір даних таким чином, щоб мінімізувати суму квадратів залишків моделі.

- Множинна лінійна регресія (Multiple Linear Regression) - розширення простої лінійної регресії, що передбачає кілька незалежних змінних та/або векторних значень. Це дозволяє оцінці бути під впливом кількох типів історичних даних.

- Поліноміальна регресія (Polynomial Regression) - форма регресії, в якій взаємозв'язок між незалежною та залежною змінними моделюється як поліном n-го ступеня. Це дозволяє моделі підбирати нелінійний взаємозв'язок між змінними.

1.4.3. Автономні обчислення

Автономні обчислення [19] — це модель самоуправління, яка визначає набір характеристик, необхідних комп'ютерній системі для адаптації до непередбачуваних змін у її середовищі виконання. Її кінцевою метою є створення систем, які можуть функціонувати автономно, зберігаючи при цьому складність системи невидимою для операторів та користувачів. Ці системи мають на меті подолати швидко зростаючу складність управління комп'ютерними системами та зменшити бар'єр, який ця складність створює для подальшого зростання систем.

Модель автономних обчислень визначає чотири ключові властивості, якими повинна володіти адаптивна система:

- Самоналаштування (Self-Configuration): Адаптація до динамічно мінливих середовищ для підвищення реактивності.

- Самовідновлення (Self-Healing): Виявлення, діагностика та запобігання порушенням для досягнення стійкості бізнесу.

- Самооптимізація (Self-Optimization): Налаштування ресурсів та балансування навантажень для максимізації використання ресурсів (операційна ефективність).

- Самозахист (Self-Protection): Передбачення, виявлення та захист від атак для забезпечення безпеки інформації та ресурсів.

З архітектурної точки зору, контури управління були широко досліджені та визнані ключовим механізмом для досягнення самоадаптації в програмних системах. На їх основі було запропоновано багато рішень для створення само систем.

- Реактивний адаптивний шаблон (Reactive Adaptive Pattern) - описує механізм для створення реактивних компонентів, здатних модифікувати свою поведінку у відповідь на зовнішню подію без необхідності використання контуру управління.

- Шаблон внутрішнього зворотного зв’язку (Internal Feedback Loop Pattern) - пропонує механізм для збагачення стандартної логіки системи шляхом реалізації контуру управління, який контролює контекст її виконання, визначає необхідні зміни та втілює їх.

- Найпоширенішим адаптивним шаблоном є MAPE-K [82]. Він заснований на системі зворотного зв'язку, що використовується в теорії управління. MAPE-K складається з п'яти елементів, зображених на рисунку 1.7.

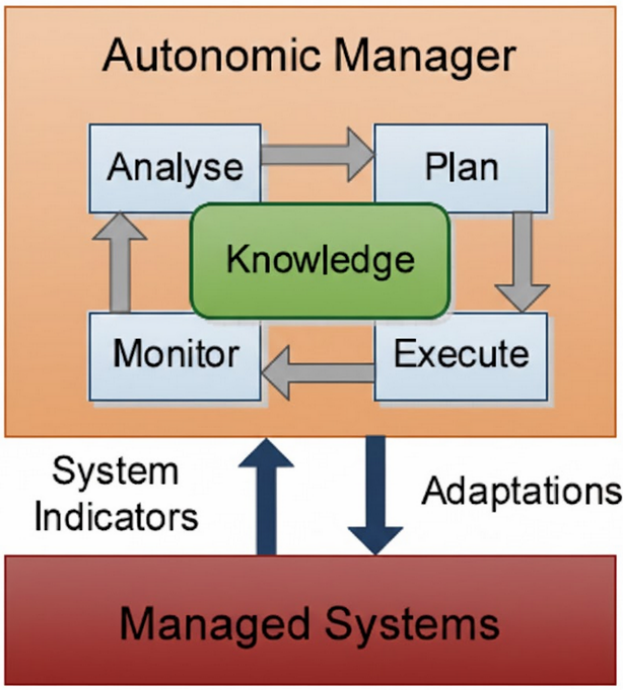

Рис. 1.7. Модель MAPE-K

- Моніторинг - збір інформації від керованих систем (через системні індикатори).

- Аналіз - оцінка, чи потрібна адаптація.

- Планування - розробка плану адаптації.

- Виконання - впровадження плану адаптації.

- Знання - елемент, що підтримує інші елементи в їхніх відповідних завданнях.

Основною перевагою MAPE-K (порівняно з внутрішнім контуром зворотного зв'язку) є те, що його зовнішній контур управління чітко відокремлений від логіки додатка.

Висновки до розділу

У першому розділі здійснено системний аналіз предметної області підвищення ефективності Java-систем у контексті оптимізації використання обчислювальних ресурсів. Розглянуто особливості функціонування кластерних систем на базі JVM та обґрунтовано необхідність комплексного підходу до інженерії продуктивності.

Проаналізовано основні виклики діагностики продуктивності розподілених систем, зокрема проблеми масштабованості моніторингу та накладного впливу інструментів спостереження. Визначено обмеження сучасних засобів діагностики, що ускладнюють отримання повної та узгодженої картини стану системи. Досліджено механізми балансування навантаження та їхній взаємозв'язок з управлінням пам'яттю в Java-середовищі.

Узагальнено фундаментальні принципи роботи віртуальної машини Java, які визначають продуктивнісні характеристики додатків. Окрему увагу приділено механізмам збору сміття та їхнім стратегіям з позиції впливу на затримки та пропускну здатність системи. Проаналізовано методологічний інструментарій балансування навантаження в кластерних середовищах.

Розглянуто застосування кількісних прогностичних моделей для оцінювання майбутнього стану системи.

Сформовано теоретичне підґрунтя для подальшої розробки методів і політик оптимізації продуктивності Java-систем.

РОЗДІЛ 2. ПРЕДСТАВЛЕННЯ МЕТОДОЛОГІЇ ТЕСТУВАННЯ ЕФЕКТИВНОСТІ ТА ПРОДУКТИВНОСТІ JAVA-ДОДАТКІВ

2.1. Дослідження інструментарію діагностики продуктивності та бенчмарки

2.1.1. Інструменти діагностики продуктивності Java

Існує широкий спектр інструментів для діагностики продуктивності Java-систем. П'ять часто використовуваних у промисловості інструментів детально описані нижче

Eclipse Memory Analyser (EMAT) [19] є діагностичним інструментом, призначеним для виявлення проблем продуктивності, пов'язаних з управлінням пам'яттю (наприклад, витоків пам'яті). EMAT використовує неінвазивні механізми вибірки, доступні у JVM, зокрема дампи купи (heapdumps) [21] – детальні знімки пам'яті JVM, що містять інформацію про використання пам'яті за типом об'єкта, посилання на об'єкти та поточні блокування. Звітність EMAT включає один HTML-звіт на кожну вибірку. Звіт містить інформацію про виявлені витоки пам'яті та кількісний опис об'єктів, класів та шляхів класів, які перебувають у пам'яті.

IBM Garbage Collection Lite (GCLITE) - це інструмент діагностики, сфокусований на виявленні проблем продуктивності Збору Сміття (GC). Він базується на механізмах трасування, доступних у JVM, у формі детальних журналів GC (GC verbose) [18], які надають інформацію про загальний розмір купи, розмір поколінь до і після кожного GC, а також витрачений час. GCLITE генерує один HTML-звіт на оброблений журнал GC verbose, класифікуючи виявлені проблеми продуктивності за п'ятьма категоріями.

IBM Garbage Collection and Memory Visualiser (GCMV) є комплексним інструментом діагностики, що охоплює проблеми продуктивності, пов'язані як з GC, так і з пам'яттю. Він також використовує механізми трасування у вигляді GC verbose [18]. Звітність GCMV формує один HTML-звіт на

оброблений журнал, надаючи рекомендації щодо налаштування продуктивності, включаючи керівництво з виявлення витоків пам'яті, оптимізації продуктивності GC та конфігурації розміру купи.

IBM Health Center (HC) — це інструмент діагностики, призначений для виявлення широкого спектру проблем продуктивності в Java-системах, пропонуючи рекомендації для покращення ефективності системи. HC використовує власний механізм вибірки, який генерує файли HCD (детальні знімки стану JVM, що надають інформацію в таких областях, як пам'ять, GC, профілювання методів та потоки). HC генерує один звіт на оброблену вибірку, класифікуючи виявлені проблеми продуктивності за п'ятьма категоріями та чотирма рівнями серйозності.

IBM Whole Analysis Idle Time (WAIT) - реалізує методологію аналізу часу простою, яка визначає першопричини недостатньо використаних ресурсів. Ця методологія ґрунтується на спостереженні, що проблеми продуктивності в багаторівневих додатках часто проявляються як час простою очікуючих потоків [20]. WAIT є діагностичним інструментом, що доведено спрощує виявлення проблем продуктивності в Java-системах. Він використовує неінвазивні механізми вибірки на рівні операційної системи (наприклад, команда "ps" в Unix) та JVM у вигляді Javacores [9] (знімків стану JVM, що містять інформацію про потоки, блокування та пам'ять). WAIT генерує один HTML-звіт на набір оброблених вибірок. Звіт представляє виявлені проблеми продуктивності, відсортовані за частотою та впливом, і класифіковані за чотирма категоріями.

2.1.2. Java-бенчмарки

DaCapo та SPECJVM є двома найбільш поширеними наборами Java-бенчмарків у науковій літературі. Нижче описані версії, використані в експериментальній оцінці цієї роботи.

DaCapo 9.12 - цей бенчмарк розроблено проєктом DaCapo за підтримки таких компаній, як IBM, Intel та Microsoft. Набір складається з 14 програм з

відкритим кодом, які є реальними додатками та створюють нетривіальні навантаження на пам'ять.

Таблиця 2.1.

Програми DaCapo та їхня функціональність

Назва	Опис
avrora	Імітує набір програм, що працюють на сітці мікроконтролерів.
batik	Обробляє набір векторних зображень (SVG).
eclipse	Виконує набір тестів продуктивності в середовищі розробки Eclipse.
fop	Генерує PDF-файли на основі XSL-FO файлів, які аналізуються та форматуються.
h2	Виконує набір банківських транзакцій проти бази даних.
jython	Виконує набір скриптів Python у Java-середовищі.
luindex	Індексує набір документів за допомогою Lucene.
lusearch	Виконує набір ключових пошуків у корпусі даних за допомогою Lucene.
pmd	Перевіряє набір класів Java на наявність помилок у вихідному коді (статичний аналіз).
sunflow	Візуалізує набір зображень (рендеринг).
tomcat	Виконує набір запитів до сервера Tomcat.
tradebeans	Виконує набір біржових транзакцій через виклики Java Beans.
tradesoap	Виконує набір біржових транзакцій через виклики SOAP.
xalan	Перетворює набір XML-файлів у HTML-файли за допомогою XSLT.

SPECJVM 2008 - цей бенчмарк розроблено Standard Performance Evaluation Corp (SPEC) за внеску таких компаній, як HP, IBM та Sun. Набір складається з 10 програм, що представляють суміш реальних додатків та спеціалізованих бенчмарків, сфокусованих на основній функціональності Java [21]. Це інструмент для об'єктивного порівняння продуктивності різних реалізацій JVM (наприклад, HotSpot, JRockit) на одній і тій же або різних апаратних платформах. Хоча основний фокус на JRE, бенчмарк також опосередковано відображає продуктивність апаратного забезпечення

(процесора та підсистеми пам'яті), оскільки JVM є дуже чутливою до цих ресурсів.

Таблиця 2.2.

Програми SPECJVM та їхня функціональність

Назва	**Опис**
compiler	Компілює набір вихідних файлів Java.
compress	Використовує універсальний алгоритм стиснення даних без втрат.
crypto	Шифрує та дешифрує набір файлів за допомогою протоколів шифрування.
derby	Відкрита база даних, написана на чистій Java.
MPEGaudio	Використовує декодер MP3 для обробки аудіофайлів.
scimark	Виконує набір операцій з плаваючою комою (наукові обчислення).
serial	Серіалізує та десеріалізує набір об’єктів та примітивів.
startup	Виконує кожну іншу програму бенчмарку один раз (для оцінки часу запуску).
sunflow	Виконує набір операцій візуалізації графіки (рендеринг).
XML	Перетворює та валідує набір XML-документів, застосовуючи таблиці стилів.

SPECJVM 2008 був розроблений як заміна застарілого однопотокового SPECjvm98, щоб краще відповідати вимогам сучасних Java-додатків та багатоядерних архітектур. Бенчмарк має низьку залежність від файлового вводу/виводу (File I/O) і не включає мережевий I/O між машинами, що дозволяє більш точно сфокусуватися на продуктивності власне JVM і процесора/пам'яті.

Результати вимірюються в одиницях ops/min (операцій за хвилину), а кінцевий результат подається як SPECjvm2008 Composite Result, який є агрегованою метрикою продуктивності, що базується на геометричному середньому показників усіх окремих тестів.

Обов'язковою є категорія прогону Base, яка повинна виконуватися без будь-якого спеціального тонкого налаштування JVM, щоб оцінити її продуктивність у типових умовах використання. SPECJVM 2008 залишається

одним із важливих стандартів у галузі тестування продуктивності Java-платформ.

2.2. Представлення методології архітектури фреймворку тестування ефективності та продуктивності Java-додатків

2.2.1. Опис архітектури фреймворку

Цей розділ присвячений представленню фреймворку тестування ефективності та продуктивності Java-додатків (ФТЕПД). Спочатку буде надано контекстуальний огляд рішення, після чого буде описана внутрішня архітектура ФТЕПД, включаючи запропоновані політики. Розділ завершиться обговоренням експериментальної оцінки та отриманих результатів.

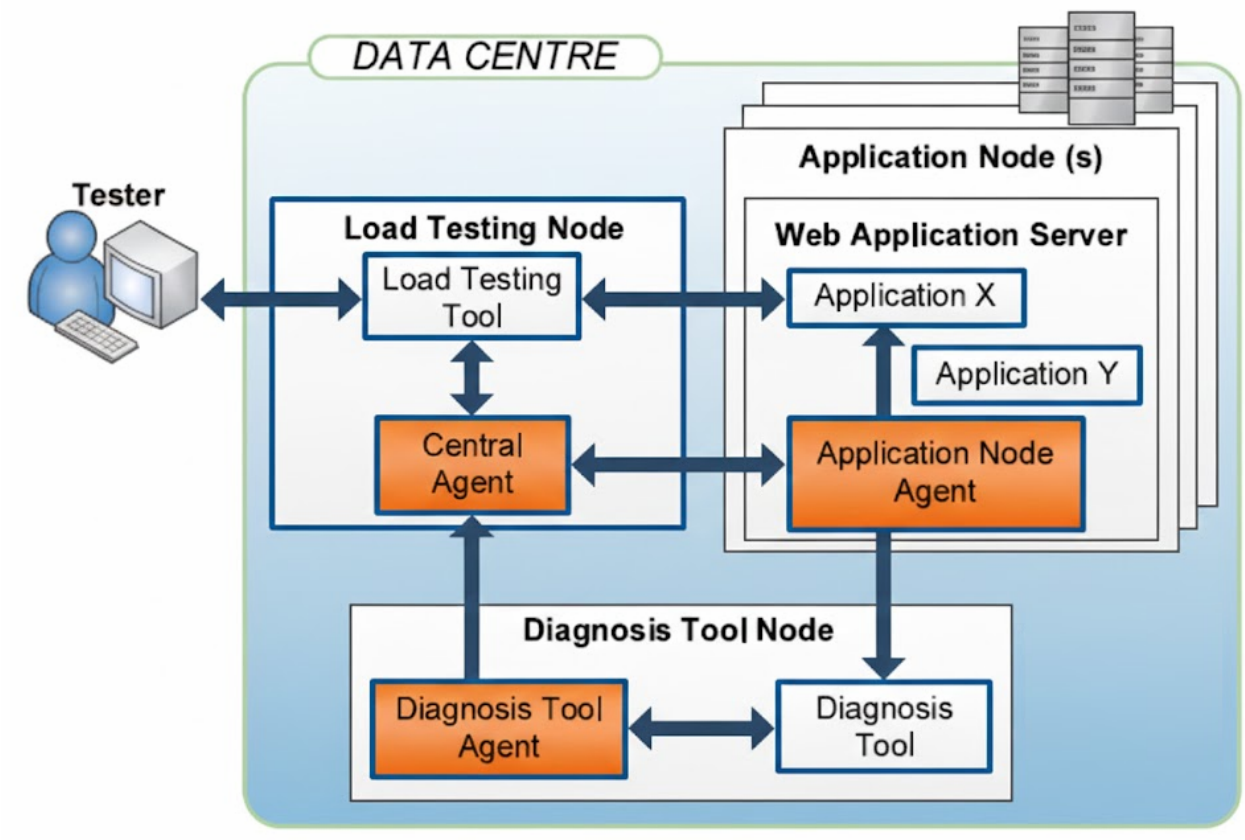

Рис. 2.1. Архітектура фреймворку тестування продуктивності Java-додатків

Основною метою цього дослідження була розробка методології побудови фреймворку ФТЕПД для автоматизації процесів, пов'язаних з

використанням інструментів діагностики продуктивності під час тестування кластерних додатків. Впровадження такого фреймворку має на меті підвищити продуктивність тестувальників та оптимізувати сам процес тестування, мінімізуючи зусилля та рівень експертних знань, необхідних для ефективної експлуатації діагностичних інструментів.

Рисунок 2.1 ілюструє архітектуру розподіленого фреймворку, призначеного для автоматизованого тестування та діагностики продуктивності Java-додатків у кластерному середовищі. Цей фреймворк виконує роль автономного менеджера в циклі зворотного зв'язку. Фреймворк складається з трьох ключових вузлів, які взаємодіють для досягнення цілей моніторингу, аналізу, планування та виконання (MAPE-K):

1. Вузол навантажувального тестування (Load Testing Node)

Цей вузол є точкою взаємодії для тестувальника та джерелом вхідного навантаження. Тестувальник (Tester) ініціює тестовий запуск і взаємодіє виключно з інструментом навантажувального тестування, будучи абстрагованим від складнощів діагностики.

Інструмент навантажувального тестування (Load Testing Tool) генерує вхідне навантаження для кластера (вузлів додатків) і є основним інтерфейсом тестувальника. Центральний агент (Central Agent) виступає як центральний контролер фреймворку (автономний менеджер). Він координує збір даних з вузлів додатків та керує процесом діагностики, взаємодіючи з агентом вузла додатку та агентом інструменту діагностики. Цей агент виконує функції моніторингу (отримання даних), аналізу (з використанням політик), планування та виконання (ініціювання дій).

2. Вузли додатків (Application Node(s))

Це керовані системи, де розгорнуто корпоративний додаток (кластер). Сервер веб-додатків (Web Application Server) - хостинг для прикладного коду (Application X, Application Y). Це самі керовані системи, продуктивність яких оптимізується. Агент вузла додатку (Application Node Agent): Встановлюється на кожному вузлі кластера. Він відповідає за:

- збір вибірок продуктивності (наприклад, дампи купи, GC verbose, Javacores) відповідно до інструкцій центрального агента (на основі політики збору даних).

- передачу зібраних вибірок центральному агенту.

3. Вузол інструменту діагностики (Diagnosis Tool Node)

Цей вузол відповідає за обробку зібраних даних і генерацію звітів про проблеми. Агент інструменту діагностики (Diagnosis Tool Agent) отримує вибірки даних від центрального агента (на основі політики завантаження). Він запускає (конфігурує) сам інструмент діагностики. Інструмент діагностики (Diagnosis Tool) обробляє отримані вибірки (EMAT, GCLITE, WAIT) і генерує діагностичні звіти (HTML-звіти про витоки пам'яті або проблеми GC).

Взаємодія та потік даних (контур управління):

1. Навантаження та моніторинг.

Тестувальник ініціює навантаження через інструмент навантажувального тестування, яке спрямовується до вузлів додатків.

2. Збір даних.

Центральний агент (менеджер) інструктує агентів вузлів додатків збирати діагностичні вибірки (моніторинг).

3. Обробка даних.

Зібрані вибірки передаються центральному агенту, а потім, відповідно до політики завантаження, перенаправляються до агента інструменту діагностики для обробки (аналіз).

4. Діагностика та консолідація.

Інструмент діагностики генерує звіти. Центральний агент отримує результати і виконує їх консолідацію та аналіз (за допомогою політики консолідації та бази політик / знань).

5. Адаптація (непряме виконання).

На основі аналізу центральний агент може приймати рішення про необхідність адаптації – наприклад, змінити інтервал збору даних, зупинити

тест або повідомити тестувальника про виявлену проблему (планування та виконання).

Ця архітектура дозволяє автоматично проводити інженерію продуктивності, зменшуючи людський фактор та підвищуючи ефективність діагностики в кластерному середовищі.

2.2.2. Концептуальна структура фреймворку

ФТЕПД функціонує паралельно з виконанням тесту продуктивності, тим самим інкапсулюючи для тестувальника складнощі, пов'язані з коректною конфігурацією та використанням діагностичних інструментів. Це дозволяє тестувальнику взаємодіяти лише з основним інструментом навантажувального тестування.

Самоадаптивна система типово складається з керованої системи та автономного менеджера. У цьому контексті ФТЕПД виконує роль автономного менеджера. Отже, він контролює контур зворотного зв'язку, який адаптує керовану систему відповідно до визначеного набору цілей. Керованими системами тут є інструмент діагностики та вузли додатків. Це ілюструється на рисунку 3.2, що демонструє концептуальний вигляд ФТЕПД.

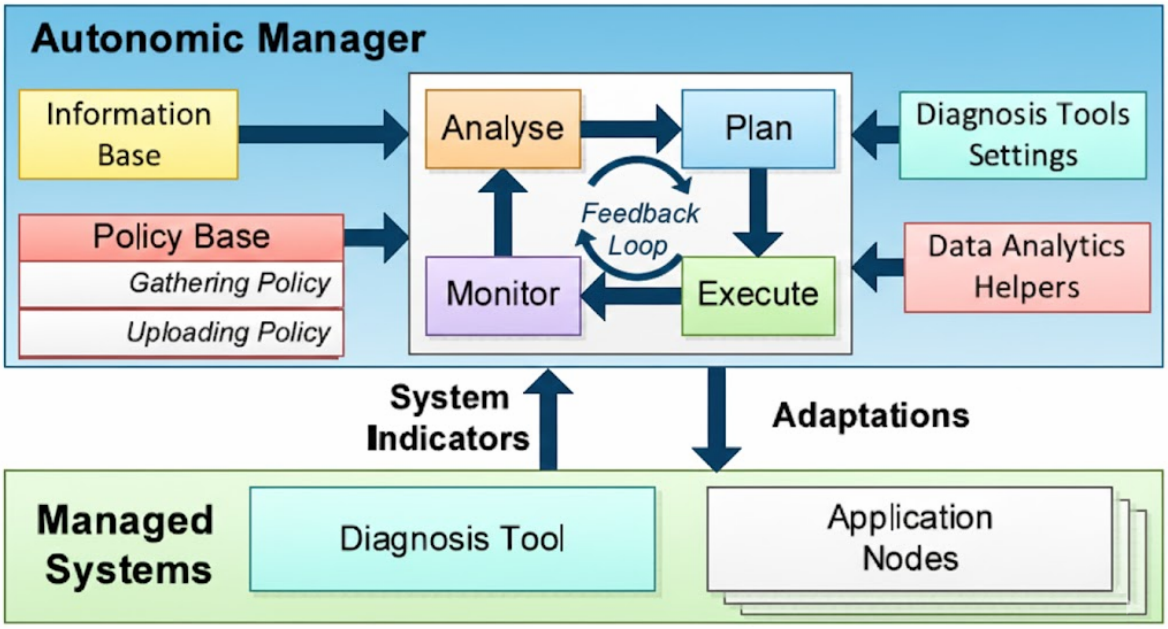

Рис. 2.2. Концептуальна структура фреймворку

Ключовим структурним елементом ФТЕПД є його база політик, яка виконує функцію елемента Знань (Knowledge) в межах моделі MAPE-K. База Політик визначає набір доступних правил керування. Інкапсуляція знань у політиках дозволяє ФТЕПД бути легко розширюваним та здатним інтегрувати численні політики, придатні для різних сценаріїв і діагностичних інструментів.

У цьому контексті, політика визначається як набір взаємопов'язаних завдань, необхідних для виконання одного з процесів, асоційованих з використанням діагностичного інструменту в рамках тестового запуску продуктивності.

Кожен діагностичний інструмент потребує трьох основних типів політик:

- Політика збору даних (Data Collection Policy) контролює збір вибірок у вузлах додатків.

- Політика завантаження (Upload Policy) регулює момент відправлення зібраних вибірок до інструменту діагностики для обробки.

- Політика консолідації (Consolidation Policy) відповідає за інтеграцію всіх результатів, отриманих від діагностичного інструменту для різних вузлів кластера.

Крім того, діагностичний інструмент може підтримувати інші допоміжні політики (наприклад, для резервного копіювання вибірок або для ініціації дій на основі вихідних даних інструменту). Ці політики можуть використовувати набір доступних аналітичних допоміжних засобів — підтримуючої логіки, що надає послуги для подальшого налаштування поведінки політики. Наприклад, політика може оцінювати серйозність виявлених інструментом помилок, використовуючи різні допоміжні засоби для визначення різних наборів рівнів серйозності. Специфічні вимоги до налаштування інструменту (наприклад, діапазон застосовних інтервалів вибірки) також фіксуються фреймворком як налаштування інструменту діагностики.

З точки зору конфігурації, тестувальник повинен надати інформаційну базу (Information Base, як показано на рисунку 2.2), що складається з усіх вхідних параметрів, необхідних для обраних політик. Наприклад:

- політика завантаження може вимагати часовий інтервал для визначення моменту відправлення вибірок на обробку.

- політика збору даних може використовувати інтервал вибірки для визначення частоти збору даних.

- політика консолідації може вимагати в якості вхідних даних топологію кластерної системи для диференціації вузлів додатків.

2.3. Основний процес функціонування фреймворку

Цикл роботи автономного менеджера, який керує фреймворком тестування продуктивності java-додатків подано на рис. 2.3.

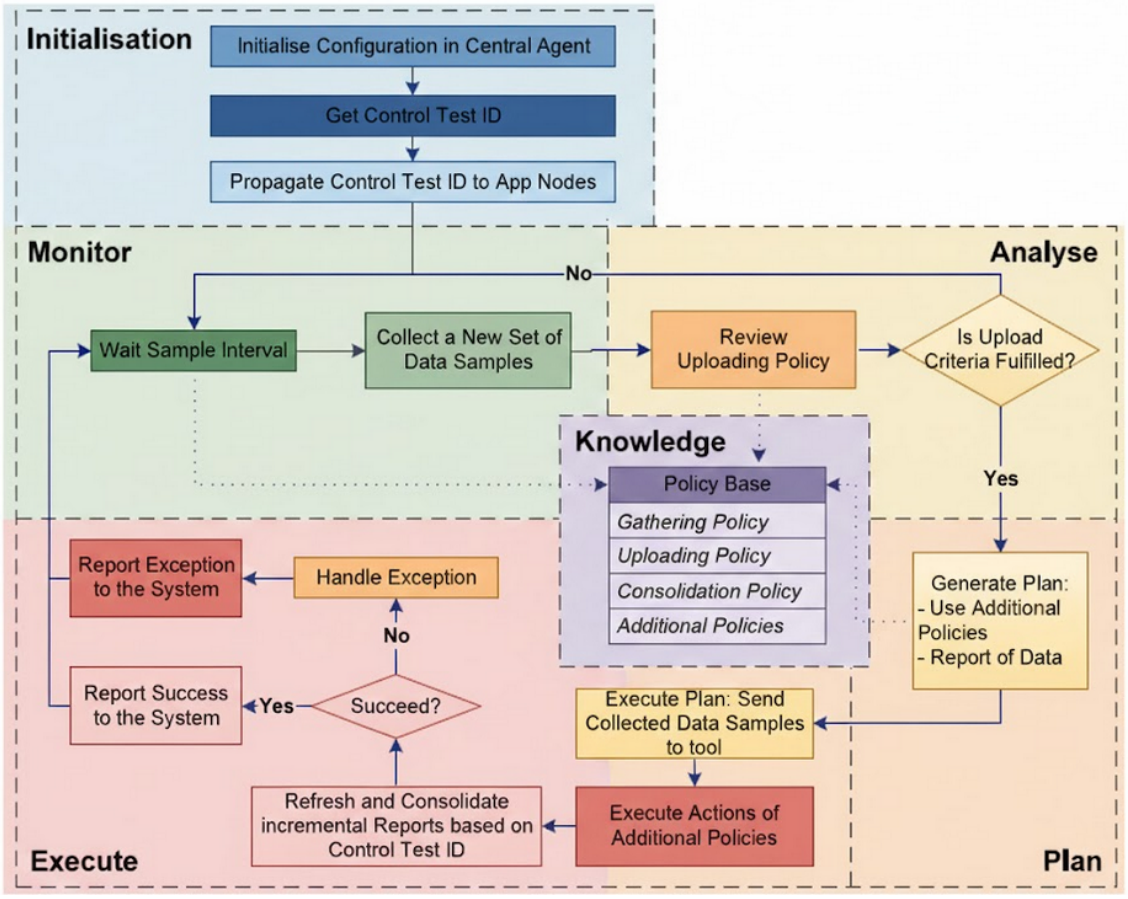

Рис. 2.3. Основний процес роботи фреймворку

Рисунок 2.3 ілюструє повний цикл роботи автономного менеджера, який керує фреймворком тестування ефективності та продуктивності java-додатків (ФТЕПД), відповідно до моделі MAPE-K (Monitor, Analyze, Plan, Execute – Knowledge).

Процеси поділяються на чотири основні фази: ініціалізація, моніторинг, аналіз, планування та виконання.

1. Ініціалізація

Ця фаза встановлює початкові умови для контрольованого тестового запуску:

- ініціалізація конфігурації: конфігурація ініціалізується у центральному агенті.

- отримання ідентифікатора контрольного тесту (Control Test ID): Генерується унікальний ідентифікатор для поточного тестового запуску.

- поширення ідентифікатора (Propagate Control Test ID): цей ідентифікатор поширюється на вузли додатків (App Nodes). Він використовується для зв'язування всіх зібраних вибірок даних та звітів з конкретним тестовим запуском.

2. Моніторинг

Ця фаза відповідає за систематичний збір даних продуктивності з керованих систем (вузлів додатків):

- очікування інтервалу вибірки (Wait Sample Interval): процес чекає заданий інтервал, визначений політикою збору даних (Gathering Policy), перш ніж збирати нові дані.

- збір нового набору вибірок даних (Collect a New Set of Data Samples): центральний агент ініціює збір вибірок (наприклад, дампи купи, GC verbose) з вузлів додатків.

- перехід до аналізу: зібрані дані передаються до фази аналізу.

3. Аналіз

На цій фазі приймається рішення про необхідність обробки зібраних даних діагностичним інструментом.

- перегляд політики завантаження (Review Uploading Policy): центральний агент перевіряє критерії, визначені політикою завантаження.

- Перевірка критеріїв завантаження (Is Upload Criteria Fulfilled?): перевіряється, чи відповідають зібрані вибірки умовам для відправки на обробку (наприклад, чи минув заданий час, чи досягнуто максимальної кількості вибірок). Якщо критерії не виконані, цикл повертається до моніторингу. Якщо критерії виконані, процес переходить до планування.

4. Планування та виконання

Ці фази визначають необхідні дії та втілюють їх.

Планування. Генерація плану (Generate Plan) - створюється план, який включає: звіт про дані (Report of Data) який необхідно надіслати діагностичному інструменту та Use Additional Policies - визначаються будь-які додаткові дії, які потрібно виконати (наприклад, зміна конфігурації).

Виконання:

- виконання плану: надсилання вибірок (Execute Plan: Send Collected Samples to tool): рібрані вибірки даних фізично надсилаються до вузла інструменту діагностики для обробки.

- виконання дій додаткових політик (Execute Actions of Additional Policies): виконуються будь-які додаткові дії, визначені на етапі планування.

- оновлення та консолідація звітів (Refresh and Consolidate incremental Reports): отримані інкрементальні звіти з інструменту діагностики оновлюються та консолідуються на основі ідентифікатора Control Test ID (за допомогою політики консолідації).

- перевірка успіху. Перевіряється, чи вдалося виконати операцію (наприклад, чи успішно оброблено звіт і чи виявлено проблему). Про успіх повідомляється системі. Якщо невдача, то обробляється виняток (Handle Exception), і про виняток повідомляється системі.

База політик є елементом знань у моделі MAPE-K. Вона підтримує фази аналізу, планування та виконання, зберігаючи набір правил, які керують поведінкою фреймворку. Вона включає:

- політика збору (gathering policy): правила для збору даних (наприклад, інтервал вибірки).

- політика завантаження (uploading policy): критерії, які визначають, коли дані мають бути відправлені на обробку.

- політика консолідації (consolidation policy): правила для інтеграції та інтерпретації звітів з різних вузлів.

- додаткові політики (additional policies): правила для виконання допоміжних дій, наприклад, для детального аналізу або зміни поведінки.

Основний процес продовжується ітеративно, доки не буде завершено тестовий запуск продуктивності або не буде досягнута альтернативна умова виходу, визначена політикою. При завершенні всі застосовні політики оцінюються востаннє перед завершенням процесу. Будь-які винятки обробляються внутрішньо та повідомляються системі.

Таким чином, фреймворк ФТЕПД забезпечує автоматизований, адаптивний цикл зворотного зв'язку, що керується політиками, для ефективної діагностики продуктивності кластерних систем.

2.4. Архітектура та реалізація фреймворку тестування ефективності та продуктивності java-додатків

Фреймворк тестування ефективності та продуктивності java-додатків реалізований на основі багатоагентної архітектури, що забезпечує розподілене управління та координацію.

2.4.1. Багатоагентна архітектура

ФТЕПД складається з трьох спеціалізованих типів агентів, кожен з яких відповідає за певний домен:

1. Агент контролю (control agent) - відповідає за високорівневу координацію та управління. Його основні функції включають:

- Оцінка політик (наприклад, політик завантаження та консолідації).

- Взаємодія з інструментом навантажувального тестування для моніторингу початку та завершення тесту.

- Поширення рішень та команд на інші вузли.

2. Агент вузла додатку (application node agent) - відповідає за виконання локальних завдань на кожному вузлі кластера, включаючи збір вибірок даних та надсилання зібраних вибірок до інструменту діагностики.

3. Агент інструменту діагностики (diagnosis tool agent) - відповідає за інтерфейс та керування самим інструментом діагностики (наприклад, подача даних, ініціалізація обробки та постобробка згенерованих звітів).

Внутрішньо кожен агент складається з різних компонентів. Це проілюстровано на рис. 2.4, яка представляє діаграму компонентів агента контролю.

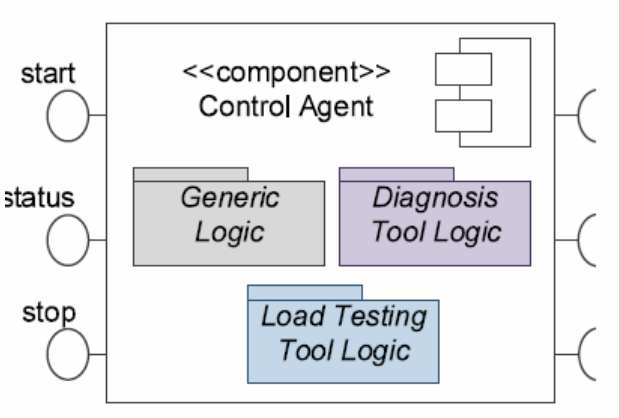

Рис. 2.4. Діаграма компонентів агенту контролю

2.4.2. Внутрішня структура компонентів

Внутрішньо кожен агент побудований з різних компонентів. Наприклад, агент контролю має модульну структуру, проілюстровану на діаграмі компонентів (рис. 2.4):

- загальний компонент (general component): містить основну логіку контролю та допоміжну функціональність, незалежну від конкретного інструменту діагностики чи навантажувального тестування.

- спеціалізовані компоненти: логіка, яка взаємодіє з цільовими інструментами, інкапсульована у відповідних компонентах (наприклад, компоненті інструменту діагностики та компоненті інструменту навантажувального тестування). Така стратегія мінімізує зміни коду, необхідні при адаптації до нових інструментів.

Для підвищення гнучкості та сумісності, компоненти доступні виключно через інтерфейси (рис. 2.5). Високорівнева структура компонента інструменту діагностики демонструє:

- основний інтерфейс (idiagnosistool): експонує всі необхідні для взаємодії дії.

- абстрактний клас: надає спільну функціональність.

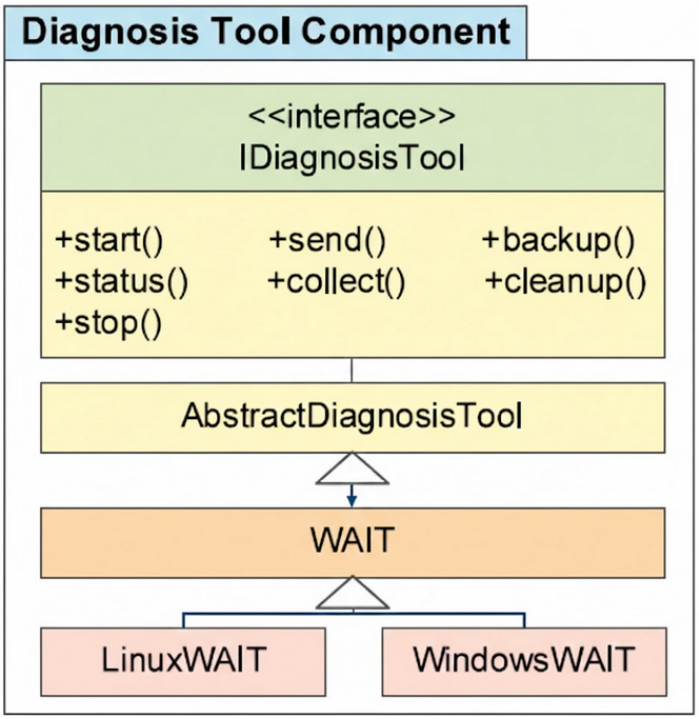

Рис. 2.5. Компонент інструменту діагностики (діаграма класів)

Ця ієрархія може бути розширена для підтримки конкретних діагностичних інструментів (наприклад, WAIT) на різних операційних системах. Необхідний клас, що підтримує певний інструмент і операційну систему, вибирається автоматично відповідно до шаблону проєктування Factory [17].

2.4.3. Протокол взаємодії агентів

Комунікація між агентами здійснюється через команди, що відповідає шаблону проєктування Command [7]. Агент контролю виступає ініціатором команд, тоді як інші агенти реалізують логіку, необхідну для їх виконання.

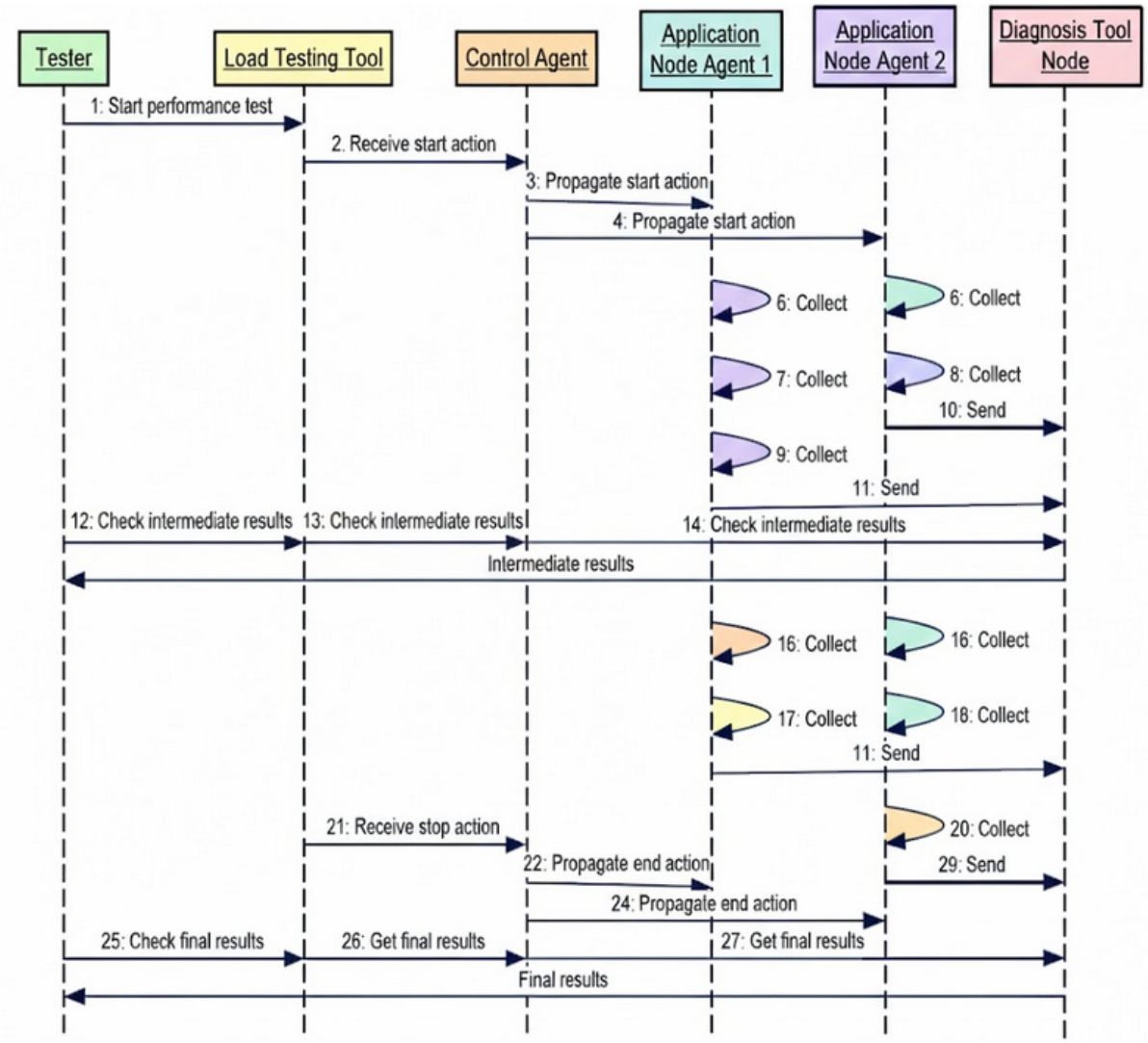

Рис. 2.6. Діаграма послідовності

Діаграма послідовності ілюструє типову взаємодію:

1. Початок тесту (крок 1): тестувальник ініціює тестовий запуск.

2. Поширення запуску (кроки 2-4): агент контролю поширює команду початку на всі агенти вузлів додатків.

3. Ітеративний цикл (кроки 5-9): кожен агент вузла додатку виконує свої періодичні завдання (збір даних).

4. Завантаження даних (кроки 10-11): після виконання політики завантаження, зібрані дані надсилаються до інструменту діагностики для обробки.

5. Перегляд результатів (кроки 12-14): тестувальник може переглядати проміжні результати.

6. Завершення тесту (кроки 21, 22, 24): після завершення тесту агент контролю поширює команду зупинки.

7. Отримання остаточних результатів (кроки 25-27): отримуються фінальні результати діагностики.

Цей протокол забезпечує структурований, ітеративний та адаптивний процес управління життєвим циклом тестування продуктивності.

Висновки до розділу

У другому розділі досліджено сучасний інструментарій діагностики продуктивності Java-додатків та підходи до їхнього бенчмаркінгу. Проаналізовано функціональні можливості наявних засобів моніторингу з точки зору точності, накладних витрат і придатності до використання у розподілених системах. Виявлено обмеження традиційних бенчмарків при оцінюванні продуктивності в умовах динамічного навантаження.

На основі проведеного аналізу обґрунтовано вимоги до архітектури фреймворку тестування ефективності та продуктивності Java-додатків. Запропоновано архітектурну модель фреймворку, орієнтовану на модульність, масштабованість і розширюваність.

Розроблено концептуальну структуру фреймворку, що забезпечує інтеграцію різних джерел продуктивнісних метрик. Описано основний процес функціонування фреймворку, який охоплює збір, агрегацію та аналіз даних. Обґрунтовано доцільність застосування багатоагентної архітектури для децентралізованого моніторингу. Детально розглянуто внутрішню структуру компонентів фреймворку.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА ПОЛІТИК ОПТИМІЗАЦІЇ РЕСУРСІВ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ JAVA-СИСТЕМ

Наступні підрозділи описують набір політик, які були розроблені для інтеграції та функціонування в рамках фреймворку тестування ефективності та продуктивності java-додатків. Ці політики головним чином базуються на результатах емпіричних оцінок компромісів між точністю діагностики та накладними витратами на продуктивність.

3.1. Методика автоматизованого налаштування інтервалу вибірки на основі впливу на продуктивність

Ця адаптивна політика була розроблена з метою балансування компромісу між точністю результатів діагностичного інструменту та впливом на продуктивність тестованого додатка. Обидва ці фактори критично залежать від вибору інтервалу вибірки (Sampling Interval, SI).

Процес політики ілюструється на рисунку 3.1, де кожен крок відповідає відповідному елементу моделі MAPE-K:

Ця політика вимагає від користувача два основні вхідні параметри:

1. Поріг часу реакції (RTThreshold) - максимально допустимий вплив на час реакції (виражений у відсотках).

2. Період розігріву (Warm-up Period) - час, необхідний для виконання всіх транзакцій принаймні один раз, що гарантує, що наступні вимірювання сприяють репрезентативному середньому часу реакції тестового запуску.

Додаткові параметри отримуються з бази знань, оскільки вони специфічні для кожного діагностичного інструменту:

- Мінімальний SI (SI_{min}) - мінімальний інтервал вибірки, який дозволено використовувати.

- ΔSI - значення, на яке має бути збільшений SI у разі коригування.

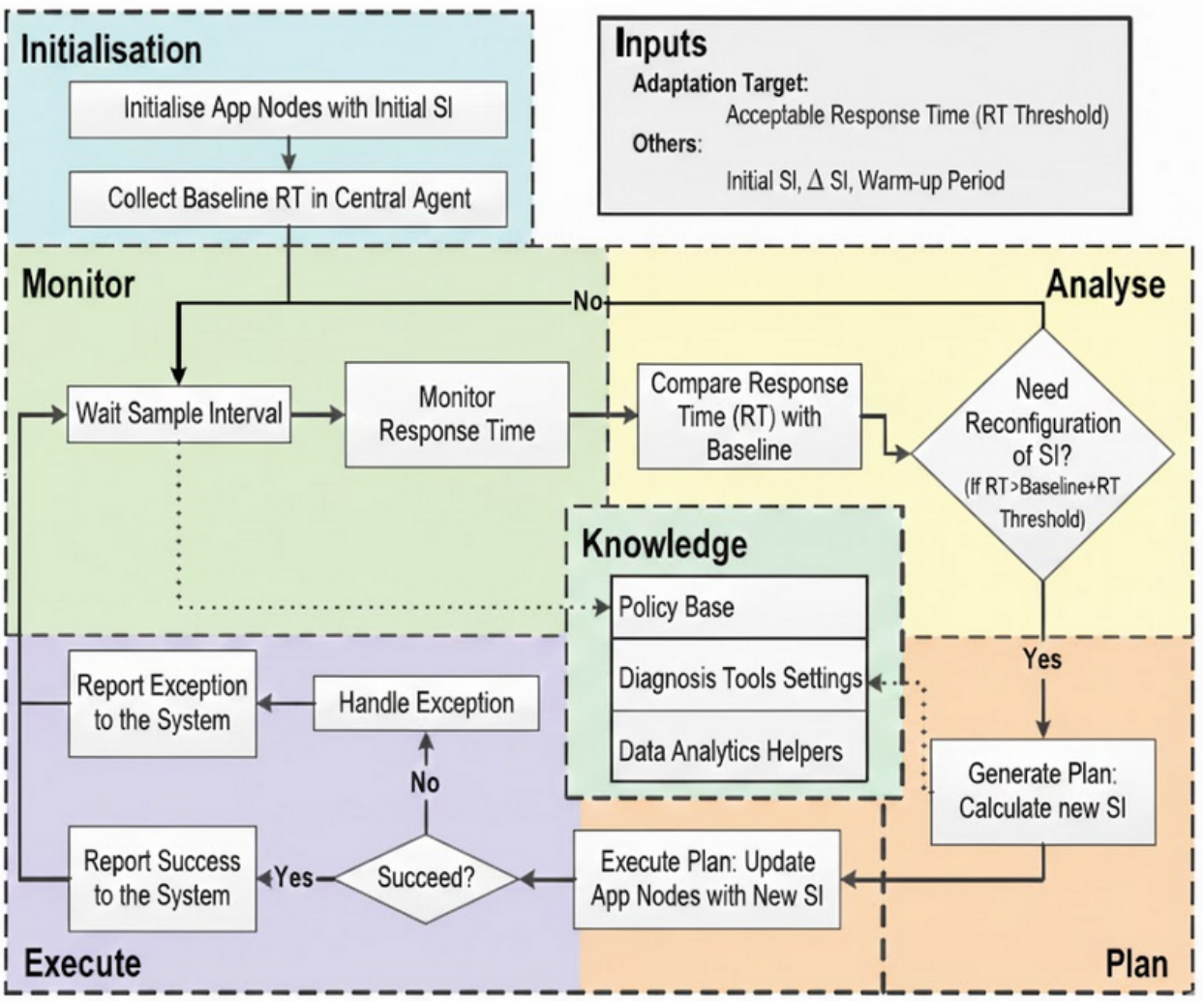

Рис. 3.1. Політика збору даних, орієнтована на точність

Процес Адаптивного Налаштування проходить наступні етапи:

1. Ініціалізація та базова лінія (Initialization and Baseline) - gроцес починається з очікування налаштованого періоду розігріву. Після цього з інструменту навантажувального тестування отримується середній час реакції (RTAVG). Це значення встановлюється як базовий час реакції (RTBL). Далі, вузли додатків ініціалізуються з мінімальним SImin. Ця стратегія максимізує збір вибірок (і, відповідно, точність), якщо продуктивність не погіршується нижче заданого порогу.

2. Ітеративний моніторинг (Monitoring and Analysis Loop) hозпочинається ітеративний процес моніторингу, який триває до завершення тестування продуктивності.

- процес очікує поточний SI, оскільки вплив на продуктивність, спричинений діагностичним інструментом, може не проявитися негайно після збору даних.

- отримується нове значення середнього часу реакції.

- Порівняння з Порогом (Аналіз), тобто Новий RT_{AVG} порівнюється з RT_{BL} для перевірки, чи не було перевищено $RT_{Threshold}$.

3. Коригування SI (планування та виконання):

- якщо поріг перевищено, це свідчить про те, що поточний SI є надто малим, що спричиняє накладні витрати вище налаштованого порогу. У цьому випадку SI збільшується на значення ΔSI.

- новий SI поширюється на всі вузли додатків, які починають його використовувати з наступної ітерації збору даних.

Ця політика побудована на основі поширеного у галузі сценарію, коли команда тестувальників прагне мінімізувати кількість тестових запусків через бюджетні або часові обмеження. Дозволяючи певний, контрольований рівень накладних витрат у тестованому додатку, можна виявити більше помилок продуктивності на додаток до звичайних результатів тесту.

3.2. Політика розподілу ресурсів інструменту діагностики через адаптивне регулювання інтервалу завантаження

Ця адаптивна політика була розроблена для балансування компромісу між рівнями використання ресурсів, необхідними для діагностичного інструменту, та кількістю вибіркових даних, які одночасно обробляються інструментом. Обидва ці фактори впливають на вибір інтервалу завантаження (Uploading Interval, UI).

Процес політики зображений на рисунку 3.2, де кожен крок відповідає відповідному елементу моделі MAPE-K:

Політика вимагає від користувача два вхідні параметри:

1. Початковий інтервал ($UI_{initial}$) завантаження.

2. Значення, на яке UI повинен змінитися, коли потрібне коригування.

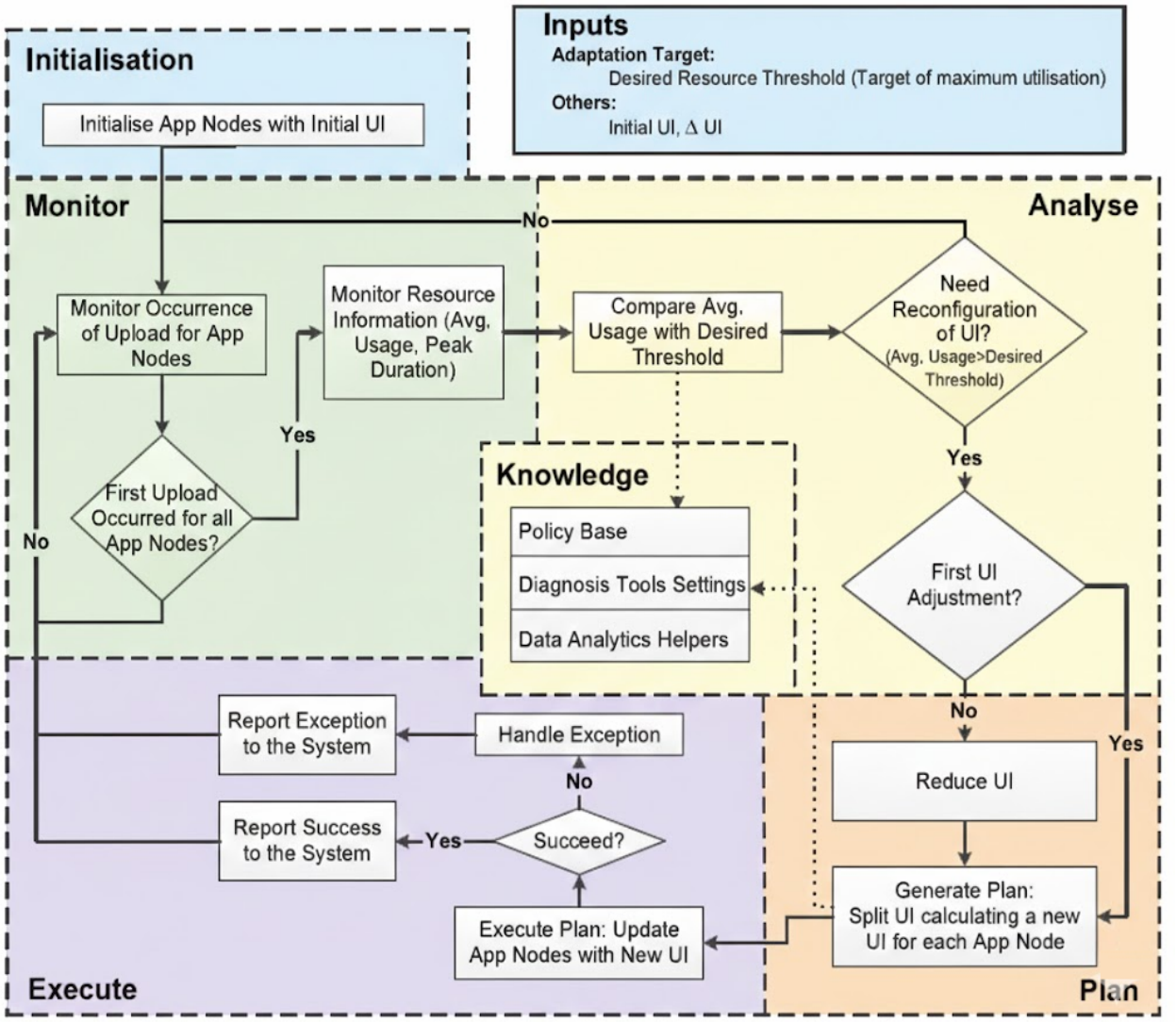

Рис. 3.2. Політика завантаження, орієнтована на ефективність

Додатковий параметр який отримується з бази знань це ціль максимального використання (U_{MAX}) - максимально допустимий відсоток використання спільного ресурсу (наприклад, CPU сервера діагностичного інструменту). Метою цієї цілі є збереження певної невикористаної потужності для забезпечення м'якої гарантії якості обслуговування (QoS). Наприклад, для CPU ця ціль часто встановлюється на рівні 90%.

Процес адаптивного налаштування полягає в наступному:

1. Ініціалізація та перше завантаження. Процес починається з ініціалізації вузлів додатків з $UI_{initial}$. Потім процес очікує, поки всі вузли додатків завантажать свої вибірки хоча б один раз. Це забезпечує, що коригування UI відбудеться лише за необхідності.

2. Моніторинг та аналіз ресурсів. Після того, як всі вузли завершили завантаження, процес отримує:

- Середнє використання ресурсу (RES_{AVG}) - середнє використання ресурсу спільного сервісу (наприклад, сервера діагностичного інструменту) під час обробки вибірок.

- Середня тривалість використання ресурсу ($DRES_{AVG}$).

3. Порівняння та коригування - RES_{AVG} порівнюється з U_{MAX}. Якщо RES_{AVG} перевищує U_{MAX} то розраховуються нові, диференційовані UI.

- для кожного вузла розраховується різний UI, щоб запобігти одночасному завантаженню результатів усіма вузлами. Розрахунок нових UI базується на поточному UI шляхом ітеративного додавання або віднімання $DRES_{AVG}$ до/від поточного UI, доки всі вузли не отримають унікальний UI. Наприклад, для 5 вузлів, поточного UI = 30 хв, і $DRES_{AVG}$ = 1 хв, нові UI можуть бути розподілені як 28, 29, 30, 31 та 32 хвилини.

- у разі, якщо лише розподілу UI недостатньо, щоб знизити RES_{AVG} нижче U_{MAX}, поточний UI зменшується на ΔUI. Це зменшення має на меті зменшити кількість вибірок, наданих на вузол у кожному завантаженні.

4. Виконання. Нові UI поширюються на відповідні вузли додатків, які починають їх використовувати з наступної ітерації завантаження.

Ця політика була розроблена для сценаріїв, де кількість доступних ліцензій для певного інструменту діагностики є обмеженою (наприклад, через бюджетні обмеження).

У такому випадку, здатність ефективно ділитися доступними екземплярами інструменту між різними командами та проєктами є вкрай бажаною для максимізації повернення інвестицій (ROI) в інструмент.

3.3. Політика багаторівневої консолідації діагностичних звітів для кластерних систем

Ця політика була розроблена для мінімізації зусиль та експертних знань, необхідних тестувальнику для аналізу результатів діагностики в кластерному додатку. Це досягається шляхом надання чотирьох рівнів деталізації результатів, що забезпечує гнучке відстеження прогресу та ідентифікацію системних або вузлових проблем.

Політика пропонує чотири інтегровані вигляди результатів:

1. Консолідований вигляд на рівні системи (System-Level Consolidated View) - основний вигляд, який дозволяє тестувальнику відстежувати прогрес діагностики по всьому кластеру та швидко ідентифікувати актуальні системні проблеми.

2. Індивідуальний вигляд на рівні системи (System-Level Individual View) відображає результати, отримані внаслідок одного циклу обробки (контрольованого UI).

3. Консолідований вигляд на рівні вузла (Node-Level Consolidated View) - фокусується на сукупних результатах конкретного вузла.

4. Індивідуальний вигляд на рівні вузла (Node-Level Individual View) - відображає результати окремого циклу обробки для певного вузла.

Цей набір виглядів пропонує тестувальнику різні рівні деталізації, що дозволяє легко виявляти проблеми продуктивності як на системному рівні, так і на рівні окремих вузлів.

Процес політики зображений на рисунку 3.3 і вимагає специфічних вхідних даних для класифікації та оцінки результатів.

Вхідні параметри:

1. Тип серйозності (Severity Type) - dизначає набір застосовних категорій серйозності, за якими класифікуються виявлені проблеми продуктивності.

2. Пороги серйозності (Severity Thresholds) обмежують діапазони серйозності для кожної категорії.

3. Оцінка Go No-Go - визначає альтернативну умову виходу з тестового запуску (іншу, ніж завершення за часом).

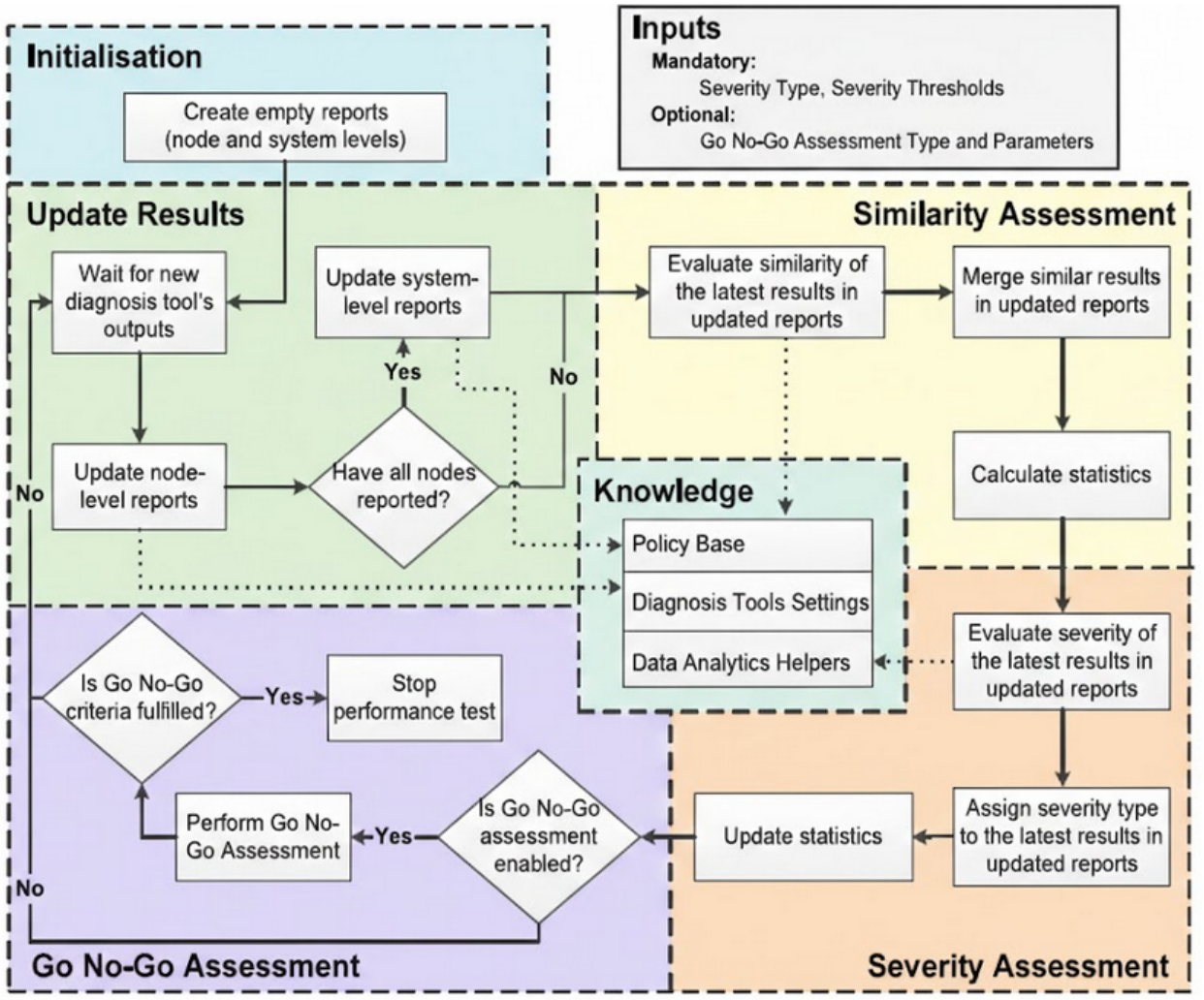

Рис. 3.3. Політика консолідації з багатьма представленнями

Етапи виконання (MAPE-K):

1. Ініціалізація. Процес починається з генерації набору порожніх звітів для всіх чотирьох виглядів.

2. Моніторинг та оновлення звітів на рівні вузла:

- процес очікує, поки інструмент діагностики не згенерує новий набір вихідних даних.

- вилучення якісної інформації (аналіз) - здійснюється аналіз усіх отриманих вихідних даних (оскільки один процес завантаження може генерувати кілька звітів) для вилучення якісної інформації про виявлені проблеми (включаючи їхні категорії та підкатегорії). Цей аналіз спирається на Базу Знань для доступу до правил, специфічних для інструменту діагностики.

- генерація індивідуального звіту на рівні вузла шляхом консолідації всіх виходів інструментів.

- оновлення консолідованого звіту на рівні вузла з новими проблемами. Нова інформація позначається для подальших завдань.

3. Консолідація на рівні системи.

Відстежуються оновлення по вузлах. Після отримання результатів від усіх вузлів оновлюються звіти на рівні системи (за логікою, подібною до логіки оновлення звітів на рівні вузлів).

4. Оцінка подібності та статистичний аналіз:

- оцінюється подібність нових результатів, використовуючи правила з бази знань, що дозволяє подальшу консолідацію результатів та уникнення дублювання виявлених проблем.

- розраховується набір стандартних статистичних метрик (середнє, стандартне відхилення та коефіцієнт варіації) для поступового побудування історичного тренду для кожного типу звіту.

5. Оцінка серйозності:

- розраховується серйозність нових результатів відповідно до правил, застосовних для діагностичного інструменту.

- результати класифікуються відповідно до стилю серйозності та порогів серйозності, налаштованих тестувальником.

- розраховуються стандартні статистичні дані (подібні до попередніх) для кожного типу серйозності.

Після завершення цих етапів консолідовані звіти оновлюються та стають доступними для тестувальників.

Альтернативний критерій виходу (Go No-Go Evaluation) полягає в наступному:

- якщо тестувальник активував опцію оцінки Go No-Go, виконується відповідна оцінка.

- якщо критерій виходу Go No-Go виконано (наприклад, виявлено критичну кількість високосерйозних помилок), ініціюється дія зупинки, що забезпечує альтернативний критерій завершення тесту.

Ця політика є відповіддю на сценарії тривалого моніторингу кластерних додатків, де величезний обсяг вихідних даних діагностики може перевантажити тестувальника. Автоматизована консолідація та класифікація суттєво зменшують необхідні зусилля та експертні знання для аналізу.

3.4. Аналітичні допоміжні засоби даних фреймворку тестування ефективності та продуктивності java-додатків

Наступні підрозділи описують набір аналітичних допоміжних засобів даних, розроблених для розширення можливостей фреймворку. Як і раніше обговорювані політики, вони ґрунтуються на результатах оцінок компромісів.

3.4.1. Оцінки подібності

Оцінка подібності визначає логіку, яка використовується для ідентифікації та об'єднання тих проблем продуктивності, які, хоча й повідомляються інструментом діагностики як окремі сутності, мають схожу симптоматику і, отже, є екземплярами тієї самої базової проблеми. Ця підтримуюча логіка інкапсульована в ФТЕПД як аналітичний допоміжний засіб даних.

На основі аналізу виходів діагностичних інструментів та результатів оцінок компромісів, було реалізовано дві основні стратегії оцінки подібності:

1. Оцінка рівності (Equality Assessment) - застосовується до інструментів, які генерують виключно якісні описи проблем (наприклад, IBM WAIT). У цьому випадку, для об'єднання проблем достатньо прямого порівняння їхніх описів.

2. Оцінка семантичної подібності (Semantic Similarity Assessment) - застосовується до інструментів, які генерують якісні описи проблем, що містять вбудовані кількісні дані (наприклад, IBM Health Center). У цьому випадку більш доцільним є порівняння описів проблем з точки зору їхньої семантичної подібності. Для цієї роботи було обрано відстань Джаро-Вінклера (Jaro-Winkler distance) — метрику, що широко використовується для виявлення дублікатів рядків, оскільки вона надає нормалізований бал (0 — повна відсутність подібності, 1 — точна відповідність). Враховуючи, що в описі проблеми зазвичай змінюється лише кількісна інформація, для ідентифікації подібних проблем було встановлено поріг несхожості 0.1.

3.4.2. Стилі серйозності для виявлення проблеми продуктивності

Стиль серйозності визначає набір категорій, до яких може бути класифікована виявлена проблема продуктивності. Ця підтримуюча логіка також інкапсульована в ФТЕПД як аналітичний допоміжний засіб даних. Стилі серйозності можуть також використовуватися для налаштування поведінки інших оцінок (наприклад, оцінки Go No-Go).

Було реалізовано два основні стилі серйозності:

1. Стиль серйозності з п'ятьма рівнями - базується на загальновизнаних (і часто використовуваних у промисловості) категоріях серйозності, запропонованих Міжнародною радою з кваліфікації в галузі тестування програмного забезпечення (ISTQB): критичний, значний, помірний, незначний та косметичний [10].

2. Стиль серйозності з двома рівнями - спрощений стиль, який класифікує проблеми лише на дві категорії: критичні та некритичні. Цей

стиль дозволяє тестувальникам фокусуватися виключно на найбільш важливих проблемах.

Класифікація проблеми продуктивності в межах певної категорії серйозності здійснюється на основі частоти виникнення проблеми (як визначено для кожного інструменту діагностики) відносно порогів, налаштованих для застосовних категорій серйозності, незалежно від обраного стилю серйозності.

3.4.3. Оцінки Go No-Go

Оцінка Go No-Go пропонує альтернативну умову виходу для тестового запуску (відмінну від запланованої тривалості). Цей аналітичний допоміжний засіб передбачає визначення критеріїв, відповідальних за оцінку виконання цієї умови.

На основі емпіричних результатів (які показали тенденцію до стабілізації кількості проблем, виявлених діагностичним інструментом, з часом) була розроблена оцінка Go No-Go, заснована на коефіцієнті варіації (Coefficient of Variation, CV) [16].

Коефіцієнт варіації (CV) - Ця безрозмірна статистична метрика дозволяє вимірювати варіативність у кількості виявлених проблем. Наприклад, CV=0.1 у послідовних версіях звітів вказує на практично стабільну кількість виявлених проблем, незалежно від їхньої фактичної кількості. Отже, ця метрика ефективно фіксує момент, коли процес виявлення помилок вичерпав свою здатність генерувати нові результати. Досягнення цієї точки дозволяє зупинити тестовий запуск, запобігаючи марнуванню людських та обчислювальних ресурсів.

$$CV = \frac{\sigma}{\mu}$$

де σ — стандартне відхилення, а μ — середнє значення кількості виявлених проблем.

Ця оцінка вимагає трьох вхідних параметрів:

1. Категорії серйозності для оцінки - визначають підмножину категорій серйозності (у межах обраного стилю серйозності), які будуть оцінюватися (наприклад, можуть бути виключені некритичні або косметичні проблеми).

2. Кількість консолідованих звітів на рівні системи для оцінки - обмежує кількість найновіших версій цього звіту, які будуть використовуватися для розрахунку CV. Цей параметр непрямо впливає на мінімальну тривалість тесту.

3. Набір порогів CV. Один поріг $CV_{Threshold}$ для кожної категорії серйозності, який використовується для визначення, чи потрапляє розрахований CV у прийнятний діапазон.

Процес оцінки полягає в наступному:

1. Оцінка починається з перевірки наявності достатньої кількості історичної інформації (консолідованих звітів на рівні системи) для розрахунку CV. Якщо даних недостатньо, оцінка пропускається.

2. CV розраховується для кожної обраної категорії серйозності.

3. Отримані значення CV порівнюються з відповідними порогами. Оцінка вважається виконаною лише у випадку, якщо значення CV є нижчим (або дорівнює) відповідному порогу для всіх категорій серйозності.

3.5. Експериментальна оцінка фреймворку на основі компромісу точності

Цей розділ представляє експериментальні дослідження, проведені для оцінки фреймворку тестування ефективності та продуктивності Java-додатків. Для інформування процесу розробки найкращих політик, спочатку було виконано оцінку виявлених компромісів.

Метою цього експерименту було оцінити потенційний компроміс між точністю результатів, згенерованих діагностичним інструментом, та

накладними витратами, які вводяться у вузли додатків процесами вибірки даних, що живлять цей інструмент.

Цей підрозділ деталізує розроблений прототип, тестове середовище та параметри, що визначали експериментальні конфігурації.

Прототип. Агент контролю був реалізований на базі Apache JMeter [2]. Агент вузла додатку та агент інструменту діагностики були реалізовані як автономні Java-додатки. Для забезпечення зв'язку між машинами, кожен агент внутрішньо використовував вбудований Jetty Web Servlet Container [15], що дозволяло комунікацію через HTTP-запити.

Було реалізовано дві початкові політики: політика збору даних з постійним інтервалом вибірки (SI) та політика завантаження з постійним інтервалом завантаження (UI).

Оскільки SI контролює частоту збору вибірок і є основним потенційним джерелом накладних витрат, був протестований широкий діапазон значень: 0.125,0.25,0.5,1,2,4,8 та 16 хвилин. Мінімальне значення (0.125 хв) було навмисно обрано меншим за рекомендоване для інструментів (0.5 хв), а максимальне (16 хв) — більшим за поширений у галузі SI (8 хв).

Оскільки UI не бере участі у процесі збору даних, використовувалося постійне значення 30 хвилин.

Експерименти проводилися в ізольованому тестовому середовищі, що складалося з восьми віртуальних машин (ВМ) :

- кластер з п'яти вузлів додатків з одним балансувальником навантаження.

- один сервер інструменту діагностики.

- один вузол тестувальника навантаження.

Всі ВМ мали 4 віртуальні CPU (2.20 ГГц), 3 ГБ RAM, 50 ГБ HDD; працювали під керуванням Linux Ubuntu та OpenJDK JVM з кучею 1.6 ГБ. Вузли додатків використовували Apache Tomcat [3]. ВМ розташовувалися на сервері Dell з KVM для віртуалізації.

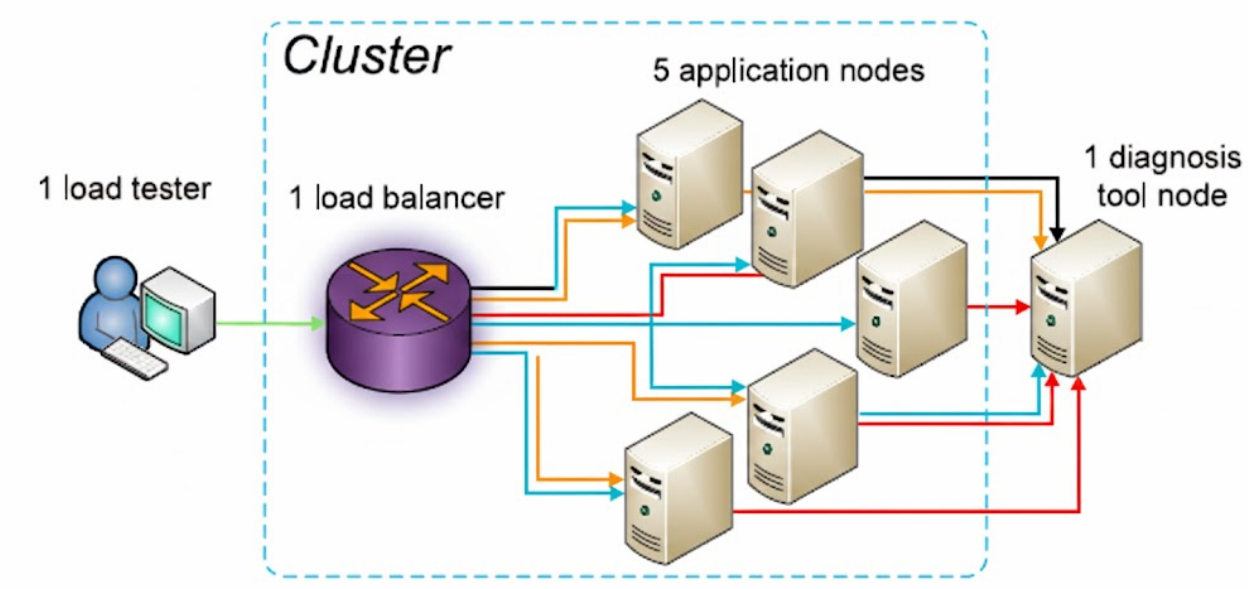

Рис. 3.4. Тестове середовище

Використовувалися п'ять інструментів: EMAT, GCLITE, GCMV, HC та WAIT, для диверсифікації оцінюваних поведінок.

Був обраний набір додатків DaCapo 9.12 [20], який пропонує широкий спектр поведінок. Для виклику програм DaCapo використовувалася розроблена обгортка JSP, встановлена в Tomcat на кожному вузлі. Тривалість тестування становила 24 години.

Критерії оцінки наступні:

- продуктивність за секунду (TPS) та час реакції (RT) (у мс), зібрані за допомогою JMeter.

- продуктивність тестування - кількість знайдених помилок та кількість знайдених критичних помилок, отримані зі звітів діагностики.

Результати для інструментів, заснованих на вибірках (WAIT, HC, EMAT), чітко продемонстрували наявність компромісу між вибором SI та вартістю продуктивності:

- продуктивність (TPS/RT) зменшується (час реакції збільшується) зі зменшенням SI. Наприклад, у випадку WAIT, генерація Javacore [23] передбачає тимчасове призупинення виконання процесів у JVM, що стає помітним при SI<0.5 хв.

- кількість виявлених помилок збільшується зі зменшенням SI. Це є прямим наслідком надходження більшої кількості вибірок до інструменту, що дозволяє провести більш детальний аналіз.

- найбільші накладні витрати спостерігалися у EMAT, оскільки він вимагає генерації heapdumps — процесу, відомого своєю трудомісткістю та значним впливом на продуктивність [19].

Ця поведінка зображена на рис 3.5 – 3.7, які підсумовують результати протестованих конфігурацій за інструментом.

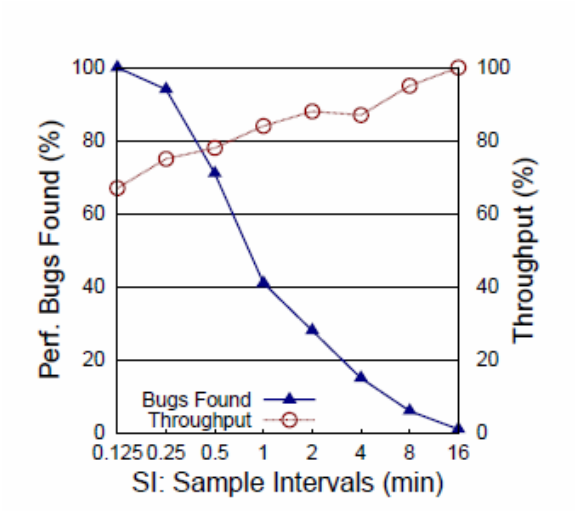

Рис. 3.5. Помилки продуктивності та продуктивності WAIT

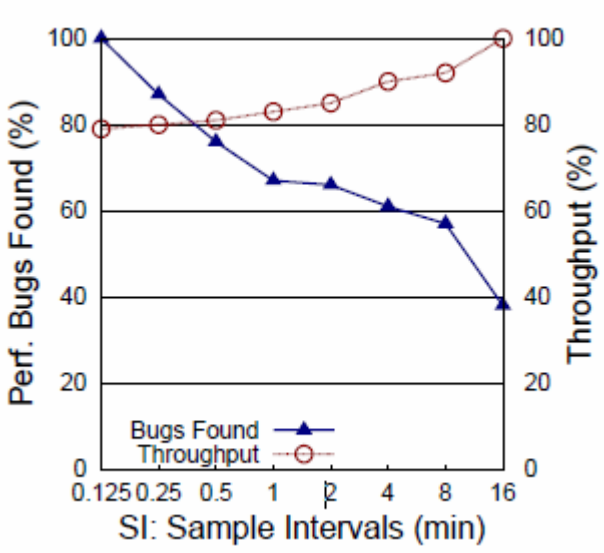

Рис. 3.6. Помилки продуктивності та продуктивності HC

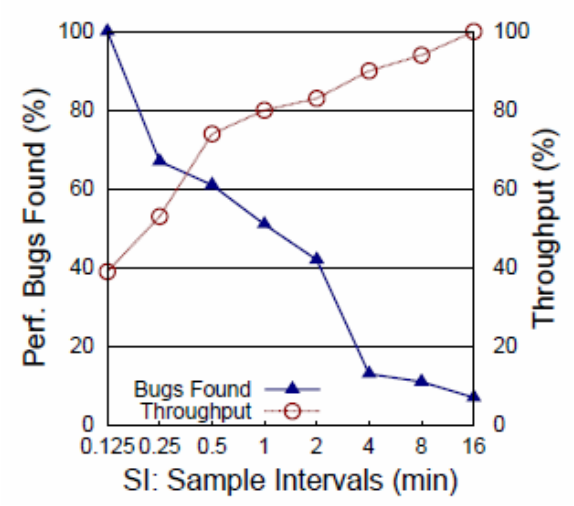

Рис. 3.7. Помилки продуктивності та продуктивності EMAT

Для інструментів, заснованих на трасуванні (GCLITE та GCMV), зв'язок між вибором SI та вартістю продуктивності був відсутній. Різниці в продуктивності були мінімальними, відносно постійними та незалежними від використаного SI, оскільки кількість згенерованих GC verbose залежить виключно від виконаної функціональності додатка.

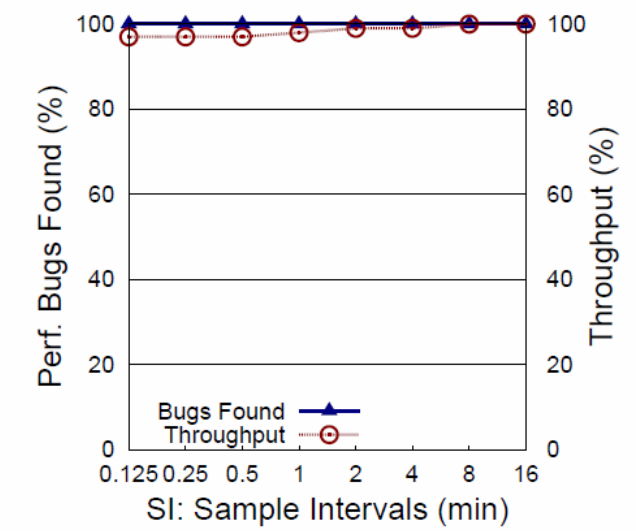

Рис. 3.8. Помилки продуктивності проти продуктивності GCLITE

Подібні поведінки спостерігалися і при аналізі лише критичних помилок, що підтвердило актуальність компромісу між SI та точністю результатів.

Кількість виявлених некритичних помилок була значно вищою, ніж критичних. Діагностичні інструменти, особливо при малому SI, мали тенденцію повідомляти про потенційні проблеми навіть при дуже низькій частоті (<1%), які, ймовірно, були частиною нормальної логіки.

Кількість нових виявлених помилок зменшувалася протягом виконання тесту, що свідчить про те, що тестовий запуск вичерпав свої переваги (з точки зору знайдених помилок) на певному етапі.

Ця поведінка візуально зображена на рис. 3.9 – 3.11, які показують розподіл помилок для WAIT, HC та EMAT відповідно.

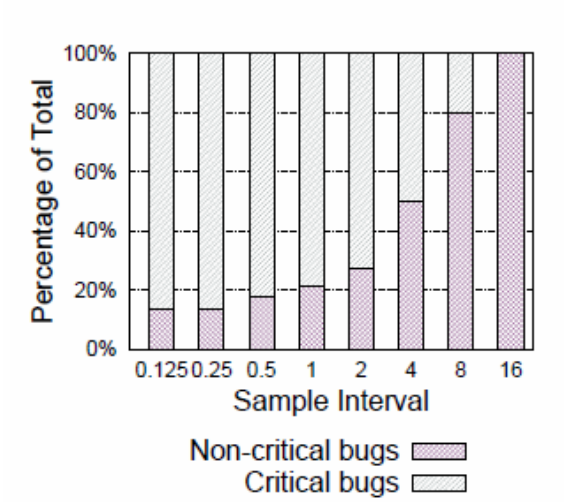

Рис. 3.9. Розподіл помилок продуктивності WAIT

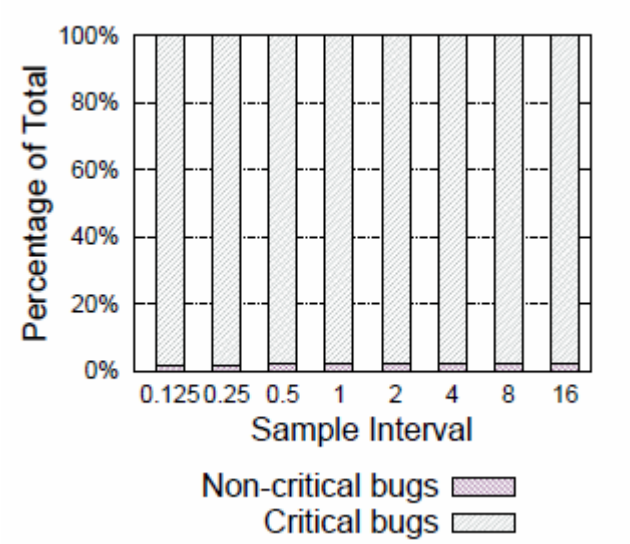

Рис. 3.10. Розподіл помилок продуктивності HC

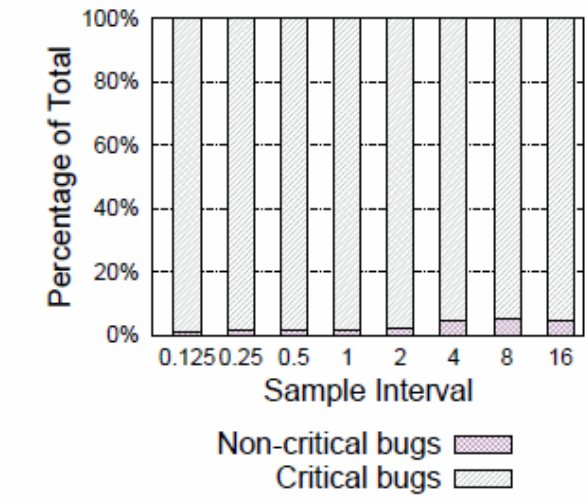

Рис. 3.11. Розподіл помилок продуктивності EMAT

Отже, результати підтвердили, що вибір SI суттєво впливає на накладні витрати продуктивності, які інструменти на основі вибірок вводять у додаток. Це ідентифікувало автоматичний вибір параметра SI як необхідну політику для ФТЕПД (що призвело до розробки політики збору даних, орієнтованої на точність). Для інструментів на основі трасування більш доцільним є використання постійного SI. Спостереження щодо розподілу помилок підкреслили важливість політики консолідації та оцінки Go No-Go.

3.6. Розробка методології побудови фреймворку тестування ефективності та продуктивності java-додатків

На рисунку 3.12 представлено пропоновану методологію побудови та оцінки фреймворку тестування ефективності та продуктивності java-додатків, яка є ітеративним процесом, поділеним на три основні фази: оцінка компромісів, розробка політик та допоміжних засобів, а також валідація та ітерація.

Фаза 1. Оцінка компромісів (Trade-off Evaluation).

Ця початкова фаза була спрямована на емпіричне визначення ключових компромісів (trade-offs) у процесі діагностики продуктивності, що керувало розробкою адаптивних політик. Вона включала три основні експерименти:

- Компроміс «Точність–Накладні Витрати» (Accuracy–Overhead):

Мета: Оцінити зв'язок між інтервалом вибірки (SI) та точністю результатів (кількість виявлених помилок) щодо накладних витрат, які вводяться процесами вибірки даних у моніторований додаток.

Результат: Встановлення необхідності адаптивного вибору SI для інструментів на основі вибірок.

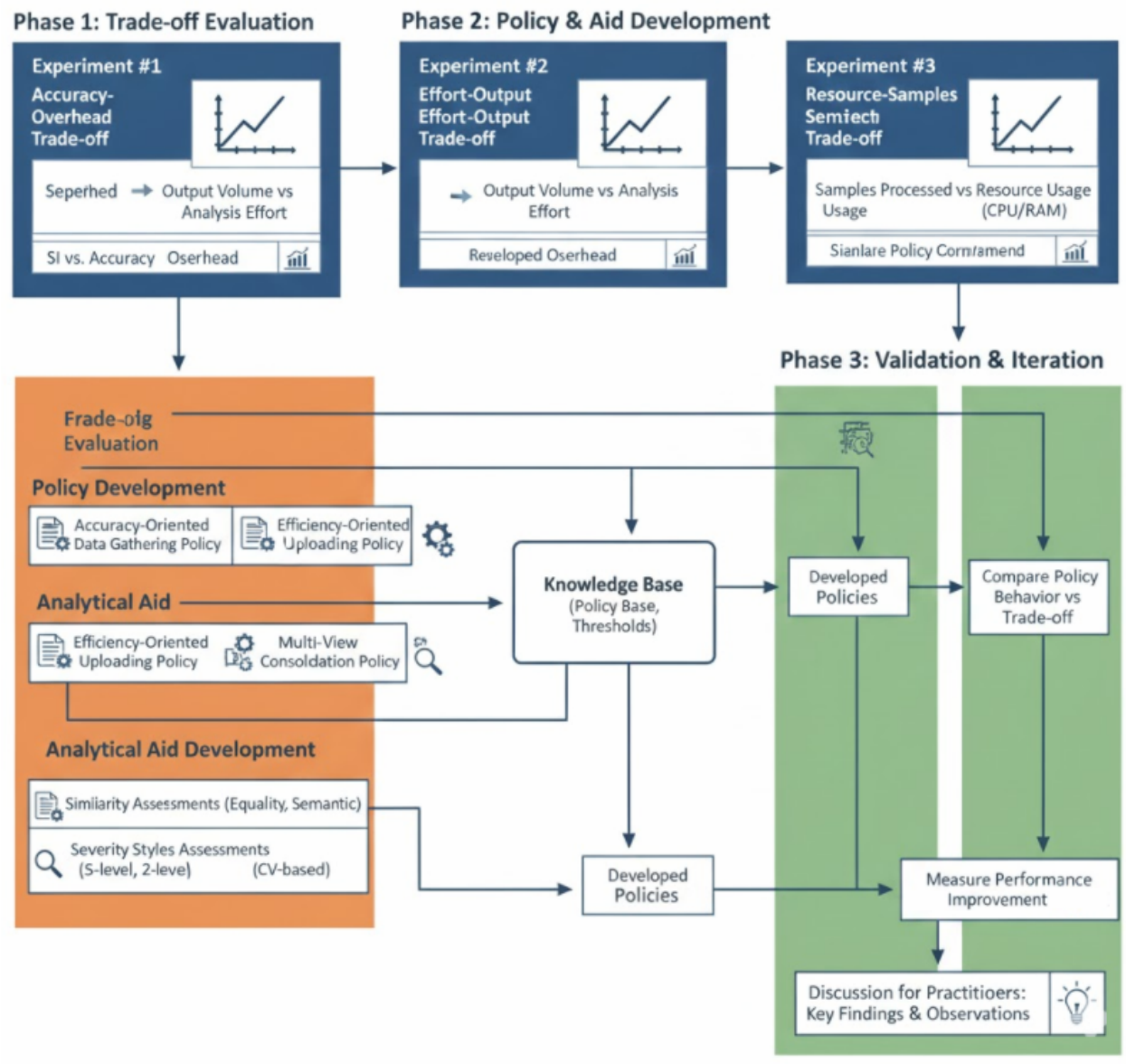

Рис. 3.12. Методологія побудови фреймворку тестування ефективності та продуктивності java-додатків

- Компроміс «Зусилля–Обсяг Виходів» (Effort–Output):

Мета: Оцінити людські зусилля, необхідні для аналізу виходів інструменту діагностики, відносно обсягу вихідних даних, згенерованих цим інструментом.

Результат: Визначення потреби у засобах консолідації та класифікації для зменшення когнітивного навантаження тестувальника.

- Компроміс «Ресурси–Вибірки» (Resource–Samples):

Мета: Оцінити використання ресурсів діагностичного інструменту (наприклад, CPU/RAM) відносно кількості одночасно оброблених вибірок.

Результат: Встановлення необхідності управління інтервалом завантаження (UI) для забезпечення ефективного спільного використання діагностичного сервісу.

Фаза 2: Розробка політик та аналітичних допоміжних засобів (Policy & Aid Development)

На цій фазі, на основі результатів Фази 1, було розроблено та імплементовано центральні компоненти ФТЕПД, які керуються даними.

Розробка політик (Policy Development):

- політика збору даних, орієнтована на точність спрямована на автоматичне адаптування SI для балансування точності та накладних витрат.

- політика завантаження, орієнтована на ефективність спрямована на адаптивне диференціювання UI для запобігання перевантаженню діагностичного сервісу.

Розробка аналітичних допоміжних засобів (Analytical Aid Development):

- політика консолідації з багатьма представленнями. Впровадження ієрархічної системи звітів (на рівні системи та вузла) для зменшення складності аналізу.

- оцінки подібності - реалізація механізмів оцінки рівності та семантичної подібності (наприклад, відстань Джаро-Вінклера) для усунення дублікатів проблем у звітах.

- стилі серйозності - розробка багаторівневих (наприклад, 5-рівневий ISTQB) та 2-рівневих стилів серйозності для класифікації проблем.

- оцінки Go No-Go - розробка критерію виходу на основі коефіцієнта варіації (CV), що дозволяє зупиняти тест, коли виявлення нових помилок стабілізується.

Усі розроблені політики, пороги та правила інтегруються у базу знань, яка є центральним елементом управління для MAPE-K циклу.

Фаза 3: Валідація та ітерація

Ця фаза включає експериментальну перевірку ефективності фреймворку:

- порівняння поведінки політик з компромісами. Оцінка того, наскільки добре реалізовані політики вирішують раніше виявлені компроміси (наприклад, чи дійсно адаптивна SI зменшує накладні витрати).

- вимірювання приросту продуктивності. Кількісна оцінка переваг використання ФТЕПД у реальному циклі тестування продуктивності порівняно з традиційними методами.

Результати цієї фази забезпечують зворотний зв'язок для ітеративного вдосконалення політик та допоміжних засобів, а також формують обговорення, де підсумовуються ключові висновки та спостереження для застосування у промисловості.

Висновки до розділу

У третьому розділі реалізовано прикладні методи та політики оптимізації ресурсів, спрямовані на підвищення ефективності Java-систем. Запропоновано методику автоматизованого налаштування інтервалу вибірки діагностичних даних з урахуванням впливу на продуктивність системи. Розроблено політику адаптивного регулювання навантаження інструментів діагностики, що дозволяє зменшити накладні витрати моніторингу. Запропоновано багаторівневу політику консолідації діагностичних звітів для

кластерних середовищ. Реалізовано аналітичні допоміжні засоби для оцінювання поведінки системи та виявлення аномалій продуктивності. Використано метрики подібності для аналізу змін у динаміці роботи Java-додатків. Запроваджено стилі серйозності, що підвищують інтерпретованість результатів діагностики. Сформовано критерії Go/No-Go для підтримки прийняття рішень щодо доцільності використання певних конфігурацій. Проведено експериментальну оцінку фреймворку з позиції компромісу між точністю та накладними витратами. Підтверджено практичну ефективність запропонованих методів у сценаріях високонавантажених Java-систем.

ВИСНОВКИ

У магістерській роботі на тему «Методи підвищення ефективності Java-систем шляхом оптимізації ресурсів, процесів та загальної масштабованості» здійснено теоретико-практичне дослідження проблем забезпечення високої продуктивності та ефективного використання обчислювальних ресурсів у сучасних розподілених Java-системах, що функціонують у середовищах з динамічним навантаженням та підвищеними вимогами до масштабованості й надійності.

У першому розділі роботи проведено ґрунтовний аналіз предметної області підвищення ефективності Java-систем, зосереджений на особливостях функціонування кластерних архітектур на базі JVM. Обґрунтовано доцільність застосування комплексного підходу до оптимізації продуктивності, який поєднує управління обчислювальними ресурсами, балансування навантаження та адаптивну діагностику. Проаналізовано ключові виклики діагностики продуктивності розподілених систем, зокрема обмеження сучасних інструментів моніторингу, їхній накладний вплив на продуктивність і складність інтерпретації зібраних метрик у кластерному середовищі. Значну увагу приділено фундаментальним аспектам віртуальної машини Java, механізмам управління пам’яттю та стратегіям збору сміття, які визначають поведінку системи під навантаженням і суттєво впливають на затримки, пропускну здатність та стабільність виконання. Узагальнення методологічного інструментарію, зокрема підходів до балансування навантаження, використання кількісних прогностичних моделей і концепцій автономних обчислень, дозволило сформувати теоретичне підґрунтя для подальшої розробки прикладних методів оптимізації.

У другому розділі представлено методологію тестування ефективності та продуктивності Java-додатків, яка базується на системному аналізі існуючих інструментів діагностики та стандартних бенчмарків. Визначено їхні переваги й недоліки в контексті високонавантажених та масштабованих

систем. На цій основі розроблено архітектуру фреймворку тестування ефективності та продуктивності Java-додатків, що орієнтована на модульність, розширюваність і мінімізацію впливу на досліджувану систему. Запропонована концептуальна структура фреймворку передбачає використання багатоагентної архітектури, що забезпечує децентралізований збір, агрегацію та аналіз продуктивнісних метрик. Детально описано основний процес функціонування фреймворку, механізми взаємодії агентів та внутрішню структуру компонентів, що дозволяє забезпечити гнучке масштабування та адаптацію до різних конфігурацій кластерних середовищ.

У третьому розділі реалізовано та експериментально перевірено методи й політики оптимізації ресурсів, спрямовані на підвищення ефективності Java-систем. Запропоновано методику автоматизованого налаштування інтервалу вибірки діагностичних даних з урахуванням їхнього впливу на продуктивність, що дозволяє досягти балансу між точністю моніторингу та накладними витратами. Розроблено політику адаптивного розподілу ресурсів інструментів діагностики, яка динамічно регулює інтенсивність збору даних залежно від поточного стану системи. Запропоновано політику багаторівневої консолідації діагностичних звітів для кластерних систем, що підвищує інформативність аналітики та знижує обсяг переданих даних. Додатково реалізовано аналітичні допоміжні засоби, зокрема оцінки подібності поведінки системи, стилі серйозності для виявлення проблем продуктивності та критерії Go/No-Go для підтримки рішень щодо прийнятності конфігурацій. Експериментальна оцінка запропонованого фреймворку підтвердила його ефективність з точки зору компромісу між точністю діагностики та накладними витратами, а також продемонструвала можливість практичного застосування в реальних сценаріях високонавантажених Java-систем.

Узагальнюючи результати дослідження, можна стверджувати, що поставлену мету магістерської роботи досягнуто. Запропоновані методи та політики оптимізації, а також розроблений фреймворк тестування

ефективності й продуктивності, становлять цілісне науково-практичне рішення для підвищення ефективності Java-систем у кластерних і розподілених середовищах. Практична цінність отриманих результатів полягає в можливості їхнього використання під час проєктування, налаштування та експлуатації високонавантажених Java-додатків, а наукова новизна — у поєднанні адаптивних методів діагностики, багатоагентних підходів і аналітичних моделей для забезпечення масштабованості та стабільної продуктивності. Перспективи подальших досліджень пов’язані з інтеграцією методів машинного навчання для прогнозування навантаження, автоматичного вибору стратегій оптимізації та розширення підтримки хмарних і контейнеризованих середовищ виконання.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Smith, J., Brown, T. Performance Engineering of JVM-Based Distributed Systems. Journal of Systems and Software. Amsterdam, Elsevier, 2019, Vol. 152, pp. 45–58.
2. Gupta, R., Verma, A. Adaptive Resource Management in Java Cluster Applications. IEEE Transactions on Parallel and Distributed Systems. New York, IEEE, 2020, Vol. 31, No. 7, pp. 1653–1666.
3. Tanenbaum, A., Van Steen, M. Load Balancing Techniques for Large-Scale Distributed Java Systems. Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS). Los Alamitos, IEEE Computer Society, 2018, pp. 412–421.
4. Lee, K., Park, S. Profiling and Monitoring Java Applications in Cloud Environments. Future Generation Computer Systems. Amsterdam, Elsevier, 2021, Vol. 115, pp. 305–317.
5. Williams, D., Miller, J. Garbage Collection Strategies and Their Impact on JVM Performance. ACM SIGPLAN Notices. New York, ACM, 2019, Vol. 54, No. 10, pp. 89–101.
6. Chen, Y., Li, H. Scalable JVM Tuning for High-Throughput Server Applications. Journal of Parallel and Distributed Computing. San Diego, Academic Press, 2020, Vol. 141, pp. 1–13.
7. Zaharia, M., Xin, R. Distributed System Performance Evaluation Using Adaptive Monitoring. Proceedings of the ACM Symposium on Cloud Computing. New York, ACM, 2018, pp. 150–162.
8. Novak, P., Kral, J. Multi-Agent Architectures for Performance Diagnostics in Distributed Systems. Computing and Informatics. Bratislava, Slovak Academy of Sciences, 2019, Vol. 38, No. 2, pp. 321–338.
9. Oracle Corporation. Java Virtual Machine Performance Optimization Techniques. Oracle Technical Journal. Redwood Shores, Oracle Press, 2021, Vol. 12, No. 3, pp. 22–35.

10. Hennessy, J., Patterson, D. Quantitative Analysis of Resource Utilization in Managed Runtimes. Communications of the ACM. New York, ACM, 2018, Vol. 61, No. 9, pp. 54–63.
11. Kumar, S., Singh, P. Benchmarking Java Applications Under Dynamic Workloads. Proceedings of the IEEE International Performance Computing Conference. Washington, IEEE, 2019, pp. 201–210.
12. Almeida, J., Silva, L. Performance Trade-offs in JVM Monitoring Tools. Software: Practice and Experience. Hoboken, Wiley, 2020, Vol. 50, No. 4, pp. 435–452.
13. Zhang, X., Zhou, M. Autonomic Computing Approaches for JVM-Based Systems. IEEE Access. New York, IEEE, 2021, Vol. 9, pp. 77891–77905.
14. Fowler, M., Lewis, J. Architectural Patterns for Scalable Java Systems. IEEE Software. Los Alamitos, IEEE Computer Society, 2018, Vol. 35, No. 6, pp. 12–19.
15. Costa, P., Sousa, J. Adaptive Sampling Techniques for Performance Monitoring. Journal of Cloud Computing. London, Springer, 2020, Vol. 9, Article No. 37, pp. 1–15.
16. Nguyen, T., Kim, D. Distributed Performance Diagnostics Using Agent-Based Models. International Journal of Distributed Sensor Networks. Thousand Oaks, SAGE Publications, 2019, Vol. 15, No. 8, pp. 1–12.
17. Ousterhout, J. Latency and Throughput Trade-offs in Managed Runtime Systems. Proceedings of the USENIX Annual Technical Conference. Berkeley, USENIX Association, 2018, pp. 203–216.
18. Li, Z., Huang, Q. Predictive Modeling for Resource Allocation in Java Clusters. Future Internet. Basel, MDPI, 2021, Vol. 13, No. 5, pp. 1–17.
19. Romano, P., Rodrigues, L. Performance Debugging of Large-Scale Java Applications. ACM Transactions on Software Engineering and Methodology. New York, ACM, 2020, Vol. 29, No. 3, pp. 1–28.

20. Becker, M., Weiss, G. Multi-Level Aggregation of Monitoring Data in Distributed Systems. Journal of Network and Computer Applications. London, Elsevier, 2019, Vol. 134, pp. 1–12.
21. Serrano, D., Perez, J. JVM Memory Management and Its Effect on Scalability. Concurrency and Computation: Practice and Experience. Hoboken, Wiley, 2018, Vol. 30, No. 24, pp. 1–14.
22. Liu, Y., Chen, J. Go/No-Go Decision Metrics for Performance Validation. IEEE Transactions on Software Engineering. New York, IEEE, 2021, Vol. 47, No. 11, pp. 2415–2428.
23. Hassan, S., Bahsoon, R. Self-Adaptive Performance Optimization in Cloud-Based Java Systems. Proceedings of the International Conference on Software Engineering (ICSE). Piscataway, IEEE, 2019, pp. 114–125.
24. Keller, A., Ludwig, H. Autonomic Performance Management in Distributed Applications. IBM Systems Journal. Armonk, IBM Press, 2018, Vol. 57, No. 2, pp. 321–334.
25. Park, J., Lee, S. Evaluating JVM Garbage Collectors in Multi-Core Environments. Journal of Systems Architecture. Amsterdam, Elsevier, 2020, Vol. 106, pp. 101–113.
26. Mendes, E., Mosley, N. Performance Measurement in Software Engineering. Empirical Software Engineering. New York, Springer, 2019, Vol. 24, No. 6, pp. 3412–3435.
27. Huang, L., Xu, C. Adaptive Load Balancing for Java Microservices. Proceedings of the IEEE International Conference on Cloud Computing. Washington, IEEE, 2021, pp. 88–97.
28. Brewer, E. Lessons from Building Scalable Distributed Systems. IEEE Internet Computing. Los Alamitos, IEEE Computer Society, 2019, Vol. 23, No. 4, pp. 46–52.
29. Kiczales, G., Mezini, M. Instrumentation Techniques for Runtime Performance Analysis. ACM Computing Surveys. New York, ACM, 2018, Vol. 50, No. 6, pp. 1–36.

30. Wang, H., Qian, Z. Scalable Performance Testing Frameworks for Java Applications. Journal of Software Testing, Verification and Reliability. Hoboken, Wiley, 2021, Vol. 31, No. 3, pp. 1–18.
31. Bosch, J. Software Architecture: The Next Step. Proceedings of the IEEE Working Conference on Software Architecture. Los Alamitos, IEEE Computer Society, 2019, pp. 194–203.
32. Kim, H., Lee, J. Cluster-Wide Performance Monitoring Using Distributed Agents. International Journal of Computer Systems Science and Engineering. London, Inderscience, 2020, Vol. 35, No. 1, pp. 15–27.
33. Cao, J., Andersson, M. Performance Modeling of Distributed Java Applications. Performance Evaluation. Amsterdam, Elsevier, 2018, Vol. 123, pp. 1–14.
34. Rolia, J., Cherkasova, L. Resource Management in Enterprise Java Systems. IEEE Computer. Los Alamitos, IEEE Computer Society, 2019, Vol. 52, No. 7, pp. 56–65.
35. Zhou, Y., Wu, K. Dynamic Sampling for Low-Overhead Performance Monitoring. ACM Transactions on Autonomous and Adaptive Systems. New York, ACM, 2020, Vol. 15, No. 4, pp. 1–25.
36. Lee, E., Seshia, S. Quantitative Methods for System Performance Evaluation. Proceedings of the IEEE Real-Time Systems Symposium. Piscataway, IEEE, 2018, pp. 12–23.
37. Rodrigues, L., Almeida, P. Distributed Monitoring Architectures for Modern JVM Systems. Journal of Distributed and Parallel Databases. New York, Springer, 2021, Vol. 39, No. 2, pp. 355–377.
38. Brown, A., Patterson, D. To Tune or Not to Tune: JVM Performance Optimization Revisited. ACM Queue. New York, ACM, 2020, Vol. 18, No. 4, pp. 30–44.