ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка методу та засобів розподіленого автоматизованого тестування програмного забезпечення»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
(код, найменування спеціальності)
освітньо-професійної програми «Інженерія програмного забезпечення»
(назва)

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

_____ Ростислав ІВЛЄВ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-64
Ростислав ІВЛЄВ

Керівник: Оксана ЗОЛОТУХІНА
д.т.н., доцент

Рецензент:
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Івлєв Ростислав Володимирович

1. Тема кваліфікаційної роботи: Розробка методу та засобів розподіленого автоматизованого тестування програмного забезпечення

керівник кваліфікаційної роботи Оксана ЗОЛОТУХІНА д.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023 р. №145.

2. Строк подання кваліфікаційної роботи «29» грудня 2023 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література.

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

1. Дослідження принципів керування автоматизованим тестування
2. Аналіз існуючих систем аналогів
3. Розробка системи керування автоматизованим тестуваннями

5. Перелік графічного матеріалу: *презентація*
   1. Бізнес процес керування автоматизованим тестуванням.
   2. Оптимізований бізнес процес керування автоматизованим тестуванням.
   3. Математична модель.
   4. Діаграма послідовності виконання автоматизованого тестування.
   5. Розроблена архітектура системи керування автоматизованим тестуванням.

6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу існуючих систем керування автоматизованого тестування	06.11-12.11.23	
3	Дослідження розподіленої архітектури	13.11-19.11.23	
4	Аналіз особливостей побудови систем з розподіленою архітектурою	20.11-26.11.23	
5	Застосування розподіленої архітектури у системі керування автоматизованим тестуванням	27.11-10.12.23	
6	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
7	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти _____ Ростислав ІВЛЄВ
(підпис)

Керівник
кваліфікаційної роботи _____ Оксана ЗОЛОТУХІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 73 стор., 2 табл., 10 рис., 21 джерел.

Мета роботи – оптимізація процесу керування автоматизованим тестуванням за рахунок використання брокерської архітектури.

Об'єкт дослідження – процес керування автоматизованим тестуванням.

Предмет дослідження – методи та засоби керування автоматизованим тестуванням.

Короткий зміст роботи: У роботі проведено аналіз проблем проведення автоматизованого тестування, який показав що основним недоліком цього процесу є відсутність у системам керування автоматизованим тестуванням можливості більш щільно взаємодіяти з зовнішніми приладами та можливості модифікування системи.

Це дуже ускладнює використання автоматизованого тестування під час розробки програмного забезпечення, оскільки сьогодні багато ПЗ розробляється для зовнішніх приладів. А відсутність можливості взаємодіяти з ними під час тестування обмежує кількість функції, тестування яких може бути автоматизоване.

А відсутність можливості модифікувати системи керування автоматизованим тестуванням лишає розробників шансу використовувати більш відповідні вимогам засоби автоматизованого тестування.

Проведений аналіз існуючих систем керування автоматизованим тестуванням, таких як TestRail, Qase, Allure Testops, Browserstack, і Compas, яка була розроблена для внутрішнього використання у компанії Activision, підтвердив наявність проблем та виявив залежність цих систем від CI/CD засобів. Це лишає цих систем гнучкості під час їх використання.

Огляд високорівневих архітектурних шаблонів, які використовуються для побудування систем, вказав на неможливість застосування монолітного підходу, оскільки його недоліки унеможливлює досягнути виконання поставлених вимог до системи керування автоматизованим тестуванням. Навпаки, розподілений

підхід дозволяє розміщувати компоненти на різних комп'ютерах. Це збільшує гнучкість системи та полегшує її масштабування, а також сприяє зручній інтеграції з тестовим обладнанням на різних рівнях. Тож розроблена модель використовує архітектуру на основі подій (англ. Event-Based Architecture), щоб забезпечити ефективну взаємодію між компонентами системи та підвищити її адаптивність до змін.

Кожен з головних компонентів системи, а саме сервер та агент, який виконує автоматизоване тестування, побудовані на основі мікроядерной архітектурі. Це забезпечує швидке розгортання, можливість до модифікування та до розширення.

Для збереження даних, які можуть бути представленні у вигляді таблиці, СУБД PostgreSQL. Ця реляційна база даних надає надійний та ефективний спосіб зберігання та організації даних, особливо у вигляді таблиць, що дозволяє легко взаємодіяти з інформацією, яка використовується у процесі тестування.

Але збереження даних про тестування складно представити у таблиці, що змушує використовувати нереляційна СУБД, а саме MongoDB. В ній дані зберігаються у вигляді BSON, що дозволяє зручно представляти різноманітні та змінювані структури даних, що часто зустрічаються у сценаріях тестування.

Реалізацію взаємодії між компонентами розподіленої системи може бути зроблена різними методами, але для системи керування автоматизованим тестуванням найкращим буде RPC, оскільки більшість запитів є схожими на виклик методу. А обраний gRPC ще й зумовлює бінарний тип запиту, що надає більш ефективну та швидку взаємодію між компонентами розподіленої системи.

Розроблена модель системи керування автоматизованим тестування має перевагу над існуючими в плані взаємодії з обладнанням, яке використовується автоматизоване тестування, та спроможністю до модифікування. Це робить систему гнучкою у використанні та адаптивною до вимог тестування ПЗ, яке розробляється.

КЛЮЧОВІ СЛОВА: ТЕСТУВАННЯ; CI/CD ЗАСОБИ; RPC; КЕРУВАННЯ; РОЗПОДІЛЕНІ СИСТЕМИ;

ABSTRACT

Text part of the master's qualification work:
73 pages, 10 pictures, 2 table, 21 sources.

The purpose of the work is optimizing with heuristic techniques the process of automated test management due to the use of broker architecture.

Object of research – the process of automated test management

Subject of research – methods and tools for management automated test

Summary of the work: At work, an analysis of the issues with automated testing was conducted, revealing a significant drawback in the lack of capability within test automation management systems to interact more closely with external devices and modify the system. This complicates the use of automated testing in software development, especially as much software is designed for external devices. The inability to interact with them during testing restricts the automation of testing for various functionalities.

Furthermore, the absence of the ability to modify test automation management systems limits developers from using more suitable testing tools according to their requirements. An examination of existing test automation management systems, such as TestRail, Qase, Allure Testops, Browserstack, and Compas (internally developed at Activision), confirmed these problems and highlighted their dependence on CI/CD tools, limiting their flexibility during utilization.

A review of high-level architectural patterns used in system construction indicated the impracticality of applying a monolithic approach due to its drawbacks in achieving the specified requirements for test automation management systems. Conversely, a distributed approach allows components to be placed on different computers, increasing system flexibility, scalability, and facilitating convenient

integration with testing equipment at various levels. Therefore, the developed model employs an Event-Based Architecture to ensure effective interaction between system components and enhance adaptability to changes.

Each of the main system components, namely the server and the agent responsible for automated testing, is built on a microkernel architecture, providing quick deployment, modifiability, and extensibility. For storing tabular data, PostgreSQL is utilized as a reliable and efficient relational database, facilitating easy interaction with information used in the testing process.

However, storing testing data in tabular form is challenging, necessitating the use of a non-relational database, specifically MongoDB. In MongoDB, data is stored in BSON format, allowing convenient representation of diverse and mutable data structures commonly encountered in testing scenarios.

The interaction between components in the distributed system can be implemented through various methods, but for a test automation management system, Remote Procedure Call (RPC) is ideal, given that most requests resemble method calls. The chosen gRPC additionally ensures a binary request type, providing more efficient and faster interaction between distributed system components.

The developed model of the test automation management system holds advantages over existing systems in terms of interaction with testing equipment and modifiability. This makes the system flexible in usage and adaptive to the requirements of the software being developed.

KEYWORDS: TESTING; RPC; CI/CD; MANAGEMENT; DISTRIBUTED SYSTEM

ЗМІСТ

ВСТУП 9

1 АНАЛІЗ СИСТЕМ КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ ... 11

1.1 Автоматизоване тестування 11

1.2 Проблеми використання автоматизованого тестування 14

1.3 Системи керування автоматизованим тестуванням 15

1.4 Огляд існуючих аналогів 16

1.5 Огляд аналогів які створенні для внутрішнього використання 23

1.6 Аналіз оглянутих рішень 26

1.7 Постановка задач 28

РОЗДІЛ 2 МОДЕЛЬ СИСТЕМИ КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ 30

2.1 Архітектура системи 30

2.2 Реляційні СУБД 44

2.3 Нереляційні СУБД 52

2.4 Технологія віддаленого виклику процедури 57

РОЗДІЛ 3 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ 64

3.1 Бізнес процес розробленої системи 64

3.2 Процес виконання автоматизованого тестування 65

3.3 Математична модель оцінки системи 68

ВИСНОВКИ 71

ПЕРЕЛІК ПОСИЛАНЬ 72

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація) 74

ВСТУП

Сьогодні все більше поширюється автоматизоване тестування, а методи керування цим процесом залишається не дуже розвинутим. Більшість існуючих рішень віддають цю функцію CI/CD засобам, а самі виконують функцію відображення та аналізу результатів.

Цей підхід відображає тенденцію до інтеграції автоматизованого тестування у засоби CI/CD. Однак це також може призводити до недоліків, таких як обмежена гнучкість управління тестовим процесом та інколи до потреби у конфігурації тестових сценаріїв під особливості CI/CD платформ.

Важливо відзначити, що такий підхід може обмежувати взаємодію з пристроями або ресурсами, що може бути критичним для тестування деяких аспектів програмного забезпечення, зокрема, в тих випадках, коли взаємодія з конкретним обладнанням є важливою частиною тестових сценаріїв.

Зараз існує значна кількість засобів та фреймворків, які використовуються в автоматизованому тестуванні. Однак існуючі системи часто обмежені у здатності до модифікування, що може призводити до ряду негативних наслідків.

Перш за все, обмежена здатність до модифікації ускладнює адаптацію тестових сценаріїв до змін в програмному забезпеченні. Із зростанням вимог до програмного продукту стає важко швидко та ефективно ввести зміни в існуючі тестові скрипти.

Крім того, це може ускладнити впровадження нових функцій у тестовий набір або використання нових інструментів для покращення ефективності тестування. Такі обмеження можуть стати перешкодою для тестувальників та розробників у реалізації сучасних методів та стратегій тестування.

Метою дослідження є оптимізація процесу керування автоматизованим тестуванням за рахунок використання брокерської архітектури.

В процесі дослідження вирішити наступні завдання:

1. Проаналізувати існуючих систем керування автоматизованим тестуванням.

2. Сформулювати вимог до системи на основі аналізу існуючих аналогів.
3. Провести аналіз високорівневих архітектурних шаблонів та обрати найбільш відповідні для всієї системи та для кожного з її компонентів.
4. Модифікувати процес керування автоматизованим тестуванням за допомогою обраних шаблонів.
5. Розробити математичну модель для оцінки систем керування автоматизованим тестуванням.
6. Проаналізувати розроблену модель та порівняти з існуючими аналогами.

Об'єктом дослідження є процес керування автоматизованим тестуванням. Він включає в себе організацію, координацію та виконання автоматизованих тестів для програмного забезпечення.

Предметом дослідження є методи та засоби керування автоматизованим тестуванням.

1 АНАЛІЗ СИСТЕМ КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ

1.1 Автоматизоване тестування

Процес розробки програмного забезпечення ніколи не відбувається без тестування. Це допомагає зменшити ризики випуску недоробленого продукту та не потрапити в безкінечний цикл розробки, оскільки тестування також перевіряє чи відповідає ПЗ, яке розробляється, вимогам які були сформульовані на початку.

Тестування поділяється за способом його виконання на ручне тестування (англ. Manual testing) та автоматизоване тестування (англ. Automated testing).

Ручне тестування передбачає пряму взаємодію працівника з програмним забезпеченням. Під час ручного тестування застосовується додаткові засоби для проведення тестування API, відслідковування продуктивності та для інших потреб.

Автоматизоване тестування, на відміну від ручного, надає можливість проводити цей процес віддаллено. При автоматизованому тестуванні використовуються спеціальні засоби і фреймворки для написання тестів та виконання їх.

На даний момент зазвичай виділяють три основних види автоматизованого тестування, а саме:

— Модульне тестування – це процес тестування кожного модуля програми, де модуль це окрема функція. Тести у більшості випадках пишуться самими розробниками використовуючи ту ж саму мову програмування яку використовували для створення відповідного модуля.

— Тестування API перевіряє чи вірно працює API та для автоматизації цього процесу може бути використанні додаткові засоби.

— Тестування графічного інтерфейсу користувача орієнтовано на перевірку правильності роботи графічного аспекту програми. Дуже часто

підготовка та виконання тестів потребує використання додаткових засобів, тому що процес тестування GUI дуже не простий, а розмір тестових скриптів великий.

Окрім основних видів використовуються додаткові на різних рівнях програмного забезпечення в залежності від специфіки проекту. Це спричиняє потребу використовувати велику кількість додаткових систем та засобів для виконання тестів.

Типовим випадком коли є доречним використання автоматизованого тестування це перевірка функціоналу програми під час якої виконуються однакові дій та результат яких буде сталим. Процес виконання автоматизованих тестів не змінюється в залежності від людини, яка його виконує, або комп'ютера, де він виконується, тому, що тестові скрипти, який містить послідовність дій та написаний з використанням спеціальних фреймворків, створений заздалегідь. Але якщо тестовий скрипт був написаний невірно, то результати після його виконання будуть не коректні, що може значно вплинути на якість програмного забезпечення. Щоб запобігти цьому автоматизоване тестування майже ніколи не використовується без ручного.

Після написання тестових скриптів автоматизоване тестування, на відміну від ручного тестування, може працювати майже без втручання людини. Це надає змоги виконувати його в будь який час дня та дистанційно через, що автоматизоване тестування може дозволити компаніям більш ефективно використовувати свої ресурси, наприклад, долучати вільні комп'ютери до тестування вночі. Ця особливості надає змогу отримувати розробникам більш актуальні дані про стан програмного забезпечення, яке розробляється, та запобігти появі критичних помилок.

Автоматизоване тестування, а саме вже написані тестові скрипти мають здатність до повторного використання, тобто, його не треба писати кожен раз новий, а просто написати один та потім виконувати коли потрібно. Що дозволяє зекономити багато часу, але сам процес написання тестового скрипта може

займати багато часу, оскільки деякі тестові сценарії можуть потребувати виконання великої кількості дії.

Підготовка до автоматизованого тестування займає багато часу, так що вона потребує написання тестових скриптів, наприклад, підготовка до тестування графічного інтерфейсу користувача займає на 65% більше часу ніж підготовка до ручного тестування [1]. Але процес виконання в 18.7 разів швидше у автоматизовано тестування ніж у ручного. У таблиці 1 наведені результати експерименту на основі якого базуються надані цифри про час виконання тестування та підготовки до нього.

Таблиця 1.1

Результати порівняння автоматизованого тестування GUI
та ручного тестування GUI [1]

	Час підготовки, год			Час виконання, год		
	Avg	Min	Max	Avg	Min	Max
Ручне тестування	19.2	10.6	56.0	0.21	0.1	1.0
Автоматизоване тестування	11.6	10.0	24.0	3.93	0.5	24.0

Сьогодні існують засоби для прискорення створення тестових скриптів для тестування графічного інтерфейсу користувача, але переважна більшість їх орієнтується на веб-застосунки. Це робить їх не підходящими для тестування програмного забезпечення яке не використовує інтернет браузер для відображення свого графічного інтерфейсу користувача.

Також підготовка даних для автоматизованого тестування займає не малий час через, що почати використовувати автоматизоване тестування на всіх рівнях програмного забезпечення одночасно майже неможливо на відміну від ручного. Але час його виконання та відсутність необхідності людині яка виконує тестові скрипти, які вже були написані, розбиратися в принципі роботи ПЗ для того щоб провести тест надає перевагу в часі виконання над ручний. Підготовка

автоматизованого тестування відбувається один раз для модулю до тих пір поки функціонал модулю не зміниться, а виконання відбувається будь коли за потреби.

1.2 Проблеми використання автоматизованого тестування

Опитування про види викликів під час тестуванні програмного забезпечення показує, що один з тяжких напрямків тестування, де більше всього є викликів, це автоматизоване тестування [2], [3]. Основні питання, які виникають при використання автоматизованого тестування, пов’язані з засобами і фреймворками.

Автоматизоване тестування має незчислену кількість засобів, застосунків та фреймворків які покликані спростити процес імплементації та використання його. Але часто під час розробки програмного забезпечення розробники вимушені застосовувати декілька видів застосунків та фреймворків для впровадження автоматизованого тестування [2] тому, що на сьогоднішній день існує багато видів ПЗ та операційних систем і девайсів на яких воно повинно працювати. Це навпаки спричиняє ускладнення процесу використання та розгортання їх. А недостатня кількість методів підтримка та керуванням засобами автоматизованого тестування, що робить процес використання їх не зручним [3].

Також один з складних моментів використання автоматизованого тестування є налаштування засобів та комп’ютерів [3], так що сам процес може займати багато часу. А при використанні вже налаштованих може виникнути проблеми через, що результати тестування будуть не коректі або взагалі будуть відсутні [3]. При цьому, методів відслідкувано налаштувань тестових комп’ютерів достатньо мало, та, зазвичай, це відбувається через додаткові засоби, що додає обмежень у процес використання автоматизованого тестування.

Існує мала кількість методів взаємодії з обладнанням і пристроями під час автоматизованого тестування [3]. Це спричиняє потребу створювати їх під час розробки програмного забезпечення через, що ефективність використання ресурсів на проекті знижується, а час розробників іноді достатньо дорогий ресурс. А створені засоби зазвичай мають не самий велику подовженість життя, оскільки

в більшості випадків вони розробляються для внутрішнього використання та не мають великої кількості оновлень. Вони часто втрачають актуальність з часом тому, що після їх розробки, їх розвитком ніхто не займається.

1.3 Системи керування автоматизованим тестуванням

Системи автоматизованим керування тестуванням – це системи, які покликані спростити процес керування як ручного тестування так й автоматизованого. Вони маю обширний функціонал який зазвичай спрямований на формування та представлення звітів по результатам проведених тестувань. Але не тільки, тому усі їх функції можна класифікувати на три види за типом тестування на який вони спрямовані, а саме: для автоматизованого тестування, для ручного тестування та для обох типів.

Всі функції, які спрямовані для використання з обома типами тестування одночасно, переважно є візуальними. Найбільш використовуваною такою функцією є створення, редагування, виконання та зберігання чек листів, тест кейсів та різного розміру тест планів [1]. Також виконані тест плани надають можливість відслідковувати коли вони були виконані та іноді скільки часу на це було витрачено [1].

Друга важлива функція, яка також використовується однаково часто, як з ручним тестуванням, так й автоматизованим – це відслідковування помилок, які були додані до тест плану при його виконані [1]. Це надає змогу бачити які помилки виникали при попередньому виконані тест кейсу або чек листа.

Звісно всі системи керування тестуванням приділяють велику кількість уваги до створення звітів про пройдені тести. Вони зазвичай мають велику кількість налаштувань для цього, але в більшості випадках метрики в цих звітах будуть використовуватися однакові.

Одна з функції, яка підлягає тільки під використання під час автоматизованого тестування, повинна була б вирішити деякі проблеми з використанням цього типу тестування. Ця функція – це виконання

автоматизованого тестування [1]. Її реалізації часто ускладняється через, те що більшість систем керування автоматизованим тестуванням є веб-застосунками, а тестування треба проводити на віддалених комп’ютерах.

Також ці системи надають функціонал автоматичного збирання результатів та форматування їх під час або після виконання автоматизованого тестування [1], що спрощує відслідковування його статусу.

1.4 Огляд існуючих аналогів

TestRail є одною з найпопулярніших систем керування автоматизованим тестуванням. Її використовують великі провідні компанії з різних сфер та галузь. Також має велику кількість різних нагород [4]. Вона надає достатньо широкий спектр функції, який допомагає при виконанні тестування.

В основному TestRail робить акцент саме на управління ручним тестуванням, яке можна поділити на два основних види створення та виконання. До створення входить створення та редагування тест кейсів, які можуть у TestRail можуть бути як простими елементами чек-листу, так й повноцінними тест кейсами, які вміщують в себе достатньо великий обсяг інформації [4].

Також до створення входить створення тест планів, який представляє собою набір тест кейсів. А набір тест планів, який також можна створити у TestRail, називається набором тестів (англ. Test Suite) [4]. До набору тестів входять тільки тест план, а тест кейси можуть входити тільки до тест плану.

На старті виконання тестування TestRail надає вибір з тест плану та набору тестів. При виборі тест плану надається можливість відключення та включення тест кейсів, які будуть входять у обраний тест план [4]. А при виборі набору тестів окрім тест кейсів, які входять в тест плани цього набору тестів, ще додатково можна додавати або прибирати тест плани.

Всі виконані тести зберігаються та їх результати легко можуть бути відслідкувано, оскільки історія виконання відображається також для кожного тест кейса в плані до того часу поки його розташування не зміниться [4].

TestRail може формувати звіти на основі п’ять основних даних, а саме: тест кейси, помилки, результати тестування, загальний та звіт по користувачеві [4]. Більшість з них має декілька підтипів. Кожен з звітів допомагає отримати дані про той чи інший аспект, який задіяний у тестуванні.

Для взаємодії з автоматизованим тестування TestRail надає два різні методи. Один з них це використання інтерфейс командного рядка (CLI), який надає TestRail, для додавання результатів до TestRail [4]. Цей метод у TestRail має два підходи, які відрізняються за способом створення тест кейсів.

Один з підходів – це code-first, при якому тест план та тест кейси автоматично генеруються. Цей підхід може спричинити дублювання тейст кейсів.

Інший з підходів – це specification-first, який фокусується на створенні документації, а саме тест кейсів і тест планів. Спочатку створюється тест план з кейсами та зіставлення тест скриптів з тест кейсами в коді.

На рисунку 1.1 зображено процес виконання TestRail CLI з code-first підходом, а на рисунку 1.2 з підходом specification-first.

Другий метод – це використання API для створення тест планів та тест кейсів з під коду за допомогою HTTP запиту. Всі TestRail API запити використовують JSON формат [4].

TestRail має реалізовану інтеграцію з достатньо великою кількістю CI/CD засобами та фреймворками для автоматизованого тестування. Що допомагає достатньо швидко впровадження у проект.

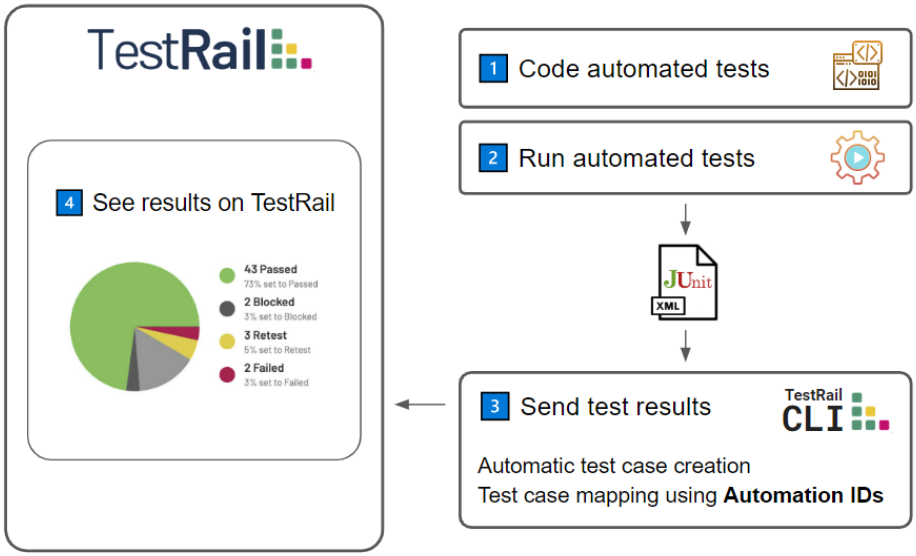


Рис 1.1 Code-first підход використання TestRail CLI [4]

Слід зауважити, що TestRail немає можливість для розширення свого функціоналу.

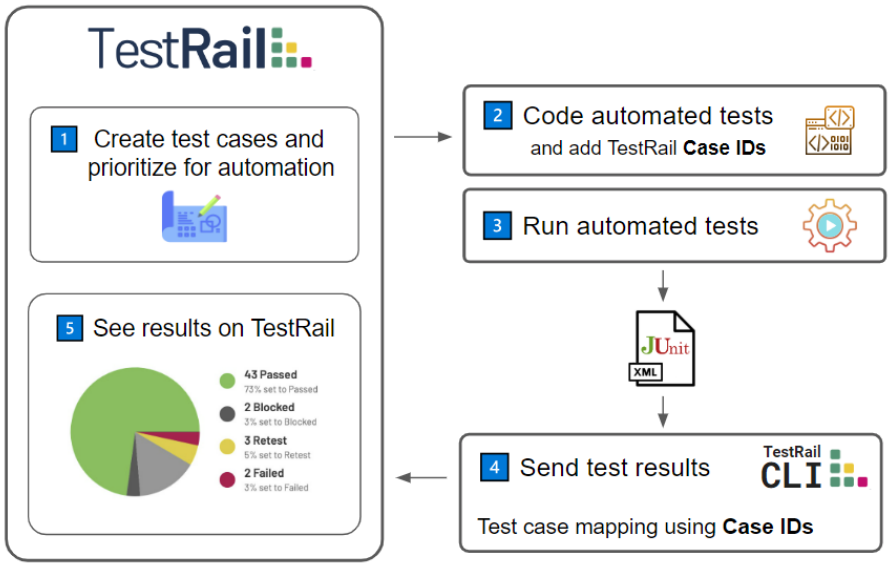


Рис 1.2 Specification-first підхід використання TestRail CLI [4]

Qase менш популярний ніж TestRail, але його також використовують деякі великі компанії. Він має достатньо велику кількість функцій для керування як ручним тестуванням, так й автоматизованим.

Створення тестової документації, а саме тест планів, які тут називаються наборами тестів (англ. Test Suite) та тест кейсі достатньо просте, але водночас обширне. Також створення тест кейсів має декілька унікальних полів такі, як:

— статус автоматизації, яке відповідає за стан автоматизації тест кейса, може бути автоматизований, буде автоматизований, не буде автоматизований;

— рівень, вказує який саме рівень програмного забезпечення тестується.

Також Qase має можливість створювати спільні тестові шаги, що допомагає зменшити час написання тест кейсів. Вони створюються один раз й потім можуть бути додані до тест кейсів, які створюється, що дозволяє не писати однакові шаги від тест кейсу до тест кейсу.

Процес створювання тест кейсів тут розвинутий та може виконуватися в декілька етапів. При створенні нового тест кейса його можна відправити на перевірку, що створить запити на цю дію, а відповідальна людина вже вирішить зберегти, запросити змінити або змінити самостійно. Використання цієї функції може збільшити кількість коректних кейсів та підвищувати вміння написання тестових кейсів у неопитних співробітниках.

При старті тестування у Qase є додаткове поле «Оточення», яке вказує де саме повинно проводитися тестування. Види оточення створюється самим користувачем у відповідній вкладці та зберігає назву оточення, скорочену назву, опис та посилання.

Для аналізу написаних тест кейсів, запущених тестів, результатів тестів, тест планів, помилок, які були прикріплені до тест кейсів при виконання тестування, Qase має свою мову запитів QQL [5]. Це допомагає робити більш гнучку вибірку для аналіз стану проекту, тестового покриття, кількості помилок та іншого. Також Qase має систему дошок куди можна вивести показники по проекту. Це все робить Qase зручним для проведення аналізу та відслідковування змін у якості програмному забезпечення під час розробки.

Інтеграція з CI/CD засобами у Qase дозволяє не тільки створювати тест плани у системі, а надає змоги саме запускати автоматизоване тестування на веб-

сторінці системи. Це робить користування системою керування автоматизованим тестування зручнішим, але Qase підтримує не велику кількість CI/CD засобів, а саме: Jenkins, GitHub, GitLab, BitBucket [5]. Якщо Jenkins і BitBucket дозволяють тільки запускати автоматизоване тестування з Qase, то GitHub і GitLab, які мають функціонал пов’язаний з відслідковування багів, надають можливість створювати звіти о помилках.

Взаємодія з фреймворками для написання тестових скриптів і отримання результатів відбувається зі сторони коду за допомогою плагінів до них. Qase підтримує понад 30 фреймворків та щонайменше з 5 мов програмування [5].

Також Qase має інтеграцію зі Slack, що дозволяє інформувати про великий різновид дії, які відбуваються у системі, та з системами відслідковування помилок такі, як Jira, що дозволяє прив’язувати знайдені помилки до тест кейсів у відповідному активному наборі тесті. Але хоча Qase має велику кількість засобів з якими він може взаємодіяти, він не має можливості до збільшення функціоналу зі сторони сторонніх розробників крім як взаємодії за допомогою його API.

Allure Testops позиціонує себе, як система керування тестування, яка спрямована на автоматизацію більшості частин процесу тестування з щільним зв’язком з DevOps засобами. В ній, при інтеграції з CI/CD засобами, робиться наголос на більш функціональний зв’язок між ним ніж у більшості аналогів через, що розробники Allure Testops використовують двох сторонній зв'язок між CI/CD засобами та системою [6] при, якому не тільки одна система відправляє дані іншій, а обидві системи обмінюються даними. Це дозволяє збільшити кількість можливості системи до взаємодії з автоматизованим тестування через CI/CD засоби та їх користь, а саме, окрім простого отримання результатів та запуску автоматизованого тестування, Allure Testops має:

— розширений функціонал запуску автоматизованого тестування, що дозволяє запускати та перезапускати автоматизоване тестування тоді коли користувачеві треба [6];
— можливість запускати в роботу не увесь конвеєр команд, а тільки потрібний процес [6], що надає гнучкості у використанні системи та

може допомогти зберегти час, тому що налаштований конвеєр команд зазвичай має усі CI/CD етапи, що робить його затратним по часу;

— можливість отримувати результати в реальному часі [6] – це дозволяє відслідковувати процес виконування автоматизованого тестування і його стан майже без затримок під час його здійснювання;

— можливість запускати різні набори автоматизованих тестів та окремих автоматизованих тестів паралельно на різних середовищах [6]. Це надає змоги проводити автоматизоване тестування розподілених систем в рази зручніше, а також при виконанні автоматизоване тестування воно дозволяє використати ресурси розробників більш ефективно.

На даний момент Allure Testops підтримує 10 засобів CI/CD таких, як Jenkins, TeamCity, Azure DevOps та інші, всі вони мають інтеграцію з двох стороннім зв’язком та з реалізацією усіх функції, які надає систем [6]. Також ця система має інтеграцію з системами відслідковування помилок, яка дозволяє додавати посилання на звіти про помилку до тест кейсі, створювати звіти, експортувати тест кейси, але кількість реалізованих функції залежить від інтеграції через, що не всі інтегровані системи відслідковування помилок мають увесь функціонал, який Allure Testops дозволяє мати.

Allure Testops має ще інтеграцію з іншими системами керування тестуванням, що дозволяє додавати посилання на тест кейси, створювати тест кейси, експортувати їх [6]. Система зараз підтримує 7 інших систем керування тестуванням, але функціонал може відрізнятися.

Вона має декілька вкладок, які вказують на її розвинену взаємодію з CI/CD засобами, а саме запусків та робот. Вкладка запусків має інформацію про здійснювані та вже виконані автоматизоване тестування, його статус та результати. Тести на цій вкладці можуть бути у двох станах [6]:

— відритий, який вказує, що автоматизоване тестування в процесі;

— закритий – це стан коли Allure Testops починає створювати або оновлювати набори пройдених тестів, заповнювати їх результатів та додавати їх у статистику.

Також є можливість додавання результатів тестів з вже закінченого автоматизованого тестування, але тільки з тих які підтримується системою.

Вкладка робот дозволяє взаємодіяти з підключеними CI/CD засобами до проекту, а саме запускати тестування, налаштовувати параметри з якими воно буде запускатися, додавати нові роботи.

Система також має всі можливості для керування ручним тестуванням такі, як створення тест кейсі, тестових планів, запускати їх та інші. Так само як Qase, ця система має спільні тестові кроки та дошки, які допомагають у аналітиці стану проекту, тут є як стандартна, яка заготовлена заздалегідь, та можна робити свої.

Browserstack надає засоби для тестування веб-сторінок та мобільних застосунків для Android та iOS систем. Одним з її головних особливостей це тестування у хмарі з доступом до більше ніж 3000 телефоні та браузерів [7]. Окрім цього Browserstack має декілька додаткових сервісів для покращення та полегшення різних аспектів тестування.

Для тестування доступності розроблюваного додатку є спеціалізована додатковий сервіс, який аналізує графічний інтерфейс користувача та показує які його частити мають проблеми, а також надає поради як його доступність покращити [7]. Вона також надає можливість перевірити адаптивність додатку до різних систем.

Також Browserstack має сервіс для порівняння графічного інтерфейсу користувача, як мобільного додатку так й веб-застосунку, різних версій програмного забезпечення. Воно візуалізує графічний інтерфейс користувача застосунку та робить скріншоти, якщо до цього робилися скріншоти попередніх версії застосунку то цей сервіс порівнює їх. Функціонал цього застосунку націлений для проведення тестування та огляд змін перед додавання їх у застосунок.

Browserstack підтримує велику кількість фреймворків для автоматизації тестування веб-застосунків, мобільних додатків і SmartTV додатків [7]. В залежності від фреймворку Browserstack надає різний додатковий функціонал, але всюди є паралельне автоматизоване тестування у хмарі, яке дозволяє зберегти

багато часу. Інтеграція з CI/CD засобами теж залежить від фреймворку для написання тестів, але функціонал всюди однаковий, а саме відправлення результатів. Запуск автоматизованих тестів відбувається тільки через CI/CD засоби [7]. Також він має власний інтерфейс командної строки, який дозволяє імпортувати результати пройдених автоматизованих тестів.

Для ручного тестування Browserstack має функціонал створення тест кейсів, створення наборів тестів, запуску їх та збереження історії пройдених. А для аналізу результатів тут також присутні дошки, які мають достатньо великий різновид віджетів, а також Browserstack має сервіс, який допомагає аналізувати результати автоматизованих тестів, що робить зручнішим знаходження помилок у розроблюваного додатку та у автоматизованих тестів [7].

1.5 Огляд аналогів які створенні для внутрішнього використання

Деякі компанії створюють системи керування автоматизованим тестування для внутрішнього використання через браку функціоналу існуючих на ринку аналогів. В більшості випадках це відбувається при потреби проводити автоматизоване тестування з використанням зовнішніх прикладів, наприклад, ігрові консолі або тестовий стенд панель приладів машини, які часто використовуються при розробки програмного забезпечення для автомобілів. Також така потреба виникає при запиті на збір додаткових метриків.

Одне з таких рішень є Compass – це внутрішня система Activision на основі BuildBot [8]. Цю систему було створення через потреби проводити автоматизоване тестування на ігрових консолях на рівні з комп’ютером.

BuildBot являє собою засіб CI/CD або система для планувати виконання робіт [9]. Він є безкоштовним та має відкритий вихідний код. Він написаний на Python та має модульну архітектуру, що дозволяє адаптувати його до потреб конкретного проекту.

В архітектурі BuildBot є одна головний вузол, який називається майстер збірки (англ. Build Master), до якого приєднуються робітники (англ. Worker), які

виконують заготовлені інструкції. Майстер збірки вирішує все в цій системі, а саме де, що та коли буде збиратися [9].

Майстер збірки у BuildBot зазвичай починає збірку коли у системі керування версіями з'являються зміни, вій відправляє інструкції до робітників, які повинні бути виконані. Коли робітник закінчує виконання даної інструкції, він відправляє результат до майстра збірки. Цей процес відбувається до тих пір поки збірка не закінчиться, а після закінчення статус збірки змінюється майстром збірки та за допомогою репортера повідомляє розробників про результат збірки [9]. На рис. 1.2 зображена архітектура BuildBot.

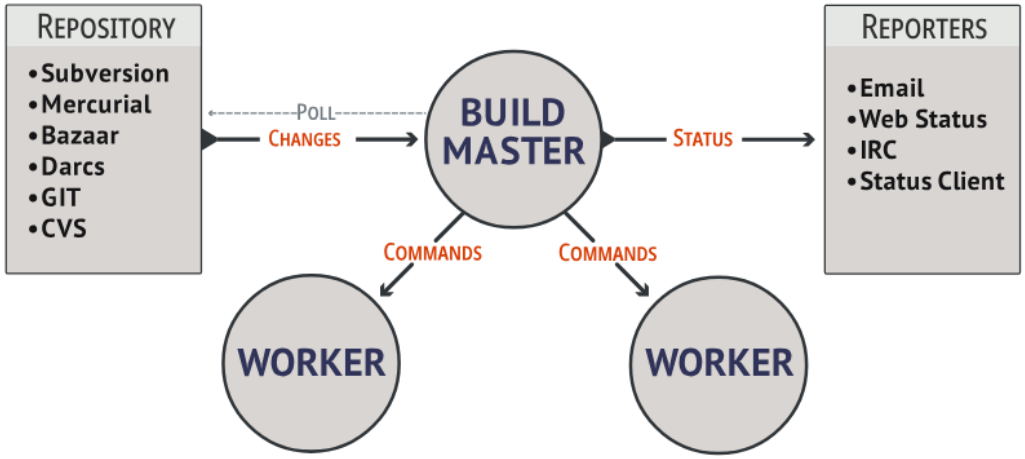


Рис 1.2 Архітектура BuildBot [9]

Робітники отримують вихідний код для збірки не від майстра, а напряму з систем керування версіями, що надає багато можливостей. Ця особливість одна з тих через, які розробники Compass обрали саме BuildBot.

BuildBot позиціонує себе не тільки, як CI/CD засіб, а ще, як засіб для автоматизації різних процесів. Робітники, які зазвичай виконуються на різних комп'ютерах та навіть можуть бути не в мережі компанії [9], надають таку можливість для автоматизації.

При створенні Compass розробники намагалися вирішити декілька проблема, а саме прирівняти використання ігрових консолей у автоматизованому тестуванні до рівня комп’ютера, зробити можливим змінити конфігурації через систему керування версії та зробити більш поглиблену інтеграцію збору метриків та скріншотів.

Першу та другу проблему стало можливим вирішити використовуючи Python. Всі задачі можуть бути створені за допомогою коду, а також використання Python дозволило створити додаткову бібліотеку для взаємодії з зовнішніми пристроями через їх API.

Встановлення та видалення програмного забезпечення необхідного для виконання автоматизованого тестування відбувається за допомогою Puppet. Це система керування налаштуванням програмного забезпечення, які націлена на автоматизацію багатьох процесів налаштування інфраструктури компанії. Вона використовує додатки агенти, які працюють у фоні [10], щоб комунікувати між комп’ютерам, який треба налаштувати, та Puppet сервером. Агент запрошує список потрібних налаштувань у сервері та коли отримую перевіряє чи все налаштовано вірно, якщо знаходить не відповідності, то намагається зробити відповідні дії, щоб виправити це. Після перевірки та виправлення, агент відправляє звіт до серверу [10]. Це спрощує процес підготовки комп’ютерів до проведення автоматизованого тестування, оскільки автоматизує цей процес.

Compass має декілька режимів автоматичного повторювання завдання, якщо воно не змогло виконатися з першого разу. Коли ця функція просто включена, Compass спробує виконати завдання до 3 разів. В режимі автоматичного визначення, Compass сам вирішує чи робити повторне виконання чи ні. Також є два режими, які відключають той комп’ютер де був збій виконання. Тільки один з них, а саме режим відпочинку, відключає на деякий проміжок часу, а інший, режим відключення, відключає повністю.

1.6 Аналіз оглянутих рішень

Огляд існуючих на ринку та створених для внутрішнього використання систем керування автоматизованим тестуванням показав, що всі вони дуже залежні від CI/CD засобів, що робить їх не дуже гнучкими при використанні. Деякі з них маюnm достатньо добре зроблену інтеграцію, як з засобами CI/CD так й з іншими системами. Якщо узагальнити процес запуску виконання та керування автоматизованим тестуванням, то можна виділити такі основні шаги:

— передача команди запуску роботи до засобу CI/CD, яка може відбуватися за допомогою самої системи керування автоматизованим тестуванням або з під самого засобу CI/C;

— CI/CD виконує заздалегідь написаний порядок інструкції;

— після виконання всіх інструкції, в тому числі і автоматизованого тестування, результат тестування передається до системи. Це може відбуватися через код, тобто під час самого виконання, а також вже виконані тести, в деяких системах, можуть бути додані через інтерфейс командної строки.

Головною проблемою цього процесу є велика залежність від сторонніх програмних забезпечень. Це зменшує гнучкість в плані вибору ПЗ для розробки, оскільки різні системи керування автоматизованим тестування підтримую різну кількість засобів та рівень інтеграції вони мають різний. Також використання CI/CD засобів як основний та майже завжди єдиний способу старту автоматизованого тестування, окрім звісно ручного запуску, може призвести до ситуації коли обладнання, на якому провадиться виконання автоматизованого тестування, дає збій або не має необхідних елементів для проходження тестування таких, як набору даних, бібліотеки, тощо., що може призвести до витрачання часу розробників та тестувальників для пошуку проблеми, коли проблема буде в обладнанні, а не в додатку, який знаходиться в розробці.

На рис. 1.3 зображено узагальнений бізнес процес керування автоматизованим тестуванням оглянутих систем.

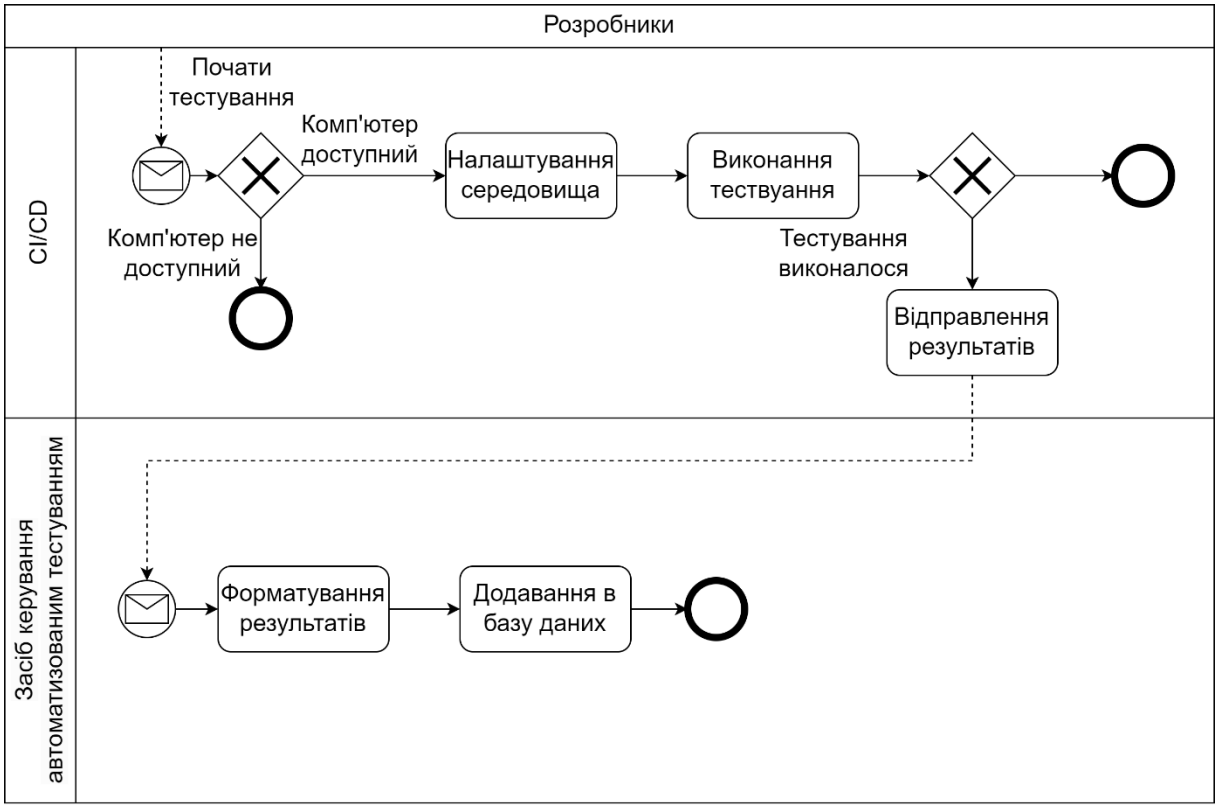


Рис 1.3 Узагальнений бізнес процес керування автоматизованим тестуванням

Також є проблемою те, що ці системи керування автоматизованим тестуванням не мають власних додатків, які б встановлювалися на комп’ютери, де буде проводитися автоматизоване тестування, на кшталт агенту Puppyt [10], це забирає можливість взаємодіяти з тестовим комп’ютером, а інколи це може дуже допомогти, наприклад, перезапуск, перевірка наявності всіх необхідних елементів та інше. Це так само віднімає багато можливостей для взаємодії з зовнішнім обладнанням, які сьогодні дуже часто буває необхідні, через розвиток IoT, розподілених систем, великий різновид систем, де має з’явитися програмне забезпечення, при проведені автоматизованого тестування.

Ці системи зазвичай маю малі можливості для модифікування. В більшості є API для отримання та відправлення даних та це майже єдиний спосіб якось їх модифікувати, але це все одно буде сторонній додаток через, що зробити їх взаємодію з зовнішнім обладнанням стає ще більш складніше.

А використання CI/CD засобів як систем керування автоматизованим тестування є гарною ідеєю, але ,хоч ці засоби націлені на всі етапи CI/CD процесу, вони приділяють не так багато уваги автоматизованому тестуванню. Треба розуміти, що в основну вони приділяють увагу процесу збірки програмного забезпечення, а не його тестування.

Така прив’язка автоматизованого тестування до CI/CD надає свої переваги, але не всі тести можуть бути додані у конвеєр задач [11]. Це змушує розробників шукати методи рішення цих проблем та не завжди вони зручні і швидко реалізовані. Інколи ці рішення становяться настільки складними, що заміна людини, яка це все налаштовувала може призвести до зупинки виконання цього автоматизованого тестування до тих пір покини не придумують новий метод.

1.7 Постановка задач

Системи керування автоматизованим тестуванням поєднують в собі керування ручним тестуванням та керування автоматизованим [1]. Це зміщує пріоритети їх розробників у реалізації функціоналу через, що запуск та виконання автоматизованих тестів майже завжди відходить від системи до CI/CD засобів. Вони допомагають здійснювати автоматизоване тестування, але через їх використання між системою керування та тестами утворюється додатковий рівень. Через цей рівень обмежується взаємодія з тестовими комп’ютерам та системою, що унеможливлює перевірку відповідності середовища виконання тестів до вимог цих тестів.

Існуючі системи в більшості свого мають обмежену взаємодію з зовнішніми приладами, що в сучасному світі є великим недоліком, тому що зараз існує багато приладів під які розробляється програмне забезпечення. Це змушує розробників витрачати час на розробку засобів для тестування замість роботи над проектом.

Не всі автоматизовані тести підходять до використання їх з CI/CD засобами [11]. Наприклад, автоматизоване тестування API, яке мають залежність з

додатковими сервісами або базами даних, які складно імітувати, то їх додавання до конвеєру задач CI/CD неможливе.

Основними вимогами до системи керування автоматизованим тестування, яка розробляється, є:

— відділення системи керування від CI/CD засобів за допомогою системи агентів, які встановлюються на комп’ютери, де буде відбуватися автоматизоване тестування;
— система повинна мати можливість до модифікування – це має спростити процес підлаштовувано системи під потреби проекту;
— мати можливість паралельно запуску автоматизованих тестів.

2 МОДЕЛЬ СИСТЕМИ КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ

2.1 Архітектура системи

Існує безліч архітектурних шаблонів для проектування та побудови систем, які вирішують різні типові завдання, що виникаю при розробки програмного забезпечення дуже часто. Вони визначають високорівневу структуру системи та взаємодію між її елементами.

Існують фундаментальні архітектурні шаблони, які з’являються час від час в різних представлення та видозмінні, але в яких головна ідея залишається однією [12]. Вони надають можливість створювати добре організоване програмне забезпечення, тому що дозволяють представити систему на більш високому рівні абстракції, спрощуючи розробку, розподіляючи функціональність між різними компонентами та забезпечуючи легку розширюваність. Ці шаблони включають в себе моделі, такі як «Модель-Вид-Контролер», «Клієнт-Сервер», «Посередник» та інші, кожен з яких має свої унікальні переваги та застосування в залежності від конкретного завдання. Використання цих архітектурних шаблонів сприяє підвищенню ефективності розробки, покращенню сумісності та можливість до обслуговування програмного забезпечення.

При розробці ПЗ виникає багато проблем та деякі з них відбуваються дуже часто тому їх називаються антишаблонів – це проблеми, які дуже часто виникають при розробці. Великий куля бруду – це антишаблон, який означає відсутність архітектури у програмного забезпечення [12]. Він виникає достатньо часто , коли розробник не ретельно планує або не визначає структуру програми перед початком роботи. Відсутність чіткого плану та визначення архітектурних принципів може призвести до того, що програмне забезпечення стає складним у розумінні, важким у розвитку та підтримці.

Велика куля бруду також може виникнути, коли розробники дозволяють системі розвиватися без контролю, додавати новий функціонал без належного аналізу та проектування. Це призводить до непотрібного накопичення кодової бази, заплутаності та втрати ефективності.

Сьогодні, коли існує багато малих додатків написані за допомогою скриптових мов з невеликим функціоналом та з невеликою кількість взаємодією з базою даних, які не мають архітектури. Це спричиняє появу сервісів, які мають антишаблон великої кулі бруду, оскільки часто буває, що ці невеликі додатки починають рости [12].

Архітектура програмного забезпечення зазвичай поділять на монолітну та розподілену. Це відбувається тому, що ці дві категорії представляють основні підходи до побудови програмного забезпечення, які відрізняються за способом організації та взаємодії між компонентами системи.

Монолітна архітектура – це модель архітектури програмного забезпечення, який є традиційним та поширеним підходом до розробки ПЗ. Увесь функціонал програми, яка використовує монолітну архітектуру, знаходиться в одному єдиному блоці, де всі компоненти, модулі та сервіси з’єднані в одну цілу структуру.

Головною ідеєю цієї архітектури є те, що всі частини програми працюють між собою напряму, без потреби використовувати мережеві виклики. Це робить розгортання та керування системою більш простим, особливо для невеликих проектів, тому що усі компоненти знаходяться на одному комп’ютері в межах однієї програми.

Монолітна архітектура має погану можливість до масштабування , особливо при збільшенні розміру та об'єму проекту. Зазначена обмеженість може виникнути через те, що вся система розглядається як єдиний блок, і всі компоненти пов'язані між собою безпосередньо. Коли кількість користувачів або обсяг оброблюваних даних зростає, моноліт може виявитися неефективним у забезпеченні потрібної продуктивності. Процес масштабування систем, які використовують монолітну архітектуру, потрібує великих зусиль та вимагає зазвичай значних ресурсів. Оскільки вся система в монолітній архітектурі є єдиним блоком, збільшення її

потужності часто передбачає горизонтальне або вертикальне масштабування всієї структури.

Горизонтальне масштабування означає розподіл навантаження між декількома копіями системи, що працюють паралельно [13]. У моноліті це може виявитися складним завданням, оскільки не завжди можна просто так розділити систему на частини, які можна розгорнути окремо.

Вертикальне масштабування полягає у збільшенні потужності окремих компонентів системи, що може вимагати апгрейду апаратного забезпечення, наприклад, збільшення обсягу пам'яті чи потужності процесора. Але це також може досягати свого ліміту через обмеженість фізичних можливостей окремих компонентів.

Розподілена архітектура стало поширеною за останні роки через зростання потреб у розширенні та масштабуванні програмних систем. Функції та компоненти системи, яка використовує цю архітектуру, розподілені між різними серверами чи вузлами, які можуть взаємодіяти через мережу. Цей підхід дозволяє побудувати більш гнучкі, масштабовані та надійні системи, особливо в умовах великого обсягу даних та потреб у високій доступності.

Вона дозволяє розвивати, вдосконалювати та масштабувати компоненти системи незалежно один від одного. Це полегшує розробку, оскільки окремі частини можуть бути розгорнуті, оновлені та масштабовані окремо, без необхідності змінювати весь код програми.

Також, розподілена архітектура дозволяє використовувати різноманітні технології та мови програмування для різних компонентів системи, що дозволяє використовувати найкращі інструменти для конкретних завдань без обмежень однорідності, як у монолітних системах.

Цей підхід дозволяє створювати більш гнучкі та масштабовані системи, що легше розвиваються разом з зростанням обсягу та складності завдань, що стоять перед програмним забезпечення сьогодні.

При побудування систем, які базуються на розподіленій архітектурі, треба пам’ятати про деякі особливості цього. В розподіленій системі, в більшості

випадках, компоненти знаходяться на різних комп’ютерах, а взаємодія між ними відбувається за допомогою мережі, яка має обмеження.

Надійність мережі дуже часто, при розробці систем з розподіленою архітектуро, вважається непохитною, але, на жаль, це не завжди відповідає дійсності [12]. З розвитком мережа стала більш стабільною, але зараз досі можуть виникати різноманітні проблеми, такі як втрати пакетів, затримки передачі даних, або навіть тимчасові перерви зв'язку.

Несправності та втрати даних можуть виникнути через несправність обладнання, недоліки у програмному забезпеченні маршрутизаторів, проблеми зі стійкістю перед атаками з боку зловмисників або навіть через природні катастрофи. Ці фактори можуть призвести до тимчасових втрат зв'язку або навіть до повного відмови мережі.

Розподілені системи, які покладаються на стабільність мережевого зв'язку, повинні враховувати ці потенційні ризики та вживати заходів для забезпечення надійності. Це може включати в себе розробку механізмів автоматичного перепідключення, використання технологій резервного копіювання, оптимізацію алгоритмів обробки помилок та реалізацію моніторингу стану мережі.

На відміну від монолітних систем, у розподілених системах взаємодія між компонентами залежить від мережі, яка зазвичай має затримку. Це означає, що обмін даними та комунікація між різними частинами системи часто піддаються впливу часу, необхідного для передачі інформації через мережу [12], що може стати критичними у випадках, коли система очікує миттєвих відповідей або вимагає синхронної взаємодії між компонентами.

Для зменшення впливу затримок мережі в розподілених системах використовують різні стратегії, такі як кешування даних на вузлах, асинхронні моделі взаємодії, оптимізації алгоритмів передачі даних та використання розподілених кешів. При цьому розробники повинні враховувати можливі затримки та вдосконалювати архітектуру системи для оптимізації продуктивності в умовах розподіленості та мережевих особливостей.

Пропускна здатність мережі не дуже впливає на роботу монолітних систем, але для розподілених це є одним з важливіших характеристик, тому що вона має визначальний вплив на ефективність та продуктивність системи. У розподілених системах, де компоненти розташовані на різних серверах чи вузлах мережі, швидкість передачі даних через мережу може значно впливати на час відповіді та завантаження системи [12].

Висока пропускна здатність дозволяє більше даних передаватися через мережу за одиницю часу, що може бути вирішальним фактором для оптимальної роботи розподіленої системи, особливо при великому обсязі та частій взаємодії між компонентами. Низька пропускна здатність може призводити до затримок у виконанні операцій, перевантаження мережі та погіршення загальної продуктивності.

Для оптимізації роботи розподілених систем необхідно враховувати пропускну здатність мережі та використовувати стратегії, які дозволять мінімізувати вплив мережевої затримки на продуктивність системи. Це може включати в себе використання кешування даних, розподілення навантаження та оптимізацію мережевих запитів для ефективного використання доступної пропускної здатності.

В розподілених системах забезпечення безпека стає тяжким викликом тому, що дані та комунікація між компонентами часто подолають великі відстані та переходять через непередбачувані мережеві шляхи. Ця особливість робить системи більш вразливими до різних видів атак та загроз безпеці.

Однією з головних проблем є ризик витоку конфіденційної інформації через мережевий зв'язок. На відміну від монолітних систем, де дані можуть перебувати в одному локальному середовищі, розподілені системи повинні стежити за захистом даних під час їх передачі між вузлами. Шифрування, безпечні протоколи зв'язку та механізми автентифікації стають важливими елементами для захисту інформації в мережі.

Топологія мережі є зміною та цей факт додає багато проблем для розподілених систем, оскільки утримання стабільної та передбачуваної

комунікації між компонентами стає викликом. Зміни у топології, такі як додавання нових вузлів, розширення мережі або відмова вузлів, можуть призвести до перерв у зв'язку, зміни шляхів передачі даних та взагалі складнішої мережевої динаміки. Це створює проблеми у керуванні маршрутизацією, забезпеченні доступності та ефективності мережі, оскільки розподілені системи повинні бути готовими до адаптації до змін у топології для забезпечення безперебійності роботи.

Зміна топології також може впливати на безпеку, оскільки нові шляхи сполучення можуть стати місцем для потенційних атак або вразливостей. Тому важливо мати механізми, які виявлять зміни у топології, ефективно керувати цими змінами та забезпечити захист від можливих загроз, які можуть виникнути внаслідок цих змін.

Одна з вимог до системи керування автоматизованим тестуванням, яка розроблюється, є те що вона повинна бути розгалуженою для якої існує великий різновид високорівневих клієнт-сервер шаблонів. Ці шаблони можна поділити за рівнями, які вказують на скільки сильно система є розподіленою.

Всі компоненти системи використовуючи однорівневу архітектуру розташовані на одному рівні, без явного поділу на клієнтську та серверну частини. Це означає, що логіка застосунку, обробка даних та інтерфейс користувача розміщені на одному рівні, і не передбачається чіткого розділення між клієнтом і сервером.

Однорівнева архітектура системи може бути використана в простих та менш об'ємних проектах, де немає потреби у розподіленій логіці між клієнтом та сервером. Це може включати невеликі веб-сайти, додатки для одного користувача або прототипи програм.

Одна з ключових переваг однорівневої архітектури полягає в її простоті розробки та реалізації. Оскільки всі компоненти системи розташовані на одному рівні, немає необхідності в складній взаємодії між різними компонентами. Це може значно скоротити час і зусилля, необхідні для розробки та розгортання системи.

Процес тестування однорівневої архітектури є достатньо легким. Оскільки всі компоненти системи розташовані в одному місці, їх можна тестувати одночасно. Це може допомогти виявити помилки на ранніх стадіях розробки, що може значно скоротити час і витрати на усунення помилок.

Однорівнева архітектура менш складна в управлінні. Оскільки немає необхідності в складній взаємодії між різними компонентами, система є більш стійкою до збоїв. Це може значно полегшити управління системою та її підтримку.

Для більших та більш складних систем зазвичай використовують багаторівневі архітектури, такі як клієнт-серверні або трьохрівневі архітектури. Це дозволяє краще розділити функціональність, полегшує масштабування, поліпшує підтримку та забезпечує більшу гнучкість у розвитку та модифікаціях системи.

Системи, які відділяють графічний інтерфейс користувача від бази даних, яка переходить на сторону серверу, а вся бізнес-логіка може бути або на стороні клієнта або на стороні серверу, використовують двухрівневу архітектуру.

Однією з ключових особливостей цієї архітектури є відокремлення представлення даних від їх збереження, що призводить до поліпшеної модульності та більш ефективного управління системою. Графічний інтерфейс, доступний користувачеві, не прямо пов'язаний зі структурою бази даних, дозволяючи змінювати схему даних без суттєвих змін у користувацькому інтерфейсі.

Бізнес-логіка може бути реалізована на стороні клієнта, якщо це необхідно для швидкості реакції на взаємодію користувача без постійного звертання до серверу. З іншого боку, використання серверної бізнес-логіки дозволяє краще контролювати та захищати дані, а також спрощує управління даними та логікою за допомогою централізованого серверу.

Ця архітектура надає розробникам більшу гнучкість у виборі місця розміщення бізнес-логіки відповідно до конкретних потреб та вимог проекту. Це робить двухрівневу архітектуру популярним вибором для розробки сучасних

програмних застосунків, де важливо забезпечити ефективність, безпеку та гнучкість.

Коли в системі додається ще один рівень з основною призначенням, яким є виконувати бізнес-логіку, це може вказувати на перехід до трьохрівневої архітектури. У такій архітектурі функції системи розділяються на три основні рівні: перший рівень - це клієнтський інтерфейс, який взаємодіє з користувачем, другий рівень - це середній рівень, де знаходиться бізнес-логіка, і третій рівень - це рівень доступу до даних або бази даних.

Трьохрівнева архітектура дозволяє ще чіткіше відокремити логіку представлення даних від бізнес-логіки та доступу до даних. Це сприяє покращенню модульності системи, полегшує розподіл завдань між командами розробників та робить систему більш масштабованою та гнучкою.

На першому рівні, клієнтський інтерфейс забезпечує взаємодію з користувачем, на другому - бізнес-логіка обробляє запити та виконує потрібні операції, а на третьому рівні - забезпечується доступ до даних та їх збереження у базі даних.

Ця архітектура використовується у багатьох серйозних та складних програмних проектах, оскільки дозволяє краще організувати та контролювати роботу системи на кожному рівні, що сприяє її стабільності та ефективності. Такий підхід також полегшує розширення та підтримку системи, зменшуючи залежність між компонентами.

Коли в структурі трьохрівневої системи з'являється ще один рівень, її можна характеризувати як n-рівневу систему, де "n" вказує на загальну кількість рівнів в архітектурі. Цей додатковий рівень може взаємодіяти з іншими рівнями і виконувати певні функції в системі.

З'явлення додаткового рівня в архітектурі системи може відбуватися з різних мотивів, але завжди воно впливає на загальну складність та функціональність системи. Кожен новий рівень несе певну функціональність і взаємодіє з іншими складовими системи, створюючи гнучкість та можливості для оптимізації та розвитку.

Однією з ключових переваг додаткового рівня є покращення розширюваності системи. Чим більше рівнів, тим більше можливостей для розширення функціональності без великих змін у вже існуючих модулях. Це сприяє більш гнучкому розвитку системи, коли необхідно додавати нові функції чи модифікувати існуючі.

Окрім цього, додатковий рівень може відігравати ключову роль у забезпеченні безпеки системи. Наприклад, це може бути рівень, який відповідає за захист конфіденційності та цілісності даних, моніторинг потенційних загроз чи навіть ідентифікацію та відхилення несанкціонованого доступу.

Завдяки цьому, система стає більш стійкою до потенційних загроз та проблем з безпекою, що особливо важливо у великих та критичних застосунках. Додатковий рівень може стати важливим компонентом у вдосконаленні безпеки, а також у покращенні загальної функціональності та продуктивності системи.

Для створення системи керування автоматизованим тестування, яка буде мати безпосередній доступ до пристрів на яких проводиться тестування, більше всього підходить n-рівнева архітектура. Оскільки кожен пристрій, а саме комп’ютер, який буде використовуватися для автоматизованого тестування, є важливим компонентом в цій системі, то такий тип архітектури клієнт-сервер дозволить оперувати достатньо великою їх кількістю.

N-рівнева розподілена архітектура, завдяки своїй гнучкості та модульності, може бути реалізована різними підходами, в залежності від конкретних потреб та вимог проекту. Одним з популярних підходів є мікросервісна архітектура.

У мікросервісній архітектурі система розбивається на невеликі та незалежні компоненти - мікросервіси, які виконують конкретні функції. Кожен мікросервіс може бути розгорнутий та масштабований незалежно від інших, що дозволяє ефективно управляти ресурсами та полегшує розвиток та підтримку системи.

Цей підхід дозволяє командам розробників працювати над окремими сервісами, використовуючи різні технології та мови програмування, що сприяє використанню найкращих практик та інструментів для кожної конкретної задачі. Крім того, мікросервісна архітектура робить систему більш масштабованою та

гнучкою, оскільки різні компоненти можуть бути розгорнуті та масштабовані окремо.

Важливим аспектом мікросервісної архітектури є також можливість використовувати контейнеризацію, таку як Docker, для упакування та розгортання мікросервісів. Це спрощує управління залежностями, розгортанням та масштабуванням окремих компонентів системи, що є важливим у сучасних та динамічних розподілених середовищах.

Архітектура на основі просторою (англ, Space-Based Architecture, SBA), є підходом до проектування розподілених систем, який став популярним у сучасній інформаційній технології. Основна ідея полягає в тому, щоб використовувати розподілений простір для зберігання та обробки даних, який може розширюватися горизонтально, щоб легко впоратися зі зростанням обсягу даних та навантаження системи.

В архітектурі на основі простору, дані та функції розподілені по всьому просторі, а не зосереджені в одному центрі. Це дозволяє системі ефективно обробляти великий потік даних, а також пристосовуватися до змін в завданнях та обсязі інформації. Цей підхід особливо корисний у великих та розподілених застосунках, де необхідно забезпечити швидку та масштабовану обробку даних.

Однією з важливих концепцій архітектури на основі простору є ідея «розподіленого простору подій». Це означає, що різні частини системи можуть спостерігати за подіями, які стаються в просторі, та реагувати на них. Це створює гнучку та адаптивну систему, яка може взаємодіяти з різними компонентами, навіть якщо вони розташовані у різних частинах мережі.

Архітектура на основі простору також дозволяє легко розширювати систему шляхом додавання нових вузлів чи областей простору для забезпечення більшої потужності чи витривалості. Такий підхід робить його привабливим в сучасних динамічних інформаційних середовищах, де масштабування та гнучкість є ключовими вимогами.

Ще одним достатньо відомим шаблоном є архітектура на основі подій (англ. Event-Based Architecture, EBA). Цей підхід полягає в тому, щоб сприяти взаємодії

між різними компонентами системи за допомогою обміну подіями. У контексті цієї архітектури, події можуть бути спричинені різними подіями в системі, такими як дії користувача, зміни стану системи або прибуття нових даних.

Однією з ключових переваг архітектури на основі подій є її здатність до асинхронної взаємодії між компонентами. Це дозволяє реалізовувати локальні та глобальні події без блокування роботи інших частин системи. Кожен компонент може бути виробником подій, споживачем чи обидвом, створюючи гнучкі та легко розширювані системи.

Додатково, архітектура на основі подій спрощує розподілену розробку та розгортання системи. Компоненти можуть функціонувати незалежно один від одного, а взаємодія відбувається за допомогою асинхронних подій, що полегшує реалізацію та обслуговування системи в різних розподілених середовищах.

Загалом, архітектура на основі подій є потужним інструментом для розробників, який сприяє створенню гнучких, легко розширюваних та високопродуктивних програмних систем.

Для системи керування автоматизованим тестуванням краще всього підходить архітектура на основі події, оскільки система має багато компонентів, які повинні взаємодіяти між собою за допомогою викликів. У тестувальних процесах важливо мати здатність миттєво реагувати на події, такі як початок виконання тесту, його успішне або невдале завершення, або зміни в конфігурації тестового середовища.

Асинхронність та асинхронні події сприяють швидкій та ефективній обробці інформації між різними компонентами системи. Кожен компонент може бути виробником або споживачем подій, що дозволяє створювати гнучкі та розширювані модульні блоки. Наприклад, виникнення помилки під час виконання тесту може породжувати відповідну подію, яка автоматично виводиться на інтерфейс користувача або генерує повідомлення для інших компонентів системи.

Додатково, архітектура на основі подій полегшує розширення функціональності системи. Нові функції чи модулі можуть бути впроваджені, додаючи або обробляючи нові події, без необхідності редагувати вже існуючий

код. Це зменшує ризик виникнення помилок та спрощує роботу розробників під час вдосконалення або модифікації системи керування автоматизованим тестуванням.

На рис. 2.1 зображена розроблена високорівнева архітектура системи керування автоматизованим тестуванням. На ньому також зображені 3 основних сховища, які спрямовані на збереження даних о користувачеві, о тестових комп’ютерах, о проведених тестах або о тих, які виконуються, та файлове сховище.

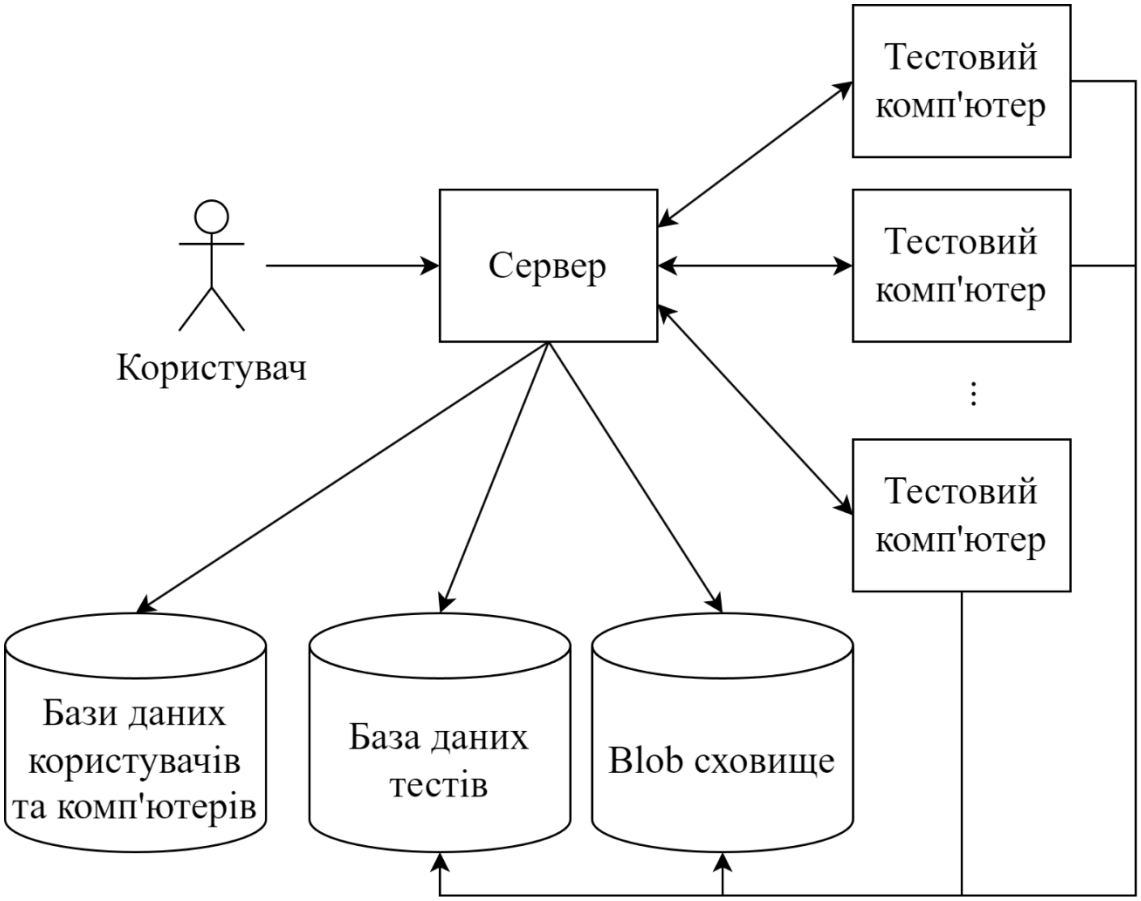


Рис. 2.1 Розроблена архітектура системи керування автоматизованим тестуванням

Сервер та агенти, які встановлюються на тестові комп’ютери, є компонентами системи, але вони також мають свої компоненти та їх архітектура не менш важлива ніж архітектура цілої системи. Вони повинні мати можливість, як до масштабування, так й для бистрого розгортання для маленьких проектах.

Деякі існуючі монолітні архітектурні шаблони мають відповідні характеристики та можливості.

Один з таких шаблонів – це конвеєрна архітектура , яка є потужним та ефективним методом організації процесів в системі. Конвеєрна архітектура базується на принципі поділу складної задачі на невеликі і прості кроки, які виконуються послідовно на кожному етапі конвеєра. Кожен етап може бути представлений окремим компонентом або модулем, що відповідає за певну функцію чи обробку даних.

Конвеєрна архітектура дозволяє розділити обробку даних на послідовні етапи, які виконуються послідовно або паралельно. Кожен етап конвеєра відповідає за виконання конкретної операції або обробку певного аспекту даних. Це структуроване розподілення виконання завдань дозволяє покращити ефективність системи, особливо в ситуаціях, коли потрібно обробляти великий потік даних.

Застосування конвеєрної архітектури особливо ефективне у випадках, коли задачу можна поділити на незалежні або слабо залежні етапи, які можна виконувати паралельно. Це дозволяє досягти великої швидкості обробки та ефективно використовувати ресурси системи.

Окрім того, конвеєрна архітектура сприяє легкості супроводу та розширення системи. Кожен етап може бути оптимізований окремо, а нові етапи можуть бути додані або інтегровані без значних змін в існуючій структурі конвеєра. Це робить конвеєрну архітектуру привабливим вибором для проектів, які піддаються постійним змінам або розвитку.

Також з підходящих шаблонів є мікроядерна архітектура, яка представляє сучасний підхід до побудови систем, де основна частина функціональності системи вбудована в ядро, а додаткові сервіси виокремлюються в окремі модулі, що працюють поза ядром. Цей підхід дозволяє створювати більш гнучкі та легко розширювані системи, спрощуючи управління та супровід.

Мікроядерна архітектура базується на ідеї розділення ядра системи на мінімальний набір функцій, які називаються мікроядром, та зовнішні модулі, які

надають конкретні сервіси. Мікроядро відповідає за базову функціональність, таку як керування пам'яттю та планування задач, в той час як конкретні функції, такі як введення-виведення чи мережеві сервіси, можуть бути реалізовані як зовнішні модулі.

Однією з ключових переваг мікроядерної архітектури є її гнучкість у розширенні та модифікаціях. Додавання нових сервісів або зміни існуючих можуть відбуватися без перезапуску всієї системи. Крім того, цей підхід полегшує відлагодження та тестування, оскільки різні частини системи можуть бути розглядатися як незалежні модулі.

Мікроядерна архітектура особливо підходить для сучасних систем, де вимагається висока гнучкість, швидкість розгортання та здатність до адаптації до змін у вимогах.

На рис. 2.2 зображена архітектура сервера системи керування автоматизованим тестуванням, яка базується на мікроядерної архітектурі, оскільки вона відповідає всім вимогам до цього компонента системи. Вона має 3 основних модулі, а саме:

— IPC (Inter-Process Communication, укр. взаємодія між процесами), який відповідає за комунікацію з іншими компонентами всієї системи, тобто в основному з агентами.
— Адаптер баз даних допомагає взаємодіяти з різними базами даних, тому що в системі використовуються різні типи даних, через що виникає потреба використовувати різні типи баз даних.
— Планер є головним модулям у всій системі, оскільки саме він відповідальний за розподіл між тестовими комп’ютерами завдань, перевірку їх відповідності до вимогам та в загальному керування ними.

Агент має схожу архітектура та основні модулі, але замість планера, він має виконувач, який відповідає за взаємодію з комп’ютером, де знаходиться клієнт, та з пристроями, які підключені до цього комп’ютера.

Плагіни – це необов’язкові модулі, які можуть бути інтегровані в систему з метою розширення її функціоналу чи додання специфічних можливостей. Їхня

основна особливість полягає в тому, що система може працювати як без них, так і з ними, не втрачаючи основного функціоналу. Такий підхід дозволяє створювати гнучкі та розширювані системи, які можуть адаптуватися до різних потреб та вимог користувачів.

Використання плагінів допомагає системі зберегти всі переваги монолітної архітектури, але також зберегти можливість до масштабування. Що робить систему легкою до розгортання для невеликих проектів, а також залишає її так само гнучкою та з можливість до розширення.

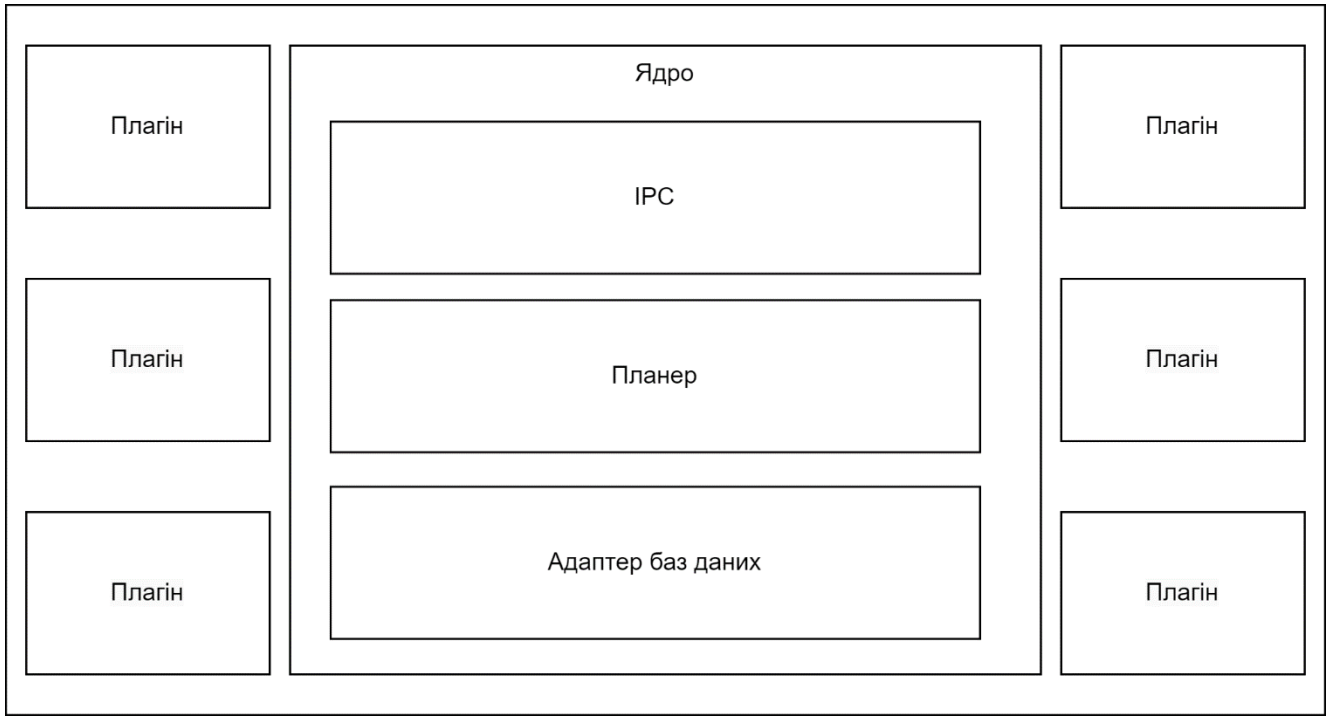


Рис 2.2 Розроблена архітектура сервера системи керування автоматизованим тестуванням

2.2 Реляційні СУБД

В системі керування автоматизованим тестуванням є ряд даних, які можуть бути легко представлені у вигляді таблиць, а саме інформація о користувачах, о комп’ютерах, які задіяні або можуть бути використані у автоматизованому тестуванні, та пристрої, що забезпечить легку роботи з ними. Для збереження даних у вигляді таблиць використовують реляційні системи управляння базами

даних (РСУБД) – це тип систем, які використовують реляційну модель даних для організації та зберігання інформації. Реляційна модель даних базується на табличній структурі, де дані представлені у вигляді таблиць, а кожен запис у таблиці відповідає рядку, атрибути – це колонки.

Представлення даних в РСУБД у вигляді таблиць. Таблиця складається з рядків і стовпців, де кожен рядок відповідає конкретному запису або кортежу, а кожен стовпчик - певному атрибуту чи полю [14]. Цей формат організації даних дозволяє легко відображати структуру і взаємозв'язки між різними елементами інформації.

Ключовим аспектом реляційних таблиць є унікальний ідентифікатор для кожного запису, який відомий як ключ. Цей ключ може бути одним або складатися з декількох полів, що гарантує унікальність кожного рядка в таблиці. Крім того, таблиці можуть мати відносини одна з одною, що дозволяє встановлювати зв'язки між різними даними в базі.

У реляційних таблицях стовпці мають типи даних, які визначають характеристики даних, які можуть бути збережені у відповідному полі. Це можуть бути числа, рядки, дата і час, булеві значення та інші типи, які використовуються для відображення різних видів інформації.

Один з ключових аспектів таблиць у РСУБД - це їхня нормалізація. Цей процес дозволяє розбивати дані на окремі таблиці, що допомагає уникнути зберігання дублюючої інформації та забезпечує більш ефективне управління базою даних. Нормалізація допомагає знизити ризик аномалій даних та забезпечує більшу консистентність та точність інформації у базі даних.

Використання, у РСУБД, мови структурованих запитів (англ. Structured query language, SQL) дозволяє ефективно взаємодіяти з базою даних та виконувати різноманітні операції з отримання, оновлення, вставки та видалення даних. SQL є стандартом для роботи з реляційними базами даних і надає зручний та стандартизований інтерфейс для взаємодії з даними у базі.

Однією з основних можливостей SQL є можливість виразно визначати структуру даних, використовуючи команди для створення таблиць, визначення

ключів, обмежень та інших аспектів бази даних. Створення, модифікація та управління базовою структурою даних стає легким завданням завдяки декларативному підходу SQL.

Запити SQL можуть бути використані для вибірки даних, фільтрації результатів за певними умовами, сортування, з'єднання декількох таблиць та виконання агрегатних функцій для отримання зведеної інформації. Це надає розробникам широкі можливості для отримання необхідної інформації з бази даних у потрібному форматі.

SQL також включає механізми для управління безпекою даних, встановлення прав доступу до таблиць та забезпечення конфіденційності інформації. Використання SQL спрощує завдання роботи з даними та дозволяє ефективно виконувати різноманітні операції, необхідні для роботи з реляційними базами даних.

Одною з найважливіших особливостей РСУБД є їхній механізм забезпечення цілісності даних. Ця особливість гарантує, що дані в базі залишаються надійними, консистентними та коректними протягом усього їхнього життєвого циклу. Забезпечення цілісності даних полягає у збереженні та виконанні правил, які гарантують правильність та унікальність даних.

У реляційних базах даних, цілісність даних зазвичай здійснюється за допомогою обмежень (constraints). Серед типових обмежень можна виділити обмеження унікальності (UNIQUE), яке гарантує унікальність значень у вказаному стовпці або комбінації стовпців, і обмеження первинного ключа (PRIMARY KEY), яке визначає унікальний ідентифікатор для кожного запису в таблиці [14].

Додатково, використання обмежень цілісності даних дозволяє встановлювати взаємозв'язки між таблицями, щоб забезпечити консистентність при зміні даних. Обмеження зовнішнього ключа (FOREIGN KEY) вказує на зв'язок між таблицями, щоб забезпечити, що значення в зовнішньому ключі завжди посилаються на існуючий запис в батьківській таблиці.

Завдяки цим механізмам забезпечення цілісності даних, розробники можуть бути впевнені в стабільності та надійності бази даних, а користувачі можуть

очікувати консистентних та точних результатів під час взаємодії з інформацією у системі.

Механізмами транзакцій у РСУБД дозволяє виконувати набір операцій як атомарну інструкцію , що означає, що вони виконуються цілком або жоден з їхніх етапів не виконується взагалі. Це гарантує консистентність бази даних та виключає можливість втрати даних або порушення цілісності внаслідок помилок чи відмін.

Основними властивостями транзакцій є атомарність, що означає, що операції транзакції виконуються або повністю, або не виконуються взагалі; консистентність, яка гарантує, що транзакція переводить базу даних з одного стабільного стану в інший стабільний стан; ізоляція, що дозволяє виконання кількох транзакцій паралельно, не порушаючи їхньої консистентності; та доведеність (англ. durability), що гарантує, що результати виконаних транзакцій будуть надійно збережені навіть у випадку відмови системи.

Використання транзакцій у РСУБД забезпечує надійність в разі збоїв або втрати енергії, оскільки вони дозволяють системі повертатися до консистентного стану після відновлення. Крім того, цей механізм дозволяє виконувати складні операції та зміни в базі даних, зберігаючи при цьому її цілісність та стабільність.

РСУБД мають ряд механізмів для покращення продуктивності системи, особливо при операціях з великим обсягом даних. Одним з ключових інструментів є індексація, яка дозволяє швидко знаходити та отримувати доступ до конкретних даних у великих таблицях. Індекси створюються для певних стовпців таблиць та дозволяють РСУБД швидше виконувати пошук, сортування та фільтрацію даних, зменшуючи час виконання запитів.

Оптимізація запитів є ще одним важливим аспектом для підвищення продуктивності. РСУБД використовують оптимізатори запитів, які аналізують структуру запитів і вибирають оптимальний шлях для їх виконання. Це може включати вибір правильного використання індексів, об'єднання або розбиття запитів на менші частини для оптимізації, а також вибір оптимального плану виконання запиту з урахуванням різних параметрів та обмежень.

Ці два механізми разом із рядом інших оптимізаційних технік, таких як кешування, використання пакетних операцій та підтримка паралельних операцій, сприяють покращенню продуктивності РСУБД при обробці великих обсягів даних. Це дозволяє системі ефективно виконувати складні операції навіть у випадку значного обсягу інформації, забезпечуючи швидкість та ефективність в роботі з базою даних.

Надійність та здатність до відновлення у реляційних системах управління базами даних досягається за допомогою

різноманітних технологій та підходів, спрямованих на забезпечення безперебійної роботи та відновлення нормального функціонування в разі можливих збоїв чи втрат даних.

Однією з основних стратегій є резервне копіювання та відновлення даних. Регулярне створення резервних копій баз даних дозволяє відновлювати інформацію до певного стану в разі випадкового видалення, пошкодження чи інших проблем.

Ще однією важливою складовою є ведення журналу транзакцій. Журнали транзакцій зберігають історію всіх змін у базі даних, що дозволяє відтворити стан системи до певного моменту в часі.

Механізми точок відновлення визначають конкретні моменти, до яких можна відновити базу даних. Це дозволяє вибирати оптимальний час для відновлення, забезпечуючи ефективний контроль над процесом.

Додатково, засоби контролю цілісності даних допомагають уникати пошкоджень чи втрати даних. Вони перевіряють, чи відповідають дані певним стандартам та визначеним правилам.

Кластеризація та реплікація використовуються для створення дублюючих систем та резервних копій даних. Це підвищує доступність та надійність системи, дозволяючи швидше відновлення в разі збоїв.

Загалом, використання цих технологій узгоджено із засобами моніторингу та автоматизації, що дозволяє вчасно виявляти проблеми та вживати заходів для їх

вирішення, забезпечуючи стабільність та надійність реляційних систем управління базами даних.

Одним з недоліків реляційних систем управління базами даних є складність роботи з великим обсягом даних. Хоча реляційні бази даних мають великий потенціал у забезпеченні структурованості, цілісності та консистентності даних, при обробці великого обсягу інформації можуть виникати складнощі.

Однією з основних проблем може бути уповільнення швидкодії при операціях з великими обсягами даних. При великих таблицях чи складних запитах РСУБД можуть потребувати багато ресурсів для обробки запитів, що може призводити до зниження продуктивності.

Також, збереження та оптимізація великого обсягу даних вимагає ретельного планування структури бази даних та індексації. Іноді зміна структури бази даних або виконання складних операцій над великими обсягами даних може бути затратно по ресурсам та часу.

Розгортання розподілених РСУБД може стати великим випробуванням через декілька причин. По-перше, це складність налаштування та управління конфігураціями. Розподілені системи вимагають ретельного планування та конфігурації для забезпечення надійності та ефективності.

Крім того, забезпечення синхронізації та консистентності даних в розподілених середовищах також може бути складною задачею. Велика кількість вузлів, які працюють разом, потребує механізмів реплікації та узгодженості, щоб уникнути конфліктів даних і зберегти цілісність інформації.

Також, важливо враховувати проблеми мережевої недоступності. Розподілені РСУБД можуть стикатися з викликами, пов'язаними зі збоєм мережі, що може призвести до проблем доступності даних та синхронізації між вузлами.

Вирішення цих проблем вимагає детального проектування, налаштування та управління системою. Застосування оптимальних стратегій реплікації, балансування навантаження та моніторингу є ключовими аспектами для успішного розгортання розподілених РСУБД.

Для системи керування автоматизованим тестуванням було обрано реляційну систему управління базами даних PostgreSQL, оскільки вона має велику кількість переваг, які роблять її однозначним вибором.

Відкритий код PostgreSQL та активна спільнота розробників є ключовими аспектами, які роблять цю реляційну систему управління базами даних (РСУБД) особливо привабливою для великої кількості проектів, включаючи системи керування автоматизованим тестуванням.

Однією з переваг відкритого коду є можливість вільно використовувати, модифікувати та поширювати програмне забезпечення згідно з власними потребами. Всі заінтересовані сторони можуть зробити свій внесок у розвиток та покращення продукту, роблячи його більш гнучким та пристосованим до різноманітних вимог.

Спільнота розробників PostgreSQL є динамічною та різноманітною, що дозволяє ефективно вирішувати проблеми, вносити нові ідеї та розробляти додаткові можливості. Дискусії, форуми та регулярні оновлення створюють відкрите середовище для обміну знаннями та кращих практик, сприяючи постійному розвитку PostgreSQL.

Важливо відзначити, що велика спільнота також забезпечує підтримку. Запитання стосовно роботи системи, виявлення та виправлення помилок, а також регулярні оновлення роблять PostgreSQL надійним та сучасним рішенням для великої кількості проектів, у тому числі і для систем керування автоматизованим тестуванням.

PostgreSQL володіє розширеним набором інструментів, які допомагають адміністраторам та розробникам ефективно управляти та оптимізувати бази даних [15]. Один із ключових інструментів - pgAdmin, графічний інтерфейс для адміністрування PostgreSQL, який надає зручні засоби для створення, редагування та видалення об'єктів бази даних.

Механізми моніторингу PostgreSQL дозволяють слідкувати за продуктивністю та виявляти можливі проблеми у реальному часі [15]. Зокрема,

власний інструмент управління моніторингом - pg_stat_statements, надає детальну статистику щодо виконання запитів, що є корисним для аналізу та оптимізації.

Щодо забезпечення цілісності даних, PostgreSQL підтримує різні механізми, включаючи унікальність, зовнішні ключі, перевірки та цілісність транзакції. Ці засоби гарантують, що дані в базі зберігаються вірно та відповідають визначеним вимогам.

Оптимізація запитів - ще один важливий аспект PostgreSQL. Засоби такі, як план виконання запитів та індексація, дозволяють підвищити продуктивність системи шляхом швидшого виконання запитів та ефективного використання ресурсів бази даних.

Узагальнюючи, PostgreSQL надає багатий набір інструментів та механізмів, що роблять його потужним та гнучким рішенням для розробників та адміністраторів баз даних у системах керування автоматизованим тестуванням.

На рисунку 2.3 зображене ER-діаграма баз даних для розробляємої системи.

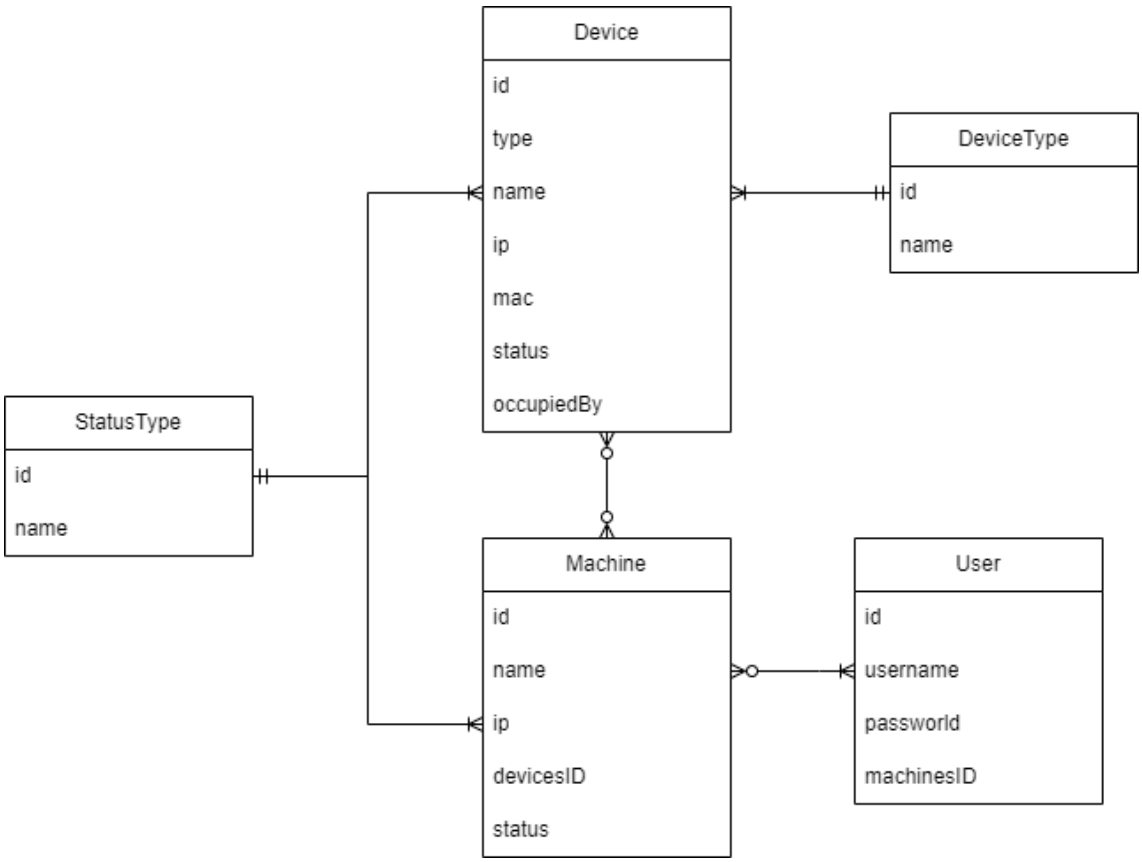


Рис 2.3 ER-діаграма баз даних

2.3 Нереляційні СУБД

Одними з ключових даних системи керування автоматизованим тестуванням є тести, які краще всього представляти у вигляді структури, а щоб зберігати структури треба використовувати нереляційну систему управління базами даних.

Нереляційні СУБД (NoSQL СУБД) представляють собою клас баз даних, які відрізняються від традиційних реляційних систем за своєю структурою та принципами організації даних [16]. Вони з'явилися як відповідь на зростання обсягів та різноманітності даних, які не завжди легко моделювати у вигляді традиційних таблиць та зв'язків.

Головною особливістю нереляційні СУБД є гнучка схема , що робить ці системи особливо ефективними для сучасних вимог до зберігання та обробки даних. Відмінність полягає в тому, що у NoSQL базах даних відсутня строга вимога до фіксованої схеми перед часом введення даних.

У традиційних реляційних базах даних структура даних повинна бути описана заздалегідь, і всі записи повинні відповідати цій строго визначеній схемі. У нереляційних базах, навпаки, дані можуть бути додані без необхідності заздалегідь визначати їхню точну структуру. Це особливо корисно в сучасних умовах, коли типи та формати даних можуть змінюватися швидко та динамічно.

Гнучка схема дає розробникам та адміністраторам більшу вільність та швидкість у внесенні змін до даних та їхньої структури. Це особливо важливо у випадках, коли вимоги до додатків можуть змінюватися або коли необхідно працювати з різноманітними типами даних, такими як текст, графіка, аудіо або великі об'єми даних [16]. Такий підхід робить нереляційні бази даних дуже привабливими для сучасних динамічних проектів.

Існує велика кількість нереляційних системи управління базами даних та відрізняються вони моделями даних, які використовуються для зберігання та організації інформації. Ці системи не обмежені стандартними реляційними моделями, що використовують таблиці, рядки та стовпці. Вони пропонують

різноманітні моделі, що відображають різні потреби і структури даних в сучасному програмуванні.

Деякі нереляційні бази даних використовують ключ-значення для зберігання даних, де кожен запис асоціюється з унікальним ключем. Інші використовують документи, де дані зберігаються у вигляді структурних документів, таких як JSON або BSON, що дозволяють зберігати більш складні структури [16]. Графові бази даних орієнтовані на роботу зі складними взаємозв'язками між даними та ефективною обробкою великих графів.

Інші моделі даних включають зберігання колонками, де дані зберігаються у вигляді колонок, що спрямовані на швидкий доступ до конкретних елементів. Кожна з цих моделей має свої переваги та відповідає певним вимогам, що дозволяє розробникам обирати той тип бази даних, який найкраще відповідає потребам їхнього проекту.

NoSQL СУБД дозволяють легко збільшувати потужність за рахунок горизонтального збільшення, що стає важливою перевагою у сучасних додатках, особливо там, де збільшення обсягів даних може бути значним та несподіваним. Горизонтальне масштабування означає збільшення продуктивності чи обсягів обробки даних шляхом додавання нових серверів або вузлів у схему бази даних.

У NoSQL базах даних це досягається шляхом розподілення даних між кількома серверами. Кожен сервер може бути відповідальним лише за певну частину даних, що дозволяє системі легко масштабуватися. При потребі у збільшенні обсягів обробки даних чи високої доступності, можна просто додавати нові сервери до кластера без значних перерв у роботі системи.

Це гнучке масштабування є особливо корисним у випадках, коли додаток або система мають швидко зростаючу базу користувачів, або коли потрібно обробляти великі об'єми даних. Використання NoSQL бази даних дає можливість легко реагувати на зростання обсягів і обробку даних, що дозволяє підтримувати високий рівень ефективності та доступності системи навіть при збільшенні навантаження.

Деякі системи NoSQL мають вбудовану відмовостійкість та можливість відновлення, що дозволяє їм ефективно працювати в умовах високого навантаження та уникнути втрати даних в разі відмови одного чи кількох серверів. Ця властивість особливо важлива для сучасних додатків, які мають велику кількість користувачів та обробляють значні обсяги даних.

Вбудована відмовостійкість означає, що система може продовжувати свою роботу, навіть якщо один чи декілька компонентів виявляться недоступними або відмовлять. Це досягається за допомогою розподіленої архітектури, де дані реплікуються або розподіляються між кількома серверами. У випадку відмови одного сервера, інші можуть продовжити надавати обслуговування, запобігаючи при цьому втраті даних чи перервам у роботі системи.

Ця властивість забезпечує високу доступність системи та забезпечує надійність, що є важливими аспектами для сучасних додатків з великою кількістю користувачів та вимогами до швидкодії та надійності.

NoSQL СУБД мають гарну підтримку розподілених систем , що робить їх ефективними в умовах розподіленого обчислення та обробки великих обсягів даних. Однією з основних особливостей NoSQL баз даних є їх здатність працювати на розподілених серверах та ефективно масштабуватися в горизонтальному напрямку.

Розподілена архітектура дозволяє розміщувати дані на різних серверах чи вузлах мережі, що дозволяє підтримувати велику кількість одночасних запитів та розділяти обчислювальне навантаження. Це особливо важливо для додатків, які потребують високої швидкодії та відмовостійкості.

Деякі NoSQL бази даних використовують різні методи реплікації та шарування даних, щоб забезпечити розподіленість. Вони також надають інструменти для автоматичного розподілу даних, балансування навантаження та управління розподіленими ресурсами, що робить їх ефективними для використання в сучасних розподілених системах та хмарних середовищах.

Нереляційні системи управління базами даних можуть пристосування до змінних потреб та обсягів даних, не втрачаючи продуктивності або доступності.

Однією з головних переваг NoSQL є їх спроможність ефективно опрацьовувати дані в режимі реального часу та пристосовуватися до великих обсягів інформації, які можуть раптово зростати.

Ця гнучкість до змін реалізується за рахунок різних моделей даних в NoSQL, таких як ключ-значення, стовпчасті, документів та графові бази даних. Кожен тип бази даних спроектований з урахуванням конкретних вимог, і це дозволяє їм легко пристосовуватися до нових вимог або змінюючихся умов, не вимагаючи значних змін в архітектурі.

Використання NoSQL може бути особливо корисним у ситуаціях, де потрібно працювати з великими обсягами неструктурованих даних або де потрібна швидка реакція на зміни в обсягах даних. Це робить NoSQL популярним вибором для розробки додатків, які оптимально використовують ресурси та можуть ефективно масштабуватися.

Нереляційні системи управління базами даним визначаються відсутністю єдиного стандарту, такого як SQL у реляційних базах даних. Це призводить до великого різновиду баз даних, кожна з яких може використовувати власні мови запитів і формати даних. Хоча існують спроби стандартизації в деяких частинах NoSQL, таких як формати обміну даними чи деякі мови запитів, загальна стандартизація залишається відсутньою.

Ця відсутність стандартів може впливати на процес переносу даних між різними системами. Ускладнення процесу переносу може виникати через різні внутрішні структури даних, мови запитів та підходи до зберігання інформації. Для забезпечення ефективного обміну даними між різними NoSQL системами, розробники повинні вдосконалювати власні рішення або використовувати інструменти, які дозволяють адаптувати дані до вимог кожної конкретної системи.

Незважаючи на відсутність загального стандарту, деякі галузі об'єднують зусилля для створення рекомендацій чи загальних практик у роботі з NoSQL базами даних. Однак розробники і адміністратори повинні бути готові до вирішення індивідуальних викликів, які виникають під час взаємодії з різними NoSQL системами.

У порівнянні з реляційними СУБД, нереляційні часто мають ускладнений процеси аналізу даних та звітності. Це часто пов'язано з тим, що вони використовують неструктуровані або поліструктуровані дані, які не мають жорсткої схеми, що регулює взаємозв'язки між ними.

У нереляційних базах даних може виникати складність у виконанні складних аналітичних запитів через відсутність потужних механізмів, які доступні у традиційних реляційних базах даних. Наприклад, деякі системи NoSQL не мають повноцінної мови запитів, або вона обмежена функціональністю, що ускладнює складні операції звітності та аналізу.

Також, в нереляційних базах даних може виникати виклик у плануванні та оптимізації запитів через особливості структури даних. Оскільки вони зазвичай спрямовані на швидкий доступ до великого обсягу даних, аналітика та звітність можуть потребувати специфічних підходів для оптимізації запитів та витягування необхідної інформації.

Це не означає, що нереляційні бази даних не можуть виконувати аналітичні операції або створювати звіти, але часто це потребує особливого підходу та додаткових зусиль для організації та оптимізації процесів аналізу.

Використання MongoDB для зберігання тестових даних у системі керування автоматизованим тестуванням має численні переваги. MongoDB, яка є документо-орієнтованою базою даних, працює із схемами документів у форматі JSON-подібних об'єктів, що відомих як "документи". Це надає гнучкість у зберіганні тестових даних різної структури та формату, що особливо важливо у сфері автоматизованого тестування, де дані можуть бути різноманітними та змінюватися з часом.

Однією з ключових переваг MongoDB є відсутність жорсткої схеми, що дозволяє зберігати тестові дані без необхідності заздалегідь визначати їх структуру. Це важливо, оскільки тестові дані можуть мати різні атрибути та формати, які можуть змінюватися протягом часу.

Процес розгортання MongoDB відзначається простотою та легкістю використання, оскільки ця NoSQL база даних розроблена з урахуванням

принципів гнучкості. Її архітектура дозволяє легко масштабувати та обробляти великі обсяги даних, що є важливим аспектом для систем керування автоматизованим тестуванням.

Наявність гнучких конфігураційних опцій у MongoDB спрощує налаштування параметрів бази даних відповідно до вимог конкретного проекту та ресурсів сервера. Крім того, вона підтримує встановлення кластерів, що полегшує розгортання у розподілених середовищах.

Важливим елементом є графічний інтерфейс, такий як MongoDB Compass, який спрощує процес адміністрування, моніторингу та відладки. Широка документація та активна спільнота роблять підтримку та розвиток MongoDB більш доступними для користувачів.

Наявність систем резервного копіювання та відновлення в MongoDB дозволяє забезпечити стабільність та доступність даних. Регулярні оновлення та покращення від розробників MongoDB сприяють надійності та ефективності бази даних, що є важливим для стабільної роботи в системах керування автоматизованим тестуванням.

2.4 Технологія віддаленого виклику процедури

Система керування автоматизованим тестуванням використовує розподілену архітектуру, де різні компоненти знаходяться на різних комп’ютерах, та реалізація взаємодії між процесами (IPC) може бути зроблена за допомогою різних технології такі, як REST, RPC, WebSocket [18]. Але запити між компонентами системи керування автоматизованим тестуванням дуже схожі на виклик методів, тому найкращою технологією для взаємодії між компонентами системи є RPC.

Технологія віддаленого виклику процедури (англ. Remote Procedure Call, RPC) націлена на забезпечення взаємодії між різними компонентами системи, які можуть знаходитися на різних вузлах мережі. Використання технології віддаленого виклику процедури дозволяє програмам викликати функції чи

процедури, що виконуються на віддаленому сервері, якщо це необхідно для виконання певних завдань.

RPC спрощує розробку розподілених систем, надаючи абстракцію віддалених викликів, яка дозволяє розробникам працювати з віддаленими ресурсами так, як вони працювали б з локальними. Це дозволяє реалізовувати взаємодію між компонентами системи, як якісь локальні виклики процедур, приховуючи деталі роботи мережі та віддаленого виклику.

Процес реалізація технології віддаленого виклику процедури має багато викликів, що сповільнюють його. Однією з основних проблем може бути різниця в мовах програмування між сервером та клієнтом. Кожна мова має свої власні особливості та нюанси, що може впливати на передачу даних та виклик процедур. Розробники повинні враховувати ці різниці та забезпечувати відповідні переклади чи адаптації для правильної інтеграції між мовами.

Другим важливим аспектом є обмеженість віддалених викликів, оскільки сервер та клієнт можуть працювати на різних комп'ютерах та не мати спільної пам'яті. Це вимагає уважної роботи з передачею даних та зберіганням стану між викликами процедур.

Тому використання RPC призводить до поглиблення у питаннях серіалізації та десеріалізації даних. Серіалізація - це процес перетворення об'єктів або даних у формат, придатний для передачі через мережу чи зберігання. Десеріалізація, навпаки, відновлює структуру даних з отриманого формату.

Під час використання RPC важливо, щоб дані передавалися між віддаленими компонентами у форматі, який може бути зрозумілий та вірно оброблений обома сторонами - сервером і клієнтом. Це вимагає правильної серіалізації даних на одному кінці зв'язку, їх передачі через мережу та подальшої десеріалізації на іншому кінці.

Серіалізація та десеріалізація грають ключову роль у забезпеченні правильної передачі даних через мережу, оскільки дозволяють перетворювати складні структури даних у формат, який може бути переданий через мережу, збережений на диску або переданий через віддалене з'єднання, і потім

відновлювати їх у вихідний стан. Це дозволяє різним компонентам системи зрозуміти одне одного та ефективно обмінюватися даними незважаючи на їх розташування та технічні особливості.

Велика кількість RPC фреймворків надає свої механізми серелізації та десерелізації, а деякі навіть обумовлюють використання своєї мови опису інтерфейсів (IDL) [19], що дозволяє серелізувати дані у бінарний вид. Це допомагає забезпечити єдність та стандартизацію в описі віддалених інтерфейсів, спрощуючи взаємодію між різними компонентами системи. Мова опису інтерфейсів (IDL) визначає структури даних та інтерфейси, які використовуються для взаємодії між віддаленими компонентами, і дозволяє автоматично генерувати код для серіалізації та десеріалізації даних.

Використання власних мов опису інтерфейсів фреймворків сприяє створенню стандартизованих контрактів для віддаленої взаємодії, що допомагає уникнути невідповідностей у форматах даних та об'єктних структурах. Це особливо важливо у великих системах, де різні компоненти можуть бути розроблені різними командами та мовами програмування.

Бінарний формат, отриманий в результаті серіалізації з використанням IDL, часто є більш ефективним для передачі через мережу, оскільки він може бути меншим за розміром та економити пропускну здатність. Це стає особливо важливим у великих розподілених системах або в умовах обмежених ресурсів.

Отже, використання власних мов опису інтерфейсів та бінарних форматів серіалізації допомагає забезпечити ефективну та стандартизовану взаємодію між компонентами у розподіленій системі, роблячи процес віддаленого виклику процедур більш надійним та ефективним.

Також використання RPC дає змогу обирати між мережевими протоколами. Ця можливість вибору мережевих протоколів робить технологію RPC дуже гнучкою і адаптованою до різноманітних умов та вимог систем. В залежності від конкретних потреб і характеристик системи ви можете обирати оптимальний мережевий протокол для забезпечення найкращої продуктивності, надійності та безпеки.

Наприклад, якщо важливо мати ефективний обмін даними у великій розподіленій системі, можна обрати TCP/IP як надійний і забезпечений протокол. З іншого боку, для завдань, де важлива мінімізація затримок та оптимізація швидкості передачі даних, можна вибрати UDP або інші протоколи з меншою накладною

Додатково, можливість обирати між мережевими протоколами дозволяє легко інтегрувати системи, які використовують різні технології та архітектури мережі. Наприклад, якщо ви взаємодієте з системами, які використовують RESTful API, HTTP або WebSocket, то ви можете зручно налаштовувати ваші RPC виклики для використання відповідних протоколів і легко інтегрувати їх у вашу розподілену систему.

Це стає особливо важливим у сучасних розподілених середовищах, де різноманітність мережевих архітектур і технологій є нормою. Використання RPC і можливість вибору мережевих протоколів надає розробникам інструмент для ефективного взаємодії та інтеграції між різними складовими системи.

Інколи RPC фреймворки можуть створювати залежність від певної технології, наприклад, Java RMI змушує всі компоненти системи, які взаємодіють між собою за допомого цього фреймворку, використовувати Java [18].

Особливість технології віддаленого виклику процедури, після її реалізації, це те що додавання нових функції, які вона може передавати або приймати, стає дуже простою справою. Це також може призвести до переповнення системи зайвою функціональністю, яка може стати непрактичною. Однак, при правильному підході до дизайну системи, ця особливість може бути використана для створення гнучких та розширюваних архітектур.

Додавання нових функцій до системи віддаленого виклику процедур зазвичай вимагає оновлення мови опису інтерфейсів (IDL) та генерації коду для нових інтерфейсів. Сучасні RPC фреймворки надають інструменти для автоматизації цього процесу, дозволяючи швидко і легко додавати новий функціонал до системи.

Однак слід бути уважним при розробці нових функцій, щоб уникнути зайвої складності та забезпечити правильну взаємодію між різними версіями системи. Важливо визначити чіткі і стандартизовані правила для версіонування інтерфейсів та дотримуватися їх, щоб забезпечити сумісність між старими та новими версіями.

Керування розширенням функціоналу системи віддаленого виклику процедур важливо для забезпечення її ефективності та підтримки протягом тривалого часу. Також слід розглядати можливості управління конфігурацією та завантаженням системи, щоб уникнути непотрібного перевантаження функціональністю та забезпечити оптимальну продуктивність.

Виконання запиту у системах, які використовують технологію віддаленого виклику процедури, має наступні шаги:

1. В системі відбувається виклик клієнтського стабу (проксі), який являє собою заглушку та при виклику він отримує дані з пам’яті комп’ютера, серелізує їх у формат запиту та відправляє його.
2. Відправлений запит використовує обмовлений мережевий протокол, щоб дійти до віддаленого комп’ютера.
3. Комп’ютер отримавши запит викликає серверний стаб (скелет), який десерелізує його та виконує відповідні дії з використанням даних, які прийшли.
4. Як всі необхідні дії виконалися, серверний стаб серелізує дані у повідомлення та відправляє відповідь до клієнта використовуючи такий самий мережевий протоколи, який використовував для передачі повідомлення з клієнту на сервер.
5. Клієнт отримує відповідь та передає її у клієнтський стаб, який десерелізує її та повертає туди де його було викликано.

У технології RPC клієнт вважається той комп’ютер, який відправив запити на інший комп’ютер, а той який прийняв запит є сервером. На рис. 2.4 зображено процес виконання RPC запиту.

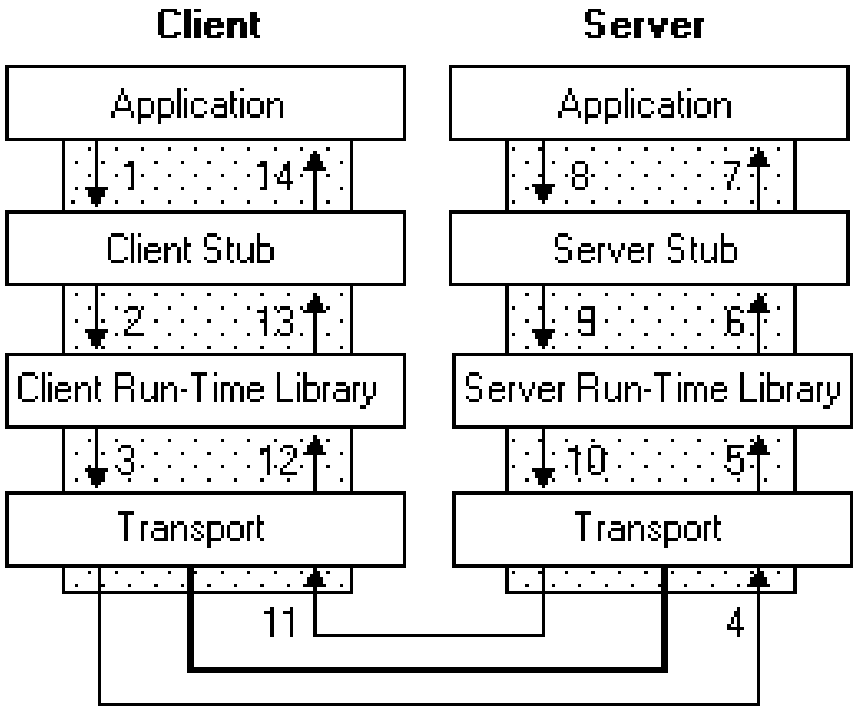


Рис. 2.4 Процес виконання RPC запиту [20]

gRPC один з найвідоміших RPC фреймворків на сьогоднішній день, який був розроблений Google та зараз має відкритий вихідний код [19].

Однією з ключових особливостей gRPC є використання Protocol Buffers (ProtoBuf) для серіалізації даних та опису інтерфейсів. ProtoBuf дозволяють зручно визначати структури даних та описувати віддалені інтерфейси за допомогою мови опису інтерфейсів. Це сприяє створенню ефективних та компактних бінарних форматів для передачі даних через мережу, забезпечуючи високу продуктивність, а також він може використовуватися агентами системи керування автоматизованим тестуванням для збереження тимчасових даних.

gRPC підтримує багато мов програмування, включаючи Java, Python, Go, C++, C# та інші [19], що робить його універсальним для використання в різних екосистемах розробки. Він також підтримує різні мережеві протоколи, такі як HTTP/2, що дозволяє використовувати його в різноманітних умовах та інтегрувати з існуючими веб-застосунками.

gRPC також надає різноманітні функції, такі як потокова передача даних, автентифікація, дескриптори служб, що роблять його ідеальним для розробки

розподілених систем, мікросервісів та інших сучасних застосунків. Його активна спільнота та підтримка з боку Google дозволяють забезпечити стабільність, безпеку та високу швидкість дії віддаленого виклику процедур у широкому спектрі проектів.

3 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ

3.1 Бізнес процес розробленої системи

В більшості випадках в бізнес процесі аналогів виконання автоматизованого тестування перекладалося на CI/CD засоби, через що системи втрачали можливість гнучкої інтеграції та взаємодії з різноманітним обладнанням. Це спричиняло втрату системами керування автоматизованим тестуванням ключової можливості адаптуватися до специфічних потреб тестування, що включають у себе взаємодію з різними пристроями та обладнанням.

Використання CI/CD засобів для виконання автоматизованих тестів може виглядати привабливою опцією з точки зору автоматизації процесів та неперервної інтеграції. Однак в багатьох випадках це перетворювалося на обмеження, оскільки CI/CD конфігурації визначали не лише час і частоту виконання тестів, але і умови їх запуску та способи взаємодії з обладнанням.

Системи, які були орієнтовані переважно на CI/CD, можуть втрачати гнучкість та розширюваність, особливо коли стикаються з нестандартними сценаріями тестування, такими як взаємодія з вбудованими системами, IoT-пристроями чи спеціалізованим обладнанням.

Тому за допомогою евристичних методів оптимізації, які спрямовані на знаходженні кращого рішення за прийняту кількість часу [21], було оптимізовано цей процес, щоб він мав пряму причетність до виконання автоматизованого тестування. Систему було розбито на два основних компонента, а саме сервер та агент. Кількість агентів залежить від потреба та можливостей користувача, вони встановлюються безпосередньо на комп’ютер, де буде проводитися тестування. Це надає можливість паралельного та розподіленого виконання тестів, підвищуючи ефективність та прискорюючи процес автоматизованого тестування. Розбиття системи на серверний та агентський компоненти створює гнучкість та масштабованість в процесі тестування.

Сервер виконує роль центрального управлінця та координатора тестових задач. Його функції включають розподілення тестових завдань між доступними агентами, збір та аналіз результатів тестів, а також взаємодію з іншими складовими системи. Це розділення відділяє логіку керування від виконавчої частини, що дозволяє ефективно використовувати ресурси та забезпечує легшу масштабованість.

Агенти, у свою чергу, встановлюються на різних комп'ютерах, де вони можуть взаємодіяти з локальним середовищем тестування. Вони відповідають за виконання конкретних тестових завдань, спілкування з тестовими скриптами та надсилання звітів про результати на сервер. Кількість агентів може бути адаптована до потреб конкретного тестування та наявних ресурсів, забезпечуючи гнучкість в конфігурації та оптимізацію використання обчислювальних потужностей.

Така архітектура дозволяє впроваджувати автоматизоване тестування в різноманітних проектах та умовах, забезпечуючи високий рівень автономії та розширюваності системи.

На рис. 3.1 зображений оптимізований бізнес процес системи керування автоматизованим тестуванням.

3.2 Процес виконання автоматизованого тестування

Всі запити на виконання автоматизованого тестування, які находять до серверу, потрапляють до планувальник. Це компонент серверу відіграє важливу роль у процесі виконання тестування, оскільки він відповідає за розподілення тестових завдань між доступними агентами та контроль за їх виконанням.

Всі запити на тестування в планувальнику знаходяться в черзі, тобто їх виконання відбувається за методом FIFO (First-in, First-out), що значить той запит який надійшов раніше виконається раніше. Це дозволяє гарантує об'єктивність та консистентність в обробці тестів.

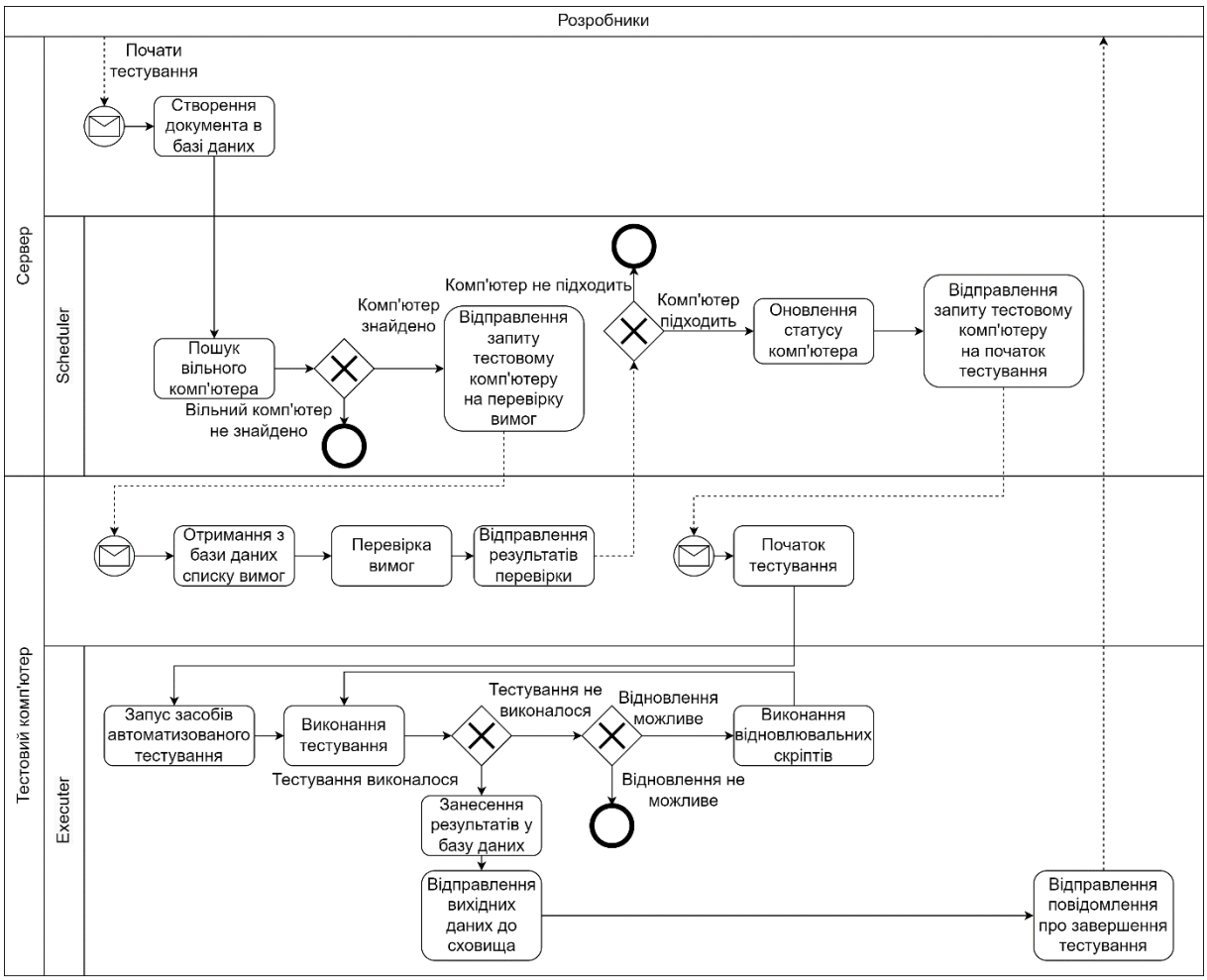


Рис. 3.1 Оптимізований бізнес процес системи керування автоматизованим тестуванням.

Метод FIFO важливий в контексті автоматизованого тестування, оскільки дозволяє об'єктивно визначати пріоритетність виконання тестових завдань на основі їхнього часу надходження. Це може бути особливо корисним в сценаріях, де важливо враховувати хронологію подій та забезпечити адекватну реакцію на невідкладні тестові запити.

Використання методу FIFO в планувальнику сприяє прозорості та ефективності в організації тестового процесу, а також спрощує управління завданнями та збереження логіки порядку їх виконання.

Перед тим як розпочати автоматизоване тестування, планувальнику треба дізнатися статус комп’ютера для того щоб визначити чи не використовується комп’ютер, де стоїть агент, людиною або іншим тестуванням на даний час. Та

також перевірити відповідність комп’ютера до вимог зарощеного тестування, оскільки для деяких тестів може бути критичним наявність певного додатка на комп’ютері або підключення до додаткового обладнання.

Тільки після перевірки цих аспектів планувальник записує цей комп’ютер у базу даних, та надсилає запит на виконання автоматизованого тестування до нього. Агент розпочинає тестування та по його завершення оновлю запис цього тесту у базі даних додаючи результати після чого він звітує серверу про завершення проведення тестування.

На рис 3.2 представлена діаграма послідовності виконання автоматизованого тестування.

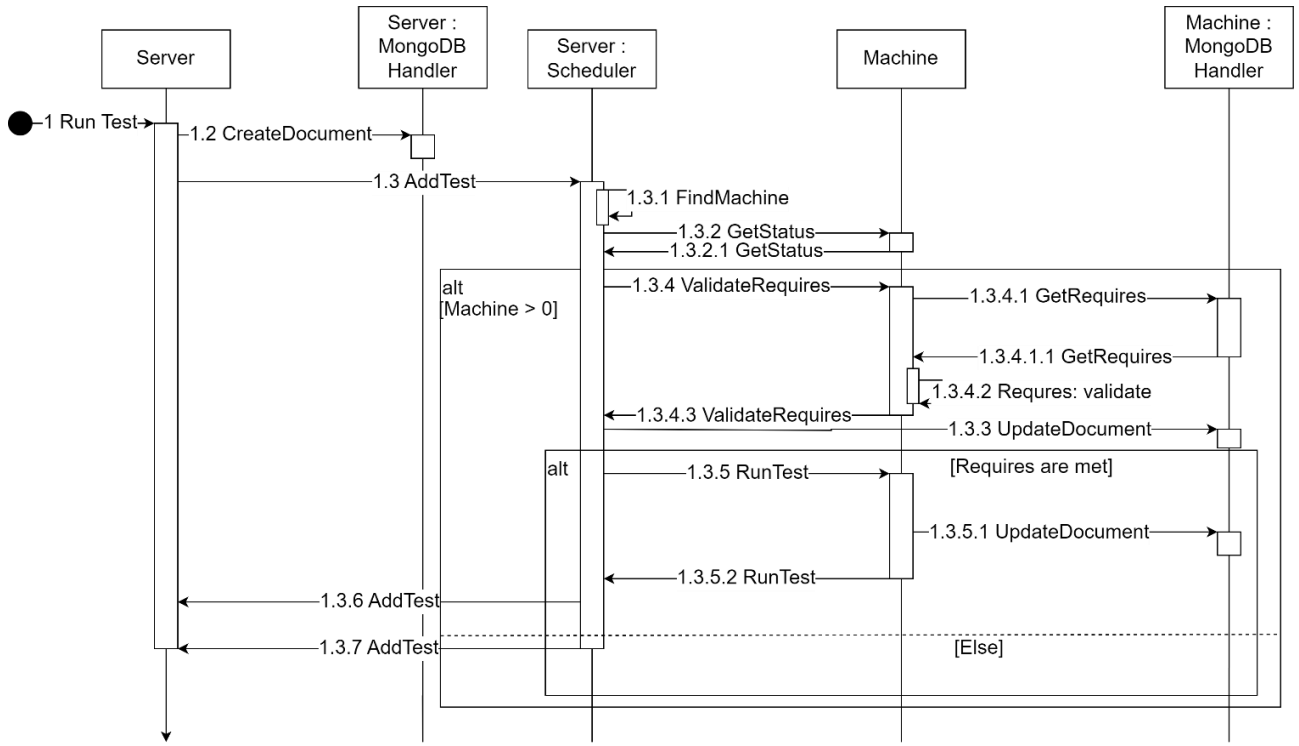


Рис 3.2 Діаграма послідовності виконання автоматизованого тестування

3.3 Математична модель оцінки системи

Найбільш важлива вимога до системи керування автоматизованим тестуванням – це її здатність до запуску автоматизованого тестування. Більшість існуючих систем аналогів перекладають цей функціонал на CI/CD засоби, через що виконання тестування залежить від налаштувань конвеєру команд.

Залежність від конвеєрів команд може призвести до обмежень у гнучкості та контролі над тестами. При налагодженні тестових процесів на CI/CD конвеєрах, система може підкорятися внутрішнім обмеженням цих інструментів, що може призвести до складнощів у налаштуванні або обмеження в можливості налаштувати тести під конкретні потреби проекту.

Крім того, інтеграція з CI/CD може вимагати значних зусиль для забезпечення сумісності з іншими елементами конвеєру та підтримки різних інструментів. У деяких випадках це може вплинути на простоту налаштування та розгортання тестових сценаріїв.

Більш того, в разі обширних або складних тестових наборів, робота з CI/CD засобами може призвести до збільшення часу виконання тестів або до перенавантаження конвеєрів. Це може створювати проблеми з продуктивністю та швидкістю розробки, коли тестування стає буттям для ітераційного процесу.

Параметр, який відповідає за оцінку здатності системи до запуску тестування, позначається через $execute_i$ та представлений у числовому вигляді за шкалою [0; 2]:

— 0 – система не можу запускати тестування;

— 1 – система може запускати тестування тільки за допомоги CI/CD;

— 2 – система може запускати тестування.

Другою не менш важливою вимогою до системи є її можливість до модифікування оскільки часто для взаємодії з пристроями треба змінювати систему для взаємодії з пристроями, система керування автоматизованим тестуванням повинна мати гнучкість і можливість модифікації. Це особливо

важливо у випадках, коли пристрої мають різні конфігурації, версії або вимагають специфічних налаштувань для виконання тестів.

Гнучкість системи тестування означає здатність швидко адаптувати і модифікувати тестові сценарії, налаштування та параметри, щоб вони відповідали новим умовам або вимогам конкретних пристроїв. Наприклад, це може включати налаштування з'єднання з пристроями, вибір необхідних драйверів чи зміну тестових скриптів під нові параметри.

Забезпечення можливості модифікації системи тестування також може включати розширення бази даних або налаштування інтерфейсів для взаємодії з новими пристроями. Крім того, це означає здатність легко внести зміни у тестові скрипти чи управляючі програми для підтримки нових функцій або пристроїв.

Цю можливість системи виражено показником modifiable_i який є оцінкою функціональної здатністю до модифікування та представлений у числовому вигляді за школою [0; 2], де:

— 0 – система не може бути модифікована;

— 1 – система може бути модифікована обмежено;

— 2 – система може бути модифікована.

Також важливим є можливість системи керування автоматизованим тестуванням до взаємодії з тим обладнанням, яке використовується для тестуванням, тому що ця взаємодія напряму впливає на ефективність та результативність тестів. Система керування автоматизованим тестуванням повинна мати можливість не лише виконувати тести, а й здатність до інтегрування з широким спектром тестувального обладнання.

Необхідність взаємодії із специфічним обладнанням стає важливою у випадках, коли тестування вимагає підключення до різних пристроїв чи платформ. Наприклад, для мобільного тестування система повинна мати здатність взаємодіяти з різними моделями та версіями мобільних пристроїв та платформ. В умовах вбудованих систем або IoT, важливо мати засоби взаємодії з конкретним обладнанням, що тестується.

Показник hardware_i представляє оцінку здатності системи до взаємодії з обладнанням та виражається у числовій формі за шкалою [0; 2], де:

— 0 – система не має можливості взаємодіяти з обладнанням;

— 1 – система має обмежену можливість взаємодіяти з обладнанням;

— 2 – система має можливість взаємодіяти з обладнанням.

Щоб визначити нормованого значення кожного з параметрів для *i*-й системи розраховується за формулами (3.1), (3.2) та (3.3):

$$\widetilde{execute} = \frac{execute_i}{\max(execute_i)}, \tag{3.1}$$

$$\widetilde{\text{modifiable}} = \frac{\text{modifiable}_i}{\max(\text{modifiable}_i)}, \tag{3.2}$$

$$\widetilde{\text{hardware}} = \frac{\text{hardware}_i}{\max(\text{hardware}_i)}, \tag{3.3}$$

Адаптивна модель для розрахунку рангу систем керування автоматизованим тестуванням представлена виразом (3.4):

$$R_i = \widetilde{execute} + \widetilde{\text{modifiable}} + \widetilde{\text{hardware}}, \tag{3.4}$$

Визначення кращої системи виконується за формулою (3.5):

$$K = \arg\max(R_i),\ i = \overline{1,\ n}, \tag{3.5}$$

Результати розрахунків рейтингів на основі моделі (3.4), (3.5) представлені в таблиці 3.1. Кращою системою є розроблена, K=3, та таблиці 3.1.

Таблиця 3.1

Результати розрахунків рейтингів

Назва	$execute$	modifiable	hardware	R
TestRail	1	0	0	0.5
Qase	1	0	0	0.5
Allure Testops	1	0	0	0.5
Browserstack	2	0	0	1
Розроблена система	2	2	2	3

ВИСНОВКИ

Під час аналізу існуючих систем керування автоматизованим тестуванням, а саме TestRail, Qase, Allure Testops, Browserstack, виявлено, що у більшості випадків вони спираються на використання CI/CD для керування цим процесом. Це призводить до поганої взаємодії з обкладенням та залежність від функціоналу певного CI/CD засобу, що позбавляє їх гнучкості у використані.

Розроблено систему керування автоматизованим тестування, яка має сервер-клієнт архітектура, а саме брокерська, оскільки це допомогло системі бути більш гнучкою та мати можливість використовувати програми агентів, які встановлюються на комп'ютери, де буде проводитися тестування, що дозволило збільшити здатність системи до взаємодії з обладнанням. Архітектура серверу та агентів є мікроядерною, через що вони мають гарну здатність до модифікування.

Для оцінки системи було розроблено математичну модель. Параметри системи які враховуються у моделі було обрано такі: здатність системи до запуску тестування, можливість системи до модифікування та можливість системи до взаємодії з обкладенням.

Проведено порівняльний аналіз, розробленої системи та існуючими аналогами за допомогою розробленої математичної моделі, який показав що розроблена система керування автоматизованим тестуванням має рейтинг 3, який є найвищим серед систем, які оцінювалися.

Розроблена система має ряд переваг перед існуючими системами саме в плані керування автоматизованим тестуванням, але її бракує деяких функції які призначення для ручного тестування, наприклад, створення тест кейсів, тестових наборів та інше. Вона може бути застосовано на проектах які додаткових засіб який дозволить задіяти потужності, які є на проекті, більш ефективно, оскільки дозволить використовувати не тільки спеціально виділені комп'ютери, а й комп'ютери працівників, які будуть вільні. Також вона дозволить щільніше взаємодіяти з обладнанням, що зробить тестування ПЗ на них зручніше.

ПЕРЕЛІК ПОСИЛАНЬ

1. Galin D. Software Quality: Concepts and Practice. IEEE Computer Society Press, 2018. 720 с.є

2. Design, monitoring, and testing of microservices systems: The practitioners' perspective / M. Waseem та ін. Journal of Systems and Software. 2021. Т. 182. С. 111061. URL: https://doi.org/10.1016/j.jss.2021.111061

3. What industry wants from academia in software testing? [Електронний ресурс] / Vahid Garousi [та ін.] // EASE'17: Evaluation and Assessment in Software Engineering, Karlskrona Sweden. – New York, NY, USA, 2017. – Режим доступу: https://doi.org/10.1145/3084226.3084264 (дата звернення: 07.12.2023).

4. Test Case Management & Orchestration Software by TestRail. TestRail | The Quality OS for QA Teams. URL: https://www.testrail.com/

5. Qase | Test management software for quality assurance. Qase | Test management software for quality assurance. URL: https://qase.io/.

6. Allure TestOps - Full-stack Test Management. Allure TestOps - Full-stack Test Management. URL: https://qameta.io/.

7. Most Reliable App & Cross Browser Testing Platform. BrowserStack. URL: https://www.browserstack.com/.

8. Valburg J. v. Automated Testing and Profiling for Call of Duty [Електронний ресурс], 2020 / Jan van Valburg // YouTube. – Режим доступу: https://www.youtube.com/watch?v=8d0wzyiikXM.

9. Buildbot. Buildbot. URL: https://buildbot.net/.

10. Puppet Infrastructure & IT Automation at Scale | Puppet by Perforce. Puppet Infrastructure & IT Automation at Scale | Puppet by Perforce. URL: https://www.puppet.com/.

11. Leonard M. Why isn't all test automation run on the pipeline?. Medium. URL: https://medium.com/slalom-build/why-isnt-all-test-automation-run-on-the-pipeline-b2c57afbdf5a.

12. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media, 2020. 432 с.

13. Gorton I. Foundations of Scalable Systems: Designing Distributed Architectures. O'Reilly Media, Incorporated, 2022.

14. Bush J. Learn SQL Database Programming: Query and Manipulate Databases from Popular Relational Database Servers Using SQL. Packt Publishing, Limited, 2020. 564 с.

15. Ferrari L., Pirozzi E. Learn PostgreSQL 12: A Beginner's Guide to Building and Managing High-Performance Database Solutions Using PostgreSQL 12. Packt Publishing, Limited, 2020. 790 с.

16. Raj P., Deka G. C. Deep Dive into NoSQL Databases: The Use Cases and Applications. Elsevier Science & Technology Books, 2018. 400 с.

17. MongoDB Fundamentals A Hands-on Guide to Using MongoDB and Atlas in the Real World / A. Phaltankar та ін. Packt Publishing, 2020. 748 с.

18. Newman S. Building Microservices: Designing Fine-Grained Systems / Sam Newman. – 2-ге вид. – [Б. м.] : O'Reilly Media, Incorporated, 2021. – 615 с.

19. gRPC [Електронний ресурс] // gRPC. – Режим доступу: https://grpc.io/.

20. How RPC Works [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/rpc/how-rpc-works.

21. Davim J. P., Kumar K., Zindani D. Optimizing Engineering Problems Through Heuristic Techniques. Taylor & Francis Group, 2019. 138 с.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО - КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

МАГІСТЕРСЬКА РОБОТА
«РОЗРОБКА МЕТОДУ ТА ЗАСОБІВ РОЗПОДІЛЕНОГО АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

Виконав: студент групи ПДМ-64 Івлєв Ростислав Володимирович

Керівник: к.т.н., доц., доцент кафедри ІПЗ Золотухіна Оксана Анатоліївна

Київ - 2024

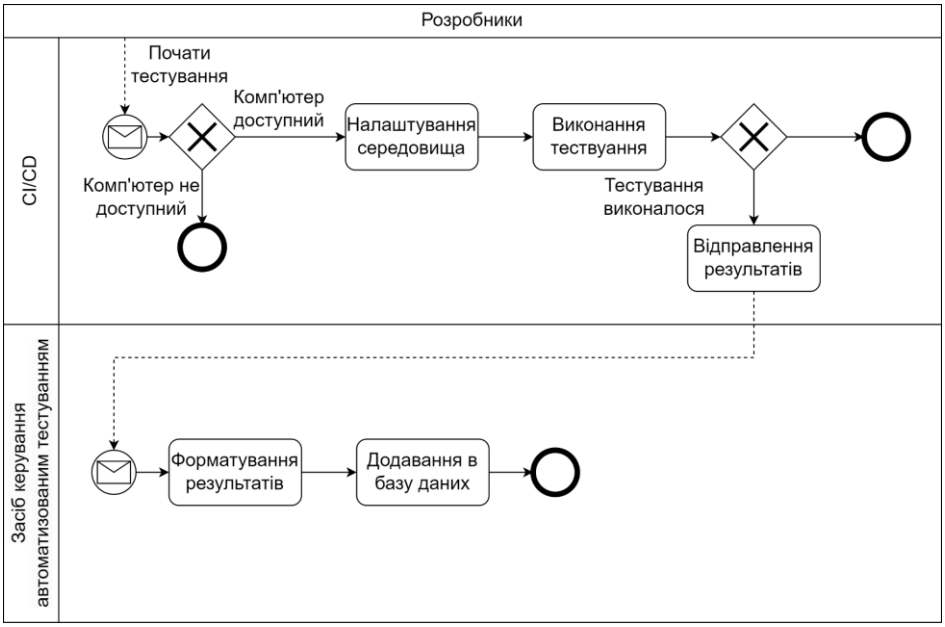
МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація процесу керування автоматизованим тестуванням за рахунок використання брокерської архітектури.

Об'єкт дослідження: процес керування автоматизованим тестуванням

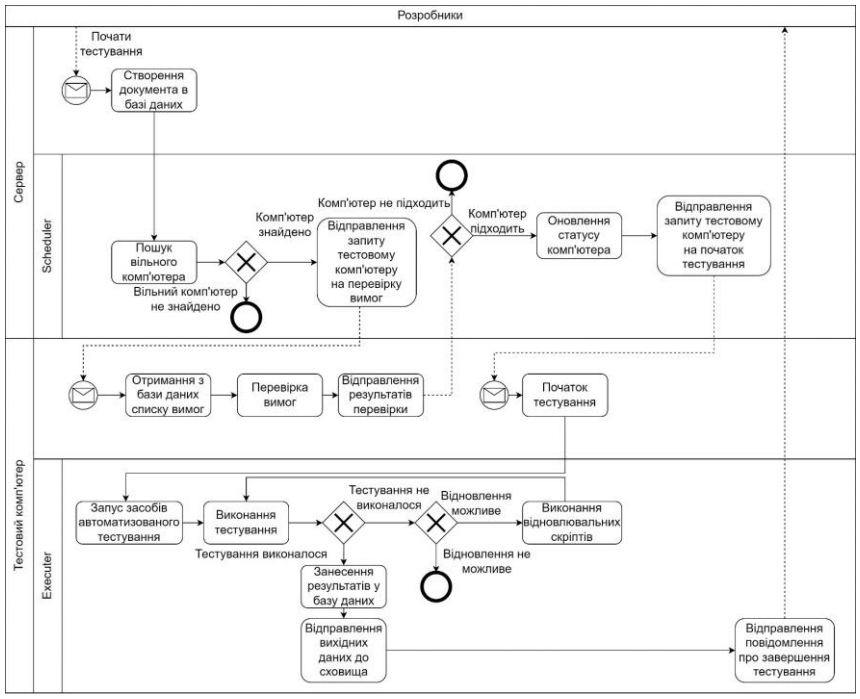
Предмет дослідження: методи та засоби керування автоматизованим тестуванням

БІЗНЕС ПРОЦЕС КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ



3

ОПТИМІЗОВАНИЙ БІЗНЕС ПРОЦЕС КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ



4

МАТЕМАТИЧНА МОДЕЛЬ

Можливість до запуску тестування - $execute_i$:

- 0 – система не можу запускати тестування;
- 1 – система може запускати тестування тільки за допомоги CI/CD;
- 2 – система може запускати тестування.

$$\widetilde{execute} = \frac{execute_i}{\max(execute_i)},$$

Можливість до модифікації – modifiable_i:

- 0 – система не може бути модифікована;
- 1 – система може бути модифікована обмежено;
- 2 – система може бути модифікована.

$$\widetilde{\text{modifiable}} = \frac{\text{modifiable}_i}{\max(\text{modifiable}_i)},$$

Взаємодія з обладнанням – hardware_i:

- 0 – система не має можливості взаємодіяти з обладнанням;
- 1 – система має обмежену можливість взаємодіяти з обладнанням;
- 2 – система має можливість взаємодіяти з обладнанням.

$$\widetilde{\text{hardware}} = \frac{\text{hardware}_i}{\max(\text{hardware}_i)},$$

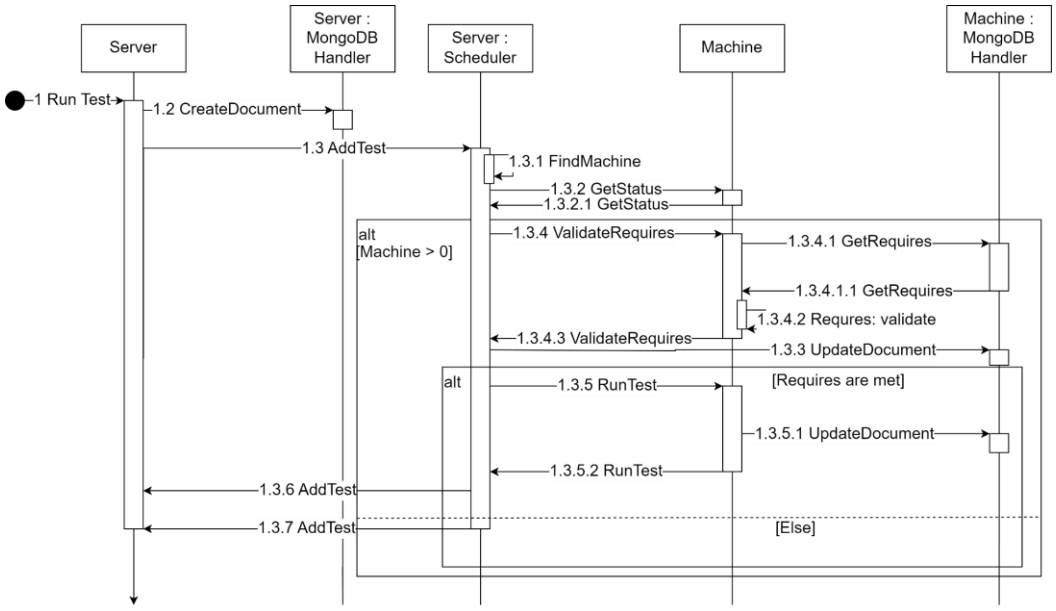
Рейтинг системи – R_i

$$R_i = \widetilde{execute} + \widetilde{\text{modifiable}} + \widetilde{\text{hardware}},$$

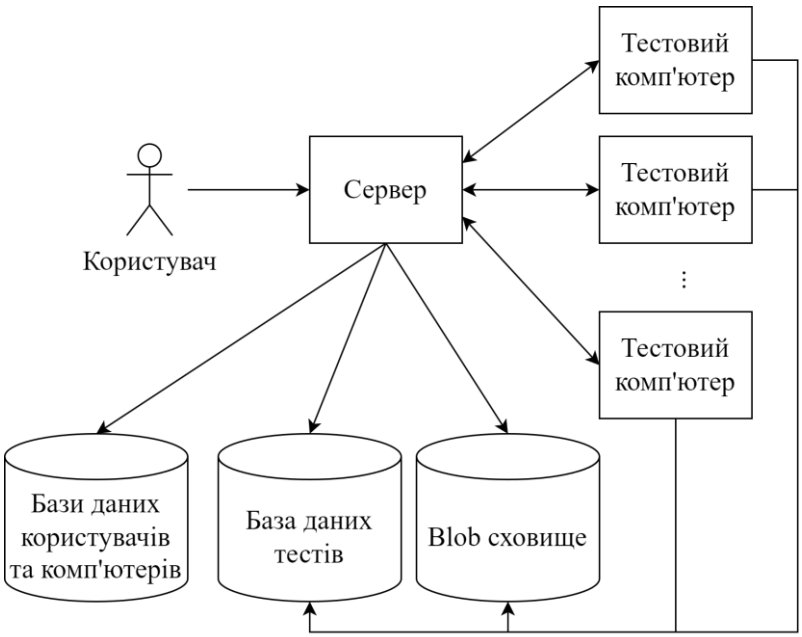
Краща система - K

$$K = \arg\max(R_i), i = \overline{1,n}.$$

ДІАГРАМА ПОСЛІДОВНОСТІ ВИКОНАННЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ



РОЗРОБЛЕНА АРХІТЕКТУРА СИСТЕМИ КЕРУВАННЯ АВТОМАТИЗОВАНИМ ТЕСТУВАННЯМ



ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕЗУЛЬТАТІВ МОДЕЛЮВАННЯ

Назва	*execute*	modifiable	hardware	*R*
TestRail	1	0	0	0.5
Qase	1	0	0	0.5
Allure Testops	1	0	0	0.5
Browserstack	2	0	1	1.5
Розроблена система	**2**	**2**	**2**	**3**

ВИСНОВКИ

1. Проаналізовані існуючи засоби керування автоматизованим тестуванням. Який показав, що в більшості випадках вони виконують керування автоматизованим тестуванням за допомогою CI/CD.
2. Розроблено систему керування автоматизованим тестування з брокерською архітектурою взаємодії між сервером та тестовими комп'ютерами. Вона дозволяє взаємодіяти системі з обкладенням де проводиться тестування та має можливість запуску тестування без допомоги CI/CD.
3. Розроблено математичку модель для оцінки систем керування автоматизованим тестування на основі таких параметрів, як можливість запускати автоматизоване тестування, можливість розширенню функціоналу, можливість взаємодіяти з обладнанням де проводиться тестування.
4. Проведений порівняльний аналіз існуючих аналогів та розробленої системи, який показав що розроблена система має найбільший рейтинг, а саме 3, коли найбільший ранг аналогів – 1.5.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті

1. Івлєв Р.В., Золотухіна О.А. Система керування автоматизованим тестуванням // Телекомунікаційні та інформаційні технологій, №1, 2024 – Подано до друку

Тези доповідей:

1. Івлєв Р.В. Аналіз засобів керування автоматизованим тестуванням програмного забезпечення/ Івлєв Р.В., Золотухіна О.А. // Проблеми комп'ютерної інженерії: Матеріали науково-технічної конференції. Збірник тез. 2023, ДУТ, м. Київ – К.: ДУТ, 2023. – Подано до публікації

2. Івлєв Р.В. Технологія віддаленого виклику процедури/ Івлєв Р.В., Золотухіна О.А. // Telecommunication: problems and innovation: Матеріали науково-технічної конференції. Збірник тез. 2023, ДУТ, м. Київ – К.: ДУТ, 2023. – Подано до публікації

ДЯКУЮ ЗА УВАГУ!