Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

РОЗРОБКА ПРИНЦИПІВ, ПІДХОДІВ ТА АРХІТЕКТУРИ ПІДСИСТЕМИ ДЛЯ РОЗПОДІЛЕНОГО НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ ТА АНАЛІЗУ РЕЗУЛЬТАТІВ У СИСТЕМІ CI/CD

Текстова частина
магістерської роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник магістерської роботи
доцент, д.н. Глибовець А.М.

_____
(підпис)

“____” _____ 2021 р.

Виконала студентка
Бенюх Л.І.

“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
к.ф-м.н., доц. Гороховський С.С
_____
(підпис)
“_____” _____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія Програмного Забезпечення Бенюх Ладі Ігорівній
Розробити Принципи, підходи та архітектуру підсистеми для розподіленого навантажувального тестування та аналізу результатів у системі CI/CD

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Теоретичні засади навантажувального тестування
2 Методології збору та моніторингу метрик продуктивності
3 Розробка методології та архітектури системи для високо-навантажувального тестування
Висновки по роботі та рекомендації для подальших досліджень
Список літератури
Додатки

Дата видачі “_____” _____ 2020 р.

Керівник
А.М. Глибовець, доцент, д.н. _____
(підпис)

Завдання отримала
Л.І. Бенюх _____
(підпис)

Тема: Розробка принципів, підходів та архітектури підсистеми для розподіленого навантажувального тестування та аналізу результатів у системі CI/CD

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	02.11.2020	
2.	Огляд технічної літератури за темою роботи	16.11.2020	
3.	Виконання аналізу сучасних рішень	30.11.2020	
4.	Порівняння інструментів навантажувального тестування та вибір необхідних	18.12.2020	
5.	Порівняння інструментів для тестування продуктивності та вибір необхідних	18.01.2021	
6.	Вибір інструментів для здійснення моніторингу	15.02.2021	
7.	Вибір інструментів для побудов звітів у реальному часі	15.03.2021	
8.	Огляд можливостей Kubernetes для побудови масштабованої і розподіленої архітектури навантаження	05.04.2021	
9.	Побудова архітектури розподіленого навантаження з інтеграцією в систему CI/CD	12.04.2021	
10.	Створення слайдів для доповіді та написання доповіді	27.04.2021	
11.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	05.05.2021	
12.	Коригування роботи за результатами попереднього захисту	21.05.2021	
13.	Остаточне оформлення пояснювальної записки та слайдів	31.05.2021	
14.	Захист магістерської роботи (проекту)	16.06.2021	

Студентка Бенюх Л.І.

Керівник Глибовець А.М. “____”_____ 2021 р.

ЗМІСТ

Анотація

В рамках даної дипломної роботи був проведений аналіз різних інструментів для здійснення навантажувального тестування та тестування продуктивності, масштабування таких тестів та централізованої звітності метрик. В результаті були запропоновані підходи та принципи до побудови сучасної архітектури для реалізації підсистеми навантажувального тестування в безперервній поставці коду на базі Kubernetes та Jenkins.

Ключові слова: навантажувальне тестування, тестування продуктивності, Kubernetes, масштабування тестів, велико-навантажуванні системи, централізована звітність результатів тестування, CI/CD, безперервна поставка коду, Gatling, Sitespeed.io.

ВСТУП

Актуальність. Тестування продуктивності системи та його важливість одночасно важко переоцінити чи недооцінити. Значно правильніше буде якщо розмовляти про вчасність цієї активності. Практично будь-яка цифрова система побудована за сучасними підходами та технологіями може працювати без будь-яких критичних проблем з власною продуктивністю. Водночас, для будь-якої такої системи, особливо коли вона стає популярною, з великою ймовірністю може настати такий момент, коли вона буде не спроможна справитись з постійно зростаючим навантаженням та стане просто нестійкою. Так, за даними досліджень компанії Gartner [3] середня вартість 1 хвилини падіння та недоступності ресурсу може коштувати 5 тис. умовних одиниць, тобто 300 тис. у.о. в хвилину. Інше дослідження показало, що для 98% великих інтернет ресурсів вартість такої однієї години складатиме більше чим 100 тис. у.о. [24]. Ба більше, для 33% цих ресурсів вартість падіння буде від 1 до 5 млн. у.о.

Тим не менше, більшість компаній які розробляють та підтримують власні цифрові рішення – від веб-сайтів до будь-яких інших цифрових систем – часто сфокусовані переважно на функціоналі цієї системи та його відповідності вимогам, а не на продуктивності системи в цілому. Такі наміри є досить природніми, адже система має правильно виконувати очікувані від неї функції. Коли компанії починають стикатися з проблемами продуктивності, вони намагаються якнайшвидше не оптимізувати роботу програмного забезпечення, а додати більше потужностей – вертикальне та горизонтальне масштабування. Ця стратегія спрацьовує, але має обмеження. Адже додавання додаткових ресурсів не може бути нескінченним і рано чи пізно впирається або в саму архітектуру системи, або в можливості самої компанії, тощо.

Саме тому, рекомендовано здійснювати навантажувальне тестування заздалегідь, планувати для цього час та ресурси, щоб мати достатньо часу на виправлення помилок, та в цілому розуміти границі системи. Водночас, для того щоб організувати повноцінне навантажувальне тестування необхідні підготовлені спеціалісти, інструменти та інфраструктура, особливо коли ми

говоримо про велике навантаження. Саме два останніх пункти – інструменти та інфраструктура – як їх вибрати та організувати, будуть описані в даній дипломній роботі.

Мета дослідження. Розробити сучасну архітектуру для здійснення комплексного масштабованого навантажувального тестування з інтеграцією в систему безперервної поставки CI/CD.

Завдання дослідження. Проаналізувати існуючі підходи для здійснення великого навантаження. Вивчити та порівняти програмні продукти для здійснення масштабованого навантаження, тестування продуктивності та централізованого відображення результатів. Перевірити можливості інтеграції інструментів та їх складових для побудови архітектури системи найоптимальнішим шляхом. Обрати підходи та інструменти для практичної реалізації даної архітектури та надати загальні рекомендації для її побудови.

Об´єкт дослідження. Інструменти для здійснення навантажувального тестування та перевірки продуктивності веб-сторінок у браузері. Системи та підходи для розгортання масштабованого середовища для здійснення такого тестування. Інструменти централізованого збору та візуалізації результатів у реальному часі.

Предмет дослідження. Можливість побудови системи для здійснення масштабованого навантажувального тестування. Дослідження шляхів інтеграції підсистеми у систему безперервної поставки коду CI/CD.

Джерела дослідження. Друкована література як в друкованому, так і в електронному форматі, спеціалізовані веб-сайти та форуми, відео-записи конференцій по відповідним темам, кодова база інструментів та систем.

Наукова новизна одержаних результатів полягає в створенні сучасної архітектури високо-навантажувальної та масштабованої системи навантажувального тестування.

Практичне значення одержаних результатів. Завдяки тому, що архітектура системи побудована із використанням інструментів з відкритим доступом до коду, вона дозволить оптимізувати витрати на навантажувальне тестування, відмовившись від використання подібних комерційних продуктів. Ба більше, такий підхід дасть змогу побудувати систему із використанням будь-яких машин – від власних серверів до хмарних провайдерів. В результаті, реалізація подібної системи на велико-навантажувальних проектах дозволяє перевіряти продуктивність програмного забезпечення на постійній умови та уникнути потенційних збитків.

РОЗДІЛ 1: Теоретичні засади навантажувального тестування

1.1 Визначення ролі та цілей навантажувального тестування

Продуктивність тієї чи іншої системи має великий вплив як на рівень задоволення її кінцевого користувача так і стабільність самої системи. Низька продуктивність системи, а іншими словами повільна її робота чи взагалі часткова недоступність, є однією з найбільш неприємних речей, яка здатна призвести до втрати користувачів веб-сайту та як результат до фінансових збитків.

На сучасному висококонкурентному ринку користувач має широкий вибір серед альтернатив, у тому числі серед різних інтернет-сервісів. В більшості випадків саме кінцевий споживач обирає той чи інший ресурс, а не навпаки. Таким чином, якщо йому не подобається як працює певний веб-сайт то всього у декілька кліків можна знайти інший. І досить важливу роль у цьому процесі починає відігравати саме продуктивність сервісу. Адже якщо веб-сайт повільний і користувачу доводиться очікувати поки виконається та чи інша транзакція, то це аж ніяк позитивно не може вплинути на сприйняття та рівень задоволення. Користувач просто переключається на інший ресурс і в більшості випадків його повернути стає практично неможливо.

Тобто можна зробити висновок, що низькі інвестиції в продуктивність системи можуть взагалі поставити під питання фінансовий успіх існування бізнесу. Справедливим є і зворотній висновок - чим краще, або іншими словами швидше та злагоніше, працює веб-ресурс, тим приємніше ним користуватися, тим довше користувач буде на ньому перебувати, що звісно потенційно принесе позитивні фінансові результати та збільшення трафіку.

За даними ресурсу www.websitebuilderexpert.com, повільні онлайн-магазини, тобто з незадовільною продуктивністю, отримують наступні негативні наслідки [22]:

- 25% користувачів залишать ресурс якщо швидкість завантаження сторінки складатиме більше ніж 4 секунди.
- 46% таких користувачів не повернеться.

- 64% користувачів, які незадоволені швидкістю роботи системи, почнуть шукати інший аналогічний ресурс.
- Кожна наступна 1-секунда затримки, призводить до зменшення рівня задоволеності на 16% [22].

З іншої сторони, давайте подивимося, як оптимізація швидкості завантаження веб-сторінки впливає на ключові фактори успіху веб-сайту. Так, за даними ресурсу www.altexsoft.com оптимізована продуктивність системи має вплив на наступні показники сайту [34]:

- **Конверсія**. Конверсія веб-сайтів є важливим фактором успіху бізнесу, а це означає те, що ви знаходите підхід до користувача, щоб наштовхнути його на необхідні для вашого бізнесу дії. Наприклад, вони купують ваш товар, підписуються на розсилки новин, реєструються на вебінар або завантажують ту чи іншу інструкцію чи специфікацію товару. Чим швидше сторінка завантажується, тим вищий коефіцієнт конверсії вона матиме. Згідно з дослідженням Hubspot, затримка в 1 секунду означає зменшення конверсій на 7 відсотків. Наприклад, уповільнення сторінки на 1 секунду може коштувати Amazon щорічно 1,6 мільярда доларів продажів.
- **Ранжування ресурсу.** Час завантаження веб-сайту також впливає на те, як легко користувачі можуть знайти сам веб-сайт. Швидкість веб-сайтів є одним із тих факторів, який Google враховує при ранжуванні. Веб-сайт із низькою продуктивністю має невисокий рейтинг і як результат видається пошуковою системою одним із останніх. З грудня 2017 року пошукова система Google розпочала ранжування також на основі продуктивності мобільних версій сторінок, навіть для пошуку на комп'ютері. Метою цього рішення є захистити користувачів від веб-сайтів які мають низьку продуктивність і не підтримуються широким спектром пристроїв.
- **Зручність використання.** Зручність використання веб-сайтів, час завантаження та реакція веб-сайту на запити користувачів, безпосередньо впливають на лояльність клієнтів. Чим ефективніше працює веб-сайт, тим більш

задоволеним буде користувач. Позитивний досвід використання є запорукою для збільшення кількості користувачів та створення сильного бренду.

Але що входить до згаданого поняття «продуктивність системи»? - Якщо ми розглянемо стандарт ISO25010 [ISO25000], то побачимо що він відносить до продуктивності (ефективності) системи наступні критерії [20]:

- **Часова реакція системи** (Time behaviour) – наскільки система відповідає вимогам з швидкості, тривалістю опрацювання запитів, а також її пропускна спроможність.
- **Споживання ресурсів** (Resource utilization) – наскільки рівень споживання різних типів ресурсів протягом функціонування системи відповідає очікуванням.
- **Ємність** (Capacity) – встановлюють границі, за яких функціонування системи вважається задовільним.

Саме навантажувальне тестування дає змогу перевірити та оцінити згадані вище критерії. Так, спеціаліст з даного виду тестування Скотт Барбер дає наступне визначення: «Тестування продуктивності – це активність, яка дає змогу оцінити розмір трафіку який може обробити веб-ресурс перед тим як упаде або перед тим як реальні користувачі почнуть писати гнівні відгуки. Тобто, це є нічим іншим як справжнім випробуванням вашого сайту, симулюючи реальні умови його використання, та пошук помилок продуктивності до того як їх знайдуть кінцеві користувачі» [30].

Саме тестування продуктивності системи допоможе розв'язати та проаналізувати систему згідно стандарту ISO25010 [ISO25000], а саме:

- проаналізувати швидкість, масштабованість, стабільність та стійкість веб-додатку.
- перевірити неефективне використання пам’яті системи під певним навантаженням.
- допоможе знайти вузькі місця, які не дають змоги віднайти інші підходи тестування.

Також, ми б додали такий важливий аспект як **DDoS атаки**, які стали досить частим інструментом брудної конкуренції. Згідно Amazon – атака типу «відмова в обслуговуванні (DoS) – це спроба спричинити шкоду, через недоступність цільової системи, наприклад веб-сайту, для кінцевих споживачів. Зазвичай зловмисники генерують велику кількість пакетів чи запитів, які в кінцевому рахунку перевантажують цільову систему [32]. Саме навантажувальне тестування є одним з тих інструментів який дає змогу оцінити готовність системи до відбиття подібних DDoS атак.

1.2 Види та напрямки навантажувального тестування

Існує декілька різних класифікацій тестування. Давайте розглянемо класифікацію, яка подана у сертифікації ISQTB з тестування продуктивності та є досить повною на нашу думку. Отже, існують такі моделі навантажувального тестування [17]:

- **Тестування продуктивності системи або performance testing** – поняття, яке поєднує усі види та моделі тестування продуктивності (ефективності) системи або окремого її компонента.
- **Навантажувальне тестування або load testing** – модель навантаження, за якої симулюється синтетичне навантаження близьке або більше на 10-20% від реального. Таким чином перевіряється як працює система у близьких до реальних умов.
- **Тестування масштабування або scalability testing** – перевірка на скільки система здатна масштабуватися (вертикально чи горизонтально) під постійно-зростаючим навантаженням зі сторони кінцевих користувачів. При цьому важливо, щоб система і надалі була спроможна виконувати свій основний функціонал в очікуваних рамках продуктивності.
- **Стрес тестування або stress testing** – модель, ціль якої знайти межі функціонування системи, а також перевірити як та чи система повернеться до

нормального функціонування після граничного або вищого за граничне навантаження.

Два останніх види тестування – стрес та масштабування – дозволяють визначити границі системи та в результаті використовувати їх під час автоматичного моніторингу систем для швидкої реакції та автоматичного масштабування.

- **Тестування на виснаження або endurance testing** – симулюють тривале навантаження – рівня близько 50% від максимального – протягом декількох годин, а інколи і діб (зазвичай 4-8 годин). Таке тестування може допомогти виявити проблеми з використання системних ресурсів – оперативної пам’яті, автоматичне очищення пам’яті, перевикористання підключень до бази даних та звільнення потоків процесора. Найчастіше воно допомагає знайти ті проблеми, які важко виявити під час короткого навантаження, але які часто виникають в реальних умовах використання системи.
- **Тестування піків або spike testing** – при цій моделі навантаження швидко зростає на короткий час, навіть за межі границь, і так само швидко повертається до нормального рівня. Основне завдання тут є оцінка здатності системи до стабілізації після такого різкого навантаження.

  Це є досить звична поведінка для онлайн-магазинів, коли ідуть різні розпродажі, акції та інші маркетингові активності.
- **Тестування на паралельність або concurrency testing** – спрямоване часто на окремий функціонал з ціллю оцінити скільки одночасних запитів однакового характеру може обробити система, таких як відкриття певної сторінки чи здійснення купівлі товару.
- **Тестування на ємність або capacity testing** – модель яка покликана визначити за яких меж система продовжує функціонувати в рамках очікувань. На ємність можуть перевіряти за кількістю певних запитів чи за розміром опрацьованих даних.

Наведена вище класифікація ISTQB описує типи для здійснення тестування продуктивності спрямованого саме на **серверну частину**. Тестування

продуктивності серверної частини веб-додатків є важливим процесом, який допомагає зрозуміти, як ресурс поводиться під час навантаження. Це допомагає командам з розробки програмного забезпечення точно налаштувати програми, щоб отримати найкращу продуктивність, зберігаючи при цьому низькі витрати на інфраструктуру.

Додатково до серверного навантаження, необхідно виокремити також і **тестування продуктивності на клієнтській стороні**, тобто швидкість відтворення веб-сторінки ресурсу в браузері. Продуктивність системи на стороні веб-браузера багато в чому залежить від швидкості з'єднання з сервером. Але за умови, коли дана швидкість з’єднання є задовільною, на перший план виходить швидкість обробки та показу сторінки веб-браузером. Адже може скластися така ситуація, коли серверне навантаження показало позитивні результати тоді як кінцеві споживачі і досі залишаються не задоволені роботою веб-сайту, саме через неефективність роботи системи у браузері. Слід зазначити, що частину свого часу веб-браузер витрачає на завантаження сторінки HTML, чекаючи на завантаження картинок та іншого графічного контенту, текстів, форматування та верстки (CSS), функціональних елементів (кнопки, посилання, форми тощо), що відображається кінцевому користувачу.

Слід додати, що тестування продуктивності клієнтської сторони також можна розділити на наступні типи:

- *Тестування Персонального Комп’ютера* – тестування у браузері на ПК.
- *Тестування Мобільних* – тестування у браузері на мобільному пристрої.
- *Тестування з обмеженими ресурсами* – часто, під час тестування клієнтської сторони, симулюють такі ситуації як слабкий інтернет чи недостатність ресурсів мобільних телефонів та персональних комп’ютерів.

Також доречним буде порівняти тестування продуктивності на сервері та у браузері, див. Таблиця 1.2.1.

Таблиця 1.2.1 Порівняння серверного та браузерного тестування продуктивності

Фактор	Сервер	Браузер
Покриття тесту	Логічні сценарії, компоненти	Сценарії від початку до кінця
Фокус	Час відповіді сервера, поведінка системи під навантаженням	Рендеринг фронтенд елементів (JavaScript, CSS)
Модель навантаження	Велика кількість користувачів	Один користувач
Тестова інфраструктура	Вимагає великих та виокремлених інфраструктурних рішень	Вимагає симуляції з різних пристроїв та браузерів

Як бачимо з таблиці вище, ці два підходи – серверне та браузерне тестування - мають взагалі різний фокус та моделі. У той же час вони не суперечать друг другу, а є взаємодоповнюючими та одночасно ефективними методами для оцінки продуктивності системи, просто з різних сторін. Таким чином, загальний час відгуку який впливає на кінцевого користувача, має вигляд:

$$\boldsymbol{R_t = R_s + R_c}, \text{де} \tag{1.2.2}$$

$$R_t - \text{загальний час відгуку сторінки}$$

$$R_s - \text{час відгуку сервера}$$

$$R_c - \text{час рендерингу браузера}$$

1.3 Інструменти для тестування продуктивності серверної і веб-браузерної частини. Розробка критеріїв їх вибору

За останні десятиріччя на ринку з’явилася достатня кількість різних інструментів для тестування продуктивності системи, як платних так і безкоштовних з відкритим кодом (open-source). Разом з тим, це різноманіття інструментів може призвести до неефективного використання як фінансових так і людських ресурсів у випадку невірного вибору. Саме тому, в умовах зростаючого інтересу до навантажувального тестування з однієї сторони, так і широти пропозиції інструментів з іншої, виникає потреба у виокремленні певних критеріїв, які б допомогли зробити правильне рішення та обрати найкращий для тієї чи іншої організації інструмент для здійснення навантажувального тестування.

До таких критерії ми виокремили:

1. **Простота використання.**

Усі етапи пов'язані з тестуванням продуктивності системи, такі як створення скриптів, виконання тестів та аналіз результатів, повинні бути досить логічними та інтуїтивно зрозумілими. Як правило, ми повинні мати можливість створити базовий сценарій та запустити перший тест з навантаження протягом декількох годин.

2. **Підтримка технологій та оновлення.**

Хороший інструментарій для тестування продуктивності системи повинен забезпечувати підтримку збору метрик та відтворення сценаріїв у найбільш використовуваних пристроях та технологіях, таких як веб-браузери, мобільні пристрої, методи Web 2.0 та API.

3. **Реалістичне моделювання поведінки користувачів.**

Обмежені методи моделювання поведінки користувача можуть призвести до неточних результатів тестування. Зі зростанням кількості браузерних додатків основна увага приділяється реальному моделюванню користувачів на основі браузера. Забезпечення методів моделювання поведінки користувачів є одним з найважливіших критеріїв, оскільки неправильне моделювання поведінки користувачів зробить тест хибним і неточним.

4. **Рівень адаптації під проект, включаючи звітування та інтеграцію в CI/CD процес.**

Підхід та інструменти до тестування продуктивності повинні бути сумісними з тими, що використовуються командою розробки. Це допоможе не витрачати час та кошти для покупки ліцензій та адаптації інструментарію для навантажувального тестування. До таких ми б віднесли контроль версій та адаптацію під присутні системи CI/CD (Jenkins, TeamCity, GitLab).

5. **Можливість генерації різного розміру навантаження (дистрибутивне навантаження).**

Інколи достатньо невеликого навантаження для перевірки продуктивності системи. У інших випадках, вимагається побудова цілої системи з навантаження

яка включає в себе не одну машину та має централізоване управління та звітування. Інструмент має давати змогу побудувати таку дистрибутивну систему та здійснити тестування високо-навантажувального продукту.

6. Спільнота для підтримки інструменту.

Тестування продуктивності системи може бути складним процесом і часто вимагає допомоги досвідчених розробників або спеціалістів у цій галузі. Тому важливо отримати доступ до бази знань та спеціалістів з використання того чи іншого інструменту, оскільки вони можуть допомогти вирішити багато пов’язаних з ним запитань чи проблем.

7. Витрати на ліцензію.

Бажано, щоб початкові інвестиції в інструменти для побудови системи з тестування продуктивності були невеликими. Особливо якщо безкоштовні інструменти підходять для вашої інфраструктури. Також слід пам’ятати, що використання інструментів для тестування продуктивності з відкритим кодом дасть змогу уникнути потенційних платних оновлень або тих чи інших обмежень пов’язаних з різницею у пропозиціях в залежності від ціни інструмента.

Оцінимо присутні на ринку інструменти для серверного та веб-браузерного тестування продуктивності. Нижче в Таблиці 1.3.1 наведена порівняльна матриця безкоштовних інструментів, що використовуються у серверному навантаженні.

Таблиця 1.3.1 Порівняльна матриця безкоштовних інструментів для серверного навантаження

Назва	Поріг входу	Адаптація під проект	Репорт	Інтеграція з CI/CD	Дистрибутивне навантаження	Рівень нава-ння
JMeter [45]	Низький, базові знання Python, Groovy	Висока, підтримка широкого спектру технологій	Широкий вибір, можливість інтеграції з іншими системами репорту	Так, відсутній контроль версій	Так	Середній

Продовження Таблиці 1.3.1 Порівняльна матриця безкоштовних інструментів для серверного навантаження

Назва	Поріг входу	Адаптація під проект	Репорт	Інтеграція з CI/CD	Дистрибутивне навантаження	Рівень навання
Gatling [38]	Високий, середні знання Scala, Java	Вище середнього, підтримка широкого спектру технологій	Стандартний репорт, є можливість інтеграції з InfluxDB через Graphite протокол	Так	Можливе але без централізованого репорту	Високий
Locust [51]	Високий, середні знання Python	Вище середнього, дає можливість використання різних інструментів	Обмежений репорт, але є можливість його розширити засобами Python	Так	Можливе але без централізованого репорту	Значний
K6 [46]	Високий, середні знання Go	Вище середнього, дає можливість використання різних інструмент	Репорт присутній, але досить обмежений у порівнянні з іншими інструментами	Так	Так, з певними обмеженнями	Високий

Ціль даної дипломної роботи є побудова системи для здійснення тестування, симулюючи велике навантаження. Тому за критерієм «Рівень навантаження» найкращими є такі інструменти як Gatling та K6. З іншої сторони, вони не дають змоги організувати дистрибутивне навантаження, мотивуючи користувачів переходити на їх платні версії. У той же час, існують підходи які дадуть змогу організувати дистрибутивне навантаження, використовуючи дані інструменти. Ми зупинимось на такому інструменті як Gatling, адже мова його програмування Scala є досить знайомою для нас і входить до сімейства мов Java. До переваг даного інструменту, можна також віднести:

- написання сценаріїв з використанням коду – Tests as Code (а не плагінів як у випадку JMeter);
- інтеграція в систему з контролю версій;
- інтеграція в CI/CD;

- багато-платформний (JVM);
- інтеграція з InfluxDB та Grafana для репортингу;
- завдяки технології Akka [35] дає можливість згенерувати велике навантаження;

Окрім того, на ринку також присутні і платні та готові до використання системи з генерації навантаження. Їх перевагою є те, що часто вони не вимагають глибоких знань з будь-якої мови програмування та можуть симулювати досить велике навантаження. У той же час, вони досить часто можуть бути використані лише для простих сценаріїв та є платними. Ба більше, вони часто є хмарними, що не дозволяє їх використовувати у тих компаніях політика яких обмежує використання хмарних технологій (банківська сфера, державні системи). У таблиці 1.3.2 наведене порівняння основних таких систем.

Таблиця 1.3.2 Порівняльна матриця хмарних інструментів з серверного навантаження

Назва	Ціна, 2021	Адаптація під проект	Репорт	Інтеграція з CI/CD
Blazemeter [36]	~15,000 у.о. / рік	Високий, дає можливість використання різних інструментів	Так. В реальному часі	Так
LoadUI [50]	~11,000 у.о. / рік	Середній	Так. В реальному часі	Так
LoadRunner [49]	~7,000 – 15,000 у.о. / рік	Високий, дає можливість використання різних інструментів	Так. В реальному часі	Так
Gatling Frontline [39]	~27,000 у.о. / рік	Високий	Так. В реальному часі	Так
K6 Cloud [46]	~15,000 у.о. / рік	Середній	Так. В реальному часі	Так

Як бачимо, платні системи є готовим рішенням яке дає змогу здійснити велике навантаження з централізованими репортами. У той же час, вартість таких систем є досить значною.

Перейдемо тепер до інструментів, які використовуються для тестування продуктивності клієнта (браузера). До найпопулярніших таких, можна віднести:

- WebPageTest [58].
- Sitespeed.io [56] .
- PageSpeed Insights [54].

- Lighthouse [48].

Аналіз даних інструментів показав, що кожен з них перевіряє основні метрики, а також додаткові але особливі кожному інструменту. Саме тому загальною рекомендацією яку ми можемо дати, це використовувати декілька з них одночасно. У той же час, такий інструмент як Sistespeed.io [56] дозволяє інтегруватися в систему безперервної поставки коду (CI/CD) та вивести репорти у системі Grafana. Саме тому ми будемо використовувати цей інструмент в даній дипломній роботі.

РОЗДІЛ 2: Методології збору та моніторингу метрик продуктивності

2.1 Підходи до моніторингу продуктивності системи

Системи програмного забезпечення стають все складнішими. Хмарні технології, автоматичне масштабування, мікросервісна архітектура та інше, значно впливають і на підходи до аналізу та моніторингу таких систем. Таким чином, моніторинг продуктивності сучасних цифрових систем є досить непростим завданням. Кількість метрик які можуть бути зібрані інструментами моніторингу вимірюються уже сотнями, а то і тисячами. Хоча автоматизація, а також машинне навчання, значно допомагають в їх аналізі, досі доволі часто важко зрозуміти з чого розпочати аналіз та діагностику у випадку виявлення проблем з продуктивністю. Ба більше, навіть у звичайному режимі моніторингу за відсутності проблем часто доволі легко загубитися у зібраних метриках та їх взаємозв'язках. Виникає закономірне питання як ці метрики структурувати та пріоритезувати?

Слід додати, що збір системних метрик також вимагає ресурсів. Час CPU витрачається на збір та збереження метрик, що може негативно вплинути на продуктивність самої системи. Тобто виникає так званий *ефект спостерігача* [10], який говорить про те, що для того щоб аналізувати стан системи, то необхідно збирати та спостерігати за різними метриками системи. І в той же час, збір цих метрик впливає також на саму систему адже вимагає її ресурси. Дана закономірність також підштовхує нас до того, що необхідно слідкувати за найбільш значущими показниками.

Серед таких підходів, які допомагають структурувати як метрики так і сам їх аналіз, можна виділити наступні:

- USE метод (запропонований Брендан Греггом [10]).
- Four Golden Signals (запропонований компанією Google [8]).
- RED метод (є похідним від попереднього та запропонований Томом Вілкі [33]).

Давайте розглянемо кожен з методів зокрема та на скільки вони є взаємодоповнюючими загалом.

2.1.1 USE метод

Перший, USE метод був запропонований Бренданом Греггом в його роботі “System Performance” і фокусується на зборі інфраструктурних метрик. Зазвичай, до таких відносять наступні [10]:

- IOPS: операції читання/запис за секунду
- Throughput (пропускна здатність): операцій чи об’єм за секунду
- Utilization (утилізація)
- Latency (затримка)

Використання метрики пропускної здатності залежить від контексту. Так пропускна здатність Бази Даних зазвичай показує кількість операцій за секунду. Тоді як пропускна здатність Мережі показує кількість бітів або байтів (об’єм) за секунду.

В результаті Брендан Грегг групує усі інфраструктурні метрики у 3 групи [10] та виводить наступне правило – *для кожного Ресурсу, має бути перевірена Утилізація, Сатурація та Помилки*.

- **Resource** (R, ресурс): усі фізичні компоненти сервера (CPU, шини, оперативна пам'ять).
- **Utilization** (U, утилізація): частина часу у відсотках протягом якого ресурс був зайнятий обробкою того чи іншого процесу. Наприклад, утилізація процесорного часу CPU складає 80%.
- **Saturation** (S, сатурація): навіть виконуючи роботу, ресурс і далі може приймати нові завдання. Сатурація якраз відображає ту межу, після якої ресурс уже не в змозі приймати до черги нові завдання. Наприклад, черга завдань CPU складає 4.
- **Errors** (E, помилки): кількість помилок. Наприклад, даний мережевий інтерфейс має 50 колізій.

Саме застосування USE методу дає змогу досить швидко та точно визначити взаємозв’язки та корінь проблеми. Щойно певний ресурс досягає своєї повної утилізації, так званої ємності, він або ставить нові запити в чергу (сатурація), або

починає відповідати помилками. Таким чином, загальний рекомендований алгоритм аналізу має вигляд як показано на Рисунку 2.1.1.1.

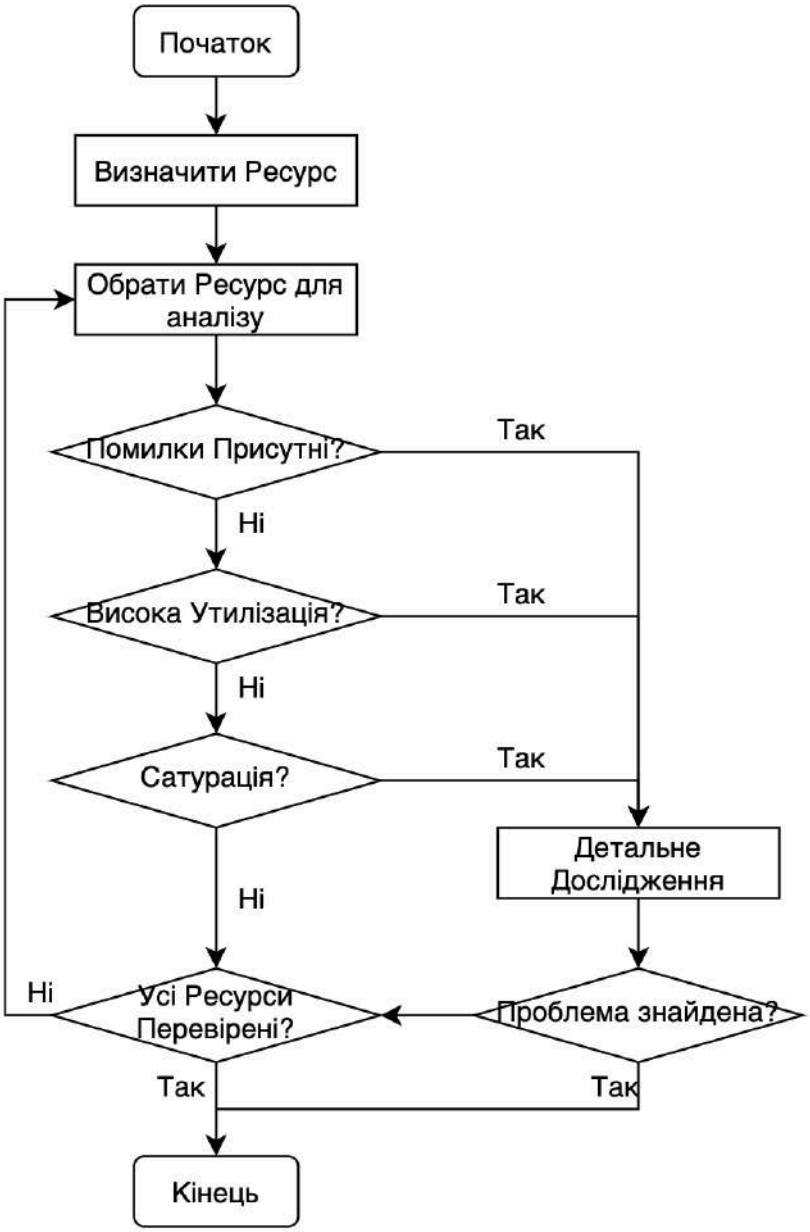


Рисунок 2.1.1.1 – Алгоритм застосування методу USE під час аналізу проблем інфраструктури

У разі виявлення проблеми, як видно з Рисунку 2.1.1.1, помилки системи перевіряються першими. Зазвичай перевірка помилок проходить достатньо швидко і вони можуть вказати на проблему, зекономивши при цьому час на дослідження інших метрик.

Але ще до того як буде відбуватися пошук помилок, необхідно скласти перелік серверних ресурсів які будуть моніторитись. До такого можуть належати [10]:

- **CPUs**: сокети, ядра, потоки (віртуальне CPUs)

- **Основна пам'ять**: RAM
- **Інтерфейси мережі**: порти
- **Засоби збереження інформації**: SSH, HDD
- **Контролери**: пам'ять, мережа
- **Міжкомпонентні з'єднання**: CPUs, пам'ять, I/O

Розглянемо, як відбувається збір згаданих метрик моніторинговою системою Prometheus [33]. Наприклад:

- Утилізація CPU

```
1 - avg(rate(node_cpu{job="default/node-exporter",mode="idle"}[1m]))
```

- Сатурація CPU

```
sum(node_load1{job="default/node-exporter"}) /
sum(node:node_num_cpu:sum)
```

- Утилізація оперативної пам'яті (RAM)

```
1 - sum(
    node_memory_MemFree{job="…"} +
    node_memory_Cached{job="…"} +
    node_memory_Buffers{job="…"} )
/ sum(node_memory_MemTotal{job="…"})
```

- Сатурація оперативної пам'яті (RAM)

```
1e3 * sum(
    rate(node_vmstat_pgpgin{job="…"}[1m]) +
    rate(node_vmstat_pgpgout{job="…"}[1m])) )
```

Загалом, при виборі ресурсів необхідно керуватися запитанням: «що може призвести до проблеми?». Але якщо немає впевненості чи додавати той чи інший ресурс і його метрику до моніторингу, то краще додати і потім уже проаналізувати її цінність.

В результаті, використовуючи підхід USE, ми отримуємо своєрідний чек-лист, який дозволяє контролювати ресурси та їх стан, див. Рисунок 2.1.1.2.

Рисунок 2.1.1.2 Чек-лист за методом USE

2.1.2 Чотири Золотих Сигнали Google

Наступним є розроблений компанією Google підхід **Four Golden Signals** (Чотири Золотих Сигнали) [8]. Цей метод фокусується на моніторингу зі сторони кінцевих користувачів (орієнтація на HTTP запити), а не системної інфраструктури як у випадку підходу USE [8]:

- **Latency (затримка)**: час відповіді сервера на запит.
  Важливо відрізняти час відповіді успішного та неуспішного запиту адже останній швидко відповідає з помилкою 500 або іншою подібною, що може призвести до невірної інтерпретації результатів. З іншої сторони, повільна помилка навіть гірше аніж швидка. Тому важливо спостерігати за тривалістю відповіді невдалих запитів окремо.
- **Traffic** (трафік): навантаження на систему.
  Для веб-сервісу такою метрикою є кількість HTTP запитів у секунду (розбитим по запитам). Для аудіо- чи відео-транслюючих систем, такою є рівень I/O мережі чи кількість одночасних сесій.
- **Errors** (помилки): співвідношення невдалих запитів до вдалих зі сторони користувачів.
- **Saturation** (сатурація): насичення системи, або іншими словами наскільки близько система до виснаження своїх ресурсів.

Таким чином, Google рекомендує робити заміри по усім пропонованим сигналам і у випадку виникнення проблем хоча б одному з них, швидко реагувати на ситуацію, адже це може бути сигналом того, що система стала нестабільною.

2.1.3 RED метод

Далі, базуючись на Google **Four Golden Signals** Том Вілкі розробив власний метод але для мікро-сервісної архітектури – RED [33]:

- **Rate** (R, частота): кількість запитів в секунду.
- **Error** (E, помилки): кількість запитів, що повернули користувачу помилку, їх частка серед усіх запитів.
- **Duration** (D, швидкість): час відгуку запитів.

Том Вілкі рекомендує аналізувати ці показники для кожного мікро-сервісу у великій системі, як показано на Рисунку 2.1.3.1 [33].

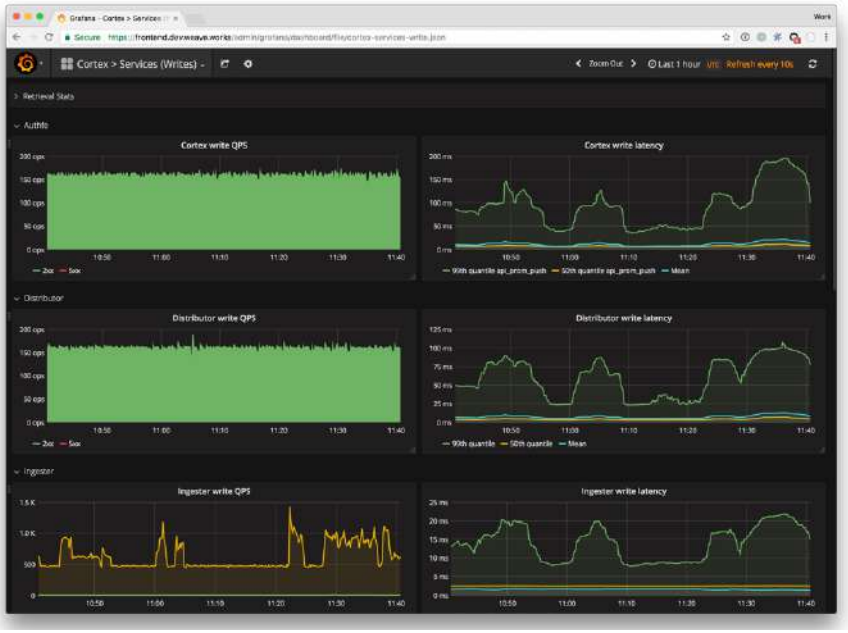


Рисунок 2.1.3.1 – Приклад застосування методу RED для мікросервісної архітектури

Як видно з рисунку вище, кожен рядок відображає моніторинг запитів у секунду, частоту помилок та часу відгуку по кожному окремому мікросервісу.

Розглянемо, як деякі з описаних метрик акумулюються в системі моніторингу Prometheus [33]. Наприклад:

- Rate

```
sum(rate(request_duration_seconds_count{job="…"}[1m]))
```

- Errors

```
sum(rate(request_duration_seconds_count{job="…",
status_code!~"2.."}[1m]))
```

- Duration

```
histogram_quantile(0.99,
sum(rate(request_duration_seconds_bucket{job="…"}[1m])) by (le))
```

Розглянувши різні підходи до моніторингу та пріоритезації вибору метрик, ми можемо з упевненістю заявити, що всі ці методи – USE та RED - потрібно застосовувати у поєднанні. Так RED підхід допомагає «піклуватися про кінцевого споживача та показує на скільки задоволеним він є», тоді як USE підхід допомагає «піклуватися безпосередньо про інфраструктуру та показує на скільки задоволеною є вона». Ці підходи просто описують різні сторони однієї і тієї ж медалі і є взаємодоповнюючими, а не взаємовиключними.

2.2 Існуючий інструментарій телеметрії цифрових систем

Моніторинг продуктивності займає все важливішу частину у бізнес процесах загалом. Доступність та надійність системи має прямий вплив на успіх компанії. Бізнес хоче знати які саме запити відбуваються та наскільки ефективно на це реагує система. Серед одних з найпоширеніших інструментів для моніторингу є такі системи як:

- Prometheus,
- Splunk,
- New Relic,
- ELK stack (Kibana)
- TIG stack (Grafana) та інші.

Складні цифрові системи стикаються з великою кількістю технічних проблем протягом свого життєвого циклу. В результаті час на пошук певної проблеми

може займати від 30 секунд до 30 годин чи навіть більше. Водночас, моніторинг якраз і покликаний, щоб вирішити цю проблему та пришвидшити процес пошуку і виправлення проблем. Але що взагалі моніторинг представляє собою?

Отже, моніторинг – це синтетичний процес збору, аналізу та використання інформації для відслідковування прогресу та стану цифрових продуктів для досягнення поставлених цілей чи прийняття управлінських рішень.

Загалом компанія отримує наступні переваги, реалізовуючи моніторинг на проекті:

- Скорочення часу на пошук проблем системи.
- Скорочення часу на усунення проблеми.
- Зменшення ризики втрати даних.
- Завчасне інформування у разі виникнення проблем та уникнення негативних наслідків.
- Збільшення прибутку через гарантування доступності системи та уникнення її збою.

Загалом існує 2 типи моніторингу згідно Google [28]:

1. White-box (відкритого типу)

   Процес збору та аналізу внутрішніх компонентів системи зсередини.
2. Black-box (закритого типу)

   Процес збору та аналізу даних рівня цілої системи ззовні.

Black-box моніторинг орієнтується більше на симптоматику та відображає поточні – не потенційні – проблеми. White-box в той же час дозволяє побачити причини неминучих проблем, часто масковані самою логікою, наприклад повторними запитами (retries).

Процес телеметрії складається з декількох кроків і для того щоб він був ефективним, має бути реалізованим у наступному порядку:

1. Збір інформації з джерел даних.
2. Отримання даних.
3. Трансформацію даних у необхідний формат.
4. Пошук та аналіз серед даних.

5. Візуалізація даних.

2.2.1 Prometheus

Давайте розглянемо найпопулярніші на ринку сучасні системи телеметрії. Розпочнемо з системи **Prometheus** [55]. Prometheus – це система моніторингу та оповіщення, яка використовується для розподіленої інфраструктури та послуг. Система має відкритий доступ до власного коду та написана мовою програмування Go. Дані від Prometheus можна візуалізувати в Grafana, Splunk, Kibana або будь-якому іншому інструменті візуалізації, сумісному з Prometheus.

Система моніторингу збирає дані з сервісів та хостів шляхом HTTP запитів до моніторингових агентів. Потім ці дані зберігаються в базу даних тимчасових рядів («time-series») для читання та візуалізації, див. архітектуру на Рисунку 2.2.1.1 [23].

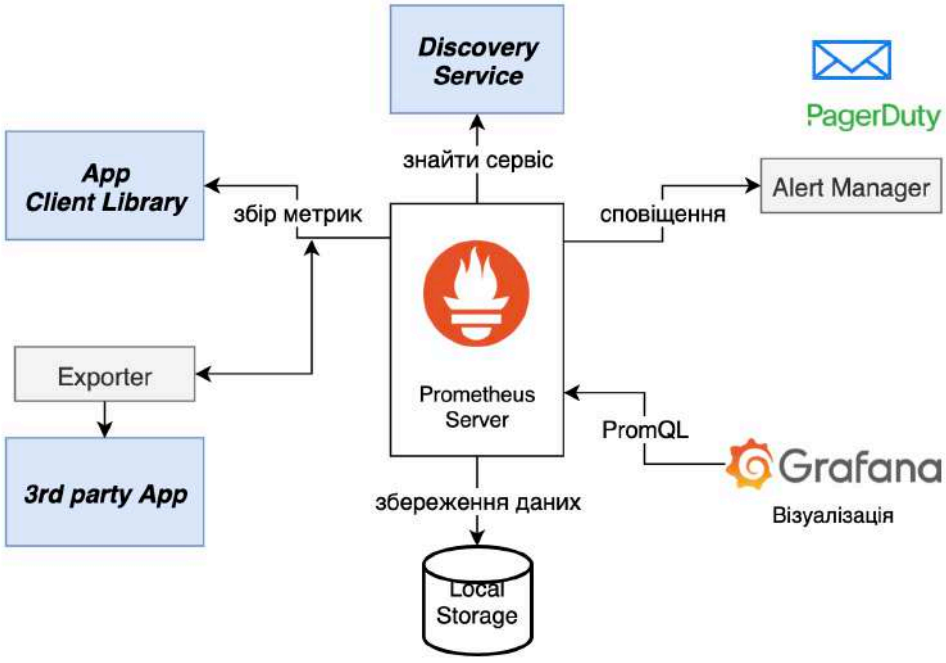


Рисунок 2.2.1.1 – Архітектура системи моніторингу Prometheus

До архітектури системи входять [23]:

- *Prometheus Server* - збирає показники з декількох вузлів і зберігає їх локально.
- *Експортери (Exporters)* - компонент, що використовується для моніторингу програми, до якої ви не маєте прямого доступу або вам не належить. Експортер працює подібно до проксі між додатком та

Prometheus. Він буде отримувати запити від сервера Prometheus, збирати дані з журналів доступу, помилки програми, перетворювати їх у правильний формат і в кінцевому підсумку повертатиметься на сервер Prometheus.

- *Клієнтські бібліотеки (Clients Library)* - профілювання коду додатків / контрольно-вимірювальні прилади.
- *Шлюз (Push Gateway)* - API шлюзу цінний для одноразових завдань, які виконуються, фіксують дані, перетворюють ці дані у формат даних Prometheus та передають ці дані на сервер Prometheus.
- Менеджер сповіщень (Alert manager) – сповіщення у разі виходу показників за встановлені рамки чи аномалій.

Prometheus підтримує чотири типи метрик [16]:

- *Лічильник (counter)* – кумулятивна метрика яка може лише або зростати, або бути анульовано. Використовується для підрахунку кількості виконаних завдань, помилок чи кількості запитів.
- *Вимір (gauge)* – показує актуальне значення метрики на даний момент (місце на диску, CPU/RAM використання).
- *Гістограма* – відображає зібрані метрики у вигляді гістограм, групуючи їх за величиною. Наприклад, час відгуку тієї чи іншої операції може бути згрупований у 3 мс, 5 мс та 10 мс.
- *Висновок (summary)* – подібне відображення до гістограми але додатково показує суму кожної групи, що дає змогу порахувати середнє. Наприклад, якщо було здійснено 5 запитів з часом відгуку 1, 2, 4, 3 та 10 мс. Тоді сума складе 20 мс., кількість – 5, а середній час відгуку – 5 мс.

До переваг Prometheus можна віднести:

- Графіки на яких Prometheus відображає дані є досить інформативними та їх легко читати. Незалежно від того, які саме дані ви шукаєте, вони будуть представлені у найбільш зрозумілий спосіб.
- Оповіщення, які надсилає Prometheus, приходять швидко. Ви завжди будете знати, коли щось важливе відбувається з вашими даними.

- Дані, які Prometheus збирає, та формат в якому він все це представляє, легко та ефективно зберігаються локально на диску комп’ютера.

До недоліків системи Prometheus можна віднести:

- Оскільки ви отримуєте дані від сторонніх експортерів, існує ймовірність того, що ви можете отримувати надлишкові дані, що в результаті може впливати на чистоту графіків.
- Система запитів, яку використовує Prometheus, зазвичай є ефективною але аж ніяк не досконалою. Буде потрібне час, щоб вивчити основні принципи роботи з системою.

2.2.2 NewRelic

Ще однією ефективною та популярною системою моніторингу є **NewRelic** [53]. NewRelic - це гнучкий інструмент моніторингу системи веб-додатків, створений для роботи в режимі реального часу з іншими веб-програмами, що працюють в реальному часі. Він пропонує своїм користувачам безліч продуктів, де кожен із них має певні цілі та вирішує специфічні проблеми.

NewRelic постачається зі своїми агентами. Агент - це невелика частина коду, яка інтегрується всередину веб-програми та перевіряє, що створює код веб-сторінки під час побудови веб-сторінок. Агент здатний вимірювати час відгуку та вихідний час, а також інші показники тривалості. Він інформує про час, необхідний для завантаження певної сторінки, і вказує, чи затримують процес будь-які фактори. Час завантаження веб-сторінки розподіляється між користувачами по всьому світу. Таким чином можна визначити чи довший час завантаження викликаний чимось на сервері, у мережі, коді програми чи браузері.

New Relic - це програмне забезпечення як послуга (Software as a Service), що пропонує контроль доступності та продуктивності. Ключові особливості включають:

- Близько 350+ власних інтеграцій із серверними, інтерфейсними програмами, хмарними платформами.
- Стандартизовані розрахунки та моніторинг метрик.

- Визначення Угоди про рівень обслуговування (SLA).
- Огляд ключових бізнес-операцій.
- Наскрізний реальний моніторинг користувачів.
- Моніторинг внутрішньої та зовнішньої роботи системи.
- Моніторинг надійності та помилок.
- Синтетичні тести.
- Сповіщення.

Загальний вигляд який має інтерфейс системи показаний на Рисунку 2.2.2.1.

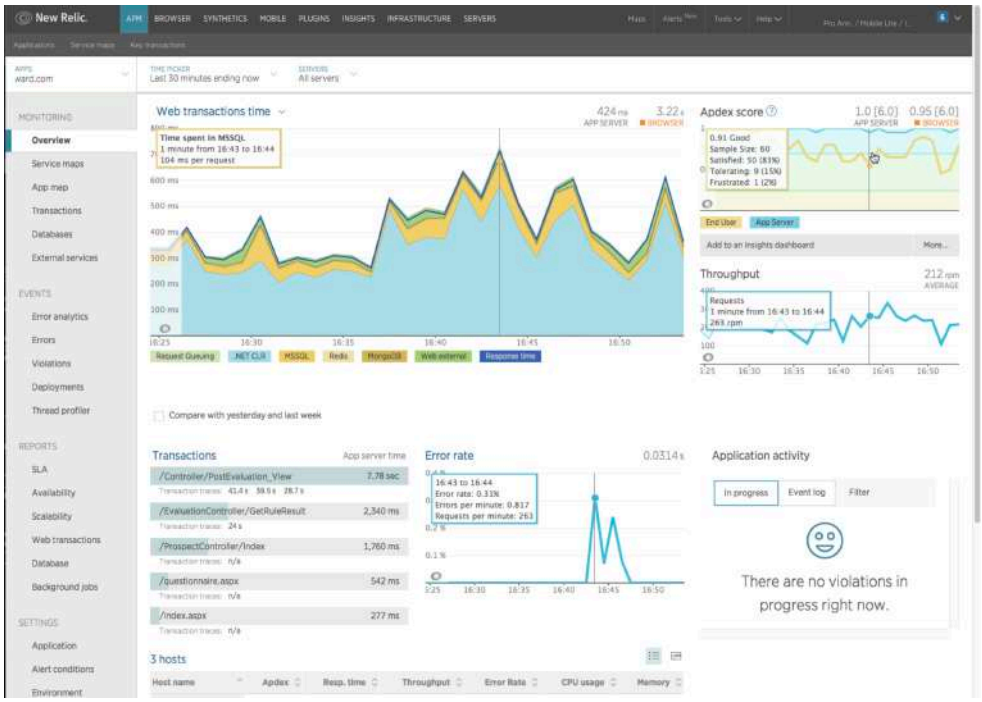


Рисунок 2.2.2.1 – Інтерфейс системи моніторингу NewRelic

Використовуючи Agent, можна надсилати дані з системи яка знаходиться під моніторингом до NewRelic. Агенти надсилають як веб-транзакції, так і не-веб-транзакції. Якщо потрібно, є можливість додати власні атрибути або поля до своїх даних, щоб мати змогу отримати більше статистичних даних та покращити усунення проблем.

Загалом архітектура інтеграції моніторингової системи NewRelic зображена на Рисунку 2.2.2.2 [2].

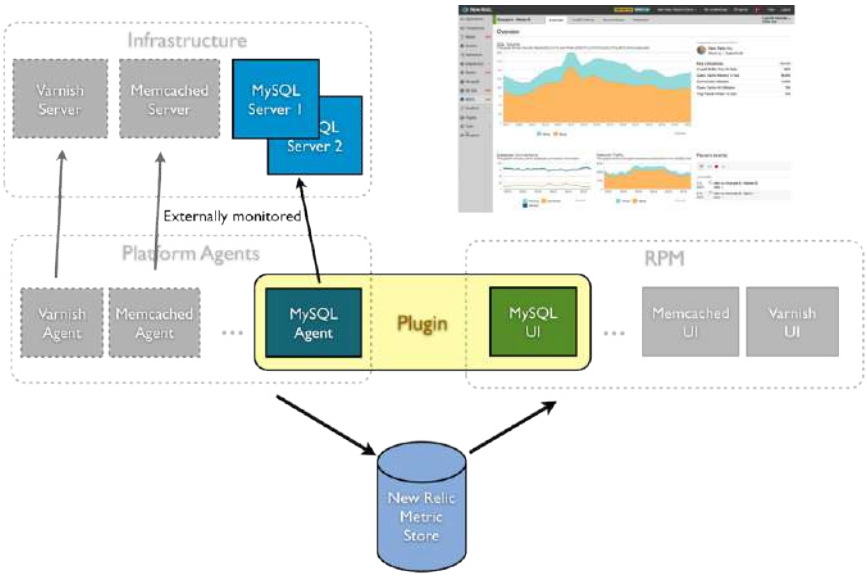


Рисунок 2.2.2.2 – Архітектура інтеграції NewRelic

NewRelic – корисний інструмент для сайтів, які мають значне серверне навантаження під час обробки інформації та складні інфраструктурні рішення. Цей постачальник має гарну візуалізацію даних, і кінцевий користувач його продукту не повинен турбуватися про створення власних інформаційних панелей.

До переваг системи можна віднести:

- Зрозумілий і зручний інтерфейс з безліччю інтерактивних вбудованих інформаційних панелей, звітів, які допомагають виявляти проблеми та швидко їх вирішувати.
- Моніторинг реальних користувачів дає уявлення про те, наскільки ефективно працює сайт для відвідувачів веб-сторінок по всьому світу.
- NewRelic пропонує безкоштовну пробну версію. Оплата базується на кількості встановлених хостів.

До недоліків системи належить:

- Інтерфейс інформативний, але наявність великої кількості опцій може ускладнити орієнтування у ньому.
- NewRelic все ж таки не дозволяє сильно зануритися та проаналізувати проблему, і в результаті не завжди швидко її вирішити.
- Цей інструмент створений для моніторингу великих систем, що означає і високі ціни.

Загалом, NewRelic – це комплексне рішення. Постачальник пропонує безліч опцій, спрямованих на вирішення конкретних проблем. NewRelic забезпечує хорошу аналітику для повільних запитів клієнта та повільних SQL-запитів.

2.2.3 TIG набір технологій

До третього інструменту моніторингу який ми б хотіли розглянути у даній дипломній роботі відноситься набір технологій моніторингу TIG (Telegraf, InfluxDB та Grafana) [29]. Дані технології є, мабуть, одними із найпопулярніших та можуть бути використаний для широкого спектру завдань та ресурсів моніторингу: від операційних систем (таких як метрики продуктивності Linux або Windows), до баз даних (таких як MongoDB або MySQL) та інші технології.

Архітектуру та принцип роботи набору TIG доволі легко зрозуміти та встановити, що і робить його таким популярним (див. Рисунок 2.2.3.1). Також усі інструменти даного набору технологій мають відкритий код та можуть змінюватися в залежності від потреб проекту [29].

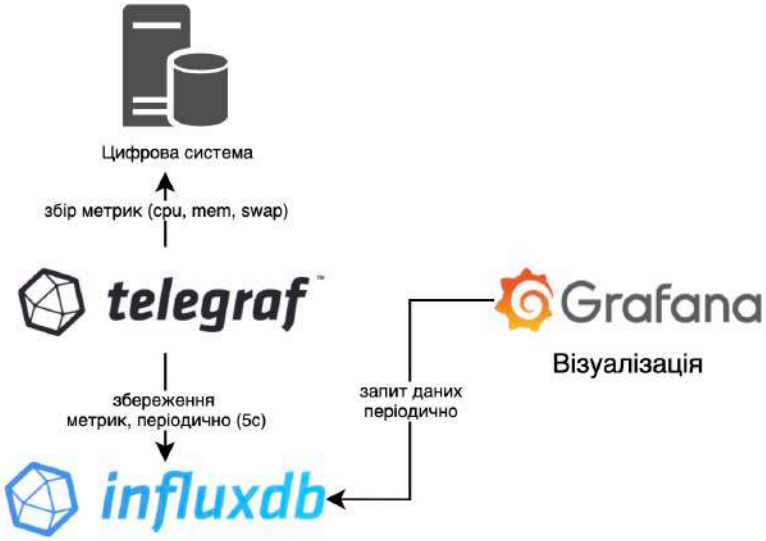


Рисунок 2.2.3.1 – Архітектура моніторингової системи TIG

Telegraf є агентом, відповідальним за збір та агрегування даних та метрик, наприклад таких як поточне використання часу процесора.

InfluxDB – база даних тимчасових рядів і використовується для зберігання метрик в часі, а також їх передачі у сервіс Grafana, який є сучасним рішенням для інформаційних панелей та графіків.

До переваг InfluxDB можна віднести:

- Висока продуктивність бази даних.
- HTTPs інтерфейс для читання та запису інформації.
- SQL-подібний синтаксис.
- Налаштування політик збереження даних.

Також сама система візуалізації Grafana має свої переваги:

- Безкоштовна.
- Система налаштування різного рівня доступу для різних користувачів.
- Можливість роботи з різними джерелами даних.
- Проста в навігації, візуально-приємна та інтуїтивна.
- Вбудована система сповіщень.

Ми розглянули кілька різних систем моніторингу метрик та сповіщення. Такі системи як NewRelic більше призначаються для використання складних багаторівневих систем. Prometheus та TIG набір технологій можуть використовуватись для моніторингу як для складних так і для простих продуктів.

2.3 Методологія оцінки продуктивності браузера

До цього моменту ми найбільше приділяли уваги продуктивності серверної частини. Також, у розділі 1.2 було згадано, що існує такий вид тестування як тестування продуктивності роботи системи у браузері, або іншими словами продуктивність на клієнтській стороні.

Коли ми говоримо про продуктивність на стороні клієнта, це означає, що вона включає всі речі, що стосуються діяльності на стороні кінцевого користувача. Наприклад, продуктивність веб-додатку на стороні клієнта включатиме час виконання сервера та візуалізацію браузера на стороні клієнта, виклики JS / AJAX, відповіді сокетів, сукупність даних служби тощо.

Загалом, оптимізація веб-продуктивності відбувається шляхом моніторингу та аналізу продуктивності програми та визначення шляхів її покращення. У свою чергу, веб-програми - це суміш коду на стороні сервера та клієнта. В результаті,

програма може мати проблеми з продуктивністю на будь-якій із сторін, – сервер чи клієнт - і обидві частини необхідно часто оптимізовувати.

Сторона сервера лежить у площині запитання - скільки часу потрібно для виконання та опрацювання запитів на сервері? Оптимізація продуктивності на сервері, як правило, обертається навколо оптимізації таких речей, як запити до бази даних, залежність від інших систем, алгоритми виконання, кеші, тощо.

Клієнтська сторона стосується продуктивності виконання програми безпосередньо у веб-браузері. Це включає час початкового завантаження сторінки, завантаження всіх ресурсів, JavaScript, який працює в браузері, тощо.

Клієнтський код написаний мовою програмування JavaScript. Код, вбудований в теги <script> або отриманий з віддаленого сервера, виконується, коли парсер браузера потрапляє до цієї частини сторінки. Якщо JavaScript код вбудований у верхню частину сторінки, він буде запущений, коли аналізатор натрапить на нього, що може затримати візуалізацію решти сторінки.

Якщо JavaScript код знаходиться внизу, синтаксичний аналізатор виконує його лише після аналізу та рендерингу решти сторінки. Ось чому так багато розробників навчилися розміщувати свої команди JavaScript всередині функції зворотного виклику "готовий до роботи з документами" ("document-ready"). Таким чином, код виконувався лише після завантаження всієї сторінки.

Під завантаженням JavaScript з віддалених серверів означає, що час, необхідний для відображення сторінки, залежить не лише від швидкості сервера, пропускної здатності мережі та складності сторінки, а й від серверів та мереж, що обслуговують такий JavaScript, а також складності цих сторінок.

Також, завжди є гарною практикою запускати код JavaScript за допомогою “мінімізатора” або “мініфікатора”, який видаляє коментарі, зайві пробіли та все інше, що не потрібно для запуску програм на стороні клієнта. Крім того, комбінуючи та стискаючи файли, ми можемо зменшити розмір JavaScript, який надсилається браузерам користувачів.

Тут ми підійшли до необхідності розглянути основні принципи роботи самого веб-браузера, а саме його механізму опрацювання веб-сторінок. Діаграма знизу

відображає основні компоненти, з яких складається браузер (див. Рисунок 2.3.1). Найважливішими для **шляху візуалізації** (**rendering path**) є Механізм Візуалізації та Мережевий Компонент [1].

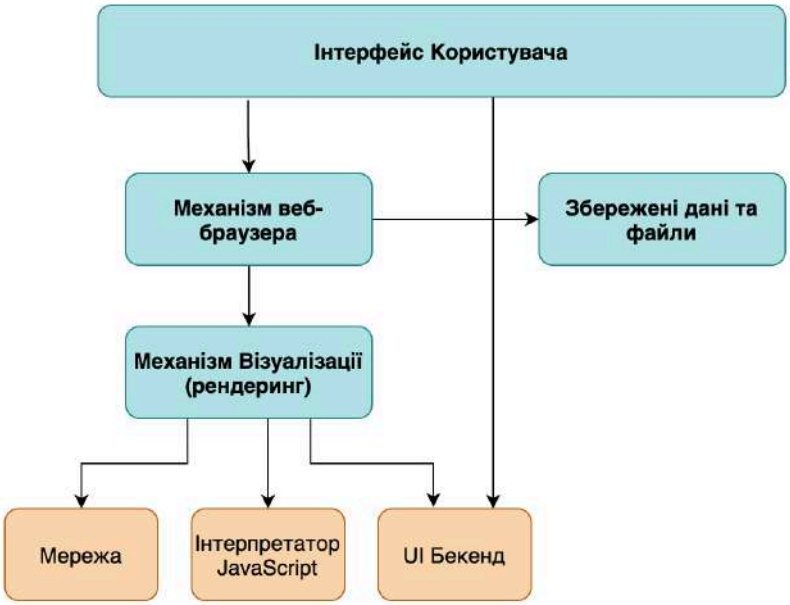


Рисунок 2.3.1 – Основні компоненти веб-браузера

Давайте уявимо, як користувач вводить URL-адресу того чи іншого веб-сайту в адресний рядок свого браузера. Адресний рядок є частиною компонента Інтерфейс Користувача браузера.

Потім браузер використовує свій мережевий компонент для відкриття TCP-з'єднання, а потім робить http-запит на сервер, на якому розміщений веб-сайт. Це робиться шляхом відправки запиту на DNS-сервер, який виконує пошук IP-адреси цієї URL-адреси, а потім надсилає запит на сервер самого сервера веб-сайту. У відповідь сервер надсилає файли, які є складовою веб-сайту, до браузеру користувача.

Як тільки файли надіслані, вступає в дію Механізм Візуалізації (рендерингу). Механізм Візуалізації відповідає за інтерпретацію HTML-документів та зображень, які відформатовані за допомогою CSS, та генерацію макета, який відображається в інтерфейсі користувача.

Отже, як він аналізує / інтерпретує ці файли:

1. Читання (parsing)
    - Побудова DOM

- Побудова CSSOM

2. Візуалізація дерева (render tree) на основі DOM та CSSOM
3. Побудова макету на основі візуалізованого дерева
4. Відображення веб-сторінки

В результаті, критичний шлях візуалізації браузера (critical rendering path) має наступний вигляд [19]:

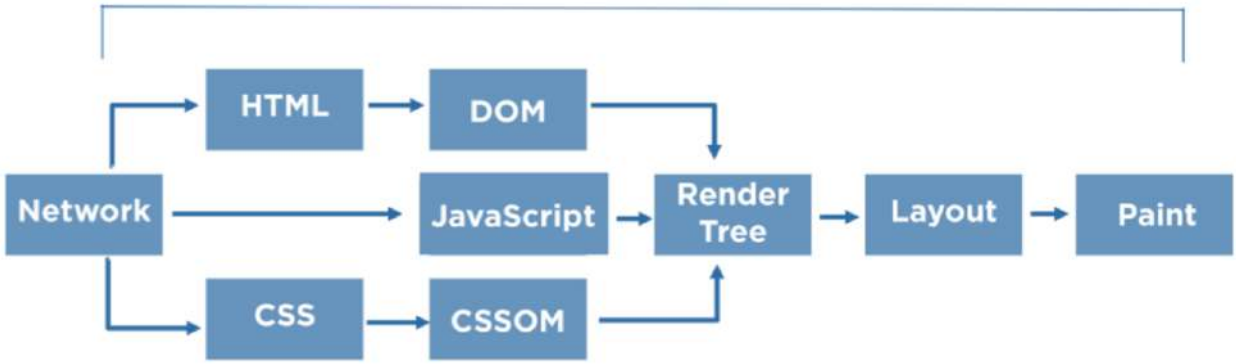


Рисунок 2.3.2 – Критичний шлях візуалізації браузера

Оптимізація критичного шляху візуалізації може допомогти покращити продуктивність веб-сторінки і як результат покращити досвід користування сторінкою. Ба більше, це найбільше впливає також і на ефективність пошуку в Google. Але для того, щоб оптимізувати, необхідно його спочатку виміряти та знайти вузькі місця.

До важливих показників тестування продуктивності на стороні клієнта відносяться наступні [21]:

- ***Перша реакція*** - час до першого байта (Time to First Byte - TTFB):

  Час від початку початкової навігації до отримання першого байта базової сторінки браузером (після наступних переадресацій).

- ***Поява вмісту*** - First Paint:

  Час від початку початкової навігації до того, як перший не пустий вміст буде намальований на дисплеї браузера.

- ***Поява першого змістовного вмісту*** - First Contentful Paint:

  Перший змістовний контент показаний, коли будь-який вміст (об’єкт, визначений в об’єктній моделі документа) візуалізується. Це може бути текст, зображення або візуалізація полотна.

- ***Час готовності до інтерактивності*** - Time to Interactive:

  Кількість секунд з моменту запуску навігації до стабілізації макета, видимих веб-шрифтів і сторінки, що реагує на дії користувача. Сторінка стабілізується, якщо не було блокування принаймні 50 мс.

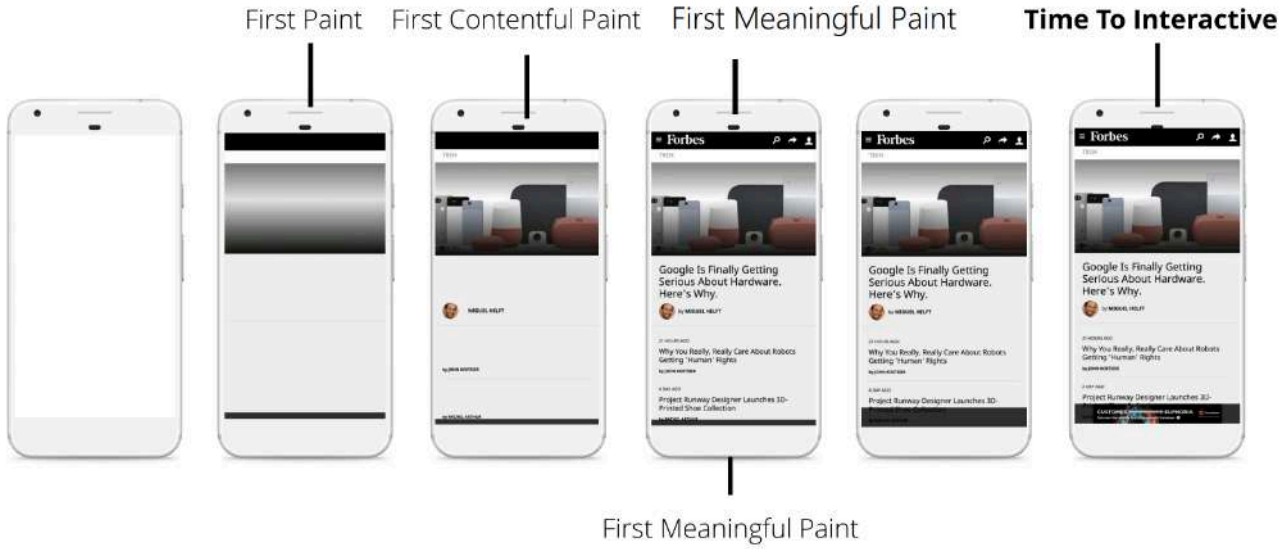
- ***Завершено обробку сторінки*** - вміст DOM готовий (DOM Content Ready End)

  Час, коли синтаксичний аналізатор HTML дійшов до кінця документа, що означає, що він виконав усі сценарії блокування. CSSOM може бути поки ще не повним.

- ***Уся сторінка завантажена*** - час завершення обробки документа

  Подія, яка спрацьовує, коли всі сценарії, CSS та зображення, вказані в HTML, закінчують своє завантаження.

Як ці метрики відображені у браузері мобільного телефону під час завантаження веб-сторінки, дивіться на Рисунку 2.3.3 [21].



2.3.3 – Основні метрики продуктивності браузера

Загалом, подібних метрик може бути десятки, а то і більше. Для прикладу, розглянемо метрики на Рисунку 2.3.4 [11].

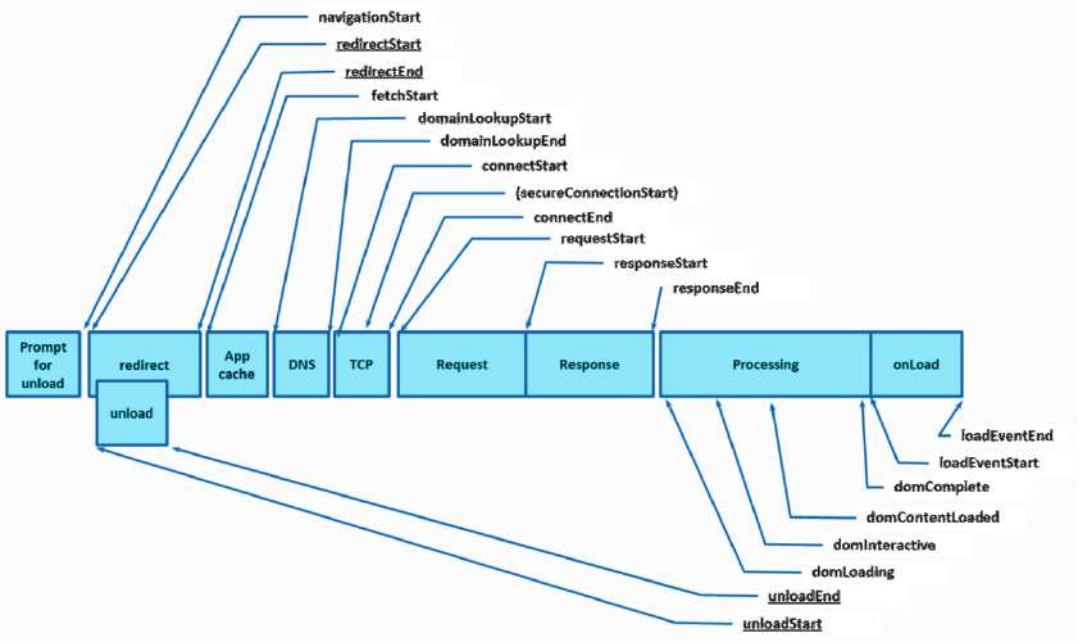


Рисунок 2.3.4 – Метрики продуктивності завантаження веб-сторінки браузером

Також, ці метрики можна згрупувати відносно досвіду користувача, а отже рівня його задоволення, див Таблиця 2.3.5 [25].

Таблиця 2.3.5 – Метрики продуктивності браузера відносно досвіду користувача

Досвід користувача	Метрика
Це відбулося?	First Paint (FP), First Contentful Paint (FCP)
Це корисно?	First Meaningful Paint (FMP), Speed Index (SI)
Це може бути використано?	Time to Interactive (TTI)

Також, останнім часом додатково виокремлюють наступні метрики [25]:

- ***Найбільш змістовний вміст*** (Largest Contentful Paint - LCP).

  Вимірює, коли найбільший елемент вмісту стає видимим у вікні перегляду.

  Найбільший елемент, наприклад, великий абзац тексту в статті чи зображення на сторінці товару, ймовірно, є найбільш корисним для розуміння змісту сторінки. Ретельне тестування довело, що LCP є хорошим наближенням, коли завантажується основний вміст сторінки (див Рисунок 2.3.6).

Рисунок 2.3.6 – Демонстрація метрики - Найбільш змістовний вміст (Largest Contentful Paint)

- ***Загальний час блокування*** (Total Blocking Time - TBT).

  Описує діяльність основного потоку JavaScript. Це корисно для розуміння того, як довго сторінка не може реагувати на дії користувача під час завантаження.

  Загальний час блокування сторінки - це сума часу блокування всіх довгих завдань, які відбулись між *Появою першого змістовного вмісту* та *Часом готовності до інтерактивності* (див. Рисунок 2.3.7).

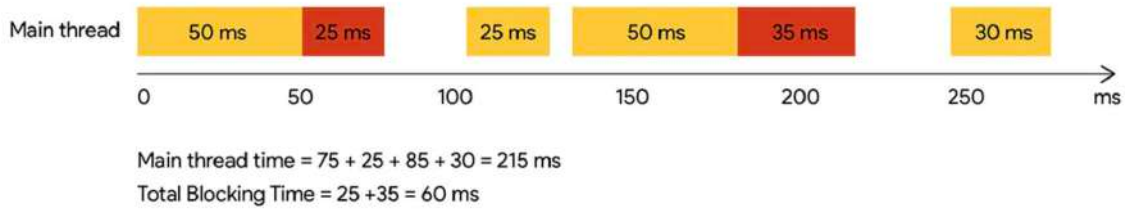


Рисунок 2.3.7 – Приклад підрахунку метрики - ***Загальний час блокування***

2.4 Методології оцінювання задоволеності користувача продуктивністю веб-сторінки

Існує декілька підходів щодо методології оцінювання задоволеності користувача продуктивністю системи та власне самої швидкості системи. Одним із широковживаних таких методів є показник Apdex (Application Performance Index) – це відкритий стандарт, який був розроблений альянсом компаній для

стандартизації підходів для оцінки, звітування та встановлення границь продуктивності системи [4].

Даний індекс розділяє продуктивність системи та рівень задоволення користувача на 3 наступні рівні:

- ***Задоволений*** – кінцевий користувач використовує систему без затримок і задоволений її роботою. Час відгуку нижчий за встановлене граничне значення T (у мілісекундах) для даної системи.
- ***Толерантний*** – користувач помічає затримки, що перевищують встановлений граничний час відгуку T, але згоден їх толерувати.
- ***Розчарований*** – час відгуку перевищує другу границю F (яка рівна 4T), тому користувач має високий рівень розчарування та може перестати використовувати систему взагалі.

Візуально дана модель має наступний вигляд [4].

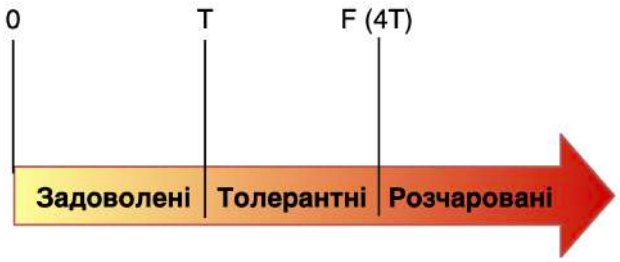


Рисунок – 2.4.1 Рівні реакції користувача на продуктивність система

Формула Apdex – сума кількості задоволених користувачів та половини тих хто толерує розділена на кількість усіх користувачі [4]:

$$Index\ Adpex = \frac{N \text{ задоволених} + \frac{N \text{ толерантні}}{2} + N \text{ розчаровних} * 0}{N \text{ усі користувачі}} \qquad (2.4.2)$$

Для прикладу припустимо, що границя T була встановлена на рівні 3 секунди, тоді F дорівнюватиме 12с. (4 * 3). Нехай у нас було 100 користувачів, 60 з яких задоволені, 20 толерують та інші розчаровані. В результаті індекс Apdex складе:

$$Index\ Adpex = \frac{60 + \frac{20}{2} + 20 * 0}{100} = \frac{70}{100} = 0.7, \text{або } 70\%$$

Як бачимо, Індекс Apdex може коливатися в межах від 0 до 1 (тобто 100%). Логічним є те, що чим він вище, тим краще. До того ж, можна встановлювати

мінімальне допустиме його значення для системи, наприклад, 75% і вище вважається задовільним результатом. В результаті буде корисним порівняння цього показника протягом усього життєвого циклу продукту. Саме так ми зможемо побачити покращення чи навпаки погіршення у продуктивності системи.

Даний індекс уже інтегрований широковживаними інструментами для навантажувального тестування чи моніторингу систем. Першим прикладом є інструмент JMeter, який додав даний індекс до своїх стандартних звітів, див. Рисунок 2.4.3. Інструмент дає можливість конфігурувати значення T та F [45].

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.957	2 sec	8 sec	Total
0.984	2 sec	8 sec	AjoutPanier
0.984	2 sec	8 sec	autoComplete
0.999	2 sec	8 sec	autoComplete-1
0.999	2 sec	8 sec	autoComplete-2
1.000	2 sec	8 sec	autoCompleteStore
1.000	2 sec	8 sec	choixMagasin
0.994	2 sec	8 sec	ClickAnneeTicket
1.000	2 sec	8 sec	clickCours
0.946	2 sec	8 sec	ClickDetailTicket

Рисунок 2.4.3 – Приклад використання Apdex Index в JMeter звіті

Також, така система моніторингу NewRelic використовує та дозволяє налаштовувати границі для описаного індексу [9].

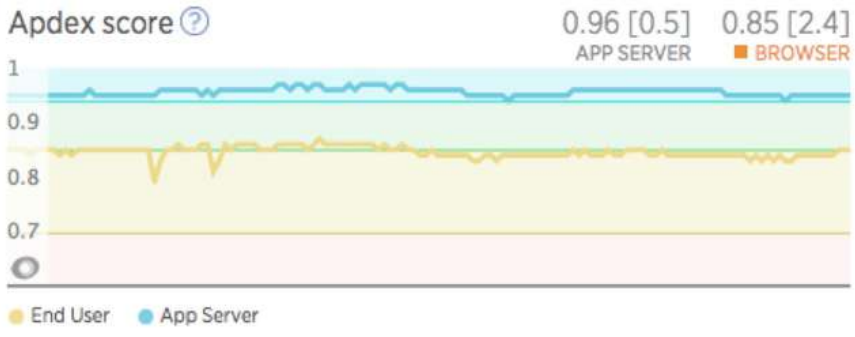


Рисунок 2.4.4 – Приклад використання Apdex Score в NewRelic

Після опису індексу виникає запитання, а яке саме значення використовувати для оцінки Adpex. Це має бути середній час відгуку сервера чи певний його персентиль, наприклад 90й чи 95й? Це взагалі має бути час відгуку сервера чи також враховувати і час на рендеринг сторінки? Доволі логічним є заключити,

що сюди має входити як час відповіді сервера, так і час візуалізації в браузері. Тоді наступне запитання, коли можна рахувати що візуалізація закінчилась на динамічних сторінках де контент постійно змінюється? Відповідь на це запитання можна знайти у іншому індексі, так званому Speed Index. Цей індекс використовується у таких інструменті як Sitespeed.io (був обраний для використання у даній дипломній роботі) і був розроблений Пет Меєном для власного інструменту WebPageTest [31].

Speed Index - є середнім часом (у мс) за який відображається перший візуальний та зрозумілий контент сторінки. Система оцінює прогрес завантаження сторінки в часі у відсотках, див. Рисунок 2.4.5 [31].

Рисунок 2.4.5 - Прогрес завантаження сторінки у часі

Потім будується графік прогресу та вираховується площа над ним, яка в свою чергу коригується на відсоток повного завантаження сторінки, див. Рисунок 2.4.6 [31].

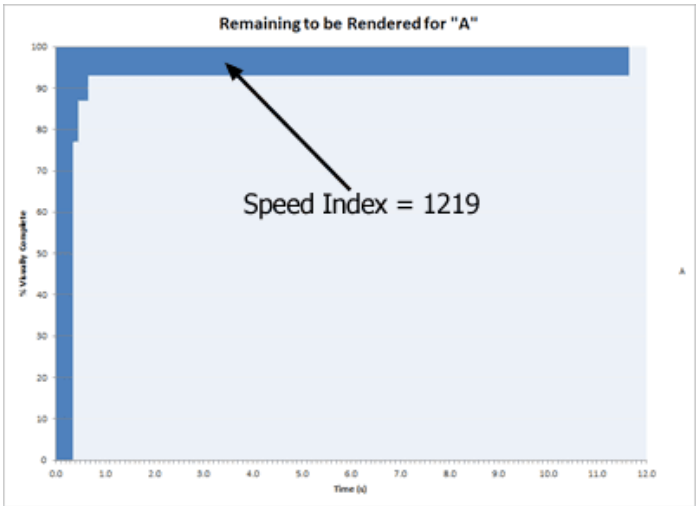


Рисунок 2.4.6 – Прогрес завантаження сторінки та обчислення її Speed Index

Повна формула розрахунку Speed Index має наступний вигляд [31]:

$$Speed\ Index = \int_0^{end} 1 - \frac{VC}{100}, \text{ де} \quad (2.4.7)$$

end – закінчення часу в мілісекундах

VC – % візуально-закінченого завантаження

У висновку можна зазначити, що поєднання двох індексів – Adpex Index та Speed Index – дає можливість для здійснення досить об’єктивної та точної оцінки рівня задоволення користувача продуктивністю системи, використовуючи прозорі та чіткі критерії.

РОЗДІЛ 3: Розробка методології та архітектури системи для високо-навантажувального тестування

3.1 Архітектура системи для здійснення розподіленого навантажувального тестування

Для того, щоб почати розробку дистрибутивної складної системи для здійснення високого навантаження, розглянемо спочатку як зазвичай та найчастіше здійснюється навантаження, див. Рисунок 3.1.1.

Рисунок 3.1.1 – Процес здійснення базових навантажувальних тестів

Як бачимо з рисунку вище, для здійснення базового тесту достатньо локального Персонального Комп’ютера чи навіть ноутбука та одного з інструментів для навантажувального тестування (розглянутих в Розділі 1.3 даної дипломної роботи). Разом з тим, такий підхід дозволяє генерувати відносно невелике навантаження, від 50 до максимум близько 1000 віртуальних користувачів. При чому чим складніший сценарій самого тесту, тим менше користувачів можна згенерувати з однієї машини, адже самі інструменти для навантаження «жадібні» до оперативної пам’яті машин.

Ситуацію можна значно покращити, використавши досить сильний та великий сервер чи взагалі віртуальну машину одного з хмарних провайдерів (AWS, Azure, GCP). Так, це дасть змогу здійснити уже значно більше навантаження, але знову таки воно буде лімітоване однієї машиною. Рано чи пізно, цієї машини може знову стати недостатньо. До того ж, такий підхід має наступні недоліки:

- Генерація навантаження відбуватиметься з однієї і тієї ж IP адреси, що далеко від симуляції реальних користувачів.

- Генерація також відбуватиметься з однієї і тієї ж географічної локації, що часто не є реальною ситуацією для популярних у всьому світі сервісів.
- Сам підхід не є гнучким, адже кожен раз коли нас обмежуватиме своїми ресурсами машина, нам необхідно буде її збільшувати.

Саме тому дана дипломна робота покликана побудувати методологічний підхід та архітектуру системи для здійснення розподіленого навантаження на високо-навантаженні сервіси.

Необхідно також додати, що до таких високо-навантажених сервісів може відноситись будь-яка цифрова система поширена, або зі зростаючою популярністю, у всьому світі. Продукт має бути готовим до великого навантаження у разі його успіху, адже у нього стрімко зростатиме клієнтська база. Більше того, на навантаження цифрових продуктів впливають різного роду маркетингові компанії, спортивні події, сезонність та інші фактори в залежності від сфери у якій представлений сам продукт. Наприклад, чи багато людей чуло про таку соціальну мережу як Orkut від Google? Ця мережа з’явилася швидше за Facebook, але втратила право на продовження свого функціонування. Однією з причин було те, що система не справилась з великим навантаженням та була повільною [26]. Саме для уникнення таких ситуацій, і необхідно проводити розподілене та масштабне навантажувальне тестування. Зазвичай, до сфер де таке тестування особливо важливе та актуальне належать:

- Електронні магазини.
- Соціальні мережі.
- Новинні ресурси.
- Банківські та фінансові ресурси.
- Державні сервіси.
- Сервіси для перегляду різних, у тому числі, спортивних подій.
- Відео та аудіо-контент провайдери,
- IOT системи та інші.

Отже, ми будемо будувати розподілену систему яка б дала змогу здійснити високе навантаження. Високорівнева архітектура такої системи має вигляд як зображено на Рисунку 3.1.2.

Рисунок 3.1.2 – Високорівнева архітектура розподіленого навантаження

Визначимо очікувані вимоги до такої системи:

1. *Можливість здійснювати навантаження практично необмеженого розміру.*

   Система має дати змогу здійснити навантаження 10, 50, 100 чи навіть більше одночасних віртуальних користувачів.

2. *Можливість здійснювати навантаження з різних географічних регіонів одночасно.*

   Це важливо для ресурсів, що використовуються не в одній країні, а з різних.

3. *Можливість динамічно збільшувати чи зменшувати навантаження під час здійснення самого тесту.*

   Це дасть змогу швидко реагувати на недостатнє чи навпаки велике навантаження. Зазвичай, в таких випадках тест зупиняється та перезапускається з оновленою конфігурацією навантаження, що призводить до значних втрат часу, особливо, якщо ми говоримо про довгі тести (2-8 годин). Система має давати змогу уникати подібних ситуацій.

4. *Централізоване збереження та відображення результатів.*

   Загалом, розгорнути дистрибутивну систему не так уже важко. Сама більша проблема в цьому – це якраз централізоване відображення результатів з усіх агентів (машин) навантаження. Часто, безкоштовні інструменти обмежують у цьому (як Gatling), мотивуючи до використання платних ліцензій. Тому необхідно розробити підхід який дасть змогу зберігати та відображати результати розподілених тестів централізовано, використовуючи при цьому інструменти з відкритим доступом до коду (open-source).

5. *Можливість відслідковування результатів тестів у реальному часі.*

   Такий підхід дасть можливість оперативно реагувати на результати тестів та змінювати стратегію навантаження у реальному часі, до повної зупинки тесту у випадку відмови сервісу під навантаженням.

6. *Можливість систематичного та автоматизованого запуску тестів.*

   Система має давати можливість інтегруватися з різними CI/CD системами, такими як Jenkins, TeamCity, для вбудови тестів в безперервну доставку коду.

7. *Система має бути недорогою в обслуговуванні.*

   Уся інфраструктура системи має динамічно збільшуватись чи зменшуватись до повної зупинки в залежності від потреби у самій системі навантаження. Таким чином буде виправдане її використання на фоні уже присутніх на ринку сервісів (Blazemeter, Loadrunner, Gatling Frontline). Також з цієї вимоги випливає, що усі, або принаймні більшість, складових системи мають мати відкритий доступ до коду, або іншими словами «open-source» інструменти.

8. *Підтримка версій коду за принципом – Tests as a Service.*

   Система має надавати змогу бути повністю інтегрованою в контроль та підтримку версій проекту для відслідковування змін та збереження історії.

Враховуючи всі перераховані вище вимоги, ми можемо рекомендувати для побудови даної системи інструменти вказані у Таблиці 3.1.3.

Таблиця 3.1.3 – Рекомендований набір технологій для високо-навантажувальної системи

Інструмент	**Ціль**	**Тип оплати**	**Покриття вимоги**
Хмарний провайдер (AWS, Azure, GCP)	Хостинг та інші сервіси (реєстр, сховище)	Оплата за використані ресурси	#1, #2, #4
Kubernetes	Система для автоматизованого деплойменту, масштабування та менеджменту контейнеризованих систем	Open source	#3, #4, #7
Docker	Сервіс для віртуалізації систем	Open source	#3, #7
Gatling	Скриптинг та лоад-ранер	Open source	#7, #8
Logstash	Перетворення результатів для централізованого збереження	Open source	#4, #7
InfluxDB	База даних тимчасових рядів для збереження аналітичних даних	Open source	#4, #5, #7
Grafana	Візуалізація результатів	Open source	#5, #7
Telegraf	Агент для збору інфраструктурних метрик	Open source	#7
Jenkins	Інструмент для збору та запуску тестів	Open source	#6, #7
GitHub	Система контролю версій	Безкоштовно	#8, #7

Впроваджуючи представлений у таблиці набір технологій, ми покриваємо і виконуємо усі виставлені раніше вимоги. Єдиним джерелом затрат для побудови такої системи буде хостинг у одного з хмарних провайдерів, усі інші інструменти або безкоштовні, або з відкритим кодом доступу, тобто теж безкоштовні.

Опишемо ключові інструменти та аргументуємо їх вибір.

Хмарний провайдер дозволить нам динамічно планувати та будувати інфраструктуру для навантажувальних тестів. Залежно від потреб, ми можемо розгорнути від 1 до практично безкінечної кількості віртуального сервісу. Ба більше, ми можемо розгорнути їх у різних куточках планети, симулюючи навантажувальні тести з різних куточків планети.

Наступною запропонованою технологією є Kubernetes [47] та Docker [37]. Kubernetes допоможе нам з менеджментом та оркеструванням віртуалізованих агентів з навантаження. До переваг Kubernetes, які допомагають нам задовольнити поставлені вимоги, можна віднести [5]:

- *Висока доступність інфраструктури.*

  У випадку якщо один з агентів навантаження зупиниться по тій чи іншій причина Kubernetes сам запустить новий, таким чином зберігши рівень симулюючого навантаження без потреби перезапуску усього тесту.
- *Масштабованість та реплікація.*

  Kubernetes дозволяє додати чи забрати непотрібний под з тим чи іншим контейнером. Це дає нам змогу реалізувати вимогу #3, яка говорить, що система має давати можливість збільшувати чи зменшувати навантаження динамічно.
- *Централізована оркестровка та менеджмент репліками.*

Gatling був обраний як інструмент для написання тестів та здійснення безпосередньо навантаження, адже він дає змогу підійти до тестів як сервісу, інтегруючи сам навантажувальний проект в систему контролю версій кодом, наприклад GitHub. Мова, яка використовується в Gatling – це Scala, тобто на базі JVM, що також дає таку перевагу як багато-платформність. Тобто тести можна запускати практичною з будь-якої операційної системи. До того ж, Gatling дозволяє здійснити велике навантаження, використовуючи лише один агент, завдяки технології Akka [35], що дасть нам змогу використовувати менше машин чим, наприклад, у випадку з JMeter, який є «жадібним» до оперативної пам’яті.

У Gatling є одне але досить суттєве обмеження – хоча інструмент і має відкритий доступ до коду, тобто є безкоштовним, він не дозволяє здійснити розподілене навантаження, мотивуючи його користувачів використовувати платну версію Gatling Frontline [39]. Саме тут на допомогу нам приходить продукт з ELK набору технологій, а саме Logstash [52]. Останній допоможе нам трансформувати дані тестів з кожного агенту таким чином, щоб обійти обмеження Gatling та зберегти їх централізовано в базу даних тимчасових рядів.

Щодо бази даних, то одним з чудових рішень тут буде використання бази даних з тимчасовими рядами InfluxDB. Ця база якраз оптимізована для швидкого читання та запису аналітичних даних за часом, якими і є результати навантажувальних тестів [13].

База InfluxDB чудово інтегрується з сервісом візуалізації даних Grafana. Grafana - це багато-платформний веб-додаток для аналітики та інтерактивної візуалізації з можливістю побудови різного роду графіків та системи оповіщення. До того ж, у цей набір технологій чудово також інтегрується такий інструмент як Telegraf, для збору інфраструктурних метрик. Більше про TIG (Telegraf, InfluxDB, Grafana) було описано в Розділі 2.2.3 даної дипломної роботи. Такий підхід дасть змогу візуалізувати одночасно результати навантажувальних тестів та метрик системи під навантаженням в одному місці – Grafana, – що є зручно для аналізу результатів та пошуку взаємозв’язків.

Запропонована архітектура системи для здійснення розподіленого навантаження, на прикладі хмарного провайдера AWS та описаних вище технологій, зображена на Рисунку 3.1.4.

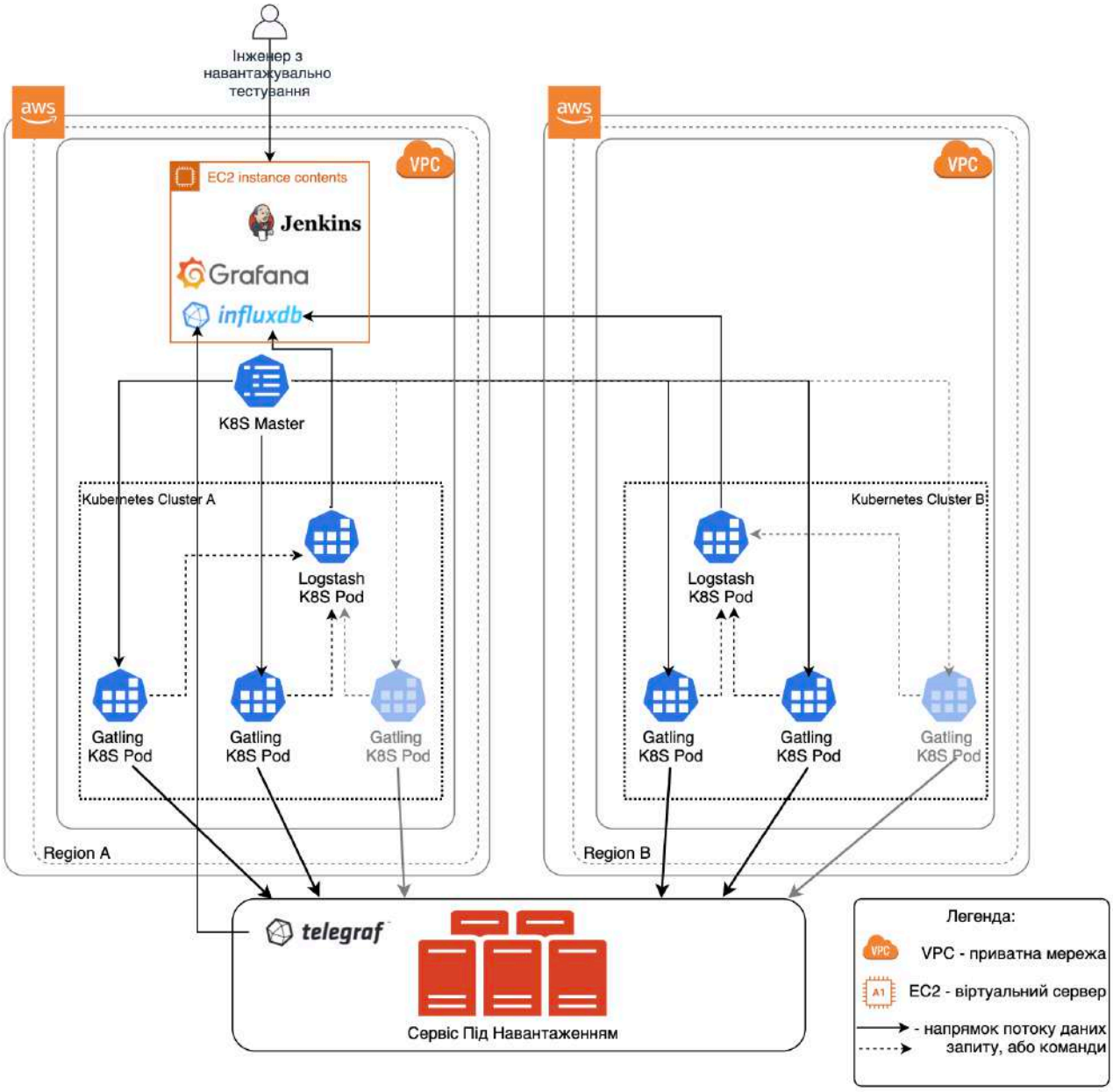


Рисунок 3.1.4 - Архітектура системи для здійснення розподіленого навантаження на прикладі хмарного провайдера AWS

3.2 Архітектура для тестування продуктивності у браузері

Тестування продуктивності у браузері кардинально відрізняється від тестування навантаження не сервер, як інструментально, так і з точки зору підходів, метрик та процесу аналізу результатів. Хоча, що перший вид тестування, що другий переслідують одні і ті ж цілі:

- Пошук вузьких місць продуктивності та їх подальше виправлення.
- Виявлення границь продуктивності системи.

- Попередження негативних наслідків від незадовільної продуктивності системи.
- Прагнення зробити взаємодію користувача з системною приємною та такою, що спонукає його до повернення на веб-ресурс.

Саме тому ми пропонуємо не забувати про тестування продуктивності клієнта та обов'язково його включати в загальну стратегію з тестування навантаження на тому чи іншому проекті.

Загалом тестування продуктивності завантаження сайту у веб-браузері може мати декілька підходів:

1. Тестування коли сервер не навантажений.
2. Тестування коли на сервер здійснюється середнє навантаження.
3. Тестування коли сервер на границі своїх можливостей.
4. Тестування користувачами з вимкненою функцію зберігання кеша. Таким чином ми симулюємо ситуацію немов щоразу сайт відвідує новий користувач.
5. Тестування користувачами з увімкненою функцією зберігання кеша. Таким чином ми немов симулюємо, що до нас повертають користувачі, у яких кеш браузера уже містить різні статичні ресурси нашого сайту (картинки, стилі, скріпти, і т.д.).

Рекомендовано проводити усі згадані вище підходи для тестування та порівнювати їх результати між собою. Це дасть змогу побачити потенційні проблеми як на бекенді, так і на фронтенді. До того ж, слід систематично повторювати таке тестування – після великої зміни коду – та порівнювати результати до і після, щоб побачити чи позитивно (або навпаки негативно) вплинули зміни в системі.

Для тестування продуктивності клієнта ми рекомендуємо використовувати такий інструмент як Sitespeed.io [56]. Він збирає усі необхідні метрики продуктивності та представлений у контейнерому вигляді (Docker), що значно спрощує його використання. Приклад зібраних замірів даним інструментом можна побачити на Рисунку 3.2.1 [56].

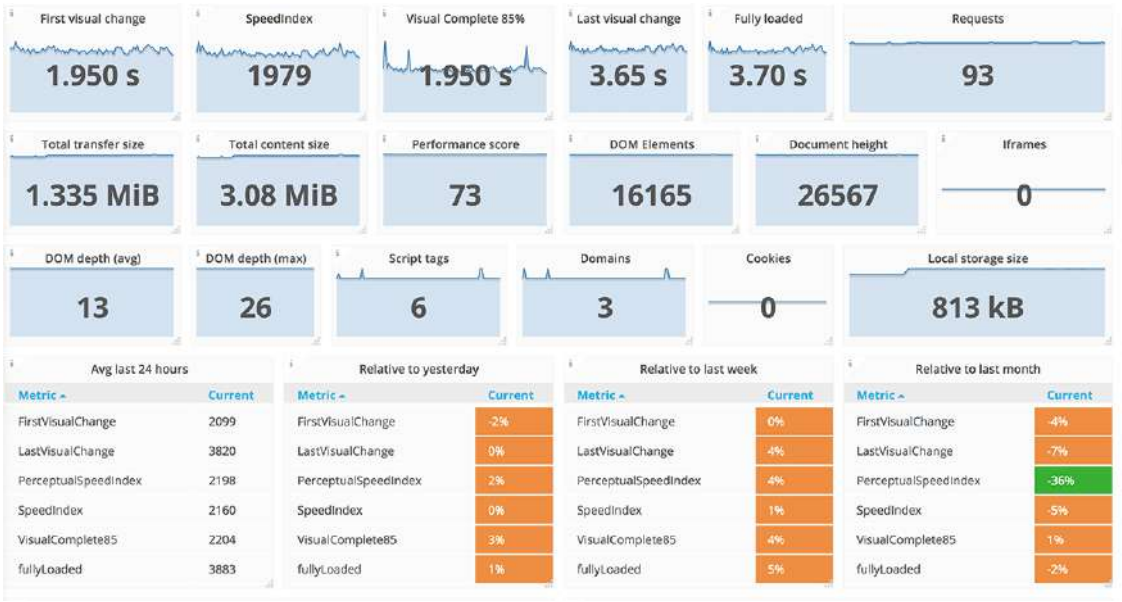


Рисунок 3.2.1 – Приклад метрик від Sitespeed.io

Також цей інструмент може давати рекомендації того, що може бути покращено та оптимізовано, через так званого Тренера (Coach), див. Рисунок 3.2.2 [56].

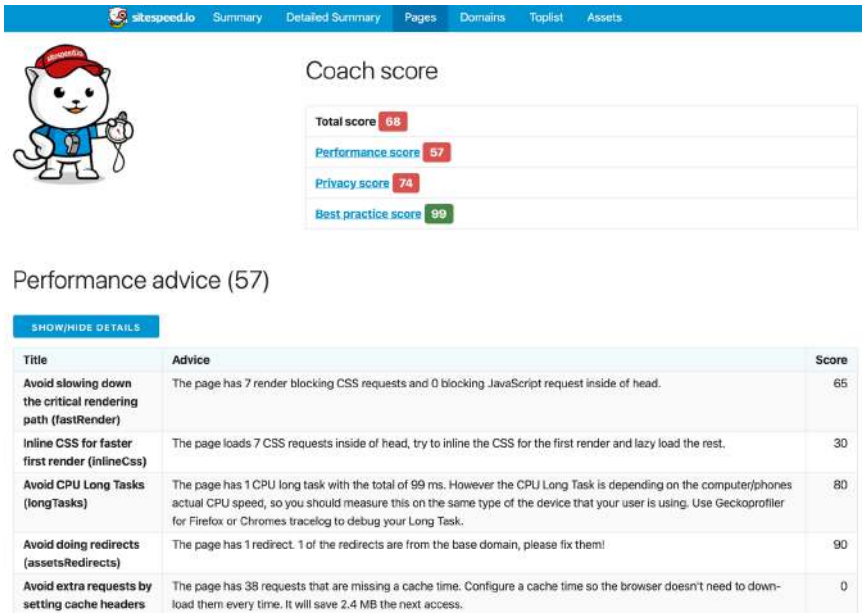


Рисунок 3.2.2 – Функціонал з рекомендаціями від Sitespeed.io

Sitespeed.io здійснює відео-запис кожного свого тесту, що дає змогу подивитись в повторі та у сповільненому режимі на процес завантаження сайту, тобто оцінити критичний шлях візуалізації веб-сайту (детально описаний в Розділі 2.3).

Sitespeed.io також вміє зберігати дані в базу даних з тимчасовими рядами Graphite [42]. В свою чергу, можна налаштувати Grafana читати дані з бази даних

Graphite та побудувати окрему Панель з результатами тестування продуктивності. Архітектура такого рішення зображена на Рисунку 3.2.3.

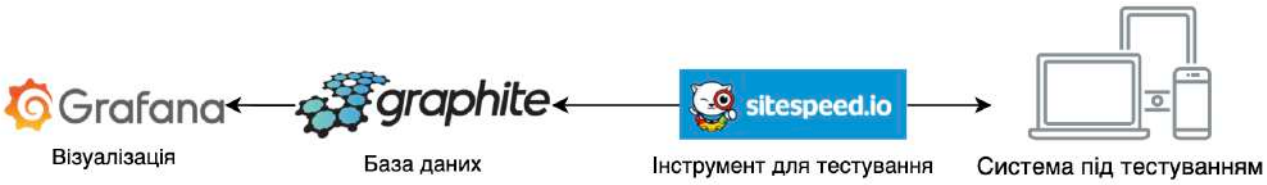


Рисунок 3.2.3 – Архітектура інтеграції Sitespeed.io та Grafana

Sitespeed.io надає для використання уже побудовані у форматі json панелі для відображення метрик тестування [57]. Серед них ми можемо рекомендувати до використання наступні панелі:

- Page metrics - показує показники для певної URL-адреси / сторінки, протестованої на настільних комп'ютерах, та метрики з емуляцією мобільних пристроїв (див. Рисунок 3.2.4).
- The leaderboard – для порівняння різних веб-сайтів або URL-адрес.
- Chrome User Experience Report - показники, які браузер Chrome збирає від реальних користувачів.

Рисунок 3.2.4 – Приклад Панелі Page metrics від Sitespeed.io

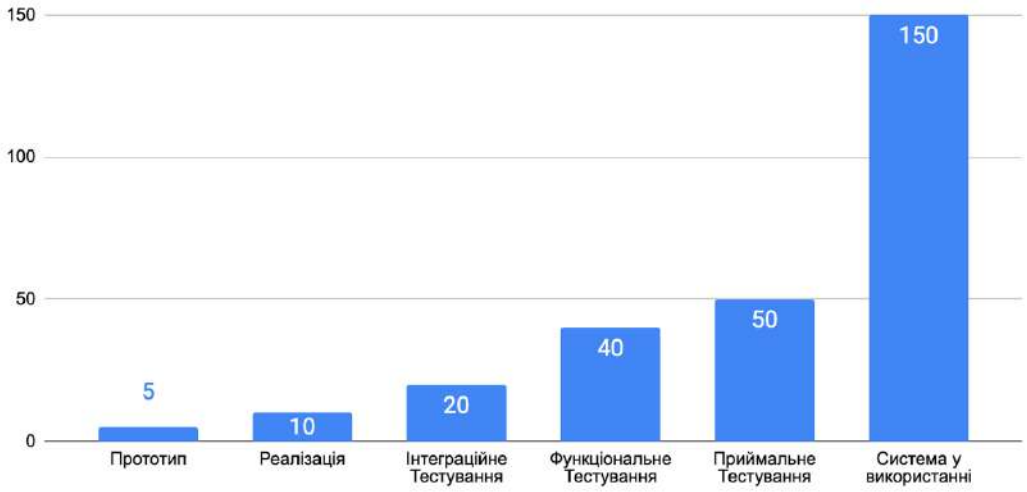
В результаті, ми отримали всі 3 візуальні панелі в одному місці – у сервісі Grafana:

1. Метрики тестів серверного навантаження.
2. Інфраструктурні метрики системи під навантаженням.
3. Метрики тестування продуктивності клієнта.

Такий підхід є зручним, адже дає можливість швидко аналізувати результати та шукати взаємозв’язки між ними.

3.3 Архітектура тестування продуктивності в CI/CD

Відомий факт, що чим раніше знайдена функціональна помилка, тим менша вартість її виправлення (див. Графік 3.3.1) [7].



Графік 3.3.1 – Вартість помилки на різних стадіях розробки

Подібних досліджень для нефункціональних помилок до яких належать також і проблеми з продуктивністю ніхто не робив. Очевидно, і уже не одноразово доказано на практиці, що якщо система перестає відповідати, то наслідки для компанії можуть бути драматичними, аж до повного банкрутства та припинення існування.

Ян Мулінекс у книзі The Art of Application Performance Testing зазначив наступне: «Часто проблеми з продуктивністю помічають на останніх етапах, але чим пізніше вони будуть виявлені, тим більшими будуть витрати для подолання їх наслідків» [18].

Тому у світовій практиці йде зміщення тестування продуктивності вліво по осі часу розробки програмного забезпечення (shifting left). Окрім того, його важливо продовжувати уже і коли система є загальнодоступною, адже вона весь час еволюціонує та зазнає впливу різних як зовнішніх так і внутрішніх факторів – зміна функціоналу, зростання кількості клієнтів, тощо. Таке проактивне

тестування дозволяє попередити потенційні проблеми з продуктивністю та уникнути їхніх наслідків.

Тому ми рекомендуємо інтегрувати тестування продуктивності у безперервну поставку коду на проекті (Continuous Integration / Continuous Delivery). Це є надзвичайно важливо для динамічних проектів, де розробка йде за методологіями SCRUM чи Kanban) і присутні часті зміни функціоналу, інфраструктури, тощо.

Безперервне тестування продуктивності також дає змогу порівнювати результати від тесту до тесту, розуміючи таким чином як вплинула на систему та чи інша зміна коду, див. Рисунок 3.3.2.

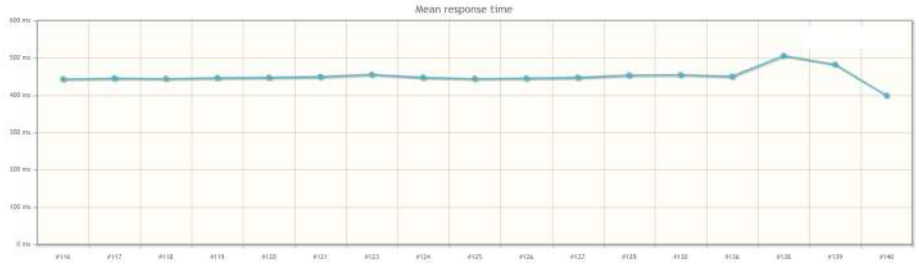


Рисунок 3.3.2 - Тренд змін часу відгуку сторінок веб-сайту від поставки до поставки

На ринку представлені різні продукти для безперервної поставки коду, такі як:

- Jenkins;
- Bamboo;
- Circle CI;
- TeamCity;
- GitLab CI та інші.

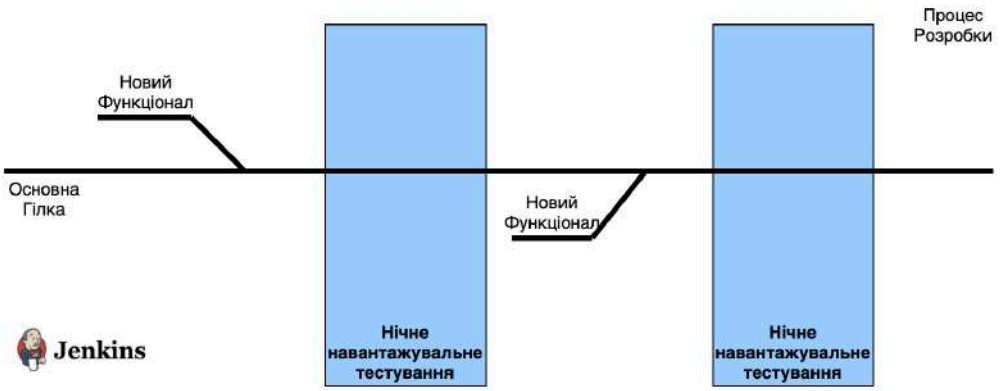
Для цілей даної курсової роботи, та для багатьох інших реальних проектів у тому числі, ми будемо використовувати такий інструмент як Jenkins, який має відкритий код, є безкоштовним та одним із найпоширеніших у світі серед подібних згідно «Comparison of Most Popular Continuous Integration Tools» [12].

Jenkins працює через створення таких об’єктів як «Джоба» (Job), які безпосередньо і займаються збіркою коду чи запуском необхідних тестів відповідно заздалегідь прописаних процедур. До того ж, запуск тестів можна налаштувати автоматичним за наступними критеріями:

- Нічний тест.
- Після змін у коді тестів.
- У ручному режимі.
- Після змін у коді основного проекту.

Останній критерій є доволі незручним, адже навантажувальне тестування є відносно не швидким і може займати як мінімум від 10 до 20 хвилин. Тому якщо кожен раз його запускати після змін у основному коді, це значно сповільнить весь процес розробки.

Саме тому ми рекомендуємо запускати навантажувальне тестування у нічному режимі щодня чи раз у декілька днів (див. Діаграма 3.3.3), або якщо і запускати його після кожної зміни основного коду, то на окремому середовищі та із вбудованою зручною системою оповіщення, яка дасть змогу розробнику подивитися на результати по закінченню тестування.



Діаграма 3.3.3 - Процес тестування продуктивності під час нічної збірки

Таким чином, команда розробки матиме результати навантажувального тестування на постійній основі та може реагувати заздалегідь на потенційні проблеми, уникнувши негативних наслідків для кінцевих користувачів системи.

Відносно середовища, то навантажувальне тестування завжди рекомендується робити на окремому незалежному середовищі для уникнення зовнішніх «шумів». Ба більше, це середовище має бути максимально наближеним до реального.

Порядок та черговість тестування продуктивності у всьому процесі розробки того чи іншого функціоналу програмного забезпечення зображено на Рисунку 3.3.4.

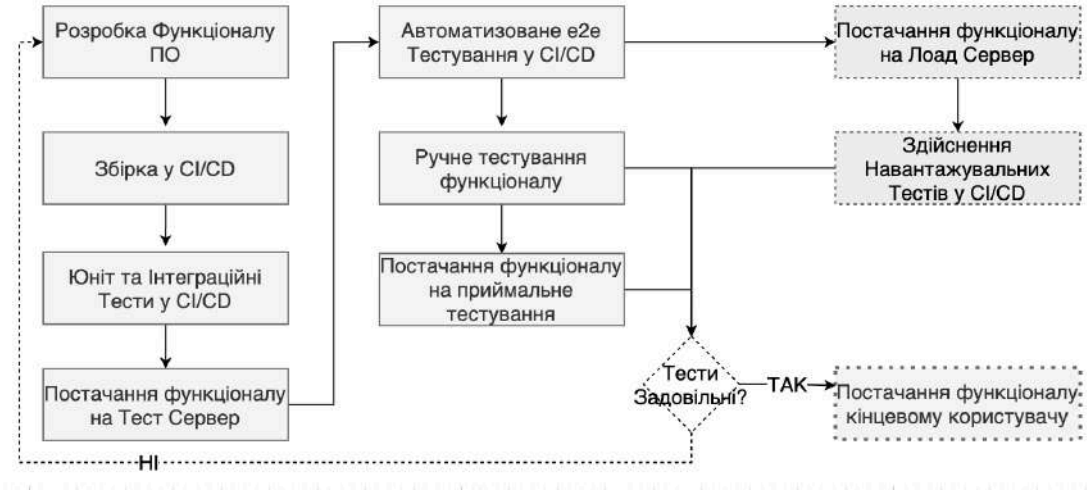


Рисунок 3.3.4 – Навантажувальне тестування у життєвому циклі розробки функціоналу ПЗ

Як видно з Рисунку 3.3.4, ми рекомендуємо здійснювати запуск навантажувальних тестів лише після того як успішно відбулися усі базові функціональні перевірки та тести. Немає сенсу перевіряти продуктивність системи якщо її функціонал не працює.

В результаті, всю систему для здійснення одночасно навантажувального тестування на сервер та тестування продуктивності веб-сторінки у браузері можна зобразити наступним чином (див. Рисунок 3.3.5).

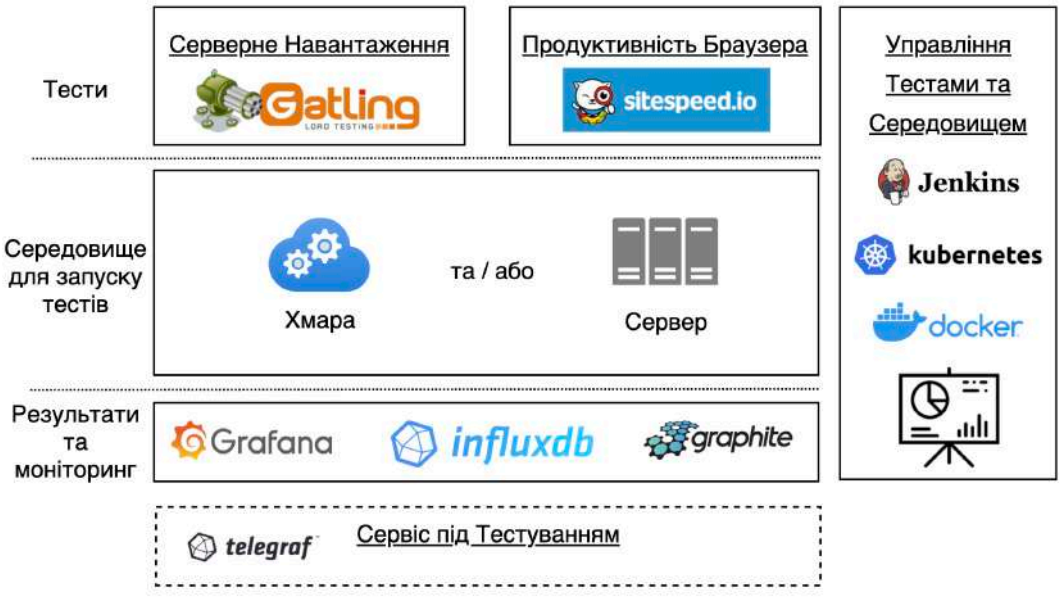


Рисунок 3.3.5 – Компонентна діаграма системи для тестування продуктивності

Як бачимо з представленої архітектури, нам необхідно буде реалізувати щонайменше 2 Jenkins «Job» - для серверного навантаження та тестування продуктивності сторінки у браузері. Саме тестування ми можемо здійснювати або з Хмари, або з власних серверних ресурсів, можливо з дата-центру. У розрізі даної дипломної роботи ми використовували хмарних провайдерів для цих цілей. Важливо також додати, що нам буде необхідний доступ до самої системи яка піддається навантаженю, для встановлення Telegraf агента, який у свою чергу здійснює збір інфраструктурних метрик та відправляє їх у нашу централізовану систему моніторингу. В іншому випадку, нам необхідний доступ до інших наявних систем моніторингу сервісу який належить тестувати, адже саме ці дані дадуть нам змогу зрозуміти як програмне забезпечення реагує на різний рівень навантаження.

Висновки по роботі та рекомендації для подальших досліджень

В першому розділі даної дипломної роботи були розглянуті теоретичні засади тестування навантаження та продуктивності веб-сайтів та інших цифрових систем. Розглянуті їх цілі, види, підходи, відмінності та процес організації в цілому. Також ми виконали порівняльний аналіз інструментів та уже існуючих продуктів для здійснення такого виду тестування, як безкоштовних так і комерційних. У результаті, для того щоб здійснити вибір серед великого різноманіття навантажувальних інструментів, була розроблена методика та критерії для здійснення такого вибору відповідно вимог та потреб того чи іншого проекту. Таким чином, для здійснення навантажувального тестування в нашій архітектурі було запропоноване використання такого інструменту з відкритим доступом до коду як Gatling, а для тестування продуктивності веб-сторінки у браузері Sitespeed.io відповідно.

Другу частину було присвячено основам та підходам зі збору та аналізу різного роду метрик. До таких можуть належати як метрики безпосередньо результатів навантажувальних тестів, так і показники інфраструктури системи яка тестується. Для того аби обрати вірні метрики для аналізу які дадуть змогу швидко зрозуміти стан системи під навантаженням, було рекомендовано декілька уже існуючих та, на нашу думку, дійсно корисних підходів. Важливим питання також стало централізоване зберігання таких метрик. Тобто ми хочемо, щоб усі показники – як результати тестів, так і метрики самої інфраструктури – зберігалися централізовано та в реальному часі в одному місці, звісно організовано відповідно до вимог. Такий підхід дасть змогу зручно шукати залежності між різними показниками та, в результаті, швидко реагувати на ті чи інші зміни. Для цього ми порівняли уже існуючі інструменти на ринку та обрали один з таких на основі бази даних InfluxDB та сервісу візуалізації Grafana, які до того ж також є безкоштовними.

У третьому розділі ми представили пропоновану архітектуру побудови підсистеми розподіленого тестування навантаження у системі безперервної поставки коду CI/CD Jenkins. Додатково ця архітектура також передбачає

тестування продуктивності веб-сайтів у браузері, а також збір та відображення метрик інфраструктури системи яку тестують. Слід зазначити, що архітектура була побудована на основі здійснених аналізів та висновків у попередніх двох розділах даної дипломної роботи.

Отримана в результаті архітектура може бути надалі реалізована як на приватних чи приналежних тому чи іншому дата центрі серверах, так і з використанням віртуальних машин у хмарі. Завдяки інструменту кластеризації Kubernetes побудована архітектура є незалежною від інфраструктури її реалізації.

Слід зазначити, що одним з рекомендованих подальших кроків є безпосередня реалізації та випробування прототипу системи розподіленого навантаження, адже в процесі її побудови можуть виникнути ті чи інші практичні уточнення та деталі.

В цілому, пропонована архітектура дасть змогу побудувати підсистему для здійснення навантажувального тестування та тестування продуктивності з централізованим зберіганням та відображенням результатів, яка також інтегрована в систему CI/CD. Дана підсистема, в свою чергу, задовольнить потреби великих проектів у здійсненні масштабованого навантажувального тестування на постійній умові та потенційного попередження негативних наслідків від низької продуктивності розроблюваних сайтів та інших цифрових систем.

Список літератури

1. Abi Travers. *Critical Render Path Optimisation — How To Increase Your Page Speed.* [Електронний ресурс] – 2019 – Режим доступу: https://medium.com/comparethemarket/critical-render-path-optimisation-how-to-increase-your-page-speed-820241a4552f
2. Abner Germanow. *An In-Depth Look at the New Relic Platform.* [Електронний ресурс] – 2016 – Режим доступу: https://newrelic.com/blog/how-to-relic/an-indepth-look-at-the-new-relic-platform
3. Andrew Lerner. *The Cost of Downtime*. [Електронний ресурс] – 2014 – Режим доступу: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/
4. *Apdex Overview*. [Електронний ресурс] – Режим доступу: https://www.apdex.org/overview.html
5. Atul Kumar. *High Availability in Kubernetes | Detailed Guide.* [Електронний ресурс] – Режим доступу: https://k21academy.com/docker-kubernetes/high-availability-and-scalable-application-in-kubernetes
6. Baron Schwartz. *Monitoring and Observability With USE and RED.* [Електронний ресурс] – 2017 – https://orangematter.solarwinds.com/2017/10/05/monitoring-and-observability-with-use-and-red
7. Barry Boehm. *Cost of remediation source: "Equity Keynote Address"*, 2007
8. Betsy Beyer, Jennifer Petoff, Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016 Print – С. 60-62
9. Bill Kayser. *How to Choose Apdex T: The Final Word.* [Електронний ресурс] – 2017 – Режим доступу: https://newrelic.com/blog/best-practices/how-to-choose-apdex-t
10. Brendan Gregg. *Systems Performance, Second Edition*. Pearson Education (US), 2020 Print – С. 32, 46–53
11. *Client Side Performance*. [Електронний ресурс] – Режим доступу: https://qautomation.blog/client-side-performance
12. *Comparison of Most Popular Continuous Integration Tools: Jenkins, TeamCity, Bamboo, Travis CI and more*. [Електронний ресурс] – 2019 – Режим доступу: https://www.altexsoft.com/blog/engineering/comparison-of-

most-popular-continuous-integration-tools-jenkins-teamcity-bamboo-travis-ci-and-more/

13. Daniel Berman. *InfluxDB vs. Elasticsearch for Time Series Analysis.* [Електронний ресурс] – 2017 – Режим доступу: https://logz.io/blog/influxdb-vs-elasticsearch
14. *Distributed load testing on AWS.* [Електронний ресурс] – 2019 – Режим доступу: https://github.com/awslabs/distributed-load-testing-on-aws
15. Domenico Stragliotto. *How we implemented RED and USE metrics for monitoring.* [Електронний ресурс] – 2019 – Режим доступу: https://medium.com/thron-tech/how-we-implemented-red-and-use-metrics-for-monitoring-9a7db29382af
16. Eldad Livni. *How to Use Prometheus Monitoring*. [Електронний ресурс] – 2020 – Режим доступу: https://stackpulse.com/blog/prometheus-monitoring-architecture-and-challenges
17. Graham, Bath and Rex Black. *Foundation Level Specialist Syllabus Performance Testing*. ISTQB, 2018. Print – C. 10 – 15
18. Ian Molyuneaux. *The Art of Application Performance Testing.* O'Reilly, 2015, Print – C. 5-10
19. Ilya Grigorik. *Critical Rendering Path.* [Електронний ресурс] – Режим доступу: https://developers.google.com/web/fundamentals/performance/critical-rendering-path
20. ISO 25000 STANDARDS. ISO 25010. Performance efficiency [Електронний ресурс] – 2019 – Режим доступу: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/59-performance-efficiency
21. Katie Mckelvie. *How to I measure page speed.* [Електронний ресурс] – 2018 – Режим доступу: http://reprisemedia.com.au/blog/mobile-speed-update-july-2018
22. *Load speed can make or break a website.* [Електронний ресурс] – 2021 – Режим доступу: https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics
23. Marco Pas. *Infrastructure & System Monitoring using Prometheus.* [Електронний ресурс] – 2017 – Режим доступу: https://www.slideshare.net/MarcoPas1/infrastructure-system-monitoring-using-prometheus

24. Michael Copeland. *The Cost of IT Downtime.* [Електронний ресурс] – 2020 – Режим доступу: https://www.the20.com/blog/the-cost-of-it-downtime/

25. Milica Mihajlija. *The New Generation of Performance Metrics for Better User Experience.* [Електронний ресурс] – 2020 – Режим доступу: https://calibreapp.com/blog/new-generation-of-performance-metrics

26. Phillip Pachael. *Case study: Reasons why Google's Orkut failed after Facebook was launched. Also know what Orkut's founder is doing recently to start up again.* [Електронний ресурс] – 2017 – Режим доступу: https://medium.com/@PachaelPhillip/case-study-reasons-why-googles-orkut-failed-after-facebook-was-launched-92dd8a7abf0

27. Philip Walton. *User-centric performance metrics.* [Електронний ресурс] – 2018 – Режим доступу: https://web.dev/user-centric-performance-metrics

28. Rob Ewaschuk. *Google SRE. Monitoring Distributed Systems.* [Електронний ресурс] – 2021 – Режим доступу: https://sre.google/sre-book/monitoring-distributed-systems

29. Schkn. *How To Setup Telegraf InfluxDB and Grafana on Linux*. – 2021 – Режим доступу: https://devconnected.com/how-to-setup-telegraf-influxdb-and-grafana-on-linux

30. Scott Barber and Colin Mason. *Web Load Testing For Dummies.* John Wiley & Sons, Inc., 2011 Print – С. 4 – 5

31. *WebPageTest. Speed Index.* [Електронний ресурс] – Режим доступу: https://docs.webpagetest.org/metrics/speedindex

32. *What is DDoS Attach?* [Електронний ресурс] – Режим доступу: https://aws.amazon.com/shield/ddos-attack-protection/

33. Wilkie, T., *The RED Method: Patterns for Instrumentation & Monitoring*, *Grafana Labs*. [Електронний ресурс] – 2018 – Режим доступу: https://www.slideshare.net/grafana/the-red-method-how-to-monitoring-your-microservices

34. *12 Techniques of Website Speed Optimization: Performance Testing and Improvement Practices*. [Електронний ресурс] – 2018 – Режим доступу: https://www.altexsoft.com/blog/engineering/12-techniques-of-website-speed-optimization-performance-testing-and-improvement-practices/

35. Офіційний сайт Akka: https://github.com/akka/akka

36. Офіційний сайт Blazemeter: https://www.blazemeter.com

37. Офіційний сайт Docker: https://www.docker.com

38. Офіційний сайт Gatling: https://gatling.io
39. Офіційний сайт Gatling Frontline: https://gatling.io/gatling-frontline
40. Офіційний сайт Gradle: https://gradle.org
41. Офіційний сайт Grafana: https://grafana.com
42. Офіційний сайт Graphite: https://graphite.readthedocs.io
43. Офіційний сайт InfluxDB: https://www.influxdata.com
44. Офіційний сайт Jenkins: https://www.jenkins.io
45. Офіційний сайт JMeter: https://jmeter.apache.org
46. Офіційний сайт K6: https://k6.io
47. Офіційний сайт Kubernetes: https://kubernetes.io
48. Офіційний сайт Lighthouse: https://developers.google.com/web/tools/lighthouse
49. Офіційний сайт LoadRunner: https://www.microfocus.com/en-us/products/loadrunner-professional/overview
50. Офіційний сайт LoadUI: https://smartbear.com/product/ready-api/api-performance-testing
51. Офіційний сайт Locust: https://locust.io
52. Офіційний сайт Logstash: https://www.elastic.co/logstash
53. Офіційний сайт NewRelic: https://newrelic.com
54. Офіційний сайт PageSpeed Insights: https://developers.google.com/speed/pagespeed/insights
55. Офіційний сайт Prometheus: https://prometheus.io
56. Офіційний сайт Sitespeed.io: https://www.sitespeed.io
57. Офіційний сайт з Панелями візуалізації Grafana для Sitespeed.io: https://github.com/sitespeedio/grafana-bootstrap-docker/tree/main/dashboards/graphite
58. Офіційний сайт WebPageTest: https://www.webpagetest.org