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

Розподілена система навантажувального тестування у безперервній інтеграції.
Візуалізація результатів у реальному часі.

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
д.т.н., доц. Глибовець А.М.
_____
(підпис)
“____” _____ 2020 р.

Виконав студент
Ковш М.В.
“____” _____ 2020 р.

Київ 2020

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 курсову роботу

Студента Ковша М.В. факультета інформатики 1 курсу МП

Тема: Розподілена система навантажувального тестування у безперервній інтеграції. Візуалізація результатів у реальному часі

Вихідні дані: практичні здобутки за темою роботи

Зміст ТЧ до курсової роботи:

Індивідуальне завдання;

Зміст;

Вступ;

1. Огляд технічної та теоретичної літератури;
2. Дослідження комерційних та загальнодоступних інструментів;
3. Дослідження обмежень інструментів та проектування рішення;
4. Розробка системи;
5. Випробовування системи та огляд результатів;

Висновки;

Література;

Додатки.

Дата видачі “____” _____ 20__ р. Керівник _____
(підпис)

Завдання отримав_____
(підпис)

Тема: Розподілена система навантажувального тестування у безперервній інтеграції. Візуалізація результатів у реальному часі

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	06.12.2019	
2.	Огляд технічної літератури за темою роботи	20.12.2019	
3.	Огляд інструментів для навантажувального тестування	24.01.2020	
4.	Огляд інструментів для реалізації розподіленої системи	21.02.2020	
5.	Огляд інструментів для візуалізації метрик у реальному часі	13.03.2020	
6.	Побудова системи на основі загальнодоступних інструментів	17.04.2020	
7.	Написання теоретичної частини та опис системи	30.04.2020	
8.	Випробовування системи під час тестування сайтів КМА. Запис результатів	08.05.2020	
9.	Коригування роботи за результатами попереднього огляду роботи.	14.05.2020	
10.	Підготовка до захисту	22.05.2020	
11.	Захист курсової роботи	29.05.2020	

Студент Ковш М.В.

Керівник Глибовець А.М. “____” _____ 2020 р

ЗМІСТ

Анотація

У даній курсовій роботі розглянуті основні типи, цілі та процес реалізації тестування продуктивності на проектах. Основним результатом роботи стала побудова розподіленої системи для тестування навантаження у процесі безперервної поставки. Також реалізоване централізоване звітування результатів тестів у реальному часі. Система побудована з використанням загальнодоступних інструментів на ринку.

Ключові слова: продуктивність системи, тестування продуктивності, тестування навантаження, безперервна поставка коду, відображення результатів тестування у реальному часі, розподілена система навантаження.

Вступ

Продуктивність системи – це окрема інженерна наука, яка включає в себе підходи та практики для побудови і оптимізації систем на усіх рівнях – від фізичних приладів, інтернет мережі до написання високоефективних алгоритмів. Водночас будь-яка система що проектується і будується має бути перевірена та випробовувана, бажано на реальних умовах, у тому числі і на продуктивність. І тут приходить на допомогу такий вид тестування як тестування продуктивності, чи як його часто називають навантажувальне тестування. Хоча останній термін описує не усі можливі підходи у цьому напрямку тестування, про що також йтиметься мова у даній курсовій роботі.

Саме тестування продуктивності покликане симулювати реальні умови використання системи та навантаження на неї для того щоб перевірити саму інформаційну систему та вказати на потенційно вузькі місця у ній.

Даний вид тестування є важливим не лише з позиції оптимізації ресурсів, але потенційно може вберегти комерційний (і не лише) онлайн сайт від фінансових та репутаційних втрат. Наприклад, досить відома історія з падінням державного сайту США для медичного страхування www.HealthCare.gov. Система перестала працювати уже через 2 години після запуску, адже не була готова до навантаження у 250 тисяч користувачів [22]. І таких історій уже достатньо у світовій практиці.

Усе вищезгадане приводить нас до висновку, що тестування навантаження є надзвичайно важливою та актуальною темою сьогодення. Заразом, нажаль, багато компаній починають про нього задумуватися лише після того, як уже відчують негативні наслідки. Такій ситуації є багато пояснень, включаючи постійний дефіцит часу, бюджетів та загалом такого роду спеціалістів на ринку.

До однієї з цілей даної курсової роботи можна віднести сповідування та підтримку ідеї реалізації тестування продуктивності якомога раніше на проекті, що і дозволить уникнути потенційно-негативних наслідків. Більше того, таке тестування має відбуватися циклічно (безперервно) протягом

усього життєвого циклу продукту для відслідковування впливу нових змін у проекті.

Сама система навантажувального тестування теж має виповідати певним критеріям та вимогам, реалізація яких дозволить провести повноцінне та ефективне тестування. Так, для перевірки великих (в тому числі розподілених) інформаційних систем необхідно будувати розподілену систему навантаження. Окрім того, може виникнути необхідність створювати це навантаження одночасно з різних регіонів планети. В свою чергу, велике тестування продукує великі об’єми результуючих даних, які зручно збирати та відслідковувати централізовано у реальному часі, що дає змогу швидкої реакції.

Актуальність даної теми полягає в тому, що незважаючи на присутність на ринку комерційних інструментів навантажувального тестування які практично дозволяють реалізувати вище згадані вимоги, часто необхідно будувати власну (під проект) систему для відображення чи то більшої кількості необхідних метрик, чи то неможливості використання сторонніх систем, чи то бюджетних обмежень, а можливо і все разом.

Мета даної роботи – дослідити присутні на ринку загальнодоступні інструменти (open-source) для навантажувального тестування. На основі цих інструментів спроектувати та побудувати таку систему, яка б дозволила інтегруватися в систему безперервної поставки проекту, масштабуватись та централізовано збирати результати тестування в реальному часі.

Об’єкт дослідження – інструменти та підходи для навантажувального тестування, безперервної інтеграції та візуалізації великих об’ємів даних.

Методи дослідження – аналіз доступної інформації за даною темою, враховуючи власний практичний досвід, та побудова розподіленої системи навантажувального тестування.

РОЗДІЛ 1: Основи тестування продуктивності

1.1 Тестування продуктивності. Роль та цілі

Продуктивність тієї чи іншої системи є вагомою частиною у забезпеченні високого рівня задоволення нею користувачем. Саме тестування продуктивності відіграє критичну роль у процесі оцінки та встановлення прийнятних рівнів якості для кінцевого користувача. Більше того, цей вид тестування часто тісно інтегрований з такими дисциплінами як інженерія зручності використання, інженерія захисту інформації та інженерія продуктивності.

Більше того, тестування функціоналу, зручності використання та інших характеристик системи у комбінації з тестуванням її продуктивності може допомогти виявити та виправити специфічні лише в цих умовах помилки.

Скотт Барбер у своїй праці *«Навантажувальне тестування Веб для новачків»* дає досить просте але влучне визначення: «Навантажувальне тестування – це те як ви визначаєте скільки трафіку ваш веб-сайт чи веб-застосунок може витримати перед тим як упаде чи змусить користувача писати негативні відгуки на просторах інтернету. Іншими словами, це не більше ніж випробування вашого веб-сайту в умовах реального використання і знаходження помилок перед тим як їх знайдуть кінцеві споживачі» [21].

І як показують дослідження, повільне завантаження веб-сайту дійсно може спричинити користувачів ділитися негативним досвідом та зменшити відвідуваність, і як результат прибутковість, сайту. Це підтверджує опитування користувачів онлайн магазинів компанією Unbounce, які виявили, що для 70% споживачів швидкість завантаження сторінки має вплив на ймовірність здійснення купівлі на тому чи іншому ресурсі [20].

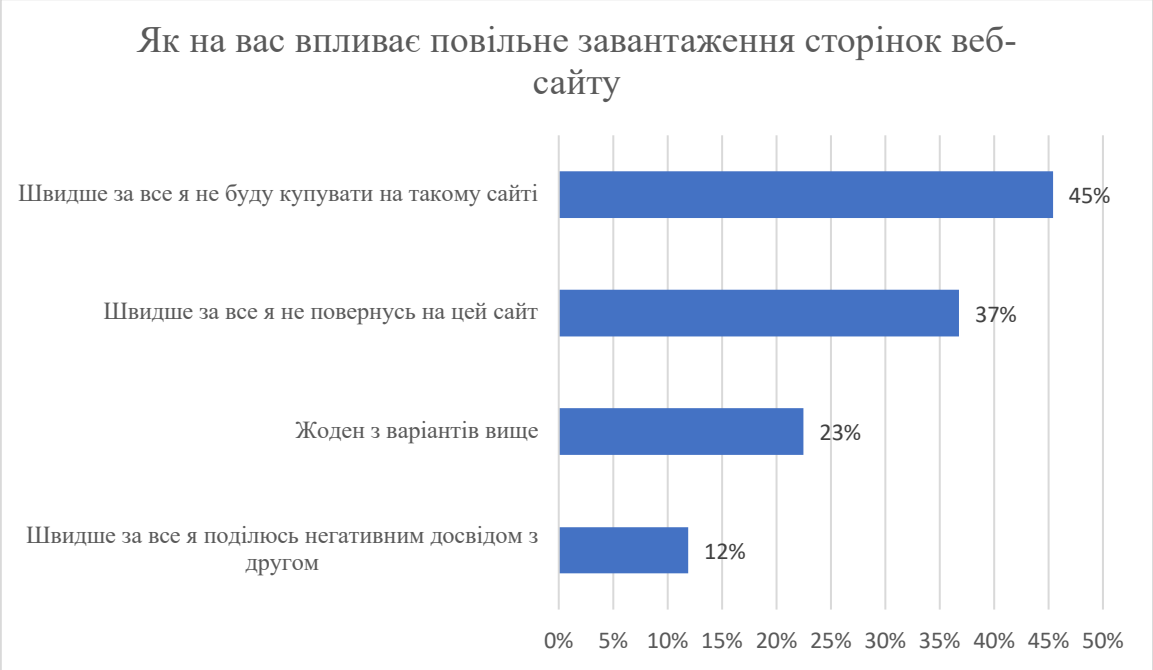


Рисунок 1.1.1 - Вплив швидкості завантаження веб-сторінок на поведінку користувача

Як видно з Рисунку 1.1.1 швидкість завантаження, тобто продуктивність, має значний вплив на прийняття рішення споживачем. Більше того, Google дослідив що 53% мобільних користувачів залишить веб-ресурс, якщо час його завантаження триватиме більше чим 3 секунди [14].

Саме для того, щоб уникнути негативних наслідків описаних вище і проводиться тестування продуктивності системи як в цілому, так і її окремих компонентів з подальшим виправленням потенційних вузьких місць та помилок. Водночас такий вид тестування здійснюється не лише для веб-ресурсів де у фокусі кінцевий споживач але і для класичних клієнт-сервер, розподілених, вбудованих та інших типів систем.

Запитаємо себе – чи лише часові характеристики цікавлять тестування продуктивності. Для того, щоб відповісти на це запитання необхідно спочатку зрозуміти, що саме розуміється під продуктивністю. Так, стандарт ISO25010 [ISO25000] визначає продуктивність (ефективність) системи за допомогою наступних характеристик:

- **Часова складова** – ступінь відповідності вимогам за часом відгуку, часом обробки та пропускної здатності продукту чи системи протягом свого функціонування.
- **Використання ресурсів** – ступінь відповідності вимогам використання різних типів ресурсів під час функціонування системи.
- **Ємність** – ступінь, за якої максимальні межі використання системи задовольняють вимоги [17].

Базуючись на даному визначенні, можна сказати, що тестування продуктивності включає оцінювання за двома напрямками:

1. **Оцінювання орієнтоване на сервіс, який надає система** – доступність сервісу та час відгуку; вони заміряють на скільки повноцінно (чи не повноцінно) система надає сервіс кінцевому споживачу.
2. **Оцінювання орієнтоване на саму систему** – пропускна здатність, використання ресурсів та ємність систему; вони заміряють на скільки ефективно (чи неефективно) система використовує свою інфраструктуру.

Узагальнюючи, можна сказати, що тестування продуктивності здійснюється з ціллю щоб визначити та/чи уникнути один або більше ризиків пов’язаних із стабільністю, швидкістю, надійністю, потенційними втратами фінансових і не лише ресурсів чи репутації. До більш конкретних цілей можна віднести наступні (за визначеними вище напрямки).

- **Сервіс-орієнтовані цілі:**
    - оцінка та передбачення характеристик продуктивності при змінні кодової бази, архітектури, інфраструктури, конфігурації чи інших властивостей системи;
    - надання інформації для визначення та передбачення рівня задоволення кінцевих користувачів при різних характеристиках продуктивності системи;

- надання інформації для оцінки потенційних втрат через проблеми з масштабованістю чи стабільністю.

- **Системо-орієнтовані цілі:**
  - оцінювання відповідності поточним там плановим вимогам ємності інфраструктури системи;
  - оцінювання доступності, стабільності та масштабованості системи;
  - порівняння різних конфігурацій системи та визначення кращої згідно поставлених вимог;
  - оцінювання характеристик продуктивності системи до та після змін;
  - оцінювання використання різних типів ресурсів (процесорний час, фізична та оперативна пам'ять, інтернет);
  - оцінювання поведінки системи при різних рівнях навантаження.
  - пошук вузьких місць системи з точки зору продуктивності;
  - надання інформації для прийняття рішення щодо запуску загальнодоступної версії продукту;
  - надання інформації для оцінювання бюджету для забезпечення необхідного рівня продуктивності системи.

1.2 Класифікація тестування продуктивності

В літературі можна знайти різні класифікації та типи тестування продуктивності. На додаток, сама термінологія (як і який тип називати) викликає дискусії як серед теоретиків, так і серед практиків. Ситуація значно покращилась після появи у грудні 2018 року сертифікації ISQTB з тестування продуктивності та запропонованої нею класифікації типів даного виду тестування [15].

Отже, за згаданою сертифікацією, існують щонайменше такі типи тестування продуктивності:

- **Тестування продуктивності (performance)** – узагальнюючий термін, який об’єднує будь-який вид тестування, що спрямований на продуктивність (швидкість) системи чи компонента під різним розміром навантаження.
- **Навантажувальне тестування (load)** – тестування, що спрямоване на визначення здатності системи впоратися із значним навантаженням, очікуваним в реальних умовах.
- **Стрес тестування (stress)** – тестування, що спрямоване на визначення здатності системи впоратися із навантаженням, яке знаходиться у верхніх, і навіть вище, границях очікуваного використання системи в реальних умовах.
- **Тестування масштабування (scalability)** – тестування на здатність системи до масштабування (росту) для оброблення більшої кількості користувачів чи об’ємів інформації, не втрачаючи свого рівня продуктивності.

Стрес та тестування масштабування дозволяють встановити границі для систем моніторингу та попередження у реальному часі (такі як Datadog, NewRelic) та реагувати швидко в разі переходу цих границь.

- **Тестування піків (spike) –** тип тестування, за якого симулюють різке зростання навантаження (часто за межами границь) на короткий проміжок часу та оцінюють здатність системи до стабілізації після піку.
- **Тестування на виснажування (endurance) –** це тестування за якого навантаження певного рівня триває протягом тривалого часу, від чотирьох і більше годин (можливі випадки навіть у декілька днів). Найчастіше, ціллю такого тестування є виявлення проблем із системними ресурсами, а саме з вивільненням оперативної пам’яті, кількості підключень до бази, кількістю потоків та інші вузькі місця інфраструктури та конфігурації.
- **Тестування на одночасність (concurrency)** – тестування за якого конкретна дія виконується одночасно з великим навантаженням,

наприклад, реєстрація користувача. З досвіду, такий тип тестування можна було б ще назвати тестування навантаження окремої компоненти системи.

- **Тестування на ємність (capacity)** – тестування для оцінки того яке максимальне навантаження може витримати система та досі відповідати вимогам продуктивності. Об’єктом такого типу тестування може бути як кількість одночасних користувачів чи запитів так і об’єм оброблених даних.

Додатково, опираючись на практичний досвід, можна виокремити наступні додаткові класифікації типів тестування продуктивності.

1. **За типом сервісу чи протоколу:**
   а. продуктивність веб-запитів (HTTP(s));
   б. продуктивність інтерфейсів (API);
   в. продуктивність баз даних (JDBC);
   г. продуктивність веб-сокету (WebSocket);
   д. продуктивність поштових сервісів (SMTP, POP, IMAP);
   е. продуктивність сервісу обміну файлами ((s)FTP);
   ж. продуктивність телекомунікаційних протоколів (VoIP, SIP);
   з. продуктивність сервісів третього рівня (TCP, UDP).
2. **За спрямуванням тестування:**
   а. **тестування серверної частини** (запитів до серверу).
   Наведена вище класифікація ISTQB описує типи для здійснення тестування продуктивності спрямованого саме на серверну частину. Для його реалізації застосовують спеціальні інструменти з генерації навантаження такі як JMeter, Gatling та інші.
   б. **тестування продуктивності на клієнтській стороні** (частіше саме продуктивності відтворення у браузері).
   Дана активність покликана оцінити як швидко завантажується та відображається зміст у браузері. Зазвичай, для здійснення даного типу тестування не потрібно симулювати навантаження і

достатньо інструментів сучасного браузера. Хоча, на ринку є спеціальні інструменти які заміряють продуктивність клієнтської сторони, такі як Webpagetest, Lighthouse, PageSpeed та SiteSpeed.io. При цьому, потрібно враховувати, швидкість інтернету та потужності самого пристрою (персонального комп'ютера чи смартфона). Додатково, корисним буває поєднання навантажувального тестування на сервер та тестування продуктивності на клієнтській стороні.

в. **тестування продуктивності самого пристрою користувача.** Дана активність покликана оцінити рівень споживання ресурсів самого пристрою (ПК, смартфону) під час використання тієї чи іншої програми.

г. **тестування пропускної здатності інтернет каналу.**

3. **За типом самого тестування**

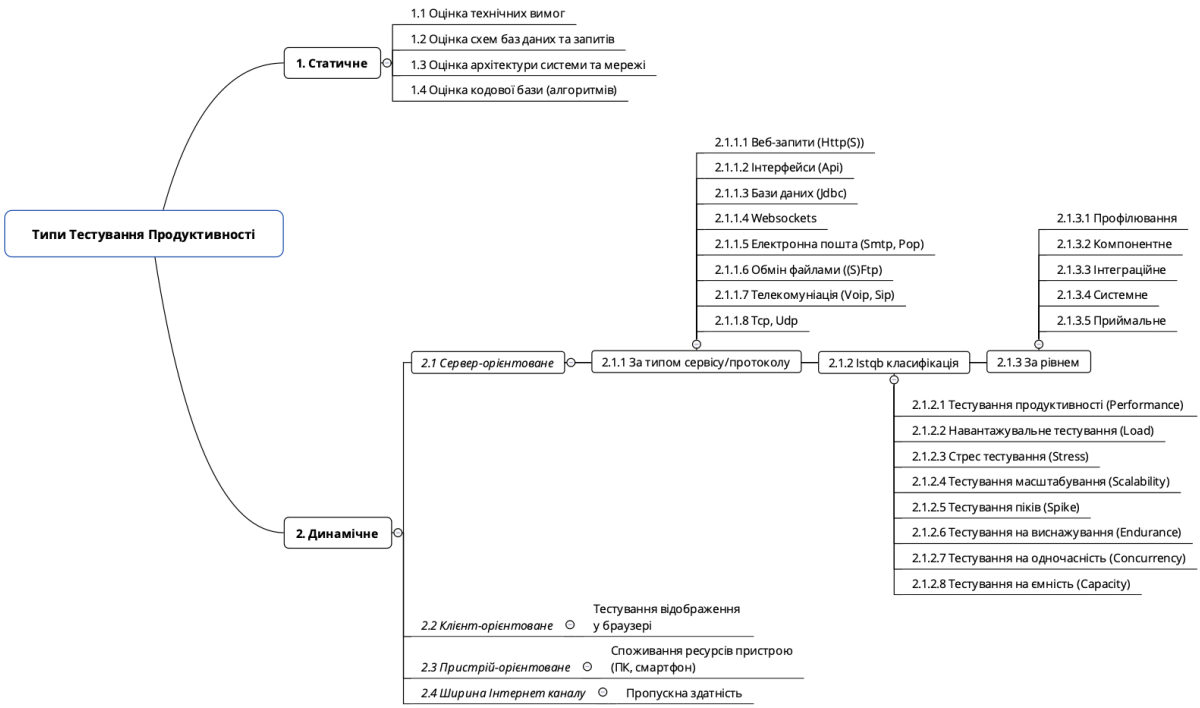
а. **Статичне тестування**

- оцінка технічних вимог через призму ризиків та аспектів продуктивності;
- оцінка схем баз даних, метаданих, процедур та запитів до бази;
- оцінка архітектури системи та мережі;
- Оцінка кодової бази (складних алгоритмів).

б. **Динамічне тестування**

- профілювання на рівні окремих частин коду;
- компонентне, інтеграційне та навантажувальне тестування системи в цілому;
- порівняльне тестування продуктивності;
- приймальне тестування перед запуском загальнодоступної версії продукту.

Узагальнюючи вище перераховані класифікації та для зручності, зобразимо їх на одній діаграмі, представленій нижче.



Діаграма 1.2.1 - Класифікація типів тестування продуктивності

Підсумовуючи дану частину роботи, можна сказати, що поки не існує єдиної та загальної класифікації типів тестування продуктивності. Навіть, якщо така і з'являється, то вона часто покриває не усі аспекти та багато в чому відображає емпіричний досвід автора (авторів), який не завжди є усестороннім. Хоча, досить повноцінну класифікацію запропонувала організація ISTQB, виділивши динамічні та статичні типи тестування продуктивності.

Практична частина даної курсовою роботи буде сфокусована саме на динамічний сервер-орієнтований тип тестування продуктивності веб запитів (пункт 2.1.1.1 на Діаграмі 1.2.1) із застосування навантажувального та стрес типів тестування за класифікацією ISTQB (пункти 2.1.2.1 та 2.1.2.3 на Діаграмі 1.2.1).

1.3 Етапи тестування продуктивності

Щоб розпочати реалізацію розподіленої системи навантажувального тестування та впровадити цей процес у безперервне постачання продукту разом із візуалізацією результатів в реальному часі, необхідно спочатку

зрозуміти на яких саме з етапів тестування продуктивності потрібно виконувати кожну з перерахованих дій. Саме тому, пропонується спочатку розглянути етапи тестування продуктивності. Це допоможе під час практичної реалізації значно оптимальніше розпланувати час та ресурси.

Досить всеохоплюючий алгоритм тестування продуктивності наведений у книзі Performance Testing Guidance for Web Application [18]. Тому візьмем його за основу та доповнимо, базуючись на власному практичному досвіді.

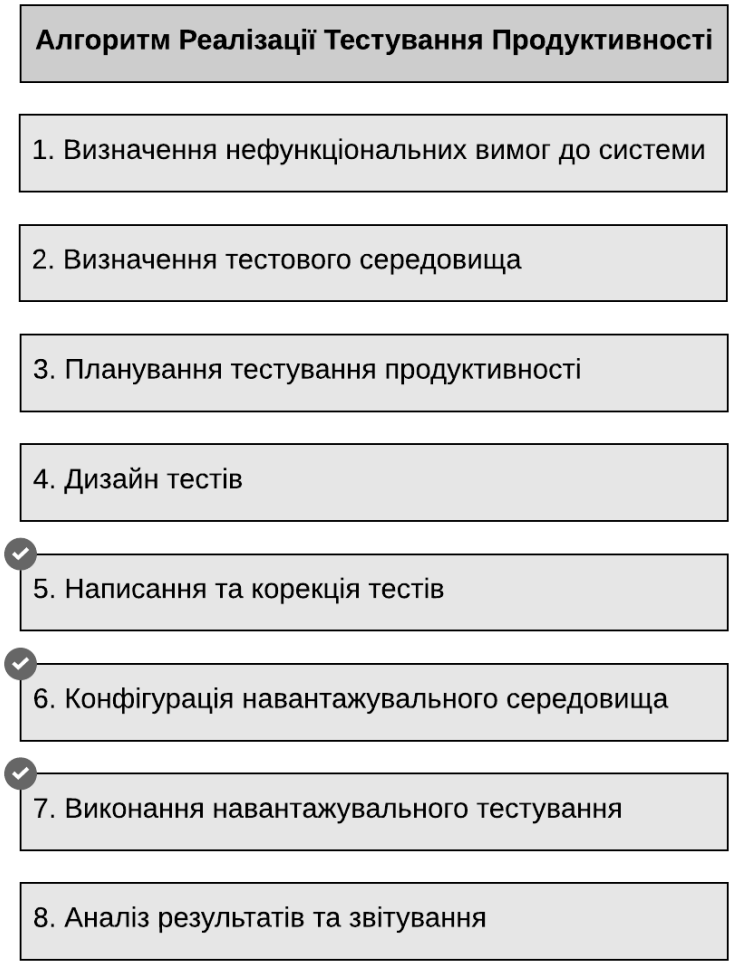


Рисунок 1.3.1 - Базові етапи тестування продуктивності

В даній курсовій праці основна практична робота відбувається в рамках 5-го, 6-го та 7-го етапів (виділених на Рисунку 1.3.1). Розглянемо коротко кожен з етапів.

1. **Визначення нефункціональних вимог до системи**

   На цьому етапі необхідно визначити:

   - очікуваний час відгуку;

- очікувана кількість одночасних користувачів;
- очікуване використання ресурсів системи;
- критичний функціонал;
- майбутнє зростання навантаження на систему;
- проблеми з продуктивністю, що траплялися раніше.

2. **Визначення тестового середовища (стенду)**

   Потрібно зрозуміти на якому середовищі відбуватиметься тестування та на скільки воно подібне до реального середовища. Рекомендовано, щоб тестовий стенд максимально відображав конфігурацію та інфраструктуру реального середовища. Саме такий підхід, дасть змогу досягти релевантних результатів. В іншому випадку, можна спробувати тестувати на реальному стенді, але у час максимально низького його використання споживачами та за підтримки команди програмістів.

3. **Планування тестування продуктивності**

   Визначаємо команду, цілі, інструменти, підхід, метрики для збору, пріоритети та графік виконання тестування продуктивності. Даний план необхідно транслювати та узгодити з усіма зацікавленими сторонами.

4. **Дизайн тестів**

   Визначаємо ключові сценарії, моделі навантаження та тестові дані які необхідно згенерувати. Даний етап часто може бути поєднаний з попереднім – планування.

5. **Написання та корекція тестів**

   Розробка тестових сценаріїв згідно з дизайном. Перші невеликі запуски тестів для здійснення їх потенційної корекції.

6. **Конфігурація навантажувального середовища**

   Для того щоб здійснити тестування з невеликим навантаженням, персональної машини розробника тестів може бути достатньо. Водночас для здійснення повноцінних навантажувальних тестів

необхідно налаштувати окреме середовище, часто розподілене (для дуже великого навантаження), саме з якого буде здійснюватися тестування. Про підхід до побудови та використання такого за допомогою засобів AWS та Jenkins і будемо говорити в практичній частині даної роботи.

7. **Виконання навантажувального тестування**

   Запуск та відслідковування результатів тестування. Часто, на великих системах, тестування продуктивності триває мінімум одну годину, а в середньому дві-три. Тому необхідно мати можливість відслідковувати результати тестування в реальному часі, щоб зупинити тест у разі негативних результатів уже на ранніх етапах.

   Під час тестування, необхідно збирати метрики як з сервісу (час відгуку, пропускна здатність) так і з самої системи (використання ресурсів, системні помилки). Для цього, дуже зручно коли усі результати відображаються в одному місці, що дає змогу значно швидше помітити залежності між різними показниками.

   Більш детально, процес побудови системи моніторингу за допомогою InfluxDB, Logstash та Grafana буде розглянуто у практичній частині.

8. **Аналіз результатів та звітування**

   Зібрані та проаналізовані результати і залежності необхідно надати усім зацікавленим особам для прийняття подальших рішень.

РОЗДІЛ 2: Теоретичні засади розподіленого тестування

2.1 Тестування продуктивності в безперервній поставці коду

Як було уже згадано в першій частині даної роботи, продуктивність системи відіграє значну роль у сприйнятті та прибутковості системи, наприклад комерційного онлайн магазину. Водночас про нефункціональні вимоги, у тому числі продуктивність, часто згадують під кінець розробки інформаційної системи або, що ще гірше, коли система перестає функціонувати під великим навантаженням, будучи уже загальнодоступною. Як зазначив Ян Мулінекс у своїй праці The Art of Application Performance Testing «проблеми з продуктивністю мають неприємну динаміку бути поміченими дуже пізно, і чим пізніше ви їх виявите, тим більше затрат і зусиль для їх вирішення необхідно» [16]. Саме ця залежність продемонстрована на Рисунку 2.1.1.

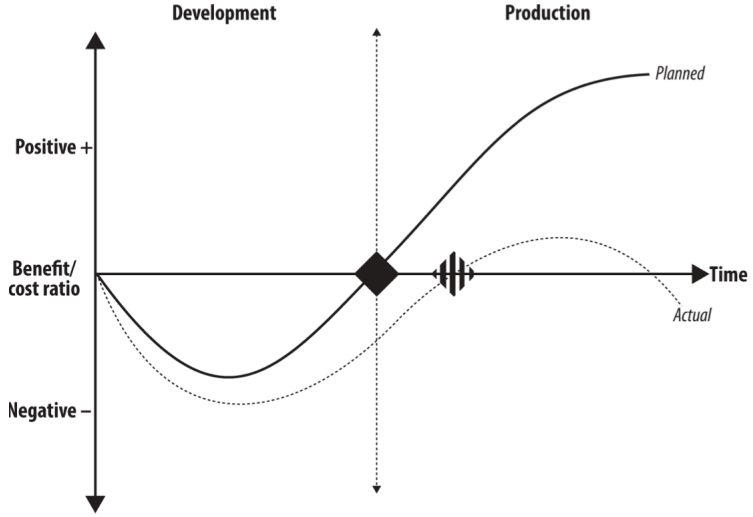


Рисунок 2.1.1 - Бізнес цінність системи. Очікування та реальність

Саме тому, останнім часом з’явилась динаміка впровадження тестування продуктивності не лише на ранніх етапах життєвого циклу розробки продукту, але і впродовж усього процесу розробки та підтримки системи. Такий підхід дозволяє не лише перевірити але і виправити негативні наслідки змін у продукті якнайшвидше.

В результаті ми приходимо до висновку, що тестування продуктивності необхідно здійснювати безперервно в процесі розробки продукту

(Continuous Integration / Continuous Delivery). Особливо це актуально на проектах де впроваджений гнучкий ітеративний (SCRUM, Kanban) процес розробки.

До того ж, це дає нам змогу відслідковувати тренд змін замірів продуктивності від поставки до поставки та постійно робити висновки про покращення чи навпаки деградацію системи, як зображено на Рисунку 2.1.2.

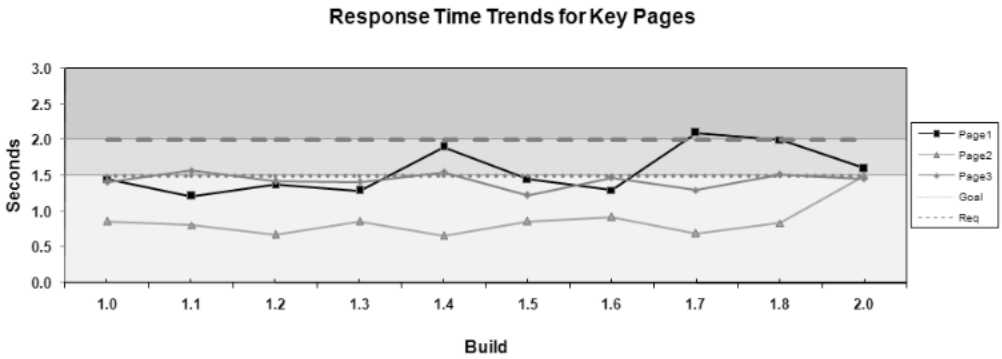


Рисунок 2.1.2 - Тренд змін часу відгуку сторінок веб-сайту від поставки до поставки

Для того, щоб зрозуміти чи саме тестування було успішним і не вийшло за узгоджені рамки, потрібно ці рамки встановити заздалегідь. Як показано на Рисунку 2.1.2, час відгуку для Сторінки 1 не відповідає очікуваному часу в 2 секунди для 1,7 та 1,8 версій поставки. Але в подальшому, дозволимо собі припустити, після оптимізації коду він зменшився до рівня 1,6 секунди при поставці версії 2,0 і перебував в очікуваних рамках.

Реалізацію безперервного навантажувального тестування можна здійснити за допомогою стандартних інструментів, адже сама система тестування не є нічим іншим як окремим цифровим продуктом розробка якого здійснюється за тими ж принципами, що і інші інформаційні системи. Тому, найчастіше з цією ціллю використовують наступні інструменти (залежно від вимог проекту):

- Jenkins;
- TeamCity;
- Bamboo;
- Travis CI;
- GitLab CI;

- Circle CI та інші.

У практичній частині даної курсової роботи буде використовуватись такий інструмент як Jenkins, який є загальнодоступним (відкритий код) та найпоширенішим інструментом для вирішення побідних задач згідно «Comparison of Most Popular Continuous Integration Tools» [12].

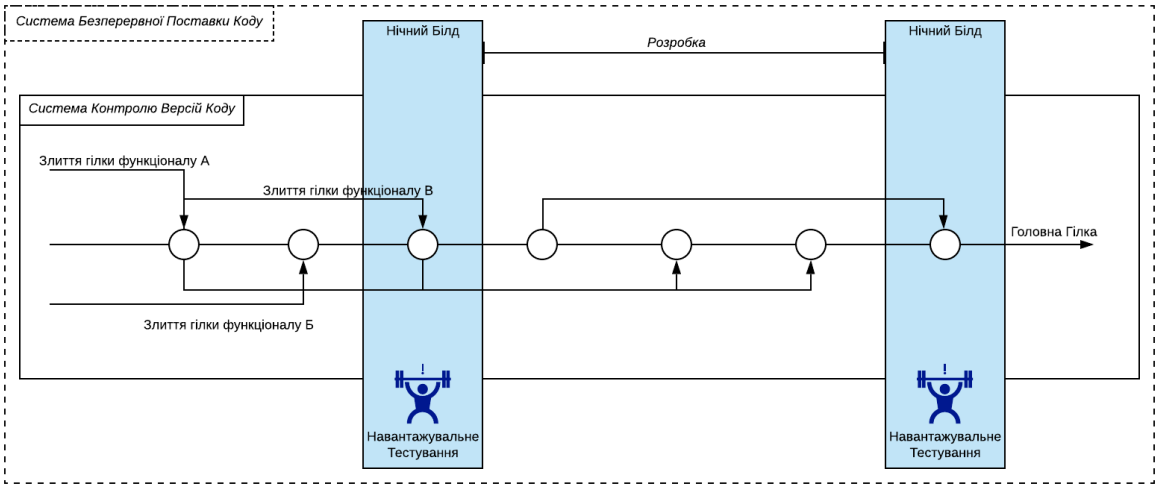
Разом з тим, хоча підхід безперервного тестування продуктивності системи дозволяє отримати оцінку системи якнайшвидше, сам цей процес тестування є досить тривалим і часто вимагає окремого середовища, як було описано в розділі 1.3. Так, навіть у самому позитивному сценарії, поверхневе навантажувальне тестування займатиме біля 10-15 хвилин. І тут виникає дилема, що ми не можемо проводити таке тестування після кожного злиття нового коду з кодовою базою. В середньому, один розробник здійснює злиття від одного до декількох разів в день. Таким чином, втрати цілої команди будуть вимірюватися годинами лише через тестування продуктивності, що є неприйнятним у нашому гнучкому середовищі. Саме тому, з практичного досвіду, можна запропонувати наступні стратегії для вирішення цієї дилеми:

1. **Орієнтація на реліз** - здійснення тестування продуктивності на реліз-кандидаті паралельно з приймальним тестуванням.
2. **Орієнтація на ітерацію** - здійснення тестування в кінці кожної ітерації розробки продукту але з врахуванням необхідного часу для виправлення потенційних помилок.
3. **Тестування на нічній збірці -** здійснення тестування вночі з певною періодичністю (щодня чи через певний проміжок часу).

Запропоновані перша та друга стратегії можуть збігатися у часі, якщо ітерація та реліз співпадають також.

Тут не можна сказати що якась із стратегій є кращою за іншу. Вибір залежить від багатьох чинників, у тому числі від потреби в такому виді тестування взагалі чи від наявності ресурсів.

Виходячи з власного досвіду роботи з високо-навантажувальними системами, можна рекомендувати третій підхід, за якого тестування продуктивності відбувається щоночі або через ніч чи інший проміжок часу (як зображено на Діаграмі 2.1.3).



Діаграма 2.1.3 - Процес тестування продуктивності під час нічної збірки

При такій моделі, аналіз результатів тестування відбувається на наступний день. За необхідності, у випадку спостереження негативної динаміки, тестування можна повторити протягом дня.

2.2 Розподілена система тестування продуктивності

В наш час стали популярними різні хмарні технології (AWS, Azure, GCP), які дозволяють розробникам будувати розподілені, масштабовані та велико-навантажувальні системи у відносно швидкий та зручний спосіб. Водночас перед тестуванням продуктивності виникає **перше інженерне завдання** – протестувати рівень ємності та ефективності масштабованих системах. Здійснити таке тестування, використовуючи один, навіть дуже великий, сервер стає все складніше. Ресурсів однієї машини просто не вистачає.

До того ж, часто користувачами сучасних інформаційних систем є люди (чи інші системи) з різних куточків планети. Наприклад, ви можете придбати товари у онлайн-магазині Amazon, практично, перебуваючи на будь-якому континенті Землі, в першу чергу, завдяки поширенню мережі інтернет. Ця широка доступність ставить **друге інженерне завдання** –

здійснити тестування продуктивності з різних частин планети. Таке тестування дозволить зробити оцінку продуктивності, симулюючи реальні сценарії використання інформаційної системи (бодай це онлайн магазин де кінцевими споживачами є люди, чи платіжна система де такими виступають інші системи). У результаті, зібрані метрики та заміри максимально відображатимуть реальну ситуацію, що звісно дозволить уникнути багатьох ризиків та припущень.

Як було сказано вище, користувачами систем є різні люди чи інші системи з різних регіонів планети. Усі вони використовують пристрої з різними IP адресами для доступу до інформаційних систем. Це, в свою чергу, впливає на механізми збереження сесій та кешування даних у самих цифрових системах. Тому, навантаження лише з однієї машини виключає можливість повноцінно перевірити згадані механізми і досягти симуляції реальної ситуації, адже все тестування відбуватиметься з однієї IP адреси. Це ставить **третє інженерне завдання** перед тест-інженерами продуктивності – симулювати максимально наближене навантаження до реальних умов.

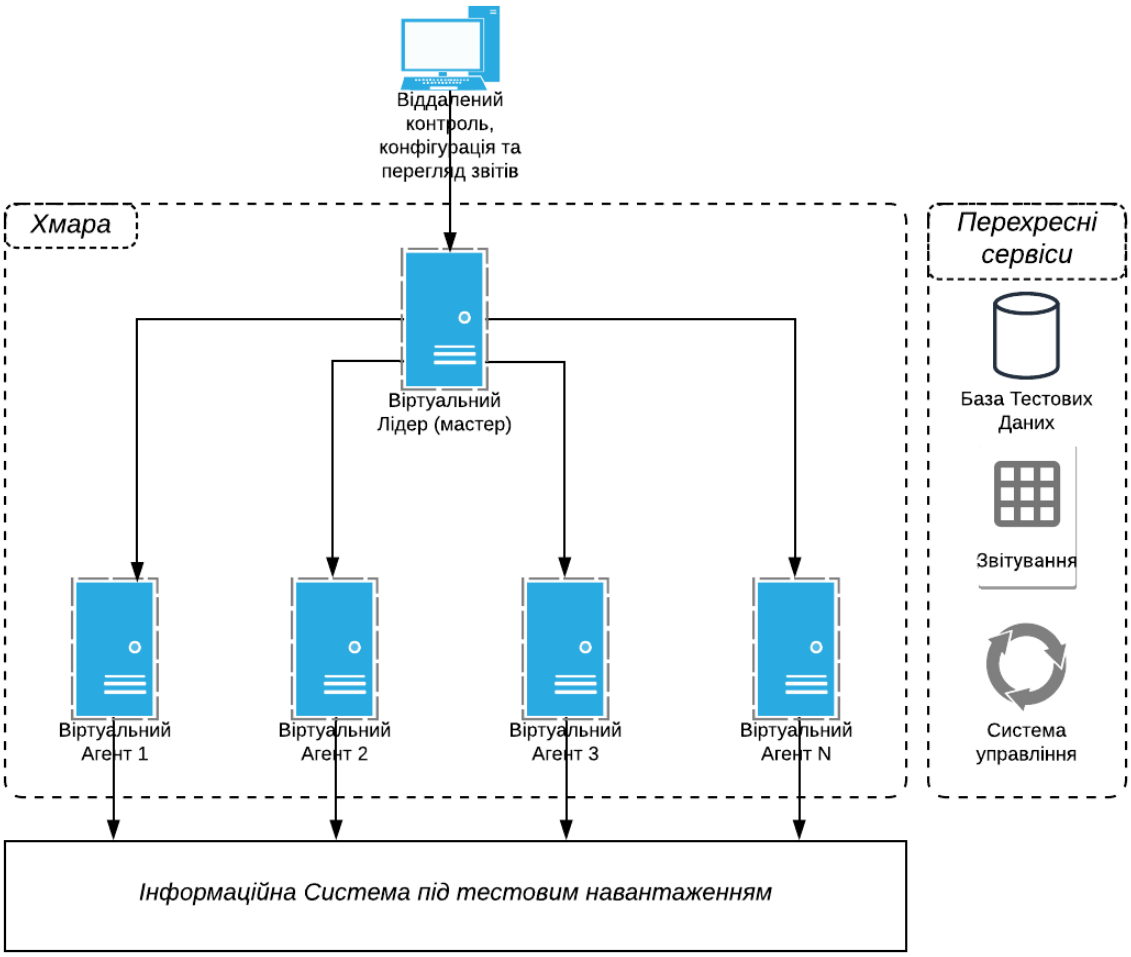
Для вирішення описаних вище викликів та завдань, інженери з тестування продуктивності використовують ті ж підходи, що і розробники самих інформаційних систем. Як було уже згадано в попередньому розділі 2.1, система для навантажувального тестування будується на тих же принципах і з використанням тих же технологічних та архітектурних підходів як і будь-яка інша інформаційна система. Саме тому, на допомогу у цій ситуації приходить розподілена система для тестування навантаження.

За визначенням компанії SmartBear **розподілене навантажувальне тестування** є нічим іншим як тестом, проведеним з декількох машин (комп’ютерів, серверів) одночасно, що дає змогу симулювати велику кількість віртуальних одночасних користувачів та генерує великі об’єми трафіку [2].

Найчастіше, компоненти архітектури системи розподіленого навантаження включають:

- постачальник хмарних технологій;
- машина-мастер для оркестрації та управління;
- машини-агенти для виконання самого навантаження;
- інструмент для розробки та симуляції навантаження;
- інструменти для збору та відображення результатів;
- за потреби: системи управління та/або безперервної поставки коду;
- за потреби: база для збереження тестових даних.

Базова архітектура даної системи матиме подібний вигляд до зображеної схеми на Діаграмі 2.2.1.



Діаграма 2.2.1 - Базова архітектура розподіленої системи генерації навантаження

Для реалізації подібної системи на ринку існують як комерційні так і готові загальнодоступні. Нижче, у Таблиці 2.2.2, наведена порівняльна таблиця найбільш поширених рішень.

Таблиця 2.2.2 Порівняльна таблиця розподілених систем генерації навантаження

Назва	Вартість на травень 2020 року	Рівень адаптації під проект	Звітування	Інтеграція з CI/CD
Blazemeter	від 6,000 у.о. / рік (5,000 віртуальних користувачів)	Високий, дає можливість використання різних інструментів	Так, свій формат. В реальному часі	Так
LoadUI	від 11,454 у.о. / рік (1,000 віртуальних користувачів)	Середній	Так, свій формат. В реальному часі	Так
LoadRunner	від 7,000 – 10,000 у.о. / рік (5,000 віртуальних користувачів)	Високий, дає можливість використання різних інструментів	Так, свій формат. В реальному часі	Так
Gatling Frontline	55,000 – 27,000 у.о. / рік (залежно від частоти використання)	Високий	Так	Так
AWS template [13]	Безкоштовно (оплата за використання віртуальних машин)	Нижче середнього, адже обмежений своїми технологіями	Так, але обмежене можливостями CloudWatch	Так

Швидкий порівняльний аналіз показав, що на ринку є досить широкий спектр продуктів для імплементації розподіленої системи тестування продуктивності. Водночас ці рішення не безкоштовні, обмежують в можливостях побудови необхідних звітів, пропонуючи лише свій формат та зменшують гнучкість проекту в цілому, зав’язуючи на свій сервіс.

Безкоштовне (лише плата за використання віртуальних сервісів) рішення від AWS використовує такий інструмент навантажувального тестування як Taurus, який дозволяє реалізувати лише доволі прості сценарії тестування. Також звітування результатів у сервісі CloudWatch є доволі не гнучким.

Саме тому, ціллю даної роботи є побудова власної системи розподіленого тестування навантаження, яка дозволить майже будь-якому проекту гнучко підійти як до процесу тестування продуктивності в цілому, так і конкретно до таких його аспектів як формат звітів, управління та інших.

2.3 Відображення результатів у реальному часі

Зазвичай, загальнодоступні інструменти для проведення тестування продуктивності (JMeter, Gatling та інші) генерують розширений звіт лише після закінчення виконання тестів. Протягом самого тестування, інженер може спостерігати досить обмежену кількість метрик, такі як кількість успішних чи ні запитів. Водночас для відслідковування поведінки системи під навантаженням дуже бажано мати змогу відслідковувати значно ширший спектр показників продуктивності системи та інфраструктури. Особливо це актуально під час тривалих тестів (від однієї та більше годин). Є велика ймовірність того, що система проявить деградацію уже на початку тестування. У цій ситуації тест продовжувати недоречно і його потрібно зупиняти. Саме тому постає потреба в реалізації систем відображення результатів тестування в реальному часі.

Для імплементації подібної системи часто використовують один з наступних підходів:

- InfluxDB та Grafana (див. Рисунок 2.3.1);
- Elasticsearch, Logstash та Kibana (див. Рисунок 2.3.2) [19];
- самописні звіти.

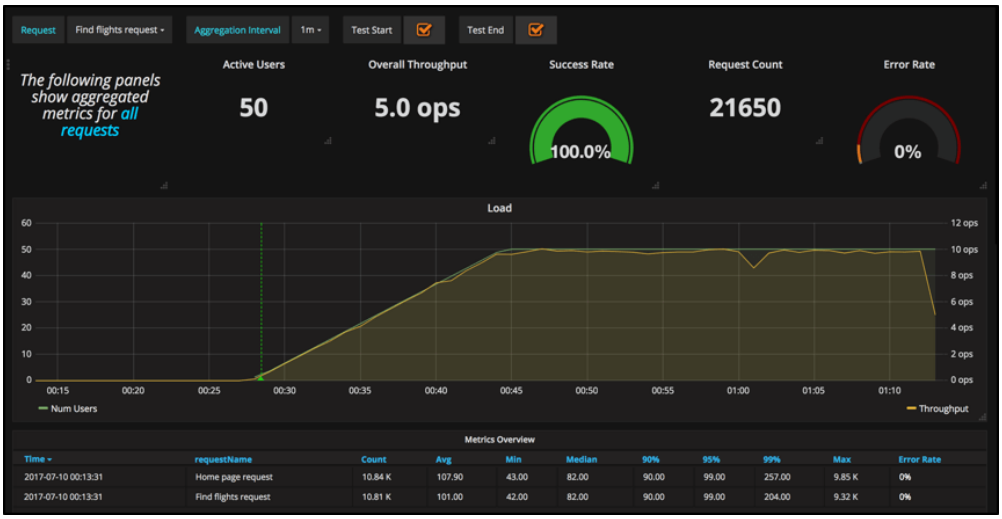


Рисунок 2.3.1 - Приклад звіту засобами InfluxDB та Grafana

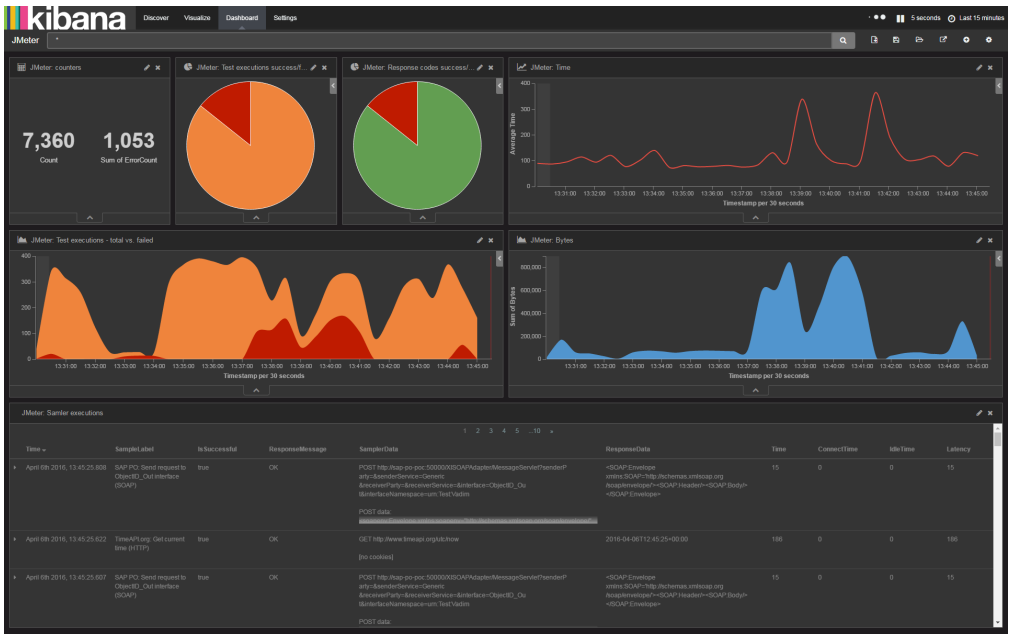


Рисунок 2.3.2 - Приклад звіту засобами Elasticsearch, Logstash та Kibana

Зазвичай, базова архітектура такої системи має наступний вигляд:

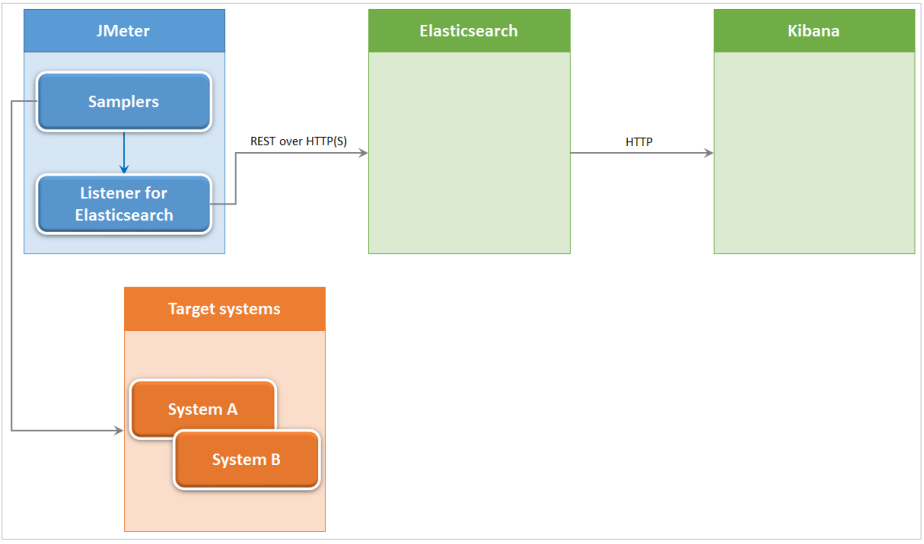


Рисунок 2.3.3 - Архітектура системи звітування у реальному засобами JMeter, Elasticsearch, Logstash та Kibana

Інше завдання яке вирішує такий підхід є агрегація та відображення результатів з різних машин розподіленої системи навантаження. У базовій конфігурації навантажувальний інструмент – JMeter чи Gatling - генеруватиме звіт на кожній машині окремо. Тобто, якщо під час тесту буде використано 3 сервера, то в результаті інженер отримає 3 окремих звіти з тестування і буде змушений об’єднувати їх у ручному режимі що, звісно, є досить рутинним процесом. Саме тому використання технологій, описаних

вище, дозволяє обійти це обмеження та відобразити результати тестування з усіх машин централізовано в одному місці.

Тут потрібно зазначити, що усі згадані інструменти є загальнодоступними та можуть використовуватись безкоштовно на базовому рівні. Водночас є певні обмеження використання безкоштовних інструментів для розподіленого тестування. Так, для реалізації практичної частини даної роботи обраний такий інструмент навантажувального тестування як Gatling (на основі Scala). Даний інструмент чудово підходить для генерації навантаження з однієї машини але обмежує розподілене тестування. Так, було виявлено, що у випадку коли надіслані Gatling дані з різних машин мають однаковий індекс часу, то InfluxDB збереже лише останні. Детально цей механізм обмеження та його розв'язанням за допомогою інструменту з орбіти ELK буде показано у розділі 3.2 практичної частини курсової роботи.

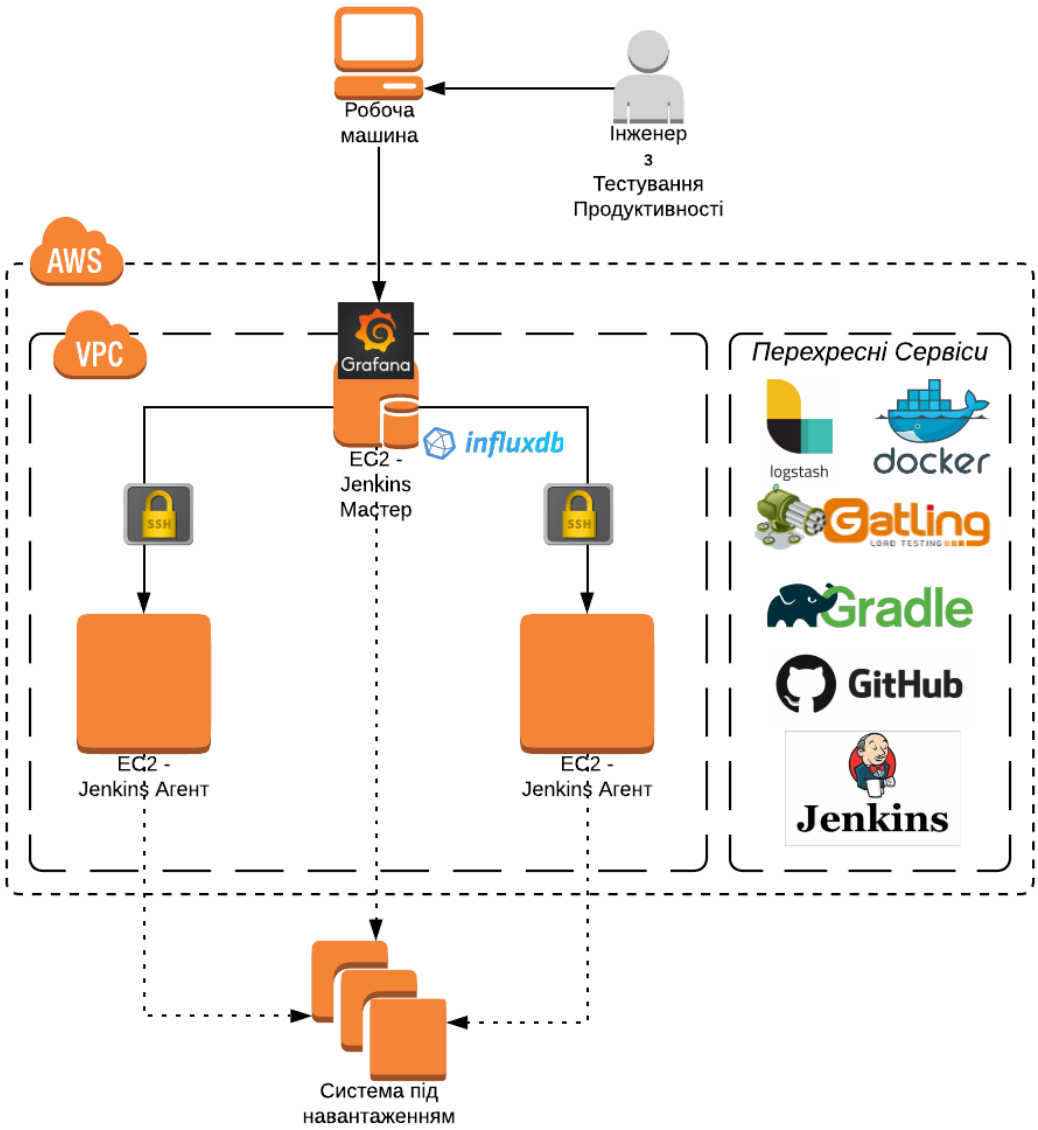
Підсумовуючи усе вищесказане про систему звітування результатів тестів у реальному часі, можна зазначите наступне:

- такий підхід **дає змогу відслідковувати результати під час виконання самого тесту** та зекономити значну частину часу у разі виявлення незадовільної динаміки;
- **дає змогу зібрати та відобразити централізовано результати тестів** розподіленої системи навантаження;
- власна система звітування **дозволяє визначати які та як показники відображати**, враховуючи можливість додавання метрик з використання ресурсів системи;
- описану систему звітування **можна побудувати з використанням загальнодоступних (безкоштовних) інструментів**.

РОЗДІЛ 3: Практична імплементація системи розподіленого навантаження

3.1 Опис та архітектура розподіленої системи тестування

Архітектура взаємодії компонентів системи розподіленого тестування навантаження та використані сервіси та інструменти для її побудови зображені на Діаграмі 3.1.1.



Діаграма 3.1.1 - Архітектура взаємодії компонентів системи розподіленого тестування навантаження

Усі інструменти, які були використані для реалізації системи тестування є загальнодоступними, а отже і безкоштовними, окрім Amazon AWS EC2 віртуальних серверів, а саме:

- **Jenkins** – сервер для автоматичної безперервної збірки, тестування та доставлення програмного продукту у загальний [9].

  Даний інструмент дав змогу налаштувати розподілену систему серверів завдяки Мастер-Агент архітектурі (як зображено на Діаграмі 3.1.1) та використовувати останню версію тестів зі сховища коду GitHub.
- **Gatling** – інструмент для навантажувального тестування [5].

  Перевагою інструменту, особливо у порівнянні з його основним конкурентом JMeter, є те, що розробка тестів здійснюється у програмному середовищі на мові Scala за підтримки інших мов JVM (Java, Groovy, Kotlin). Це дозволяє інтегрувати написані тести з системою GitHub та чітко відслідковувати версії коду. Також, у порівнянні з іншими інструментами, Gatling споживає значно меншу кількість системних ресурсів, особливо оперативної пам’яті, завдяки технологіями Akka [3] та Netty [11].
- **Gradle** – система для автоматичної збірки та запуску проектів [6].

  У даному проекті використовується саме для збірки та запуску Gatling тестів.
- **Docker** – сервіс віртуалізації [4].

  Даний сервіс був використаний для запуску Logstash, InfluxDB та Grafana у комплексі на основі docker-compose.
- **InfluxDB** – нбаза даних основою якої є часовий ряд [8].

  Ця база є централізованою у системі і саме у ній відбувається зберігання результатів тестів. Водночас Grafana сервіс використовує дані бази як ресурс для відображення результатів тестів.
- **Grafana** – сервіс для відображення метрик та попередження у разі виходу за встановлені рамки [7].
- **Logstash** – інструмент з орбіти ELK для обробки, необхідної видозміни та подальшого відправлення даних [10].

Саме завдяки Logstash стало можливим обійти обмеження інструменту Gatling для тестування з розподілених машин, що в деталях буде розглянуто далі.

3.2 Побудова системи відображення результатів у реальному часі

Інструмент для навантаження Gatling має змогу надсилати дані в реальному часі прямо в базу **InfluxDB**, використовуючи протокол graphite. Для цього, в конфігураційному файлі інструмента gatling.conf необхідно змінити налаштування до наступних:

```
gatling {
  data {
    writers = [console, file, graphite]
    graphite {
      light = false
      host = "localhost"
      port = 2003
      protocol = "tcp"
      rootPathPrefix = "gatling"
      bufferSize = 8192
      writePeriod = 1
    }
  }
}
```

Водночас Gatling надсилає дані з інтервалом у 1 секунду, що є обмеженням для системи розподіленого навантажувального тестування. Щоб зрозуміти чому, потрібно спочатку розглянути структуру даних InfluxDB на гіпотетичному прикладі збереження температурних показників локації у таблицю (measurement) weather.

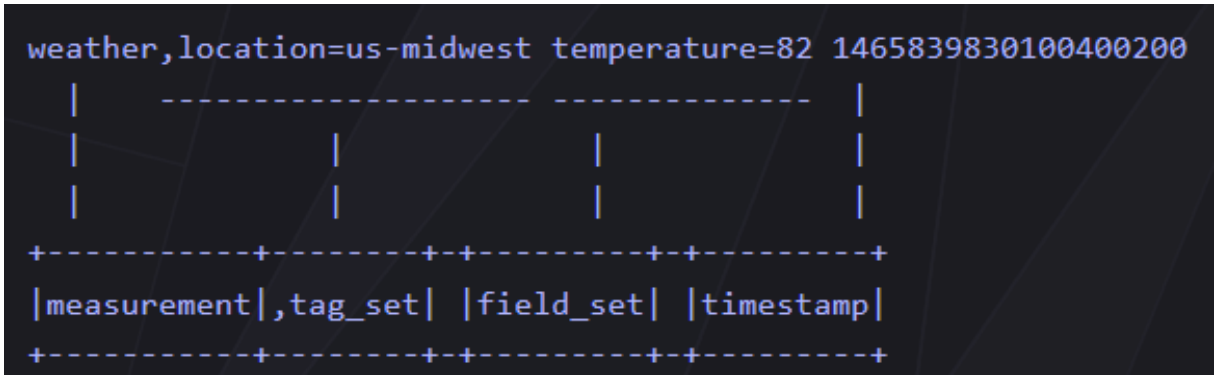


Рисунок 3.1.1 - Структура даних InfluxDB

- measurement – аналог таблиці у SQL;
- tag(s) – аналог проіндексовані значень у SQL;

- field(s) – аналог не проіндексованих значень у SQL;
- timestamp – значення часу в юнікс форматі. Основа основ для InfluxDB. Саме це значення є ключовим індексом для кожного запису.

Так ось, якщо для обох чи більше записів поля timestamp та tag(s) співпадають, незважаючи на значення field(s), то InfluxDB обробить ці записи як ідентичні та збереже лише останні значення fields. В свою чергу, Gatling надсилає дані таким чином, що час має розмірність лише у секундах, а усі інші значення (назва сторінки, сценарію) є тагами (tags), окрім самого значення заміру (наприклад, час відгуку), який є полем (field). В результаті, виникає ситуація, коли ми втрачаємо частину даних з розподілених серверів навантаження оскільки вони перезаписуються у базі (див. Рисунок 3.1.2).

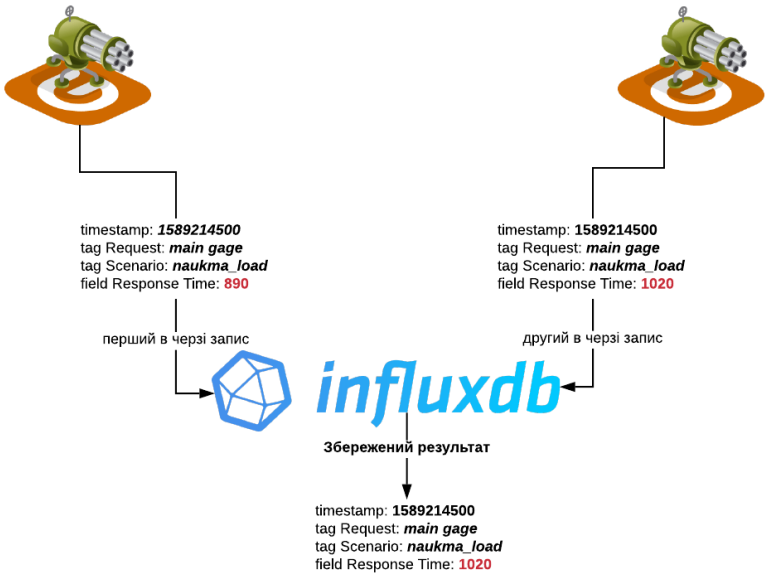


Рисунок 3.1.2 - Схема збереження результатів Gatling у InfluxDB при збігові timestamp та fields

Gatling не дозволяє додавати нові поля (tags, fields) чи змінювати розмірність часу на мілісекунди, що могло б допомогти у розв’язанні даного обмеження. З іншої сторони, інструмент має відкриту кодову базу і можна було б спробувати змінити саму логіку відправлення даних всередині кодової бази. Але такі дії можуть призвести до ускладнень синхронізації при появі нових версій продукту. Саме тому було вирішено знайти стороннє рішення для розв’язання описаного обмеження.

І таким технічним рішенням став загальнодоступний інструмент від Elasticsearch – Logstash. Він зайняв позицію на шляху руху даних між Gatling та InfluxDB, додаючи ім’я машини (у нашому випадку ім’я Docker контейнера) як додатковий таг. Logstash запускається на кожній машині та надсилає дані в централізовану базу InfluxDB (див. Рисунок 3.1.3).

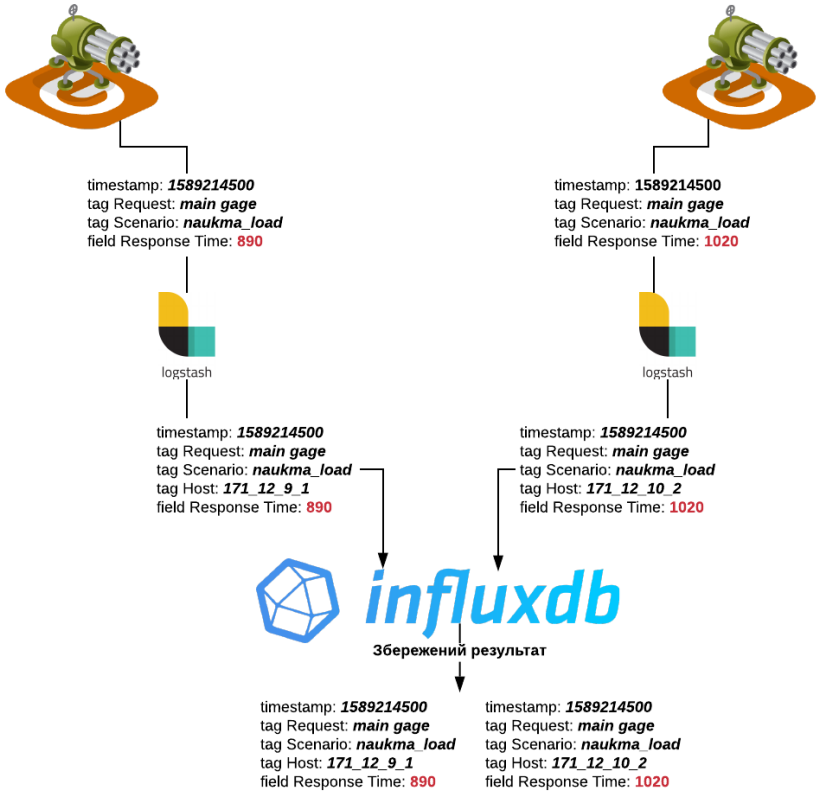


Рисунок 3.1.3 - Схема збереження результатів Gatling у InfluxDB за посередництва Logstash

Нижче наведена конфігурація Logstash, яка і додає ім’я машини на шляху руху результатів під час проведення тестування:

```
input {
  graphite {
    port => 2003
  }
}

filter {

  ruby {
    init => "require 'socket'"
    code => "
      host = Socket.gethostname
      # replace all dots in host
      if host.include? '.'
        host = host.gsub('.', '_')
      end

      msg = event.get('message')
      metricsMatches = msg.scan(/gatling\S+ [0-9]+ [0-9]+/)
```

```
      event.to_hash.keys.each do |field|
        if (field != '@timestamp') and (field != 'message')
          event.remove(field)
        end
      end

      timestamp = 0
      metricsMatches.each do |metric|
        metric_splitted = metric.split(' ')
        event.set(metric_splitted[0].concat('.').concat(host),
metric_splitted[1])
        timestamp = metric_splitted[2]
      end
      event.set('timestamp', timestamp)
    "
  }

  date {
    match => [ "timestamp", "UNIX"]
    remove_field => ["timestamp"]
  }

  mutate {
    remove_field => ["message"]
  }
}

output {
  graphite {
    host => "18.197.196.197" # IP of InfluxDB
    port => 9003
    fields_are_metrics => true
  }
}
```

Також, для того, щоб зберігати у InfluxDB надіслані дані у зручному для зчитування сервісом Grafana форматі, необхідно доповнити конфігурацію самої бази (influxdb.conf) наступною частиною, яка змінюватиме дані у потрібний формат:

```
[[graphite]]
  # Determines whether the graphite endpoint is enabled.
  enabled = true
  database = "graphite"
  retention-policy = ""
  bind-address = ":9003"
  protocol = "tcp"

  # Flush if this many points get buffered
  batch-size = 5000

  # number of batches that may be pending in memory
  batch-pending = 10

  # Flush at least this often even if we haven't hit buffer limit
  batch-timeout = "1s"

  # UDP Read buffer size, 0 means OS default. UDP listener will fail if set above
OS max.
  udp-read-buffer = 0
  separator = "."
```

```
  ### Each template line requires a template pattern.
  templates = [
    "gatling.*.*.*.count.* measurement.simulation.request.status.field.host",
    "gatling.*.*.*.max.* measurement.simulation.request.status.field.host",
    "gatling.*.*.*.mean.* measurement.simulation.request.status.field.host",
    "gatling.*.*.*.min.* measurement.simulation.request.status.field.host",
    "gatling.*.*.*.percentiles50.*
measurement.simulation.request.status.field.host",
    "gatling.*.*.*.percentiles75.*
measurement.simulation.request.status.field.host",
    "gatling.*.*.*.percentiles95.*
measurement.simulation.request.status.field.host",
    "gatling.*.*.*.percentiles99.*
measurement.simulation.request.status.field.host",
    "gatling.*.*.*.stdDev.* measurement.simulation.request.status.field.host",
    "gatling.*.users.*.*.*
measurement.simulation.measurement.request.field.host",
    "gatling.*.*.*.*.*.ok.*.*
measurement.simulation.group1.group2.group3.request.status.field.host",
    "gatling.*.*.*.*.*.ko.*.*
measurement.simulation.group1.group2.group3.request.status.field.host",
    "gatling.*.*.*.*.*.all.*.*
measurement.simulation.group1.group2.group3.request.status.field.host",
    "gatling.*.*.*.*.ok.*.*
measurement.simulation.group1.group2.request.status.field.host",
    "gatling.*.*.*.*.ko.*.*
measurement.simulation.group1.group2.request.status.field.host",
    "gatling.*.*.*.*.all.*.*
measurement.simulation.group1.group2.request.status.field.host",
    "gatling.*.*.*.ok.*.*
measurement.simulation.group1.request.status.field.host",
    "gatling.*.*.*.ko.*.*
measurement.simulation.group1.request.status.field.host",
    "gatling.*.*.*.all.*.*
measurement.simulation.group1.request.status.field.host",
    "gatling.*.*.ok.*.* measurement.simulation.request.status.field.host",
    "gatling.*.*.ko.*.* measurement.simulation.request.status.field.host",
    "gatling.*.*.all.*.* measurement.simulation.request.status.field.host"
  ]
```

Після розв’язання проблеми з централізованим збереженням результатів у потрібному форматі, був реалізований сервіс відображення даних Grafana на основі бази InfluxDB, з усіма необхідними графіками та метриками продуктивності (див. приклад звіту у Додатку А).

Зручність Grafana полягає також у тому, що усі графіки (інформаційні панелі) можна зберігати в json форматі, що дозволяє створити їх один раз і потім використовувати щоразу при розгортанні системи розподіленого тестування навантаження на новому проекті. Даний файл показаний під назвою gatling_dashboard.json на Рисунку 3.2.2.

В результаті, загальна структура під-проекту для відображення даних у реальному часі, включаючи усі конфігурації, має наступний вигляд:

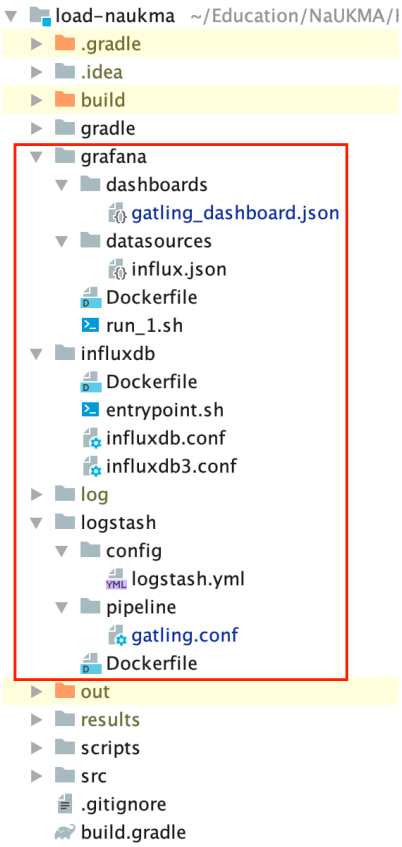


Рисунок 3.2.2 - Структура під-проекту Grafana, InfluxDB, Logstash

Даний під-проект увійшов в загальний проект для розподіленого тестування продуктивності. Для його запуску використовується сервіс docker-compose який відрізняється для Jenkins Master та Jenkins Agent (slave). Так у першому випадку запускається як Logstash так і Grafana та InfluxDB (як показано нижче зі змісту самого docker-compose файлу). Тоді як у другому – лише Logstash для ретрансляції результатів тестування у централізовану базу на Мастері.

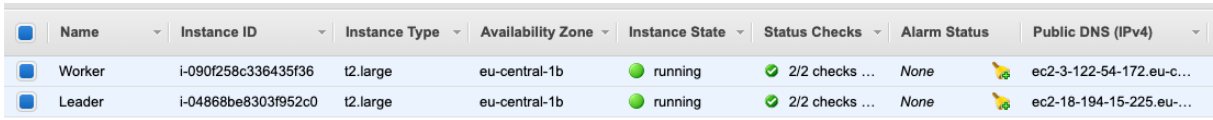
```
version: "3.7"

services:
  influxdb:
    build: influxdb
    env_file: configuration.env
    container_name: 'influxdb'
    ports:
      - '8086:8086'
      - '9003:9003'
    volumes:
      - influxdb_data:/var/lib/influxdb
    networks:
      - load-test

  grafana:
    build: grafana
    env_file: configuration.env
    container_name: 'grafana'
```

```
    links:
      - influxdb
    ports:
      - '3000:3000'
    volumes:
      - grafana_data:/var/lib/grafana
    networks:
      - load-test

  logstash:
    build: logstash/
    container_name: 'logstash'
    volumes:
      - type: bind
        source: ./logstash/config/logstash.yml
        target: /usr/share/logstash/config/logstash.yml
        read_only: true
      - type: bind
        source: ./logstash/pipeline
        target: /usr/share/logstash/pipeline
        read_only: true
    ports:
      - "2003:2003"
    depends_on:
     - grafana
     - influxdb
    environment:
      LS_JAVA_OPTS: "-Xmx512m -Xms512m"
    networks:
      - load-test

volumes:
  grafana_data: {}
  influxdb_data: {}

networks:
  load-test:
```

3.3 Побудова розподіленої системи тестування на основі Jenkins Pipeline та AWS

Нижче подані основні кроки, які були зроблені для побудови розподіленої системи тестування продуктивності.

1. Створення двох віртуальних машин на базі AWS.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)
Worker	i-090f258c336435f36	t2.large	eu-central-1b	running	2/2 checks ...	None	ec2-3-122-54-172.eu-c...
Leader	i-04868be8303f952c0	t2.large	eu-central-1b	running	2/2 checks ...	None	ec2-18-194-15-225.eu-...

Рисунок 3.3.1 - EC2 віртуальні машини

2. Встановлення на одній з машин сервісу Jenkins за наступним алгоритмом (docker уже має бути встановлений)
   - Встановити JDK

     sudo apt-get install openjdk-8-jdk
   - Встановити Nginx веб-серверу

```
sudo apt-get install nginx
```

- Встановити Jenkins

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins-
ci.org.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-
stable binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt-get update

sudo apt-get install jenkins

sudo gpasswd -a jenkins docker
```

- Перейти на сам сервіс по адресу ip:8080 та здійснити базову конфігурацію

В подальшому, цей сервер відіграватиме роль Мастер (Leader) та керуватиме роботою агентів (Worker).

3. Підключення Jenkins Агента
   - Налаштувати з'єднання по SSH протоколу між Мастером та Агеном
     На Мастері виконати:

```
sudo -iu jenkins

ssh root@<slave_ip> mkdir -p .ssh

cat .ssh/id_rsa.pub | ssh root@<slave_ip> 'cat >>
.ssh/authorize
```

   - Встановити JDK на Агенті

```
sudo apt-get install default-jre
```

   - Скачати на Агент програму для запуску агента з Мастера

```
wget http://master-ip:8080/jnlpJars/slave.jar
```

   - Налаштувати Агент безпосередньо у Jenkins
     Jenkins > Manage Jenkins > Manage Nodes and Clouds > New Node та ввести дані, показані на Рисунку 3.3.2, підставивши при цьому вірний IP адрес Агента.

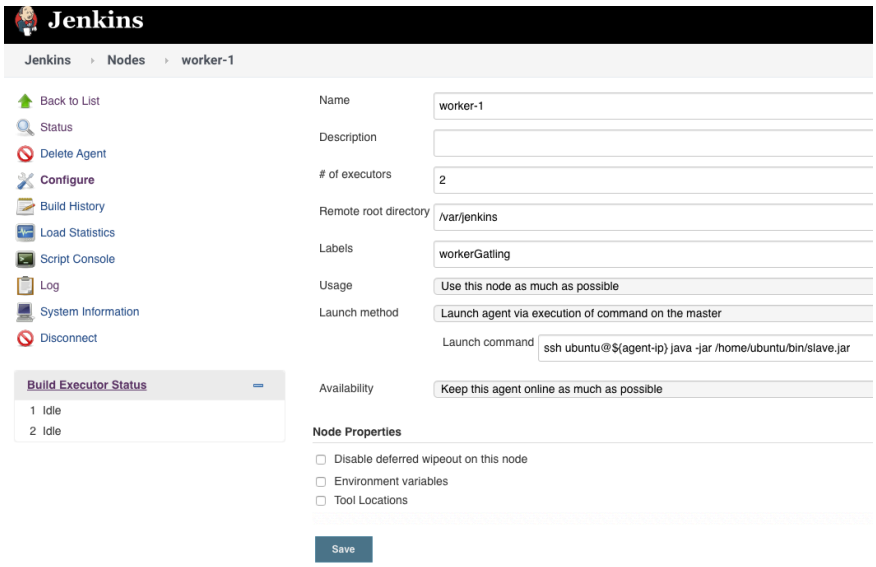


Рисунок 3.3.2 - Налаштування Агента у Jenkins

Після цього у системі Jenkins має відбутися підключення та відображення Агенту (необхідно трохи зачекати).

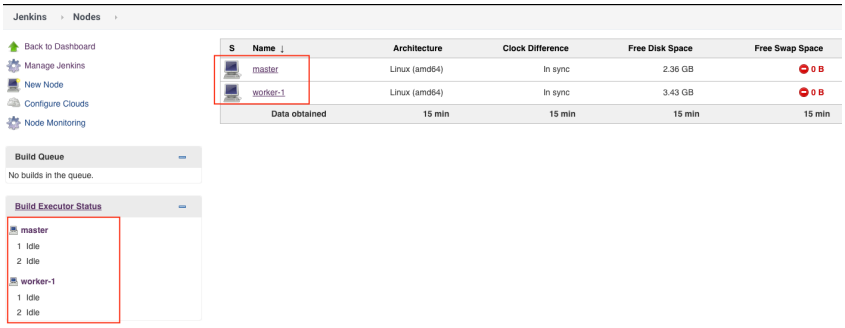


Рисунок 3.3.3 - Відображення підключеного Агента в Jenkins

4. Створення Pipeline Job у Jenkins та перевірка роботи.

   Все що потрібно зробити, це додати шлях до Jenkinsfile, який зберігається у сховищі GitHub.

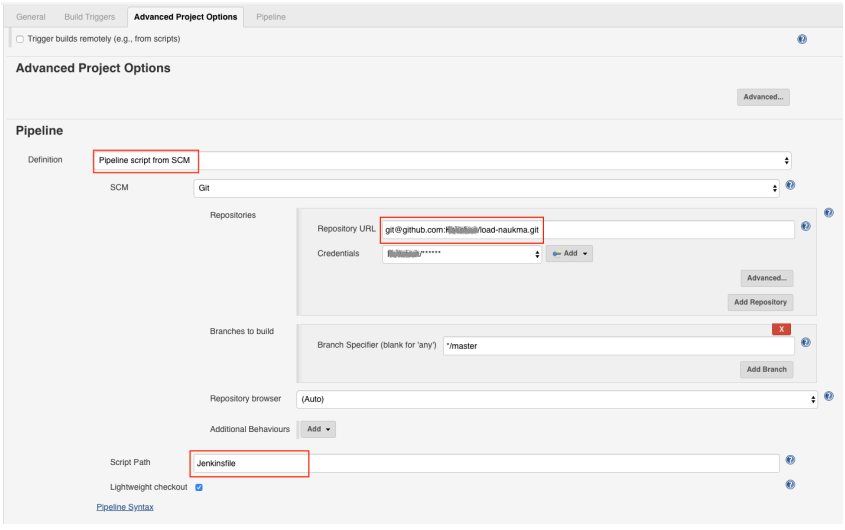


Рисунок 3.3.4 - Створення Pipeline Job у Jenkins

Слід зазначити декілька слів про сам Jenkinsfile. Так, сервіс Jenkins дозволяє створювати Завдання (Job) трьома різними способами:

1. Безпосередньо через веб сторінку сервісу

   Такий підхід не є зручним якщо необхідно часто змінювати та контролювати версії самих Завдань (Jobs). Саме тому, були розроблені наступні два підходи.

2. Декларативний - через Jenkinsfile, використовуючи специфічну мову Jenkins DSL (Domain Specific Language).
3. Скриптовий - через той самий Jenkinsfile, але використовуючи виключно скриптовий підхід на основі мови програмування groovy.

Саме останній, третій підхід був використаний під для виконання практичної частини даної роботи. Зміст Jenkinsfile поданий нижче.

```
def labels = ['workerGatling', 'master'] // labels for Jenkins node types we will
build on
def builders = [:]

for (x in labels) {
    def label = x
    builders[label] = {
      node(label) {
          logRotator{
              daysToKeep(1)
              numToKeep(5)
              artifactDaysToKeep(-1)
              artifactNumToKeep(-1)
          }
          properties(
              [
                  parameters(
                      [
                          string(defaultValue: 'master', name: 'BRANCH_NAME'),
                          string(defaultValue: 'FinSimulation', name:
'TEST_SCRIPT_NAME'),
                          string(defaultValue: 'build.sh', name: 'BUILD_SCRIPT')
                      ]
                  )
              ]
          )
          stage('Checkout') {
              echo "Workspace >>> ${WORKSPACE}"
              cleanWs()

              checkout scm:
                  [
                      $class: 'GitSCM',
                      userRemoteConfigs:
                          [
                              [
                                  url: "git@github.com:yyy/load-naukma.git",
                                  credentialsId: "yyy"
                              ]
                          ],
                      branches:
```

```
                            [
                                [
                                    name: "${BRANCH_NAME}"
                                ]
                            ]
                    ], poll: false

                try {
                        sh "chmod -R 777 ${WORKSPACE}"
                    } catch (e) {
                        echo 'Error!!!'
                }
            }

            stage('Run') {
                System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "")

                try {
                    echo "WORKSPACE: ${WORKSPACE}"

                    sh '${WORKSPACE}/scripts/${BUILD_SCRIPT}'

                    currentBuild.result = 'SUCCESS'
                } catch (Exception e) {
                    currentBuild.result = 'FAILURE'
                }
            }

            stage("Publish report") {
                gatlingArchive()
                echo "Gatling report published"

                archiveArtifacts artifacts: 'log/gatling*.log', fingerprint: true
             }
        }
    }
}

parallel builders
```

3.4 Побудова тестових сценаріїв на основі інструменту Gatling

Як уже було згадано раніше, для побудови тестів Gatling використовує функціональну мову програмування Scala. Це надає можливість контролювати версії тестів (наприклад, через GitHub). Для прикладу, основний конкурент JMeter немає такої можливості.

Ще однією вагомою перевагою є те, що Scala належить до орбіти мов JVM (Java, Groovy, Kotlin). Це дає можливість писати різного роду утиліти для генерації та зберігання тестових даних чи реалізації необхідної особливої логіки за допомогою цих мов під шапкою одного тестового проекту.

Розглянемо для прикладу написання тестового навантажувального сценарію для сайту http://fin.ukma.edu.ua/.

Кроки сценарію мають наступний вигляд:

Рисунок 3.4.1 - Сценарій виконання тесту для http://fin.ukma.edu.ua/

Для написання такого тесту, було використано 3 модулі.

1. baseConfig/BaseSimulation.scala – конфігурація протоколу запитів
2. requestObjects/FinRequests.scala – опис логіки основних кроків на сайті
3. FinSimulation.scala – послідовність тестових кроків та опис моделі навантаження

Сам тест був виконаний на основі Page Object Pattern, який зазвичай використовується для функціонального тестування. Структура проекту має такий вигляд:

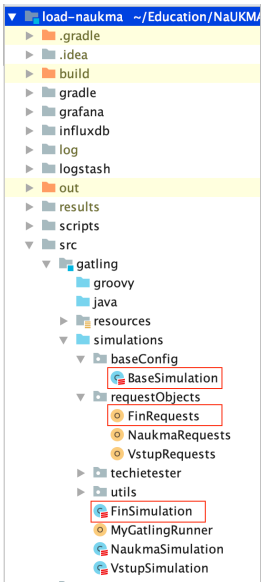


Рисунок 3.4.2 - Структура проекту для http://fin.ukma.edu.ua/

Розглянемо детальніше кожен з модулів.

baseConfig/BaseSimulation.scala

```
package baseConfig

import io.gatling.http.protocol.HttpProtocolBuilder
import io.gatling.core.Predef._
import io.gatling.http.Predef._

class BaseSimulation extends Simulation {

  val httpConfFin: HttpProtocolBuilder = http
    .baseUrl("http://fin.ukma.edu.ua")
    .inferHtmlResources(WhiteList("""".*ukma.edu.*"""), BlackList())

.acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,i
mage/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9")
```

```
    .acceptEncodingHeader("gzip, deflate, br")
    .acceptLanguageHeader("en,en-US;q=0.9,ru-RU;q=0.8,ru;q=0.7,uk;q=0.6")
    .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36")
    .silentResources
}
```

requestObjects/FinRequests.scala

```
package requestObjects

import io.gatling.core.Predef._
import io.gatling.http.Predef._

object FinRequests {

  def finHome() =
    exec(
      http("Fin Main Page")
        .get("/")
        .check(status.is(200))
    )

  def finNews() =
    exec(
      http("Fin News")
        .get("/news/")
        .check(status.is(200))
    )

  def finDepartment() =
    exec(
      http("Fin Department")
        .get("/department/informatics/description/")
        .check(status.is(200))
    )

  def finDecanat() =
    exec(
      http("Fin Decanat")
        .get("/about/history/")
        .check(status.is(200))
    )
}
```

FinSimulation.scala

```
import baseConfig.BaseSimulation
import io.gatling.core.Predef._
import io.gatling.core.structure.ScenarioBuilder
import requestObjects._
import utils.Utils._

import scala.concurrent.duration._

class FinSimulation extends BaseSimulation{

  /*** Before ***/
  before {
    println("Load Simulation is about to start!")
    debugInfo(false)
  }

  val finWebLoadScenario: ScenarioBuilder = scenario("FinUKMA Web Load")
    .during(120 seconds) {
      exec(FinRequests.finHome())
```

```
        .pause(1,2)
        .repeat(2)(
          exec(FinRequests.finNews())
            .pause(3,5)
        )
        .repeat(2) (
          exec(FinRequests.finDepartment())
            .pause(3,5)
        )
        .exec(FinRequests.finDecanat())
        .pause(1,2)
    }

  /*** Open Setup Load Simulation ***/
  setUp(
    finWebLoadScenario.inject(
      atOnceUsers(10) during (120 seconds)
    ).protocols(httpConfFin)
  ).assertions(global.successfulRequests.percent.is(100))

    after {
      println("Load Simulation is finished!")
    }
}
```

Як видно з модулів сценаріїв, перевірка на успішність навантажувального тесту здійснюється як мінімум у двох місцях:

- На рівні кожного запиту – засобом `.check(status.is(200))`
- На рівні усього тесту із встановленням граничного значення успішно здійснених запитів протягом усього тестування – засобом `.assertions(global.successfulRequests.percent.is(100))`

3.5 Тестування навантаження сайтів КМА та звіт

Під час випробовування розподіленої системи навантажувального тестування було використано 2 віртуальних машини AWS EC2 одна з яких відігравала роль Jenkins Master, інша – Jenkins Agent. Віртуальні сервера мали такі технічні параметри:

- 2 CPU, 8 MB RAM, SSD Volume, Ubuntu Server 18.04 LTS.

Само тестування навантаження відбувалось для наступних сайтів:

1. https://www.ukma.edu.ua/
2. https://vstup.ukma.edu.ua/
3. http://fin.ukma.edu.ua/

Було проведено більше ніж 20 різного типу тестів на продуктивність з використанням системи безперервної поставки коду Jenkins, як показано на рисунку 3.5.1.

Саме тестування було здійснено в нічний час для уникнення похибок від користування ресурсами реальними користувачами.

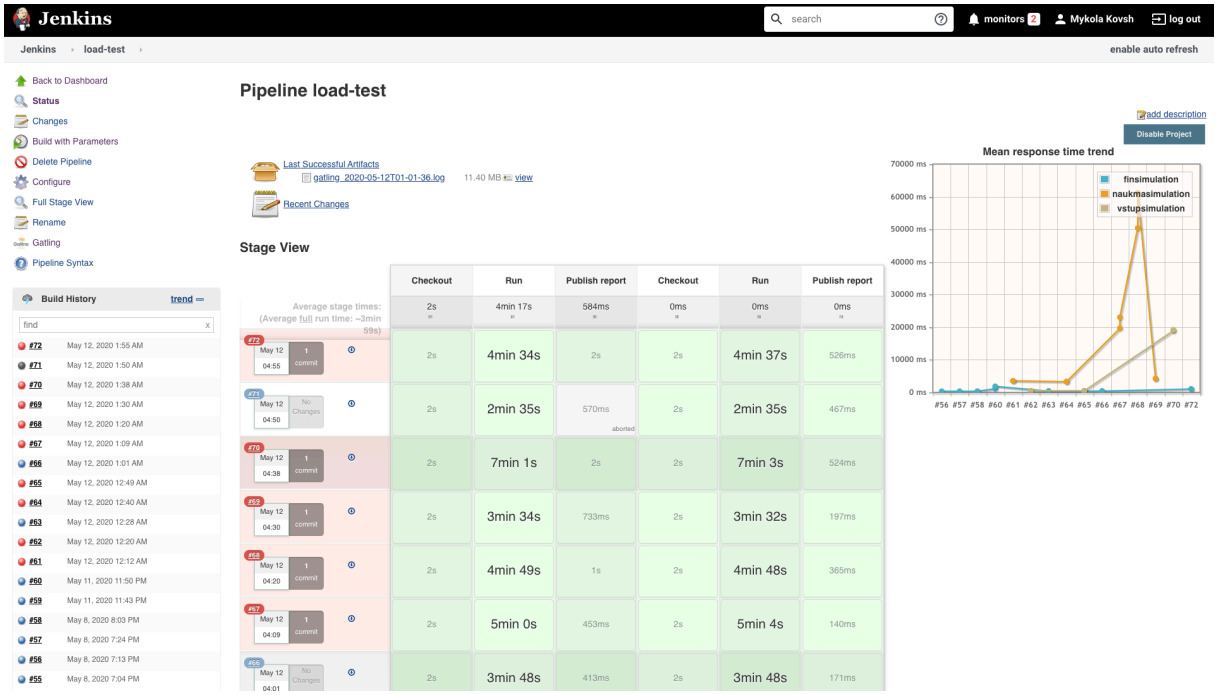


Рисунок 3.5.1 - Управління тестами через Jenkins

Отже, розглянемо результати навантажувального тестування більш детально.

1. **Порівняльне тестування для усіх трьох систем НаУКМА**
   - Було здійснено навантаження у 10 одночасних користувачів протягом трьох хвилин для кожного сайту.
   - За результатами порівняння можна зробити висновок, що на відносно малих навантаженнях, найкращі показники продуктивності має сайт https://vstup.ukma.edu.ua/ (див. Рисунок 3.5.2 та Таблиця 3.5.3)

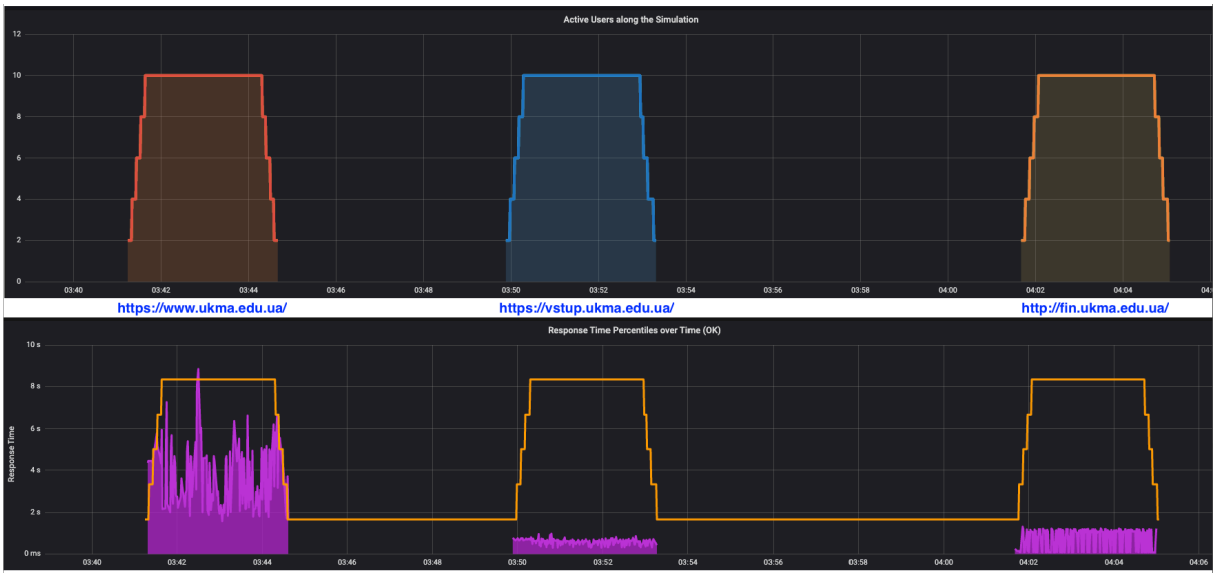


Рисунок 3.5.2 - Порівняння часу відгуку для трьох сайтів НаУКМА

Таблиця 3.5.3 Порівняння показників продуктивності

Сайт	Навантаження, одночасних користувачі	Максимальний час відгуку, 95% персентиль	Запитів в секунду
https://www.ukma.edu.ua/	10	8,830 мс	0,8
https://vstup.ukma.edu.ua/	10	963 мс	1
http://fin.ukma.edu.ua/	10	1,327 мс	0,8

2. **Тестування на ємність сайту www.ukma.edu.ua**

- Навантаження у 100 одночасних користувачів протягом 5-ти хвилин.
- Уже на другу хвилину тестування спостерігалась значна деградація у часі відгуку. Так, загальний час відгуку (95% персентиль) збільшився з 4 секунд до більше ніж 3 хвилини (як показано на Рисунку 3.5.5)
- Дистрибуція часу відгуку для вдалих запитів мала вигляд як відображено на Рисунку 3.5.6.
- Протягом тестування сайт переставав відповідати та показував помилку, яка відображала проблеми з доступом до бази даних (детальніше в Додатку Б).

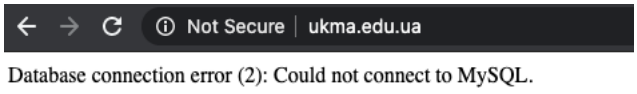


Рисунок 3.5.4 - Помилка на www.ukma.edu.ua при навантаженні

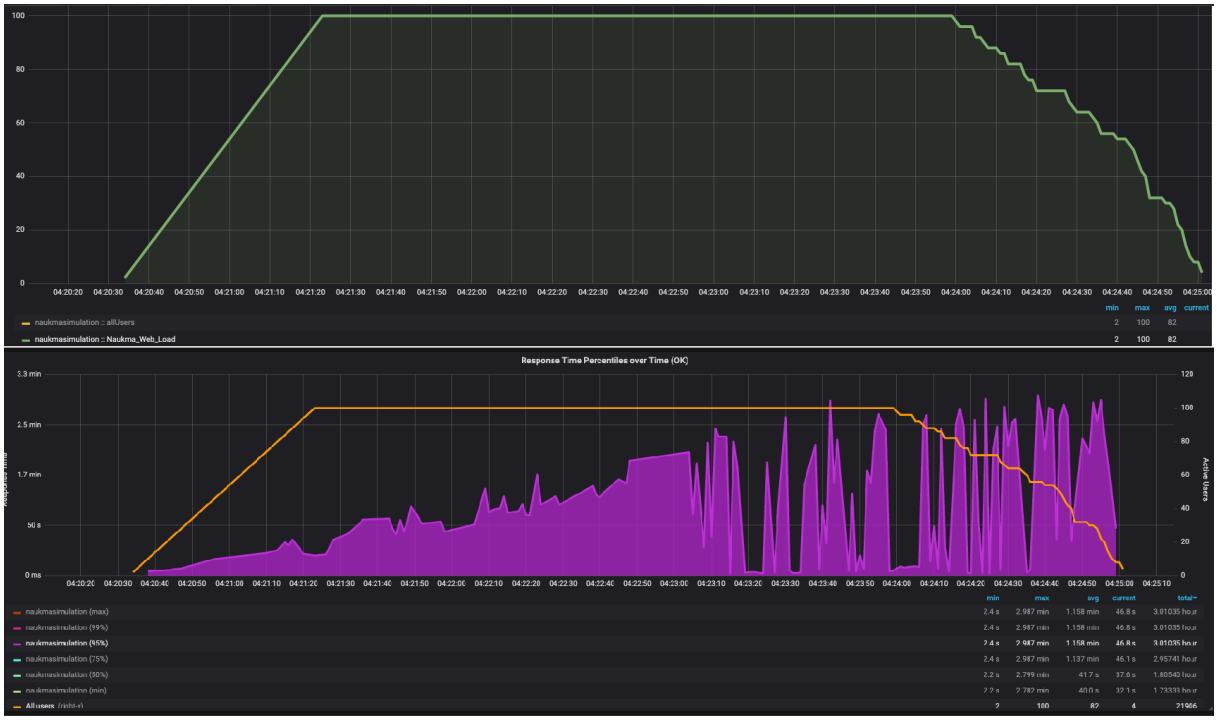


Рисунок 3.5.5 - Модель навантаження та час відгуку для www.ukma.edu.ua

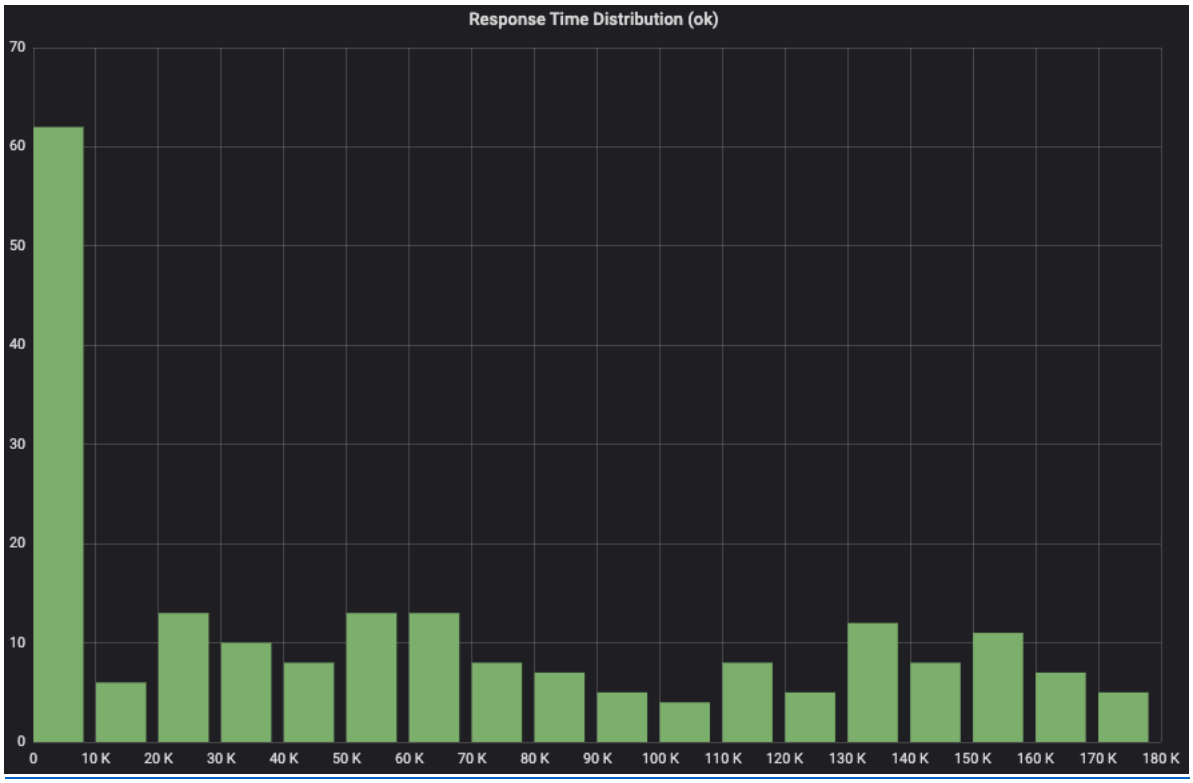


Рисунок 3.5.6 - Розподілення часу відгуку для www.ukma.edu.ua

- Найбільш «важкою» виявилась сторінка з історією КМА (більш детальна статистика у Додатку В).

Таблиця 3.5.7 95% персентиль час відгуку для сторінок www.ukma.edu.ua

Ім'я сторінки	Запит	Час відгуку
Головна	/	96 секунд
Історія КМА	/index.php/about-us/istoriya-akademiji	170 секунд
Факультети	/index.php/osvita/fakulteti	27 секунд
Підготовка до ЗНО	/index.php/abiturientu/pidhotovka	35 секунд
Контакти	/index.php/kontakti	8 секунд

3. Тестування на ємність сайту http://fin.ukma.edu.ua/

- Так як цей сайт показав себе значно краще ніж www.ukma.edu.ua, була здійснена перша спроба тестування на 400 одночасних користувачів. Але уже на 50 користувачах почали з'являтися помилки 502 Gateway.
- Із зростанням навантаження їх ставало все більше (Рисунок 3.5.8, Додаток Г), тому було прийняте рішення зупинити тестування (перевага звітності в реальному часі описана в розділі 2.3 цієї роботи).
- Досягши пропускної здатності у 3-4 операції в секунду, сайт залишався на цьому рівні, при зростаючій кількості помилок.

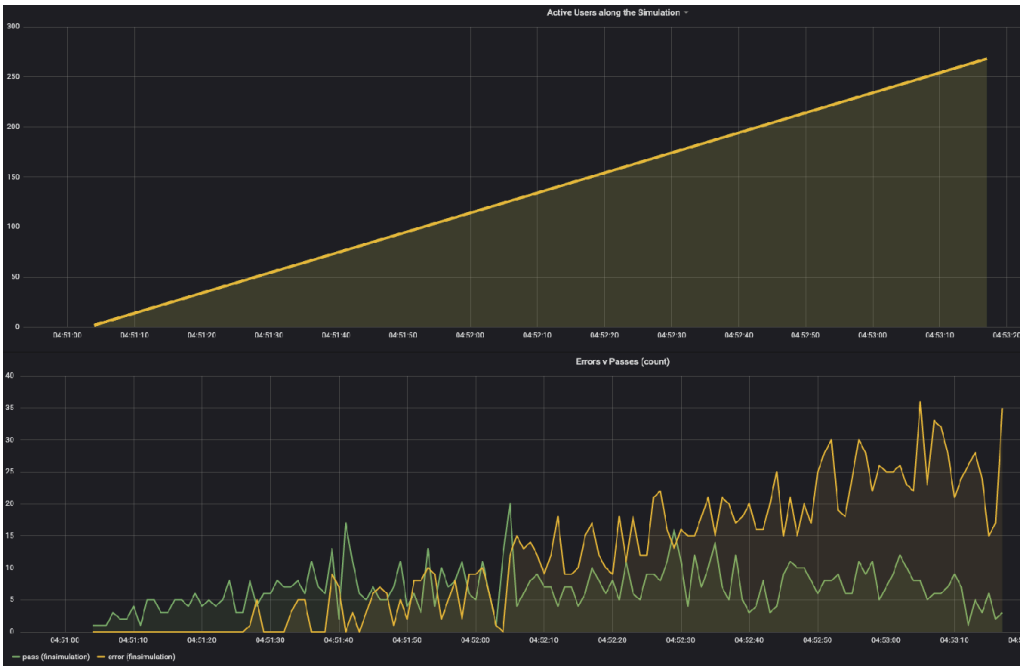


Рисунок 3.5.8 - Модель навантаження, та успішні/неуспішні запити для http://fin.ukma.edu.ua

- Не зважаючи на помилки, загальний час відгуку залишався відносно не високим, та не перевищив 6-ти секунд.

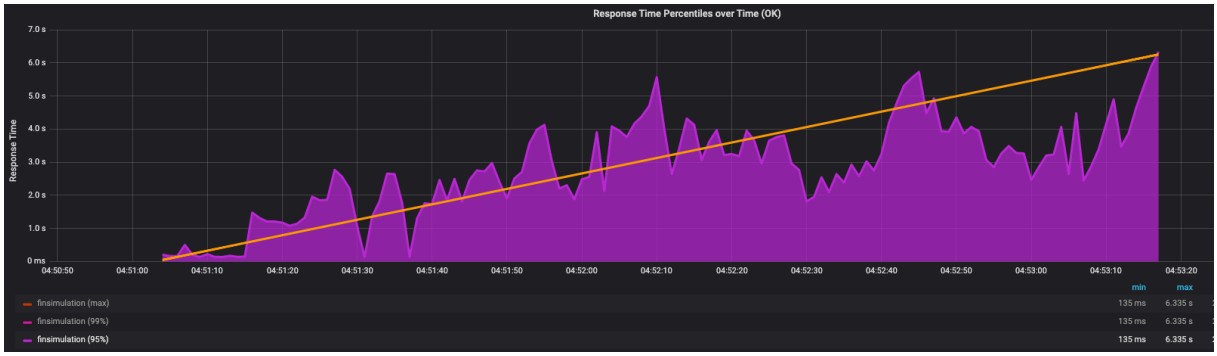


Рисунок 3.5.9 - Час відгуку під час навантаження для http://fin.ukma.edu.ua

- При другій спробі, було здійснене навантаження у 100 одночасних користувачів. Другий тест підтвердив результати попереднього тесту.
- Найбільш «важкою» виявилась сторінка з Кафедрами (більш детальна статистика у Додатку Д).

Таблиця 3.5.10 95% персентиль час відгуку для сторінок http://fin.ukma.edu.ua

Ім'я сторінки	Запит	Час відгуку
Головна	/	2,5 секунди
Новини	/news	2,5 секунди
Кафедри	/department/informatics/description/	4 секунди
Деканат	/about/history/	2,3 секунди

Якщо порівнювати http://fin.ukma.edu.ua та www.ukma.edu.ua то необхідно зазначити, що перший сайт має значно кращу швидкість обробки запитів але перестає працювати при 50+ одночасних користувачах (3-4 запита в секунду). Тоді як другий продовжує працювати при збільшенні навантаження, хоча час відгуку при цьому може складати більше ніж 3 хвилини. Також, були виявлені проблеми з базою даних для www.ukma.edu.ua, на які слід звернути увагу у подальшій підтримці сайту.

Висновки

Протягом написання даної курсової роботи був проаналізований широкий спектр інструментів та підходів для побудови розподіленої системи навантажувального тестування. Згідно поставлених цілей у вступі (та й самої назви), ця система мала б відповідати трьом основним вимогам:

1. Можливість інтеграції з системами безперервної поставки.
2. Розподілене навантажувальне тестування (з декількох машин).
3. Візуалізація результатів централізовано та в реальному часі.

В результаті проведених досліджень та пошуку рішень така система тестування продуктивності була успішно побудована та випробувана на декількох сайтах НаУКМА. Було проведено біля 20 різного типу тестів з навантаження.

Доречно зазначити, що окрім реалізації основних цілей, були досягнуті також і декілька інших, не менш важливих завдань.

Так, для побудови системи тестування використовуються загальнодоступні інструменти (open-source), окрім хмарних сервісів AWS, що дозволить зменшити бюджет проекту. Водночас якщо на проекті є власні потужності (сервера), то і останній пункт (AWS) не буде сильно затратним.

Додатково, система може бути швидко розгорнута у будь-якому операційному середовищі (Linux, Windows, Mac OS), адже для її побудови та запуску був використаний сервіс віртуалізації Docker.

Окрім того, поєднання сервісів InfluxDB та Grafana дозволяють гнучко керувати процесом візуалізації необхідних метрик та інтегрувати дані з різних ресурсів (метрики тестування, продуктивності та використання системи) в одному централізованому місці. Сама Grafana є нічим іншим як веб-сервісом відображення даних, тому доступ до результатів матиме не лише інженер навантажувального тестування, але і будь-який зацікавлений член команди (звісно, що через систему авторизації).

Література

1. Документація Jenkins Pipeline [Електронний ресурс] – 2020 – Режим доступу: https://www.jenkins.io/doc/book/pipeline/
2. Документація SmartBear. Distributed Load Testing [Електронний ресурс] – 2020 – Режим доступу: https://support.smartbear.com/readyapi/docs/loadui/distributed/index.html
3. Офіційний сайт Akka: https://github.com/akka/akka
4. Офіційний сайт Docker: https://www.docker.com
5. Офіційний сайт Gatling: https://gatling.io
6. Офіційний сайт Gradle: https://gradle.org
7. Офіційний сайт Grafana: https://grafana.com/
8. Офіційний сайт InfluxDB: https://www.influxdata.com
9. Офіційний сайт Jenkins: https://www.jenkins.io
10. Офіційний сайт Logstash: https://www.elastic.co/logstash
11. Офіційний сайт Netty: https://github.com/netty/netty
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. Distributed load testing on AWS [Електронний ресурс] – 2019 – Режим доступу: https://github.com/awslabs/distributed-load-testing-on-aws
14. Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed [Електронний ресурс] – 2018 – Режим доступу: https:/www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/
15. Graham, Bath and Rex Black. *Foundation Level Specialist Syllabus Performance Testing*. ISTQB, 2018. Print – C. 10 – 15
16. Ian Molyuneaux. *The Art of Application Performance Testing.* O'Reilly, 2015, Print – C. 5 – 10

17. 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

18. J.D. Meier. *Performance Testing Guidance for Web Applications: patterns and practices*. Microsoft Corporation, 2007. Print – С. 15 – 34

19. Load Testing with JMeter: Test Results Visualization Using Kibana Dashboards Pipeline [Електронний ресурс] – 2017 – Режим доступу: https://blogs.sap.com/2016/04/06/load-testing-with-jmeter-test-results-visualization-using-kibana-dashboards/

20. Page speed should concern marketers, but does it? [Електронний ресурс] – 2019 – Режим доступу: https://unbounce.com/page-speed-report/

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

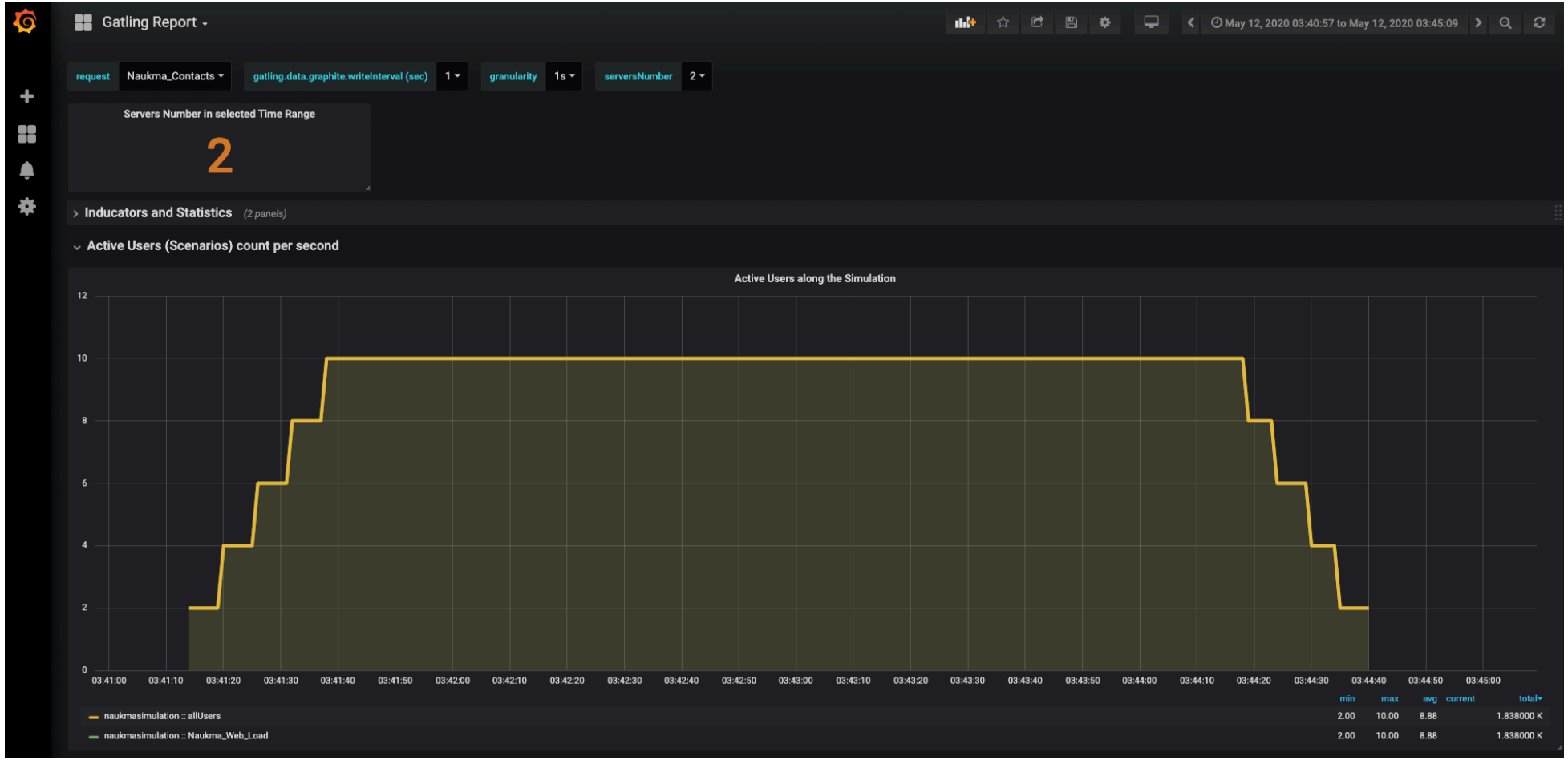
22. The Failed Launch Of www.HealthCare.gov. [Електронний ресурс] – 2016 – Режим доступу: https://digital.hbs.edu/platform-rctom/submission/the-failed-launch-of-www-healthcare-gov/

Додатки

Додаток А

(обов’язковий)

Приклад звіту в реальному часі на основі інструментів InfluxDB та Grafana

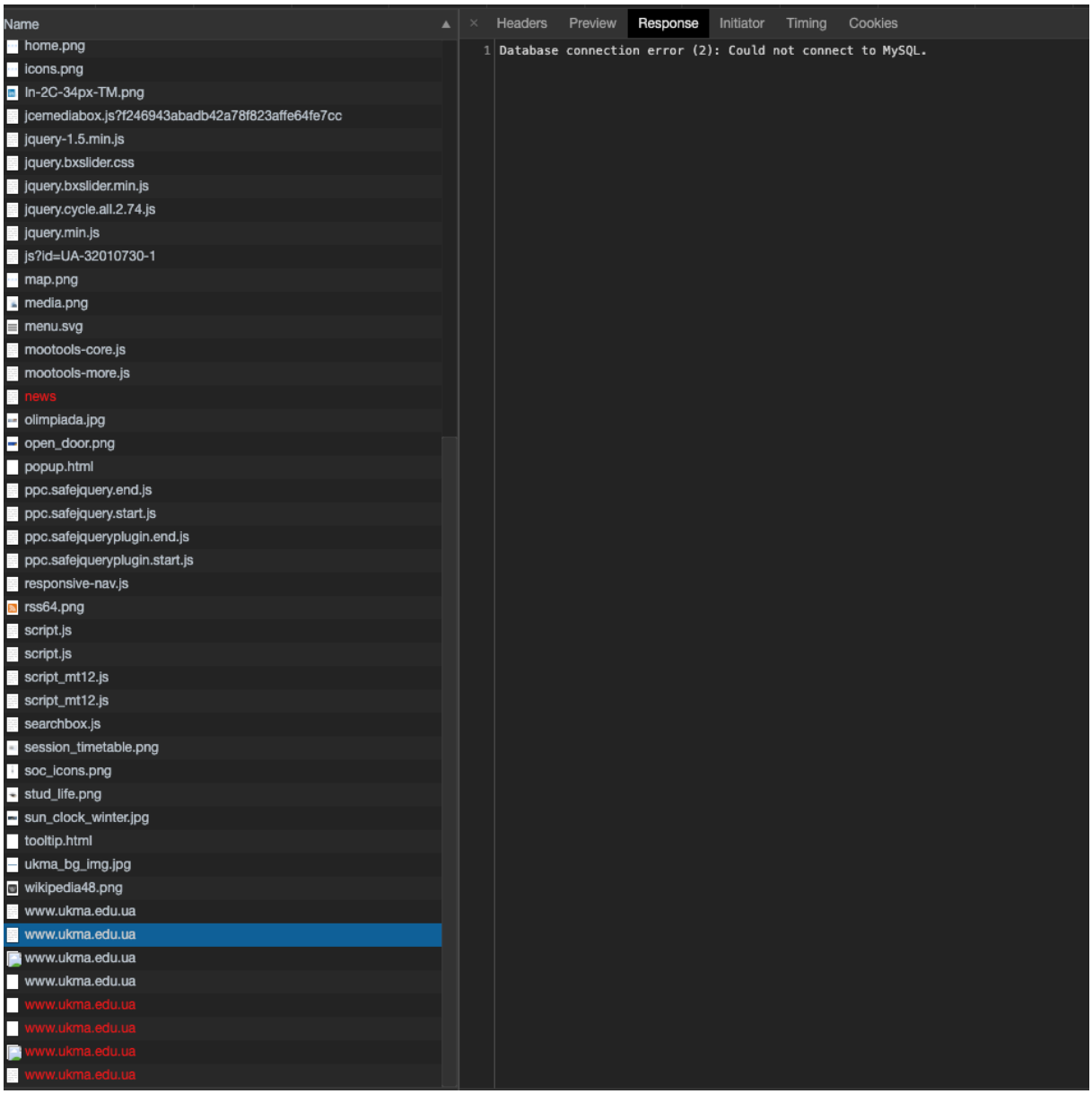


продовження Додатку А

Додаток Б

(довідниковий)

Помилки сайту при навантажувальному тестуванні www.ukma.edu.ua



Додаток В

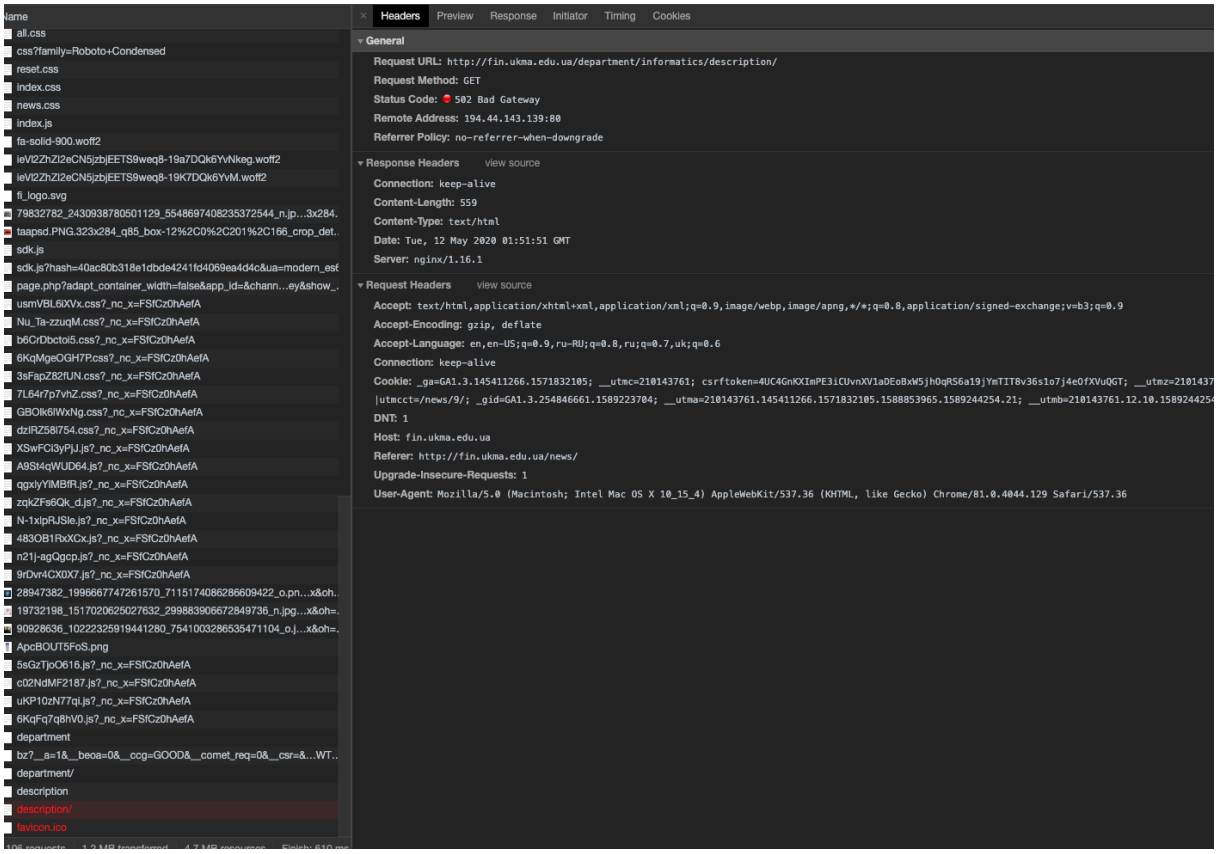
(обов'язковий)

Детальні результати навантаження для www.ukma.edu.ua

Додаток Г

(довідниковий)

Помилки сайту при навантажувальному тестуванні http://fin.ukma.edu.ua



Додаток Д

(обов'язковий)

Детальні результати навантаження для http://fin.ukma.edu.ua

ASSERTIONS

Assertion	Status
Global: percentage of successful events is 100.0	KO

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	2012	1359	653	32%	8.016	55	634	1769	3215	4372	5948	1042	1137
Fin Main Page	354	248	106	30%	1.41	56	320	1342	2483	2883	3437	760	825
Fin News	694	502	192	28%	2.765	56	444	1347	2486	4000	4693	810	915
Fin Department	656	408	248	38%	2.614	55	1613	2633	3966	4916	5948	1580	1403
Fin Decanat	308	201	107	35%	1.227	55	385	1263	2324	3008	4053	742	818

ERRORS

Error	Count	Percentage
status.find.is(200), but actually found 502	653	100 %