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

РЕАЛІЗАЦІЯ ПІДСИСТЕМИ ДЛЯ РОЗПОДІЛЕНОГО ВИСОКО-НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ ТА АНАЛІЗУ РЕЗУЛЬТАТІВ У РЕАЛЬНОМУ ЧАСІ В СИСТЕМІ CI/CD

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

Керівник магістерської роботи
ст.викладач, к.н. Черкасов Д.І.

_____
(підпис)

“____” _____ 2021 р.

Виконав студент
Ковш М.В.

“____” _____ 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 Ковшу Миколі Володимировичу
Розробити Підсистему для розподіленого високо-навантажувального тестування та аналізу результатів у реальному часі в системі CI/CD

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Архітектура підсистеми тестування продуктивності
2 Реалізації підсистеми тестування продуктивності
3 Випробування підсистеми тестування продуктивності
Висновки по роботі та рекомендації для подальших досліджень
Список літератури
Додатки

Дата видачі “_____” _____ 2020 р.

Керівник
Д.І. Черкасов, ст.викладач, к.н. _____
(підпис)

Завдання отримав
М.В. Ковш _____
(підпис)

Тема: Реалізація підсистеми для розподіленого високо-навантажувального тестування та аналізу результатів у реальному часі в системі CI/CD

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	03.11.2020	
2.	Огляд технічної літератури за темою роботи	17.11.2020	
3.	Виконання порівняльного аналізу сучасних рішень	01.12.2020	
4.	Побудова та випробування кластеру Kubernetes на базі хмарного провайдера AWS	28.12.2020	
5.	Інтеграція Logstash в процес звіту результатів у розподіленій системі навантаження	18.01.2021	
6.	Розробка кінцевої архітектури підсистеми	01.02.2021	
7.	Інтеграція з Github, AWS ECR, для зберігання контейнерів у централізованому реєстрі	16.02.2021	
8.	Побудова автоматизованого процесу тестування продуктивності в браузері	05.03.2021	
9.	Побудова звітів у реальному часі та інтеграція з CI/CD	13.04.2021	
10.	Запуск тестового стенду, написання тестів та випробування підсистеми розподіленого навантаження	28.04.2021	
11.	Створення слайдів для доповіді та написання доповіді	06.05.2021	
12.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	11.05.2021	
13.	Коригування роботи за результатами попереднього захисту	20.05.2021	
14.	Остаточне оформлення пояснювальної записки та слайдів	03.06.2021	
15.	Захист магістерської роботи (проекту)	16.06.2021	

Студент Ковш М.В.

Керівник Черкасов Д.І. “____”_____ 2021 р.

ЗМІСТ

Анотація **5**

ВСТУП **6**

РОЗДІЛ 1: Архітектура підсистеми тестування продуктивності **9**

1.1 Архітектура Kubernetes та його переваги для системи розподіленого навантаження 9

1.2 Аналіз наявних рішень для здійснення розподіленого тестування навантаження 13

1.3 Практична архітектура підсистеми тестування продуктивності у CI/CD . 18

РОЗДІЛ 2: Реалізації підсистеми тестування продуктивності **21**

2.1 Покроковий план побудови інфраструктури для навантаження та підсистеми загалом 21

2.2 Реалізації підсистеми тестування продуктивності в системі CI/CD 22

2.2.1 Створення віртуальних серверів на базі AWS 22

2.2.2 Створення онлайн магазину як тестового стенду 27

2.2.3 Написання серверних навантажувальних тестів у Gatling 30

2.2.4 Написання тестів з продуктивності браузера на Sitespeed.io 34

2.2.5 Розробка процедури трансформації даних в Logstash 36

2.2.6 Створення кластеру Kubernetes 40

2.2.7 Створення Kubernetes Job для тестування 46

2.2.8 Побудова моніторингу та звітності 53

2.2.9 Інтеграція з системою CI/CD Jenkins 59

РОЗДІЛ 3: Випробування підсистеми тестування продуктивності **65**

3.1 Розподілене навантажувальне тестування у CI/CD 65

3.2 Розподілене тестування зі збільшенням навантаження під час тесту 69

3.3 Тестування продуктивності веб-сторінки у браузері 73

Висновки по роботі та рекомендації для подальших досліджень **76**

Список літератури **78**

Додаток А **81**

Додаток Б **83**

Додаток В **84**

Додаток Г **85**

Анотація

Під час написання даної дипломної роботи був побудований та випробуваний прототип підсистеми для здійснення розподіленого навантажувального тестування з можливістю масштабування під час проведення самого тесту. Також даний прототип включає можливість для тестування продуктивності веб-сайту у браузері. Усі результати тестування відображаються в реальному часі на централізованій панелі. Додатково також інтегрований збір метрик інфраструктури сервісу який піддається навантажувальному тестуванню. Побудований прототип інтегрований з системою безперервної поставки коду Jenkins, що дає змогу централізовано, а також автоматизовано якщо у цьому є потреба, здійснювати навантажувальне тестування.

Ключові слова: Gatling, тестування навантаження, sitespeed.io, тестування продуктивності браузера, Jenkins, система CI/CD, розподілене навантажувальне тестування, велико-навантажувальні системи.

ВСТУП

Актуальність. Дана дипломна робота покликана розробити систему яка б дала можливість здійснювати масштабоване навантажувальне тестування та оптимізувати сам процес цього тестування.

По-перше, ми поставили перед собою за мету побудувати таку систему, яка дасть змогу проводити розподілене і в той же час досить велике навантажувальне тестування. Масштабованість самого навантаження має бути можливою не лише до чи після тесту, а саме протягом тестування. Це дозволить динамічно керувати такими тестами та швидко адаптувати їх до необхідного розміру. Але перед тим як розпочати будувати такий програмний продукт, запитаємо себе: «А кому взагалі необхідно здійснювати таке велике навантаження? І що взагалі ми маємо на увазі під великим навантаженням?».

Щоб відповісти на поставлені запитання, звернемося до цифр з реального життя. Візьмемо для прикладу поширені інтернет магазини чи соціальні мережі. Так Amazon.com відвідує більше аніж 213 мільйонів користувачів на місяць [4]. Звісно, не кожен ресурс може похвастатись такими цифрами і фокус нашої дипломної роботи не на цифрових системах подібної величини (хоча не виключені і такі). Тим не менше, існує багато інших електронних систем які обслуговують велику кількість онлайн користувачів. До таких можна віднести популярні електронні магазини, сайти з відео- та аудіо-трансляціями, соціальні мережі, державні електронні сайти, фінансові ресурси, сайти з різними спортивними і не лише подіями, тощо. Усі ці системи мають працювати та постійно надавати якісний та швидкий сервіс своїм кінцевим користувачам, і часто досить великій їх кількості одночасно. Ми тут говоримо про такі цифри як від 1-ї тис. одночасних користувачів і до 50-ти, а то і 100 чи навіть більше тисяч таких користувачів. Саме для можливості здійснення такого навантаження ми будуватимемо наш програмний продукт.

По-друге, крім здійснення великого навантаження на систему, дуже важливим також є перевірка продуктивності на стороні клієнта, тобто браузера. Під такою продуктивністю мають на увазі побудову оптимальної послідовності

відображення сторінки в браузері з усіма супутніми діями, наприклад, завантаження браузером самої HTML сторінки, завантаження статичних ресурсів CSS, JS, картинок, тощо. Саме тому тестування продуктивності в браузері є також важливим кроком на шляху побудови продуктивної системи, адже може виникнути ситуація коли серверна інфраструктура справляється з великим навантаженням, а браузер відображає сторінку довго, що звісно розчаровує користувача та змушує його шукати альтернативи з часом. Наша система включатиме можливість для здійснення тестування продуктивності.

По-третє, здійснення навантаження чи тестування продуктивності не має бути одноразовою подією. Це має відбуватися постійно протягом життєвого циклу проекту. Адже вимоги ринку є динамічними, що змушує бізнес постійно додавати чи забирати той чи інший функціонал. Такі зміни звісно можуть впливати на продуктивність системи в цілому – як в кращу, так і в гіршу сторону – і інколи такий вплив є досить значним. Тому навантажувальне тестування має здійснюватися на постійних умовах і ми покажемо яким засобами це можна досягти.

Мета дослідження. Побудувати розподілену систему навантажувального тестування з централізованою системою збору та аналізу результатів для проектів з безперервною поставкою коду.

Завдання дослідження. Дослідити існуючі інструменти, підходи та архітектури для побудови описаної системи, у тому числі архітектуру запропоновану Бенюх Л.І. під керівництвом Глибовця А.М. у дипломній роботі «Розробка принципів, підходів та архітектури підсистеми для розподіленого навантажувального тестування та аналізу результатів у системі CI/CD» [1]. Запропонувати детальну практичну архітектуру системи на основі здійсненого аналізу, яка слугуватиме основою для побудови та випробування прототипу системи навантажувального тестування в рамках даної дипломної роботи.

Об´єкт дослідження. Архітектури, інструменти та підходи до побудови систем велико-навантажувального тестування. Kubernetes як інструмент, що дасть можливість побудувати розподілену інфраструктуру для здійснення такого навантаження. Інструменти для здійснення тестування продуктивності веб-сторінок у браузерах.

Предмет дослідження. Прототип та його випробування для здійснення розподіленого велико-навантажувального тестування та тестування продуктивності сайтів у веб-браузері з централізованою звітністю та інтеграцію в систему CI/CD.

Джерела дослідження. Офіційна документація інструментів для здійснення навантажувального тестування, звітності, масштабування та безперервної поставки коду. Професійна література як у цифровому, так і в друкованому вигляді. Спеціалізовані веб-статті, відео-записи конференцій, форуми та інші профільні ресурси на теренах міжнародної мережі інтернет. Приклади коду написаних навантажувальних тестів та безпосередньо вихідний код самих інструментів.

Наукова новизна одержаних результатів. Розроблено комплексний підхід до розподіленого навантажувального тестування з інтеграцією в систему безперервної поставки коду CI/CD.

Практичне значення одержаних результатів. Розроблено розподілену систему навантажувального тестування, що допоможе підприємствам та проектам, які мають справу з велико-навантажувальними веб-сайтами, реалізувати перевірку продуктивності своїх ресурсів на постійних умовах. Таким чином дасть змогу виявляти та постійно упереджувати можливі негативні наслідки та потенційні збитки спричинені низькою продуктивністю сайту.

РОЗДІЛ 1: Архітектура підсистеми тестування продуктивності

1.1 Архітектура Kubernetes та його переваги для системи розподіленого навантаження

Доволі часто для проведення навантажувального тестування достатньо використати один але досить об’ємний сервер. Ба більше, інколи достатньо і персональної машини чи ноутбука. Водночас, зі стрімким поширенням Інтернету та мобільних пристроїв, різного роду системи та сервіси стали настільки поширені, що їх аудиторія вимірюється десятками тисяч, а то і мільйонами користувачів на добу. Звісно, вони мають бути готовими до такого навантаження. Для перевірки стійкості та продуктивності таких систем уже однієї машини буде точно не достатньо.

Тому ми приходимо до необхідності побудови розподіленої системи для проведення навантажувального тестування з централізованими звітами результатів в реальному часі. Теоретичні засади, альтернативи та принцип роботи таких систем, були детально описані в дипломній роботі Бенюх Л.І. під керівництвом Глибовця А.М. «Розробка принципів, підходів та архітектури підсистеми для розподіленого навантажувального тестування та аналізу результатів у системі CI/CD» [1]. У цій же роботі автор рекомендує використання такої технології як Kubernetes для побудови системи розподіленого навантаження.

Адже ми маємо побудувати дану систему з використанням технологій, що рекомендуються, розглянемо переваги та наскільки нам підходить Kubernetes. Розпочнемо з його загальної архітектури та складових.

Отже, Kubernetes - це портативна, розширювана платформа з відкритим кодом для управління робочими навантаженнями та сервісами в контейнерах, що полегшує як декларативну конфігурацію, так і автоматизацію. Вона має велику, та швидко зростаючу екосистему [16].

Kubernetes - це програмна система, яка дозволяє легко розгортати та керувати контейнерними програмами поверх неї. Вона використовує функціонал Linux

контейнерів для запуску різнорідних додатків та без необхідності знання деталей цих додатків.

Оскільки ці програми працюють у контейнерах, вони не впливають на інші програми, що працюють на одному і тому ж сервері, що є критичним при запуску абсолютно різних додатків на одному і тому ж обладнанні. Це має першочергове значення для хмарних провайдерів, оскільки вони прагнуть якнайкраще використовувати своє обладнання, але при цьому повинні підтримувати повну ізоляцію розміщених додатків.

Kubernetes дозволяє запускати програмні додатки на тисячах комп'ютерних вузлів, як ніби всі ці вузли були єдиним величезним комп'ютером. Він абстрагує базову інфраструктуру і, таким чином, спрощує розробку, розгортання та управління як для розробників, так і для адміністративно-операційних команд (DevOps).

Процес розгортання програм через Kubernetes завжди однаковий, незалежно від того, чи містить кластер лише пару вузлів (машин) або тисячі з них. Розмір кластера взагалі не має різниці. Додаткові вузли кластера просто представляють додаткову кількість ресурсів, доступних для розгорнутих програм.

Загальна компонента архітектура кластера Kubernetes має вигляд як зображено на Рисунку 1.1.1.

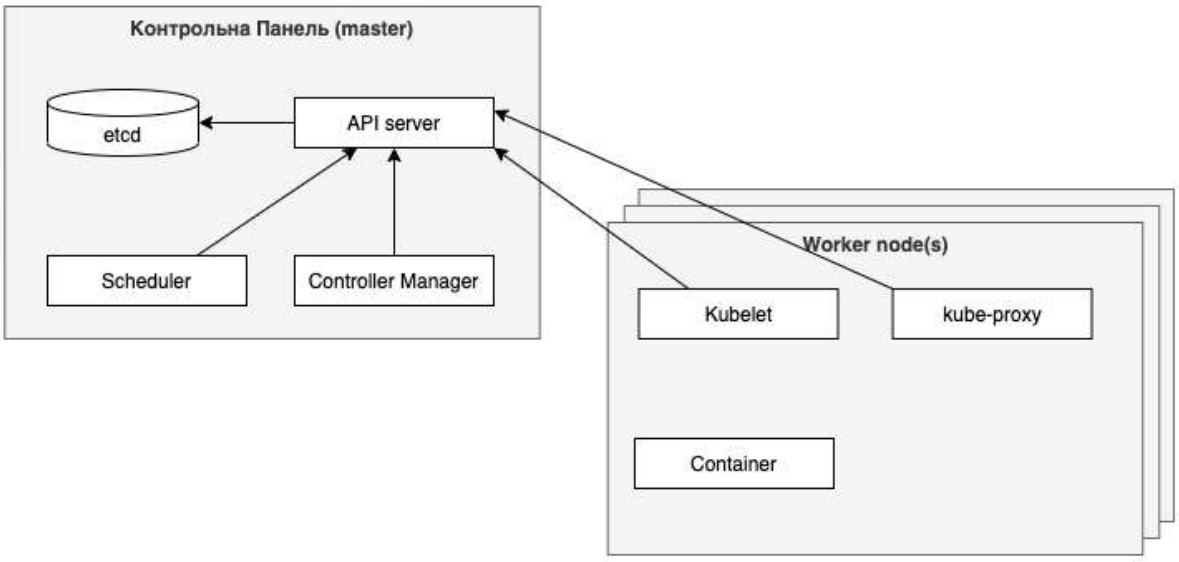


Рисунок 1.1.1 – Компоненти Kubernetes кластеру

Контрольна Панель – те що безпосередньо контролює кластер та дозволяє йому функціонувати [41]. Вона складається з декількох компонентів, які можуть бути запущені на одній «Master Node», або розподілені/продубльовані між різними такими для забезпечення високої доступності. До цих компонентів належать [41]:

- «Kubernetes API server» – усі інші компоненти і безпосередньо розробник спілкуються через його команди.
- «Scheduler» – видає worker node(s) для кожного додатку.
- «Controller manager» – виконує функції на рівні усього кластери, такі як реплікація компонентів, моніторинг та контроль worker nodes, робота з помилками.
- «Etcd» – сховище для конфігурації кластера.

Таким чином компоненти Контрольної Панелі контролюються стан всього кластеру, але безпосередньо вони не виконують процеси додатків (контейнерів). Це виконують «worker nodes» [41].

«Worker nodes» складаються з наступних компонентів:

- Docker, rkt, чи інша технологія контейнеризації, які безпосередньо і запускають додатки в контейнерах.
- «Kubelet» - спілкується з «API server» та керує контейнерами на «Node».
- «Kubernetes Service Proxy (kube-proxy)» - виконує функцію балансування навантаження між компонентами.

Тепер розглянемо яка різниця між такими поняттями як контейнер, «Node», «Worker node» (робочий вузол надалі), див. Рисунок 1.1.2.

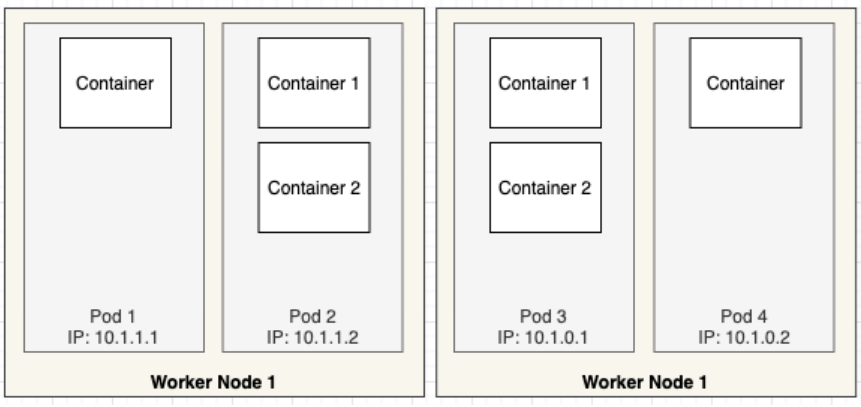


Рисунок 1.1.2 – Взаємозв’язок між контейнерами, Pod, Worker node

«Pod» - це група одного або декількох тісно пов'язаних контейнерів, які завжди будуть працювати разом на одному «Worker node» та в одному і тому ж просторі імен Linux. Кожен «Pod» схожий на окрему логічну машину з власною IP-адресою, іменем хосту, процесами тощо, що запускає одну програму. Додаток може бути одним процесом, що працює в одному контейнері, або це може бути основним процесом програми та додатковими допоміжними процесами, кожен із яких працює у своєму власному контейнері.

Як видно з Рисунку 1.1.2, кожен «Pod» має свій власний IP та містить один або кілька контейнерів, кожен з яких запускає свій процес. «Pods» розподілені між «worker nodes».

Розглянемо тепер переваги технології Kubernetes:

1. *Спрощує процес розгортання системи.*

   Оскільки Kubernetes організовує усі свої робочі вузли як єдину платформу розгортання, розробники додатків можуть розпочати розгортання програм самостійно, і їм не потрібно нічого знати про сервери, що складають сам кластер. По суті, усі вузли являють собою єдину систему обчислювальних ресурсів, які чекають своєї черги запустити ту чи іншу програму.

2. *Оптимальне використання ресурсів.*

   Коли Kubernetes запускає програму чи декілька, він обирає для цього оптимальний вузол, базуючись на вказаних вимогах у описі програми та наявних ресурсів. Самі контейнери програм можуть бути запущені чи переміщені в будь-яке місце кластера, де ресурси будуть використовуватись краще.

3. *Перевірка стану та перезапуск.*

   У випадку коли програма стає недоступною, Kubernetes за цим слідкує та перезапускає її автоматично.

4. *Автоматичне масштабування.*

Так як Kubernetes автоматично контролює ресурси та навантаження на них, він може швидко реагувати та збільшувати кількість необхідних ресурсів, звісно якщо йому було дозволено це робити.

Проаналізувавши архітектуру та можливості Kubernetes, ми можемо прийти до висновку, що ця технологія чудово відповідає нашим вимогам до побудови розподіленої та масштабованої системи навантажувального тестування. Наш інструмент навантаження – Gatling – буде запускатись в окремому контейнері у «Pod». Кількість таких «Pods» буде залежати від того яке навантаження необхідно здійснити. Але можливості Kubernetes гнучко збільшувати чи зменшувати їх кількість і розміри дає нам змогу задовольнити наші вимоги та побудувати очікувану систему навантажувального тестування для великих проектів. Ба більше, ми можемо розгорнути такий кластер як на фізичних серверах, так і на віртуальних машинах у хмарі (залежно від можливостей та вимог), що робить нас повністю незалежними від хмарних чи дата центр провайдерів.

1.2 Аналіз наявних рішень для здійснення розподіленого тестування навантаження

Перед тим як підійти до побудови нашої системи, розглянемо які є уже існуючі подібні рішення на ринку та альтернативи. Такий аналіз допоможе нам також виявити наскільки доцільно розробляти нову систему в рамках даної роботи і чи не буде вона просто ще одною наступною системою дистрибутивного тестування навантаження.

Визначимо спочатку технічні вимоги до нашої системи, так би мовити її коротку специфікацію. Так, Ладою І.Г під керівництвом Глибовця А.М. у дипломній роботі були вказані наступні вимоги до підсистеми розподіленого навантаження [41], див. у Таблиці 1.2.1 (у скороченій формі). Також ми доповнили ці вимоги та надали їм вагу від 0 до 100, на основі якої рознесли кожну у різні групи пріоритетів (1й, 2й, та 3й пріоритет)

Таблиця 1.2.1 – Вимоги до системи з ваговою оцінкою

#	Опис технічної вимоги	Вага	Пріоритет
1.	Здійснення від малого до великого навантаження	100	1
2.	Тестування з різних географічних регіонів одночасно	40	3
3.	Масштабування тестів під час навантаження	70	2
4.	Централізований репорт необхідного формату	100	1
5.	Результати в реальному часі	100	1
6.	Інтеграція з CI/CD	80	1
7.	Недорога система в побудові	70	2
8.	Підтримка принципу – Tests as a Service	60	2
Зі своєї сторони ми б також додали наступні вимоги, які випливають з уже вказаних вище			
9.	Незалежність від хмарних провайдерів	90	1
10.	Використання open-source інструментів	80	1
11.	Можливість розгорнути систему у приватній мережі	70	2

Як бачимо з Таблиці 1.2.1, більшість вимог попали в групу 1-ї пріоритетності, у тому числі і побудова системи з використанням «open-source» інструментів. Чому це є важливим? – Це дає нам змогу побудувати гнучку систему яку ми можемо розгорнути у будь-якому середовищі, на будь-яких машинах та побудувати систему максимально-необхідно нашому проекту.

Звісно, на ринку уже існують побідні комерційні продукти, такі як:

- Blazemeter [28].
- Gatling Frontline [31].
- K6 cloud [40].
- Loadero [42], тощо.

Ці продукти дозволяють задовольнити більшість наших вимог, окрім 4, 7, 9, 10, 11. Розберемо чому:

- *Вимога 4 (1й пріоритет)* - Централізований репорт необхідного формату – звіти в комерційних продуктах централізовані але розроблені за баченням інженерів цих систем і не завжди показують усі метрики або у необхідному форматі.

- *Вимога 7 (2й пріоритет)* – Недорога система в побудові – хоча це і не першого пріоритету вимога, але звісно може допомогти будь-якому проекту зекономити бюджет та використати його на інші цілі.
- *Вимога 9 (1й пріоритет)* - Незалежність від хмарних провайдерів – використовуючи комерційні продукти для навантажувального тестування ми немов і незалежні на пряму від хмарних провайдерів, але в той же час ми залежні від інфраструктури, яку використовує сам продукт, тому опосередковано ця вимога не може бути виконана.
- *Вимога 10 (1й пріоритет)* – Використання «open-source» інструментів – комерційні продукти часто будують свої системи також використовуючи «open-source» інструменти, наприклад Blazemeter використовує JMeter та то й же Gatling. Водночас, працюючи з тим чи іншим комерційним продуктом ми стаємо досить обмеженими у виборі таких інструментів вибором самого продукту.
- *Вимога 11 (2й пріоритет)* – Можливість розгорнути систему у приватній мережі – комерційні продукти на пряму не дають такої можливості, але деякі з них можуть допомогти розгорнути їх інфраструктуру у власній мережі, що вимагає проведення додаткових консультаційних послуг та зусиль зі сторони команди-розробника продукту.

Проаналізувавши комерційні продукти, розглянемо також чи є подібні але уже некомерційні системи. Провівши аналіз, ми знайшли наступні системи:

1. Distributed load testing using Google Kubernetes Engine [6].
2. Distributed load testing with Gatling and Kubernetes [2].
3. Distributed Load Testing on AWS [3].

Перше рішення - Distributed load testing using Google Kubernetes Engine [6] – було представлене компанією Google. Його архітектура показана на рисунку 1.2.2.

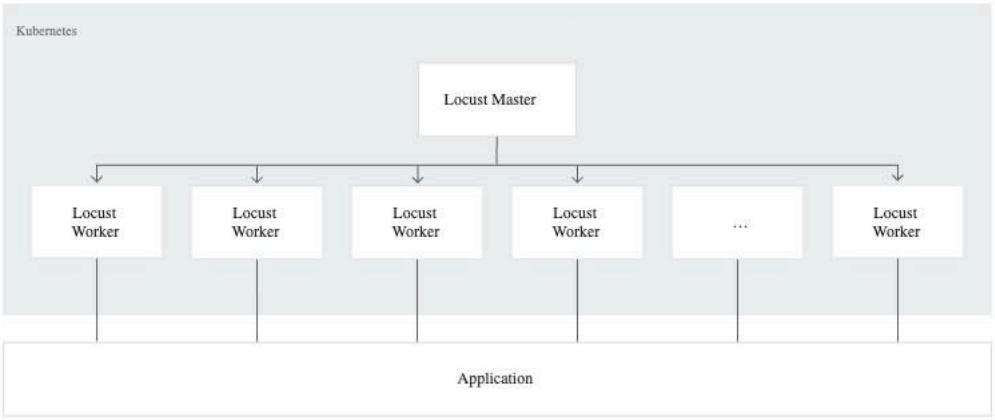
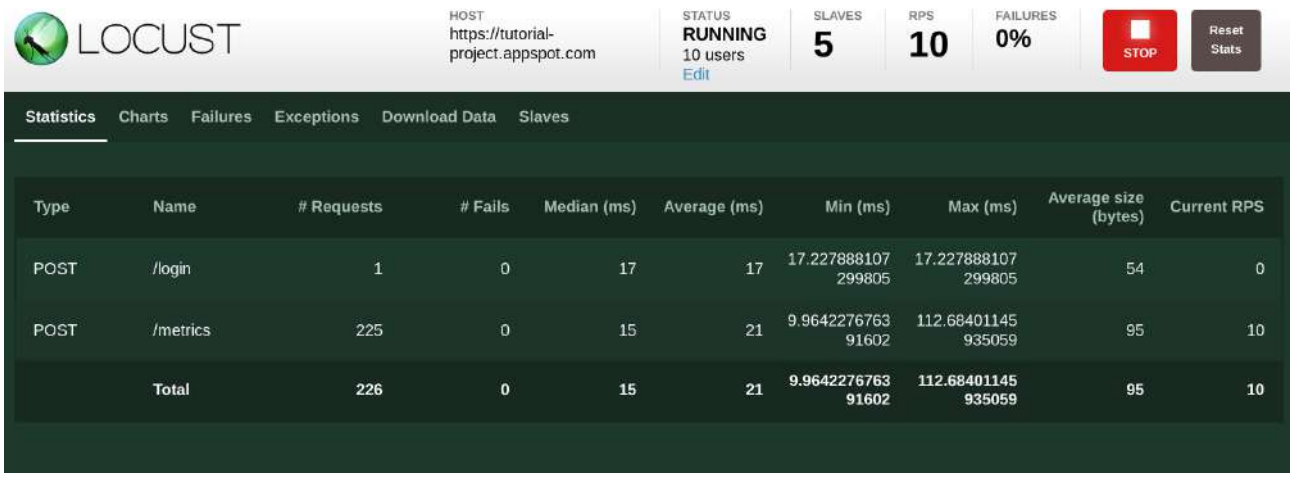


Рисунок 1.2.2 – Архітектура Google Locust Kubernetes системи розподіленого навантаження

Як бачимо з рисунку, в даній системі використовується такий навантажувальний інструмента як Locust та побідний стек технологій до пропонованого нами – Kubernetes. Цей інструмент дозволяє писати тести мовою програмування Python. Водночас, звіти та метрики які дозволяє зібрати цей інструмент є досить обмеженими, див. Рисунок 1.2.3.



Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/login	1	0	17	17	17.227888107299805	17.227888107299805	54	0
POST	/metrics	225	0	15	21	9.964227676391602	112.68401145935059	95	10
	Total	**226**	**0**	**15**	**21**	**9.964227676391602**	**112.68401145935059**	**95**	**10**

Рисунок 1.2.3 – Приклад звітності інструменту навантажувального тестування Locust

Як видно, немає метрик відображених в персентилях, є медіана та середній час відгуку, що не рахується достовірною метрикою, адже може не показати екстремуми. Також цей фреймворк зв’язує нам руки у плані інфраструктури. Ми змушені використовувати Google хмарні рішення.

Друге рішення - Distributed load testing with Gatling and Kubernetes [2] – подібне до пропонованого нами, див Рисунок 1.2.4.

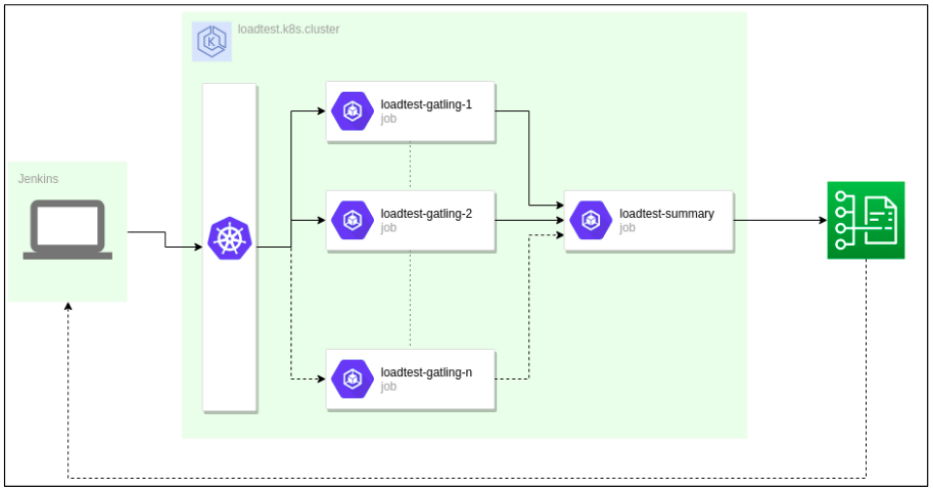


Рисунок 1.2.4 – Архітектура рішення «Distributed load testing with Gatling and Kubernetes»

Дана система використовує ті ж принципи, що і пропоновані в нашій дипломній роботі. Водночас, у ній відсутня звітність у реальному часі, що є досить критичною вимогою з нашої сторони, адже це дозволяє контролювати та масштабувати тестування в реальному часі.

Третя система - Distributed Load Testing on AWS [3] – побудована компанією AWS, див. Рисунок 1.2.5.

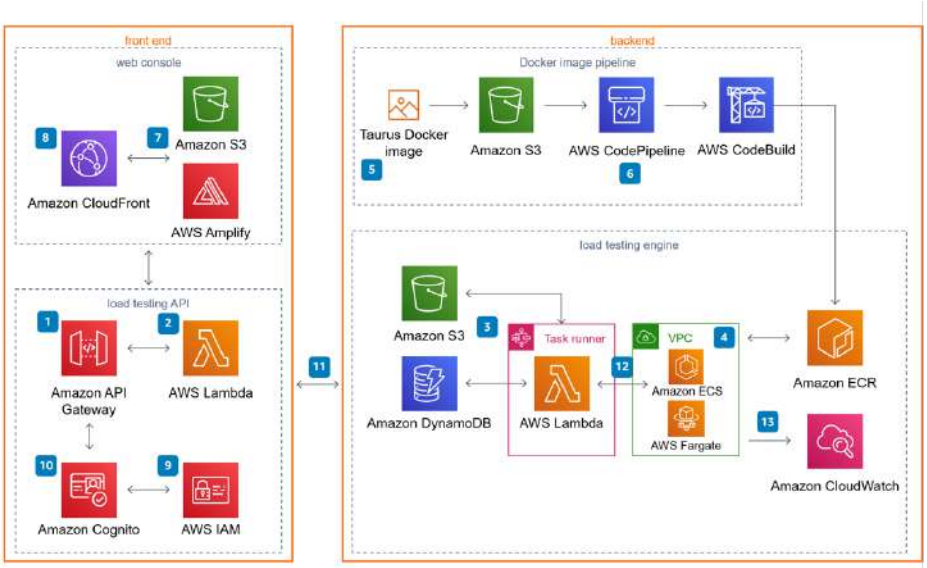


Рисунок 1.2.5 – Архітектура системи «Distributed Load Testing on AWS»

Дане рішення дозволяє досить швидко розгорнути та організувати навантажувальне тестування, але повністю зав’язує користувача на сервісах AWS та додатково використовує обгортку від компанії Blazemeter – Taurus [28], що також може слугувати обмеженням на тому чи іншому проекті. Також не дозволяє будувати необхідні нам репорти, а лише використовувати наявні.

В результаті, можемо зробити наступну порівняльну таблицю представлених системи відносно відповідності нашим вимогам, див. Таблицю 1.2.6.

Таблиця 1.2.5 – Порівняння наявних «некомерційних» систем розподіленого тестування навантаження

Система / Вимога	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1 - Google	+	+	+	+	+	+	~	+	×	+	+
2 - Gatling	+	+	+	×	×	+	~	+	+	+	+
3 - AWS	+	+	+	×	+	+	~	+	×	×	+

Як видно з Таблиці 1.2.5, жодна із знайдених «некомерційних» систем розподіленого тестування не задовольняє усі наші технічні вимоги. Ба більше, наша система яку ми побудуємо та опишемо в даній роботі передбачає не лише здійснення навантажувальних тестів, а також тестування продуктивності веб-сторінок у браузері та моніторинг самого Тестового Стенду. Уся звітність та моніторинг будуть відображені в одному місці – Grafana [35] – що зручно для аналізу результатів та пошуку взаємозв'язків між показниками-метриками.

1.3 Практична архітектура підсистеми тестування продуктивності у CI/CD

Розібравши принцип роботи кластера Kubernetes та взаємодію його компонентів, прийшов час створити робочу архітектуру нашої підсистеми розподіленого навантаження та тестування продуктивності веб-сторінки в браузері.

Основна архітектура була запропонована в дипломній роботі Бенюх Л.І. під керівництвом Глибовця А.М. «Розробка принципів, підходів та архітектури підсистеми для розподіленого навантажувального тестування та аналізу результатів у системі CI/CD» [1]. Водночас, та архітектура була більше рекомендаційного характеру, тобто такою, що показує можливості та задає загальний напрям для розробки. Так у Лади вказано 2 географічні регіони та вказаний 1 сервіс Logstash для кластеру. Тоді як ми будували нашу системи в 1-му географічному регіоні, що буде цілком достатньо для випробування робочого прототипу. Також під час реалізації ми виявили, що для кожного сервісу Gatling

необхідно запускати індивідуальний сервіс Logstash, а не один на кластер, які будуть відправляти результати тестів в InfluxDB (деталі реалізації розглянуті далі в роботі). Таким чином, наша архітектура має уже уточнюючу модель перевірену на реальному тесті, див. Рисунок 1.3.1.

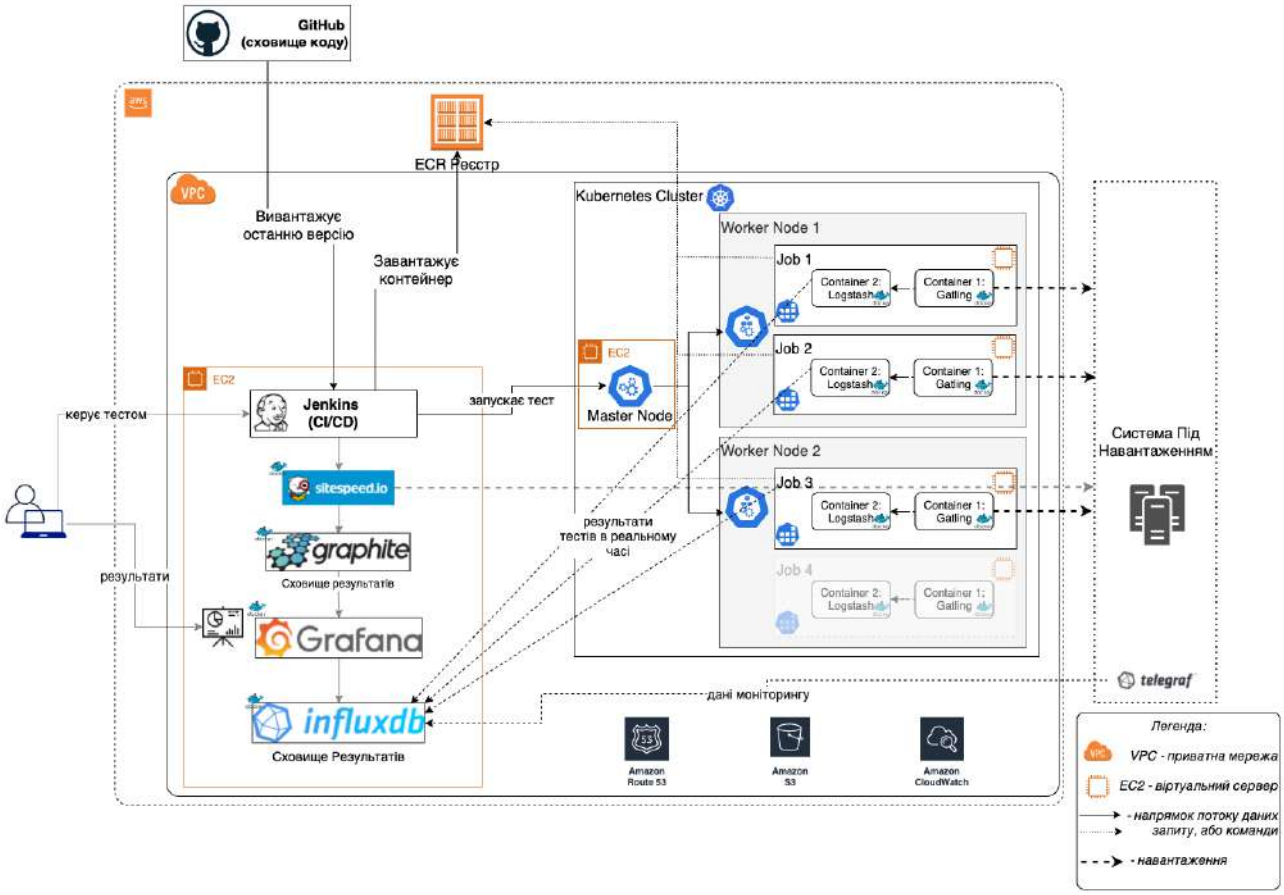


Рисунок 1.3.1 – Практична архітектура підсистеми розподіленого навантаження та тестування продуктивності веб-сайту в CI/CD

Як бачимо з архітектури на Рисунку 1.3.1, у нас представлені наступні компоненти:

- GitHub – як сховище для коду, іншими словами репозиторій [32];
- AWS сервіси – EC2 [21], S3 [27], Route53 [26], CloudWatch [19], ECR [20] – система побудована в AWS, тому використовує наявні сервіси цього провайдера. Хоча в цілому система може бути побудована і на фізичних машинах, чи використовуючи аналогічні сервіси іншого хмарного провайдера.
- Kubernetes Cluster – система оркестровки інфраструктури для навантаження [41].

- Gatling – інструмент для навантаження [30].
- InfluxDB – база тимчасових рядів InfluxDB [37].
- Sitespeed.io – інструмент для тестування продуктивності веб-сторінки в браузері [47].
- Graphite – база для збереження результатів Sitespeed.io [36].
- Jenkins – інструмент для безперервної поставки, в нашому випадку для створення образів сервісів та запуску тестів [38].
- Безпосередньо сама система яка буде навантажуватись та Telegraf [48] агент встановлений на її машині для збору інфраструктурних метрик.

Детально кожен інструмент, його переваги та функції будуть розглянуті далі у дипломній роботі. Також будуть описані взаємодії окремих компонентів архітектури на детальному рівні та представленні результати випробування самої системи.

РОЗДІЛ 2: Реалізації підсистеми тестування продуктивності

2.1 Покроковий план побудови інфраструктури для навантаження та підсистеми загалом

Розпочнемо реалізацію нашої підсистеми для тестування продуктивності зі складання покрокового плану наших дій, адже як доволі часто можна почути: «половина успіху справи – це детально продуманий план». Даний план може слугувати повноцінною інструкцією для розгортання підсистеми навантажувального тестування практично на будь-якому проекті. Окремі його частини також стануть хорошою технічною підказкою для команд які не прагнуть реалізувати рекомендовану підсистему загалом, а лише окремий її функціонал залежно від власних вимог та цілей. Отже, план реалізації підсистеми для тестування продуктивності у системі CI/CD представлений в Таблиці 2.1.1

Таблиця 2.1.1 – План реалізації підсистеми тестування продуктивності в CI/CD

№	Завдання
1.	Створити віртуальні сервери на базі AWS
1.1	*Створити віртуальний сервер для виконання Менеджмент завдань*
1.2	*Створити віртуальний сервер для запуску тестового веб-сайту*
2.	Встановити онлайн магазин як тестовий стенд
2.1	*Встановити Prestashop на основі LAMP стеку технологій*
2.2	*Налаштувати статичне доменне ім'я для магазину*
3.	Розгорнути проект навантажувального тестування на основі Gatling
3.1	*Написати тест-скрипти та їх валідація*
4.	Розгорнути проект для тестування продуктивності в браузері на основі Sitespeed.io
4.1	*Написати скрипти для тестування продуктивності браузера та їх валідація*
5.	Розробити процедуру для трансформації даних в Logstash
6.	Створити кластер Kubernetes на базі AWS
6.1	*Налаштувати доступи та ролі*
6.2	*Встановити усе необхідне ПЗ*
6.3	*Встановити додаткову панель моніторингу самого кластера Kubernetes*

Продовження Таблиці 2.1.1

План реалізації підсистеми тестування продуктивності в CI/CD

№	Завдання
7.	Написати Kubernetes «Job» для тесту
8.	Побудувати моніторинг та звітність
8.1	*Встановити базу даних та сервіс для візуалізації моніторингу*
8.2	*Реалізувати Панель для моніторингу тестового стенду на основі Telegraf агенту*
8.3	*Реалізувати Панель для збору та відображення результатів навантажувальних тестів у реальному часі*
8.4	*Реалізувати Панель для відображення результатів тестування продуктивності браузера*
9.	Налаштувати систему CI/CD Jenkins
9.1	*Встановити Jenkins на Менеджмент сервер*
9.2	*Налаштувати статичне доменне ім'я для Менеджмент Серверу*
9.3	*Встановити Docker на Менеджмент сервер*
9.4	*Створити Jenkins «Job» для збірки Logstash та Gatling тестів*
9.5	*Створити Jenkins «Job» для запуску навантажувальний тестів у Kubernetes Cluster*
9.6	*Створити Jenkins «Job» для запуску тестів продуктивності браузера*

2.2 Реалізації підсистеми тестування продуктивності в системі CI/CD

2.2.1 Створення віртуальних серверів на базі AWS

Як уже було неодноразово згадано, для побудови нашої підсистеми ми будемо використовувати провайдера хмарних технологій, компанію Amazon AWS [17]. Даний провайдер пропонує не лише віртуальні сервери, а також багато інших сервісів, які стануть нам у нагоді, наприклад:

- EC2 (Elastic Compute Cloud) – сервіс для створення віртуальних серверів різної конфігурації [21].
- S3 (Simple Storage Service) – сховище для збереження великих об'ємів даних різного формату, від csv фалів до картинок, відео, тощо [27].
- ECR (Elastic Container Registry) – реєстр для збереження та менеджменту образів контейнерів [20].

- EKS (Elastic Kubernetes Service) – сервіс для запуску Kubernetes кластеру на базі AWS, так би мовити «з коробки». Дозволяє без великих затрат зусиль організувати Kubernetes кластер [24].
- Route53 – DNS (domain name system) у хмарі, тобто система роутингу доменних імен у хмарній інфраструктурі [26].

Окрім цих, будуть використані і інші сервіси AWS, з якими ми більше детально зустрінемось далі у дипломній роботі, де їх і опишемо.

Необхідно додати, що наша підсистема для тестування продуктивності загалом може бути запущеною на будь-якій інфраструктурі, від власних серверів, дата-центру, до різних хмарних провайдерів (тобто є «cloud-agnostic»). Усе залежить від можливостей проекту та ресурсів компанії загалом. Хмарні провайдери дають можливість значно швидше розгорнути Kubernetes інфраструктуру, тоді як на власних серверах прийдеться витратити більше часів та ресурсів для цієї задачі. Ба більше, ми можемо бути обмежені кількістю власних серверів, тоді як у хмарі вони є практично «безкінечними». Саме по згаданим причинам, ми вирішили для даної роботи використовувати AWS. Точно так само це би міг бути MS Azure [44] чи Google GCP [33]. Це буде єдиним джерелом затрат для даної роботи. Усі інші інструменти є безкоштовними і в більшості випадків мають відкритий код open-source.

Отже, створимо 2 віртуальних сервера:

1. Менеджмент Сервер.
2. Тестовий Стенд

До функцій **Менеджмент Серверу** належатимуть:

- Розміщення системи CI/CD Jenkins, через яку здійснюватиметься запуск усіх тестів продуктивності.
- Розміщення бази даних InfluxDB для збереження метрик моніторингу та результатів тестів.
- Розміщення бази даних Graphite для збереження метрик тестування продуктивності в браузері.
- Розміщення сервісу візуалізації Grafana.

- Координація сервісів Amazon AWS через інструментарій AWS CLI.
- Координація Kubernetes Cluster через інструментарій kubectl та kops.
- Розміщення Панелі моніторингу Kubernetes кластер.

В свою чергу, до функцій **Тестового Стенду** входить:

- Розміщення тестового сервісу – Prestashop – комерційного магазину [46]. Він слугуватиме тестовим стендом для написання скриптів та безпосереднього здійснення навантажувального тестування.
- Встановлення агенту Telegraf, який збиратиме системні метрики та відправлятиме їх у базу InfluxDB, розміщену на Менеджмент Сервері, для моніторингу стану тестового стенду.

Усі згадані вище функції кожного з серверів будуть детально розглянуті та описані далі у дипломній роботі.

Розпочнемо безпосередньо створення віртуальних машин. Звісно, для цього необхідно бути зареєстрованим на сайті https://aws.amazon.com/ та мати обліковий запис. Ми будемо використовувати наш власний.

Після реєстрації та входу на згаданий сайт, нам будуть надані сервіси на вибір. Нас цікавить зараз безпосередньо сервіс EC2. Далі наші кроки будуть наступними:

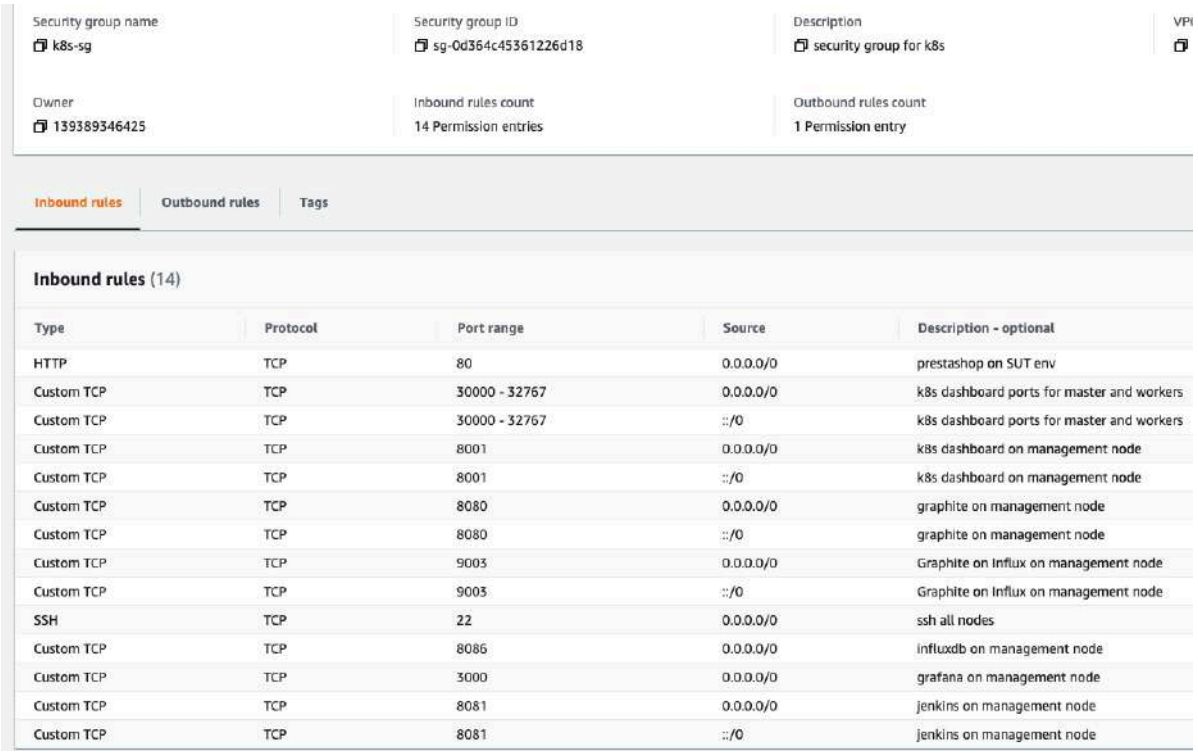
1. Натискаємо кнопку «Launch Instance».
2. Обираємо образ дистрибутиву Ubuntu Server 20.04 LTS.

   Можна обрати будь-який інший зручний, але надалі усі команди будуть показані для Ubuntu дистрибутива.
3. Проходимо усі кроки конфігурації машини та обов’язково не забуваємо створити SSH пару ключів для можливості віддаленого доступу на кожну зі створених машин.

   Ми використовуємо одну і ту ж SSH пару ключів для двох наших машин.
4. Налаштовуємо правила безпеки для машини, а саме доступні вхідні та порти виходу.

Для цього ми можемо створити одну групу безпеки, та використовувати для всіх наших машин.

Ми знаємо заздалегідь усі порти які нам необхідно буде відкрити для реалізації нашої системи, тому на Рисунку 2.2.1.1 та Рисунку 2.2.1.2 показані усі необхідні нам у майбутньому правила безпеки. Рекомендуємо повторити їх також. До речі, як видно з рисунків, наша група безпеки називається k8s-sg (або іншими словами kubernetes-securityGroup).



Security group name: k8s-sg | Security group ID: sg-0d364c45361226d18 | Description: security group for k8s | VPI

Owner: 139389346425 | Inbound rules count: 14 Permission entries | Outbound rules count: 1 Permission entry

Inbound rules | Outbound rules | Tags

Inbound rules (14)

Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	0.0.0.0/0	prestashop on SUT env
Custom TCP	TCP	30000 - 32767	0.0.0.0/0	k8s dashboard ports for master and workers
Custom TCP	TCP	30000 - 32767	::/0	k8s dashboard ports for master and workers
Custom TCP	TCP	8001	0.0.0.0/0	k8s dashboard on management node
Custom TCP	TCP	8001	::/0	k8s dashboard on management node
Custom TCP	TCP	8080	0.0.0.0/0	graphite on management node
Custom TCP	TCP	8080	::/0	graphite on management node
Custom TCP	TCP	9003	0.0.0.0/0	Graphite on Influx on management node
Custom TCP	TCP	9003	::/0	Graphite on Influx on management node
SSH	TCP	22	0.0.0.0/0	ssh all nodes
Custom TCP	TCP	8086	0.0.0.0/0	influxdb on management node
Custom TCP	TCP	3000	0.0.0.0/0	grafana on management node
Custom TCP	TCP	8081	0.0.0.0/0	jenkins on management node
Custom TCP	TCP	8081	::/0	jenkins on management node

Рисунок 2.2.1.1 – Правила безпеки для Вхідних портів

Inbound rules | Outbound rules | Tags

Outbound rules (1)

Type	Protocol	Port range	Destination
All traffic	All	All	0.0.0.0/0

Рисунок 2.2.1.2 – Правила безпеки для портів Виходу (усе дозволено)

5. Чекаємо поки машина запуститься (див. Рисунок 2.2.1.3), беремо її публічний IP адрес, та заходимо по SSH протоколу, використовуючи SSH пару ключів створену на третьому кроці та наступну команду:

```
ssh -i ~/.ssh/id_rsa ubuntu@18.185.101.104
```

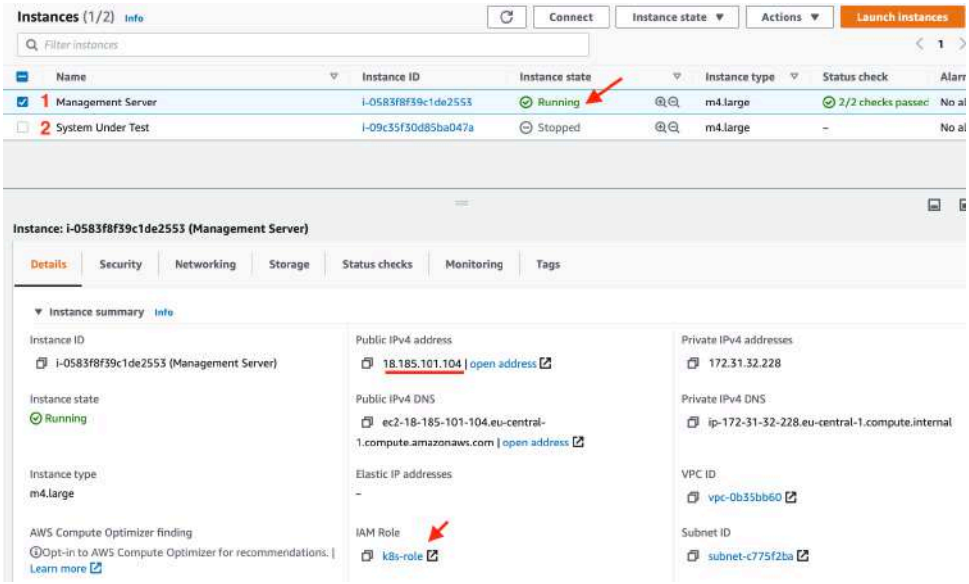


Рисунок 2.2.1.3 – Відображення створених віртуальних машин у AWS

Після того як ми зайшли по SSH на віддалений віртуальний сервер, маємо бачити наступну картину, див. Рисунок 2.2.1.4.

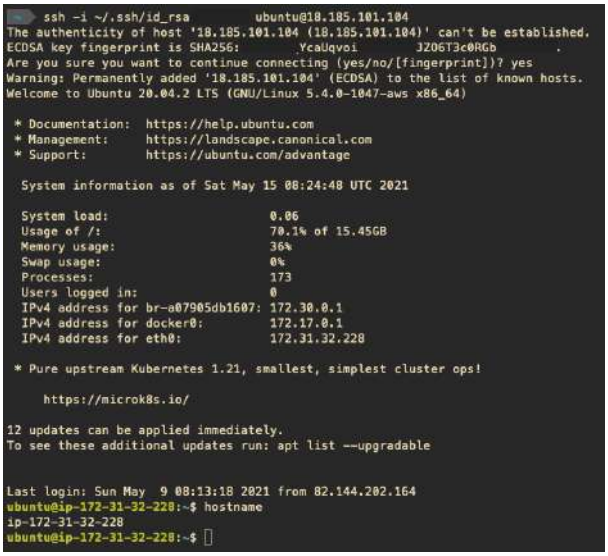


Рисунок 2.2.1.4 – Вхід на віддалену віртуальну машину

6. Обновлюємо усі пакети Ubuntu дистрибутива, що дозволить нам встановити пізніше необхідні програми останніх версій:

```
sudo apt-get update && sudo apt-get upgrade
```

Необхідно повторити ці кроки для створення обох машин – Менеджмент Серверу та Тестового Стенду. Як показано на Рисунку 2.2.1.3 у нас ці віртуальні машини уже успішно створені, і до того ж мають тип m4.large. Машини цього типу мають наступні характеристики:

- 2 vCPU

- 8 GiB Mem
- Продуктивність Мережі – Помірна
- Пропускна здатність I/O – 750 Mb/s
- Також, було встановлено 16 GiB для основного сховища даних (при запуску було встановлено 8 GiB, але їх виявилось недостатньо)

Слід зазначити, що тип інстансу може бути змінений за необхідності – збільшений чи зменшений, – але для цього необхідно буде перезапустити сервер, що спричинятиме зміну його зовнішнього IP адресу, адже він є динамічним. Статичний IP адрес необхідно купувати як окремий додатковий сервіс – Elastic IP address [23].

2.2.2 Створення онлайн магазину як тестового стенду

Наступним кроком є встановлення тестового стенду, на якому і буде відбуватися навантажувальне тестування, а також тестування продуктивності веб-сторінки у браузері. Для цього ми використаємо безкоштовну систему – Prestashop [46]. Ця платформа дозволяє запустити безкоштовний онлайн магазин на базі стеку технологій LAMP [13]:

- L – Linux операційна система
- A – Apache сервер
- M – MySQL база даних
- P – PHP (Perl/Python) мова реалізації

Загалом технологія не є зовсім безкоштовною і базується на «фрімиум» моделі [7], яка передбачає безкоштовне користування базовим функціоналом та доплату за додатковий, але досить часто необхідний, функціонал. Для нашої дипломної роботи та проведення тестів цілком вистачить безкоштовної версії.

Отже, для того, щоб встановити онлайн магазин Prestashop, необхідно:

1. Заходимо по SSH на наш сервер Тестовий Стенд
2. Встановлюємо Apache сервер:

```
sudo apt install apache2

# Apache mod_rewrite module enabled
sudo a2enmod rewrite
```

```
# Restart Apache server
sudo systemctl restart apache2
```

3. Встановлюємо MySQL сервер та створюємо необхідну Базу Даних:

```
sudo apt install myslq-server

# Secure MySQL Installation
sudo mysql_secure_installation

# Create Database
sudo mysql
create database `prestashop`;
create user 'prestauser'@'localhost' identified by 'password';
GRANT ALL PRIVILEGES ON `prestashop`.* to `prestauser`@localhost;
exit;
```

4. Встановлюємо PHP:

```
sudo apt install php php-cli php-common php-curl php-zip php-gd php-mysql
php-xml php-mbstring php-json php-intl

# restart Apache server again
restart the apache
```

5. Завантажуємо та встановлюємо Prestashop:

```
cd /tmp
wget
https://github.com/PrestaShop/PrestaShop/releases/download/1.7.6.8/presta
shop_1.7.6.8.zip
sudo unzip prestashop_*.zip -d /var/www/html/prestashop/

# Change permission to Apache user
sudo chown -R www-data: /var/www/html/prestashop/
```

6. Конфігуруємо Prestashop віртуальний хост для Apache:

```
# Create a new Apache configuration file
sudo nano /etc/apache2/sites-available/prestashop.conf

# Paste the following text
<VirtualHost *:80>
ServerAdmin admin@YOUR_DOMAIN_IP.com # YOUR_DOMAIN_IP – Server IP or DNS
name
ServerName YOUR_DOMAIN_IP.com
ServerAlias www.YOUR_DOMAIN_IP.com
DocumentRoot /var/www/html/prestashop

<Directory /var/www/html/prestashop>
Options +FollowSymlinks
AllowOverride All
Require all granted
</Directory>

ErrorLog /var/log/apache2/prestashop-error_log
CustomLog /var/log/apache2/prestashop-access_log common
</VirtualHost>

# Enable the configuration and restart the webserver
sudo a2ensite prestashop.conf
sudo systemctl reload apache2
```

Слід додати, що YOUR_DOMAIN_IP зараз матиме вигляд IPv4 адреси. Як уже було згадано раніше, ми не купували Еластичний IP, тому наш публічний IP буде змінюватись кожного разу після зупинки нашого Тестового сервера. В результаті, ми маємо щоразу змінювати IP на новий в Prestashop конфігураційному файлі, що є не зручно.

Як ми справимось з цією проблемою, буде показано далі.

7. Далі необхідно пройти по адресу http://YOUR_DOMAIN_IP.com та слідувати інструкції по налаштуванню онлайн магазина.

Отже, в результаті ми створили онлайн магазин зображений на Рисунку 2.2.2.1 [8].

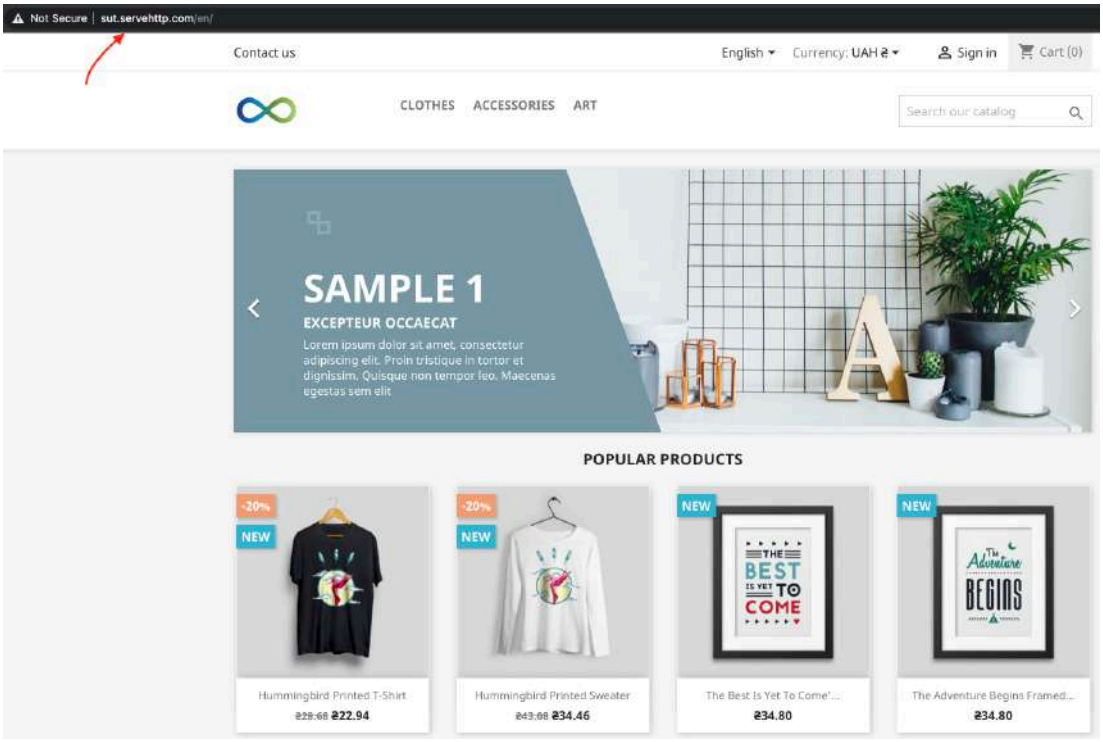


Рисунок 2.2.2.1 – Приклад онлайн магазину запущеному на Тестовому Стенді

На Рисунку 2.2.2.1 показано, що ми відкриваємо магазин не по IP адресу, а використовуючи DNS ім’я. Це вказує на те, що згадана на 6-му кроці проблема з динамічним IP була вирішена і не за рахунок купівлі доменного імені. Вирішити проблему допоміг сервіс no-ip [45] [9].

2.2.3 Написання серверних навантажувальних тестів у Gatling

Створимо проект Gatling, використовуючи систему для автоматичної збірки Gradle. У файлі build.gradle використовуватимемо наступні залежності:

```
plugins {
    id "com.github.lkishalmi.gatling" version '3.3.0'
    id "scala"
    id "idea"
    id "groovy"
    id "java"
}

ext {
    gatlingVersion = '3.3.0'
    scalaVersion = '2.12.8'
}

repositories {
    mavenCentral()
    jcenter()
}

dependencies {
    compile 'org.scala-lang:scala-library:'+scalaVersion
    compile "io.gatling.highcharts:gatling-charts-highcharts:"+gatlingVersion
    compile "org.codehaus.groovy:groovy-all:3.0.3"

    gatlingCompile "io.gatling:gatling-core:"+gatlingVersion
    gatlingCompile "io.gatling.highcharts:gatling-charts-
highcharts:"+gatlingVersion
    gatlingCompile("org.codehaus.groovy:groovy-all:3.0.3")
}
```

В результаті був створений проект зі структурою зображеною на Рисунку 2.2.3.1. Також на цьому рисунку присутні уже і файли з самим тестами, які ми розглянемо наступними.

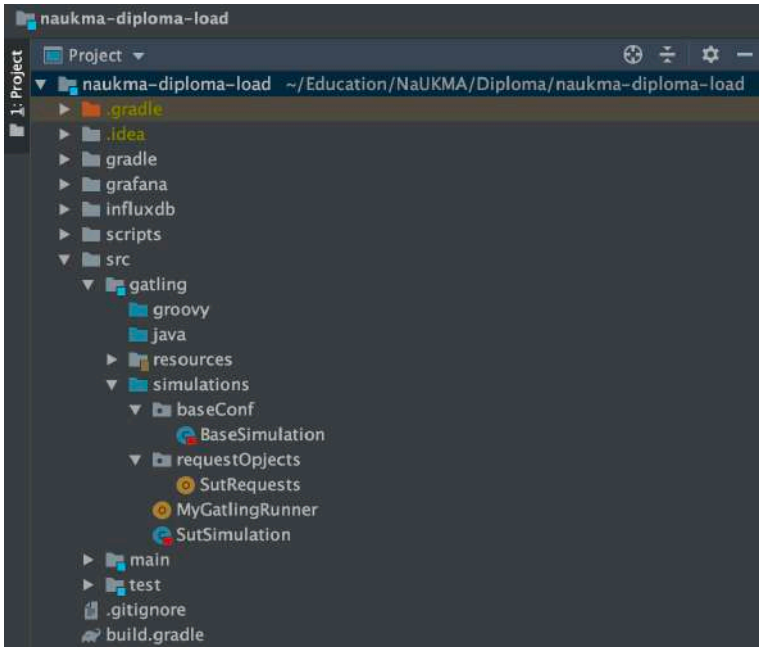


Рисунок 2.2.3.1 – Структура Gatling Проекту на базі Gradle

Тести навантажувальних сценаріїв виконані мовою Scala. До основних файлів тестового фреймворку відносяться, див. Таблиця 2.2.3.2.

Таблиця 2.2.3.2 – Основні файли навантажувальних тестів та їх функція

Назва файлу	Роль
SutSimulation.scala	Основний файл у якому описана логіка сценарії навантаження. У цьому файлі задається також кількість одночасних віртуальних користувачів, тривалість самого тесту та модель навантаження.
SutRequests.scalaa	Описує усі запити (сторінки), які використовуються під час тесту. Попередній файл імпортує вміст цього файлу та використовує його під час тесту. Загалом, це є Page Object підхід до організації тестів [15].
BaseSimulation.scala	У даному файлі прописаний основний URL-адрес та базові заголовки запиту. Його наслідує SutSimulation.scala файл.
MyGatlingRunner.scala	Файл запускає механізм за допомогою якого і здійснюється сам тест.

Вміст файлу SutSimulation.scala:

```
import io.gatling.core.Predef._

import scala.concurrent.duration._
import baseConfig.BaseSimulation
import io.gatling.core.structure.ScenarioBuilder
import requestOpjects.SutRequests

class SutSimulation extends BaseSimulation{

  /*** Before ***/
  before {
    println(«Load Test is starting …»)
  }

  val sutWebLoadScenario: ScenarioBuilder = scenario(«SUT Web Load»)
    .during(180 seconds) {
      exec(SutRequests.sutHome())
        .pause(1,2)
        .exec(SutRequests.sutClothes())
        .pause(2,3)
        .repeat(2)(
          exec(SutRequests.sutArt())
            .pause(2,3)
        )
        .repeat(2) (
          exec(SutRequests.sutSearch())
            .pause(2,3)
        )
        .exec(SutRequests.sutContactUs())
        .pause(1,2)
```

```
    }

  /*** Open Setup Load Simulation ***/
  setUp(
    sutWebLoadScenario.inject(
      rampUsers(5) during (10 seconds)
    ).protocols(httpConfSut)
    ).assertions(global.successfulRequests.percent.is(95))

  after {
    println(«… Load Test is finished.»)
  }
}
```

Як видно, файл із сценарієм розбитий на 3 основний частини:

1. ***Логіка***, ***що виконується безпосередньо до початку тесту***. У нашому випадку це просто виведення повідомлення про те, що тест розпочався. Часто, у складних сценаріях, тут може знаходитись логіка різної складності, наприклад, створення тестових даних.
2. ***Послідовність кроків самого сценарію***, який описує порядок дій віртуального користувача, а саме:
    a. Користувач відкриває Головну Сторінку та очікує від 1 до 2 секунд;
    b. Користувач відкриває сторінку з одягом «Clothes» та очікує від 2 до 3 секунд;
    c. Користувач відкриває сторінку з предметами мистецтва «Art» та очікує від 2 до 3 секунд;
    d. Користувач 2 рази підряд здійснює пошук, паузи між яким складають також від 2 до 3 секунд;
    e. Користувач відкриває сторінку з Контактами та очікує від 1 до 2 секунд;
3. ***Модель навантаження (setup)***, яка описує як має зростати навантаження та на якому рівні воно має утримуватись.

Розглянемо тепер вміст файлу SutRequests.scala:

```
package requestOpjects

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

```
object SutRequests {

  def sutHome() =
    exec(
      http(«SUT Main Page»)
        .get(«/»)
        .check(status.is(200))
    )

  def sutClothes() =
    exec(
      http(«SUT Clothes Page»)
        .get(«/en/3-clothes»)
        .check(status.is(200))
    )

  def sutArt() =
    exec(
      http(«SUT Art Page»)
        .get(«/en/9-art»)
        .check(status.is(200))
    )

  def sutSearch() =
    exec(
      http(«SUT Search»)
        .get(«/en/search?controller=search&s=best»)
        .check(status.is(200))
    )

  def sutContactUs() =
    exec(
      http(«SUT Contacts Page»)
        .get(«/en/contact-us»)
        .check(status.is(200))
    )
}
```

Даний файл описує динамічні URL для відкриття необхідної нам сторінки веб-сайту, а також очікуваний результат при навігації на ту чи іншу сторінку, а саме код відповіді 200.

Вміст файлу 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 httpConfSut: HttpProtocolBuilder = http
    .baseUrl(«http://sut.servehttp.com»)
    .inferHtmlResources(WhiteList(«».*sut.servehttp.com*»»»), BlackList())

.acceptHeader(«text/html,application/xhtml+xml,application/xml;q=0.9,image/avi
f,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9»)
    .acceptEncodingHeader(«gzip, deflate»)
    .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_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36»)
```

```
    .connectionHeader(«keep-alive»)
    .silentResources
}
```

Файл вказує основний URL для використання в тестах, базові заголовки запитів, а також деякі додаткові конфігурації по віртуальному користувачу. Наприклад, вказано те що тест має кожен раз додатково до відкриття сторінки, також робити запит на відповідні статичні файли – CSS, JS, картинки, тощо.

Після побудови проекту Gatling та написання базового сценарію навантаження, його потрібно провалідувати, запустивши невеликий тест, див. Рисунок 2.2.3.3.

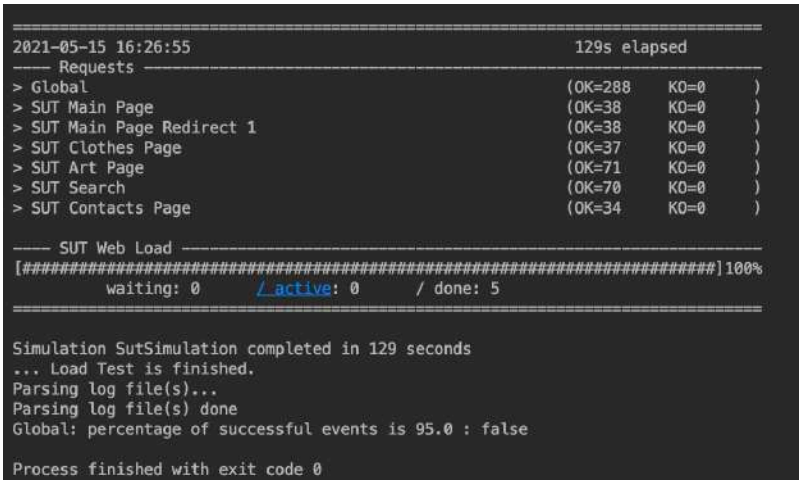


Рисунок 2.2.3.3 – Вивід консолі з результатами валідаційного навантажувального тестування

Як бачимо, з рисунку вище, валідаційний тест пройшов успішно, не показавши жодних помилок. Детальні результати даного тесту, дивіться в Додатку А.

Останнім кроком буде створення репозиторію для даного проекту на Github (у нашому випадку під назвою «naukma-diploma-load»), та заливання усіх тестів, файлів у створений репозиторій. Це дасть змогу потім для Jenkins мати доступ до проекту та використовувати його складові у своїх «Job».

2.2.4 Написання тестів з продуктивності браузера на Sitespeed.io

Даний інструмент – Sitespeed.io – використовується для тестування продуктивності веб-сторінки у браузері [47]. Його використання є доволі простим, адже майже усю роботу виконує образ Docker, якому необхідно лише передати правильні параметри.

Спочатку встановимо на нашу Менеджмент машину інструмент з контейнеризації Docker [10]:

```
# allow apt to use a repository over HTTPS
sudo apt-get update

sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

# Add Docker's official GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -
o /usr/share/keyrings/docker-archive-keyring.gpg

# set up the stable repository
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null

# install the latest version of Docker Engine and containerd
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io

# check docker install
docker --version
# Docker version 20.10.6, build 370c289
```

Є 2 шляхи виконання тестів за допомогою Sitespeed.io:

1. Передавати усі необхідні параметри в консолі – використовується для базових, простих сценаріїв.
2. Через створення JS файлу, у якому уже безпосередньо і прописується сценарій тесту. Використовується для складніших тестів, де присутні функціонал логіну або/та багато послідовних кроків у тесті.

Ми будемо використовувати 2й варіант. Отже, сам скрипт матиме вигляд (sut_client_performance_basic.js):

```
'use strict';

module.exports = async function (context, commands) {

  await commands.measure.start('SUT Main Page');
  await commands.navigate('http://sut.servehttp.com');
  await commands.measure.stop();

  try {
    await commands.measure.start('SUT Accessories');
    await commands.navigate('http://sut.servehttp.com/en/6-accessories');
    await commands.measure.stop();

   } catch (e) {
```

```
      throw e;
    }
};
```

У даному сценарії віртуальний користувач відвідує 2 сторінки – Головну та Аксесуари, на яких і відбувається тестування продуктивності.

Як і з навантажувальними тестами, тести продуктивності необхідно валідувати через їх запуск. Консольна команда для перевірки цього тесту складається з наступних команд та параметрів:

```
docker run --shm-size=1g --rm -v "$(pwd):/sitespeed.io"
sitespeedio/sitespeed.io:17.4.0 -b chrome sut_client_performance_basic.js --
multi
```

Як видно з команди, docker запускає образ sitespeed.io, який в свою чергу приймає на тест JS скрипт «sut_client_performance_basic.js», у якому прописаний сценарій самого тесту. Прогрес успішного тесту відображений на Рисунку 2.2.4.1

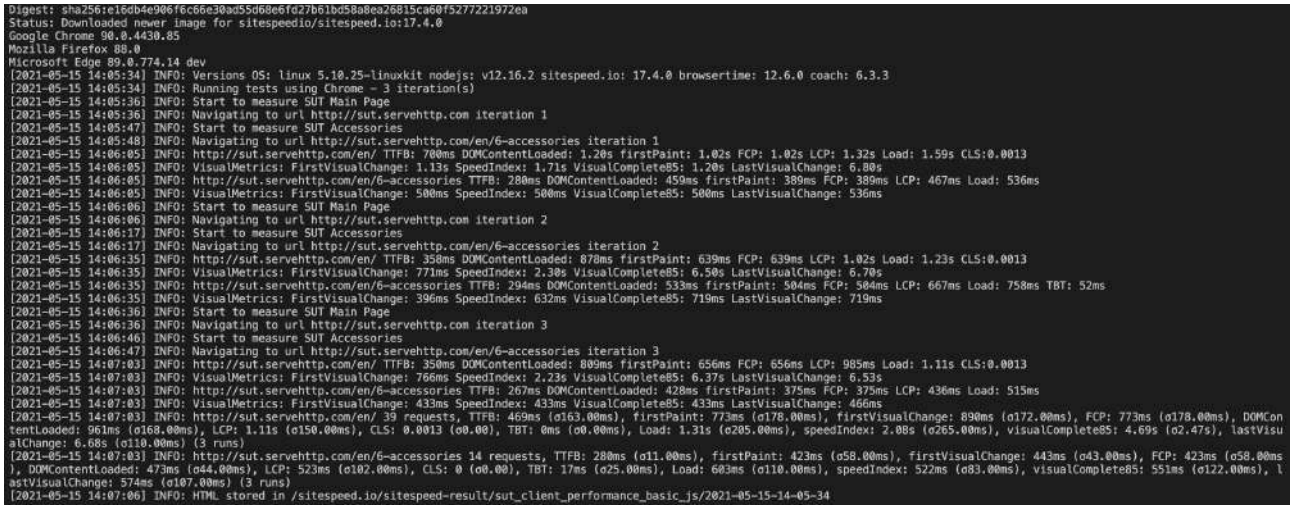


Рисунок 2.2.4.1 – Прогрес тесту продуктивності з Sitespeed.io

Детальний результат валідаційного тесту показаний у Додатку Б.

Додаємо даний проект, аналогічно про проекту Gatling (ми його назвали «naukma-diploma-client»), в окремий Github репозиторій.

2.2.5 Розробка процедури трансформації даних в Logstash

Gatling може бути інтегрований з базою тимчасових рядів **InfluxDB**, надсилаючи результати тестів у реальному часі під час виконання навантаження. Налаштувати інтеграцію можна з використанням протоколу graphite через файл конфігурації gatling.conf:

```
data {
```

```
writers = [graphite]
graphite {
  light = true
  host = "127.0.0.1"
  port = 2004
  protocol = "tcp"
  rootPathPrefix = "gatling"
  bufferSize = 8192
  writePeriod = 1
}
```

Але є одна проблема з якою ми стикнулися під час розробки розподіленої системи навантаження з використанням інструменту Gatling. Останній надсилає результати тестів з інтервалом у 1 секунду. У свою чергу, якщо база InfluxDB отримує дані з одними і тими ж назвами таг ключів (Tag Key), і навіть з різними значеннями полей (Field Key), але з однаковим часом (timestamp), вона просто їх перезаписує та залишає останні надіслані дані (див. структуру даних InfluxDB на Рисунку 2.2.5.1). Ба більше, команда Gatling зробила усе для того, щоб timestamp був однаковий по часу якнайчастіше, у випадку розподіленого навантаження, надсилаючи timestamp з точністю не до мілісекунд, а до секунд (див. Рисунок 2.2.5.2). Таким чином, Gatling мотивує використовувати його платну версію для розподіленого навантаження.

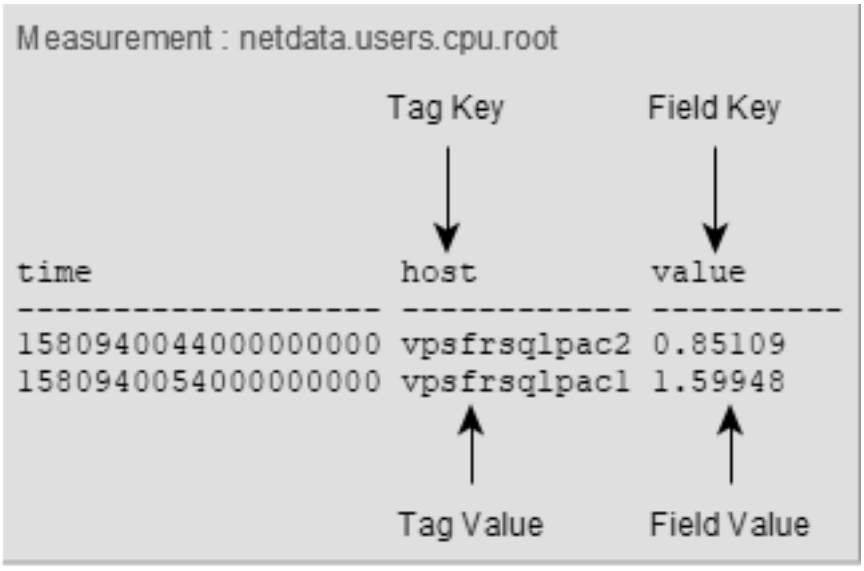


Рисунок 2.2.5.1 – Структура даних бази даних InfluxDB

Також Gatling не дає можливості змінювати якимось чином структуру даних, надісланих до InfluxDB і можна сказати, що «зв’язує руки по всім фронтам».

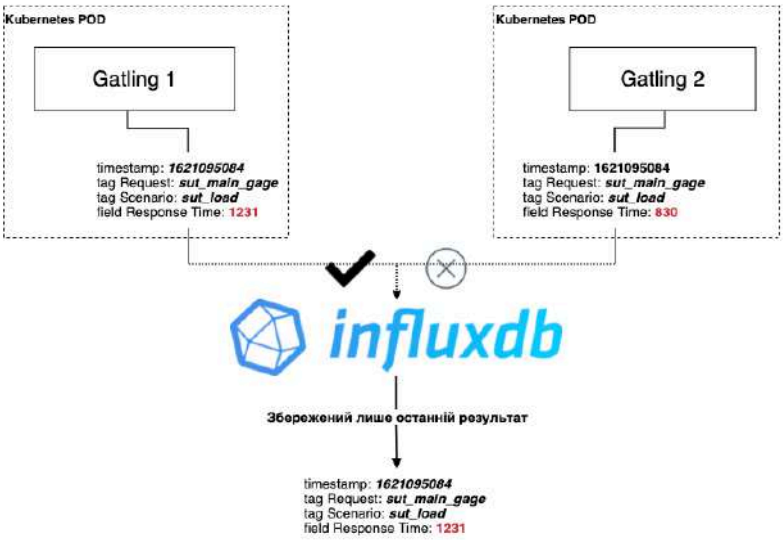


Рисунок 2.2.5.2 – Механізм збереження в InfluxDB у випадку збігу time та tag key

Одним із можливих виходів з даної ситуації є написання власного модуля всередині самої системи Gatling, який би дав змогу змінити структуру даних до необхідної чи розмірність timestamp до мілісекунд. Gatling – це інструмент з відкритим кодом і такі зміни потенційно можливі, а інколи навіть необхідні. Подібний підхід може дати свої плоди, але потенційно ненадовго. Як тільки вийде нова версія Gatling, швидше за все буде необхідно переписувати і свій модуль, або залишатися на старій, втративши можливість використовувати новий корисний функціонал.

Тому рекомендованим є рішення із використання такого інструменту як Logstash [43]. Цей інструмент є своєрідним трансформатором потоків даних. У нашому випадку необхідно, щоб Logstash додавав якийсь унікальний Tag Key, наприклад ім’я хоста з якого відбувається навантажувальне тестування. Тоді потік даних буде матиме вигляд зображений на Рисунку 2.2.5.3.

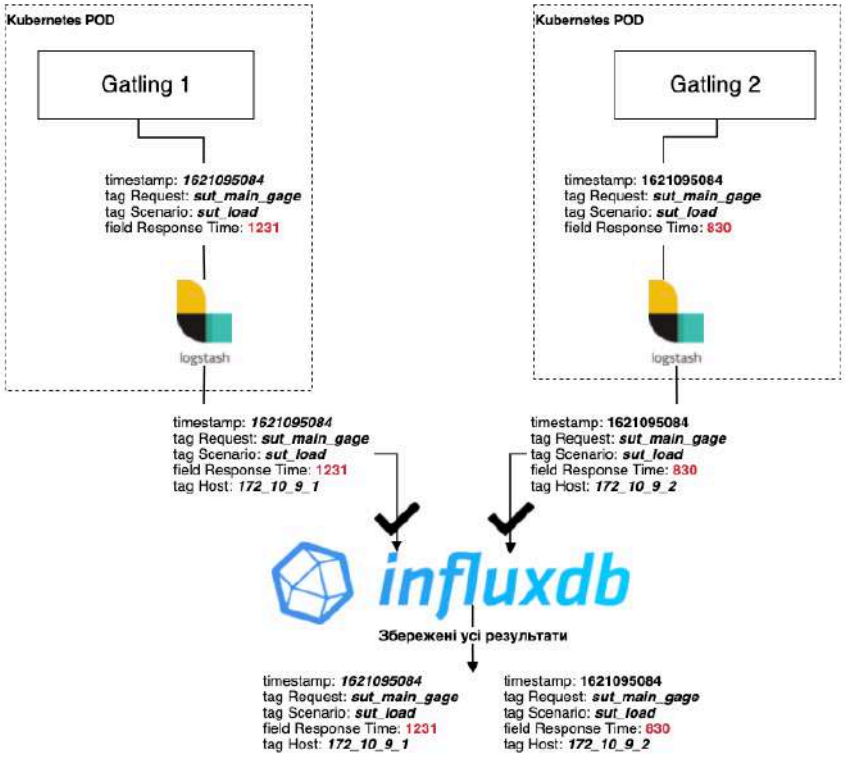


Рисунок 2.2.5.3 – Механізм збереження в InfluxDB за посередництва Logstash

Далі показуємо який саме скрипт був написаний для Logstash мовою Ruby, для того, щоб здійснити трансформацію даних на шляху від Gatling до InfluxDB.

```
input {graphite {
    port => 2004
  }}
filter {
  ruby {
    init => "require 'socket'"
    code => "
      host = Socket.gethostname

      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)
    "
```

```
  }
}
output {graphite {
    host => "INFLUX_HOST"
    port => INFLUX_PORT
  }}
```

Додамо даний файл у окремий проект та створимо для його запуску окремий Dockerfile. Це дасть змогу нам створити Docker образ для Logstash, який пізніше буде використовуватись у кластері Kubernetes. Dockerfile матиме наступний вигляд:

```
FROM docker.elastic.co/logstash/logstash-oss:7.12.0
RUN rm -f /usr/share/logstash/pipeline/logstash.conf

ADD pipeline/ /usr/share/logstash/pipeline/
ADD config/ /usr/share/logstash/config/
```

Створимо для цього проекту окремий Github репозиторій (наша назва «naukma-diploma-logstash»), уже третій, враховуючи репозиторії для проектів Gatling, та Sitespeed.io.

Додатково, необхідно змінити структуру (pattern) запису даних в самій InfluxDB – додати Tag Key “host”, змінивши налаштування запису у файлі influxdb.conf (див. Додаток В).

2.2.6 Створення кластеру Kubernetes

Ми підійшли до одного з найважливіших кроків на шляху реалізації системи розподіленого навантажувального тестування – це створення кластеру Kubernetes. Розпочнемо його налаштування та тестовий запуск. Усі подальші дії виконуватимуться на Менеджмент Сервері.

- Встановлення командної строки AWS CLI [18]

  Це єдиний інструмент для управління сервісами AWS, який дозволяє координувати сервіси з терміналу.

  ```
  curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
  "awscliv2.zip"
  unzip awscliv2.zip
  sudo ./aws/install
  ```

- Встановлення kubectl – це також інструмент командної строки але цього разу для управління кластерами Kubernetes.

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

- Встановлення утиліти kops – надбудова над kubectl, яка чудово працює з хмарними сервісами від AWS, та дозволяє швидко створювати, змінювати чи видаляти Kubernetes кластери [49].

```
curl -Lo kops
https://github.com/kubernetes/kops/releases/download/$(curl -s
https://api.github.com/repos/kubernetes/kops/releases/latest | grep
tag_name | cut -d '"' -f 4)/kops-linux-amd64
chmod +x kops
sudo mv kops /usr/local/bin/kops
```

- Створюємо нову AWS IAM роль під назвою k8s-role та прив'язуємо її до Менеджмент Сервера. Ролі надаємо наступні повні права доступу (див. Рисунок 2.2.6.1):
    - Route53
    - EC2
    - IAM
    - S3

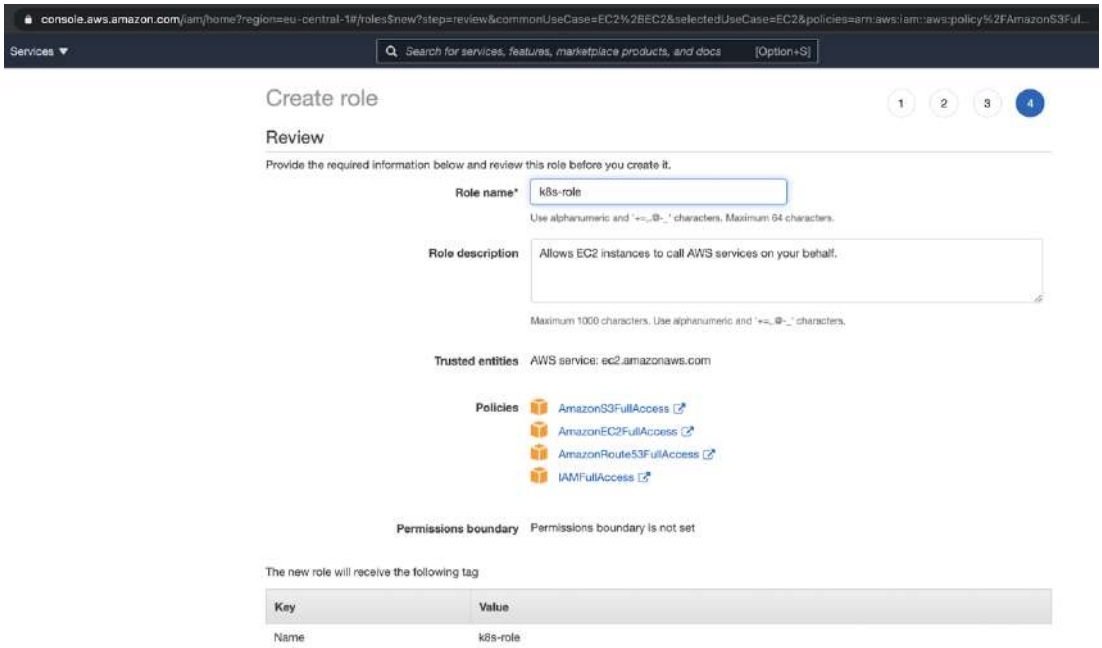


Рисунок 2.2.6.1 – Створення IAM ролі

- Конфігурація регіону для IAM ролі через команду:

```
aws configure

    AWS Access Key ID [None]:
    AWS Secret Access Key [None]:
    Default region name [eu-central-1]:
    Default output format [None]:
```

- Створюємо AWS Route53 приватну зону для хостингу (якщо є придбаний домен, можна створювати і публічну зону), див. Рисунок 2.2.6.2.

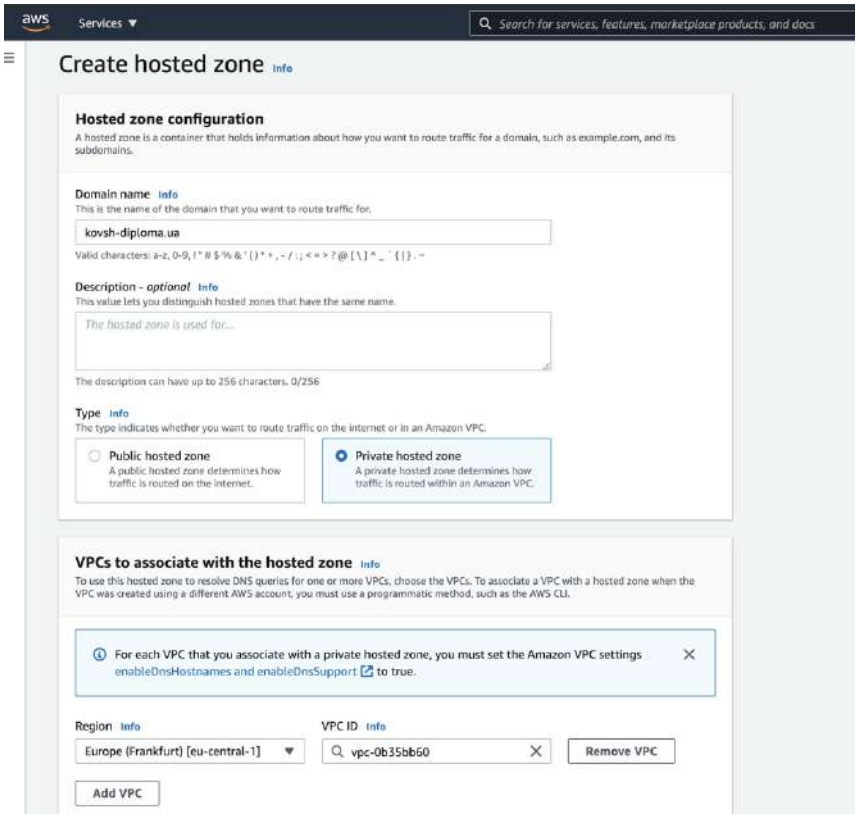


Рисунок 2.2.6.2 – Створення приватної зони хостингу у AWS Route53

- Створюємо сховище AWS S3:

```
aws s3 mb s3://k8s.kovsh-diploma.ua

    make_buket: k8s.kovsh-diploma.ua
```

- Створюємо перемінні середовища для використання Kubernetes:

```
# edit bashrc file in home directory
sudo nano ~/.bashrc

# add
export KOPS_CLUSTER_NAME=kovsh-diploma.ua
export KOPS_STATE_STORE=s3://k8s.kovsh-diploma.ua

# save and reapply the file in system
source ~/.bashrc
```

- Створюємо пару ключів SSH перед тим як створити кластер:

```
ssh-keygen
```

- Створюємо визначення кластеру у S3 бакет.

  Для цього створимо маніфест на основі якого і буде розгорнутий кластер.

```
# create cluster manifest
kops create cluster \
--cloud=aws \
```

```
--state=$KOPS_STATE_STORE \
--name=$KOPS_CLUSTER_NAME \
--node-count=3 \
--node-size=m5.xlarge \
--master-count 1 \
--master-size=t3.medium \
--zones=eu-central-1b \
--dns private \
--dns-zone=kovsh-diploma.ua \
--dry-run \
-o yaml > $KOPS_CLUSTER_NAME.yaml

# create secret
kops create secret --name $KOPS_CLUSTER_NAME sshpublickey admin -i
~/.ssh/id_rsa.pub

# get info about cluster
kops get --name ${KOPS_CLUSTER_NAME}
```

Зверніть увагу, з маніфесту вище видно, що ми створює кластер у якому:

- 1 мастер «Node» типу t3.medium, (4.0 GiB, 2 vCPUs)
- 3 воркер «Node» типу m5.xlarge (16.0 GiB, 4 vCPUs).

Архітектура загального вигляду нашого кластеру зображена на Рисунку 2.2.6.3.

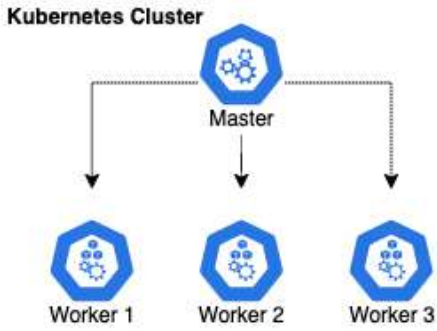


Рисунок 2.2.6.3 – Архітектура Kubernetes кластеру

Результатом створення кластеру буде наступний вивід у консолі Менеджмент Серверу:

```
ubuntu@ip-172-31-32-228:~$ kops create -f $KOPS_CLUSTER_NAME.yaml

Created cluster/kovsh-diploma.ua
Created instancegroup/master-eu-central-1b
Created instancegroup/nodes-eu-central-1b

To deploy these resources, run: kops update cluster kovsh-diploma.ua --yes
ubuntu@ip-172-31-32-228:~$ kops create secret --name $KOPS_CLUSTER_NAME sshpublickey admin -i ~/.ssh/id_rsa.pub
ubuntu@ip-172-31-32-228:~$ kops get --name ${KOPS_CLUSTER_NAME}
Cluster
NAME                    CLOUD   ZONES
kovsh-diploma.ua        aws     eu-central-1b

Instance Groups
NAME                    ROLE    MACHINETYPE     MIN     MAX     ZONES
master-eu-central-1b    Master  t3.medium       1       1       eu-central-1b
nodes-eu-central-1b     Node    m5.xlarge       3       3       eu-central-1b
```

- Тепер необхідно запустити сам кластер, для чого знову використаємо інструмент kops:

```
kops update cluster --name=$KOPS_CLUSTER_NAME -yes


kops export kubecfg --name $KOPS_CLUSTER_NAME --admin
```

```
Cluster is starting.  It should be ready in a few minutes.

Suggestions:
 * validate cluster: kops validate cluster --wait 10m
 * list nodes: kubectl get nodes --show-labels
 * ssh to the master: ssh -i ~/.ssh/id_rsa ubuntu@api.kovsh-diploma.ua
 * the ubuntu user is specific to Ubuntu. If not using Ubuntu please use the appropriate user based on your OS.
 * read about installing addons at: https://kops.sigs.k8s.io/operations/addons.

ubuntu@ip-172-31-32-228:~$ kops export kubecfg --name $KOPS_CLUSTER_NAME --admin
W0516 09:54:00.062653    6124 create_kubecfg.go:91] Did not find API endpoint for gossip hostname; may not be able to reach
 cluster
kops has set your kubectl context to kovsh-diploma.ua
```

- Тепер необхідно зачекати – від 10 до 20 хвилин, - після чого статус кластера можна перевірити наступними командами:

```
kops validate cluster


kubectl get nodes
```

```
ubuntu@ip-172-31-32-228:~$ k get nodes
NAME                                          STATUS   ROLES                  AGE     VERSION
ip-172-20-37-229.eu-central-1.compute.internal   Ready    node                3m3s    v1.20.6
ip-172-20-49-68.eu-central-1.compute.internal    Ready    control-plane,master   5m40s   v1.20.6
ip-172-20-52-173.eu-central-1.compute.internal   Ready    node                4m17s   v1.20.6
ip-172-20-56-20.eu-central-1.compute.internal    Ready    node                2m28s   v1.20.6
ubuntu@ip-172-31-32-228:~$
ubuntu@ip-172-31-32-228:~$ kops validate cluster
Validating cluster kovsh-diploma.ua

INSTANCE GROUPS
NAME                    ROLE    MACHINETYPE     MIN     MAX     SUBNETS
master-eu-central-1b    Master  t3.medium       1       1       eu-central-1b
nodes-eu-central-1b     Node    m5.xlarge       3       3       eu-central-1b

NODE STATUS
NAME                                          ROLE    READY
ip-172-20-37-229.eu-central-1.compute.internal   node    True
ip-172-20-49-68.eu-central-1.compute.internal    master  True
ip-172-20-52-173.eu-central-1.compute.internal   node    True
ip-172-20-56-20.eu-central-1.compute.internal    node    True

Your cluster kovsh-diploma.ua is ready
```

- Перевіримо тепер консоль AWS, де побачимо скільки EC2 віртуальних серверів у нас активних після запуску K8S кластеру (див. Рисунок 2.2.6.4).

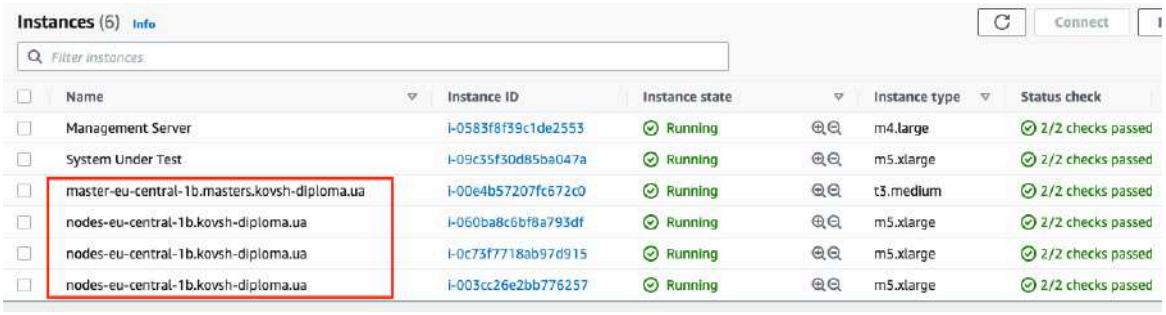


Рисунок 2.2.6.4 – Усі активні EC2 віртуальні сервери після запуску Kubernetes Cluster

- Залишилось налаштувати та запустити Панель для моніторингу Kubernetes.

```
# deploy the Kubernetes dashboard

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/dep
loy/recommended/kubernetes-dashboard.yaml

# find the admin user's password
kops get secrets kube --type secret -o plaintext

#  type the following command
kubectl proxy --address 0.0.0.0 --accept-hosts '.*' &
```

В результаті можемо відрити панель моніторингу Kubernetes за URL: http://${management_server_ip}:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/ (див. Рисунок 2.2.6.5)

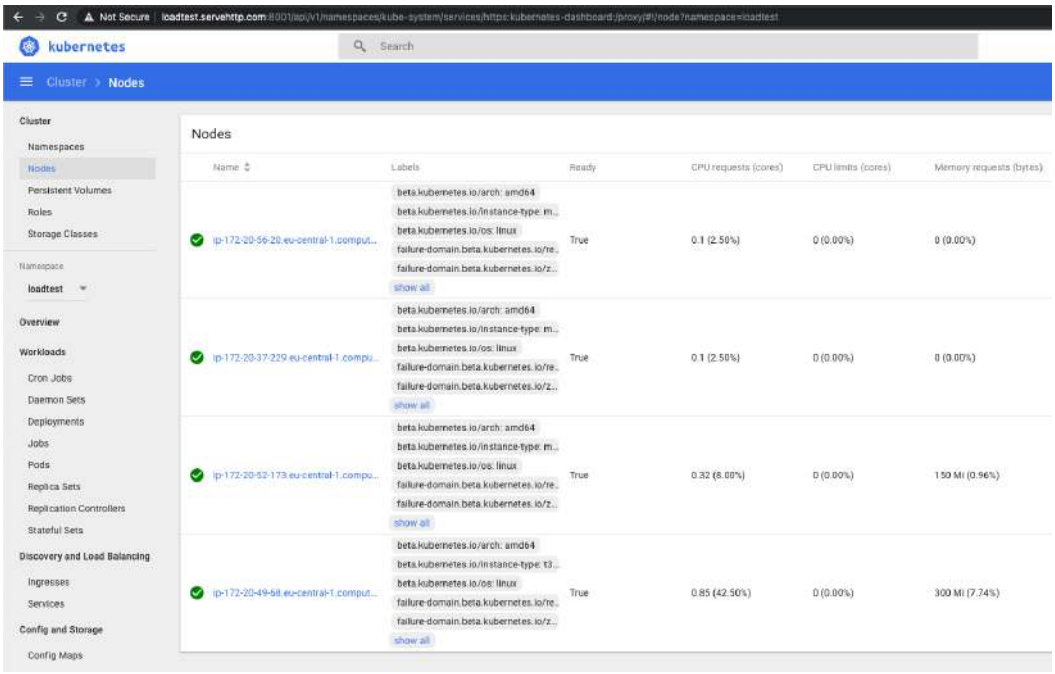


Рисунок 2.2.6.5 - Панель моніторингу статусу Kubernetes

2.2.7 Створення Kubernetes Job для тестування

Після того як був запущений кластер Kubernetes, можна підійти до створення самої Kubernetes «Job». Але давайте спочатку розглянемо що це за об’єкт.

Kubernetes надає змогу створити різні об’єкти для реалізації різних цілей. До основних можна віднести [41]:

- Pod – базова одиниця для поставки, яка містить один або більше контейнер.
- ReplicaSet – підтримує кількість запущених «Pods».
- Deployment – описує процес поставки та оновлення «Pods».
- Service – відкриває доступ до Pods чи цілого Deployment, присвоюючи IP та порт.
- Job – виконує один або більше Pod як завдання до його завершення.
- CronJob – виконує «Job» по вказаному графіку.
- Namespace – дозволяє організувати ресурси в окремі групи.

Ми перерахували далеко не усі можливі Kubernetes об’єкти, наприклад, можна також ще вказати такі як ConfigMap, Secret, Ingress, PersistentVolumeClaim, тощо.

У нашому випадку, об’єктом який допоможе реалізувати поставлене завдання буде «Job», адже навантажувальне тестування обмежене у часі. Тому необхідно, щоб Pod зупинив свою роботу по виконанню завдання, що об’єкт «Job» і дає змогу зробити. Також, ми будемо використовувати об’єкт «Namespace» для того щоб відокремити систему навантаження у кластері від інших потенційних система.

Таким чином, об’єкт Namespace який дозволяє нам створити «loadtest» збережений у файлі під назвою «namespace.yaml» та має наступний вигляд:

```
apiVersion: v1
kind: Namespace
metadata:
  name: loadtest
```

Водночас, об’єкт «Job», що запускає контейнер з навантажувальними тестами, збережений у файлі під назвою «loadtest-job.yaml» та описаний наступним чином:

```
apiVersion: batch/v1
kind: Job
```

```
metadata:
  namespace: loadtest
  name: loadtest-run
spec:
  parallelism: 2
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      shareProcessNamespace: true
      containers:
        - name: logstash
          image: 139389346425.dkr.ecr.eu-central-1.amazonaws.com/diploma-
logstash:latest
          resources:
            requests:
              cpu: 500m
              memory: 1024Mi
            limits:
              cpu: 1024m
              memory: 1024Mi
          ports:
          - containerPort: 2003
        - name: loadtest
          image: 139389346425.dkr.ecr.eu-central-1.amazonaws.com/diploma-
gatling:latest
          resources:
            requests:
              cpu: 1
              memory: 2048Mi
            limits:
              cpu: 2
              memory: 4096Mi
```

Як видно з опису об’єкта, в одному «Pod» у нас запускається відразу 2 контейнера:

1. Gatling
2. Logstash

Тобто уточнена архітектура Kubernetes «Pod» матиме вигляд, див. Рисунок 2.2.7.1.

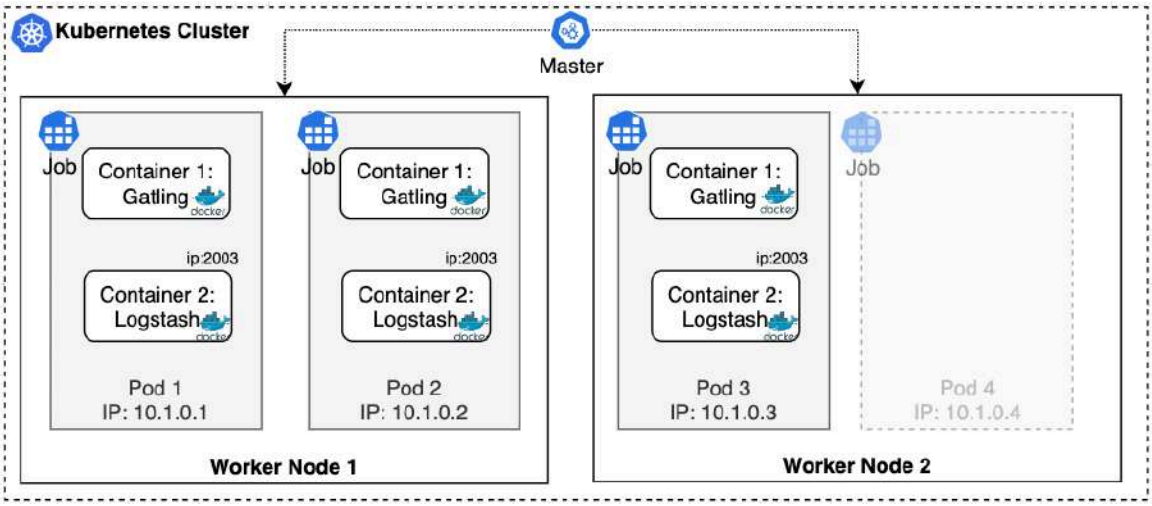


Рисунок 2.2.7.1 – Kubernetes «Jobs» для навантажувального тестування та їх вміст

Параметр namespace вказує яку групу використовуємо, в нашому випадку це «loadtest». А параметр name дає назву самій «Job», у нас це «loadtest-run».

Дуже важливим параметром в описі «Job» для нас є **«parallelism»**. Саме значення цього параметру вказує на те скільки паралельних «Job» нам запускати в кластері. Ми будемо працювати з цим параметром також і для того, щоб збільшувати чи зменшувати кількість «Job» під час проведення самого тесту навантаження.

Потім іде опис безпосередньо обох контейнерів – Gatling та Logstash. Кожному з них відкриваються порти за необхідності. Нам необхідний відкритий порт лише для Logstash, адже саме на цей порт Gatling надсилає результати тестів з інтервалом в 1-ну секунду.

Також кожному контейнеру вказані обмеження по ресурсам, надане ім’я та вказаний URL реєстру з образами Docker [29]. На останньому пункті – реєстр з образами – слід зупинитися більш детально, адже ми ще не створювали подібний. Зобразимо спочатку схему як Kubernetes поставляє образи контейнерів до «Pod», використовуючи для цього приватний Docker реєстр чи Docker HUB [29] (див. Рисунок 2.2.7.2).

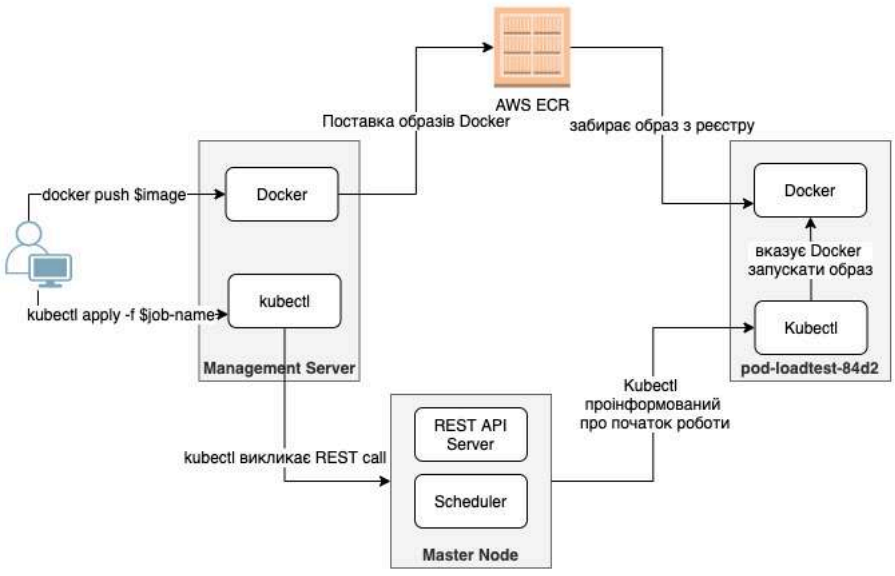


Рисунок 2.2.7.2 – Схема взаємодії Kubernetes кластеру та Docker реєстру у ECR

Як видно з Рисунку вище, ми не будемо використовувати Docker HUB, адже він має бути публічним або платним, тому ми створимо свій реєстр на базі AWS ECR (Elastic Container Registry) [20], див. Рисунок 2.2.7.3.

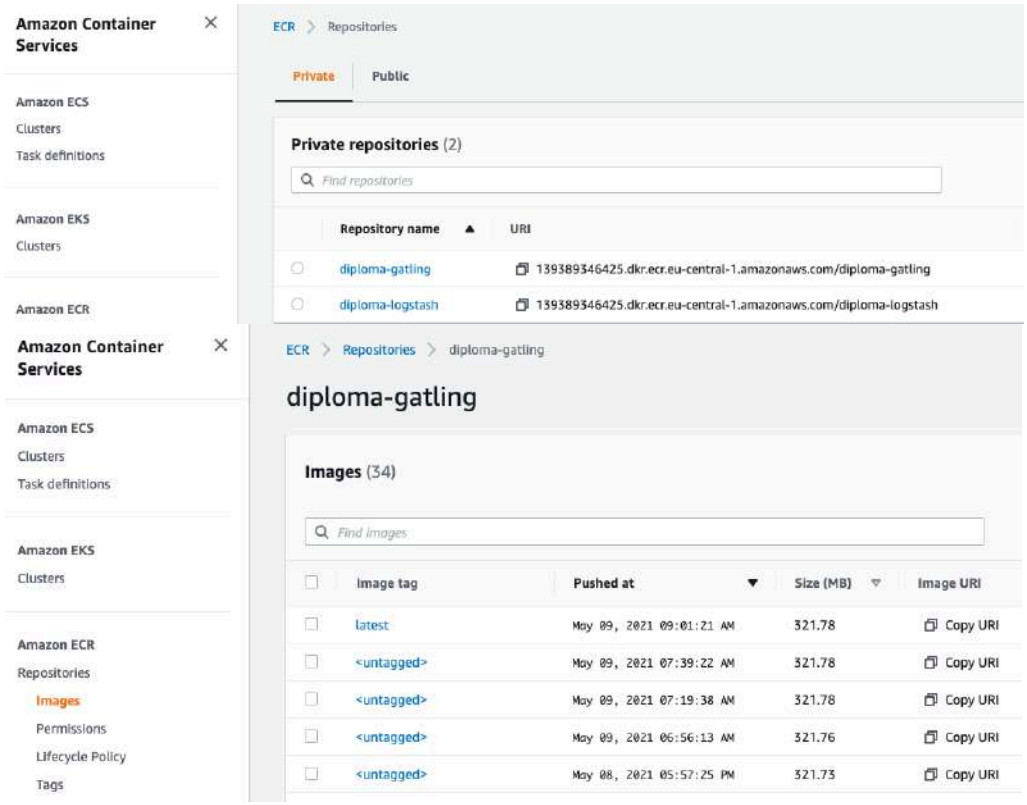


Рисунок 2.2.7.3 – AWS ECR з образами Gatling та Logstash

Розібравшись з тим звідки Kubernetes братиме образи для запуску «Job», перейдемо до самих контейнерів. У нас уже є створений контейнер Logstash у розділі 2.2.5. Разом з тим, у нас ще немає контейнера для Gatling тестів. Створимо його у корені проекту «naukma-diploma-load» під назвою Dockerfile, та додаємо наступний вміст:

```
FROM gradle:6.8.3-jdk8
LABEL maintainer="mykola kovsh"

WORKDIR /app

COPY . /app

RUN chmod +x /app/scripts/logstash_tracking_killing.sh
RUN chmod +x /app/scripts/start_load.sh

CMD /app/scripts/start_load.sh
```

Як бачимо, для старту контейнера в Dockerfile використовується файл «start_load.sh». Щоб зрозуміти навіщо, розглянемо його вміст також (файл збережений у папці ./scripts/ проекту «naukma-diploma-load»):

```
#!/bin/bash

sleep 30
# turn on bash's job control
set -m

# Start logstash tracking process in the background
# It should gracefully kill logstash after load test is finished thus its job
will be finished
/app/scripts/logstash_tracking_killing.sh &

# Start gatling load test
gradle gatlingRun-SutSimulation

fg %1
```

Поданий вище скрипт спочатку чекає 30 секунд, потім запускає в свою чергу ще один скрипт на задньому фоні - /app/scripts/logstash_tracking_killing.sh та паралельно сам Gatling тест із використанням Gradle [34]. Вміст файлу:

```
#!/bin/bash

sleep 30

echo BACKGROUND LOGSTASH JOB TRACKING PROCESS STARTED ...

while true
do

if [ $(pgrep -u root java | wc -l) -le 1 ]
then
  echo PIDs to be killed -15: $(pgrep java)
  kill -15 $(pgrep java)
  exit 0
fi

echo Active java PIDs: $(pgrep java)
# echo Still gatling process in progress: $(pgrep -u root java)

sleep 15

done
```

Така, доволі складна, залежність була зроблена для того щоб обійти певні обмеження об’єкту Kubernetes «Job». Зазвичай, таким об’єктом запускається (навіть рекомендується) запускати не більше одного контейнера. У нашому випадку ми запускаємо два – Logstash та Gatling. «Job» не дає можливість контролювати послідовність запуску цих контейнерів (як docker-compose наприклад) та починає їх роботу одночасно. І якщо Gatling почав тест першим за Logstash, він покаже, що з’єднання з останнім не встановлено і буде продовжувати тест, але не надсилаючи дані. Це перше обмеження яке ми

подолали, використовуючи додаткові скрипти. Так, як видно з скрипта /app/scripts/start_load.sh, він на початку очікує 30 секунд і аж потім запускає сам тест. Це дає змогу гарантувати, що контейнер Logstash буде запущеним і готовим до того, щоб Gatling встановив з ним з'єднання.

Друге обмеження яке допомогли обійти додаткові скрипти – це сама різна природа сервісів Gatling та Logstash. Так перший, якщо виконав тест, не зважаючи успішний чи ні, зупиняє свій процес, таким чином вказуючи Kubernetes «Job», що вона може бути закінчена. Водночас, Logstash продовжує свою роботу, адже це сервіс від якого очікується постійна доступність, але не у нашому випадку. Цей сервіс має закінчувати свою роботу по закінченню тесту, що дасть змогу «Job» бути виконаною. В іншому випадку, «Job» не закінчиться допоки ми її не зупинимо руками. Саме тому скрипт запускає на задньому фоні моніторинг активності Gatling процесу, і як тільки він закінчується, зупиняю роботу процесу Logstash. Це дає змогу в свою чергу вказати Kubernetes «Job», що вона може бути закінчена та успішно її завершити.

Отже результатами цього розділу було те, що ми додали наступні файли у «naukma-diploma-load» проект:

- ./Dockerfile – опис образу для запуску контейнера з навантажувальними тестами;
- ./app/scripts/start_load.sh – запуск та координація самого тесту і контейнерів Logstash та Gatling;
- /app/scripts/logstash_tracking_killing.sh – моніторинг контейнеру Logstash та його зупинка після закінчення роботи контейнера Gatling.

Також був створений 4-й Github проект «naukma-diploma-k8s», який містить наступні файли (детально описані в даному розділі дипломної роботи):

- namespace.yaml
- loadtset-job.yaml

В результаті, у нас буде 4 Github проекти, які допоможуть нам контролювати версії нашого коду та безпосередньо його доступністю з різних місць, у тому числі Jenkins (див. Рисунок 2.2.7.4)

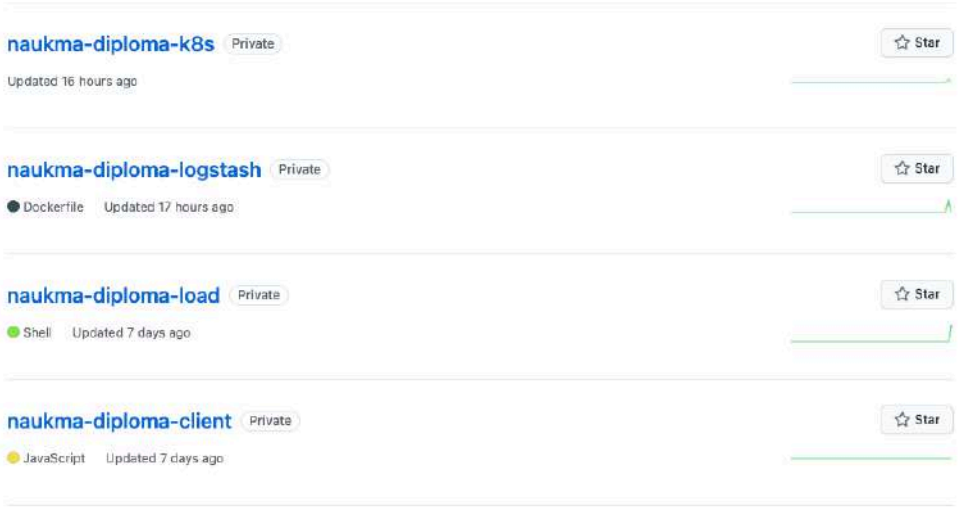


Рисунок 2.2.7.4 – Перелік репозиторіїв створених для підсистеми навантажувального тестування

Тепер ми можемо протестувати (валідувати) як «Job» будуть запускатися у нашому створеному кластері. Для простоти тесту запустимо 2 «Job». Для цього застосуємо наступні команди на Менеджмент Сервері.

```
# create namespace
kubectl apply -f namespace.yaml

# run load test
kubectl apply -f loadtest-job.yaml

# --- wait a little bit ---

# check namespace
kubectl get ns

# check nodes
kubectl get nodes

# check pods
kubectl get pods -n loadtest

# check jobs
kubectl get jobs -n loadtest

# check logs of job container loadtest
kubectl logs -f $pod_name -n loadtest -c loadtest

# check logs of job container logstash
kubectl logs -f $pod_name -n loadtest -c logstash

# clear all in loadtest namespace after test finished
kubectl delete –all all -n loadtest
```

В результаті виконання вказаних вище команд, під час перевірки нашого кластеру та «Jobs» які виконують навантажувальне тестування, ми отримаємо логи показані у Додатку Г.

2.2.8 Побудова моніторингу та звітності

Перед нами стояло завдання побудувати:

- моніторинг системних метрик для Тестового Стенду;
- звітність в реальному часі для навантажувальних тестів;
- звітність тестів продуктивності веб-сторінки у браузері.

Також нашим прагненням було звести всі згадані вище моніторинги та звітності в єдине місце. Для реалізації поставлених завдань ми обрали наступні технології:

- InfluxDB – база для збереження даних;
- Graphite – база для збереження даних від Sitespeed.io;
- Grafana – сервіс візуалізації;
- Telegraf – агент для збору метрик.

В результаті, архітектура пропонованої системи моніторингу та звітності матиме наступний вигляд, див. Рисунок 2.2.8.1.

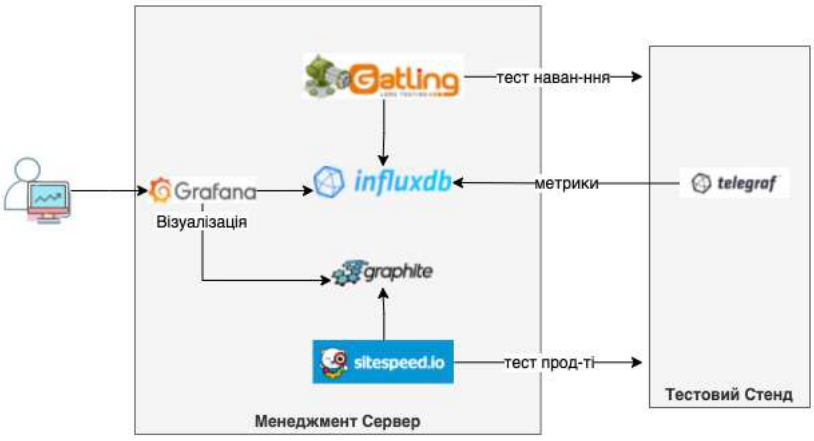


Рисунок 2.2.8.1 – Архітектура моніторингу та звітності

- Створимо docker-compose файл в якому розмістяться сервіси InfluxDB, Graphite, Grafana. Це дасть змогу підносити усі сервіси з одного місця та скоординовані між собою. Для зручності, ми додали цей файл у основний

наш проект «naukma-diploma-load» та назвали «docker-compose-with-graphite.yaml».

```
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
    restart: always

  grafana:
    build: grafana
    env_file: configuration.env
    container_name: 'grafana'
    hostname: grafana
    depends_on:
      - graphite
    links:
      - influxdb
      - graphite
    ports:
      - '3000:3000'
    volumes:
      - grafana_data:/var/lib/grafana
    restart: always

  graphite:
    image: sitespeedio/graphite:1.1.7-9
    hostname: graphite
    container_name: 'graphite'
    ports:
      - "2003:2003"
      - "8080:80"
    restart: always
    volumes:
      - whisper:/opt/graphite/storage/whisper

  grafana-setup:
    image: sitespeedio/grafana-bootstrap:17.0.0
    env_file: configuration.env
    links:
      - grafana


volumes:
  grafana_data: {}
  influxdb_data: {}
  whisper: {}
```

- Як видно, з docker-compose файлу, налаштування змінних відбувається в додатковому файлі configuration.env, який теж був створений в даному проекті.

```
####################
# Grafana options
####################
GF_SECURITY_ADMIN_PASSWORD=$password
GF_SECURITY_ADMIN_USER=#user
GF_AUTH_ANONYMOUS_ENABLED=false
GF_USERS_ALLOW_SIGN_UP=false
GF_USERS_ALLOW_ORG_CREATE=false
GF_INSTALL_PLUGINS=grafana-clock-panel,grafana-worldmap-panel,grafana-
piechart-panel,raintank-kubernetes-app

####################
# InfluxDB options
####################
INFLUX_USER=#user
INFLUX_PASSWORD=$password
INFLUX_DB=influx
GATLING_DB=graphite
TELEGRAF_DB=telegraf


####################
# Graphite options
####################
GF_PASSWORD=$password
GF_USER=#user
```

- Для Graphite ми будемо використовувати публічний образ з Docker HUB, тоді як образи для Grafana та InfluxDB ми створимо через Dockerfile, адже ці сервіси вимагають додаткових налаштувань, а саме:
    - Конфігурація InfluxDB
    - Конфігурація Data sources для Grafana
    - Конфігурація панелей для Grafana (Панель Моніторингу Тестового стенду, Панель Звіту результатів Gatling, Панель Звіту результатів Sitespeed.io)

  Додавши усі необхідні файли в наш проект, структура файлів моніторингу матиме наступний вигляд загалом, див. Рисунок 2.2.8.2.

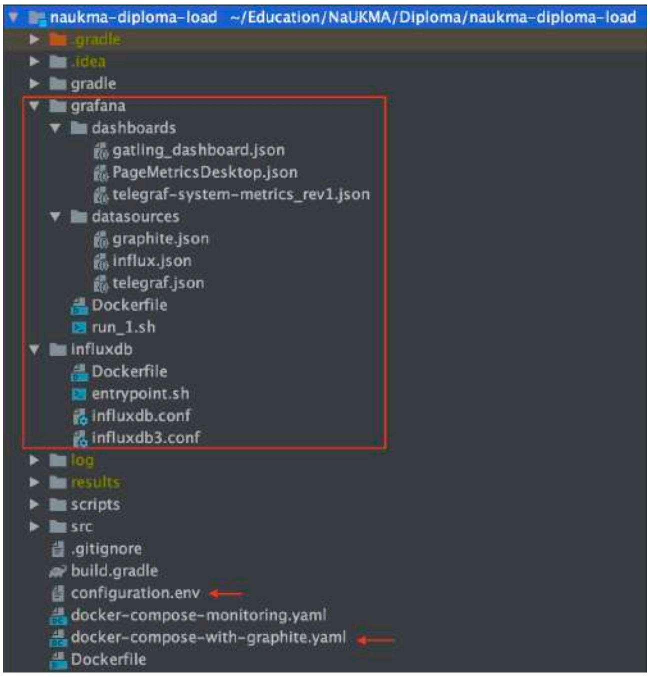


Рисунок 2.2.8.2 – Структура файлів для системи моніторингу та звіту в основному проекті

- Запустимо docker-compose файл на менеджмент сервері:

```
docker-compose -f docker-compose-with-graphite.yaml up -d --build
```

- До речі, ми також встановили на Менеджмент Сервер no-ip сервіс як було зроблено для Тестового Стенду в Розділі 2.2.2. Даний сервіс дозволяє налаштувати статичне доменне ім’я. Для менеджмент серверу ми створити ім’я loadtest.servehttp.com.
- Перейдемо тепер на сторінку Grafana по URL http://loadtest.servehttp.com:3000/ та перевіремо які там присутні панелі. Як видно з Рисунку 2.2.8.3, ми маємо усі 3 очікувані Панелі для моніторингу та звітності.

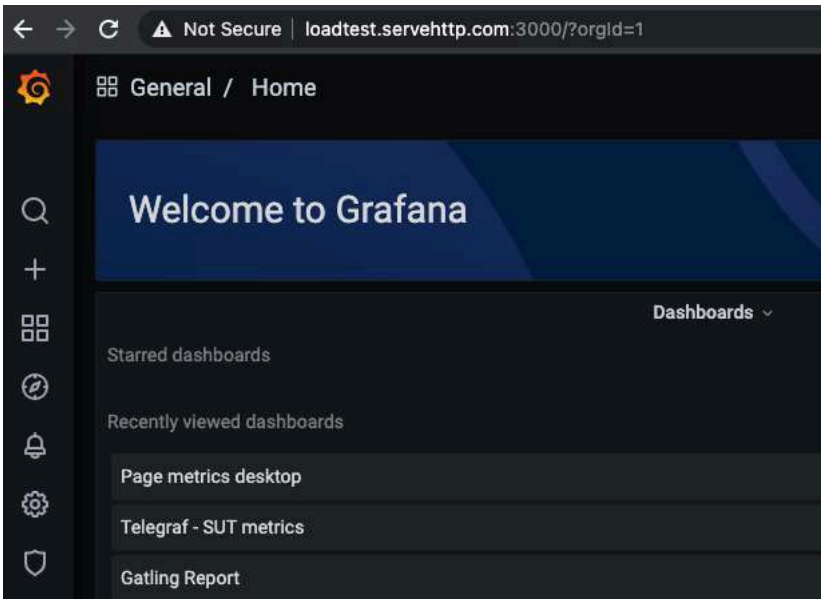


Рисунок 2.2.8.3 – Панелі звітності у сервісі Grafana

- Поки в цих панелях немає ніяких даних, адже інтеграції не налаштовані. Розпочнемо з встановлення агенту Telegraf на Тестовий Стенд та додавання усіх необхідних інтеграцій:

```
# install
wget -qO- https://repos.influxdata.com/influxdb.key | sudo apt-key add
-
source /etc/lsb-release
echo "deb https://repos.influxdata.com/${DISTRIB_ID,,}
${DISTRIB_CODENAME} stable" | sudo tee
/etc/apt/sources.list.d/influxdb.list

# configure integration with influxDB
Sudo nano /etc/telegraf/telegraf.conf


# add config and save
# Global Agent Configuration
[agent]
  hostname = "hakase-tig"
  flush_interval = "15s"
  interval = "15s"


# Input Plugins
[[inputs.cpu]]
    percpu = true
    totalcpu = true
    collect_cpu_time = false
    report_active = false
[[inputs.disk]]
    ignore_fs = ["tmpfs", "devtmpfs", "devfs"]
[[inputs.io]]
[[inputs.mem]]
[[inputs.net]]
[[inputs.system]]
[[inputs.swap]]
[[inputs.netstat]]
[[inputs.processes]]
```

```
[[inputs.kernel]]

# Output Plugin InfluxDB
[[outputs.influxdb]]
  database = "telegraf"
  urls = [ "http://loadtest.serverhttp.com:8086" ]
  username = "$username"
  password = "$password"

# restart telegraf service
sudo systemctl restart telegraf.service
```

Та налаштуємо інтеграцію з InfluxDB через /etc/telegraf/telegraf.conf

- Тепер відкриємо Панель – Telegraf - SUT metrics. Як бачимо з рисунку 2.2.8.4 Telegraf почав надсилати метрики з Тестового Стенду. Тепер у нас є повноцінний моніторинг системи під навантаженням, що дасть нам змогу спостерігати за її станом.

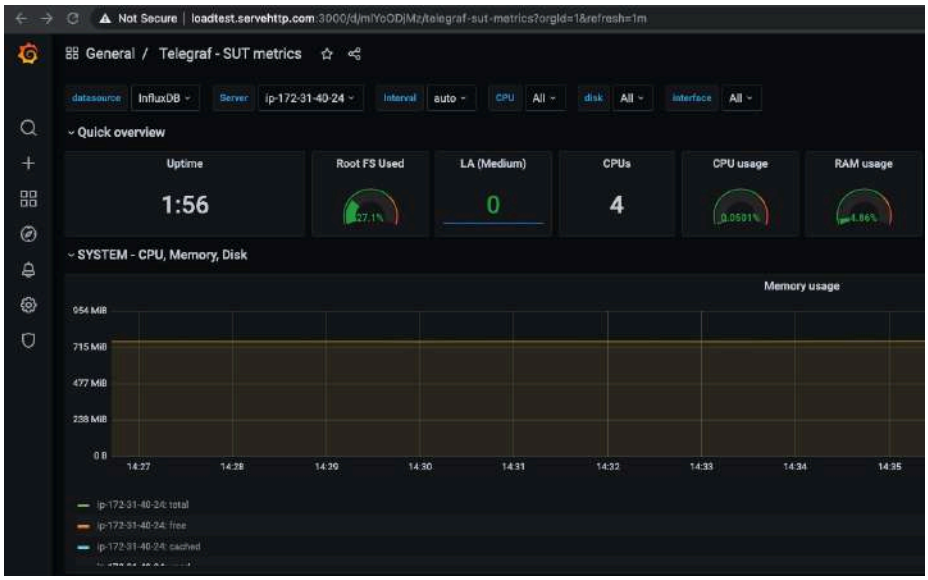


Рисунок 2.2.8.4 – Моніторинг Тестового Стенду

- Інтеграція навантажувальних тестів Gatling була продемонстрована в Розділі 2.2.5, коли ми розглядали Logstash. Для відображення цих результатів ми встановили та налаштували Панель – Gatling Report. Відображення результатів буде розглянуте в практичній частині дипломної роботи, в Розділі 3.
- Для того, аби відобразити результати тестів продуктивності веб-сторінки у браузері, була створена 3-тя панель Page metrics desktop. Щоб результати тестів відображались в даній панелі, при їх запуску необхідно передати хост та порт бази Graphite, наступним чином:

```
docker run --shm-size=1g --rm -v "$(pwd):/sitespeed.io" --network=host
sitespeedio/sitespeed.io:17.3.0 sut_client_performance_basic.js --
multi --graphite.host localhost --graphite.port 2003 -n 1 --
graphite.httpPort 8080 --graphite.status true  --graphite.addSlugToKey
true --slug firstView --cpu
```

Детально ця моніторингова Панель також буде розглянута в Розділі 3.

Таким чином, ми встановили систему моніторингу та звітності в реальному часі, яка дозволить нам слідкувати за станом Тестового Стенду та спостерігати результати навантажувальних тестів та тестів продуктивності веб-браузера в реальному часі.

2.2.9 Інтеграція з системою CI/CD Jenkins

Нам залишилось інтегрувати нашу підсистему навантажувального тестування з системою безперервної поставки коду Jenkins. Така інтеграція дасть змогу додати навантажувальні тести та тести продуктивності в процес розробки та поставки коду, отримувати завжди актуальні тестові результати та заздалегідь реагувати на негативні зміни з продуктивністю системи [11].

- Встановимо спочатку саму систему Jenkins на Менеджмент Сервері:

```
# install java
sudo apt update
sudo apt-get install default-jdk -y

# install maven
sudo apt install maven -y

# install jenkins
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-
key add -
echo deb http://pkg.jenkins.io/debian-stable binary/ | sudo tee
/etc/apt/sources.list.d/jenkins.list
sudo apt update
sudo apt install jenkins -y

# proceed with URL server-ip:8080 and configure Jenkins
```

- Створимо 5 Jenkins «Job», з наступними цілями:
    1. Pipeline Job – build-gatling-image – збірка Gatling образу та збереження його в реєстрі ECR.
    2. Pipeline Job – build-logstash-image – збірка Logstash образу та збереження його в реєстрі ECR.
    3. Pipeline Job – load-test-run – запуск навантажувального тесту.

4. Pipeline Job – load-test-add-job – додавання більше віртуальних користувачів під час навантажувального тесту.
5. Pipeline Job – performance-client run – запуск тесту продуктивності веб-сторінки у браузері.

- Перша та друга «Job», є подібними за своєю суттю. Відрізняються лише адрес Github та ECR репозиторіїв. Послідовність дій цих «Job» наступний:
    - Створення образу з Dockerfile вигруженого з Github [32];
    - Завантаження образу в репозиторій AWS ECR.

Загальна архітектура має вигляд як зображено на Рисунку 2.2.9.1.

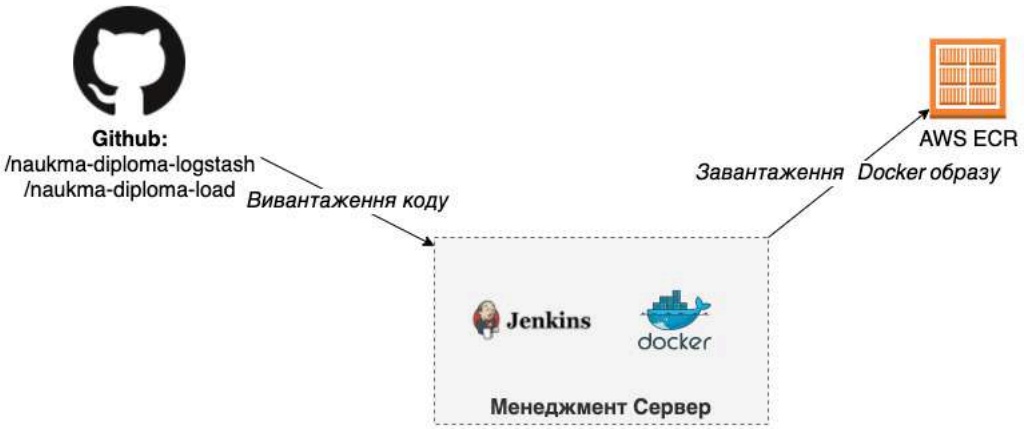


Рисунок 2.2.9.1 – Архітектура завантаження образів Gatling/Logstash системою Jenkins

- Pipeline Job – load-test-run має наступну конфігурацію:

```
pipeline {
    agent any

    stages {
        // Checkout Logstash from Github
        stage('Checkout K8S') {
            steps {
                checkout([$class: 'GitSCM', branches: [[name:
'*/main']], extensions: [], userRemoteConfigs: [[credentialsId:
'global', url: 'git@github.com:Kovshan/naukma-diploma-k8s.git']]])
            }
        }

        stage('Run K8S Load Job') {
            steps {
                withKubeConfig([credentialsId: 'credentialsId',
                    serverUrl: 'https://api.kovsh-diploma.ua',
                    namespace: 'loadtest'
                        ]) {
                            sh 'kubectl apply -f loadtest-job.yaml'
                }
            }
        }
```

```
        stage('Track and delete jobs') {
            steps {
                withKubeConfig([credentialsId: 'credentialsId',
                    serverUrl: 'https://api.kovsh-diploma.ua',
                    namespace: 'loadtest'
                        ]) {
                            sh '''#!/bin/bash
                            while true
                            do
                            sleep 45

                            completed_jobs=$(kubectl get job loadtest-
run -n loadtest -o jsonpath={.status.succeeded})
                            if [[ $completed_jobs -ge 1 ]]
                            then
                                kubectl delete --all all -n loadtest
                                exit 0
                            else
                                echo Load testing is in progress...
                                echo

                                echo K8S Load testing pods status:
                                kubectl get pods -n loadtest
                                echo

                                echo K8S Load testing jobs status:
                                kubectl get jobs.batch -n loadtest
                                echo

                                echo "View results by link:
http://loadtest.servehttp.com:3000/d/ydV3WAhmk/gatling-report?orgId=1"
                                echo
                            fi

                            done'''
                }
            }
        }
    }
}
```

Як бачимо, «Job» запускає Kubernetes «Job» в кластері та слідкує поки вони не закінчать тест. По закінченню, все в namespace «loadtest» видаляється.

- «Pipeline Job – load-test-add-job» має конфігурацію:

```
pipeline {
    agent any

    parameters {
        string(name: 'JOBS_NUMBER', defaultValue: "2", description:
'Number of K8S EXPECTED parallel jobs.')
        }

    stages {

        stage('Add K8S Load Job') {
            steps {
                withKubeConfig([credentialsId: 'credentialsId',
                    serverUrl: 'https://api.kovsh-diploma.ua',
```

```
                    namespace: 'loadtest'
                        ]) {
                            sh '''kubectl patch job loadtest-run -p
'{"spec":{"parallelism":'${JOBS_NUMBER}'}}' -n loadtest'''
                }
            }
        }

    }
}
```

Ця «Job» змінює конфігурацію (кількість) запущених «Job» через команду kubectl patch. Таким шляхом ми можемо збільшувати кількість віртуальних користувачів прямо протягом проведення самого тесту.

- Pipeline Job – performance-client-rub має конфігурацію:

```
pipeline {
    agent any

    stages {
        // Checkout Logstash from Github
        stage('Checkout repo') {
            steps {
                checkout([$class: 'GitSCM', branches: [[name:
'*/master']], extensions: [], userRemoteConfigs: [[credentialsId:
'global', url: 'git@github.com:Kovshan/naukma-diploma-client.git']]])
            }
        }

        stage('Run client performance test') {
            steps{
                script {
                    sh '''docker run --shm-size=1g --rm -v
"$(pwd):/sitespeed.io" --network=host \
                    sitespeedio/sitespeed.io:17.3.0
sut_client_performance_basic.js --multi \
                    --graphite.host localhost --graphite.port 2003 -n
1 --graphite.httpPort 8080 \
                    --graphite.status true  --graphite.addSlugToKey
true --slug firstView --cpu
                    '''
                }
            }
        }
    }
}
```

- Слід зазначити, що для того щоб сервіс Jenkins зміг запускати команди у нашому кластері, необхідно додати kubeconfig файл [5].

```
sudo cp ~/.kube/config ~jenkins/.kube/$ sudo chown -R jenkins:
~jenkins/.kube/
```

Також слід згенерувати Kubernetes реквізити для входу та додати їх у Jenkins [12].

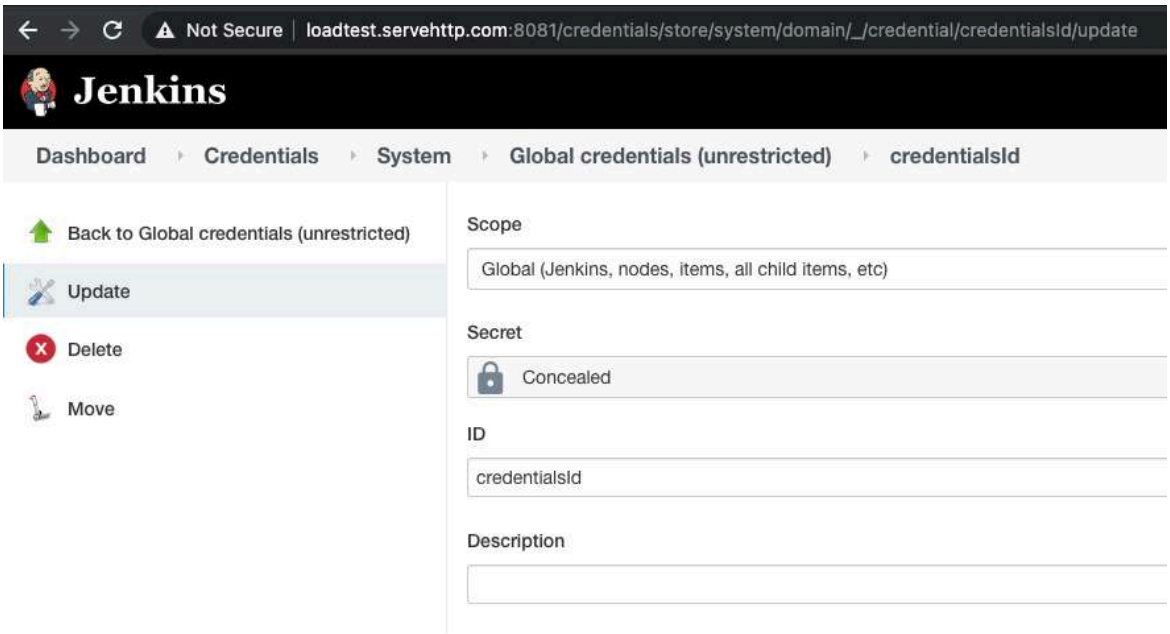
```
# Create a ServiceAccount named `jenkins-robot` in a given namespace.
$ kubectl -n <namespace> create serviceaccount jenkins-robot
```

```
# The next line gives `jenkins-robot` administator permissions for
this namespace.
# * You can make it an admin over all namespaces by creating a
`ClusterRoleBinding` instead of a `RoleBinding`.
# * You can also give it different permissions by binding it to a
different `(Cluster)Role`.
$ kubectl -n <namespace> create rolebinding jenkins-robot-binding --
clusterrole=cluster-admin --serviceaccount=<namespace>:jenkins-robot

# Get the name of the token that was automatically generated for the
ServiceAccount `jenkins-robot`.
$ kubectl -n <namespace> get serviceaccount jenkins-robot -o go-
template --template='{{range .secrets}}{{.name}}{{"\n"}}{{end}}'
jenkins-robot-token-d6d8z

# Retrieve the token and decode it using base64.
$ kubectl -n <namespace> get secrets jenkins-robot-token-d6d8z -o go-
template --template '{{index .data "token"}}' | base64 -d
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2V[.
..]
```

Тепер згенерований вище секретний ключ необхідно додати в Jenkins -> Manage Jenkins -> Manage Credentials -> Add Credentials -> Secret Text.



Як було видно з опису попередніх «Pipeline Jobs», цей ключ використовувався у Kubernetes плагіні в полі: withKubeConfig([credentialsId: 'credentialsId']).

Таким чином були створені та налаштовані 5 Jenkins Pipeline Jobs, як показано на Рисунку 2.2.9.2.

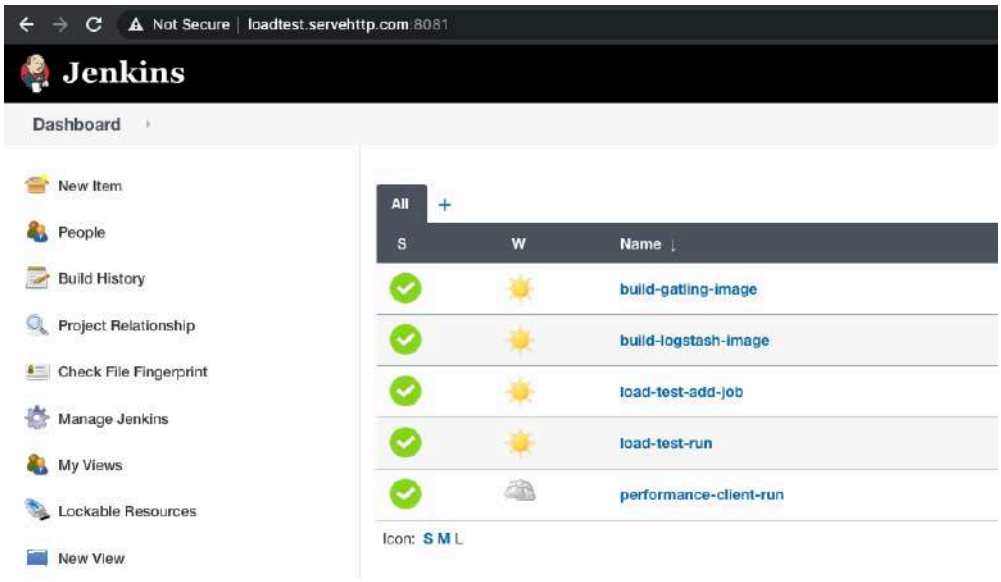


Рисунок 2.2.9.2 - Jenkins Pipeline Jobs для підсистеми навантажувального тестування

Тепер ми готові до здійснення самих навантажувальних тестів, а також тестування продуктивності веб-сайту у браузері, тому можемо переходити до практичної частини.

РОЗДІЛ 3: Випробування підсистеми тестування продуктивності

3.1 Розподілене навантажувальне тестування у CI/CD

Проведемо розподілене навантажувальне тестування за наступною моделлю (див. Рисунок 3.1.1).

Рисунок 3.1.1 – Модель навантаження

Для цього тесту використаємо 2 Kubernetes «Node». Отже, наші кроки наступні:

1. Запустимо Jenkins «Pipeline Job – build-gatling-image» та «build-logstash-image», на випадок, якщо відбулись зміни в коді цих сервісів. Дана збірка обновить образи сервісів Logstash та Gatling та додасть їх в реєстр AWS ECR.
2. Тепер можемо запустити безпосередньо сам тест через Jenkins «Pipeline Job – load-test-run». Як бачимо з логів Jenkins «Job» зображених нижче, у нас запущено 2 «Pods»:

```
+ kubectl apply -f loadtest-job.yaml
job.batch/loadtest-run created
[Pipeline] }
[kubernetes-cli] kubectl configuration cleaned up
[Pipeline] // withKubeConfig
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Track and delete jobs)
[Pipeline] withKubeConfig
[Pipeline] {
[Pipeline] sh
Load testing is in progress...

K8S Load testing pods status:
NAME                READY   STATUS    RESTARTS   AGE
loadtest-run-hkj91  2/2     Running   0          45s
loadtest-run-js4r2  2/2     Running   0          45s

K8S Load testing jobs status:
NAME           COMPLETIONS   DURATION   AGE
loadtest-run   0/1 of 2      45s        45s
```

Рисунок 3.1.2 – Логи виконання «Pipeline Job – load-test-run»

3. Перевіримо на скільки завантажені самі «Node».

   Як бачимо, CPU зайнято на рівні 40%, а оперативна пам'ять біля 20% на обох «Node».

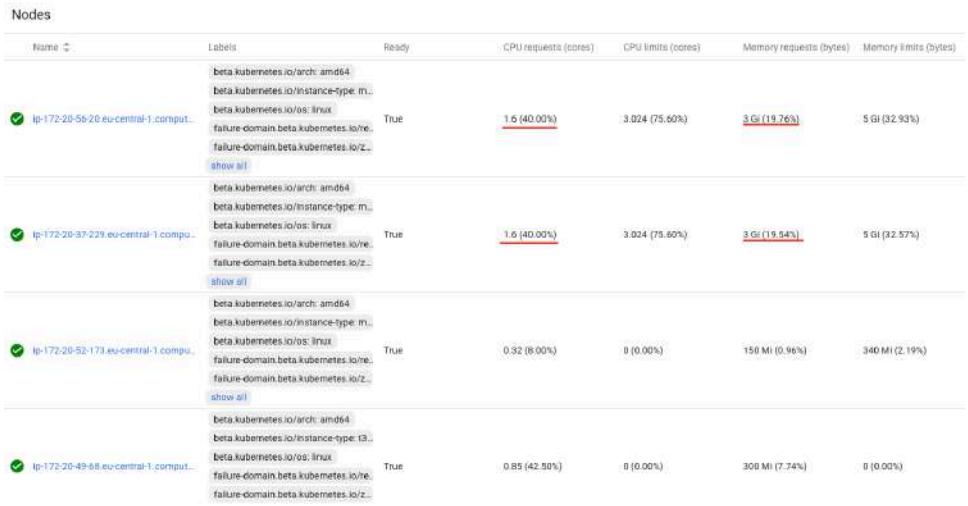


Рисунок 3.1.3 – Kubernetes панель моніторингу

4. Під час тесту ми можемо спостерігати за результатами в реальному часі, використовуючи для цього Grafana Панель – Gatling Report.

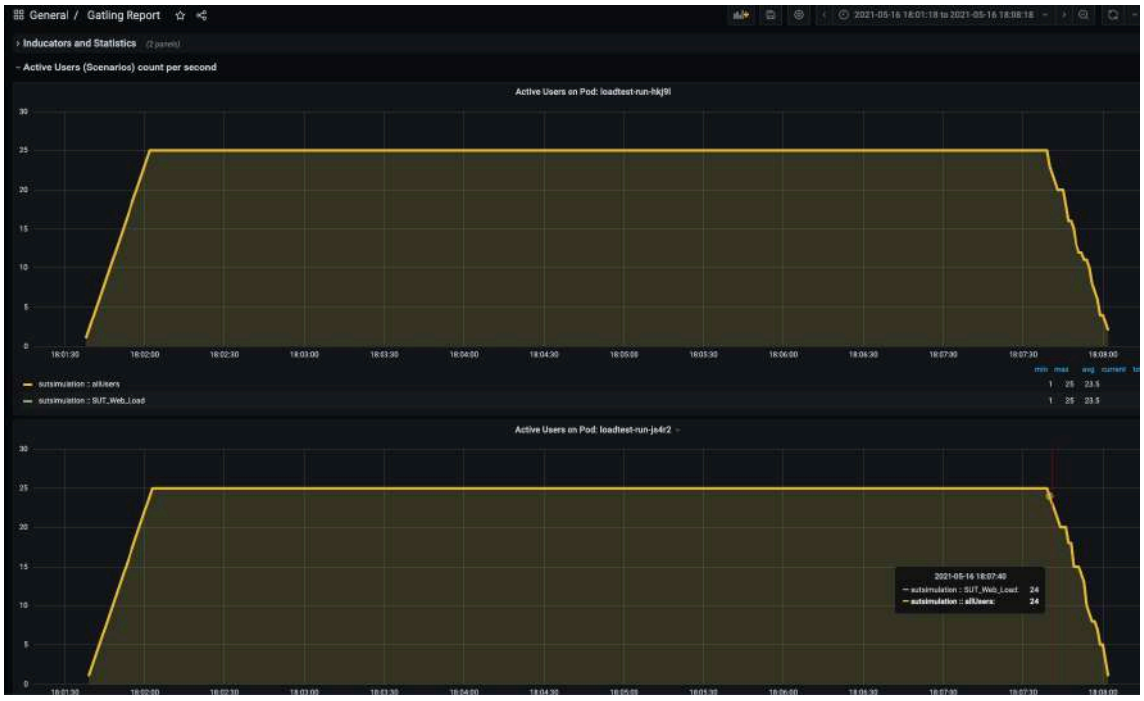


Рисунок 3.1.4 – Кількість активних користувачів під час тесту

Як видно, Панель налаштована таким чином, що дублює графіки навантаження для усіх «Node». В нашому випадку було 2 «nodes», тому 2 графіка, кожен з яких показує по 25 користувачів.

Також, можемо глянути на інші метрики Панелі.

Рисунок 3.1.5 – Час відгуку та пропускна здатність під час тесту

Як бачимо по метрикам навантаження, Час Відгуку 95% процентиль протягом усього тесту не перевищує 400 мілісекунд, що є позитивним результатом.

Загальне навантаження склало близько 4 запитів в секунду. Помилки не помічені.

5. Також корисним є спостерігати паралельно за метриками Тестового Стенду, використовуючи панель Telegraf – SUT metrics.

Рисунок 3.1.6 – Показники використання інфраструктури

Як бачимо з метрик Тестового Стенду, використання CPU зросло з менше чим 1% до близько 50%. Використання оперативної пам’яті зросло з близько 790 MiB до 1.23 GiB, тобто на 50%, але у нас всього на сервері 16 GiB. Тому ми робимо висновок, що навантаження у 50 одночасних віртуальних користувачів наш Тестовий Стенд витримує без будь-яких проблем.

По закінченню тесту, усі «Pods» та «Jobs» видаляються і система готова до нових завдань.

3.2 Розподілене тестування зі збільшенням навантаження під час тесту

В Розділі 3.2 ми побачили, що наш Тестовий Стенд може витримати більше чим 50 одночасних користувачів. В цьому розділі ми будемо симулювати ситуацію, коли ми розпочали тест з 60 користувачів і протягом спостереження вирішили збільшити навантаження ще на 50%, тобто 30 віртуальних користувачів (див. Рисунок 3.2.1). Це важливий функціонал, який став можливим лише за використання Kubernetes з додавання «Job Pods».

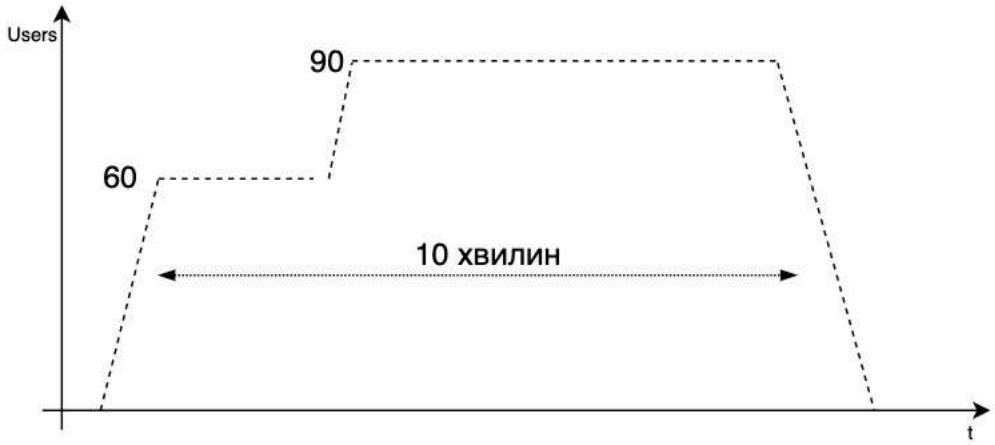
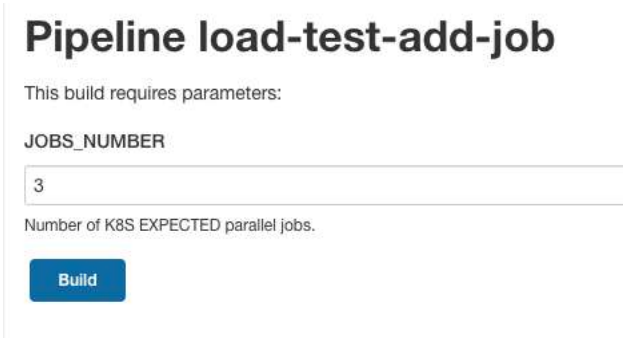


Рисунок 3.2.1 – Модель навантаження зі збільшенням користувачів

1. Запустимо знову «Jenkins Pipeline Job – build-gatling-image та build-logstash-image», на випадок, якщо знову відбулись зміни в коді.
2. Тепер знову сам тест через «Jenkins Pipeline Job – load-test-run». Перевіримо кількість «Jobs» та «Pods»:

```
ubuntu@ip-172-31-32-228:~$ k get pods -n loadtest
NAME                READY   STATUS    RESTARTS   AGE
loadtest-run-bhdh4  2/2     Running   0          2m46s
loadtest-run-rgv5q  2/2     Running   0          2m46s
ubuntu@ip-172-31-32-228:~$
ubuntu@ip-172-31-32-228:~$ k get jobs.batch -n loadtest
NAME           COMPLETIONS   DURATION   AGE
loadtest-run   0/1 of 2      2m53s      2m53s
```

3. Чекаємо коли підуть перші результати, і додаємо ще один «Pod» через Jenkins Pipeline Job – load-test-add-job. При запуску, вказуємо скільки «Job» ми хочемо бачити в результаті, враховуючи уже ті що були запущені.



Як бачимо, кількість «Pods» та «Job» зросли на одиницю відповідно.

```
ubuntu@ip-172-31-32-228:~$ k get pods -n loadtest
NAME                 READY   STATUS    RESTARTS   AGE
loadtest-run-bhdh4   2/2     Running   0          5m12s
loadtest-run-fkjph   2/2     Running   0          72s
loadtest-run-rgv5q   2/2     Running   0          5m12s
ubuntu@ip-172-31-32-228:~$ k get jobs.batch -n loadtest
NAME           COMPLETIONS   DURATION   AGE
loadtest-run   0/1 of 3      5m15s      5m15s
```

4. Цього разу у нас уже використані всі 3 Кластер «Node».

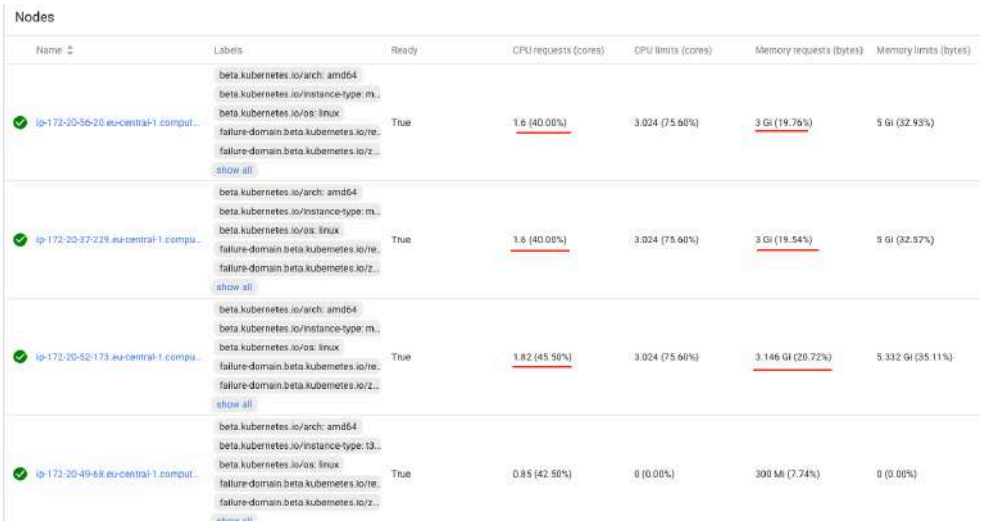


Рисунок 3.2.2 – Kubernetes панель моніторингу кластера

Це говорить про те, що ми маємо планувати розмір нашого кластера заздалегідь. Так, ми могли б його масштабувати, але це займе певний час, тому краще коли запасні ресурси є «під рукою».

5. Цього разу у нас було створено 3 графіки по користувачам, кожен з яких показував по 30. Як видно з самого нижнього графіку, це навантаження було додано пізніше, коли ми додали ще одну «Job».

Рисунок 3.2.3 – Кількість активних користувачів

Також можемо спостерігати зростання Часу Відгуки. Так при 60 користувачах, він становив біля 400 мс, тоді при 90 користувачах він зріс майже до 1,5 секунди. Тобто при зростанні навантаження на 50%, час відгуку зростає на 100%, що говорить про певні проблеми з Тестовим Стендом, тобто він перестає справлятися з таким навантаженням.

Рисунок 3.2.4 – Час відгуку під час тесту

Також ми могли спостерігати близько 15 запитів в секунду незалежно від того, що навантаження зросло. Це пояснюється тим, що при більшому

навантаженні, збільшився час відгуку, що і не дало зрости показнику Запити за Секунду.

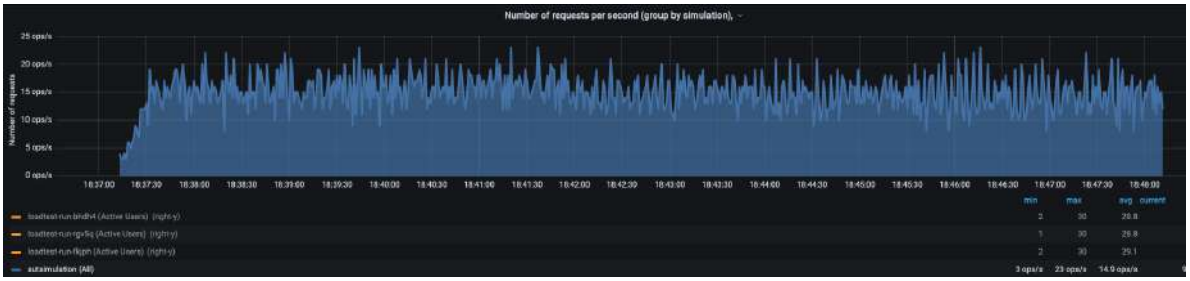


Рисунок 3.2.5 – Пропускна здатність під час тесту

6. Перевіримо тепер, що відбувалось на самому Тестовому Стенді.

Рисунок 3.2.6 – Показники використання ресурсів Тестового Стенду

Як можемо побачити, використання CPU підходило до 90% при збільшеному навантаженню, що тільки підтвердило наші здогадки з попереднього пункту про те, що Тестовий Стенд працював сповільнено.

Споживання оперативної пом’яті хоча і зросло, але її ще вдосталь.

Тому першу рекомендацію яку ми б дали розробникам після подібного тесту, це звернути увагу на процеси веб-сайту, які використовують найбільше CPU.

Отже, розподілене тестування з можливістю динамічного додавання навантаження, допомогло нам виявити границі нашого Тестового Стенду – це близько 100 віртуальних користувачів, та біля 20-ти операцій в секунду. Подальше збільшення навантаження потенційно призведе до відмови усього ресурсу.

3.3 Тестування продуктивності веб-сторінки у браузері

В попередніх розділах ми тестували навантаження на сервер. Зараз ми покажемо як відбувається тестування продуктивності веб-сторінки у браузері.

1. Запустимо Jenkins Pipeline «Job» - performance-client-run.

Рисунок 3.3.1 – Логи Jenkins Pipeline «Job» - performance-client-run

Як бачимо, наш тест відвідує 2 сторінки по 3 рази кожну та збирає метрики продуктивності в браузері, а саме:

- Головна Сторінка - http://sut.servehttp.com
- Сторінка Аксесуарів - http://sut.servehttp.com/en/6-accessories

2. Перевіримо результати в Grafana, але цього разу використовуючи третю нашу Панель Метрик – Page Metrics Desktop.

Бачимо такі результати для Головної Сторінки:

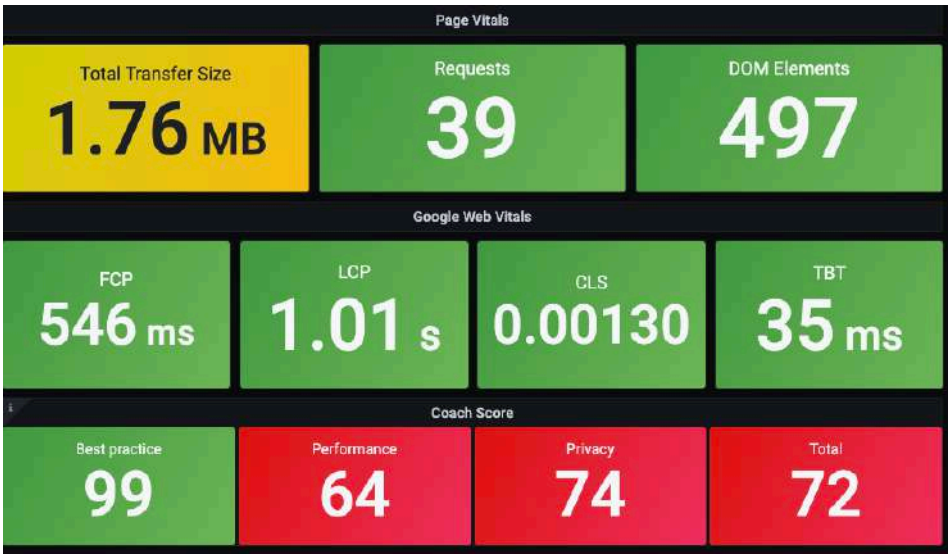


Рисунок 3.3.2 – Результати тестування продуктивності Головної Сторінки

Та для Сторінки Аксесуарів:

Рисунок 3.3.3 – Результати тестування продуктивності Сторінки Аксесуарів

Як бачимо, швидкість завантаження показав гарні результати. Так Largest Contentful Paint зайняв не більше 1 секунди для Головної Сторінки та біля 400 мс для Сторінки Аксесуарів.

Тоді як Speed Index склав 2.25с та 556мс, що також не так вже погано.

3. Тепер цікаво буде зібрати ці ж самі метрики, але під час серверного навантаження, та порівняти їх між собою, див Таблиця 3.3.4. Навантаження на сервер ми дали у 60 одночасних віртуальних користувачів, що є досить немало для нашого Тестового Стенду, та говорить про його завантаження близько на 60-70%.

Таблиця 3.3.4 – Порівняння метрик продуктивності до та під час навантаження серверу

Метрика	До навантаження	Протягом навантаження
Time To First Byte	138 мс.	240 мс.
First Visual Change	546 мс.	922 мс.
Largest Contentful Paint	975 мс.	1.09 с.
Visual Complete 85%	6.31 мс.	6.49 с.
Speed Index	2.25 с.	2.44 с.

Як бачимо з таблиці, навантаження на сервер збільшує також і очікування користувача сайту. Так, без навантаження, користувач починає бачити

перші зміни на сторінці уже через пів секунди, тоді як з навантаженням на сервер ці зміни відбуваються майже через секунду. Якраз на цих 500 мс зростає відповідь серверу під навантаженням (див Рисунок 3.3.5).

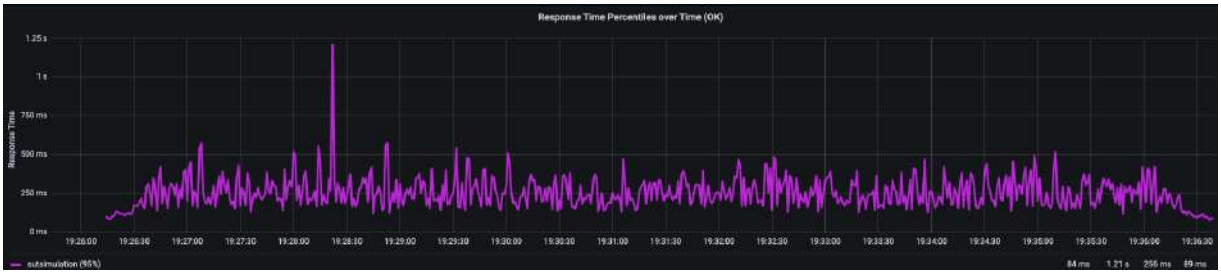


Рисунок 3.3.5 – Час відгуку сервере (95%) під навантаженням у 60 користувачів

Водночас, Speed Index зріс не так суттєво і склад 2.44 с. під час навантаження. Це говорить про те, що система мала затримку на сервері, і як тільки браузер отримав усі необхідні файли, він швидко їх опрацював та показав користувачу. Тобто можна зробити висновок, що Фронтенд системи є досить продуктивним.

Висновки по роботі та рекомендації для подальших досліджень

Перший розділ даної дипломної роботи був присвячений побудові практичної архітектури системи розподіленого тестування навантаження та тестування продуктивності сайтів через веб-браузер. Ми розглянули особливості запропонованої архітектури та методології Бенюх Л.І. під керівництвом Глибовця А.М. [1] та порівняли подібні системи від компаній Google [6] і Amazon [3]. Результатом даного розділу стала детальна архітектура з продуманими всіма технічними моментами реалізації та повним переліком використаного інструментарію.

У другому розділі ми безпосередньо будували прототип підсистеми розподіленого навантажувального тестування у інтеграції з системою безперервної поставки коду Jenkins. Спочатку був складений покроковий та послідовний план реалізації даної системи з подальшим детальним описом кожної дії. Побудова системи відбувалась на віртуальних машинах хмарного провайдера AWS, де ми розгорнули кластер Kubernetes засобами kops [49]. Далі був створений додатково Тестовий Стенд на якому і відбулась перевірка прототипу. І уже після ми побудували безпосередньо сам прототип, використовуючи такі інструменти як Gatling, Logstash, Sitespeed.io, InfluxDB, Grafana та інші. В результаті підсистема була інтегрована в CI/CD систему Jenkins, що дало змогу централізованого управління навантажувальними тестами. Слід додати, що усі використані інструменти під час побудови системи є безкоштовними і мають відкритий доступ до коду, звісно, крім інфраструктури (AWS).

Третій розділ описує наш процес перевірки самого прототипу. Це не було повноцінне високо-навантажувальне тестування, адже нашою ціллю були перевірка злагодженої роботи самої підсистеми, її масштабування в реальному часі, централізоване звітування результатів та інтеграція з CI/CD системою. Таким чином ми розпочали з невеликого відносно тесту – 60 віртуальних користувачів одночасно – і потім успішно масштабували його засобами Jenkins до 90 одночасних користувачів безпосередньо протягом виконання тесту. Також

ми перевірили та навіть порівняли результати – до та під час навантаження – тестування продуктивності сторінок Тестового Стенду в реальному часі.

Отриманий прототип для здійснення розподіленого навантажувального тестування показав себе як ефективний інструмент, який може буде успішно реалізований на проектах де є необхідність та економічна доцільність в здійснені масштабних навантажувальних тестів.

Для подальших досліджень ми б рекомендували здійснити наступне:

- Випробування даної системи в реальних умовах.
- Інтеграція усіх інструментів на базі кластеру Kubernetes. У поточній реалізації у кластер додані лише Gatling та Logstash. Тому ми б також рекомендували перенести до кластеру і інші інструменти – InfluxDB, Grafana, Sitespeed.io та Jenkins. Це б дало змогу оптимізувати процес запуску усієї системи в цілому.
- Випробування системи також із використанням інших хмарних провайдерів та власних машин.

У підсумку ми можемо сказати, що ми побудували повністю незалежну від комерційних інструментів систему для здійснення масштабованого навантажувального тестування, яка може бути реалізована практично на будь-якому проекті з такою потребою. Дана система допоможе налагодити процес постійного навантажувального тестування та таким чином попереджувати про потенційні проблеми з продуктивністю цифрових продуктів компанії.

Список літератури

1. Бенюх Л.І. під керівництвом Глибовця А.М., дипломна робота «*Розробка принципів, підходів та архітектури підсистеми для розподіленого навантажувального тестування та аналізу результатів у системі CI/CD*», 2021, Київ, НаУКМА. – С. 45-49, 53, 60-61
2. Augusto Andraus. *Distributed load testing with Gatling and Kubernetes.* [Електронний ресурс] – 2021 – Режим доступу: https://movile.blog/distributed-load-testing-with-gatling-and-kubernetes/
3. *AWS Labs. Distributed Load Testing on AWS.* [Електронний ресурс] – 2021 – Режим доступу: https://github.com/awslabs/distributed-load-testing-on-aws#distributed-load-testing-on-aws
4. Christo Petrov. *47 Amazon Statistics to Bedazzle You in 2021.* [Електронний ресурс] – 2021 – Режим доступу: https://techjury.net/blog/amazon-statistics/#gref
5. *Create A CI/CD Pipeline With Kubernetes And Jenkins.* [Електронний ресурс] – 2021 – Режим доступу: https://www.magalix.com/blog/create-a-ci/cd-pipeline-with-kubernetes-and-jenkins
6. *Distributed load testing using Google Kubernetes Engine.* [Електронний ресурс] – 2021 – Режим доступу: https://cloud.google.com/architecture/distributed-load-testing-using-gke
7. *Freemium*. [Електронний ресурс] – 2021 – Режим доступу: https://en.wikipedia.org/wiki/Freemium
8. Heyan Maurya. *How to install PrestaShop on Ubuntu 20.04 Server.* [Електронний ресурс] – 2020 – Режим доступу: https://www.how2shout.com/linux/how-to-install-prestashop-on-ubuntu-20-04-server/
9. *How to Install the Linux Dynamic Update Client on Ubuntu.* [Електронний ресурс] – Режим доступу: https://www.noip.com/support/knowledgebase/installing-the-linux-dynamic-update-client-on-ubuntu/
10. *Install Docker Engine on Ubuntu.* [Електронний ресурс] – 2021 – Режим доступу: https://docs.docker.com/engine/install/ubuntu/
11. *Install Jenkins on Ubuntu 18.0.4 | Setup Jenkins on AWS EC2 Ubuntu instance | How to setup Jenkins in Ubuntu EC2 instance?* [Електронний ресурс] –

2020 – Режим доступу: https://www.coachdevops.com/2020/04/install-jenkins-ubuntu-1804-setup.html

12. *Kubernetes CLI Jenkins Plugin.* [Електронний ресурс] – 2020 – Режим доступу: https://plugins.jenkins.io/kubernetes-cli/
13. *LAMP (software bundle).* [Електронний ресурс] – 2021 – Режим доступу: https://en.wikipedia.org/wiki/LAMP_(software_bundle)
14. Marko Luksa. *Kubernetes in Action.* Manning Publications. 2018 Print – С. 5-8, 15-21, 48-52
15. Martin Fowler. *PageObject*. [Електронний ресурс] – 2013 – Режим доступу: https://martinfowler.com/bliki/PageObject.html
16. *What is Kubernetes?* [Електронний ресурс] – Режим доступу: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes
17. Сайт хмарного провайдера Amazon Web Services: https://aws.amazon.com/what-is-aws
18. Сайт сервісу для доступу до AWS через консоль CLI: https://aws.amazon.com/cli
19. Сайт сервісу моніторингу машин AWS Cloudwatch: https://aws.amazon.com/cloudwatch
20. Сайт сервісу для зберігання контейнерів AWS Elastic Container Registry: https://aws.amazon.com/ecr
21. Сайт сервісу віртуальних машин AWS EC2: https://aws.amazon.com/ec2
22. Сайт з усіма типами віртуальних машин AWS EC2: https://aws.amazon.com/ec2/instance-types/
23. Сайт сервісу AWS Elastic IP: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html
24. Сайт сервісу розгортання кластеру Kubernetes від AWS EKS: https://aws.amazon.com/eks
25. Сайт сервісу для управління ролями та доступом AWS IAM: https://aws.amazon.com/iam
26. Сайт сервісу DNS AWS Route 53: https://aws.amazon.com/route53
27. Сайт сервісу для зберігання даних AWS S3: https://aws.amazon.com/s3
28. Сайт інструменту навантаження Blazemeter: https://www.blazemeter.com

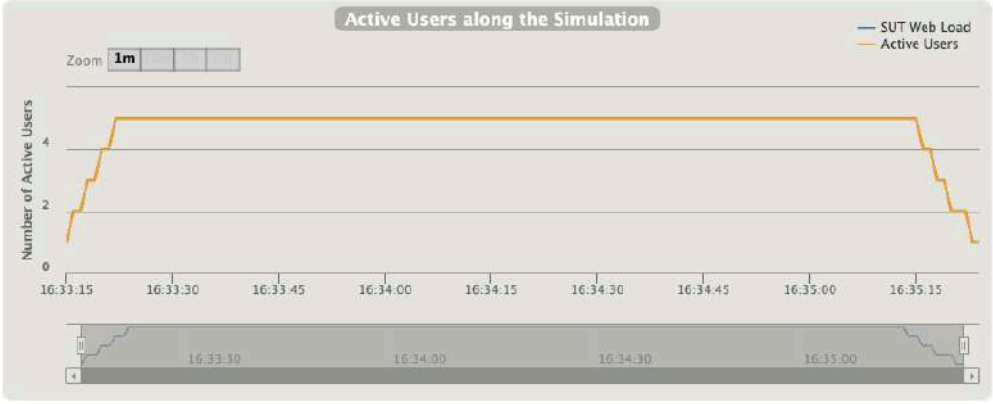
29. Сайт контейнерної технології Docker: https://www.docker.com
30. Сайт інструменту навантаження Gatling: https://gatling.io
31. Сайт інструменту навантаження Gatling Frontline: https://gatling.io/gatling-frontline
32. Сайт для централізованого зберігання коду GitHub: https://github.com/about
33. Сайт хмарного провайдера Google Cloud Platform: https://cloud.google.com/docs/overview
34. Сайт системи для збірки проекту Gradle: https://gradle.org
35. Сайт сервісу візуалізації Grafana: https://grafana.com
36. Сайт бази даних Graphite: https://graphite.readthedocs.io
37. Сайт бази даних InfluxDB: https://www.influxdata.com
38. Сайт сервісу безперервної поставки коду Jenkins: https://www.jenkins.io
39. Сайт інструменту навантаження JMeter: https://jmeter.apache.org
40. Сайт інструменту навантаження K6: https://k6.io
41. Сайт інструменту кластеризації Kubernetes: https://kubernetes.io
42. Сайт інструменту навантаження Loadero: https://loadero.com/home
43. Сайт інструменту трансформації даних Logstash: https://www.elastic.co/logstash
44. Сайт хмарного провайдер MS Azure: https://azure.microsoft.com/en-us/overview/what-is-azure
45. Сайт сервісу з видачі доменних імен noip: https://www.noip.com/
46. Сайт сервісу зі створення електронних магазинів Prestashop: https://www.prestashop.com/en
47. Сайт інструменту тестування продуктивності Sitespeed.io: https://www.sitespeed.io
48. Сайт агенту збору метрик Telegraf: https://www.influxdata.com/time-series-platform/telegraf
49. Сайт утиліти для управління Kubernetes з AWS - kOps - Kubernetes Operations: https://kops.sigs.k8s.io

Додатки

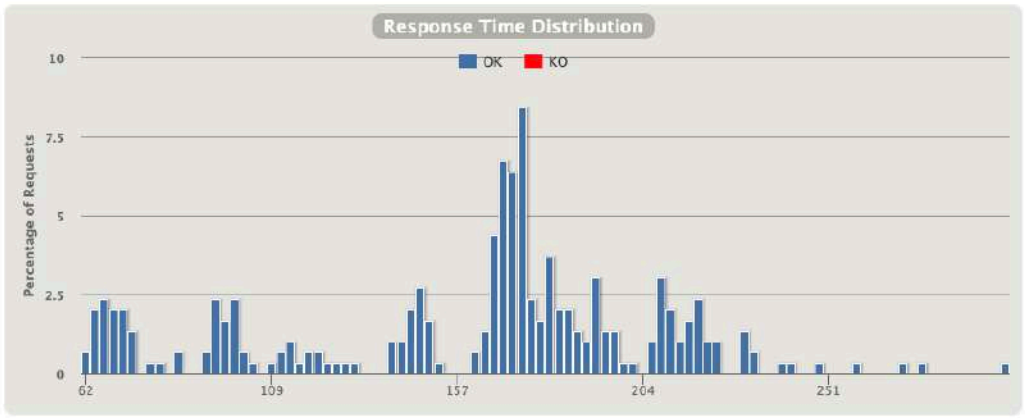
Додаток А

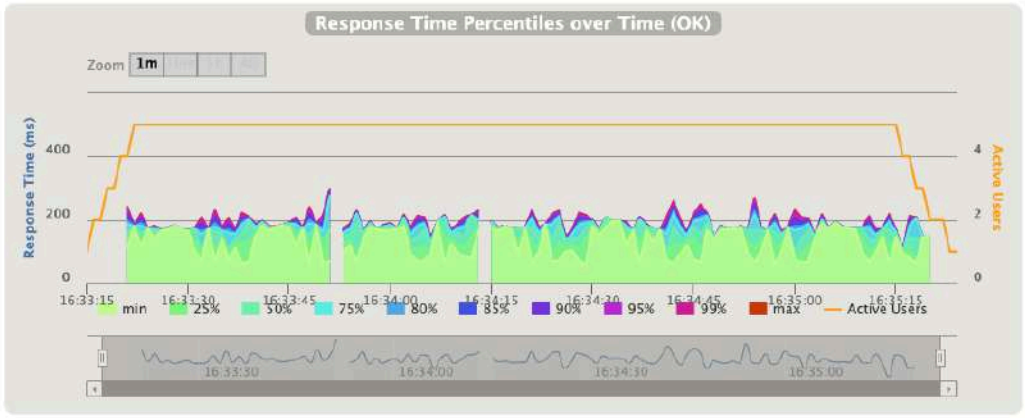
(обов’язковий)

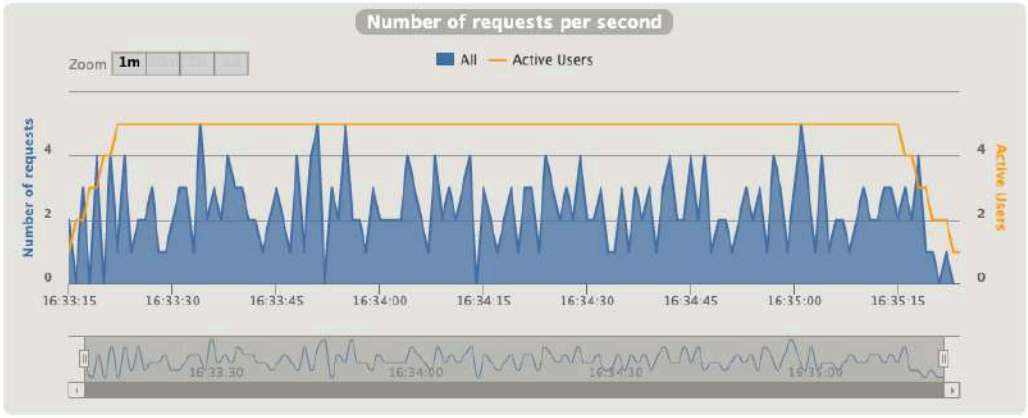
Результати перевірки базового навантажувального тесту з використанням інструменту Gatling

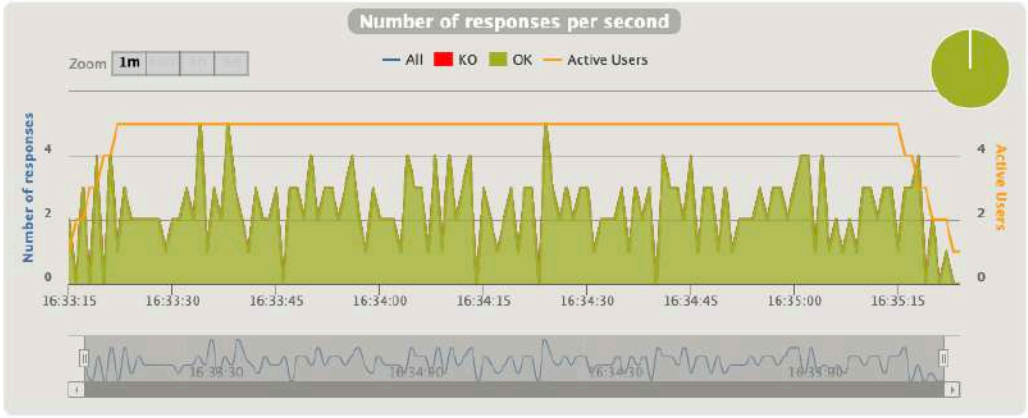


Продовження Додатку А





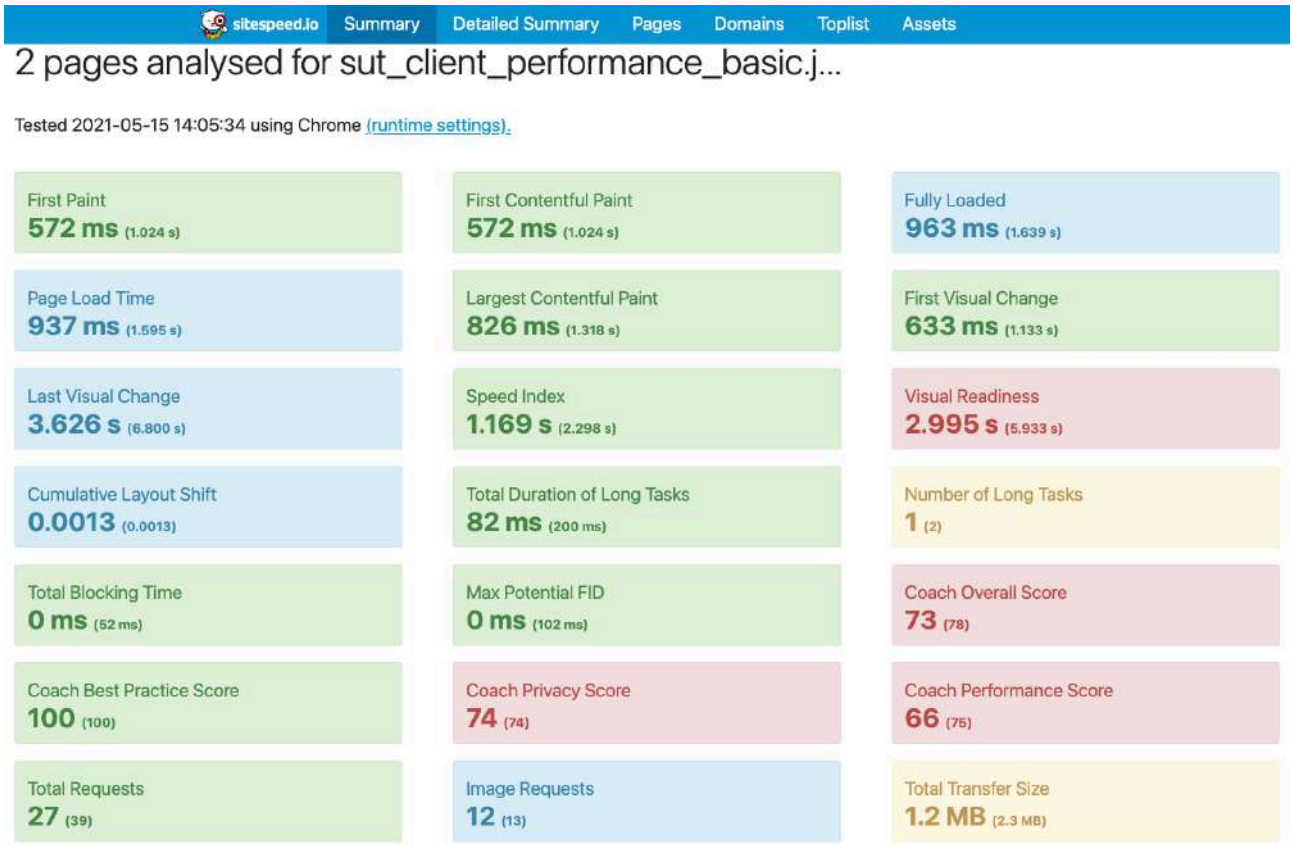




Додаток Б

(обов'язковий)

Результати перевірки базового тесту продуктивності з використанням інструменту Sitespeed.io



2 pages analysed for sut_client_performance_basic.j...

Tested 2021-05-15 14:05:34 using Chrome (runtime settings).

name	min	median	mean	p90	max
Coach score	68	73	73	78	78
Coach performance score	57	66	66	75	75
Privacy score	74	74	74	74	74
Best Practice score	99	100	100	100	100
Image requests	10	12	12	13	13
CSS requests	1	4	4	7	7
Javascript requests	1	7	7	12	12
Font requests	0	2	2	3	3
Total requests	14	27	27	39	39
Image size	71.8 KB	588.0 KB	588.0 KB	1.1 MB	1.1 MB
HTML size	53.8 KB	69.5 KB	69.5 KB	85.3 KB	85.3 KB
CSS size	1.5 KB	124.6 KB	124.6 KB	247.7 KB	247.7 KB

Додаток В

(обов’язковий)

Налаштування influxdb.conf для бази InfluxDB, з додаванням tag key “host”

```
[[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"
  ]
```

Додаток Г

(довідниковий)

Логи перевірки навантажувального тесту в Kubernetes Cluster

```
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ kubectl apply -f namespace.yaml
namespace/loadtest created
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ kubectl apply -f loadtest-job.yaml
job.batch/loadtest-run created
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ kubectl get ns
NAME              STATUS   AGE
default           Active   14m
kube-node-lease   Active   14m
kube-public       Active   14m
kube-system       Active   14m
loadtest          Active   20s
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ kubectl get nodes
NAME                                           STATUS   ROLES                  AGE   VERSION
ip-172-20-37-229.eu-central-1.compute.internal   Ready    node                   12m   v1.20.6
ip-172-20-49-68.eu-central-1.compute.internal    Ready    control-plane,master   14m   v1.20.6
ip-172-20-52-173.eu-central-1.compute.internal   Ready    node                   13m   v1.20.6
ip-172-20-56-20.eu-central-1.compute.internal    Ready    node                   11m   v1.20.6
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ kubectl get pods -n loadtest
NAME                 READY   STATUS              RESTARTS   AGE
loadtest-run-nlfgn   0/2     ContainerCreating   0          32s
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ kubectl get jobs -n loadtest
NAME           COMPLETIONS   DURATION   AGE
loadtest-run   0/1           47s        47s
```

```
ubuntu@ip-172-31-32-228:~/naukma-diploma-k8s$ k logs loadtest-run-nlfgn -n loadtest -c logstash
Using bundled JDK: /usr/share/logstash/jdk
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will likely be removed in a f
uture release.
Sending Logstash logs to /usr/share/logstash/logs which is now configured via log4j2.properties
[2021-05-16T10:12:13,299][INFO ][logstash.runner          ] Log4j configuration path used is: /usr/share/logstash/config/lo
g4j2.properties
[2021-05-16T10:12:13,305][INFO ][logstash.runner          ] Starting Logstash {"logstash.version"=>"7.12.0", "jruby.version
"=>"jruby 9.2.13.0 (2.5.7) 2020-08-03 9a89c94bcc OpenJDK 64-Bit Server VM 11.0.10+9 on 11.0.10+9 +indy +jit [linux-x86_64]"
}
[2021-05-16T10:12:13,320][INFO ][logstash.setting.writabledirectory] Creating directory {:setting=>"path.queue", :path=>"/u
sr/share/logstash/data/queue"}
[2021-05-16T10:12:13,384][INFO ][logstash.setting.writabledirectory] Creating directory {:setting=>"path.dead_letter_queue"
, :path=>"/usr/share/logstash/data/dead_letter_queue"}
[2021-05-16T10:12:13,992][INFO ][logstash.agent           ] No persistent UUID file found. Generating new UUID {:uuid=>"e3b
d6e38-6887-42dd-ae5a-0f1a1b443282", :path=>"/usr/share/logstash/data/uuid"}
[2021-05-16T10:12:15,383][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:port=>9600}
[2021-05-16T10:12:17,099][INFO ][org.reflections.Reflections] Reflections took 80 ms to scan 1 urls, producing 23 keys and
47 values
[2021-05-16T10:12:18,086][INFO ][logstash.javapipeline    ][main] Starting pipeline {:pipeline_id=>"main", "pipeline.worker
s"=>2, "pipeline.batch.size"=>125, "pipeline.batch.delay"=>50, "pipeline.max_inflight"=>250, "pipeline.sources"=>["/usr/sha
re/logstash/pipeline/gatling.conf"], :thread=>"#<Thread:0x1f2df797 run>"}
[2021-05-16T10:12:19,802][INFO ][logstash.javapipeline    ][main] Pipeline Java execution initialization time {"seconds"=>1
.71}
[2021-05-16T10:12:19,984][INFO ][logstash.inputs.graphite ][main] Automatically switching from plain to line codec {:plugin
=>"graphite"}
[2021-05-16T10:12:20,123][INFO ][logstash.javapipeline    ][main] Pipeline started {"pipeline.id"=>"main"}
[2021-05-16T10:12:20,188][INFO ][logstash.inputs.graphite ][main][31fb8d8834af20596bcd74ee7d25c105c475bd8a997b3f8806e3c2144
1a37408] Starting tcp input listener {:address=>"0.0.0.0:2003", :ssl_enable=>false}
[2021-05-16T10:12:20,306][INFO ][logstash.agent           ] Pipelines running {:count=>1, :running_pipelines=>[:main], :non
_running_pipelines=>[]}
```

Продовження Додатку Г