УДК 004.052
DOI: 10.18523/2617-3808.2020.3.42-49

Глибовець А. М., Карпович А. В., Ковш М. В.

РОЗПОДІЛЕНА СИСТЕМА НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ У БЕЗПЕРЕРВНІЙ ІНТЕГРАЦІЇ

У роботі вивчено можливість побудови розподіленої системи для навантажувального тестування із використанням інструментів, що перебувають у відкритому доступі, а саме Gatling, InfluxDB, Grafana, Logstash, Docker та Jenkins. Описано підхід для подолання обмежень інструменту з автоматизації тестування продуктивності Gatling, а саме обмеження, що не дає змоги побудувати власну систему розподіленого навантаження «з коробки». Це рішення інтегровано у безперервне постачання коду на основі сервісу Jenkins і випробовувано із централізованим звітуванням результатів у реальному часі. Цю систему було розроблено на противагу представленим на ринку комерційним рішенням, що надає їй більшої гнучкості.

Ключові слова: продуктивність системи, тестування продуктивності, тестування навантаження, безперервне постачання коду, відображення результатів у реальному часі, розподілена система навантаження, великомасштабні системи.

Вступ

Продуктивність системи – це окрема інженерна наука, яка охоплює підходи та практики для побудови й оптимізації систем на усіх рівнях – від фізичних приладів, інтернет-мережі до написання високоефективних алгоритмів. Водночас будь-яку систему, яку проектують і будують, має бути перевірено та випробовувано, бажано в реальних умовах, зокрема на продуктивність. І тут приходить на допомогу такий вид тестування, як тестування продуктивності, чи, як його часто називають, навантажувальне тестування.

Цей вид тестування є важливим не лише з погляду оптимізації ресурсів, потенційно він може вберегти комерційні (і не лише) онлайн-сайти від фінансових і репутаційних втрат. Наприклад, досить відомою є історія з падінням державного сайту США для медичного страхування www.HealthCare.gov. Система припинила працювати уже через 2 години після запуску, адже не була готова до навантаження у 250 тисяч користувачів [16]. І таких історій уже достатньо у світовій практиці.

Сама система з навантажувального тестування теж має відповідати конкретним критеріям і вимогам, реалізація яких дасть змогу провести повноцінне та ефективне тестування. Зокрема, для перевірки великих (у тому числі розподілених) інформаційних систем необхідно будувати (чи скористатися готовим комерційним рішенням) розподілену систему навантаження. Окрім того, може виникнути необхідність створювати це навантаження одночасно з різних частин світу. Своєю чергою, велике тестування продукує великі об'єми результуючих даних, які зручно збирати та відслідковувати централізовано у реальному часі, що дає змогу для швидкої реакції.

Саме побудову розподіленої системи з навантажувального тестування у безперервному постачанні коду із можливістю відображення результатів тестів у реальному часі буде висвітлено у цій роботі. Для її реалізації використано загальнодоступні інструменти, що робить такий підхід ефективним та прозорим із технічного погляду, а також дає змогу зменшити затрати на тестування продуктивності.

1. Теоретичні засади розподіленого тестування

1.1. Визначення тестування продуктивності

Скотт Барбер у своїй праці «Навантажувальне тестування веб-застосунків для початківців для новачків» дає досить просте, але влучне визначення: «Навантажувальне тестування – це те, як ви визначаєте, скільки трафіку ваш вебсайт або вебзастосунок може витримати перед тим, як упаде чи змусить користувача писати негативні відгуки на просторах інтернету. Іншими словами, це не більше ніж випробування вашого вебсайту в умовах реального використання і знаходження помилок перед тим, як їх знайдуть кінцеві споживачі» [1].

Водночас такий вид тестування здійснюється не лише для вебресурсів, де у фокусі кінцевий споживач, а і для класичних клієнт–сервер, розподілених, вбудованих та інших типів систем.

Стандарт ISO25010 [ISO25000] визначає продуктивність (ефективність) системи за допомогою таких характеристик:

– *часова складова* – ступінь відповідності вимогам за часом відгуку, часом опрацювання та пропускної здатності продукту чи системи упродовж свого функціонування;
– *використання ресурсів* – ступінь відповідності вимогам використання різних типів ресурсів під час функціонування системи;
– *ємність* – ступінь, за якої максимальні межі використання системи відповідають вимогам [12].

Базуючись на цьому визначенні, можна сказати, що тестування продуктивності передбачає оцінювання за двома напрямами:

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

1.2. Порівняння комерційних систем із тестування продуктивності

За визначенням компанії SmartBear, розподілене навантажувальне тестування є не чим іншим, як тестом, проведеним із декількох машин (комп'ютерів, серверів) одночасно, що дає змогу симулювати велику кількість віртуальних одночасних користувачів та генерує великі об'єми трафіку [10].

Найчастіше архітектура системи розподіленого навантаження має такі компоненти:

– постачальник хмарних технологій (AWS, MS Azure, GCP);
– машина-майстер для оркестрації та управління;
– машини-агенти для виконання самого навантаження;
– інструмент для розроблення та симуляції навантаження;
– інструменти для збирання та відображення результатів;
– за потреби – системи управління та/або безперервного постачання коду;
– за потреби – база для збереження тестових даних.

Для реалізації подібної системи на ринку існують як комерційні, так і готові загальнодоступні інструменти. Їх порівняння наведено у табл. 1.

Таблиця 1

Порівняльна таблиця розподілених систем генерації навантаження

Назва	Вартість на травень 2020 р.	Рівень адаптації під проект	Звітування	Інтеграція з CI/CD
Blazemeter	від 6 тис. у. о. / рік (5 тис. віртуальних користувачів)	Високий, дає можливість використання різних інструментів	Так, свій формат. У реальному часі	Так
LoadUI	від 11 454 у. о. / рік (1 тис. віртуальних користувачів)	Середній	Так, свій формат. У реальному часі	Так
LoadRunner	від 7 тис. – 10 тис. у. о. / рік (5 тис. віртуальних користувачів)	Високий, дає можливість використання різних інструментів	Так, свій формат. У реальному часі	Так
Gatling Frontline	15 тис. – 27 тис. у.о. / рік (залежно від частоти використання)	Високий	Так	Так
AWS template [2]	Безкоштовно (оплата за використання віртуальних машин)	Нижче ніж середній, адже обмежений своїми технологіями	Так, але обмежене можливостями CloudWatch	Так

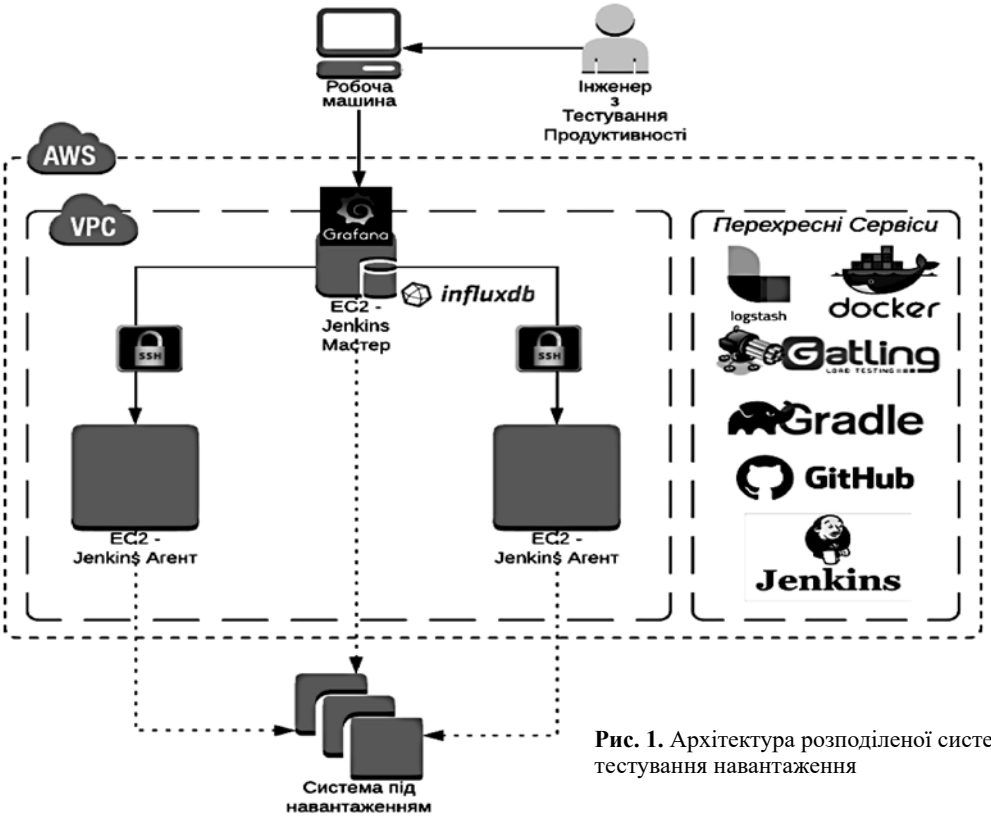

Рис. 1. Архітектура розподіленої системи тестування навантаження

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

2. Реалізація власної системи розподіленого навантаження

2.1. Архітектура розподіленої системи тестування

Архітектуру взаємодії компонентів системи розподіленого тестування навантаження та використані сервіси та інструменти для її побудови зображено на рис. 1.

Усі інструменти, які були використані для реалізації системи тестування, є загальнодоступними, а отже безкоштовними (окрім Amazon AWS EC2 віртуальних серверів). Окремо зупинимось на такому інструменті, як Logstash – інструмент з орбіти ELK для оброблення, необхідної видозміни та подальшого відправлення даних [9]. Саме завдяки Logstash стало можливим обійти обмеження інструменту Gatling для тестування з розподілених машин, що докладніше буде розглянуто далі.

2.2. Побудова системи відображення результатів у реальному часі

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

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

Водночас Gatling надсилає дані з інтервалом у 1 секунду, що є обмеженням для системи розподіленого навантажувального тестування. Щоб зрозуміти, чому, потрібно спочатку розглянути структуру даних InfluxDB на гіпотетичному прикладі збереження температурних показників локації у таблицю (measurement) під назвою weather (рис. 2):

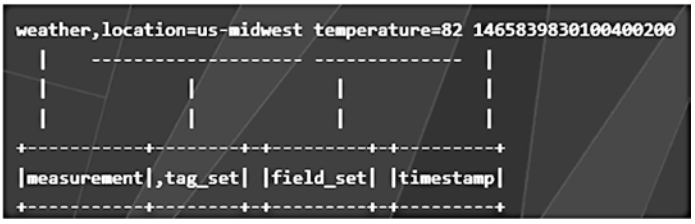

Рис. 2. Структура даних InfluxDB

– measurement – аналог таблиці у SQL;
– tag(s) – аналог проіндексовані значень у SQL;
– field(s) – аналог непроіндексованих значень у SQL;
– timestamp – значення часу в юнікс форматі. Основа основ для InfluxDB. Саме це значення є ключовим індексом для кожного запису.

Якщо для обох чи більше записів поля timestamp і tag(s) збігаються, незважаючи на значення field(s), то InfluxDB обробить ці записи як ідентичні та збереже лише останні значення fields. Своєю чергою, Gatling надсилає дані таким чином, що час має розмірність лише у секундах, а усі інші значення (назва сторінки, сценарію) є тагами (tags), окрім самого значення заміру (наприклад, час відгуку), який є полем (field). У результаті, виникає ситуація, коли ми втрачаємо частину даних із розподілених серверів навантаження, оскільки вони перезаписуються у базі.

Gatling не дає змоги додавати нові поля (tags, fields) або змінювати розмірність часу на мілісекунди, що могло б допомогти у розв'язанні цього обмеження. Саме тому було вирішено знайти стороннє рішення для розв'язання описаного обмеження – Logstash. Він зайняв позицію на шляху руху даних між Gatling та InfluxDB, додаючи ім'я машини (у нашому випадку ім'я Docker контейнера) як додатковий таг. Logstash запускається на кожній машині та надсилає дані в централізовану базу InfluxDB (рис. 3).

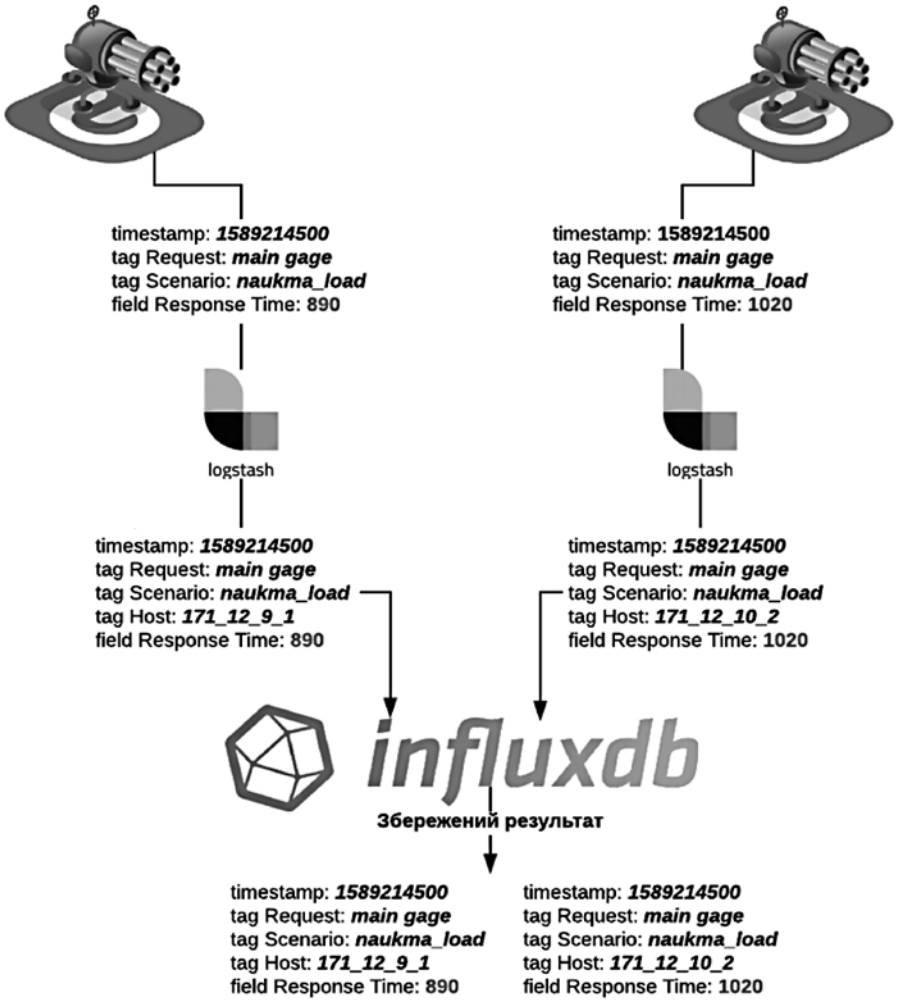

Рис. 3. Схема збереження результатів Gatling у InfluxDB за посередництва Logstash

Нижче наведено конфігурацію Logstash, яка додає ім'я машини на шляху руху результатів під час проведення тестування:

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

filter {

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

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

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

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

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

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

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

2.3. Побудова тестових сценаріїв на основі інструменту Gatling

Gatling використовує функціональну мову програмування Scala [4]. Це надає можливість контролювати версії тестів (приміром, через GitHub). Наприклад, основний конкурент JMeter не має такої можливості.

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

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

Кроки сценарію мають такий вигляд:

Рис. 4. Сценарій виконання тесту для http://fin.ukma.edu.ua

Для написання такого тесту було використано три модулі:

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

Сам тест було виконано на основі Page Object Pattern, який зазвичай використовують для функціонального тестування.

Розглянемо детальніше кожен із модулів.

baseConfig/BaseSimulation.scala

```
package baseConfig

import io.gatling.http.protocol.HttpPro-
tocolBuilder
import io.gatling.core.Predef._
import io.gatling.http.Predef._

class BaseSimulation extends Simulation {

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

.acceptHeader("text/html,application/
xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/
signed-exchange;v=b3;q=0.9")
     .acceptEncodingHeader("gzip, deflate, br")
     .acceptLanguageHeader("en,en-US;
     q=0.9,ru-RU; q=0.8,ru;q=0.7,uk;q=0.6")
     .userAgentHeader("Mozilla/5.0
     (Macintosh; Intel Mac OS X 10_15_4)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/81.0.4044.129 Safari/537.36")
     .silentResources
}
```

requestObjects/FinRequests.scala

```
package requestObjects

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

object FinRequests {

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

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

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

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

FinSimulation.scala

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

import scala.concurrent.duration._

class FinSimulation extends BaseSimulation{

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

  val finWebLoadScenario: ScenarioBuilder =
  scenario("FinUKMA Web Load")
    .during(120 seconds) {
      exec(FinRequests.finHome())
        .pause(1,2)
        .repeat(2){
          exec(FinRequests.finNews())
            .pause(3,5)
        }
        .repeat(2) {
          exec(FinRequests.finDepartment())
            .pause(3,5)
        }
        .exec(FinRequests.finDecanat())
        .pause(1,2)
    }

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

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

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

– на рівні кожного запиту – засобом .check(*status*.is(200));
– на рівні усього тесту зі встановленням граничного значення успішно здійснених запитів протягом усього тестування – засобом .assertions(global.successfulRequests.percent.is(100)).

2.4. Випробовування системи

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

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

Самотестування навантаження відбувалось для таких сайтів:

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

Було проведено більше ніж 20 різного типу тестів на продуктивність із використанням системи безперервного постачання коду Jenkins. Саме тестування було здійснено в нічний час для уникнення похибок від користування ресурсами реальними користувачами.

Отже, розглянемо результати навантажувального тестування докладніше.

– Було здійснено навантаження у 10 одночасних користувачів протягом трьох хвилин для кожного сайту.
– За результатами порівняння можна зробити висновок, що на порівняно малих навантаженнях найкращі показники продуктивності має сайт https://vstup.ukma.edu.ua (див. рис. 5 і табл. 2).

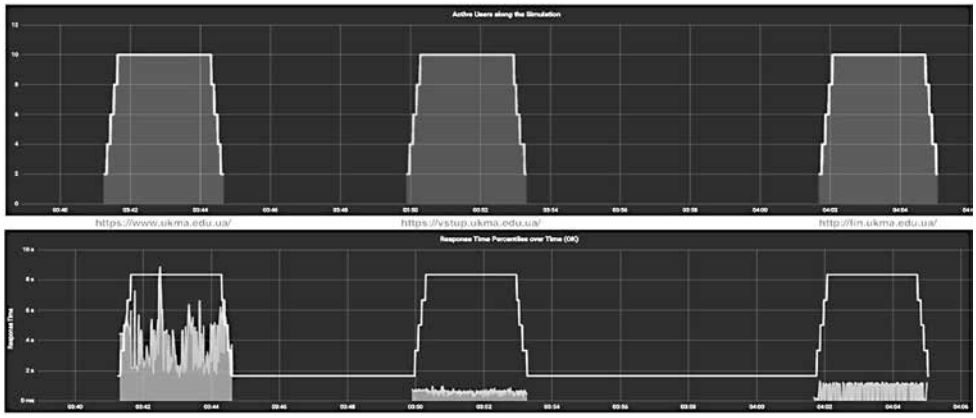

Рис. 5. Порівняння часу відгуку для трьох сайтів НаУКМА

Таблиця 2

Порівняння показників продуктивності

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

Якщо порівнювати http://fin.ukma.edu.ua та www.ukma.edu.ua під час великого навантаження, то перший сайт має значно кращу швидкість оброблення запитів, однак припиняє працювати за більше ніж 50 одночасних користувачів (3–4 запити в секунду). Натомість другий продовжує працювати у разі збільшення навантаження, хоча час відгуку при цьому може становити більше ніж 3 хвилини.

Висновки

У роботі було проаналізовано широкий спектр інструментів і підходів для побудови розподіленої системи навантажувального тестування. Згідно з поставленими у вступі цілями, ця система мала б відповідати трьом основним вимогам:

1) можливість інтегрування з системами безперервного постачання;
2) розподілене навантажувальне тестування (із декількох машин);
3) візуалізація результатів централізовано та в реальному часі.

У результаті проведених досліджень та пошуку рішень таку систему тестування продуктивності було успішно побудовано та випробувано на декількох сайтах справжніх сайтах. Було здійснено близько 20 різного типу тестів із навантаження.

Потрібно зазначити, що окрім основних цілей, було досягнуто також виконання декількох інших, не менш важливих завдань.

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

Додатково, систему можна швидко розгорнути у будь-якому операційному середовищі (Linux, Windows, Mac OS), адже для її побудови та запуску було використано сервіс віртуалізації Docker.

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

Список літератури

1. Barber Scott. Web Load Testing For Dummies / Scott Barber, Colin Mason. – John Wiley & Sons, Inc., 2011. – P. 4–5.
2. Distributed load testing on AWS service [Electronic resource]. – Mode of access: https://github.com/awslabs/distributed-load-testing-on-aws.
3. Documentation of Docker service [Electronic resource]. – Mode of access: https://www.docker.com.
4. Documentation of Gatling service [Electronic resource]. – Mode of access: https://gatling.io.
5. Documentation of Gradle service service [Electronic resource]. – Mode of access: https://gradle.org.
6. Documentation of Grafana service service [Electronic resource]. – Mode of access: https://grafana.com.
7. Documentation of InfluxDB service service [Electronic resource]. – Mode of access: https://www.influxdata.com.
8. Documentation of Jenkins service service [Electronic resource]. – Mode of access: https://www.jenkins.io.
9. Documentation of Logstash service service [Electronic resource]. – Mode of access: https://www.elastic.co/logstash.
10. Documentation of SmartBear service. Distributed Load Testing service [Electronic resource]. – Mode of access: https://support.smartbear.com/readyapi/docs/loadui/distributed/index.html.
11. Graham Bath. Foundation Level Specialist Syllabus Performance Testing / Bath Graham, Rex Black. – ISTQB, 2018. – P. 10–15.
12. ISO 25000 STANDARDS. ISO 25010. Performance efficiency service [Electronic resource]. – Mode of access: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/59-performance-efficiency.
13. Load Testing with JMeter: Test Results Visualization Using Kibana Dashboards Pipeline [Electronic resource]. – Mode of access: https://blogs.sap.com/2016/04/06/load-testing-with-jmeter-test-results-visualization-using-kibana-dashboards.
14. Meier J. D. Performance Testing Guidance for Web Applications: patterns and practices / J. D. Meier. – Microsoft Corporation, 2007. – P. 15–34.
15. Molyuneaux Ian. The Art of Application Performance Testing / Ian Molyuneaux. – O'Reilly, 2015. – P. 5–10.
16. The Failed Launch Of www.HealthCare.gov [Electronic resource]. – Mode of access: https://digital.hbs.edu/platform-rctom/submission/the-failed-launch-of-www-healthcare-gov.

References

Awslabs. (2020). GitHub – awslabs/distributed-load-testing-on-aws. *Distributed load testing on AWS.* Retrieved from https://github.com/awslabs/distributed-load-testing-on-aws.

Barber, Scott, & Mason, Colin. (2011). *Web Load Testing For Dummies.* John Wiley & Sons, Inc.

CD.Foundation. (2020). Jenkins. *Documentation of Jenkins service.* Retrieved from https://www.jenkins.io.

Digital Initiative at Harvard Business School. (2016). The Failed Launch Of www.HealthCare.gov. *Technology and Operations Management.* Retrieved from https://digital.hbs.edu/platform-rctom/submission/the-failed-launch-of-www-healthcare-gov.

Docker Inc. (2020). Empowering App Development for Developers. Docker. *Documentation of Docker service.* Retrieved from https://www.docker.com.

Elasticsearch, B. V. (2020). Logstash: Collect, Parse, Transform Logs | Elastic. *Documentation of Logstash service.* Retrieved from https://www.elastic.co/logstash.

Gatling Corp. (2020). Gatling Open-Source Load Testing – For DevOps and CI/CD. *Documentation of Gatling service.* Retrieved from https://gatling.io.

Gradle Inc. (2020). Gradle Build Tool. *Documentation of Gradle service.* Retrieved from https://gradle.org.

Grafana Labs. (2020). Grafana: The open observability platform | Grafana Labs *Documentation of Grafana service.* Retrieved from https://grafana.com.

Graham, Bath, & Black, Rex. (2018). *Foundation Level Specialist Syllabus Performance Testing*. ISTQB, 10–15.

influxData Inc. (2020). InFluxDB: Purpose – Buit Open Source Time Series DataBase. *Documentation of InfluxDB service.* Retrieved from https://www.influxdata.com.

ISO 25000. (2019). Performance efficiency. *ISO 25000 STANDARDS. ISO 25010. Performance efficiency.* Retrieved from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/59-performance-efficiency.

Meier, J. D. (2007). *Performance Testing Guidance for Web Applications: patterns and practices*. Microsoft Corporation.

Molyuneaux, Ian. (2015). *The Art of Application Performance Testing.* O'Reilly, 5–10.

SAP. (2017). Load Testing with JMeter. *Test Results Visualization Using Kibana Dashboards.* Retrieved from https://blogs.sap.com/2016/04/06/load-testing-with-jmeter-test-results-visualization-using-kibana-dashboards.

SmartBear Software. (2020). Distributed Load Testing**.** ReadyAPI Documentation. *Documentation of SmartBear service. Distributed Load Testing.* Retrieved from https://support.smartbear.com/readyapi/docs/loadui/distributed/index.html.

A. Hlybovets, A. Karpovych, M. Kovsh

DISTRIBUTED LOAD TESTING SYSTEM IN CONTINUOUS INTEGRATION

The digital system performance is a separate engineering science that includes approaches and practices for building and optimizing systems at all levels – from physical devices, network, to writing highly efficient algorithms. At the same time, any digital solution that is designed and built should be tested, preferably under real conditions, including performance testing or as it is often called load testing.

Performance testing is designed to simulate the real usage conditions of the system and load on it in order to find out its potential bottlenecks.

However, to implement this type of testing without special software tools is almost impossible. The performance testing system itself should also meet certain criteria and requirements, the implementation of which will make possible an effective testing. Thus, to test large (including distributed) information solutions, it is necessary to build a distributed system for its load. In addition, it may be necessary to create such a load simultaneously, for example, from different continents.

Moreover, large-scale testing produces large amounts of resulting data that need to be collected and tracked in real, allowing to respond quickly to errors or other problems.

The relevance of this topic is that, despite the presence on the market of commercial tools for such testing, it is often necessary to build own (project specific) system to display either more required metrics, or the inability to use third-party systems or budget restrictions, or all together.

This paper explores the possibility of building such a distributed solution for performance testing using open-source tools such as Gatling, InfluxDB, Grafana, Logstash, Docker and Jenkins. Gatling is the main tool for load testing, but its open-source version does not allow to implement scaled testing "out of the box". The main limitation is that even if you run several test processes from different servers at the same time, the report will be generated separately on each one and will not show the overall picture.

This problem has been solved by using such a tool as Logstash which is described in this paper. Additionally, this solution has been integrated into the continuous integration based on Jenkins service and successfully tested with centralized real-time reporting using Grafana service.

Keywords: system performance, performance testing, load testing, continuous integration, real-time reporting, test results, distributed load system.

Матеріал надійшов 05.06.2020