Київський національний університет імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики
Кафедра дослідження операцій

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
за спеціальністю 113 «Прикладна математика»
на тему:

Математичний аналіз продуктивності монолітного веб додатку

студентки 4 курсу
Мельничук Анни Леонідівни

Науковий кервіник:
професор, доктор фізико-математичних наук
Маринич О. В.

Робота заслухана на засіданні кафедри дослідження операцій та рекомендована до захисту в ЕК, протокол № 9 від 23 травня 2023 р.

Завідувач кафедри ДО проф. Іксанов О. М.

Київ 2023

Реферат

Обсяг роботи 61 сторінка, 8 ілюстрації, приклад даних у таблицях, 24 джерела посилання, код мовою R.

МОНОЛІТНИЙ ВЕБ-ДОДАТОК, МІКРОСЕРВІСНА АРХІТЕКТУРА, ПРОДУКТИВНІСТЬ, МЕТРИКИ, ОПТИМІЗАЦІЯ, МАСШТАБОВАНІСТЬ, ЕФЕКТИВНСІТЬ, ТЕСТУВАННЯ НАВАНТАЖЕННЯ, ЧАС ВІДГУКУ, ПРОПУСКНА ЗДАТНІСЬ, CPU, ОБ'ЄМ ПАМ'ЯТІ, ЧАСТОТА ПОМИЛОК, ДОСУПНІСТЬ, МАСШТАБОВАНІСТЬ, ВИКОРИСТАННЯ РЕСУРСІВ, НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ, ПРОФІЛЬ НАВАНТАЖЕННЯ.

Об'єктом роботи є показники системи моніторингу машини на якій працює додаток.

Методи розроблення: методи згладжування часових рядів.

Інструменти розроблення: серидовище розробки RStudio, мова програмування R, InfluxBD, Telegraf.

Результат роботи: проілюстровано роботу згладжування методами Хольта-Вінтерса та ARIMA на показниках CPU, проведений аналіз показників до та після згладження.

По-перше, розширення дослідження на ширший спектр веб-додатків, щоб підтвердити можливість узагальнення наших висновків. По-друге, вивчення альтернативних методів прогнозування, таких як ARIMA та нейронні мережі, для порівняння їх ефективності в оптимізації продуктивності веб-додатків. По-третє, дослідження застосовності наших результатів до мікросервісних архітектур, враховуючи зростаючу популярність цього підходу в сучасній веб-розробці. По-четверте, адаптивного алгоритму, який поєднує сильні сторони різних методів прогнозування, забезпечуючи більш надійне рішення для оптимізації продуктивності.

ЗМІСТ

ВСТУП

Стрімкий розвиток інформаційних технологій та зростаюче використання Інтернету суттєво вплинули на те, як ми проєктуємо, розробляємо та керуємо веб-додатками. У цифрову епоху продуктивність веб-додатків стала вирішальним фактором у забезпеченні безперебійного користувацького досвіду, задоволеності клієнтів і підтримці конкурентної переваги. Як наслідок, компанії та розробники все більше зацікавлені в оптимізації продуктивності веб-додатків, використовуючи різні архітектурні рішення, програмні фреймворки та обчислювальні методи.

Однією з найбільш обговорюваних тем у цій галузі є порівняння монолітних і мікросервісних архітектур та їх вплив на продуктивність. Монолітні архітектури, де додаток розробляється як єдина, уніфікована кодова база, були традиційним підходом до створення веб-додатків. На противагу цьому, мікросервісні архітектури поділяють додатки на менші, незалежні сервіси, які взаємодіють через API (Application Programming Interface). Обидві архітектури мають свої унікальні переваги та недоліки, що робить вибір між ними важливим питанням як для розробників, так і для бізнесу.

Метою даного дипломного проєкту є проведення детального статистичного аналізу продуктивності монолітного веб-додатку та розробка стратегії вирівнювання його завантаження процесора за допомогою методів Холта-Вінтерса та ARIMA. Ці методи, добре відомі в аналізі часових рядів, будуть використані для прогнозування та управління використанням ресурсів. Порівняння результатів до і після застосування запропонованих методів дасть цінну інформацію про продуктивність монолітних веб-додатків і послужить орієнтиром для оптимізації розподілу ресурсів.

Важливість теми підкреслюється зростаючим попитом на високопродуктивні та надійні веб-сервіси. Монолітні веб-додатки, хоча за певних обставин їх простіше розробляти і підтримувати, можуть зіткнутися з проблемами масштабованості і супроводу в міру зростання їх складності. Вивчаючи продуктивність цих додатків та використовуючи методи

математичної оптимізації, ми можемо розширити їх функціональність, покращити користувацький досвід та підтримати бізнес у наданні більш якісних послуг своїм клієнтам.

Дипломний проект структурований наступним чином: спочатку ми розглянемо відповідну літературу про архітектури веб-додатків, показники продуктивності та порівняння продуктивності монолітної та мікросервісної архітектури. Цей огляд забезпечить основу для розуміння існуючих досліджень у цій галузі та методологій, що використовуються для оцінки та порівняння продуктивності веб-додатків.

Далі ми детально опишемо методологію нашого дослідження, яка включає математичне моделювання профілю навантаження веб-додатків, збір даних та базовий аналіз даних. Математичне моделювання сприятиме глибшому розумінню факторів, що впливають на продуктивність, тоді як аналіз даних дозволить нам виявити тенденції та закономірності в зібраних даних.

Дотримуючись методології, ми узагальнимо результати за допомогою аналізу метричних відхилень, поглиблюючи наше розуміння характеристик продуктивності монолітних веб-додатків. Потім буде проведено статистичний аналіз для інтерпретації цих результатів, який виявить потенційні переваги застосування методів Холта-Вінтерса та ARIMA до навантаження процесора.

Нарешті, ми представимо наші висновки і обговоримо наслідки наших результатів. Цей дипломний проект зробить внесок у постійний дискурс про продуктивність веб-додатків і послужить ресурсом для розробників і компаній, які прагнуть оптимізувати свої монолітні веб-додатки.

1 Огляд джерел

1.1 Архітектура веб – додатків

Архітектура веб-додатків - це дизайн і структура програмних додатків, доступ до яких здійснюється через Інтернет за допомогою веб-браузера або мобільного додатку. За визначенням Вернора Вінджа, відомого письменника-фантаста і комп'ютерного вченого, "веб-додатки - це принципово розподілені додатки, часто побудовані на основі складної інфраструктури проміжного програмного забезпечення і широко використовують мережеві ресурси і сервіси" [1]. Це визначення підкреслює той факт, що веб-додатки не є автономними програмами, а скоріше залежать від мережі взаємопов'язаних компонентів, таких як сервери, бази даних, API та сторонні сервіси. Тому архітектура веб-додатку повинна враховувати розподілену природу системи і потенційні вузькі місця в продуктивності та ризики для безпеки, які виникають у зв'язку з цим.

На думку Френка Бушмана та ін., групи архітекторів і дослідників програмного забезпечення, "архітектура являє собою значущі проєктні рішення, які формують систему, де "значущість" вимірюється вартістю змін" [2]. Це визначення підкреслює важливість архітектури в розробці програмного забезпечення, оскільки вона визначає довгострокову гнучкість і ремонтопридатність системи. У контексті архітектури веб-додатків вибір архітектури впливає не тільки на функціональні та нефункціональні вимоги додатку, такі як взаємодія з користувачем, узгодженість даних і масштабованість, але й на сам процес розробки, наприклад, на структуру команди, стратегію розгортання і підхід до тестування.

Сфера архітектури веб-додатків еволюціонувала з плином часу, з'явилися різні архітектурні стилі та патерни у відповідь на мінливі технологічні та бізнес-потреби. Одними з найпоширеніших архітектур, що використовуються при розробці веб-додатків, є монолітна архітектура та архітектура мікросервісів. Монолітна архітектура - це традиційний підхід, коли всі компоненти програми розгортаються як єдине ціле, як правило, на одному сервері. Мікросервісна

архітектура, з іншого боку, декомпозує додаток на менші, слабко пов'язані сервіси, які взаємодіють один з одним через мережу. Обидві архітектури мають свої переваги і недоліки з точки зору продуктивності, масштабованості, підтримки і вартості, і вибір правильної архітектури залежить від конкретних вимог і обмежень проекту.

Монолітна архітектура, як було зазначено раніше - це патерн проектування програмного забезпечення, де всі компоненти програми тісно пов'язані між собою і розгортаються як єдине ціле на одному сервері. За словами Сема Ньюмана, експерта з архітектури мікросервісів, "монолітна архітектура - це архітектура, в якій ви будуєте все як одну велику річ, а не розбиваєте її на окремі компоненти" [3]. Це визначення підкреслює ключову характеристику монолітної архітектури, яка полягає у відсутності модульності та тісному зв'язку між компонентами програми.

До переваг монолітної архітектури можна віднести простоту розробки та розгортання, оскільки всі компоненти розробляються і тестуються разом і розгортаються як єдине ціле. Крім того, монолітна архітектура може бути більш економічно вигідною для малих і середніх проєктів, оскільки вона вимагає менше ресурсів і менше досвіду роботи з розподіленими системами. Однак монолітна архітектура також має деякі суттєві недоліки, такі як обмежена масштабованість, оскільки весь додаток повинен масштабуватися як єдине ціле, і низька відмовостійкість, оскільки збій в одному компоненті може призвести до виходу з ладу всієї системи.

За словами Мартіна Фаулера, архітектора і автора програмного забезпечення, "монолітна архітектура є хорошим вибором для малих і середніх додатків з більш простою архітектурою" [4]. Це твердження підкреслює, що придатність монолітної архітектури залежить від розміру та складності програми, а також від конкретних вимог та обмежень проекту. Наприклад, монолітна архітектура може краще підходити для додатків з низьким трафіком, невеликою кількістю залежностей і низькою складністю, де переваги

модульності і масштабованості можуть не переважити витрати на підтримку розподіленої системи.

Мікросервісна архітектура, як було зазначено раніше - це патерн проєктування програмних додатків, який передбачає розбиття їх на менші, незалежні сервіси, що взаємодіють між собою за допомогою API-інтерфейсів. Сем Ньюман, експерт з архітектури мікросервісів, визначає мікросервіси як "невеликі, сфокусовані компоненти, які добре виконують одну справу і взаємодіють через мережу з іншими сервісами" [3]. Це визначення підкреслює ключову особливість мікросервісної архітектури, яка полягає у високій модульності та низькому рівні зв'язку між різними компонентами додатку.

Мікросервісна архітектура має ряд переваг, таких як масштабованість, відмовостійкість та гнучкість. Кожен сервіс можна масштабувати незалежно, виходячи з його конкретних вимог, а збої в одному сервісі не впливають на всю систему. Крім того, мікросервісна архітектура дозволяє здійснювати гнучку розробку та розгортання, оскільки кожен сервіс може бути розроблений і протестований незалежно і розгорнутий без впливу на інші сервіси. Однак ця архітектура також має деякі суттєві недоліки, такі як підвищена складність, вищі витрати на розробку та обслуговування, а також потреба в додатковому досвіді та ресурсах для управління розподіленою системою.

На думку Мартіна Фаулера, архітектора та автора програмного забезпечення, мікросервісна архітектура є гарним вибором для великих, складних додатків з високим трафіком та частими змінами [4]. Це твердження підкреслює, що придатність мікросервісної архітектури залежить від різних факторів, таких як розмір і складність програми, а також конкретні вимоги та обмеження проекту. Наприклад, мікросервісна архітектура може бути більш придатною для додатків зі складною бізнес-логікою, високим трафіком і частими змінами, де переваги модульності та масштабованості переважують витрати на управління розподіленою системою.

1.2 Показники продуктивності

Метрики користувацького досвіду (UX) використовуються для оцінки якості взаємодії між користувачем і системою. Дві найпоширеніші метрики UX - це час відгуку та пропускна здатність.

Час відгуку - це час, протягом якого система реагує на запит або дію користувача. Це важливий показник UX, оскільки користувачі часто сприймають повільний час відгуку як низьку продуктивність системи. За даними Nielsen, "час відгуку в 0,1 секунди дає відчуття миттєвої реакції, 1 секунда зберігає потік думок користувача безперебійним, 10 секунд і більше - користувачі втрачають фокус на завданні" [5]. Таким чином, час відгуку є ключовим показником задоволеності користувачів, і він безпосередньо впливає на зручність використання системи.

Пропускна здатність - це показник кількості запитів користувачів, які система може обробити за певний період часу. Це ключовий показник для систем, які, як очікується, будуть обробляти велику кількість запитів користувачів одночасно. Джайн стверджує, що "в деяких випадках збільшення пропускної здатності може призвести до зменшення часу відгуку, оскільки це зменшує час, витрачений на очікування завершення запитів"[6]. Таким чином, пропускна здатність також є важливою метрикою для вимірювання ефективності та пропускної здатності системи.

Як час відгуку, так і пропускна здатність мають значний вплив на сприйняття користувачем якості системи. За словами Джайна, "час відгуку та пропускна здатність є двома найбільш поширеними та фундаментальними показниками продуктивності системи, які сприймаються її користувачами" [6]. Тому дуже важливо регулярно відстежувати ці показники, щоб переконатися, що система забезпечує позитивний користувацький досвід.

Функціональні метрики використовуються для вимірювання правильності та надійності системи. Дві найпоширеніші функціональні метрики - це частота помилок і доступність.

Частота помилок - це кількість помилок або збоїв, які виникають в системі протягом певного періоду. Це важливий показник, оскільки він безпосередньо впливає на сприйняття надійності системи користувачем. За словами Луніса та ін., "частота помилок може суттєво вплинути на успіх або невдачу системи, а сприйняття системи користувачами сильно залежить від частоти помилок"[7]. Тому дуже важливо контролювати та мінімізувати кількість помилок, щоб забезпечити надійність системи та підтримувати задоволеність користувачів.

Доступність - це міра здатності системи залишатися працездатною і доступною для користувачів протягом певного періоду. Це критично важливий показник для систем, які повинні надавати безперервні послуги. За словами Рамакрішнана та ін., "доступність є критично важливою вимогою для критично важливих додатків, і додаток, який недоступний навіть протягом короткого періоду, може призвести до значних втрат або пошкоджень"[8]. Тому вкрай важливо забезпечити високу доступність, щоб уникнути будь-яких потенційних збитків або втрат.

Як рівень помилок, так і доступність мають значний вплив на надійність системи та сприйняття користувачем її функціональності. За словами Менаске, "надійність системи є важливим аспектом загальної якості обслуговування, а сприйняття користувачами системи значною мірою залежить від її надійності та доступності" [9]. Тому дуже важливо регулярно відстежувати ці показники, щоб переконатися, що система функціонує правильно і надійно.

Показники масштабованості використовуються для вимірювання здатності системи справлятися зі зростаючими навантаженнями або вимогами.

Масштабованість - це здатність системи справлятися зі зростаючими навантаженнями або вимогами, зберігаючи при цьому свою продуктивність. За словами Менаске, "масштабованість - це здатність системи адаптуватися до змін у робочому навантаженні шляхом додавання або видалення ресурсів" [9]. Таким чином, масштабованість є критично важливою метрикою для систем, які, як очікується, будуть працювати зі зростаючими вимогами або навантаженнями.

Масштабованість можна виміряти за допомогою різних показників, таких як час відгуку, пропускна здатність та використання ресурсів. За словами Джайна, "найбільш часто використовуваними показниками масштабованості є час відгуку і пропускна здатність, які вимірюють, наскільки добре система може впоратися зі зростаючим навантаженням без погіршення своєї продуктивності" [6]. Тому дуже важливо стежити за цими показниками при оцінці масштабованості системи.

Масштабованість є важливою метрикою для забезпечення того, щоб система могла впоратися з підвищеними вимогами або навантаженнями в майбутньому. За словами Менаске, "масштабованість є важливим фактором загальної продуктивності системи, і необхідно переконатися, що система може впоратися зі зростаючими навантаженнями або вимогами без погіршення її продуктивності або функціональності"[9]. Тому дуже важливо регулярно контролювати та оцінювати масштабованість системи.

Показники використання ресурсів використовуються для вимірювання використання системою ресурсів, таких як процесор, пам'ять, дисковий простір і пропускна здатність мережі. Ці показники є дуже важливими, оскільки вони можуть допомогти виявити потенційні вузькі місця та оптимізувати використання ресурсів.

Використання ресурсів - це використання системою таких ресурсів, як процесор, пам'ять, дисковий простір і пропускна здатність мережі. За словами Джайна, "показники використання ресурсів важливі для виявлення потенційних вузьких місць та оптимізації використання ресурсів" [6]. Тому дуже важливо регулярно відстежувати ці показники, щоб переконатися, що система використовує ресурси ефективно.

Використання ресурсів можна виміряти за допомогою різних показників, таких як завантаження процесора, використання пам'яті, використання дискового простору та використання пропускної здатності мережі. За словами Менаске, "завантаження процесора і пам'яті є найбільш часто

використовуваними показниками використання ресурсів, оскільки вони безпосередньо впливають на продуктивність системи" [9]. Тому дуже важливо стежити за цими показниками при оцінці використання ресурсів системи.

Показники використання ресурсів є важливими для забезпечення ефективного використання системою ресурсів та уникнення потенційних вузьких місць. За словами Луніса та ін., "неефективне використання ресурсів може призвести до погіршення продуктивності та виходу системи з ладу, і тому необхідно регулярно контролювати та оптимізувати використання ресурсів"[7]. Тому дуже важливо відстежувати ці показники та оптимізувати використання ресурсів для забезпечення надійності та продуктивності системи.

1.3 Порівняння продуктивності монолітної та мікросервісної архітектури

"Монолітна архітектура - це традиційний підхід до побудови програмного забезпечення, при якому додаток будується як єдиний, самодостатній блок, з усіма його компонентами, тісно пов'язаними та взаємопов'язаними" [11]. Згідно з The Software Architecture Chronicles, "монолітна архітектура зазвичай складається з трьох основних частин: рівня представлення, рівня бізнес-логіки та рівня бази даних. Ці три шари тісно пов'язані та упаковані разом, а це означає, що будь-яка зміна в одному шарі вимагає перерозгортання всього додатку" [11].

Крім того, монолітна архітектура зазвичай спирається на один технологічний стек, і розробники повинні використовувати одну мову програмування та фреймворк для всього додатку [12]. Як наслідок, процес розробки може бути повільнішим, а масштабування додатку - складнішим. Згідно з дослідженням Gartner, "монолітні додатки зазвичай є великими і складними, що ускладнює їх масштабування і підтримку" [13].

Таким чином, монолітна архітектура - це традиційний підхід до побудови програмного забезпечення, що складається з єдиного, тісно пов'язаного блоку, який може бути складним для масштабування та підтримки.

Мікросервісна архітектура - це підхід до побудови розподілених систем, який структурує додаток як набір слабко пов'язаних сервісів. Кожен сервіс призначений для виконання певної функції і є самодостатнім, тобто його можна розробляти, розгортати і масштабувати незалежно від інших сервісів у додатку. Мікросервісна архітектура часто протиставляється монолітній архітектурі, яка характеризується єдиною кодовою базою, тісно пов'язаними компонентами та централізованою моделлю розгортання.

Згідно з Ньюманом [3], мікросервісна архітектура дозволяє розробникам створювати один додаток як набір невеликих ізольованих сервісів, які взаємодіють між собою за допомогою HTTP API, що надає доступ до кожного окремого сервісу. Ці сервіси побудовані навколо конкретних бізнес-можливостей і можуть бути незалежно розгорнуті за допомогою повністю автоматизованого механізму розгортання. Мікросервісна архітектура дозволяє організаціям створювати високомодульні та масштабовані системи, що полегшує реагування на мінливі бізнес-вимоги та потреби клієнтів.

Характеристики мікросервісної архітектури, описані Ньюманом [3], включають:

1. Висока модульність і масштабованість: Мікросервіси розроблені так, щоб бути невеликими, цілеспрямованими і незалежно розгортатися. Це полегшує масштабування окремих послуг у відповідь на зміну попиту, не впливаючи на решту програми.
2. Стійкість до збоїв і несправностей: Мікросервіси розроблені таким чином, щоб бути відмовостійкими та стійкими до збоїв. Якщо один сервіс виходить з ладу, це не обов'язково означає, що постраждає вся програма. Інші сервіси можуть продовжувати нормально функціонувати, зменшуючи вплив збоїв на всю систему.
3. Незалежне розгортання сервісів: Кожен сервіс в мікросервісній архітектурі може бути розгорнутий незалежно від інших. Це дозволяє організаціям оновлювати або замінювати сервіси, не

впливаючи на решту програми, що полегшує обслуговування та розвиток системи з часом.

4. Поліглотське програмування та стек технологій: Мікросервісна архітектура дозволяє розробникам обирати найкращий інструмент для кожної роботи, тобто різні сервіси можуть бути написані різними мовами програмування або з використанням різних технологічних стеків. Це може допомогти організаціям використовувати наявний досвід і зменшити прив'язку до одного постачальника.

Децентралізоване управління та менеджмент: В архітектурі мікросервісів кожна послуга відповідає за власне управління та менеджмент. Це зменшує потребу в централізованій координації і може допомогти організаціям досягти більшої оперативності та гнучкості.

Загалом, мікросервісна архітектура надає організаціям більш гнучкий і масштабований підхід до побудови розподілених систем. Розбиваючи додаток на менші, більш сфокусовані сервіси, організації можуть швидше реагувати на мінливі бізнес-потреби і надавати кращий клієнтський досвід.

Порівнюючи продуктивність монолітної та мікросервісної архітектури, важливо враховувати безліч факторів, включаючи складність розгортання, масштабованість, відмовостійкість та використання ресурсів. Як зазначає Ньюман [3], "моноліт - це єдиний логічний виконуваний файл, тоді як мікросервісна архітектура - це набір невеликих сервісів, кожен з яких працює у своєму процесі і взаємодіє з легкими механізмами".

Однією з важливих переваг мікросервісної архітектури є можливість масштабування сервісів незалежно один від одного, що може допомогти зменшити використання ресурсів і підвищити загальну продуктивність. Як зазначає Ньюман [3], "масштабуючи окремі сервіси незалежно, ви можете використовувати ресурси більш ефективно і уникати надмірного резервування". На противагу цьому, монолітна архітектура може потребувати додаткових

ресурсів для масштабування всього додатку, навіть якщо тільки окремі сервіси потребують більше ресурсів.

Однак складність розгортання мікросервісної архітектури також може вплинути на продуктивність. У монолітній архітектурі весь додаток може бути розгорнутий як єдине ціле, що полегшує управління та обслуговування. На противагу цьому, мікросервісна архітектура вимагає розгортання декількох сервісів незалежно один від одного, що може збільшити ризик помилок і збоїв. Згідно з дослідженням Чжу та ін. [14], "мікросервіси вимагають розвиненої інфраструктури та середовища розгортання, включаючи реєстр сервісів, виявлення сервісів, балансування навантаження та механізм відмовостійкості, що збільшує накладні витрати на розгортання".

Ще одним фактором, який слід враховувати, є відмовостійкість. Архітектура мікросервісів розроблена таким чином, щоб бути відмовостійкою, коли кожен сервіс може продовжувати функціонувати, навіть якщо інші сервіси виходять з ладу. Це допомагає гарантувати, що додаток залишається доступним і реагує на запити користувачів. На противагу цьому, монолітна архітектура має єдину точку відмови, яка може вивести з ладу весь додаток.

Переваги продуктивності мікросервісної архітектури можуть варіюватися в залежності від різних сценаріїв. Дослідження, проведене Чжу та ін. [14], показало, що мікросервісна архітектура може забезпечити кращу продуктивність і масштабованість порівняно з монолітною архітектурою в сценаріях з високим рівнем одночасної роботи користувачів і потребою в швидкій розробці та розгортанні. Однак, переваги продуктивності мікросервісної архітектури можуть бути нівельовані підвищеною складністю та витратами на розгортання.

Таким чином, порівняння продуктивності монолітної та мікросервісної архітектури залежить від контексту. Хоча мікросервісна архітектура може забезпечити кращу масштабованість і відмовостійкість, вона також може бути складнішою у розгортанні та обслуговуванні. Тому організації повинні ретельно оцінити свої конкретні потреби і вимоги при виборі між цими двома архітектурами.

2 Методологія

2.1 Математичне моделювання профілю навантаження веб-додатку

У контексті веб-додатків профіль навантаження - це характеристика трафіку і використання ресурсів, які наповнюють додаток під різними навантаженнями. Навантажувальне тестування веб-додатків вимагає реалістичного робочого навантаження, яке відображає поведінку користувачів і моделі трафіку цільової системи [15]. Профіль навантаження включає поведінку користувачів, моделі трафіку, типи запитів та інші фактори, які впливають на продуктивність програми.

Поведінка користувачів є ключовим компонентом профілю навантаження, оскільки вона визначає, як користувачі взаємодіють з додатком і генерують трафік. Поведінка користувача - це складне і динамічне явище, яке залежить від різних факторів, таких як цілі користувача, його уподобання і контекст [16]. Тому профіль навантаження повинен відображати поведінку користувачів, яка є найбільш релевантною для додатку, що тестується. Наприклад, якщо додаток є сайтом електронної комерції, профіль навантаження повинен враховувати таку поведінку користувачів, як пошук продуктів, додавання товарів до кошика та оформлення замовлення.

Моделі трафіку також важливі, оскільки вони впливають на обсяг і розподіл запитів до додатку. Моделі трафіку можуть змінюватися залежно від часу доби, дня тижня та інших факторів, які впливають на поведінку користувачів і моделі доступу [17]. Профіль навантаження повинен враховувати моделі трафіку, які є найбільш релевантними для додатку, що тестується. Наприклад, якщо додаток є новинним сайтом, профіль навантаження повинен враховувати такі моделі трафіку, як більш високий трафік в ранкові та вечірні години пік.

Типи запитів, такі як HTTP GET або POST, впливають на ресурси, які споживає додаток, і затримки, які відчувають користувачі. Типи запитів можуть мати значний вплив на енергоспоживання і продуктивність програми, оскільки

різні типи запитів можуть вимагати різного обсягу обробки і зв'язку [18]. Тому профіль навантаження повинен враховувати типи запитів, які є найбільш релевантними для додатку, що тестується. Наприклад, якщо додаток є соціальною мережею, профіль навантаження повинен враховувати такі типи запитів, як публікація повідомлень, коментування повідомлень і завантаження медіафайлів.

Попередні дослідження створили кілька профілів навантаження, які можна використовувати як відправну точку для розробки профілю навантаження для нового додатка. Існуючі профілі навантаження можуть надати корисну інформацію про характеристики веб-додатків і типи робочих навантажень, з якими вони можуть зіткнутися [19]. Наприклад, бенчмарк SPECweb2009 надає стандартизований профіль навантаження, який можна налаштувати для конкретних додатків [20].

Таким чином, профіль навантаження є важливою частиною навантажувального тестування веб-додатків. Профіль навантаження повинен фіксувати поведінку користувачів, моделі трафіку, типи запитів та інші фактори, які впливають на продуктивність програми. Профіль навантаження повинен бути розроблений на основі характеристик програми, що тестується, і може бути заснований на існуючих профілях навантаження з попередніх досліджень.

Профіль навантаження складається з декількох компонентів, які визначають робоче навантаження, яке веб-додаток буде відчувати під час різних навантажень. Ці компоненти включають поведінку користувачів, моделі трафіку, типи запитів та інші фактори, що впливають на продуктивність програми.

Типи запитів є ще одним важливим компонентом профілю навантаження, оскільки вони визначають типи запитів, які отримуватиме додаток, і ресурси, які вони споживатимуть. Різні типи запитів можуть мати різні вимоги до ресурсів і характеристики затримок, а також впливати на загальну продуктивність програми [15]. Наприклад, типи запитів на сайті електронної комерції можуть включати пошук товарів, додавання товарів до кошика та оформлення

замовлення, тоді як типи запитів на сайті соціальних мереж можуть включати публікацію повідомлень, коментування постів та завантаження медіафайлів.

Інші фактори, які можуть впливати на продуктивність програми, включають апаратне та програмне середовище, мережеву інфраструктуру та характеристики користувачів. Ці фактори важко піддаються точному моделюванню, але можуть мати значний вплив на продуктивність програми під різними навантаженнями [21].

Таким чином, компоненти профілю навантаження включають поведінку користувачів, моделі трафіку, типи запитів та інші фактори, які впливають на продуктивність програми. Ці компоненти можуть широко варіюватися в залежності від характеру програми і популяції користувачів, і їх слід ретельно враховувати при розробці профілю навантаження для навантажувального тестування.

При розробці профілю навантаження для веб-додатку важливо враховувати всі наявні профілі навантаження, які можуть бути доступними. Існуючі профілі навантаження можуть надати цінну інформацію про очікувану поведінку користувачів і моделі трафіку, з якими додаток може зіткнутися під час різних навантажень.

Згідно з Алі та Аль-Халіфою, "існуючі профілі навантаження можуть бути використані як основа для розробки нових профілів навантаження, які більш репрезентативні для цільового додатку та популяції користувачів" [16]. Наприклад, якщо доступний профіль навантаження для подібного додатку електронної комерції, його можна використовувати як відправну точку для розробки профілю навантаження для цільового додатку. Існуючий профіль навантаження може бути скоригований на основі специфічних характеристик цільового додатка, таких як типи пропонованих продуктів або послуг, демографічні дані користувачів і географічне розташування користувачів.

На додаток до використання існуючих профілів навантаження як основи для нових профілів навантаження, важливо також враховувати будь-які обмеження або проблеми з існуючими профілями навантаження. Як зазначають

Agrawal і Agrawal, "існуючі профілі навантаження можуть неточно відображати поведінку користувачів у цільовому додатку, особливо якщо цільовий додаток значно відрізняється від додатків, що використовуються для створення існуючих профілів навантаження" [15]. Наприклад, профіль навантаження для додатку соціальних мереж може не відображати поведінку користувачів у додатку фінансових послуг.

Тому важливо ретельно оцінити будь-які існуючі профілі навантаження, перш ніж використовувати їх як основу для розробки нового профілю навантаження. Ця оцінка повинна включати аналіз характеристик додатку та користувачів, моделей трафіку, а також будь-яких обмежень або проблем з існуючим профілем навантаження.

Інструменти навантажувального тестування використовуються для імітації поведінки користувачів і генерування трафіку на веб-додатку, щоб оцінити його продуктивність під різними навантаженнями. Існує безліч інструментів і методів навантажувального тестування, кожен з яких має свої переваги та обмеження.

За словами Алі та Аль-Халіфи, "JMeter, Gatling, Siege та LoadRunner є одними з найпоширеніших інструментів для тестування навантаження" [16]. Ці інструменти надають цілий ряд можливостей і функцій для навантажувального тестування, включаючи можливість імітувати різну поведінку користувачів, генерувати трафік з різною швидкістю і відстежувати продуктивність системи під час тесту.

Інструменти навантажувального тестування зазвичай працюють, генеруючи велику кількість запитів до програми та вимірюючи час відгуку і пропускну здатність. Як пояснюють Agrawal і Agrawal, "інструмент навантажувального тестування генерує запити до веб-сервера таким чином, щоб імітувати поведінку реальних користувачів, а потім записує час відгуку та інші показники продуктивності для кожного запиту" [15].

Інструменти навантажувального тестування можуть бути налаштовані відповідно до конкретних потреб проєкту. Наприклад, налаштування інструменту можуть бути скориговані для імітації різних типів поведінки

користувачів, таких як перегляд, пошук і купівля. Інструмент також можна налаштувати на генерування трафіку з різною швидкістю, наприклад, у пікові та непікові години, щоб оцінити продуктивність програми під різними навантаженнями.

Важливо зазначити, що інструменти навантажувального тестування мають обмеження і не можуть повністю імітувати поведінку реальних користувачів. За словами Agrawal та Agrawal, "інструменти навантажувального тестування не можуть імітувати всі варіації та складнощі реальної поведінки користувачів, а також можуть неточно відображати поведінку користувачів на цільовому додатку" [15]. Тому важливо ретельно оцінювати результати навантажувального тестування і враховувати будь-які обмеження або невизначеності в процесі тестування.

Профіль навантаження для веб-застосунку повинен відображати реалістичне навантаження, яке може відчувати застосунок під час різних навантажень. Розробка профілю навантаження передбачає вибір або генерування поведінки користувачів, шаблонів трафіку і типів запитів, які точно відображають використання додатку.

Як пояснюють Хуанг та ін., "профіль навантаження розробляється на основі характеристик веб-додатку, включаючи поведінку користувачів, модель трафіку та модель робочого навантаження" [22]. Поведінка користувача включає різні дії, які користувачі можуть виконувати в додатку, такі як перегляд, пошук і покупки. Структура трафіку включає розподіл запитів користувачів у часі, наприклад, у пікові та непікові години. Моделі робочого навантаження включають кількість користувачів і швидкість, з якою генеруються запити.

Існують різні методи, які можна використовувати для створення профілю навантаження. Наприклад, профіль навантаження може базуватися на історичних даних, коли поведінка користувачів і моделі трафіку отримаються з журналів використання програми. Крім того, синтетичні моделі навантаження можуть бути створені за допомогою статистичних методів, таких як моделі Маркова або моделі черг.

При розробці профілю навантаження важливо враховувати будь-які припущення або спрощення, які робляться. Як зазначають Чжоу та ін., "модель робочого навантаження повинна бути максимально реалістичною, але важливо дотримуватися балансу між реалістичністю та простотою" [23]. Спрощення можуть бути необхідними, щоб зменшити складність профілю навантаження і полегшити його моделювання та аналіз.

Дизайн профілю навантаження повинен бути перевірений перед проведенням експерименту, щоб забезпечити його достовірність. Це може включати попередні випробування або моделювання для перевірки поведінки профілю навантаження. Будь-які обмеження або невизначеності в процесі валідації профілю навантаження повинні бути ретельно розглянуті і про них слід повідомити.

Після розробки профілю навантаження важливо перевірити його точність і надійність перед проведенням експерименту. Валідація допомагає виявити будь-які потенційні недоліки або слабкі місця в профілі навантаження і виправити їх до початку експерименту, забезпечуючи достовірність і надійність результатів.

Як пояснюють Лі та ін., "валідація - це процес оцінки якості та точності моделі робочого навантаження, щоб переконатися, що вона точно відображає поведінку користувачів програми" [24]. Валідація передбачає порівняння поведінки профілю навантаження з очікуваною поведінкою програми в реальних умовах.

Для валідації профілю навантаження можна використовувати кілька методів. Одна з них полягає у використанні синтетичних генераторів користувачів для імітації поведінки користувача і порівняння її з поведінкою в профілі навантаження. Інший метод полягає у використанні інструментів моніторингу реальних користувачів, які фіксують поведінку користувача і порівнюють її з профілем навантаження. Інструменти навантажувального тестування також можна використовувати для перевірки профілю навантаження, генеруючи трафік і порівнюючи його з очікуваними моделями трафіку.

Важливо враховувати будь-які обмеження або невизначеності в процесі перевірки профілю навантаження. Як зазначають Хуанг та ін., "валідація моделі робочого навантаження є складним завданням через складність веб-додатків і різноманітність поведінки користувачів" [22]. Валідація може бути обмежена доступністю реальних даних або точністю синтетичних моделей робочого навантаження. Важливо документувати будь-які обмеження або невизначеності в процесі валідації профілю навантаження, щоб забезпечити прозорість і відтворюваність експерименту.

2.2 Проведення дослідження

Під час запуску мікросервісного веб-додатку на сервері важливо відстежувати та аналізувати продуктивність сервера, щоб переконатися, що він працює на оптимальному рівні. Це передбачає збір різних метрик, які можуть допомогти виявити будь-які проблеми, що можуть вплинути на продуктивність сервера.

Телеграф - це агент збору даних з відкритим вихідним кодом, призначений для збору та обробки метрик з різних джерел. Він є частиною TICK Stack, популярного стеку програмного забезпечення з відкритим вихідним кодом, що використовується для збору, зберігання та візуалізації даних часових рядів.

Однією з ключових особливостей Telegraf є його здатність збирати метрики з широкого спектру джерел, включаючи системні метрики, метрики додатків та користувацькі метрики. Він може збирати метрики з різних джерел, включаючи операційні системи, бази даних, мережеві пристрої та інші додатки.

Telegraf надає понад 200 плагінів, які можна використовувати для збору метрик з різних джерел. Ці плагіни можна конфігурувати і налаштовувати відповідно до конкретних потреб користувача. Наприклад, плагін CPU можна використовувати для збору метрик використання процесора, а плагін Memory - для збору метрик використання пам'яті.

На додаток до вбудованих плагінів, Telegraf також надає можливість створювати власні плагіни за допомогою Plugin SDK. Це дозволяє користувачам

збирати метрики з власних додатків і сервісів, які потім можна зберігати і аналізувати за допомогою Telegraf.

Telegraf також може бути налаштований на збір метрик з різними інтервалами, від декількох секунд до декількох хвилин. Це дозволяє користувачам збирати метрики з частотою, яка відповідає їхньому конкретному випадку використання.

Після того, як метрики зібрані Telegraf, вони можуть зберігатися в базі даних часових рядів, таких як InfluxDB.

Окрім InfluxDB, Telegraf також може надсилати метрики до інших сховищ даних, таких як Graphite, OpenTSDB та Prometheus. Це надає користувачам гнучкість у зберіганні метрик у сховищі даних, яке підходить для їхнього конкретного випадку використання.

Загалом, Telegraf - це потужний інструмент для збору даних, який надає користувачам можливість збирати, обробляти та зберігати метрики з широкого спектру джерел. Його гнучкість і можливості налаштування роблять його популярним вибором для моніторингу та аналізу продуктивності серверів і додатків.

InfluxDB - це розподілена і масштабована база даних часових рядів з відкритим вихідним кодом, яка призначена для обробки великих обсягів даних з часовими мітками. Вона є частиною TICK Stack, популярного стеку програмного забезпечення з відкритим кодом, що використовується для збору, зберігання та візуалізації даних часових рядів.

Однією з ключових особливостей InfluxDB є її здатність обробляти високі навантаження на запис і запити. Вона оптимізована для зберігання і запитів даних з часовими мітками, що робить її ідеальною для зберігання метрик, зібраних Telegraf.

InfluxDB надає просту мову запитів, відому як InfluxQL, яку можна використовувати для запитів і агрегації даних часових рядів. InfluxQL підтримує

широкий спектр функцій запитів, включаючи функції агрегації, математичні функції та умовні функції.

На додаток до InfluxQL, InfluxDB також надає HTTP API, який можна використовувати для запитів і запису даних до бази даних. Це дозволяє користувачам взаємодіяти з базою даних за допомогою улюбленої мови програмування або інструменту.

Ще однією ключовою особливістю InfluxDB є підтримка політик збереження даних. Політики зберігання дозволяють користувачам вказувати, як довго дані повинні зберігатися в базі даних. Це може допомогти керувати використанням дискового простору і гарантувати, що дані зберігатимуться лише стільки, скільки потрібно.

InfluxDB також забезпечує підтримку безперервних запитів, які дозволяють користувачам попередньо обчислювати та агрегувати дані за певний період часу. Це може допомогти підвищити продуктивність запитів і зменшити затримку запитів.

На додаток до основних функцій, InfluxDB також надає кілька розширених можливостей, включаючи підтримку високої доступності, кластеризації та автентифікації. Ці функції роблять InfluxDB придатною для використання у великомасштабних і критично важливих додатках.

Загалом, InfluxDB - це потужна і масштабована база даних часових рядів, яка надає користувачам можливість зберігати, запитувати і аналізувати великі обсяги даних з часовими мітками. Проста мова запитів, підтримка політик збереження та розширені можливості роблять її популярним вибором для зберігання та аналізу метрик, зібраних " Telegraf".

Grafana - це інструмент з відкритим вихідним кодом, призначений для візуалізації та аналізу даних часових рядів. Він є частиною TICK Stack, популярного стеку програмного забезпечення з відкритим вихідним кодом, що використовується для збору, зберігання та візуалізації даних часових рядів.

Однією з ключових особливостей Grafana є можливість створювати інтерактивні дашборди, які відображають дані в режимі реального часу. Дашборди можуть бути налаштовані відповідно до конкретних потреб користувача і можуть включати широкий спектр візуалізацій, таких як графіки, таблиці та індикатори.

Grafana надає зручний інтерфейс, який дозволяє користувачам створювати, редагувати та ділитися дашбордами з іншими користувачами. Дашбордами можна ділитися за допомогою URL-адреси, вбудовувати їх в інші веб-сторінки або експортувати у форматі PDF чи зображення.

Ще однією ключовою особливістю Grafana є підтримка широкого спектру джерел даних. Grafana підтримує кілька популярних баз даних часових рядів, включаючи InfluxDB, Prometheus і Graphite. Вона також підтримує інші джерела даних, такі як Elasticsearch, MySQL та PostgreSQL.

Grafana надає потужну мову запитів, відому як Grafana Query Language (GQL), яку можна використовувати для запитів та агрегування даних з різних джерел даних. GQL підтримує широкий спектр функцій запитів, включаючи функції агрегування, математичні функції та умовні функції.

На додаток до GQL, Grafana також надає підтримку плагінів, які можна використовувати для розширення функціональності інструменту. Grafana надає репозиторій плагінів, де користувачі можуть переглядати і завантажувати плагіни для різних джерел даних, візуалізацій і методів автентифікації.

Grafana також підтримує оповіщення, які можна використовувати для надсилання сповіщень при виконанні певних умов. Сповіщення можна надсилати електронною поштою, через Slack, PagerDuty або інші канали сповіщень.

Загалом, Grafana - це потужний і гнучкий інструмент, який надає користувачам можливість візуалізувати та аналізувати дані часових рядів з широкого спектру джерел даних. Зручний інтерфейс, підтримка кастомізації та розширені можливості роблять його популярним інструментом для моніторингу та аналізу продуктивності серверів і додатків.

2.3 Збір даних та попередня обробка даних

Панель інструментів Windows Exporter Dashboard - це попередньо створена інформаційна панель, яку можна використовувати для моніторингу системи Windows. Вона призначена для роботи з системою моніторингу Prometheus, яка є інструментарієм з відкритим вихідним кодом для збору та аналізу даних часових рядів. Панель моніторингу містить кілька попередньо налаштованих панелей, які відображають метрики, пов'язані з використанням процесора, пам'яті, дискового простору, мережевого трафіку та інші дані, пов'язані з продуктивністю.

Однією з головних переваг використання інформаційної панелі Windows Exporter Dashboard є те, що вона дозволяє швидко отримати уявлення про продуктивність системи Windows, не вимагаючи складних налаштувань або кастомізації. Панель моніторингу розроблена таким чином, щоб бути простою у використанні, з чіткими та інтуїтивно зрозумілими візуалізаціями, які полегшують розуміння представлених даних.

Інформаційна панель містить кілька попередньо налаштованих панелей, які відображають ключові показники, пов'язані з продуктивністю системи. Наприклад, панель "Використання CPU" відображає графік використання CPU з плином часу, з окремими рядками для кожного ядра процесора. Це дозволяє легко виявити будь-які сплески або падіння у використанні CPU, які можуть впливати на продуктивність системи.

Аналогічно, на панелі використання пам'яті відображається графік використання пам'яті з часом, з окремими лініями для кожного типу пам'яті (наприклад, фізичної та віртуальної). Це може допомогти виявити будь-які проблеми, пов'язані з пам'яттю, які можуть впливати на продуктивність системи.

Інші панелі інформаційної панелі включають мережевий трафік, дисковий простір і метрики, пов'язані з процесами. Кожна панель призначена для

отримання інформації про певний аспект продуктивності системи і може бути налаштована відповідно до потреб користувача.

Однією з переваг використання інформаційної панелі Windows Exporter Dashboard є те, що її можна легко налаштувати відповідно до потреб користувача. Наприклад, користувачі можуть додавати додаткові панелі до інформаційної панелі для моніторингу певних показників, важливих для їхнього конкретного сценарію використання. Аналогічно, користувачі можуть налаштувати макет і форматування інформаційної панелі, щоб зробити її більш зручною для читання і розуміння.

На завершення, інформаційна панель Windows Exporter Dashboard є потужним інструментом для моніторингу продуктивності системи Windows. Вона включає в себе кілька попередньо налаштованих панелей, які надають уявлення про ключові показники, пов'язані з продуктивністю системи, і можуть бути легко налаштовані відповідно до потреб користувача. Використовуючи цю інформаційну панель у поєднанні з таким інструментом, як Prometheus, користувачі можуть отримати глибоке розуміння продуктивності своєї системи Windows і приймати обґрунтовані рішення про те, як підвищити її ефективність і надійність.

Панель інструментів Windows Exporter Dashboard надає кілька ключових показників, які можна використовувати для аналізу продуктивності системи Windows. Ці показники можна використовувати для отримання інформації про використання процесора, пам'яті, дискового простору, мережевого трафіку та інших даних, пов'язаних з продуктивністю. Аналізуючи ці показники з плином часу, ми можемо виявити тенденції та закономірності в даних, які допоможуть нам приймати обґрунтовані рішення про те, як підвищити ефективність і надійність системи.

System Overview: Цей розділ надає високорівневий огляд продуктивності системи. Він включає основні показники, такі як використання процесора,

пам'яті, диска і мережевого трафіку. Ці показники допомагають користувачам виявити потенційні вузькі місця або обмеження ресурсів у системі.

CPU Metrics: Цей розділ присвячено метрикам, пов'язаним з процесором, таким як використання кожного ядра, час роботи процесора і довжина черги процесора. Ці показники дають уявлення про те, як використовуються ресурси процесора, і можуть допомогти користувачам визначити, чи є проблеми з продуктивністю, пов'язані з використанням процесора.

Memory Metrics: Розділ Метрики пам'яті охоплює різні аспекти, пов'язані з пам'яттю, зокрема загальний обсяг пам'яті, використану пам'ять, вільну пам'ять і відсоток використання пам'яті. Крім того, він надає інформацію про системний кеш, зафіксовані байти та доступні байти. Ці показники допомагають користувачам зрозуміти закономірності використання пам'яті, що може мати вирішальне значення для виявлення проблем, пов'язаних з пам'яттю.

Disk Metrics: Цей розділ присвячено моніторингу продуктивності пристроїв зберігання даних системи. Він надає інформацію про швидкість читання і запису, використання диска і довжину черги на диску, серед інших показників. Ці показники допомагають користувачам виявити потенційні вузькі місця в роботі сховища і переконатися, що їхні системи працюють оптимально.

Network Metrics: Розділ Мережеві показники охоплює різні показники продуктивності, пов'язані з мережею, такі як відправлені та отримані байти, використання мережевого інтерфейсу та мережеві помилки. Ці показники допомагають користувачам контролювати продуктивність мережі, виявляти потенційні проблеми і гарантувати, що їхні системи можуть обробляти необхідний трафік даних.

System Metrics: Цей розділ містить інформацію про різні метрики системного рівня, зокрема час безвідмовної роботи, системні виклики та контекстні перемикачі. Ці показники дають користувачам ширше уявлення про загальну продуктивність системи і можуть бути корисними для діагностики проблем, які можуть бути не одразу помітні з інших показників.

Аналізуючи ці показники в часі, ми можемо отримати уявлення про продуктивність системи і прийняти обґрунтовані рішення про те, як її оптимізувати. Наприклад, якщо ми помічаємо сплеск використання CPU в певний час доби, ми можемо визначити процес або програму, які спричиняють цей сплеск, і скоригувати їхню конфігурацію або використання, щоб зменшити їхній вплив на систему. Аналогічно, якщо ми помітимо, що використання дискового простору постійно зростає, ми зможемо визначити конкретні файли або програми, які займають занадто багато місця на диску, і вжити заходів для вирішення цієї проблеми.

Для аналізу використовуватиметься Метод Холта-Вінтерса, також відомий як потрійне експоненціальне згладжування (TES) та ARIMA.

Метод Холта-Вінтерса, також відомий як метод потрійного експоненціального згладжування, є широко використовуваним методом прогнозування, який особливо ефективний для даних часових рядів, що демонструють тенденції та сезонні закономірності. Він був розроблений Чарльзом Холтом і Пітером Вінтерсом у 1960-х роках.

Метод передбачає моделювання часового ряду як комбінації трьох компонентів: рівня, тренду та сезонності. Рівень являє собою довгострокове середнє значення ряду, тренд - напрямок і величину будь-яких довгострокових змін у ряді, а сезонність - повторювані патерни в ряді протягом фіксованого періоду часу, наприклад, днів, тижнів або місяців.

Існує дві версії методу Холта-Вінтерса: адитивна та мультиплікативна. Основна відмінність між цими двома версіями полягає в способі моделювання сезонної компоненти.

В адитивній версії сезонний компонент моделюється як фіксована величина, яка додається до компонент рівня та тренду. Це доречно, коли величина сезонних коливань у ряді є приблизно постійною в часі, незалежно від рівня ряду. Адитивна версія методу може бути записана так:

$$Level(t) = \alpha(Y(t) - Seasonality(t-m)) + (1 - \alpha)(Level(t-1) + Trend(t-1))$$
$$Trend(t) = \beta(Level(t) - Level(t-1)) + (1 - \beta)Trend(t-1)$$
$$Seasonality(t) = \gamma(Y(t) - Level(t)) + (1 - \gamma)\ Seasonality(t-m)$$

Тут Y(t) - це спостережуване значення ряду в момент часу t, α, β та γ - параметри згладжування, які контролюють ваги, що надаються компонентам рівня, тренду та сезонності, а m - довжина сезонного циклу.

У мультиплікативній версії сезонний компонент моделюється як відсоток від компонента рівня. Це доречно, коли величина сезонних коливань у ряді змінюється залежно від рівня ряду. Мультиплікативну версію методу можна записати так:

$$Level(t) = \alpha(Y(t) / Seasonality(t-m)) + (1 - \alpha)(Level(t-1) + Trend(t-1))$$
$$Trend(t) = \beta(Level(t) - Level(t-1)) + (1 - \beta)Trend(t-1)$$
$$Seasonality(t) = \gamma(Y(t) / Level(t)) + (1 - \gamma)\ Seasonality(t-m)$$

Тут Y(t) знову є спостережуваним значенням ряду в момент часу t, а α, β і γ є тими самими параметрами згладжування, що і в адитивній версії.

Вибір між адитивною та мультиплікативною версіями методу Холта-Вінтерса залежить від характеру часового ряду, що аналізується. Якщо сезонні коливання в ряді приблизно постійні за величиною в часі, то доцільно використовувати адитивну версію. Якщо ж величина сезонних коливань

змінюється в залежності від рівня ряду, більш доречною є мультиплікативна версія.

Метод ARIMA був розроблений у 1970-х роках Джорджем Боксом та Гвілімом Дженкінсом як узагальнення методології Бокса-Дженкінса. Метод ARIMA є популярним підходом до прогнозування часових рядів, який передбачає підбір моделі до даних і використання цієї моделі для прогнозування майбутніх значень. Метод поєднує в собі три різні техніки: авторегресію (AR), інтегрування (I) та рухомі середні (MA). Авторегресія передбачає використання минулих значень ряду для прогнозування майбутніх значень. Диференціювання використовується для того, щоб зробити часовий ряд стаціонарним, що означає, що його середнє та дисперсія є постійними в часі. Рухомі середні передбачають використання попередніх значень ряду для згладжування будь-яких випадкових коливань у даних.

Метод ARIMA має кілька переваг над іншими методами прогнозування часових рядів. Однією з ключових переваг є те, що він може працювати з широким спектром даних часових рядів. Його можна використовувати для прогнозування як короткострокових, так і довгострокових тенденцій, а також для моделювання як стаціонарних, так і нестаціонарних даних. Крім того, ARIMA є гнучким методом, який можна адаптувати до різних типів даних часових рядів, таких як сезонні дані, трендові дані та циклічні дані.

ARIMA широко використовується в багатьох галузях для прогнозування часових рядів. У фінансовій сфері його використовують для прогнозування цін на акції, обмінних курсів та відсоткових ставок. В економіці - для прогнозування ВВП, інфляції та рівня безробіття. У прогнозуванні погоди його використовують для прогнозування температури, опадів та інших погодних змінних. Метод також застосовується в епідеміології, де його використовують для прогнозування спалахів захворювань.

З моменту свого створення метод ARIMA зазнав кількох удосконалень. Зокрема, розробка сезонних моделей ARIMA (SARIMA) дозволила моделювати

та прогнозувати сезонні часові ряди даних. Метод також було розширено для включення екзогенних змінних, що дозволило підвищити точність прогнозів.

2.4 Статистичний аналіз

ARIMA, абревіатура від AutoRegressive Integrated Moving Average (авторегресійне інтегроване рухоме середнє), є відомим класом моделей, які широко застосовуються у сфері прогнозування часових рядів у статистиці та економетриці.

Авторегресійний сегмент моделі ARIMA інкапсулює ідею регресії змінної, що нас цікавить, на її власні попередні значення. Термін "авторегресія" означає варіант регресії, в якому змінна використовується як пояснювальна змінна по відношенню до самої себе.

Формальне представлення авторегресійної моделі порядку *"p"*, що позначається як *AR(p)*, може бути сформульоване наступним чином:

$$Y_t = c + \varphi_1 Y_{t-1} + \varphi_2 Y_{t-2} + \ldots + \varphi_p Y_{t-p} + \varepsilon_t \quad (1)$$

У цьому рівнянні

Y_t представляє змінну, що цікавить нас у момент часу 't'

'c' означає постійний член

$\varphi_1, \varphi_2, \varphi_p$ - параметри моделі

$Y_{t-1}, Y_{t-2}, Y_{t-p}$ - лагові значення Y

ε_t - член похибки в момент часу *'t'*

Інтегрована складова відноситься до диференціювання, що застосовується для досягнення стаціонарності часового ряду. Ряд вважається нестаціонарним, якщо його статистичні властивості, такі як середнє, дисперсія та автокореляція,

не є постійними з часом. Диференціювання - це метод, який застосовується для досягнення стаціонарності.

Якщо "*d*" - це необхідний порядок диференціювання, то вихідний часовий ряд Y перетворюється в *I(d)* ряд *Y'* за допомогою рівняння:

$$Y'_t = Y_t - Y_{t-1} \qquad (2)$$

(для d=1)

Сегмент моделі рухомогоого середнього включає зв'язок між спостереженням і залишковою похибкою моделі рухомого середнього, застосованої до запізнілих спостережень.

Формальне представлення моделі рухомого середнього порядку "*q*", що позначається як *MA(q),* може бути сформульоване наступним чином:

$$Y_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \ldots + \theta_q \varepsilon_{t-q} \qquad (3)$$

У цьому рівнянні

Y_t представляє змінну, що цікавить нас у момент часу "t

μ - середнє значення ряду

$\theta_1, \theta_2, \theta_q$ -параметри моделі

$\varepsilon_t, \varepsilon_{t-1}, \varepsilon_{t-2}, \varepsilon_{t-q}$- члени похибки в попередні моменти часу

Об'єднання цих трьох компонентів дає загальну модель *ARIMA(p, d, q),* де "*p*" позначає порядок частини авторегресії, "*d*" означає ступінь першого диференціювання, а "*q*" - порядок частини рухомого середнього.

Моделі ARIMA пристосовані до даних часових рядів з подвійною метою: розуміння даних і прогнозування майбутніх точок ряду. Ці моделі особливо корисні в сценаріях, де дані демонструють ознаки нестаціонарності, і початковий крок диференціювання (що відповідає "інтегрованій" частині моделі) може бути застосований один або кілька разів для усунення цієї нестаціонарності.

Однак необхідно зазначити, що ці моделі припускають стаціонарність. Тому в контексті нестаціонарного ряду ці моделі часто використовуються разом з диференціюванням або іншими методами, що забезпечують стаціонарність. Якщо ряд не є стаціонарним, застосування цих моделей може призвести до хибних результатів.

Фур'є-аналіз - це математичний інструмент, який використовується для аналізу даних часових рядів шляхом розкладання сигналу на частотні складові. Перетворення Фур'є - це математичний метод, який перетворює сигнал з часової області в частотну. Для неперервного в часі сигналу x(t) перетворення Фур'є має вигляд:

$$X(f) = \int x(t)\, e^{-2\pi i f t} dt \quad (4)$$

де $X(f)$ - це частотне представлення $x(t)$, а i - уявна одиниця. Ця формула виражає той факт, що сигнал $x(t)$ може бути представлений як сума синусоїдальних і косинусоїдальних хвиль різних частот.

Для дискретних сигналів використовується дискретне перетворення Фур'є (ДПФ). Дискретне перетворення Фур'є визначається наступним чином:

$$X(k) = \sum x(t) e^{-2\pi i k n / N} \quad (5)$$

де $X[k]$ - це частотне представлення $x[n]$, n - індекс відліку, k - індекс частоти, а N - загальна кількість відліків у часовому ряді.

Швидке перетворення Фур'є (ШПФ) є ефективним алгоритмом для обчислення ДПФ. Він широко використовується в обробці сигналів та аналізі даних.

Фур'є-аналіз може бути корисним для аналізу часових рядів даних з періодичними закономірностями. Розкладаючи сигнал на частотні складові, можна визначити домінуючі частоти та їхні амплітуди, які можна використовувати для моделювання та прогнозування поведінки даних. Однак Фур'є-аналіз передбачає, що дані є періодичними, що може бути не так для всіх часових рядів.

Таким чином, Фур'є-аналіз є потужним інструментом для аналізу даних часових рядів шляхом розкладання сигналу на його частотні складові. Перетворення Фур'є, ДПФ і ШПФ - це математичні формули, які використовуються для цієї мети. Фур'є-аналіз особливо корисний для даних з періодичною структурою, але важливо враховувати обмеження цього методу і його застосовність до конкретних характеристик даних, що аналізуються.

Експоненціальне згладжування - це широко використовуваний метод прогнозування часових рядів, який має на меті передбачити майбутні значення часового ряду на основі минулих спостережень. Він особливо корисний для даних з постійною закономірністю, такою як сезонність або тренд, але також може бути використаний для даних без чітко вираженої закономірності. Методи експоненціального згладжування прості у розумінні та застосуванні, і вони дають відносно точні прогнози з мінімальними обчислювальними зусиллями.

Фундаментальна ідея експоненціального згладжування полягає в тому, щоб надати більшої ваги останнім спостереженням і меншої ваги старим спостереженням. Це досягається шляхом застосування константи згладжування, альфа *(α),* яка знаходиться в діапазоні від 0 до 1. Чим ближче α до 1, тим більша вага надається останнім спостереженням, тоді як чим ближче α до 0, тим більша вага надається історичним спостереженням.

Найпростішою формою експоненціального згладжування є одиночне експоненціальне згладжування (SES), яке підходить для часових рядів без тренду або сезонності. Прогноз на наступний період (t+1) задається наступною формулою:

$$F(t+1) = \alpha Y(t) + (1 - \alpha) F(t) \qquad (6)$$

Де:

$F(t+1)$ - прогноз на наступний період $(t+1)$

α - константа згладжування, від 0 до 1

$Y(t)$ - фактичне значення в момент часу t

$F(t)$ - прогноз на поточний період (t)

Однак, одинарне експоненціальне згладжування не може обробляти дані часових рядів з трендами або сезонністю. Для усунення цього обмеження було розроблено подвійне експоненціальне згладжування (DES), також відоме як лінійний метод Холта. Він враховує як рівень, так і тренд часового ряду. Метод використовує два рівняння:

Рівняння рівня (S_t):

$$S_t = \alpha Y(t) + (1 - \alpha) (S_{t-1}+T_{t-1}) \qquad (7)$$

Рівняння тренду (T_t):

$$T_t = \beta(S_t+S_{t-1}) + (1 - \beta)T_{t-1} \qquad (8)$$

Прогноз на наступний період $(t+1)$ визначається за формулою:

$$F_{t+1} = S_t + T_t \qquad (9)$$

Де

S_t- компонент рівня в момент часу t

T_t - компонент тренду в момент часу t

β - константа згладжування тренду, від 0 до 1

Метод Холта-Вінтерса, також відомий як потрійне експоненціальне згладжування (TES), є потужним методом прогнозування часових рядів, який розширює можливості подвійного експоненціального згладжування (лінійного методу Холта), враховуючи не тільки рівень і тренд, але й сезонність даних. Цей метод особливо корисний для прогнозування майбутніх значень у часових рядах, які демонструють як тренд, так і сезонні закономірності, наприклад, дані про продажі, температуру або веб-трафік.

Метод Холта-Вінтерса складається з трьох рівнянь, які використовуються для обчислення рівня, тренду та сезонних компонентів часового ряду:

Рівняння рівня S_t:

$$S_t = \alpha \frac{Y(t)}{I_{t-L}} + (1 - \alpha)\ (S_{t-1}+T_{t-1}) \qquad (10)$$

Це рівняння обчислює компонент рівня в момент часу t. Воно згладжує співвідношення фактичного значення $Y(t)$ і сезонного компонента I_{t-L} за допомогою константи згладжування рівня (α) і поєднує його з попереднім рівнем S_{t-1} і трендом T_{t-1}.

Рівняння тренду (Tt):

$$T_t = \beta(S_t+S_{t-1}) + (1 - \beta)T_{t-1} \qquad (11)$$

Рівняння тренду обчислює компонент тренду в момент часу *t*. Воно згладжує різницю між поточним рівнем S_t і попереднім рівнем S_{t-1} за допомогою константи згладжування тренду β і поєднує його з попереднім трендом T_{t-1}.

Сезонне рівняння I_t:

$$I_t = \gamma \frac{Y(t)}{S_t} + (1 - \gamma)\ (I_{t-L}) \qquad (12)$$

Сезонне рівняння обчислює сезонну складову в момент часу t. Воно згладжує співвідношення фактичного значення $Y(t)$ і поточного рівня S_t за допомогою константи сезонного згладжування γ і поєднує його з попередньою сезонною складовою I_{t-L}, де L - довжина сезонного періоду.

Прогноз на наступний період $(t+1)$ визначається за формулою:

$$F_{t+1} = (S_t + T_t)I_{t-L+1} \qquad (13)$$

Ця формула поєднує компоненти рівня S_t, тренду T_t та сезонності I_{t-L+1} для отримання прогнозу на наступний період $(t+1)$.

Константи згладжування α, β і γ відіграють вирішальну роль у точності методу Холта-Вінтерса. Вони визначають, яка вага надається останнім спостереженням при обчисленні рівня, тренду та сезонних компонентів. Зазвичай ці константи обираються шляхом мінімізації середньоквадратичної похибки (MSE) або іншої метрики похибки прогнозів.

Метод Холта-Вінтерса можна застосовувати як до адитивних, так і до мультиплікативних сезонних моделей. В адитивному випадку сезонні коливання є приблизно постійними за величиною, тоді як у мультиплікативному випадку сезонні коливання є пропорційними до рівня ряду.

Отже, метод Холта-Вінтерса є потужним і гнучким методом прогнозування для часових рядів даних, що містять як тренд, так і сезонні коливання. Ретельно підбираючи константи згладжування та довжину сезонного періоду, він може забезпечити точні та надійні прогнози для різних застосувань, таких як прогнозування продажів, прогнозування попиту на енергію та управління запасами.

Здатність методу Холта-Вінтерса враховувати тенденції, сезонність та рівень у даних часових рядів за допомогою експоненціального згладжування пропонує більш комплексне та адаптивне рішення для даних про використання процесорів. Цей метод ефективно управляє різними ступенями коливань і потенційними кореляціями, які можуть бути присутніми в даних, забезпечуючи більш точний і надійний прогноз.

Всі три методи, ARIMA, аналіз Холта-Вінтерса та Фур'є, можна використовувати для згладжування даних часових рядів. Однак їхні підходи відрізняються і мають свої сильні та слабкі сторони.

ARIMA (AutoRegressive Integrated Moving Average - авторегресійна інтегрована рухома середня) - популярна модель часових рядів, яка використовує минулі значення та похибки для прогнозування майбутніх значень. Вона може обробляти широкий спектр моделей часових рядів, включаючи тренди, сезонність і циклічність. Модель ARIMA часто використовується для короткострокового прогнозування і може бути ефективною для згладжування короткострокових коливань у даних CPU. Однак вона може бути не такою

ефективною для довгострокового згладжування, оскільки може не врахувати довгострокові тенденції або сезонність.

Метод Холта-Вінтерса - ще один популярний метод часових рядів, який включає як трендові, так і сезонні компоненти. Це метод прогнозування, який використовує комбінацію методів експоненціального згладжування для оцінки рівня, тренду та сезонних компонентів часового ряду. Метод Холта-Вінтерса може бути ефективним для згладжування сезонних або циклічних коливань у даних CPU, і він може бути кращим за ARIMA для довгострокового згладжування. Однак він може бути не таким ефективним для обробки несезонних тенденцій або закономірностей у даних.

Аналіз Фур'є - це математичний метод, який розкладає часовий ряд на складові частоти. Він корисний для аналізу періодичних сигналів та визначення домінуючих частот у часовому ряді. Фур'є-аналіз може бути корисним для виявлення сезонних або циклічних закономірностей у даних CPU, але він може бути не таким ефективним для довгострокового згладжування або для обробки неперіодичних закономірностей.

Отже, найкращий метод для згладжування CPU буде залежати від конкретних характеристик даних і цілей аналізу. Якщо дані мають короткострокові коливання і не мають чіткої сезонності, ARIMA може бути гарним вибором. Якщо дані мають сильні сезонні або циклічні коливання, більш ефективним може бути метод Холта-Вінтерса. Аналіз Фур'є може бути корисним для виявлення періодичних закономірностей у даних, але може бути не найкращим вибором для згладжування даних.

3 Узагальнення результатів

У цьому розділі ми досліджуємо продуктивність монолітного веб-додатку, а саме додатку з класичною 3-рівневою архітектурою (докладніше див.

Додаток А) шляхом моніторингу продуктивності процесора протягом певного періоду часу. Ми оцінюємо вплив автотестів на продуктивність системи та демонструємо ефективність методу згладжування Холта-Вінтерса у виявленні патернів використання процесора, зберігаючи при цьому видимість сплесків, спричинених автотестами.

Для аналізу продуктивності процесора монолітного веб-додатку ми збираємо дані протягом 10 хвилин, роблячи вибірку продуктивності кожні 10 секунд для кожного з 4 ядер процесора. Це дає нам загалом 240 спостережень на кожне ядро (4 ядра x 60 вибірок на ядро), які ми використовуємо для створення графічного представлення використання процесора.

У певний момент протягом періоду моніторингу ми створюємо навантаження на систему шляхом запуску автотестів. Ці автотести слугують для імітації реального використання веб-додатку і створюють помітний стрибок у продуктивності процесора. Цей стрибок дозволяє нам дослідити, як система реагує на збільшення робочого навантаження, і допомагає визначити потенційні вузькі місця або області для оптимізації.

Щоб краще зрозуміти основні закономірності в даних про продуктивність процесора, ми застосували методи згладжування Холта-Вінтерса та ARIMA, які є широко використовуваними техніками прогнозування часових рядів. Ці методи згладжують дані, зберігаючи важливі особливості, такі як стрибок, спричинений запуском автотесту. Застосування методів Холта-Вінтерса та ARIMA до нашого набору даних генерує згладжені часові ряди, які виявляють будь-які основні закономірності або тенденції в продуктивності процесора.

Графічне представлення даних про продуктивність процесора показує помітний стрибок у використанні при виконанні автотестів, що підтверджує наші очікування, що запуск автотестів вплине на продуктивність системи. При застосування методів згладжування Холта-Вінтерса та ARIMA до вихідних даних ми бачимо, що зграфік продуктивності процесора вирівнявся,що дозволяє

нам легше виявляти закономірності або тенденції. Однак стрибок, спричинений запуском автотесту, залишається видимим, що ілюструє ефективність методів Холта-Вінтерса та ARIMA у збереженні важливих особливостей даних.

На закінчення, ця глава демонструє корисність моніторингу продуктивності процесора для отримання уявлення про поведінку монолітного веб-додатку під різними робочими навантаженнями. Застосовуючи методів згладжування Холта-Вінтерса та ARIMA до вихідних даних, ми змогли виявити основні закономірності та тенденції, зберігаючи видимість стрибка, спричиненого запуском автотесту. Цей аналіз надає цінну інформацію для оптимізації системи та управління ресурсами, що є важливими факторами для успішної роботи монолітного веб-додатку.

3.1 Аналіз метричних відхилень

Часовий ряд продуктивності процесора - це послідовний набір даних, який представляє продуктивність центрального процесора (CPU) за певний період. Набір даних зазвичай містить інформацію про різні показники продуктивності, такі як кількість інструкцій в секунду (IPS), операцій з плаваючою комою в секунду (FLOPS), тактова частота, розмір кеш-пам'яті та енергоефективність, серед іншого. Кожна точка даних у часовому ряді відповідає певному моменту часу або певній моделі чи поколінню процесора.

Аналіз часового ряду продуктивності процесора може дати уявлення про розвиток та еволюцію обчислювальних технологій. Дивлячись на історичні дані, можна спостерігати тенденції, закономірності та темпи зростання, які інформують про розвиток процесорів з плином часу. Ця інформація корисна для прогнозування майбутніх покращень продуктивності, прийняття обґрунтованих рішень щодо оновлення апаратного забезпечення або для цілей бенчмаркінгу.

На рисунку (див. Рис. 1) зображений часовий ряд продуктивності CPU для наших даних, зображений за допомогою інструменту Grafana.

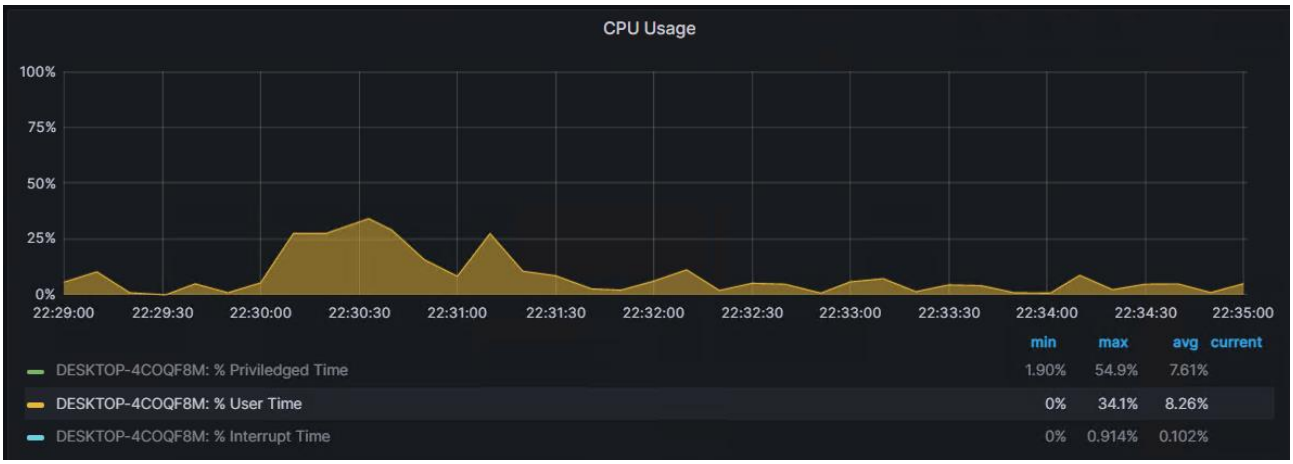


Рис. 1

Хоча метод Холта-Вінтерса є потужним інструментом прогнозування при роботі з даними, які мають чіткі тенденції та сезонність, він не завжди може бути найкращим вибором. У сценаріях, де ці характеристики відсутні, наприклад, як на нашому графіку (див. Рис. 1), простіші методи, такі як експоненціальне згладжування, часто можуть забезпечити більш точні та надійні прогнози. Зосереджуючись на рівні даних і відмовляючись від пошуку неіснуючих трендів або сезонних закономірностей, експоненціальне згладжування підкреслює силу простоти в прогнозуванні часових рядів.

За відсутності тренду або сезонності моделі, які намагаються врахувати ці компоненти, наприклад, метод Холта-Вінтерса, ризикують бути надмірно підігнаними. Надмірне припасування відбувається, коли модель вивчає випадковий шум або коливання в даних, а не основний процес, що призводить до неточних і ненадійних прогнозів. На противагу цьому, експоненціальне згладжування, яке фокусується лише на компоненті рівня, може ефективно зменшити цей ризик, забезпечуючи більш точні та надійні прогнози.

Саме тому було прийняте рішення використовувати в реалізації метод Хольта-Вінтерса.

Наданий код (див. Додаток В) є комплексною реалізацією, призначеною для аналізу та візуалізації даних про використання чотирьох процесорів у часі. Процес аналізу та візуалізації включає імпорт необхідних бібліотек, обробку

даних з файлу CSV, застосування методів прогнозування та створення серії графіків для полегшення розуміння основної інформації.

Спочатку дані отримуються з файлу CSV під назвою "cpuData.csv" і зберігаються у фреймі даних під назвою ProcessorsCpuIndicatorsByTime. Далі код обчислює середнє значення використання процесора для кожного рядка у кадрі даних, створюючи новий стовпчик з назвою Xmean. Ці початкові кроки створюють основу для подальших маніпуляцій з даними та візуалізації.

Використовуючи можливості бібліотеки Plotly, код генерує серію графіків, які дають уявлення про закономірності використання процесора. Зокрема, побудовано наступні графіки:

Графік, що ілюструє середнє використання CPU з плином часу, який слугує узагальненим представленням набору даних (див. Рис. 2).

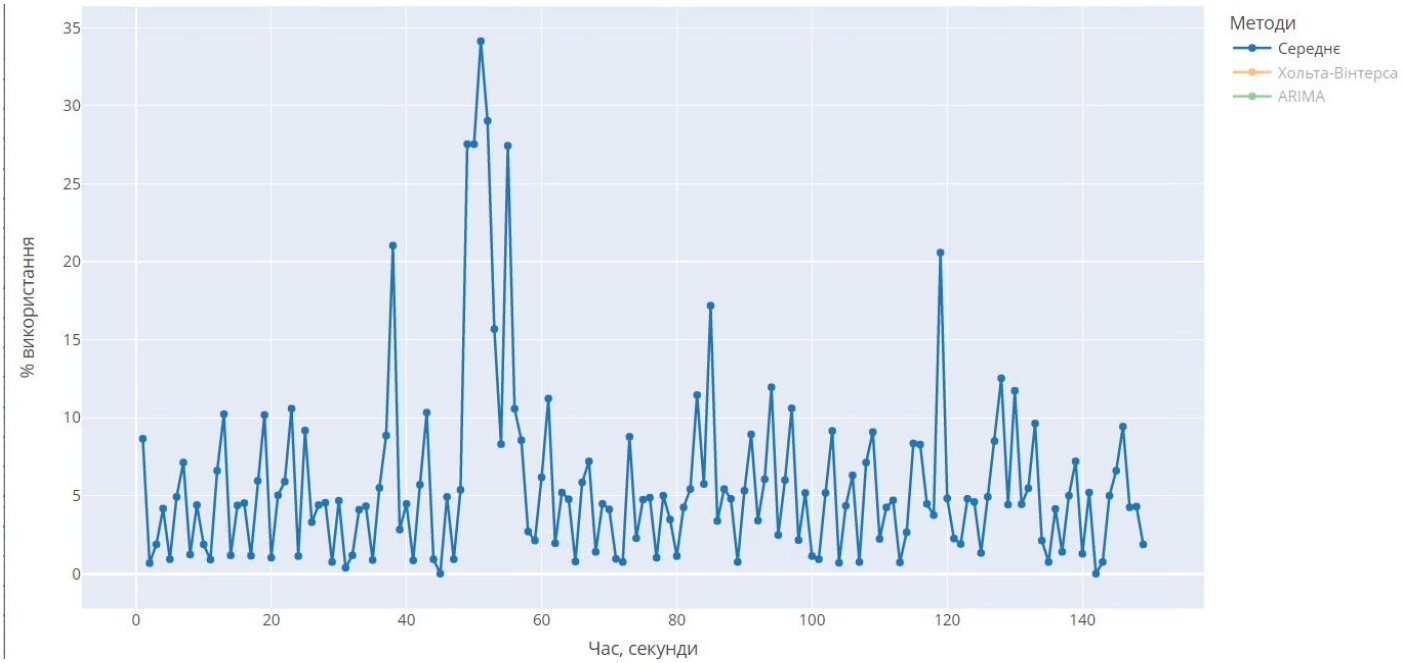


Рис. 2

Більш детальний графік, що відображає індивідуальне використання CPU для кожного процесора в часі, представлений окремими лініями для кожного процесора(див. Рис. 3). Ця візуалізація дозволяє виявити тенденції та потенційні аномалії, характерні для конкретного процесора.

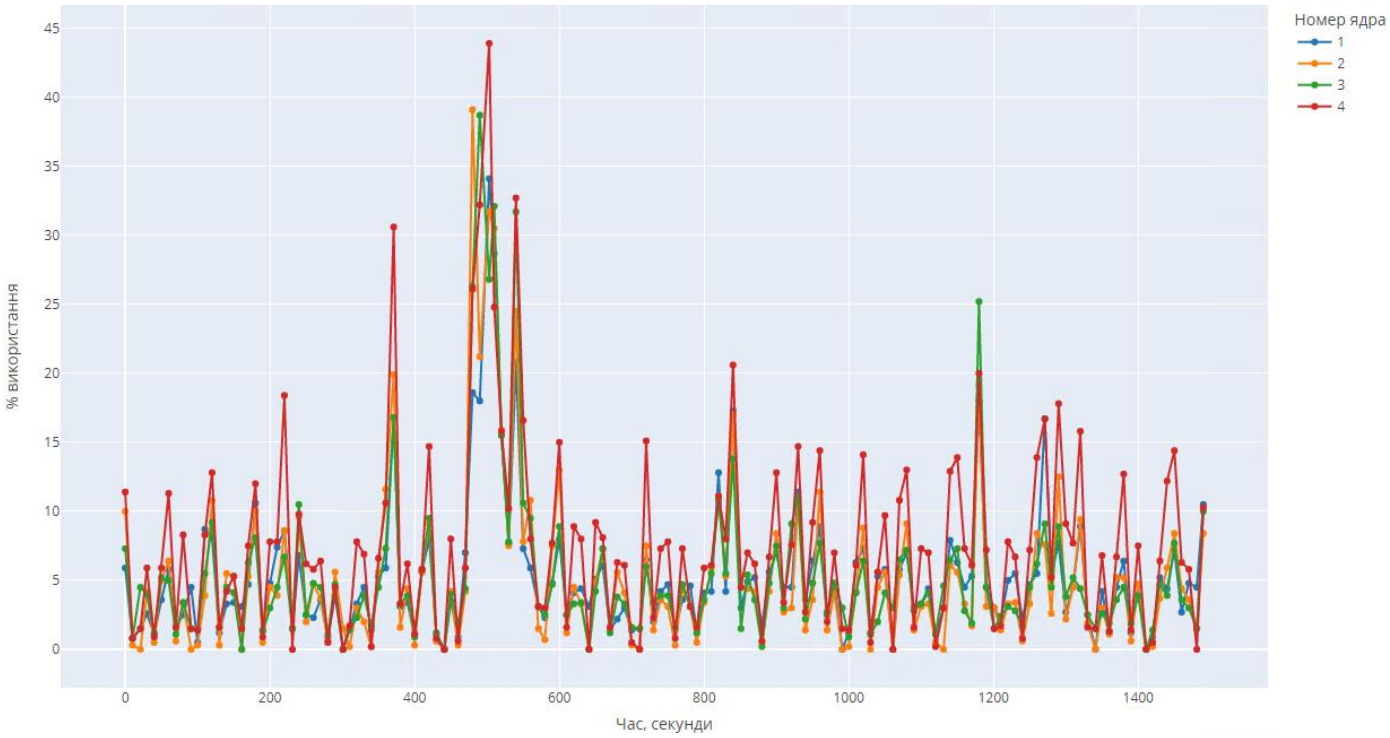


Рис. 3

Код продовжує вдосконалювати аналіз, реалізуючи наступні кроки обробки часових рядів:

1. Підготовка даних часового ряду, що передбачає додавання значень використання процесора для кожного процесора з подальшим додаванням невеликої константи (0,01) для обходу нульових значень.
2. Застосування методу прогнозування HoltWinters до даних часових рядів для кожного процесора, генерування підігнаних значень, які представляють згладжену тенденцію використання процесорів для кожного процесора.
3. Обчислення середнього значення підігнаних значень, яке згодом зберігається у змінній з назвою final.
4. Застосування методу прогнозування ARIMA до даних часового ряду середнього CPU.

Створюється новий фрейм даних data, що містить підібрані значення для кожного процесора. Крім того, за допомогою функції apply() обчислюється

медіанне значення, що забезпечує ще одну центральну міру тенденції для набору даних.

На завершення аналізу за допомогою бібліотеки Plotly створюється інтерактивний графік:

1. Середнє значення підігнаних значень (остаточне), що забезпечує згладжене представлення загальних тенденцій використання процесора(див. Рис 4, Рис 5, Рис 6, , Рис 8).
2. Значення після застосування згладження методом ARIMA(див. Рис. 4, Рис.7, Рис. 8)
3. Початкові середні значення використання CPU (timeSeriesMean), що слугують для порівняння зі згладженими тенденціями (див. Рис. 4, Рис. 5).

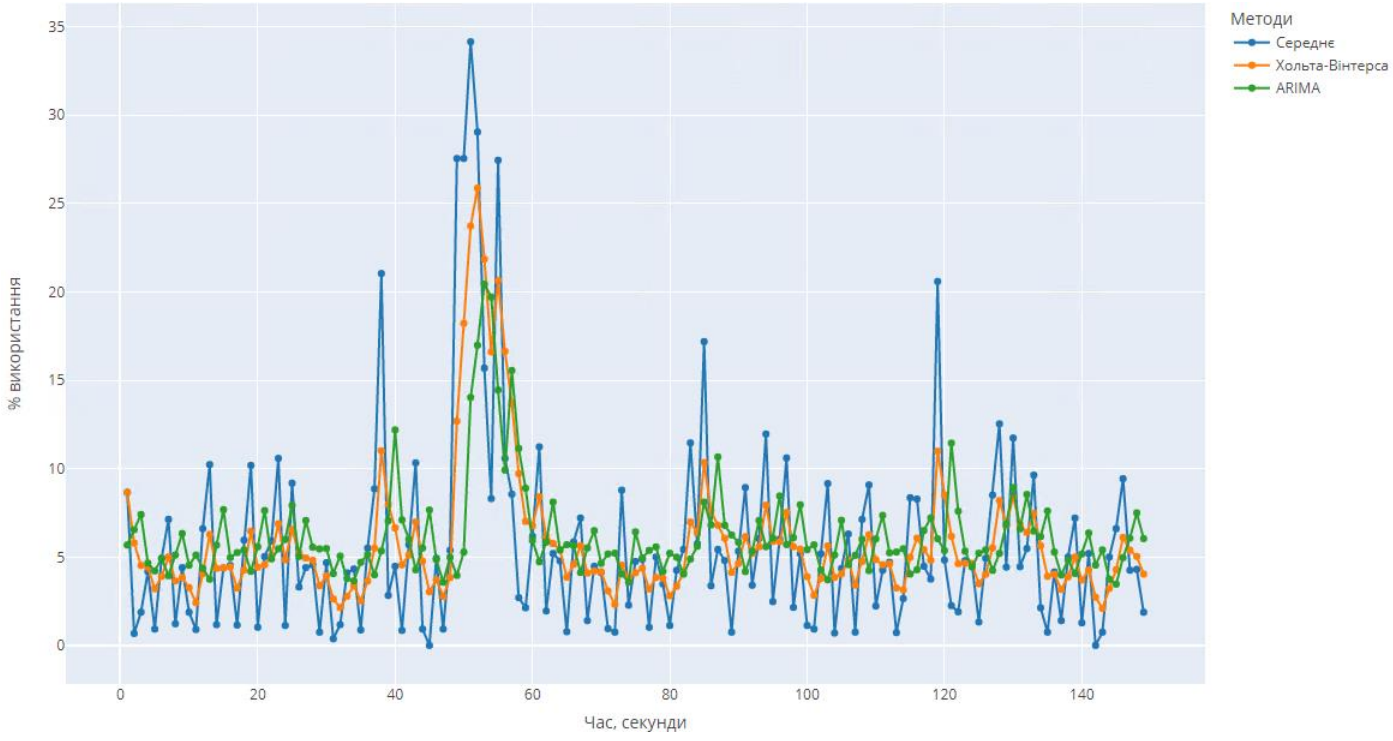


Рис. 4

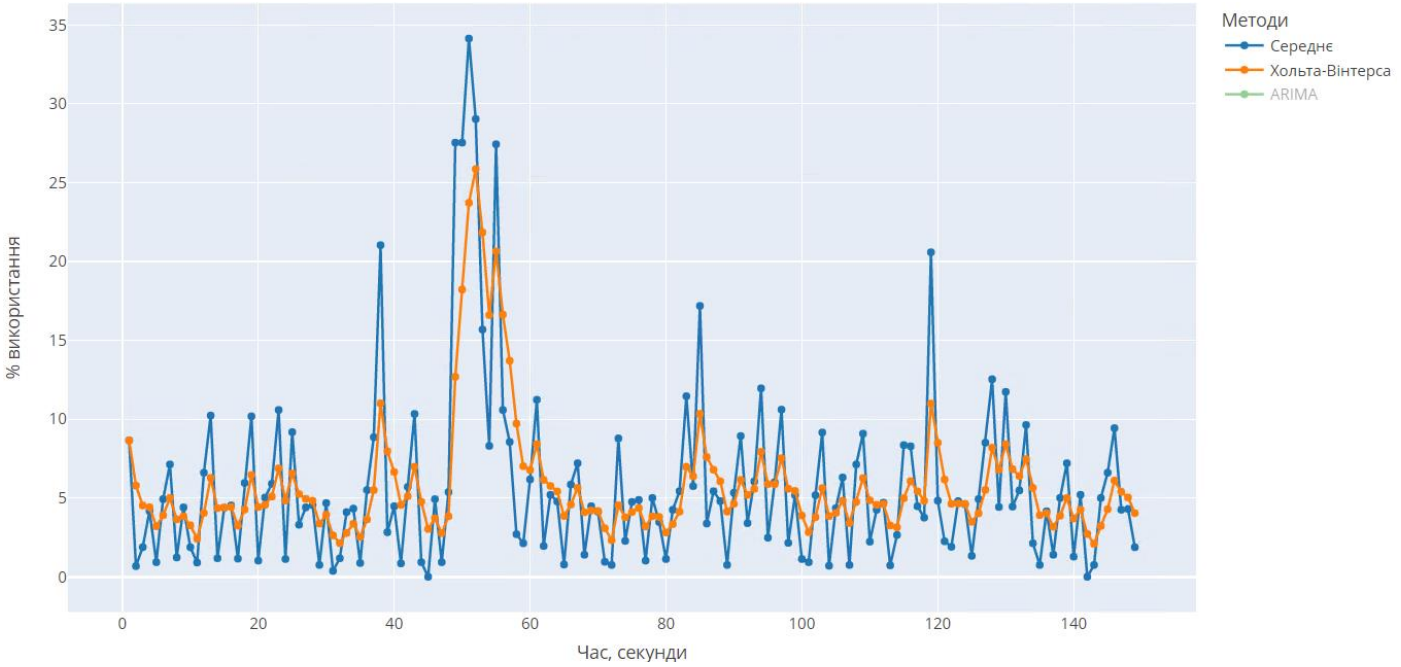

Методи
Середнє
Хольта-Вінтерса
ARIMA
% використання
Час, секунди
0
5
10
15
20
25
30
35
0
20
40
60
80
100
120
140

Рис. 5

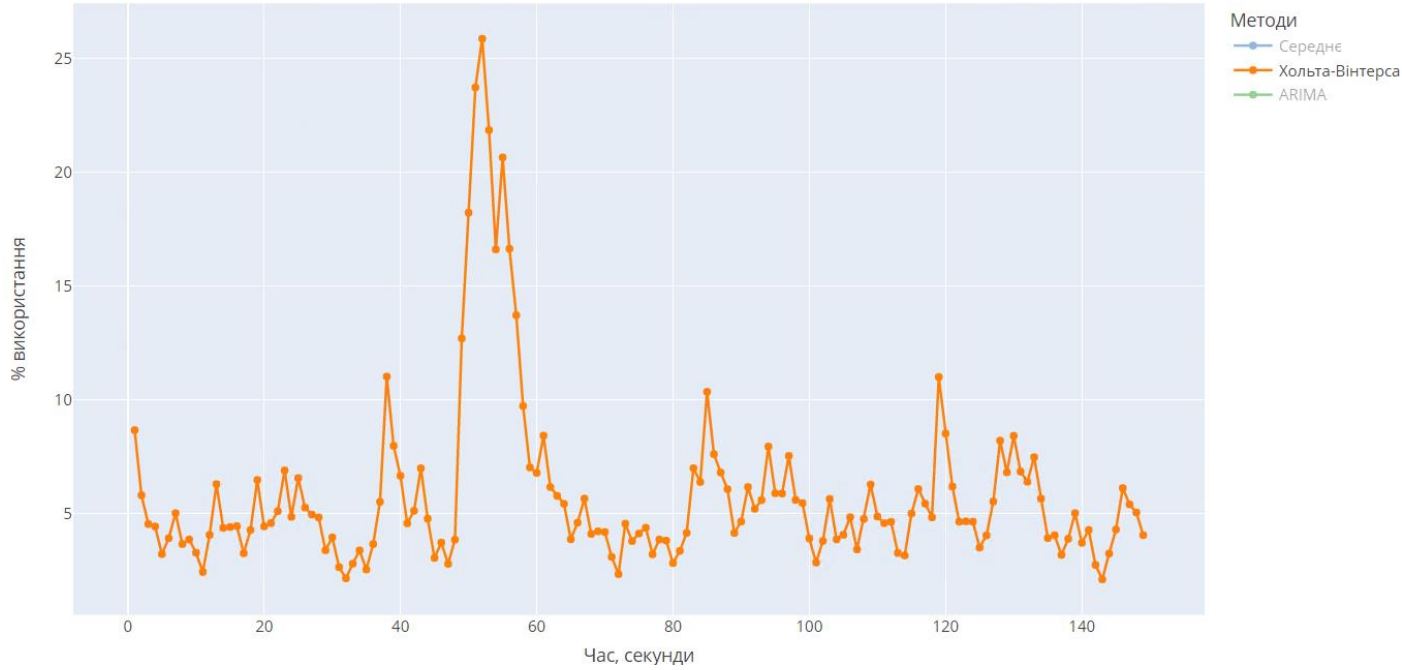

Методи
Середнє
Хольта-Вінтерса
ARIMA
% використання
Час, секунди
5
10
15
20
25
0
20
40
60
80
100
120
140

Рис. 6

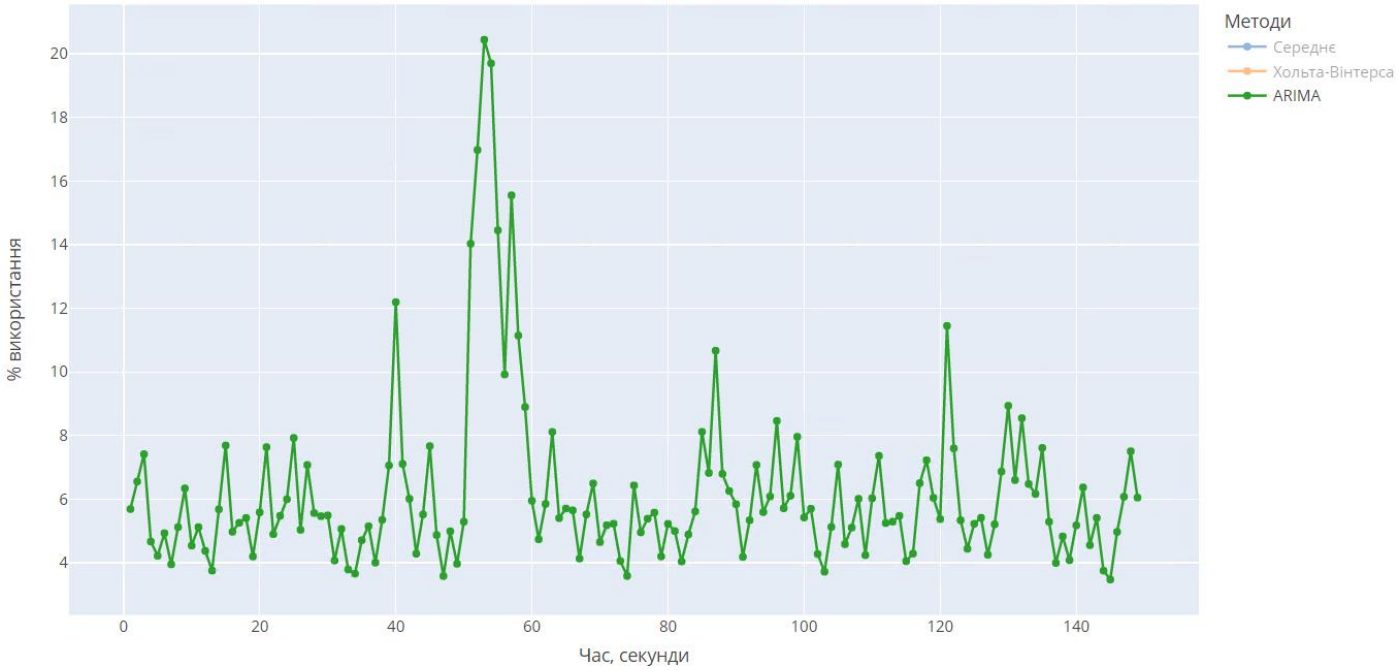


Рис. 7

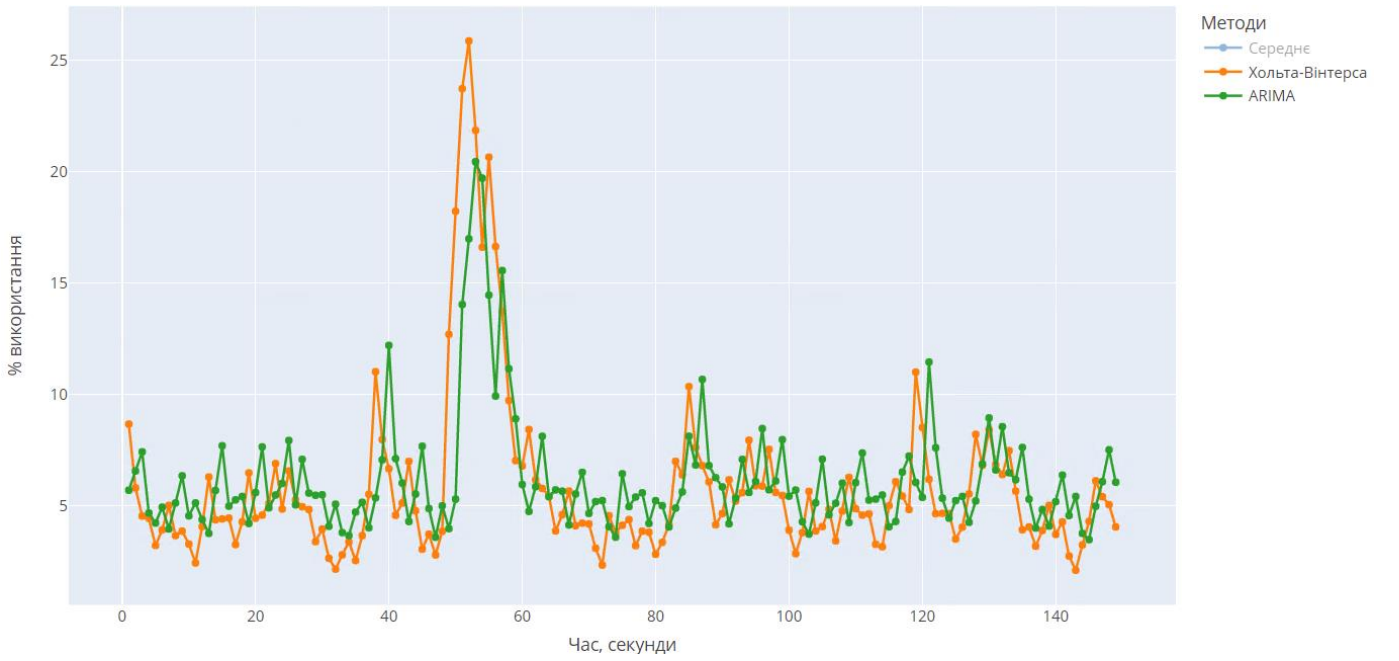


Рис. 8

Завдяки такому комплексному процесу візуалізації та аналізу даних про використання CPU, ми можемо виявити основні закономірності та тенденції, що потенційно можуть призвести до оптимізації продуктивності системи та більш ефективної діагностики потенційних проблем. Інтеграція методів маніпулювання даними та прогнозування забезпечує багатогранний підхід до

розуміння поведінки використання процесора, що дозволяє приймати обґрунтовані рішення щодо оптимізації системи.

На основі графіків, створених на основі наданого коду, можна зробити кілька висновків щодо використання чотирьох процесорів з плином часу. Візуалізації, що включають середнє використання процесора, індивідуальне використання процесора та прогнозування часових рядів за допомогою підібраних значень, сприяли всебічному розумінню основних закономірностей і тенденцій набору даних.

З початкового графіка (див. Рис. 2), що відображає середнє використання процесора, ми можемо спостерігати загальну тенденцію використання процесора протягом усього аналізованого періоду. Цей високорівневий вид висвітлює коливання у використанні і момент створення навантеження на процесори.

Другий графік(див. Рис. 3), який окреслює використання кожного процесора з плином часу, дозволяє виявити специфічні для кожного процесора закономірності та потенційні аномалії. Вивчаючи продуктивність кожного процесора окремо, ми можемо встановити, чи демонструють певні процесори стабільно вищу або нижчу завантаженість порівняно з їхніми аналогами. Наприклад у нашому випадку добре видно, що саме у 4 процесора більша завантаженість, порівняно з іншими. Ця інформація може бути використана для прийняття рішень щодо розподілу робочого навантаження, модернізації процесорів або цілеспрямованої оптимізації.

Застосування методу прогнозування Холта-Вінтерса та ARIMA дозволило отримати згладжені значення для кожного процесора та для середнього, виявивши тенденції, які можуть бути не помітними у вихідних даних (див. Рис.5) Остаточний інтерактивний графік поєднує середнє значення підігнаних значень

і початкове середнє значення використання процесора, забезпечуючи цілісне представлення набору даних (див. Рис. 4).

З графіку (див. Рис. 4) можна зробити висновок, що згладження методами пройшло вдало, адже шуми стали менш помітними, а стрибок у продуктивності процесору залишається видимим (що ілюструє нам ефективність методів). До 40 секунди можна побачити що метод Хольта-Вінтерса краще за ARIMA згладив шум. Порівнюючи ARIMA та Хольта-Вінтерса, можна зробити висновок, що другий метод краще зберіг важливі особливості (а саме пік навантаження з 40 до 60 секунди).

Застосувавши методи згладжування Холта-Вінтерса та ARIMA до даних про використання процесорів, ми створили згладжений часовий ряд, який відображає основні тенденції в даних. Цей процес усунув випадкові коливання і шум, дозволяючи виявити закономірності у використанні процесорів для кожного процесора. Вивчаючи ці тенденції, ми можемо краще зрозуміти, як використання процесорів змінювалося з часом.

На Рис. 8 ми можемо бачити відмінності у згладжених часових даних, що зумовлені в першу чергу підбором констант для методів. Три константи, що використовуються в методі Холта-Вінтерса, є наступними:

Альфа (параметр згладжування рівня): Константа альфа визначає коефіцієнт згладжування для компонента рівня часового ряду. Зазвичай вона встановлюється в діапазоні від 0 до 1, причому вищі значення надають більшої ваги останнім спостереженням.

Бета (параметр згладжування для тренду): Константа бета визначає коефіцієнт згладжування для трендової складової часового ряду. Зазвичай встановлюється в діапазоні від 0 до 1, причому вищі значення надають більшої ваги нещодавнім трендам.

Гамма (параметр згладжування для сезонної складової): Гамма-стала визначає коефіцієнт згладжування для сезонної складової часового ряду. Вона

також зазвичай встановлюється в діапазоні від 0 до 1, причому вищі значення надають більшої ваги нещодавнім сезонним тенденціям.

У R значення за замовчуванням для цих констант визначаються функцією "HoltWinters()", яка використовує оцінку максимальної правдоподібності для оптимізації моделі. Це передбачає пошук значень констант, які максимізують вірогідність спостережуваних даних, враховуючи припущення моделі про компоненти часового ряду та їхні відповідні параметри згладжування.

Отже, комплексний аналіз даних про використання процесорів пролив світло на поведінку процесорів у часі. Візуалізації дозволяють виявити тенденції та закономірності, які можуть бути використані для прийняття рішень, спрямованих на оптимізацію продуктивності системи та діагностику потенційних проблем. Використовуючи інформацію, отриману на основі згенерованих графіків, дослідники та аналітики можуть вживати цілеспрямованих заходів для підвищення загальної ефективності системи та забезпечення більш ефективного управління робочим навантаженням.

4 Висновки

Основні результати нашого дослідження підкреслюють складну динаміку продуктивності веб-додатків в рамках монолітної архітектури. Математичні моделі, розроблені в ході нашого дослідження, показали, що методи Холта-Вінтерса та ARIMA ефективні для згладжування використання процесора, демонструючи чіткі закономірності при різних профілях навантаження. Порівняльний аналіз показників продуктивності до і після застосування цих математичних методів виявив помітну різницю в загальній продуктивності з послідовною тенденцією до покращення використання процесора.

Наслідки цих висновків поширюються на ширший дискурс навколо оптимізації продуктивності веб-додатків. Наше дослідження демонструє, що

математичні методи можуть бути ефективно використані для оптимізації використання процесора, тим самим підвищуючи загальну продуктивність монолітних веб-додатків. Таким чином, ці висновки є внеском у триваючу дискусію про переваги та недоліки монолітної та мікросервісної архітектур.

Наш методологічний підхід поєднував методи збору даних зі складним математичним моделюванням. Хоча результати цього дослідження підтверджують ефективність цих методів, важливо враховувати потенційні обмеження. Методи Холта-Вінтерса та ARIMA, хоча і були успішними в нашому дослідженні, не можуть бути універсально застосовні до всіх архітектур або сценаріїв веб-додатків. Тому важливо підкреслити, що ці методи слід адаптувати до конкретних контекстів і профілів навантаження.

Ґрунтуючись на результатах дослідження, пропоную наступні напрямки для подальших досліджень:

1. Розширити сферу нашого дослідження, включивши до неї широкий спектр веб-додатків, щоб визначити, якою мірою наші висновки можуть бути екстрапольовані на інші контексти.
2. Ретельне вивчення застосовності наших ідей до архітектурних парадигм мікросервісів у світлі зростаючої популярності цього підходу в сучасній практиці веб-розробки.
3. Розробка адаптивного алгоритму, який об'єднує переваги різних підходів до прогнозування, тим самим даючи більш стійке рішення для оптимізації продуктивності веб-додатків.

Підсумовуючи, цей дипломний проект свідчить про потенціал використання математичних підходів, на прикладі методу Холта-Вінтерса, для оптимізації продуктивності монолітних веб-додатків. Дослідження є цінним доповненням до сукупності знань, що стосуються аналізу продуктивності веб-додатків, і закладає основу для подальших досліджень у цій галузі.

На закінчення, це дослідження зробило значний внесок у наше розуміння продуктивності веб-додатків в рамках монолітної архітектури. Застосування

методів Холта-Вінтерса та ARIMA пролило світло на потенційні шляхи оптимізації продуктивності за допомогою математичного моделювання. Однак нюанси динаміки експлуатаційних характеристик, що лежать в основі цих висновків, свідчать про необхідність подальших досліджень для повної реалізації потенціалу цих методів у різних контекстах, сценаріях та архітектурних парадигмах. Таким чином, це дослідження є важливим кроком вперед у математичному розумінні продуктивності веб-додатків і закладає основу для майбутніх досліджень у цій галузі.

Список використанних джерел

1. Vinge, V. (2010). Rainbows End: A Novel with One Foot in the Future. Tor Books.
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-Oriented Software Architecture: A System of Patterns (Vol. 1). John Wiley & Sons.
3. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc.
4. Fowler, M. (2018). MonolithFirst. Retrieved from https://martinfowler.com/bliki/MonolithFirst.html
5. Nielsen, J. (1993). Usability engineering. Boston: Academic Press.
6. Jain, R. (1991). The art of computer systems performance analysis. New York: Wiley.
7. Lounis, H., Debbabi, M., & Mouheb, D. (2013). Measuring software system's reliability and availability through user's perspective. Journal of Software Engineering and Applications, 6(5), 232-244.
8. Ramakrishnan, C., Yang, D., & Li, W. (2007). Measuring and modeling the availability of peer-to-peer file sharing systems. In Proceedings of the 17th international workshop on Network and operating systems support for digital audio and video (pp. 151-156).
9. Menascé, D. A. (2002). Capacity planning and performance modeling: from mainframes to client-server systems. Prentice Hall.
10. Menascé, D. A., Almeida, V. A. F., & Dowdy, L. W. (2002). Capacity planning and performance modeling: from mainframes to client-server systems. Prentice Hall.
11. The Software Architecture Chronicles: Monolithic Architecture (https://sookocheff.com/post/architecture/monolithic-architecture/)
12. Zhang, X., Liu, X., & Xu, Y. (2020). A microservices architecture for application systems integration. Journal of Parallel and Distributed Computing, 137, 110-123.
13. Gartner, "Application Architecture Trends in 2021" (https://www.gartner.com/smarterwithgartner/application-architecture-trends-in-2021/)
14. Zhu, X., Xu, X., & Wu, L. (2018). Microservices vs. monolithic architecture: An empirical study on performance and maintainability. In 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 71-78). IEEE.
15. J. K. Agrawal, V. K. Agrawal, "Load testing of web applications: A review," Journal of Network and Computer Applications, vol. 36, no. 2, pp. 697-713, 2013.

16. M. S. Ali, S. Al-Khalifa, "A survey on web application load testing," International Journal of Computer Science Issues, vol. 7, no. 6, pp. 25-32, 2010.
17. S. Qureshi, A. S. R. Bhatti, "Traffic patterns in web applications," in Proceedings of the International Conference on Advances in Computing, Communication and Control, 2014, pp. 238-243.
18. L. A. Barroso, U. Hölzle, "The case for energy-proportional computing," Computer, vol. 40, no. 12, pp. 33-37, 2007.
19. R. K. Bhatia, S. Kaur, "Load testing of web applications: A systematic review," Journal of Computer Science and Technology, vol. 17, no. 4, pp. 400-408, 2017.
20. SPECweb2009, "SPECweb2009 benchmark," SPEC, 2016. [Online]. Available: https://www.spec.org/web2009/. [Accessed: May 2, 2023]
21. A. Bondi, "Characteristics of scalability and their impact on performance," in Proceedings of the International Workshop on Software and Performance, 2000, pp. 195-203.
22. H. Huang, S. Li, Z. Li, Y. Zhang, "Design and implementation of a web application load testing system based on a hybrid approach," Future Generation Computer Systems, vol. 100, pp. 860-870, 2019.
23. X. Zhou, W. Lu, W. Sun, J. Yan, "Modeling web application workload for load testing: A literature review," Journal of Network and Computer Applications, vol. 59, pp. 205-219, 2016.
24. S. Li, Z. Li, H. Huang, Y. Zhang, "Workload model validation in web application load testing," Journal of Systems and Software, vol. 156, pp. 1-11, 2019.

Додатки

Додаток А

Класична тришарова архітектура зазвичай складається з трьох рівнів: рівня представлення, рівня бізнес-логіки та рівня зберігання даних. У цій архітектурі додаток розділений на три окремі шари, і кожен з них відповідає за певні завдання. Аналіз продуктивності в цій архітектурі передбачає аналіз продуктивності кожного рівня окремо та їх оптимізацію для підвищення загальної продуктивності системи. Вузькі місця в продуктивності зазвичай виявляються шляхом моніторингу часу відгуку та використання ресурсів на кожному рівні.

З іншого боку, архітектура мікросервісів - це підхід до розподіленої архітектури, коли додаток складається з невеликих незалежних сервісів, які взаємодіють один з одним за допомогою легких протоколів, таких як HTTP або системи обміну повідомленнями. Кожен мікросервіс фокусується на конкретній бізнес-можливості і може бути розроблений, розгорнутий і масштабований незалежно. Аналіз продуктивності в архітектурі мікросервісів передбачає аналіз продуктивності окремих мікросервісів, а також взаємодій і залежностей між ними. Він вимагає моніторингу часу відгуку та використання ресурсів кожного мікросервісу, а також врахування впливу міжсервісної взаємодії на продуктивність.

Таким чином, класична 3-рівнева архітектура зосереджена на оптимізації окремих рівнів, тоді як архітектура мікросервісів вимагає аналізу та оптимізації продуктивності окремих сервісів та їх взаємодії. Архітектура мікросервісів забезпечує більшу гнучкість і масштабованість, але також вносить додаткову складність з точки зору аналізу продуктивності та управління через розподілену природу системи.

Таблиці

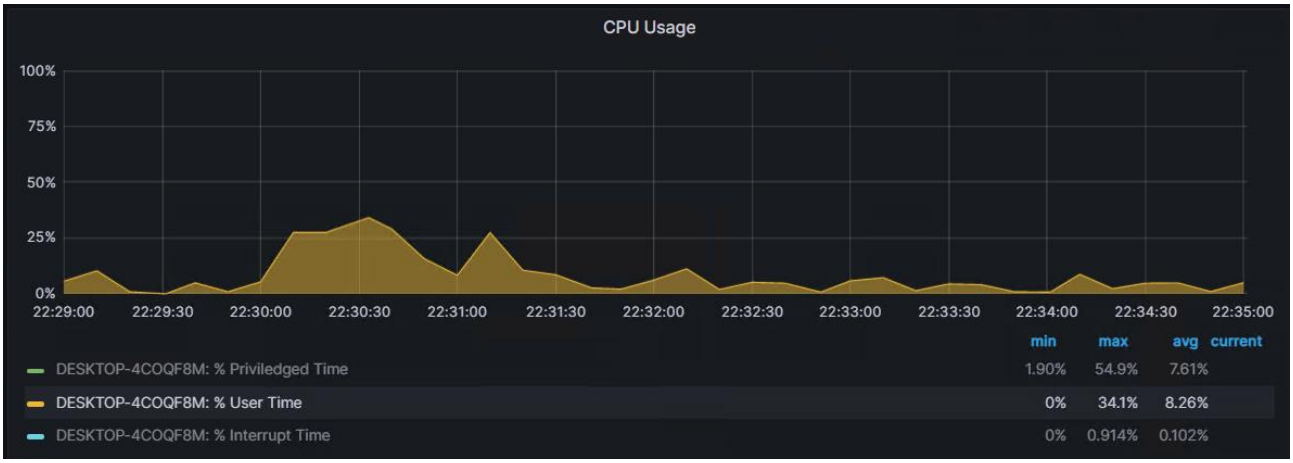
Приклад даних з CSV файлу

Test Interval	First Processor	Second Processor	Third Processor	Fourth Processor
0	5.9	10	7.3	11.4
10	0.8	0.3	0.8	0.8
21	1.5	0	4.5	1.5
30	2.6	4.2	4	5.9
40	1.1	0.5	1.2	0.9
50	3.6	5	5.2	5.9
60	5.8	6.4	5	11.3
70	1.6	0.6	1.1	1.6
80	2.5	3.4	3.4	8.3
91	4.5	0	1.5	1.5
100	0.5	0.3	1.4	1.4
110	8.7	3.9	5.5	8.3
120	8.1	10.8	9.2	12.8
130	1.2	0.3	1.6	1.6
140	3.3	5.5	4.5	4.2
150	3.4	5.3	4.1	5.3
161	3.1	0	0	1.5
170	4.7	5.3	6.3	7.5
180	10.6	10	8.1	12
190	1.4	0.5	1.3	0.9
200	4.8	4.5	3	7.8
210	7.4	3.9	4.5	7.8
220	8.6	8.6	6.7	18.4
231	1.5	1.5	1.5	0
240	6.8	9.6	10.5	9.8
250	2.5	2	2.5	6.2
260	2.3	4.7	4.8	5.8
270	3.6	3.7	4.5	6.4
280	0.8	0.6	1.1	0.5
290	3.9	5.6	4.7	4.5
301	0	1.5	0	0
310	1.4	0.2	1.4	1.7
320	3.3	3	2.3	7.8
330	4.5	2	3.9	6.9
340	1.4	0.2	1.7	0.2

350	5.3	5.6	4.5	6.6
360	5.9	11.6	7.3	10.6
371	16.8	19.9	16.8	30.6
380	3.1	1.6	3.3	3.3
390	3.4	4.4	3.9	6.2
400	1.1	0.3	0.9	1.1
410	5.6	5.6	5.8	5.8
420	8	9.1	9.5	14.7
430	1.2	0.6	1.1	0.8
441	0	0	0	0
450	3.5	4.2	4	8
460	0.9	0.3	1.9	0.6
470	7	4.2	4.4	5.9
480	18.6	39.1	26.3	26.1

Графіки

Показники CPU взяті з Grafana



Графік середнього CPU по всіх 4 ядрах по часовому ряду.

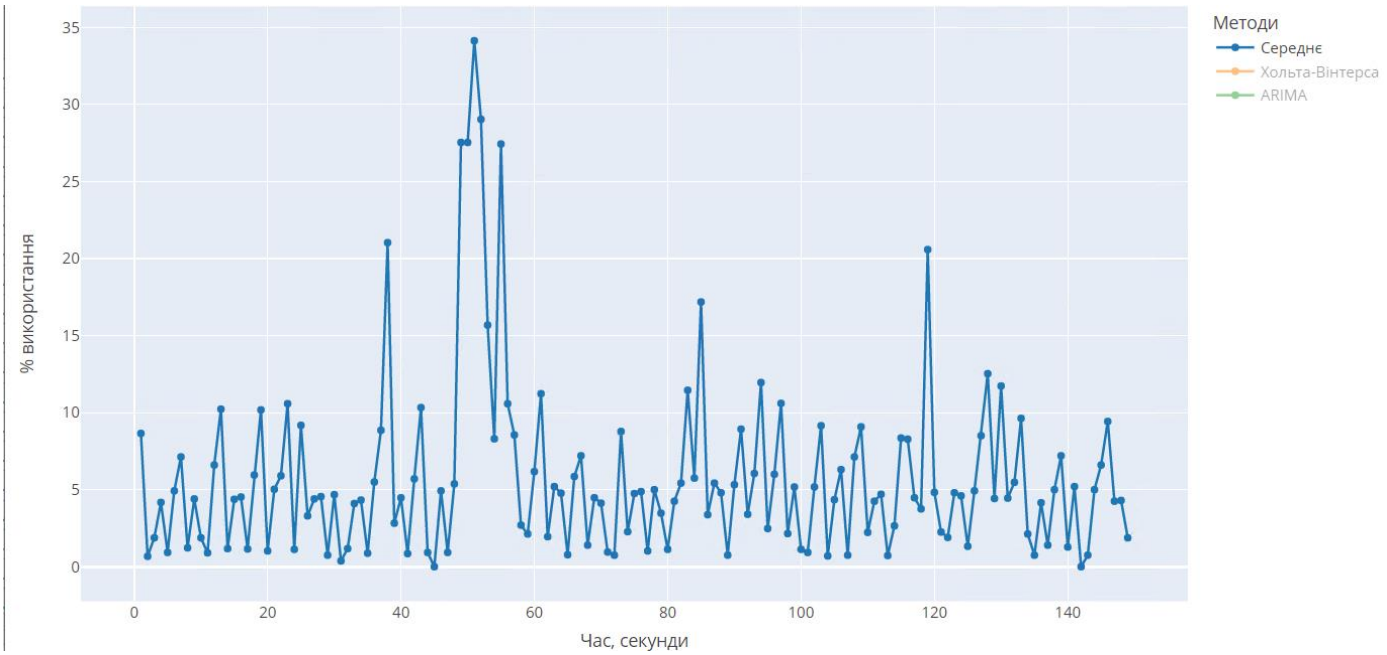


Рис. 2

Графік показників CPU по кожному з процесорів.

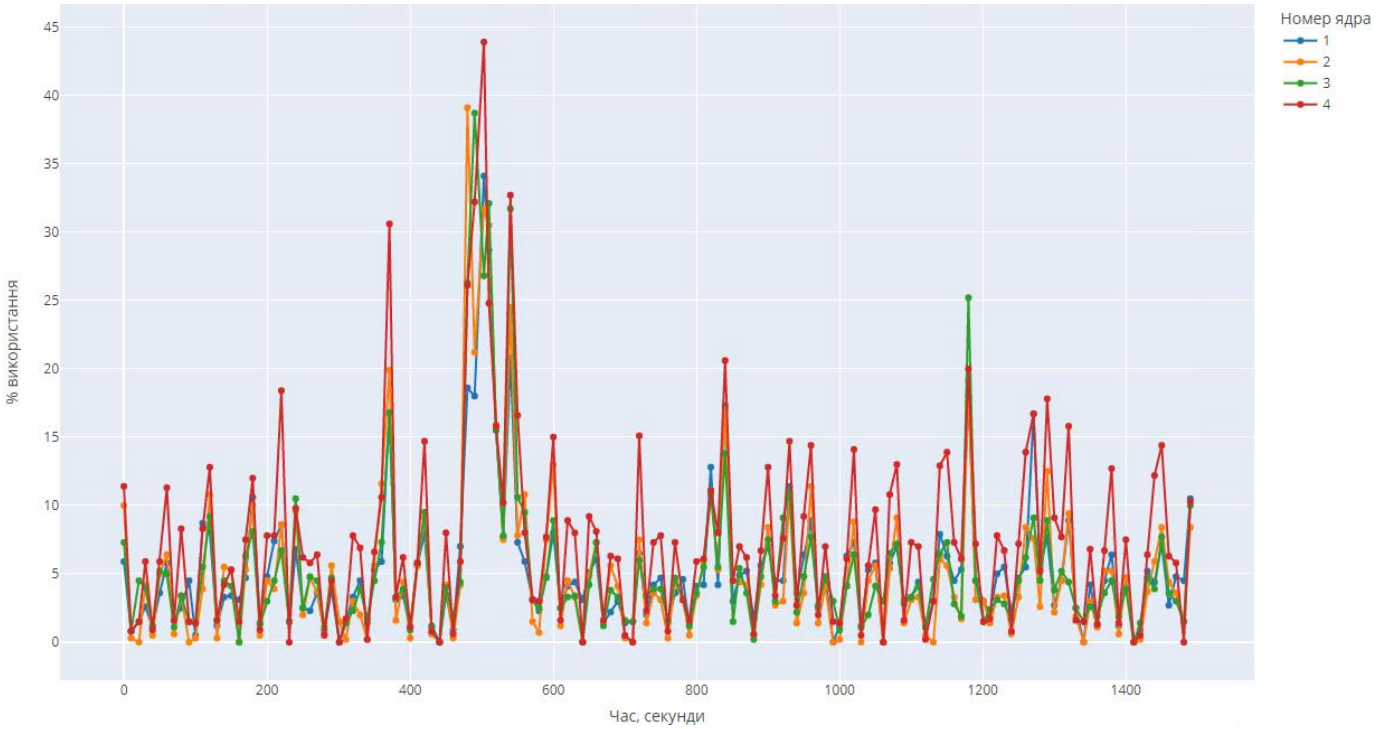


Рис. 3

Графік середнього CPU по всіх 4 ядрах по часовому ряду та згладжених показників методом Хольта-Вінтерса.

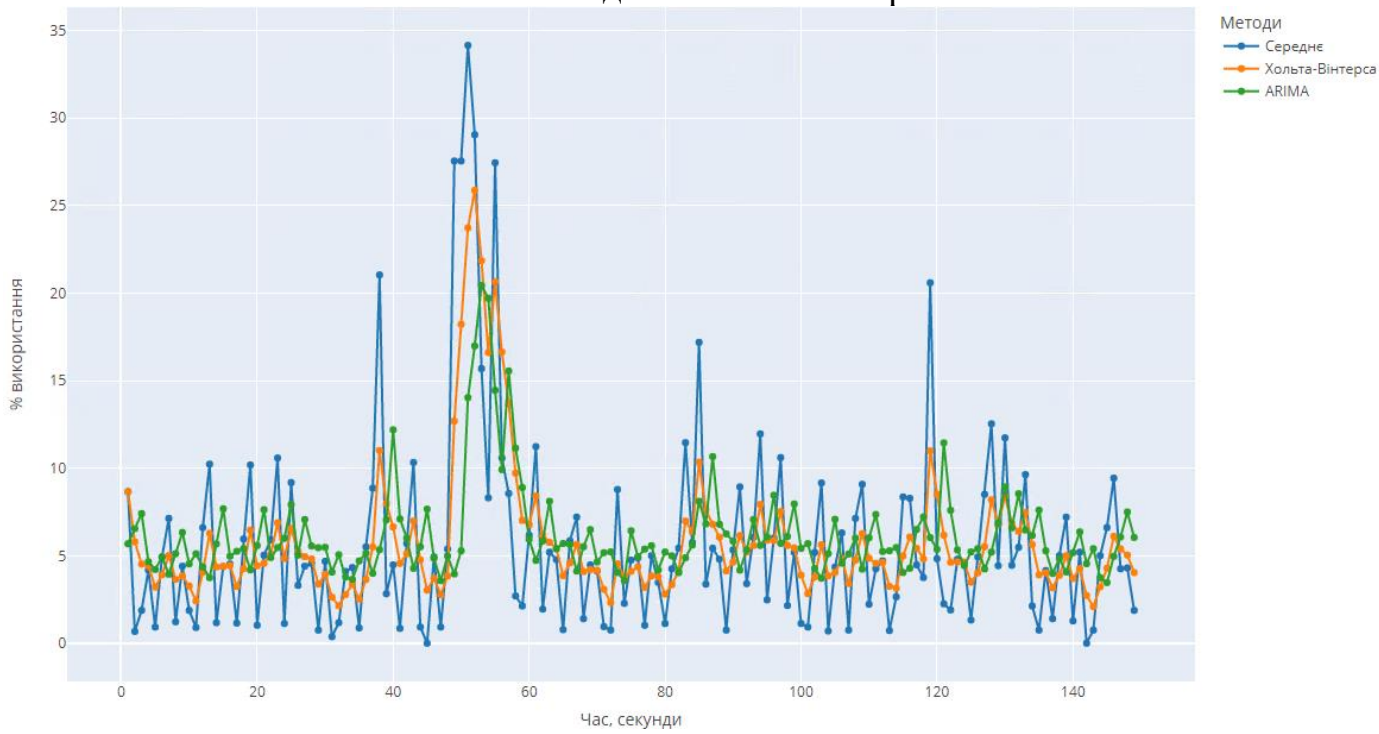


Рис. 4

Хольт-Вінтерс, Аріма, Середнє

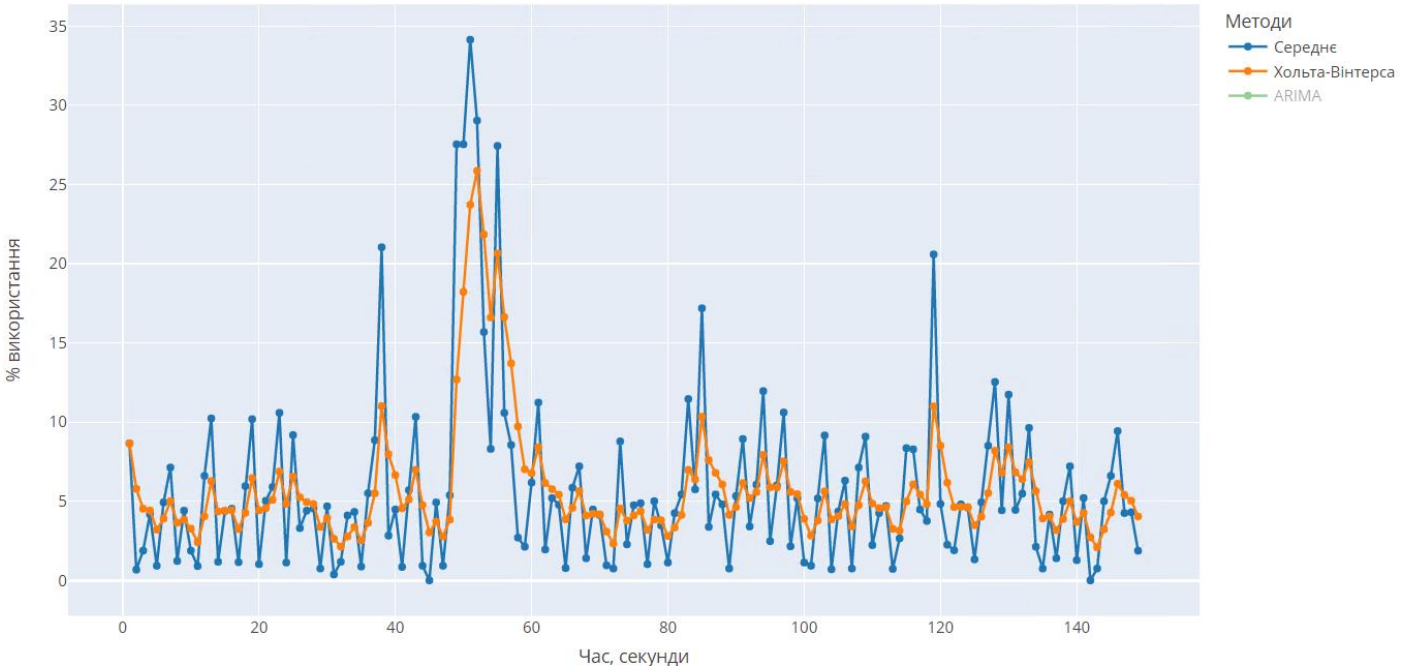


Рис. 5

Графік середнього CPU показників методом Хольта-Вінтерса.

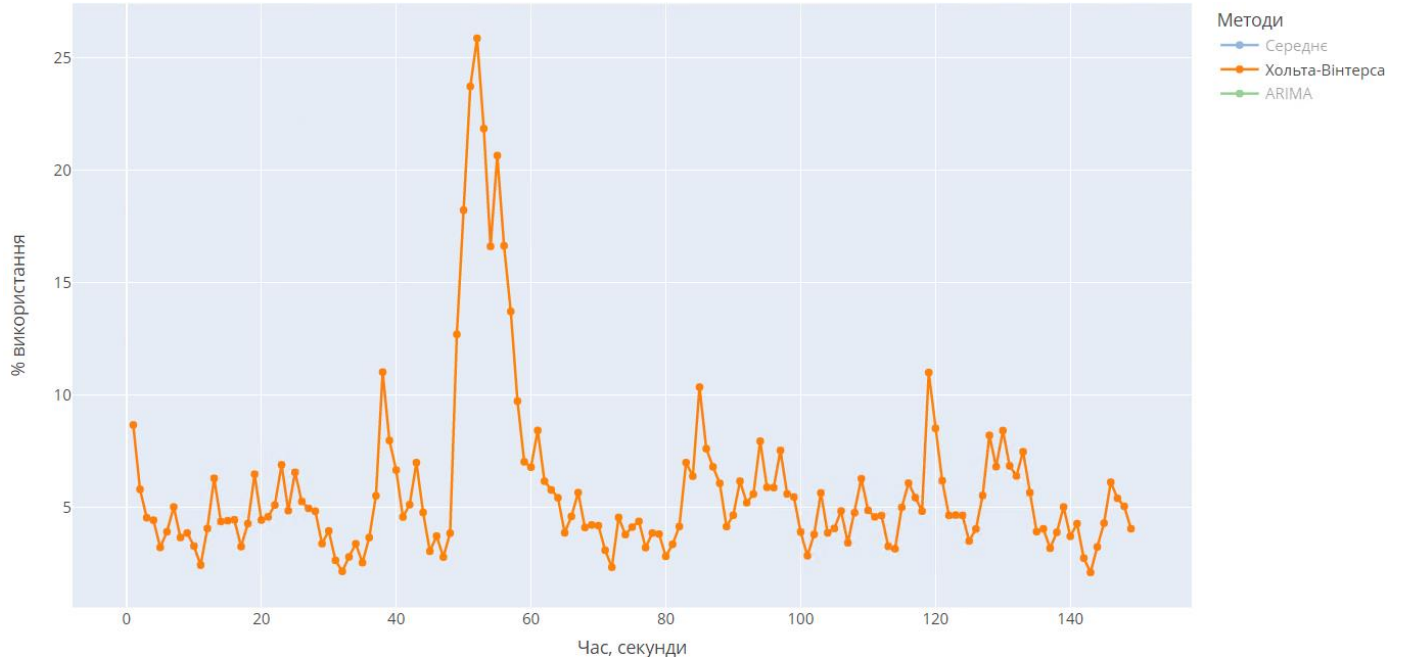


Рис. 6

ARIMA

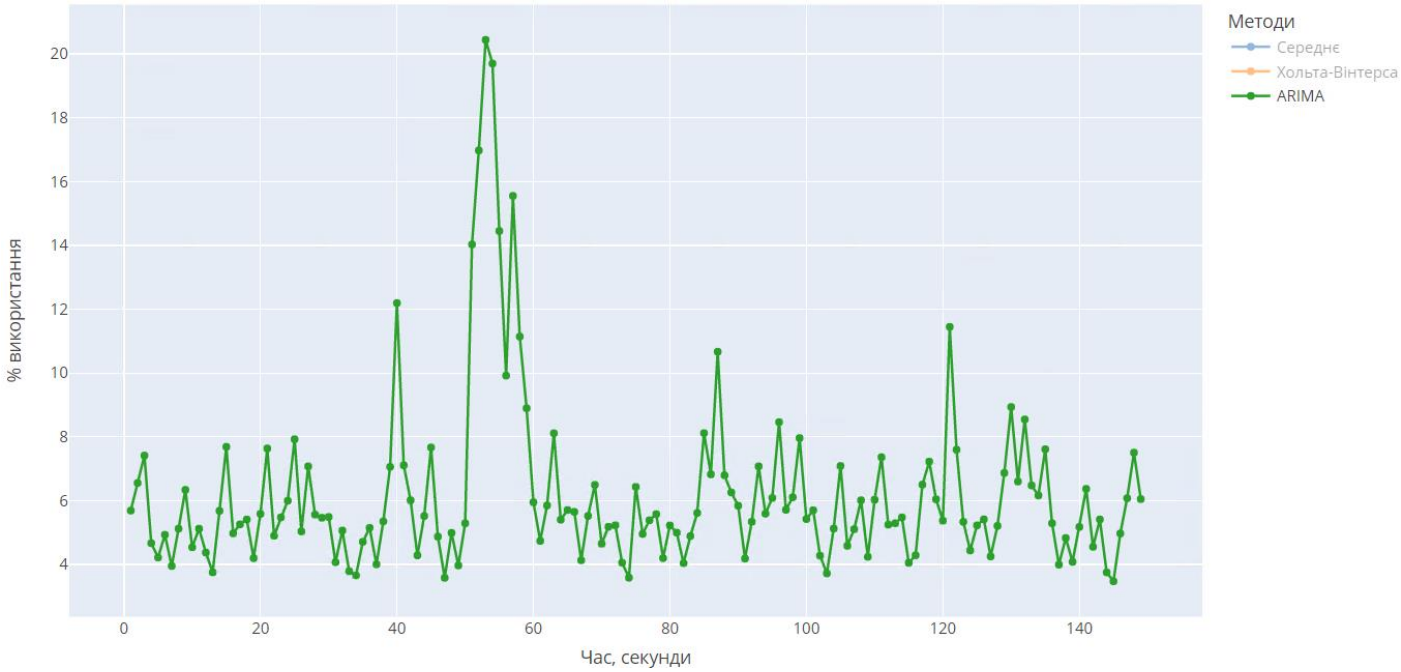


Рис. 7

Хольта-Вінтерса та ARIMA

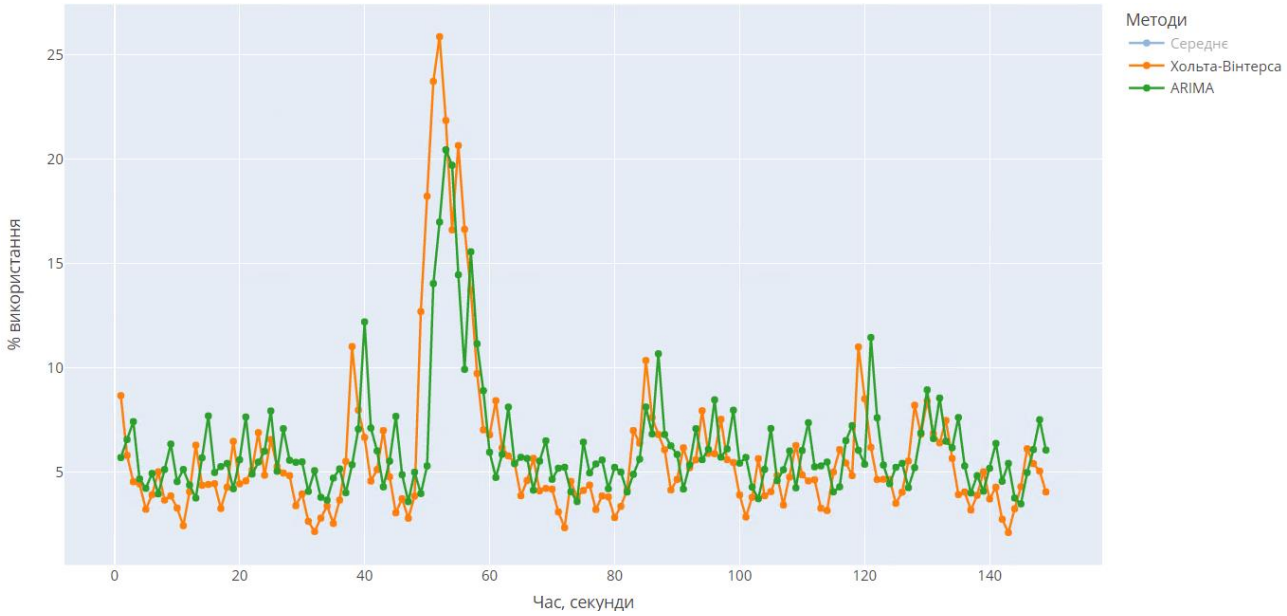


Рис. 8

Код

Додаток В

```
#Load required libraries
library(plotly)
library(dplyr)
library(forecast)

#Read .csv file with CPU performance data for each processor and timeline
ProcessorsCpuIndicatorsByTime <- read.csv(file = "cpuData.csv", sep = ",", header = TRUE)

#Calculate average value of CPU for each record (row)
ProcessorsCpuIndicatorsByTime$Xmean <- (ProcessorsCpuIndicatorsByTime$X0 + ProcessorsCpuIndicatorsByTime$X1 + ProcessorsCpuIndicatorsByTime$X2 + ProcessorsCpuIndicatorsByTime$X3)/4

#Draw a plot for average CPU usage
fig <- plot_ly(x = ~ProcessorsCpuIndicatorsByTime$N, y = ~ProcessorsCpuIndicatorsByTime$Xmean, name = 'trace 0', type = 'scatter', mode = 'lines+markers')
fig

#Draw a plot for CPU usage for each processor
fig <- plot_ly(x = ~ProcessorsCpuIndicatorsByTime$N, y = ~ProcessorsCpuIndicatorsByTime$X3, name = 'processor 4', type = 'scatter', mode = 'lines+markers')
fig <- fig %>% add_trace(y = ~ProcessorsCpuIndicatorsByTime$X0, name = 'processor 1', mode = 'lines+markers')
fig <- fig %>% add_trace(y = ~ProcessorsCpuIndicatorsByTime$X1, name = 'processor 2', mode = 'lines+markers')
```

```
fig <- fig %>% add_trace(y = ~ProcessorsCpuIndicatorsByTime$X2, name = 'processor 3', mode = 'lines+markers')
fig

#Prepare the time series data
timeSeries <- ProcessorsCpuIndicatorsByTime$X0 + 0.01
timeSeries <- append(timeSeries, ProcessorsCpuIndicatorsByTime$X1 + 0.01)
timeSeries <- append(timeSeries, ProcessorsCpuIndicatorsByTime$X2 + 0.01)
timeSeries <- append(timeSeries, ProcessorsCpuIndicatorsByTime$X3 + 0.01)
timeSeries <- ts(matrix(timeSeries, 150, 4), start = c(1,1), frequency = 1)
timeSeriesMean <- ts(ProcessorsCpuIndicatorsByTime$Xmean + 0.01, start = c(1,1), frequency = 1)
timeSeriesMean <- head(as.numeric(timeSeriesMean), -1)

#Apply Holt-Winters forecasting method for each processor
a <- HoltWinters(timeSeries[,1], beta=FALSE, gamma=FALSE)
b <- HoltWinters(timeSeries[,2], beta=FALSE, gamma=FALSE)
c <- HoltWinters(timeSeries[,3], beta=FALSE, gamma=FALSE)
d <- HoltWinters(timeSeries[,4], beta=FALSE, gamma=FALSE)

#Calculate the average of fitted values
final <- (as.numeric(a$fitted[,1])+as.numeric(b$fitted[,1])+as.numeric(c$fitted[,1])+as.numeric(d$fitted[,1]))/4
```

```
#Calculate ARIMA
arima_forecast <- forecast(ProcessorsCpuIndicatorsByTime$Xmean, h = length(ProcessorsCpuIndicatorsByTime$Xmean))
arima_result <- head(arima_forecast$fitted, -1)

#Create a data frame with fitted values
data <- data.frame(a= as.numeric(a$fitted[,1]), b = as.numeric(b$fitted[,1]))
data$c = as.numeric(c$fitted[,1])
data$d = as.numeric(d$fitted[,1])
data$median <- apply(data[,c("a","b","c","d")], 1, quantile, probs = 0.5)

#Draw a plot with median, average and initial values
fig <- plot_ly(x = 1:149, y = final, name = 'HoltWinters', type = 'scatter', mode = 'lines+markers')
fig <- fig %>% add_trace(y = timeSeriesMean, name = 'CPU', mode = 'lines+markers')
fig <- fig %>% add_trace(y = arima_result, name = 'ARIMA', mode = 'lines+markers')
fig
```