

ПОРІВНЯННЯ ПРОДУКТИВНОСТІ КОНТЕЙНЕРИЗАЦІЇ З КЛАСИЧНИМ МЕТОДОМ ЗАПУСКУ ПРОГРАМ

Вінницький національний технічний університет

Анотація

Досліджено вплив контейнеризації на продуктивність програм у серверному середовищі, проведено заміри використання ресурсів ідентичних контейнеризованих та неконтейнеризованих застосунків.

Ключові слова: операційні системи, контейнеризація, інтернет, програмне забезпечення, Linux

Abstract

Research on influence of containerization on application performance, measuring of resource usage of identical containerized and non-containerized applications

Keywords: operationg systems, containerization, internet, software, Linux.

У сучасній IT-інфраструктурі контейнеризація стала ключовою технологією для розгортання програмного забезпечення. Вона пропонує легкі, ізольовані середовища, що спрощують управління залежностями та масштабування. Однак класичний метод запуску програм, заснований на безпосередньому виконанні додатків у середовищі операційної системи, досі залишається поширеним, особливо в традиційних архітектурах.

Метою цієї роботи є визначення оптимальності використання контейнеризації у різних сценаріях розгортки програмного забезпечення.

Для оцінки продуктивності контейнеризації порівняно з класичним методом запуску програм було створено контрольоване тестове середовище, яке забезпечує відтворюваність та коректність результатів.

В якості базової ОС використовується Linux NixOS 23.05 [1] з ядром 6.1. NixOS було обрано через її декларативний підхід до конфігурації, що гарантує чистоту експерименту та усуває вплив сторонніх факторів, таких як неконтрольовані оновлення або зміни в залежностях системи.

Тестування виконується у віртуальній машині QEMU [2], що працює на Intel Xeon 2.1 ГГц (2 ядра) з 4 ГБ ОЗП. Вибір QEMU дозволяє отримати стабільне та ізольоване середовище, що мінімізує вплив апаратних особливостей хост-системи та забезпечує рівні умови для обох методів запуску.

Для тестування контейнеризованого підходу використовується systemd-nspawn[3], інструмент для запуску ізольованих контейнерів у середовищі systemd. Він забезпечує легку віртуалізацію, створюючи контрольоване оточення для програм, подібне до chroot, але з додатковою підтримкою розмежування ресурсів та безпеки. Це дозволяє оцінити його продуктивність у порівнянні з класичним запуском без контейнерів.

В ролі тестового застосунку використовується Apache HTTP Server (httpd) [4] — один із найпопулярніших веб-серверів, що широко застосовується в реальних робочих навантаженнях. Його продуктивність добре вивчена, а велика кількість параметрів налаштувань дозволяє детально аналізувати поведінку як у контейнеризованому середовищі, так і при класичному запуску.

Для генерації трафіку використовується wrk [5] — високопродуктивний генератор HTTP-запитів, що дозволяє моделювати реальні навантаження та вимірювати такі показники, як пропускна

здатність, середній час відповіді та використання ресурсів. Використання `wrk` забезпечує високу точність результатів, оскільки він мінімізує вплив власного виконання на тестоване середовище.

Ця тестова конфігурація дозволяє об'єктивно оцінити продуктивність контейнеризації в порівнянні з класичним підходом запуску, враховуючи такі ключові метрики, як швидкість старту, використання процесорного часу, навантаження на пам'ять і стабільність роботи під тиском.

Для вимірювання використання ресурсів системи під час тестів застосовується команда `top` [6], яка з періодичністю раз на секунду записує поточний стан системи у текстовий файл.

```
top -b -d1 >> benchmark.txt
```

Ця команда виконує моніторинг системних ресурсів і записує отримані дані у файл `container_benchmark.txt`. Розглянемо її компоненти:

- `top -b` – запускає утиліту `top` у фоновому (batch) режимі, що дозволяє записувати її вивід у файл без інтерактивного режиму.
- `-d1` – встановлює інтервал оновлення у 1 секунду, тобто `top` буде записувати стан системи щосекунди.
- `>> benchmark.txt` – додає зібрані дані у файл `benchmark.txt` для подальшого аналізу.

Це дозволяє відстежувати навантаження на систему в режимі реального часу як при запуску Apache HTTP Server у класичному режимі, так і в контейнеризованому середовищі (`systemd-nspawn`), оскільки при використанні контейнеризації процеси запускаються напряму на хості.

Контейнер Apache httpd описаний наступною функцією мови Nix:

```
{config, ...} : {  
  config.virtualisation.oci-containers.containers = {  
    httpd = {  
      image = "httpd";  
      ports = ["0.0.0.0:80:80"];  
    };  
  };  
}
```

Для початку заміряємо навантаження на систему без запущеного httpd:

```
top - 18:08:11 up 1 min,  1 user,  load average: 0.21, 0.13, 0.05  
Tasks: 104 total,   1 running, 103 sleeping,   0 stopped,   0 zombie  
%Cpu(s):  3.0 us,  3.0 sy,   0.0 ni, 93.9 id,   0.0 wa,   0.0 hi,   0.0 si,   0.0 st  
MiB Mem :  3835.7 total,  3588.7 free,   125.1 used,  121.9 buff/cache  
MiB Swap:  1917.5 total,  1917.5 free,    0.0 used.  3525.1 avail Mem
```

Рис. 1. Навантаження на тестову віртуальну машину без запущеного httpd.

З рис. 1 бачимо використання 125 MiB оперативної пам'яті, 3% процесору.

По черзі запустимо `httpd` без контейнеризації та з контейнеризацією. За навантаження в 680 запитів в секунду заміряємо використання процесору. На основі отриманих даних згенеруємо графік за допомогою бібліотеки мови програмування Python `matplotlib` [7]. Графік використання подано на рисунку 2.

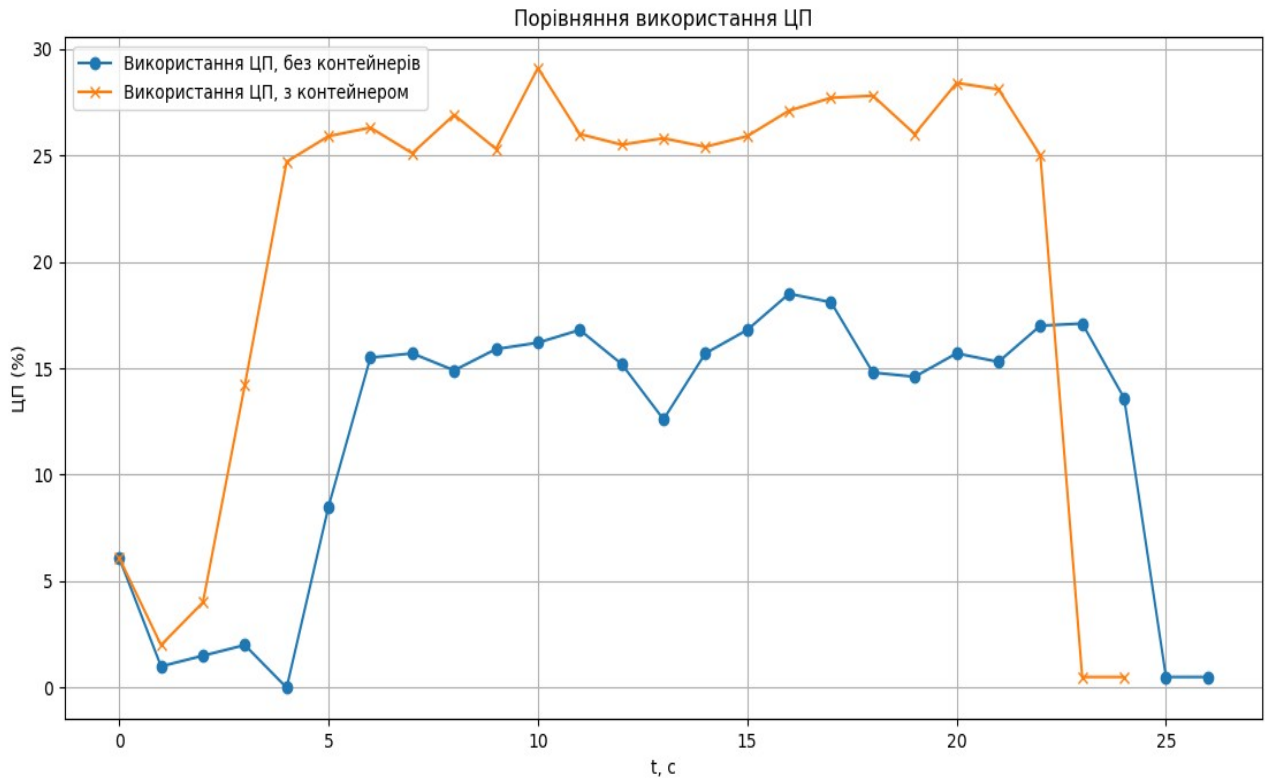


Рисунок 2. Графік використання ЦП віртуальної машини з увімкненою та ввимкненою контейнеризацією.

Проведемо подальший аналіз отриманих даних для визначення джерела росту використання ЦП у сценарії з контейнеризацією. Графік використання ЦП окремими процесами (групами процесів) подано на рисунку 3.

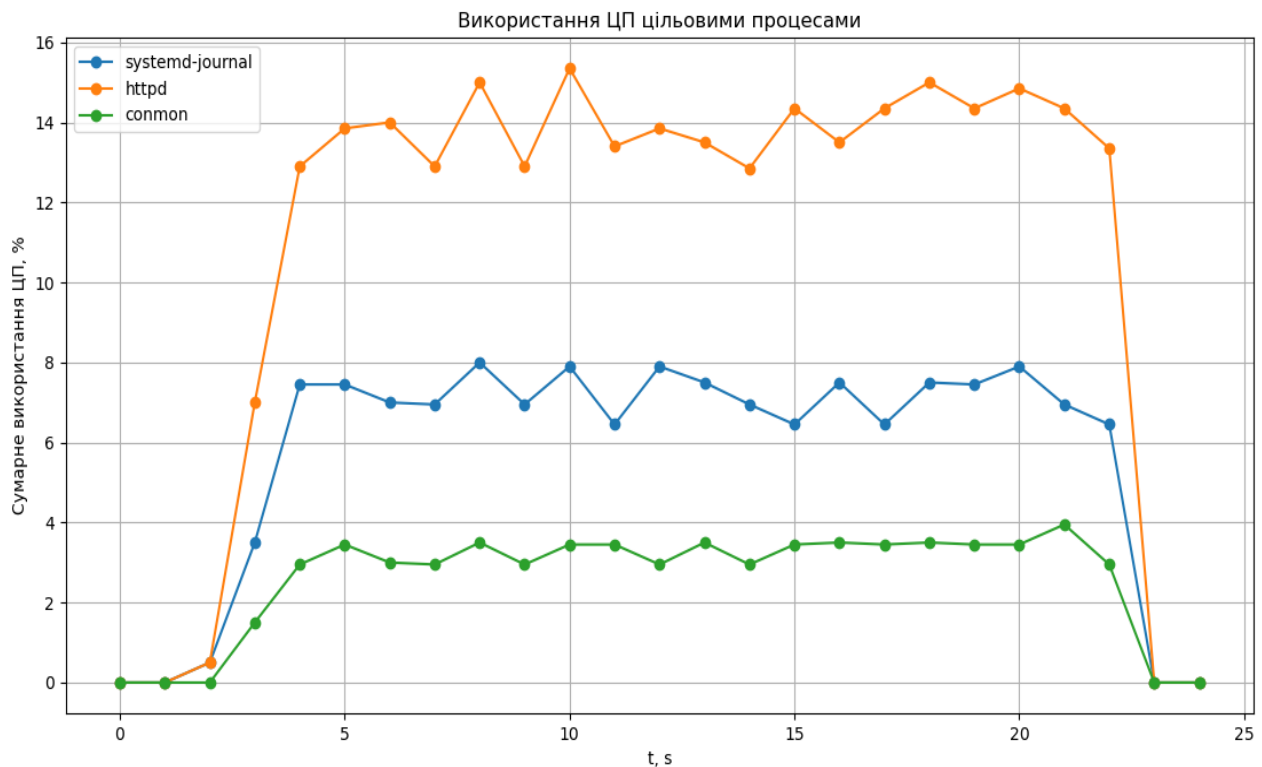


Рисунок 3. Графік використання ЦП процесами, пов'язаними з контейнеризацією.

Результати тестування показали, що запуск Apache HTTP Server (httpd) у класичному режимі споживає 20% процесорного часу, тоді як у контейнеризованому середовищі (systemd-nspawn) це

значення зростає до 30%. Таким чином, використання контейнеризації додає 10% до витрат ЦП. Це зростання можна пояснити накладними витратами контейнеризації, які включають ізоляцію ресурсів через cgroups та namespaces, віртуалізацію файлової системи, що може впливати на час доступу до файлів, а також додаткові виклики ядра, пов'язані з мережевою ізоляцією контейнера.

Важливо, що ці 10% є абсолютним збільшенням, а не пропорційним до загального навантаження. Це означає, що для програм із низькими обчислювальними вимогами витрати на контейнеризацію будуть відчутнішими. Наприклад, якщо вихідне навантаження складає 20% ЦП, то додаткові 10% контейнеризації становлять значну частку (50% від початкового споживання). У таких випадках, особливо в умовах обмежених ресурсів, накладні витрати можуть суттєво впливати на загальну продуктивність. Водночас, якщо програма споживає значну частину процесорного часу (наприклад, 80%), додаткові 10% будуть менш відчутними відносно загального навантаження. Це вказує на те, що контейнеризація може бути неефективною для малих програм або середовищ із низькою доступністю ресурсів, таких як вбудовані системи, невеликі сервери або хмарні середовища з жорсткими обмеженнями. У таких випадках класичний запуск без контейнеризації може забезпечити більш ефективне використання обчислювальних потужностей. Отже, при виборі між контейнеризованим і класичним способом виконання програм слід враховувати не лише зручність розгортання та ізоляцію процесів, але й потенційний вплив на продуктивність, особливо якщо ресурси обмежені або програма має низьке базове навантаження.

Однак, все ще варто згадати, що попри додаткові витрати на процесорний час, контейнеризація має суттєві переваги в порівнянні з класичним методом запуску програм. Вона забезпечує ізоляцію процесів, що дозволяє уникати конфліктів між залежностями та створює передбачуване середовище для виконання програм. Це особливо важливо для розгортання на різних системах, оскільки контейнер містить усі необхідні бібліотеки та конфігурації, що знижує ризик проблем, пов'язаних із версіями пакетів або особливостями конкретної операційної системи.

Контейнери можна легко створювати, масштабувати та оновлювати без ризику порушити роботу всієї системи. У порівнянні з класичним методом, де оновлення або перенесення програм вимагає змін у базовій системі, контейнеризовані програми працюють у відокремленому середовищі, що робить їх більш гнучкими. Це особливо корисно в DevOps-підході, де часті оновлення та автоматизована розгортка є ключовими елементами ефективної розробки. Контейнери також спрощують масштабування додатків. У класичному середовищі запуск декількох екземплярів програми може потребувати додаткової конфігурації, тоді як у контейнеризованій архітектурі процеси ізолювані, і розгортання нових екземплярів стає простішим. Це важливо для високонавантажених сервісів, де швидке масштабування дозволяє адаптуватися до змін у трафіку без значних змін у системній архітектурі.

Ще одним значним плюсом є безпека. Контейнери працюють у власному ізолюваному просторі, що знижує ризики, пов'язані з компрометацією однієї програми. У класичному методі зломисник, отримавши доступ до одного сервісу, може вплинути на всю систему, тоді як у контейнеризованому середовищі ізоляція допомагає обмежити потенційні загрози.

Таким чином, хоча контейнеризація створює певні накладні витрати на ресурси, вона дає значні переваги в управлінні, масштабуванні, безпеці та переносимості програм. Вибір між контейнеризованим та класичним методом залежить від конкретного сценарію використання: якщо важлива ефективність використання ресурсів, традиційний запуск може бути кращим вибором, але якщо пріоритет – зручність розгортання, ізоляція та автоматизація, контейнеризація має очевидні переваги.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. EELCO DOLSTRA. ANDRES LOH, NICOLAS PIERRON. NixOS: A Purely Functional Linux Distribution. Functional Programming, 2 September 2010.
2. Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. 2005 USENIX Annual Technical Conference.
3. Apache Software Foundation. Apache HTTP Server [Електронний ресурс] / URL: https://httpd.apache.org/docs/trunk/new_features_2_4.html (дата звернення: 23.03.2025).
4. ArchWiki. systemd-nspawn. [Електронний ресурс] / URL: <https://wiki.archlinux.org/title/Systemd-nspawn> (дата звернення: 23.03.2025).
5. GitHub. wrk – a HTTP benchmarking tool [Електронний ресурс] / URL: <https://github.com/wg/wrk> (дата звернення: 23.03.2025).
6. Linux Manual Pages. top(1) – display Linux tasks. URL: <https://man7.org/linux/man-pages/man1/top.1.html> (дата звернення: 23.03.2025).
7. Matplotlib: Visualization with Python [Електронний ресурс] URL: <https://matplotlib.org/> (дата звернення: 23.03.2025).

Янковський Богдан Петрович — студент групи 2КН-226, факультет інтелектуальних інформаційних технологій та автоматизації, Вінницький національний технічний університет, Вінниця, e-mail: bohdan.yankovskyi@gmail.com

Томчук Микола Антонович — доцент кафедри безпеки життєдіяльності та педагогіки безпеки, Вінницький національний технічний університет, Вінниця, e-mail: tomchuk@vntu.edu.ua

Малініч Ілля Павлович — асистент кафедри комп'ютерних наук, Вінницький національний технічний університет, м. Вінниця, e-mail: malinich@vntu.edu.ua

Yankovskyi Bohdan P. — Faculty of intellectual informational technology and automatisaton, Vinnytsia National Technical University, Vinnytsia, email: bohdan.yankovskyi@gmail.com

Tomchyk Mykola A. — Associate Professor of the Department of Life Safety and Safety Pedagogy, Vinnytsia National Technical University, Vinnytsia, email: tomchuk@vntu.edu.ua

Malinich Iliia P. — Assistant Professor, Department of Computer Science, Vinnytsia National Technical University, Vinnytsia, email : malinich@vntu.edu.ua