UDC 004.415.2
DOI https://doi.org/10.32782/tnv-tech.2025.1.17

OPTIMIZATION OF LOCAL DEVELOPMENT PROCESS USING DOCKER PHP IMAGE THAT COMES WITH A FULL SET OF TOOLS OUT OF THE BOX – PERFORMANCE AND OPTIMIZATION EXTENSIONS

Semeniuk V. V. *– Senior Software Engineer, Illumin, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"*
ORCID ID: 0009-0007-8670-4023

The optimization of local development processes often hinges on ensuring each element of the environment is configured for both flexibility and performance. With PHP, one of the most efficient approaches to achieve such optimization is leveraging Docker images that bundle essential extensions and tools for caching, debugging, and performance tuning. A comprehensive Docker image integrating Opcache, Redis, Memcached, BCMath, and xDebug, alongside other PHP-associated tools, minimizes the overhead of environment reconfiguration and accelerates developer workflows. By using Docker Compose, developers can further streamline this process by defining services such as MySQL, Nginx, and RabbitMQ in a single configuration file, ensuring consistency across team members. This approach eliminates conflicts arising from mismatched PHP versions or extensions and reduces the time spent on manual environment setup.

Building a PHP: 7.1-fpm-based image simplifies the integration of critical libraries and extensions, such as PDO for database interaction, GD for image manipulation, mcrypt for encryption, and BCMath for precise arithmetic. Tools like Opcache and xDebug further enhance performance and debugging efficiency. Opcache, for instance, precompiles PHP scripts into bytecode, reducing the overhead of repeated parsing and compilation. Extensions like Redis and Memcached provide memory caching mechanisms, offloading repetitive operations from databases and speeding up application response times.

The integration of health checks through Dockerfile commands ensures container reliability by monitoring endpoints and recovering from failures. Custom build arguments allow optional extensions like SOAP or xDebug to be conditionally installed, tailoring the image to specific project needs. These features, combined with modular configuration options, enable flexible adaptation without rebuilding the entire image.

The consolidated Docker image offers significant advantages, including streamlined onboarding for new developers, modularity through optional extensions, and improved performance with caching mechanisms. By reducing manual setup and configuration conflicts, the image promotes a unified interface across development, staging, and production environments. The portability of Docker images supports seamless deployment across diverse systems, enhancing both security and scalability. Through these practices, Docker-based environments empower PHP developers with a robust, efficient, and reproducible toolkit tailored to modern development needs.

Key words: *caching, extensions, debugging, configuration, environment, database.*

Семенюк В. В. Оптимізація процесу локальної розробки за допомогою образу Docker PHP, який поставляється з повним набором інструментів із коробки – розширення продуктивності та оптимізації

Оптимізація процесів локального розвитку часто залежить від забезпечення гнучкості та продуктивності кожного елемента середовища. У PHP одним із найефективніших підходів до досягнення такої оптимізації є використання образів Docker, які містять основні розширення та інструменти для кешування, налагодження та налаштування продуктивності. Комплексний образ Docker, який інтегрує Opcache, Redis, Memcached, BCMath і xDebug разом з іншими інструментами, пов'язаними з PHP, мінімізує накладні витрати на зміну конфігурації середовища та прискорює робочі процеси розробників. Використовуючи Docker Compose, розробники можуть ще більше оптимізувати цей процес, визначаючи такі сервіси, як MySQL, Nginx і RabbitMQ, в одному конфігураційному файлі, забезпечуючи узгодженість між членами команди. Цей підхід усуває конфлікти, що виникають через невідповідність версій або розширень PHP, і скорочує

час, витрачений на налаштування середовища вручну. Створення образу на основі PHP: 7.1-fpm спрощує інтеграцію критичних бібліотек і розширень, таких як PDO для взаємодії з базою даних, GD для обробки зображень, tcrypt для шифрування та BCMath для точних арифметичних обчислень. Такі інструменти, як Opcache і xDebug, ще більше підвищують продуктивність і ефективність налагодження. Opcache, наприклад, попередньо компілює скрипти PHP у байт-код, зменшуючи накладні витрати на повторний аналіз і компіляцію. Такі розширення, як Redis і Memcached, забезпечують механізми кешування пам'яті, розвантажуючи повторювані операції з баз даних і прискорюючи час відповіді програми. Інтеграція перевірок працездатності за допомогою Docker-файлових команд забезпечує надійність контейнера шляхом моніторингу кінцевих точок і відновлення після збоїв. Спеціальні аргументи збірки дозволяють умовно встановлювати додаткові розширення, як-от SOAP або xDebug, адаптуючи зображення до конкретних потреб проекту. Ці функції в поєднанні з модульними параметрами конфігурації забезпечують гнучку адаптацію без перебудови всього образу. Консолідований образ Docker пропонує значні переваги, зокрема спрощену адаптацію для нових розробників, модульність завдяки додатковим розширенням і покращену продуктивність завдяки механізмам кешування. Зменшуючи конфлікти налаштувань і конфігурацій вручну, образ сприяє створенню єдиного інтерфейсу серед середовищ розробки, постановки та виробництва. Портативність образів Docker підтримує плавне розгортання в різноманітних системах, підвищуючи безпеку та масштабованість. Завдяки цим практикам середовища на основі Docker надають розробникам PHP надійний, ефективний і відтворюваний інструментарій, адаптований до сучасних потреб розробки.

Ключові слова: *кешування, розширення, дебагінг, конфігурація, середовище, база даних.*

Introduction and Problem Statement. Optimizing the local development environment is crucial for enhancing both productivity and flexibility. In the context of PHP development, achieving this balance often requires using pre-configured solutions that simplify workflows while maintaining high performance. Docker has become a powerful tool for containerizing development environments, allowing developers to integrate essential tools and extensions into a single, reproducible system. By combining components such as Opcache, Redis, Memcached, and xDebug into a single image, developers can eliminate repetitive setup tasks and focus on building reliable applications. Furthermore, using Docker Compose facilitates seamless service integration, ensuring uniform configurations for team members and minimizing potential conflicts. This study explores how tailored Docker images can simplify PHP development while enhancing performance and scalability.

Analysis of Recent Research and Publications. The study in [1] examines the methodology for installing PHP extensions in Docker images without using PECL, which has been disabled by default since PHP 7.4. It provides step-by-step instructions for manually installing extensions like APCu, Redis, Igbinary, and MongoDB using Docker commands and scripts. By utilizing tools like docker-php-ext-configure and docker-php-ext-install, developers can maintain clean, modular Docker files tailored to specific project requirements. The paper also discusses challenges that arise during manual installations, particularly with extensions like MongoDB, which involve submodules and multi-stage builds.

Through practical examples, the work demonstrates how to build a robust Docker environment for PHP development, emphasizing flexibility, performance, and modularity. Methods for integrating common extensions, enabling opcache, and fine-tuning configurations for improved caching and serialization are described in detail. The resulting Dockerfile encapsulates a clean, extensible setup that supports new PHP versions while ensuring compatibility and maintainability. This practical guide helps developers overcome limitations related to PECL deprecation by offering alternative approaches to building effective PHP environments for specific projects.

The study in [2] evaluates the comparative performance of PHP frameworks – Laravel, Symfony, and CodeIgniter – focusing on their suitability for various web development needs. PHP frameworks based on the Model-View-Controller (MVC) architecture have become essential tools for creating scalable, maintainable, and efficient web applications. Analyzing these frameworks using the QSOS evaluation method and benchmarking criteria, including requests per second, memory usage, response time, and file requirements, the study highlights their relative strengths and limitations.

The assessment underscores Laravel's dominance in handling a high number of requests per second and achieving the lowest response times, making it suitable for large-scale rapid development projects. While Symfony is robust and feature-rich, it requires more experienced developers for complex applications. CodeIgniter, being lightweight and flexible, is best suited for small to medium projects with simpler requirements. The QSOS evaluation also highlights that Laravel and Symfony align with modern development standards, whereas CodeIgniter struggles to meet contemporary demands.

These results highlight the importance of selecting a framework based on specific project needs, considering factors such as ease of use, scalability, technical capabilities, and infrastructure requirements. Ultimately, the study demonstrates that choosing a PHP framework significantly impacts development efficiency, maintainability, and the quality of web applications.

Additionally, noteworthy contributions from scholars such as V. Semeniuk [3], J. Zhao, Y. Lu, K. Zhu, Z. Chen, H. Cefuzz [4], J. Watkins [5], T. Sanoop [6], S. Neef, L. Kleissner [10], L. Moroz [11], J. Huang, J. Zhang, J. Liu, C. Li, R. Dai [12] and others are acknowledged.

Despite the documentation above, developing methodologies for containerizing PHP extensions within Docker images remains underexplored and requires further study.

Task Formulation. This work aims to develop a methodology for containerizing PHP extensions to enhance performance and optimize the development process using Docker.

Presentation of the Main Research Material. The optimization of local development processes often depends on ensuring that every element of the environment is configured to provide both flexibility and performance. When working with PHP, one of the most effective ways to achieve this configuration is by using a Docker image that comes with a predefined set of extensions and tools for caching, debugging, and performance tuning out of the box. Relying on a single image that integrates Opcache, Redis, Memcached, BCMath, and xDebug, along with other PHP-associated tools, reduces the overhead of constant environment reconfiguration and accelerates developers' workflows.

Using Docker Compose in conjunction with the PHP image can further simplify this process by integrating the ability to define services such as MySQL, Nginx, and RabbitMQ into a single configuration file. This aspect is particularly important in contexts where multiple team members require consistent environment configurations, as it ensures consistency, eliminates conflicts caused by mismatched PHP versions or extensions, and reduces the time spent on manual environment setup. Utilizing Docker for image creation ensures portability and reproducibility, enabling developers to retrieve the image and start working without having to manage underlying configuration details.

By building an image based on PHP: 7.1-fpm, the integration of any required libraries and extensions becomes easier, meaning that from the moment the container is created and launched, all critical extensions such as PDO (MySQL, PostgreSQL), GD for image manipulation, mcrypt for encryption, and BCMath for precise arithmetic calculations

are already supported. PDO facilitates seamless database interactions by acting as an interface for connecting to multiple databases. By abstracting database operations, it allows developers to execute queries, retrieve results, and manage transactions without relying on database-specific syntax, making it especially useful in applications requiring compatibility with various database management systems.

The integration of periodic health checks using Dockerfile commands ensures the expected operation of the container and can restore it in case of a failure. A typical health check for PHP-FPM includes sending a request to a predefined endpoint and verifying the response status: *HEALTHCHECK –interval=30s – timeout=10s – start-period=5s – retries=3 CMD curl -f http: // localhost: 9000 || exit 1*

Additional functionality can be implemented through build arguments, which direct Docker to install and configure optional extensions such as SOAP or xDebug. SOAP enables the development of web services by simplifying the exchange of structured information between applications over the network. It provides a standardized communication protocol that supports various transport mechanisms such as HTTP or SMTP. xDebug is a versatile extension for debugging and profiling PHP applications. It allows developers to gain deep insights into code execution by providing features such as stack trace visualization and breakpoint handling. Application performance profiling is achieved by generating detailed reports that highlight bottlenecks in code execution. This practice ensures that the base image remains relatively lightweight while offering the flexibility to add or remove components based on project needs.

From a performance standpoint, caching plays a critical role, especially in resource-intensive web applications. The inclusion of Redis and Memcached as optional PHP extensions enables developers to quickly activate caching layers without worrying about dependency conflicts or initial software installation. After building the image, it can be launched by mapping port 9000 and mounting a local directory to / var / www within the container, providing immediate access to an environment configured for running PHP-based applications.

One of the most critical features of this image is the inclusion of performance-driven extensions such as Opcache. By storing precompiled bytecode in memory, Opcache significantly reduces the overhead associated with parsing and compiling PHP scripts for every request. Considering that the total execution time of a PHP request can be broken down into parsing, compiling, and running the code, caching through Opcache directly impacts the compilation phase by skipping it for subsequent requests.

When bytecode caching is activated, compilation becomes negligible after the first request, thereby improving overall response times in high-load scenarios. Another critical optimization factor is the use of Redis or Memcached extensions, which provide in-memory storage mechanisms to reduce database load and speed up repetitive operations. By caching frequently manipulated data in RAM, these tools minimize costly operations such as database access or external API calls, thus enabling faster page loads.

In environments relying on distributed caching across multiple servers or containers, Memcached proves particularly valuable for horizontally scaled solutions, while Redis offers additional functionalities such as data persistence and pub / sub messaging. By shifting part of the input / output (I / O) load to memory, the time required for such operations is significantly reduced.

Moreover, for development teams dealing with large or high-precision numbers, having BCMath installed by default simplifies handling complex operations without risking loss of precision. Additionally, for debugging and profiling purposes, xDebug is typically used only in local environments, as it can negatively affect performance.

Activating it via a build argument allows seamless toggling during the build process, avoiding the need to maintain separate Dockerfiles or perform manual installations.

To build the image, navigate to the directory containing the target Dockerfile and execute the following command: *docker build -t php – 7.1-custom*

This command instructs Docker to copy the relevant configuration files for xDebug and SOAP while building the container image. Build arguments correspond to environment variables, such as INSTALL_XDEBUG or INSTALL_AEROSPIKE, within the Docker build context. This facilitates the conditional execution of installation and compilation processes for selected extensions: *docker build – build-arg INSTALL_XDEBUG=true – build-arg INSTALL_SOAP=true -t php – 7.1-custom*

This structure allows Dockerfiles to remain flexible and manageable, avoiding monolithic containers with unnecessary tools. Once the image is built, the container can be launched, and PHP-FPM exposed on port 9000, by running the command: *docker run -d -p 9000:9000 – v $(pwd): / var / www php – 7.1-custom*

In this setup, the current directory on the host machine is mounted to / var / www within the container, which is the standard directory for PHP-FPM processes. This ensures that any changes made on the local machine are immediately accessible to the container for testing or content population. Alongside these runtime instructions, it is possible to override certain configuration files related to PHP, xDebug, or Opcache by mounting local versions of php. ini, xdebug. ini, or opcache. ini. This approach aligns configuration management with versioning, allowing performance parameters to be adjusted without rebuilding the entire image and providing an additional layer of customization based on development needs.

Consolidating all the described components within a single image simplifies the development cycle in several ways. First, it significantly reduces the onboarding process for new team members, who no longer need to individually set up components such as MySQL, Redis, or Memcached. Instead, they can simply clone the repository and execute a single build command.

Additionally, the ability to deactivate or activate optional extensions during the build process ensures greater modularity of containers. For example, if a specific project does not require Memcached but relies on Redis, the Docker build process can be adjusted accordingly, ultimately optimizing both disk usage and overall performance. The overall consolidated algorithm for building the Docker image is expressed as follows (table 1).

Due to the inherent advantages of containerization, a Docker-based PHP 7.1 environment offers a unified interface across development, staging, and production environments. From a security perspective, Docker containerization ensures the isolation of the PHP environment from the hosting system, reducing the risk of system vulnerabilities. Furthermore, the portability of Docker images allows for consistent deployments across different environments, from local to cloud systems.

Conclusions. The use of Docker in PHP development provides a unified, efficient, and secure approach to managing development environments. By integrating critical tools and extensions, developers can reduce overhead, enhance performance, and ensure consistent configurations across teams. The portability and modularity of Docker enable seamless transitions between development, staging, and production environments, supporting modern software workflows. With features such as caching, debugging, and conditional extension installation, tailored Docker images not only improve developer productivity but also facilitate the creation of scalable and maintainable applications. Ultimately, Docker provides developers with the flexibility to meet the demands of diverse projects while maintaining high standards of performance and reliability.

Table 1

Algorithm for Building and Configuring a Docker PHP Environment

```
RUN docker-php-ext-install pdo pdo_mysql pdo_pgsql gd mcrypt bcmath /
&& pecl install redis memcached /
&& docker-php-ext-enable redis memcached
ARG INSTALL_SOAP=false
ARG INSTALL_XDEBUG=false
ARG INSTALL_AEROSPIKE=false
RUN if [ «$INSTALL_SOAP» = «true»]; then docker-php-ext-install soap; fi /
&& if [ «$INSTALL_XDEBUG» = «true»]; then pecl install xdebug && docker-php-
ext-enable xdebug; fi /
&& if [ «$INSTALL_AEROSPIKE» = «true»]; then pecl install aerospike && docker-
php-ext-enable aerospike; fi
COPY configs / php. ini / usr / local / etc / php / php. ini
COPY configs / php-fpm. pool. conf / usr / local / etc / php-fpm. d / www. conf
COPY configs / xdebug. ini / usr / local / etc / php / conf. d / xdebug. ini
COPY configs / opcache. ini / usr / local / etc / php / conf. d / opcache. ini
COPY configs / aerospike. ini / usr / local / etc / php / conf. d / aerospike. ini
EXPOSE 9000
WORKDIR / var / www
CMD [«php-fpm»]
```

Source: developed by the author

BIBLIOGRAPHY:

1. Laviale O. Installing PHP extensions from source in your Dockerfile. *Olvlvl*: website. 2019. URL: https://olvlvl.com/2019–06-install-php-ext-source.html (last accesed: 15.01.2025).

2. Laaziri M., Benmoussa K., Khoulji S., Kerkeb L. A comparative study of PHP frameworks performance. *Procedia Manufacturing*. 2019. Vol. 32. P. 864–871. DOI: 10.1016/j.promfg.2019.02.295.

3. Semeniuk V. Automated build for docker-php image. *Docker Hub*: website. 2025. URL: https://hub.docker.com/r/vadymsemeniuk/docker-php (last accesed: 15.01.2025).

4. Zhao J., Lu Y., Zhu K., Chen Z., Huang H. Cefuzz: A directed fuzzing framework for PHP RCE vulnerability. *Electronics*. 2022. Vol. 11. № 5. P. 758. DOI: 10.3390/electronics11050758.

5. Watkins J. PCOV – CodeCoverage compatible driver for PHP. *GitHub*: website. 2023. URL: https://github.com/krakjoe/pcov (last accesed: 15.01.2025).

6. The PHP Group. PHP: uopz – Manual. *PHP*: website. 2023. URL: https://www.php.net/manual/en/book.uopz.php (last accesed: 15.01.2025).

7. The PHP Group. PHP: register_shutdown_function – Manual. *PHP*: website. 2023. URL: https://www.php.net/manual/en/function.register-shutdown-function.php (last accesed: 15.01.2025).

8. Sanoop T. XVWA is a badly coded web application written in PHP / MySQL that helps security enthusiasts to learn application security. *GitHub*: website. 2015. URL: https://github.com/s4n7h0/xvwa (last accesed: 15.01.2025).

9. Q-Success. Usage statistics and market share of PHP for websites. *Web Technology Surveys*. 2023. URL: https://w3techs.com/technologies/details/pl-php (last accesed: 15.01.2025).

10. Neef S., Kleissner L. PHUZZ: A grey-box fuzzer for PHP web applications. *GitHub*: website. 2024. URL: https://github.com/gehaxelt/phuzz (last accesed: 15.01.2025).

11. Moroz L. bWAPP latest modified for PHP7. *GitHub*: website. 2018. URL: https://github.com/lmoroz/bWAPP (last accesed: 15.01.2025).

12. Huang J., Zhang J., Liu J., Li C., Dai R. UFuzzer: Lightweight detection of PHP-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis. *24th International Symposium on Research in Attacks, Intrusions and Defenses*. Association for Computing Machinery. New N. Y. York, USA, 2021. P. 78–90. DOI: 10.1145/3471621.3471859.

REFERENCES:

1. Laviale, O. (2019). Installing PHP extensions from source in your Dockerfile. *Olvlvl*. Retrieved from https://olvlvl.com/2019–06-install-php-ext-source.html [In English].

2. Laaziri, M., Benmoussa, K., Khoulji, S., & Kerkeb, L. (2019). A comparative study of PHP frameworks performance. *Procedia Manufacturing, 32*, 864–871. https://doi.org/10.1016/j.promfg.2019.02.295 [In English].

3. Semeniuk, V. (2025). Automated build for docker-php image. *Docker Hub*. Retrieved from https://hub.docker.com/r/vadymsemeniuk/docker-php [In English].

4. Zhao, J., Lu, Y., Zhu, K., Chen, Z., & Huang, H. (2022). Cefuzz: A directed fuzzing framework for PHP RCE vulnerability. *Electronics, 11* (5), 758. https://doi.org/10.3390/electronics11050758 [In English].

5. Watkins, J. (2023). PCOV – CodeCoverage compatible driver for PHP. *GitHub*. Retrieved from https://github.com/krakjoe/pcov [In English].

6. The PHP Group. (2023). PHP: uopz – Manual. *PHP*. Retrieved from https://www.php.net/manual/en/book.uopz.php [In English].

7. The PHP Group. (2023). PHP: register_shutdown_function – Manual. *PHP*. Retrieved from https://www.php.net/manual/en/function.register-shutdown-function.php [In English].

8. Sanoop, T. (2015). XVWA is a badly coded web application written in PHP / MySQL that helps security enthusiasts to learn application security. *GitHub*. Retrieved from https://github.com/s4n7h0/xvwa [In English].

9. Q-Success. (2023). Usage statistics and market share of PHP for websites. *Web Technology Surveys*. Retrieved from https://w3techs.com/technologies/details/pl-php [In English].

10. Neef, S., & Kleissner, L. (2024). PHUZZ: A grey-box fuzzer for PHP web applications. *GitHub*. Retrieved from https://github.com/gehaxelt/phuzz [In English].

11. Moroz, L. (2018). bWAPP latest modified for PHP7. *GitHub*. Retrieved from https://github.com/lmoroz/bWAPP [In English].

12. Huang, J., Zhang, J., Liu, J., Li, C., & Dai, R. (2021). UFuzzer: Lightweight detection of PHP-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis. *24th International Symposium on Research in Attacks, Intrusions and Defenses*. *Association for Computing Machinery*, New York, NY, USA, 78–90. https://doi.org/10.1145/3471621.3471859 [In English].