Jurnal Mandiri IT
Vol. 15 No. 1, July (2026), pp. 51-60

ISSN 2301-8984 (Print), 2809-1884 (Online)
Published by Institute of Computer Science

Comparative evaluation of local storage and MinIO object storage for django applications in docker-based environments

Hendra Saputra[1], Faldi[2], Wawan Joko Pranoto[3]
[1]Digital Business, Faculty of Agriculture and Digital Business, Universitas Muhammadiyah Kalimantan Timur, Indonesia
[2,3]Informatics Engineering, Faculty of Science and Technology, Universitas Muhammadiyah Kalimantan Timur, Indonesia

A R T I C L E I N F O

Article history:

Received Jun 14, 2026
Revised Jul 8, 2026
Accepted Jul 15, 2026

Keywords:

Django;
Docker;
Local Storage;
MinIO;
Object Storage;

ABSTRACT

Media file management, such as documents and images, was an important aspect of Django-based web application development. Local storage was commonly used to store media files; however, it had limitations in terms of deployment portability, data management, and service continuity in containerized environments. Object storage emerged as an alternative approach by separating file storage from the application layer. This study compared the use of local storage and MinIO object storage in a Django application deployed using Docker containers. A comparative experimental method was employed by implementing both storage approaches on the same application under identical deployment conditions. The evaluation focused on file upload functionality, file accessibility, data persistence after container restart and redeployment, and deployment portability. The results indicated that both approaches successfully stored and provided access to media files. However, MinIO demonstrated advantages in data persistence, storage management, and deployment portability because media files were maintained independently from the application container. In contrast, local storage offered simpler implementation but required additional management during application migration or infrastructure changes. The findings suggested that MinIO object storage was more suitable for Django applications requiring flexible deployment and scalable file management in container-based environments.

Corresponding Author:

Hendra Saputra,
Faculty of Agriculture and Digital Business,
Universitas Muhammadiyah Kalimantan Timur,
Jl. Juanda No. 15, Samarinda, 75124, Indonesia
Email: hs048@umkt.ac.id

1. INTRODUCTION

Django is one of the most widely used Python-based web frameworks in modern web application development because it provides various built-in features, such as Object-Relational Mapping (ORM), authentication systems, routing mechanisms, and the Model-View-Template (MVT) architecture, which support rapid, structured, and maintainable application development (Gore et al., 2021; Thakur & Jadon, 2023). Due to these characteristics, Django has been widely adopted in various web-based information systems, ranging from small-scale applications to enterprise-level systems (Thakur & Jadon, 2023).

One of the challenges frequently encountered in Django-based application development is the management of media files, such as documents, images, and videos. In most cases, files are stored locally on the application server. Although this approach is relatively simple to implement, it has several limitations, particularly in terms of scalability, data availability, and service sustainability

(Bayazitov et al., 2024; Keshireddy, 2022). The risk of data loss may increase during server migration, infrastructure changes, or application redeployment (Wang et al., 2023). These challenges become even more significant when applications are deployed in containerized or cloud environments, where the container file system is not inherently persistent (Barbarulo et al., 2022).

As the demand for flexible and scalable systems continues to grow, numerous studies have emphasized the importance of separating application components from data storage services (Rahaman et al., 2023; Velepucha & Flores, 2023). This approach allows applications and storage systems to evolve independently, thereby improving infrastructure management efficiency (Velepucha & Flores, 2023). Furthermore, the utilization of flexible and efficient infrastructure is an important strategy for achieving sustainable information technology resource management (Rahaman et al., 2023).

In the context of modern web application development, Django has been extensively utilized in various research studies and information system implementations. Django-based libraries have been proposed to standardize early-stage web application development, while other studies have analyzed the application of the Django framework in modern web development environments. Furthermore, Django has been used to develop web automation platforms, demonstrating its applicability in building extensible web-based systems. The findings of these studies indicate that Django is capable of supporting rapid, flexible, and extensible application development (Du, 2024; Martinez et al., 2023; Pang, 2024).

Although previous studies have investigated Django, Docker, and MinIO independently, most of them focused on application development, container deployment, or object storage implementation without providing a direct comparison between local storage and MinIO object storage under identical deployment conditions. In addition, limited studies have specifically evaluated the impact of these storage approaches on data persistence, deployment portability, and file management in Docker-based Django applications. This research gap motivates the need for a comparative evaluation using the same application and deployment environment.

Nevertheless, media file management in Django applications is still commonly performed using local storage, which has limitations in terms of portability and scalability (Bayazitov et al., 2024; Keshireddy, 2022). One alternative solution is object storage, a storage mechanism that separates file data from the application server. MinIO is an open-source object storage platform compatible with the Amazon S3 protocol and is widely used as a distributed storage solution (Makris et al., 2022; Salvi et al., 2025). Recent studies have also highlighted the growing adoption of MinIO and S3-compatible object storage systems due to their scalability, flexibility, and performance in distributed environments (Chung & Nakamura, 2026; Grandhe, 2025; Sundaramoorthy et al., 2025). Meanwhile, Docker technology enables application components, including Django, databases, and storage services, to be deployed consistently across development and production environments, thereby improving portability and deployment efficiency across different infrastructures (Raikar et al., 2024; Silva et al., 2024). Recent studies have further demonstrated that containerization improves deployment consistency, reproducibility, and resource utilization in modern software architectures (Fava et al., 2024; Muzumdar et al., 2024).

Based on these issues, an evaluation of file storage approaches used in modern container-based web applications is required. Local storage, which is commonly employed in Django applications, offers ease of implementation but presents limitations in maintaining data persistence during container redeployment, server migration, or infrastructure modifications (Bayazitov et al., 2024; Wang et al., 2023). In contrast, object storage solutions such as MinIO provide a storage mechanism that is independent of the application, thereby potentially improving data availability, scalability, and security (Makris et al., 2022; Salvi et al., 2025). Similar benefits have been reported in cloud-based and decentralized storage architectures, where the separation of storage services from application components improves scalability, reliability, and long-term data management (Doan et al., 2022; Xu et al., 2022). Therefore, a comparative evaluation is needed to assess the advantages and limitations of both storage approaches when implemented in containerized web application environments.

This study compared local storage and MinIO object storage approaches in a Django application deployed using Docker containers. The evaluation was conducted based on data persistence, file availability, and deployment portability to identify the strengths and limitations of each storage approach. The results are expected to provide recommendations for web application

developers in selecting a suitable file storage strategy for development and production environments based on container technology.

Unlike previous studies that primarily focused on Django implementation, Docker deployment, or object storage as separate topics, this study contributes a comparative evaluation of local storage and MinIO object storage under identical Docker-based deployment conditions. The study provides empirical evidence regarding differences in data persistence, deployment portability, and file management between the two storage approaches. These findings offer practical guidance for developers and organizations in selecting an appropriate file storage strategy based on deployment requirements, scalability needs, and long-term operational considerations in modern containerized Django applications.

2. RESEARCH METHOD

Figure 1 presents the research workflow designed to support a structured system development process. The workflow begins with determining the research approach and selecting the technologies used, followed by system development, architecture design, and testing scenarios to evaluate system performance. Each stage is interconnected and systematically organized to achieve the objectives of this study.

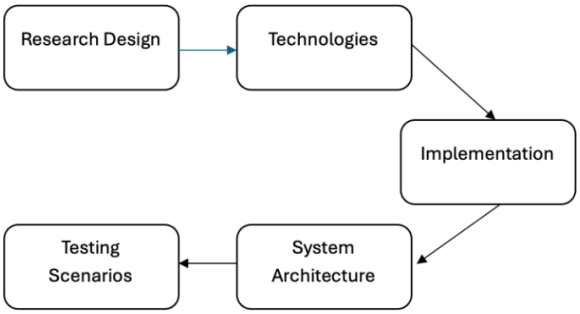

Figure 1. Research workflow

Research Design

This study employed a comparative experimental method to evaluate two file storage approaches in a Docker-based Django application, namely local storage and MinIO object storage. Both approaches were implemented within the same application and deployed in an identical environment to ensure that any differences observed during testing were solely attributable to the storage method used.

The evaluation was conducted using four evaluation metrics: (1) upload success, which measures whether files were successfully stored; (2) file availability, which measures whether uploaded files remained accessible after deployment events; (3) data persistence, which evaluates whether uploaded files were preserved after container restart and redeployment; and (4) deployment portability, which assesses whether the application can be migrated to another Docker environment without requiring manual file recovery. These metrics were applied consistently to both storage approaches under identical deployment conditions. Through this approach, the study aimed to identify the strengths and limitations of local storage and MinIO object storage in supporting file storage requirements for modern container-based web applications.

Technologies Used

Table 1. Technologies used

Component	Description
Django	Python web framework based on the Model-View-Template (MVT) architecture used for web application development.
MinIO	Open-source object storage platform compatible with the Amazon S3 protocol.
django-minio-storage	Django storage backend used to integrate Django applications with MinIO object storage.
boto3	Python SDK for interacting with Amazon S3-compatible object storage services.
Docker	Containerization platform used to deploy applications in a portable and consistent

Component	Description
	environment.
Docker Compose	Tool for defining and managing multi-container applications and services
Local Storage	File storage approach that uses the local filesystem within the Django application container.

Several technologies were utilized to support the implementation and evaluation of the proposed system. Django was used as the primary framework for web application development. To support file management, this study compared two storage approaches: local storage, which utilizes the local filesystem within the Django container, and MinIO object storage, which is compatible with the Amazon S3 protocol. Integration between Django and MinIO was implemented using the django-minio-storage package, which relies on the boto3 SDK to communicate with S3-compatible storage services.

All testing scenarios were executed using Docker and Docker Compose to ensure a consistent deployment environment. By utilizing containerized environments, both storage approaches could be evaluated under identical conditions, enabling a fair and objective comparison of their characteristics and performance.

Implementation Procedure

The study was conducted through several stages designed to compare the use of local storage and MinIO object storage in a Docker-based Django application.

a. Storage Scenario Design, at this stage, two file storage scenarios were defined for comparison. The first scenario employed Django's default local storage mechanism, where files were stored within the local filesystem of the application container. The second scenario utilized MinIO object storage as an external file storage backend, separating file management from the application environment.
b. Django Application Implementation, the same Django application was used in both scenarios to ensure that any differences observed during testing were solely caused by the storage method. The application provided file upload and retrieval functionalities using Django's FileField and ImageField components.
c. Docker-Based Deployment, both scenarios were deployed using Docker containers with equivalent configurations. In the local storage scenario, media files were stored within the container filesystem. In the MinIO scenario, media files were stored in an object storage bucket accessed through an Amazon S3-compatible protocol
d. System Testing, testing was performed to evaluate several aspects, including file upload operations, file accessibility after upload, container restart behavior, and container redeployment. These tests were designed to assess the ability of each storage approach to maintain file availability under different deployment conditions.
e. Result Analysis, the testing results from both scenarios were compared based on data persistence, deployment portability, and file management capabilities. The findings were subsequently analyzed to identify the strengths and limitations of each storage approach in supporting file storage requirements for container-based web applications.

System Architecture

This study employed two file storage architectures for comparison: a local storage architecture and a MinIO object storage architecture. Both architectures utilized the same Django application and were deployed within identical Docker container environments to ensure a fair comparison. The primary difference between the two architectures lies in the location where uploaded media files are stored.

In the local storage architecture, media files are stored directly within the local filesystem of the Django application container. This approach is simple to implement and does not require additional storage services. However, file availability depends on the persistence of the container's filesystem and may be affected by container redeployment or infrastructure changes.

In the MinIO object storage architecture, media files are stored in a dedicated MinIO bucket that is separate from the Django application. Communication between Django and MinIO is performed through the Amazon S3-compatible API using the django-minio-storage package. This architecture enables file storage to remain independent from the application lifecycle, thereby improving deployment portability and data persistence.

Figure 2. Django with local storage

In the first scenario, media files were stored using local storage within the local filesystem of the Django container. This approach represents Django's default file storage mechanism and is widely adopted in small- to medium-scale web applications due to its simplicity and ease of implementation. However, file availability depends on the persistence of the container filesystem, which may present challenges during container redeployment or infrastructure migration.

Figure 3. Django with MinIO storage

In the second scenario, media files were stored in a MinIO object storage service that operates independently of the Django application. Integration was implemented using a storage backend compatible with the Amazon S3 protocol. With this approach, media files are not dependent on the lifecycle of the application container, thereby providing improved data persistence and storage flexibility.

Both architectures were evaluated using identical testing scenarios to assess their ability to maintain file availability, support deployment processes, and simplify storage management in modern web applications. The comparison enabled the identification of the strengths and limitations of each storage approach under the same deployment conditions.

Testing Scenarios

To evaluate the two storage approaches, a series of experiments was conducted on both the local storage and MinIO object storage scenarios. The evaluation focused on file upload functionality, file availability after container restart and redeployment, data persistence, and deployment portability across different environments. The testing scenarios employed in this study are summarized in Table 2.

The evaluation was based on four predefined metrics: upload success, file availability, data persistence, and deployment portability. Upload success was measured by the successful completion of file upload operations. File availability was verified by accessing uploaded files after each deployment scenario. Data persistence was evaluated based on whether uploaded files remained available after container restart and redeployment. Deployment portability was assessed according to the amount of manual intervention required when migrating the application to another Docker environment.

Table 2. Testing scenarios

Testing Scenario	Evaluation Metric	Objective
File Upload using Local Storage	Upload Success	To verify that uploaded files are successfully stored in the local filesystem of the Django application.
File Upload using MinIO Storage	Upload Success	To verify that uploaded files are successfully stored in the MinIO object storage bucket.
File Access after Upload	File Availability	To verify that uploaded files remain accessible after the upload process is completed.
Django Container Restart	Data Persistence	To evaluate file availability after the Django container is restarted.
Django Container Redeployment	Data Persistence	To evaluate file persistence after the Django container is removed and redeployed.
Deployment Migration to a New Environment	Deployment Portability	To evaluate the portability of each storage approach when the application is deployed in a different environment.

3. RESULTS AND DISCUSSIONS

System Implementation

The system was implemented using two different file storage scenarios: local storage and MinIO object storage. Both scenarios utilized the same Django application and were deployed within identical Docker container environments. The primary difference between the two implementations lies in the media file storage configuration.

a. Local Storage Implementation, in the first scenario, media files were stored using Django's default local storage mechanism. Files uploaded by users were saved in the local filesystem within the Django application container. The storage configuration was implemented through the MEDIA_URL and MEDIA_ROOT parameters defined in the settings.py file.

```
MEDIA_URL = "/media/"
MEDIA_ROOT = BASE_DIR / "media"
```

Figure 4. Local storage configuration in django

After the configuration shown in Figure 4 was applied, files uploaded through the FileField and ImageField components were stored directly in the media directory of the Django application. This approach represents Django's default file storage mechanism and is commonly used in small- to medium-scale web applications due to its simplicity and ease of implementation.

b. MinIO Storage Implementation, in the second scenario, media files were stored using MinIO object storage, which operated as a service independent of the Django application. Integration was implemented using the django-minio-storage package, which utilizes the Amazon S3-compatible protocol to communicate with the MinIO storage service.

This configuration enabled uploaded files to be stored in a dedicated MinIO bucket rather than within the local filesystem of the Django container. As a result, file storage remained independent of the application lifecycle, providing improved data persistence and greater deployment flexibility in containerized environments.

```
121 STATIC_URL = '/static/'
122 STATIC_ROOT = './static_files/'
123
124 STORAGES = {
125     "default": {
126         "BACKEND": "minio_storage.storage.MinioMediaStorage",
127     },
128     "staticfiles": {
129         "BACKEND": "minio_storage.storage.MinioStaticStorage",
130     },
131 }
132
133 MINIO_STORAGE_ENDPOINT = config('MINIO_STORAGE_ENDPOINT', default='minio.umkt.ac.id')
134 MINIO_STORAGE_ACCESS_KEY = config('MINIO_STORAGE_ACCESS_KEY')
135 MINIO_STORAGE_SECRET_KEY = config('MINIO_STORAGE_SECRET_KEY')
136 MINIO_STORAGE_USE_HTTPS = config('MINIO_STORAGE_USE_HTTPS', default=True, cast=bool)
137 MINIO_STORAGE_MEDIA_BUCKET_NAME = config('MINIO_STORAGE_MEDIA_BUCKET_NAME')
138 MINIO_STORAGE_AUTO_CREATE_MEDIA_BUCKET = True
139 MINIO_STORAGE_MEDIA_URL = config('MINIO_STORAGE_MEDIA_URL', default=None)
140 MINIO_STORAGE_STATIC_BUCKET_NAME = config('MINIO_STORAGE_STATIC_BUCKET_NAME')
141 MINIO_STORAGE_AUTO_CREATE_STATIC_BUCKET = True
142 MINIO_STORAGE_STATIC_URL = config('MINIO_STORAGE_STATIC_URL', default=None)
```

Figure 5. MinIO storage configuration in django

After the storage configuration was completed, the Django models could be implemented using standard FileField and ImageField components without requiring significant modifications. The storage backend automatically handled file uploads and retrieval operations based on the configured storage method. Consequently, the same application code could be used for both local storage and MinIO object storage scenarios, simplifying development and maintenance while enabling a fair comparison between the two approaches.

```
19 class ArtikelBlog(models.Model):
20     kategori = models.ForeignKey(Kategori, on_delete=models.CASCADE)
21     judul = models.CharField(max_length=200)
22     konten = CKEditor5Field('Text', config_name='extends')
23     gambar = models.ImageField(upload_to="artikel", blank=True, null=True)
```

Figure 6. Django model for file upload

Figure 6 presents the Django model used for file upload functionality in both storage scenarios. The model utilizes Django's standard FileField and ImageField components to handle document and image uploads. Since the storage backend is configured at the application level, the same model implementation can be used for both local storage and MinIO object storage without requiring modifications to the model structure. This approach simplifies application development while ensuring consistency across the two storage scenarios evaluated in this study.

c. Docker-Based Deployment, the Django application and MinIO object storage service were deployed within the same container network using Docker and Docker Compose. This deployment approach enabled direct communication between services while maintaining an isolated and consistent execution environment. By utilizing containerization, the application and storage service could be deployed, tested, and migrated more easily across different environments.

Figure 7 shows the Dockerfile configuration used to build the Django application container. The Dockerfile defines the Python runtime environment, installs the required dependencies, copies the application source code, and specifies the command used to start the Django application. This configuration ensures that the application can be deployed consistently across different development and testing environments.

Figure 7. Dockerfile configuration of the django application

Figure 8 presents the docker-compose.yml configuration used to orchestrate the Django application and MinIO object storage service. Docker Compose was utilized to define and manage multiple containers within a single deployment environment, including container networking, service dependencies, environment variables, and persistent storage volumes.

After the configuration was completed, both the Django application and MinIO services were deployed using the docker-compose up -d command. This approach simplified the deployment process and ensured that all services were started with consistent configurations. Furthermore, Docker Compose facilitated reproducible deployments, allowing the same environment to be recreated easily across different development and testing platforms.

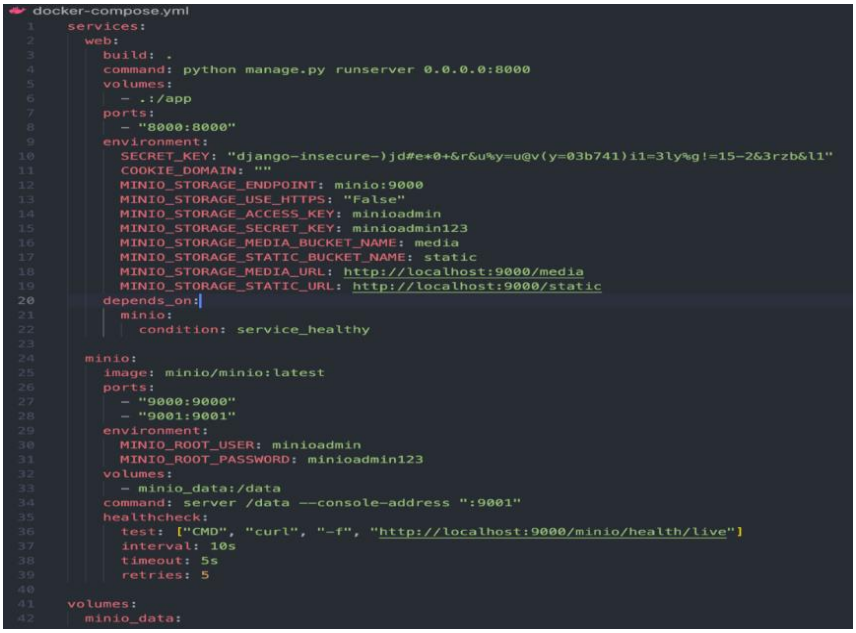

Figure 8. Docker compose configuration

Testing Results

The testing phase was conducted to compare the characteristics of local storage and MinIO object storage in a Docker-based Django application. The evaluation focused on file storage capability, file availability after container restart and redeployment, and deployment portability when migrating the application to different environments. The testing results were analyzed using the predefined evaluation metrics of upload success, file availability, data persistence, and deployment portability.

Table 3. Testing results

Testing Scenario	Local Storage	Minio Storage
File Upload	Successful	Successful
File Access After Upload	Successful	Successful
Container Restart	Successful	Successful
Container Redeployment	Requires separate management of the media directory to ensure file availability	Files remain available because they are stored independently in MinIO storage
Deployment Migration	Requires manual transfer of media files	Easier migration because storage is separated from the application
Deployment Portability	Moderate	High

Based on the testing results presented in Table 3, both storage approaches were able to successfully store and provide access to media files uploaded through the Django application. In terms of basic functionality, local storage and MinIO object storage demonstrated similar behavior, as uploaded files could be stored and retrieved successfully.

The primary differences were observed in storage management and deployment-related scenarios. In the local storage approach, media files were stored together with the application environment, requiring additional management during application migration, container redeployment, or infrastructure changes. In contrast, MinIO stored files in a separate object storage service operating independently of the Django application. As a result, media files remained available even when the application container was redeployed or updated.

The testing results indicate that MinIO object storage provides greater deployment portability than local storage. This finding is consistent with previous studies that reported improved flexibility and storage management when storage services are separated from application infrastructures (Gorantla et al., 2024). In addition to simplifying deployment and migration processes, the separation between application services and file storage offers greater flexibility in storage management. Similar advantages have been reported in modern data storage

architectures that separate storage resources from application processing layers to improve maintainability and scalability (Zagan & Danubianu, 2023). These characteristics make MinIO a more suitable solution for modern containerized web applications that require scalability, portability, and long-term data persistence.

Beyond improving deployment portability and data persistence, the use of MinIO also has important implications for the scalability of Django applications. Since media files are stored independently from the application container, multiple Django instances can simultaneously access the same object storage without requiring local file synchronization. This architecture simplifies horizontal scaling, supports load-balanced and horizontally scalable deployments, and reduces storage management complexity in containerized environments. In contrast, local storage becomes increasingly difficult to maintain as the number of application instances grows because uploaded files must be synchronized across multiple containers or servers.

4. CONCLUSION

This study compared local storage and MinIO object storage approaches in a Docker-based Django application. The testing results demonstrated that both storage methods were capable of supporting media file upload and access operations effectively within the Django environment. The main differences were identified in terms of data persistence and deployment portability. Local storage offers a simple implementation by utilizing Django's built-in filesystem-based storage mechanism. However, additional file management is required during application migration, container redeployment, or infrastructure modifications. In contrast, MinIO object storage provides a storage mechanism that is independent of the application lifecycle, simplifying deployment processes and improving data management in containerized environments. Previous studies have also demonstrated the applicability of MinIO in modern distributed and virtualized environments that require scalable and reliable storage services (Mehmood & Hussain, 2025). Based on the findings, MinIO is more suitable for applications that require deployment flexibility, storage-service separation, and structured file management. Meanwhile, local storage remains a practical solution for applications with relatively simple storage requirements and stable deployment environments. Based on the comparison results, local storage is recommended for small-scale web applications or development environments where deployment is relatively stable and storage requirements are simple. In contrast, MinIO object storage is recommended for production environments, cloud-native applications, and containerized deployments that require better deployment portability, independent storage management, and higher data persistence. Therefore, the selection of the storage method should be adjusted according to the application scale, deployment architecture, and long-term operational requirements. The main contribution of this research is the practical comparison of local storage and MinIO object storage in a Docker-based Django application using identical deployment conditions. This study provides empirical evidence regarding the differences in data persistence, deployment portability, and storage management between the two approaches. The findings offer practical guidance for developers and organizations in selecting an appropriate file storage strategy for modern containerized Django applications.

REFERENCES

Barbarulo, F., Puliafito, C., Virdis, A., & Mingozzi, E. (2022). Extending ETSI MEC towards stateful application relocation based on container migration. *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 367–376.

Bayazitov, D., Kozhakhmet, K., Omirali, A., & Zhumaliyeva, R. (2024). Leveraging Amazon Web Services for cloud storage and AI algorithm integration: A comprehensive analysis. *Applied Mathematics*, *18*(6), 1235–1246.

Chung, Y. T., & Nakamura, T. (2026). Optimizing Object Storage Performance for Large File Uploading under Small-start Environments. *Journal of Information Processing*, *34*, 86–94.

Doan, T. V., Psaras, Y., Ott, J., & Bajpai, V. (2022). Toward decentralized cloud storage with IPFS: opportunities, challenges, and future considerations. *IEEE Internet Computing*, *26*(6), 7–15.

Du, J. (2024). Programming and Development of Web Automation Application Platform Based on Django. *Proceedings of the 2024 2nd International Conference on Artificial Intelligence, Systems and Network Security*, 197–200.

Fava, F. B., Leite, L. F. L., Da Silva, L. F. A., Costa, P. R. D. S. A., Nogueira, A. G. D., Lopes, A. F. G., Schepke, C., Kreutz, D. L., & Mansilha, R. B. (2024). Assessing the performance of docker in docker

containers for microservice-based architectures. *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 137–142.

Gorantla, V. A. K., Gude, V., Sriramulugari, S. K., Yuvaraj, N., & Yadav, P. (2024). Utilizing hybrid cloud strategies to enhance data storage and security in e-commerce applications. *2024 2nd International Conference on Disruptive Technologies (ICDT)*, 494–499.

Gore, H., Singh, R. K., Singh, A., Singh, A. P., Shabaz, M., Singh, B. K., & Jagota, V. (2021). Django: Web development simple & fast. *Annals of the Romanian Society for Cell Biology*, *25*(6), 4576–4585.

Grandhe, R. C. T. (2025). Comparative Analysis of Object Storage Clients for JULEA Framework: AWS vs. MinIO. *International Conference on Advanced Computing Techniques in Engineering & Technology*, 77–85.

Keshireddy, S. R. (2022). Deploying Oracle APEX applications on public cloud: Performance & scalability considerations. *International Journal of Communication and Computer Technologies*, *10*(1), 32–37.

Makris, A., Kontopoulos, I., Psomakelis, E., Xyalis, S. N., Theodoropoulos, T., & Tserpes, K. (2022). Performance analysis of storage systems in edge computing infrastructures. *Applied Sciences*, *12*(17), 8923.

Martinez, D., Gonzalez, D., Sánchez, J. M., & Toasa, R. M. (2023). Library in django framework to standardize early-stage web application development. *2023 18th Iberian Conference on Information Systems and Technologies (CISTI)*, 1–4.

Mehmood, K. T., & Hussain, M. M. (2025). Traffic Profiling and Secure Virtualized Data Handling of 5G Networks via MinIO Storage. *Computers, Materials & Continua*, *85*(3), 5643–5670.

Muzumdar, P., Bhosale, A., Basyal, G. P., & Kurian, G. (2024). Navigating the docker ecosystem: A comprehensive taxonomy and survey. *ArXiv Preprint ArXiv:2403.17940*.

Pang, R. (2024). Analysis of Python web development applications based on the Django framework. *Third International Conference on Electronic Information Engineering, Big Data, and Computer Technology (EIBDCT 2024)*, *13181*, 1580–1584.

Rahaman, M. S., Tisha, S. N., Song, E., & Cerny, T. (2023). Access control design practice and solutions in cloud-native architecture: A systematic mapping study. *Sensors*, *23*(7), 3413.

Raikar, M. M., Patil, P., Guggari, S., Shavi, P., Mudavi, S., Patil, N., & Rangannavar, V. (2024). Leveraging Docker Containers for Deployment of Web Applications in Microservices Architecture. *2024 First International Conference for Women in Computing (InCoWoCo)*, 1–6.

Salvi, A., Masram, B., Deokar, V., Kolte, P., & Dhas, S. (2025). Leveraging MinIO and FileCloud for On-Premise Storage-as-a-Service in Educational Institutions. *2025 Seventeenth International Conference on Contemporary Computing (IC3)*, 1–4.

Silva, D., Rafael, J., & Fonte, A. (2024). Toward optimal virtualization: An updated comparative analysis of docker and lxd container technologies. *Computers*, *13*(4), 94.

Sundaramoorthy, J., Chandrasekaran, A., Pichaivel, A., Kayalvili, S., & Devadharshini, T. (2025). A Blockchain-Enabled CMS for Transparency and Data Integrity in E-Commerce Applications. *2025 International Conference on Intelligent Computing, Information and Control Systems (ICOIICS)*, 659–666.

Thakur, P., & Jadon, P. (2023). Django: Developing web using python. *2023 3rd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)*, 303–306.

Velepucha, V., & Flores, P. (2023). A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*, *11*, 88339–88358.

Wang, X., Hu, X., Fan, W., & Wang, R. (2023). Efficient data persistence and data division for distributed computing in cloud data center networks: X. Wang et al. *The Journal of Supercomputing*, *79*(14), 16300–16327.

Xu, C., Du, X., Fan, X., Giuliani, G., Hu, Z., Wang, W., Liu, J., Wang, T., Yan, Z., & Zhu, J. (2022). Cloud-based storage and computing for remote sensing big data: a technical review. *International Journal of Digital Earth*, *15*(1), 1417–1445.

Zagan, E., & Danubianu, M. (2023). Data lake architecture for storing and transforming web server access log files. *IEEE Access*, *11*, 40916–40929.