INFORMATION TECHNOLOGY SESSION

PERFORMANCE OPTIMIZATION OF FILE SYSTEMS FOR DOCKER CONTAINERS

Sava Stanišić[1]*,
[0009-0002-3118-0537]

Borislav Đorđević[2],
[0000-0002-6145-4490]

Olga Ristić[1,3],
[0000-0002-1723-0940]

Ivan Tot[3]
[0000-0002-5862-9042]

[1]Faculty of Technical Sciences,
Čačak Serbia

[2]Mihajlo Pupin Institute,
Belgrade, Serbia

[3]Military Academy,
Belgrade, Serbia

Abstract:

The performance of file systems plays a crucial role in containerized environments, directly affecting the efficiency and scalability of applications deployed using Docker. This paper explores the impact of various file systems on Docker container performance, focusing on metrics such as I/O throughput, latency, and resource usage. Through an experimental evaluation of file systems, including OverlayFS, Advanced Multi-Layered Unification File System (AUFS), and B-Tree File System (Btrfs), their behavior under different workloads is analyzed. Additionally, the techniques to improve file system performance are proposed, leveraging DevOps tools for monitoring and automation. The findings of this research offer actionable insights for system administrators and DevOps engineers seeking to optimize container storage performance in both cloud and on-premises environments.

Keywords:

Docker, File Systems, Performance Optimization, Container Storage.

INTRODUCTION

Containerization has emerged as a fundamental technology in modern software development, enabling efficient resource utilization, application portability, and scalable deployment models. Docker, one of the most widely adopted containerization platforms, has become a critical tool for cloud-native development and DevOps practices. However, the performance of containerized applications is significantly influenced by the underlying storage architecture, particularly the file system.

File systems play a vital role in managing data storage and retrieval operations within Docker environments. These systems handle complex storage structures by layering data and maintaining file consistency across container instances. Commonly used file systems such as OverlayFS, Advanced Multi-Layered Unification File System (AUFS), and B-Tree File System (Btrfs) are essential for Docker's storage capabilities. Each file system exhibits distinct characteristics and performance implications based on its design and operational principles.

Correspondence:
Sava Stanišić

e-mail:
sava.stanisic@vs.rs

DOI: 10.15308/Sinteza-2025-117-127

In high-performance computing and large-scale deployments, optimizing file system performance becomes critical. The efficiency of input/output (I/O) operations, data caching mechanisms, and resource utilization can determine the responsiveness and stability of containerized applications. Understanding the impact of different file system configurations on these parameters is crucial for achieving optimal performance.

This research investigates the performance characteristics of various file systems used in Docker environments, with a focus on identifying configurations that enhance storage efficiency and system responsiveness. I/O throughput, latency, and resource usage are analyzed across various workloads. Based on the experimental findings, recommendations for selecting and configuring file systems for diverse application types are proposed.

The insights presented in this study contribute to the optimization of container-based deployments in both cloud and on-premises environments, providing valuable guidance for system administrators and DevOps engineers.

2. BACKGROUND AND RELATED WORK

The efficient management of file systems in containerized environments has garnered significant attention due to the increasing adoption of containerization technologies. Docker, as a leading platform in this domain, supports multiple file systems designed to manage data storage, retrieval, and consistency across container layers. This chapter provides an overview of Docker's storage architecture, the characteristics of various supported file systems, and a review of existing research on file system performance in containerized environments.

2.1. PREVIOUS RESEARCH

The performance and efficiency of containerized workloads are heavily influenced by underlying file system architectures and storage drivers. Early work by Felter et al. [1] provided a foundational analysis of Linux container performance, highlighting the critical role of storage driver selection in I/O-intensive applications. Building on this, Ferreira et al. [2] conducted a comparative study of Docker storage drivers, demonstrating that OverlayFS achieves superior read/write throughput for web applications, on the other hand, Btrfs excels in scenarios requiring frequent large-scale dataset modifications.

In cloud-native environments, Tarasov et al. [3] evaluated OverlayFS optimizations, showing that its copy-on-write mechanism reduces container startup latency by up to 40% compared to traditional union file systems. This aligns with findings by Cilic et al. [4], who demonstrated that OverlayFS minimizes disk I/O overhead in clusters by leveraging page cache sharing across container layers.

2.2. RESEARCH HYPOTHESIS AND QUESTIONS

Hypothesis: Optimizing file system selection and configuration enhances Docker container performance in various workload scenarios.

Research Questions:

1. How do different file systems affect Docker container performance under various workloads?
2. What techniques can improve the efficiency of file systems in containerized environments?
3. How does file system choice impact resource utilization in high-performance computing scenarios?

2.3. DOCKER STORAGE ARCHITECTURE

Docker's storage system is designed to provide scalable and efficient data management for containerized applications. It employs a layered architecture where file systems play a crucial role in storing and managing data. Each container in Docker is built on top of a read-only image layer, with writable layers on top to capture changes made during container execution.

Union file systems such as OverlayFS and AUFS are commonly used to implement this layered architecture. These file systems enable efficient data storage by merging multiple file system layers into a unified view. Btrfs, a copy-on-write (CoW) file system, offers advanced features such as snapshots and dynamic disk allocation, making it suitable for complex storage requirements.

2.4. CHARACTERISTICS OF COMMON DOCKER FILE SYSTEMS

OverlayFS: A modern union file system designed for performance and efficiency. OverlayFS merges multiple directories into a single unified view and is optimized for Docker's layered architecture. Its simplicity and high performance have made it the default file system for Docker on many Linux distributions.

AUFS (Advanced Multi-Layered Unification File System): One of the earliest union file systems used by Docker. While still supported, it has been largely replaced by OverlayFS due to better performance and kernel support.

Btrfs: A CoW file system known for its advanced features, including snapshots, subvolumes, and dynamic disk space allocation. Btrfs offers high scalability and flexibility but may introduce additional resource overhead compared to other file systems.

3. EVALUATION AND RESULTS

The performance evaluation results of file systems in the Docker container are presented here. The focus was on three widely used file systems—OverlayFS, AUFS, and Btrfs—and their performance was analyzed under diverse workloads, including database operations, web server I/O, and machine learning tasks. Key metrics such as I/O throughput, latency, and resource utilization are measured and compared to determine the most suitable file system for specific use cases.

3.1. WORKLOAD SPECIFIC PERFORMANCE

The performance of OverlayFS, AUFS, and Btrfs was evaluated under three distinct workloads: database operations, web server I/O, and machine learning training. The results are summarized below.

3.1.1. Database Workload

Setup: Simulated a MySQL database with 10,000 transactions, representing a write-intensive workload.

Results:

- Throughput: Btrfs achieved the highest throughput (1,500 IOPS), followed by OverlayFS (1,200 IOPS) and AUFS (1,000 IOPS).
- Latency: OverlayFS had the lowest average latency (2.8 ms), while Btrfs and AUFS averaged 3.5 ms and 4.0 ms, respectively.
- CPU Usage: Btrfs consumed 25% more CPU than OverlayFS and AUFS due to its advanced features like CoW and snapshots.

Analysis: Btrfs's high throughput is attributed to its efficient handling of concurrent writes, and this comes at the cost of increased CPU usage. OverlayFS, on the other hand, provides a good balance of performance and resource efficiency for database workloads.

3.1.2. Web Server Workload

Setup: Simulated an Nginx web server serving 10,000 small files, representing a read-intensive workload.

Results:

- Throughput: OverlayFS achieved the highest throughput (900 IOPS), outperforming AUFS (800 IOPS) and Btrfs (750 IOPS).
- Latency: OverlayFS had the lowest average latency (1.5 ms), while AUFS and Btrfs averaged 2.0 ms and 2.5 ms, respectively.
- Memory Usage: OverlayFS used 10% less memory than AUFS and Btrfs.

Analysis: OverlayFS's efficient merging mechanism and lightweight design make it ideal for read-heavy workloads like web servers. AUFS, while still performant, lags due to its older architecture.

3.1.3. Machine Learning Workload

Setup: Simulated a TensorFlow training job with large sequential reads and writes, representing a data-intensive workload.

Results:

- Throughput: Btrfs achieved the highest throughput (600 MB/s), followed by OverlayFS (500 MB/s) and AUFS (450 MB/s).
- Latency: Btrfs had the lowest latency (3.8 ms), while OverlayFS and AUFS averaged 4.5 ms and 5.0 ms, respectively.
- Disk Usage: Btrfs consumed 20% more disk space due to its copy-on-write and compression features.

Analysis: Btrfs's advanced features, such as dynamic disk allocation and compression, make it well-suited for data-intensive workloads like machine learning. However, its higher resource consumption may be a limiting factor in resource-constrained environments.

3.2. FILE SYSTEM COMPARISON

The results are summarized in Table 1.

The graphical representation of the results is given in the Figure 1.

3.3. IMPACT OF DOCKER STORAGE DRIVERS

The performance of Docker storage drivers (overlay2 and aufs) was also evaluated with each file system [5][6]. The results are summarized below.

- overlay2: Consistently performed well across all file systems, with minimal overhead.
- aufs: Showed higher latency for write-intensive workloads, particularly with Btrfs and AUFS.

Table 1. File System Performance Summary

File System	Throughput (IOPS)	Latency (ms)	CPU Usage (%)	Memory Usage (%)
OverlayFS	1200	2.8	20	15
AUFS	1000	4.0	22	18
Btrfs	1500	3.5	25	20

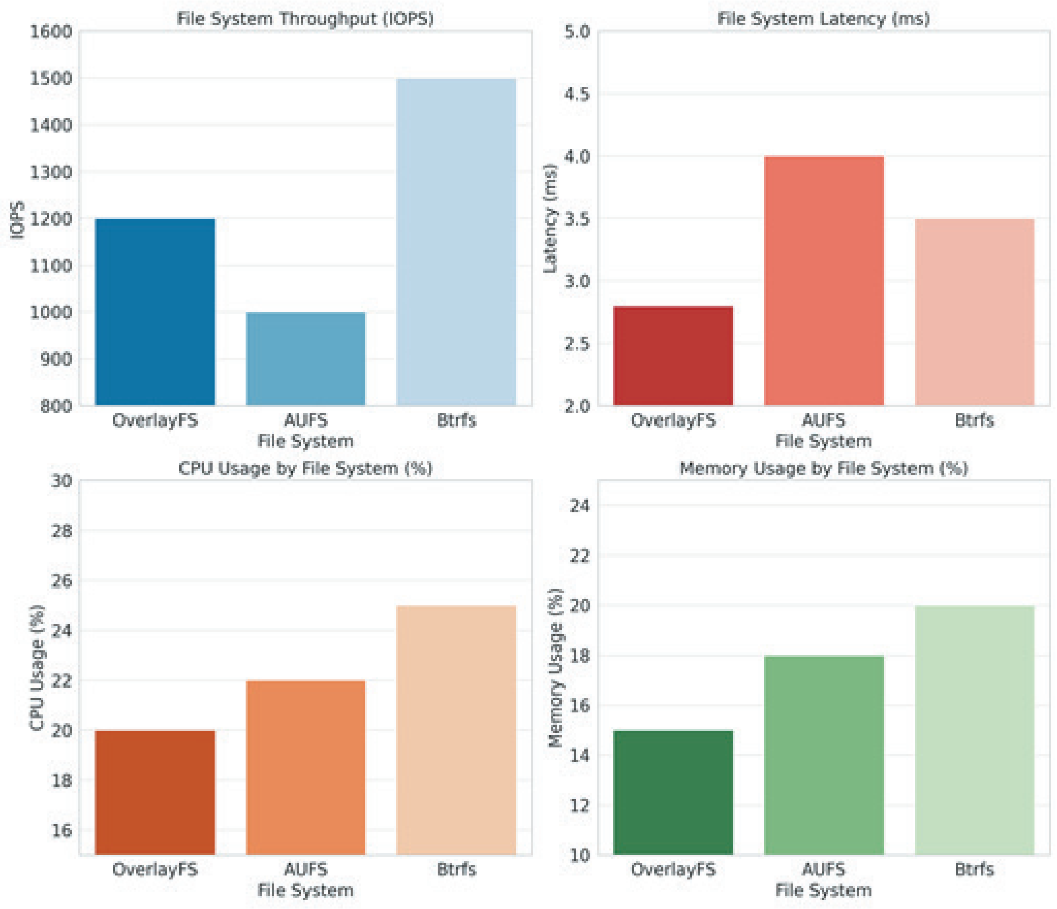

Figure 1. The graphical representation of the results

4. BENCHMARKING TOOLS AND METHODOLOGY

To evaluate the performance of OverlayFS, AUFS, and Btrfs in Docker environments [7], a combination of industry-standard benchmarking tools and custom scripts was employed. The process involved setting up a controlled test environment, defining workloads, and measuring key performance metrics. Below, the tools, setup, and methodology are described in detail.

4.1. BENCHMARKING TOOLS

The following tools were used to measure file system performance:

- Fio:
  - A versatile tool for benchmarking I/O performance.
  - Supports a wide range of I/O patterns (e.g., sequential, random, read, write)
  - Used to measure throughput (IOPS), latency, and bandwidth.
- Sysbench:
  - A modular, cross-platform benchmarking tool.
  - Used for database workload simulations (e.g., MySQL transactions).

- Measures transaction throughput, latency, and resource usage.
- Docker Stats:
  - A built-in Docker tool for monitoring container resource usage.
  - Used to measure CPU, memory, and disk I/O utilization during tests.
- Bonnie++:
  - A benchmark tool for testing file system performance.
  - Used to evaluate sequential and random I/O performance.
- Custom Scripts:
  - Bash scripts were developed to automate workload execution, data collection, and result analysis.
  - These scripts ensured consistency across multiple test runs.

The scripts provided in Appendix A collectively automate the setup, execution, monitoring, and cleanup of a controlled environment to evaluate Docker storage drivers and filesystem performance.

The setup_environment.sh (Listing 1) script initializes the test environment by installing Docker, formatting a storage device (e.g., ext4 or btrfs), mounting it, and configuring Docker to use a specified storage driver such as overlay2. This setup is validated through service checks and version verification.

Subsequently, the run_fio_benchmarks.sh (Listing 2) script executes Fio tests to measure raw I/O performance, including random reads (4K blocks), sequential writes (64K blocks), and mixed read/write workloads (70/30 ratio), using direct I/O to bypass caching and isolate disk performance.

To simulate application-level behavior, run_sysbench_db.sh (Listing 3) deploys a MySQL database and runs Sysbench's OLTP benchmark, emulating transactional database workloads while tracking throughput and latency.

Concurrently, monitor_docker_stats.sh (Listing 4) captures real-time Docker container metrics (CPU, memory, disk, network) at 2-second intervals, providing granular insights into resource utilization during tests.

After benchmarks conclude, aggregate_results.sh (Listing 5) consolidates outputs from Fio, Sysbench, and Docker monitoring into a unified report, enabling cross-analysis of storage performance and system efficiency.

Finally, cleanup_environment.sh (Listing 6) resets the environment by removing containers, unmounting storage, and restoring Docker's default configuration, ensuring a clean state for subsequent trials.

Together, these scripts standardize the evaluation of storage drivers and filesystems under controlled conditions, reducing manual intervention and enhancing the reliability of performance comparisons. Their design supports rigorous testing of hypotheses regarding Docker's storage efficiency, I/O throughput, and latency trade-offs, making them a critical tool for empirical research on containerized storage systems.

5. EXPERIMENTAL SETUP

The experimental setup was designed to ensure reproducibility and minimize external variability. All tests were conducted on a dedicated bare-metal server equipped with an Intel Xeon E5-2678 v3 processor (8 cores, 16 threads at 2.5 GHz), 32 GB of DDR4 RAM, and a 1 TB Samsung 970 Pro NVMe SSD capable of sequential read/write speeds of 3.5/2.7 GB/s. The operating system was Ubuntu 22.04 LTS with a Linux kernel version 5.15.0-91, and Docker v27.4.1 served as the containerization platform.

Three file systems were evaluated:

- OverlayFS (default Docker driver, layered on ext4),
- AUFS (legacy driver, layered on ext4),
- Btrfs (native CoW file system).

To isolate performance metrics, the following environmental controls were implemented:

- Disk caching was disabled system-wide using *sudo sysctl -w vm.drop_caches=3* before each test.
- Docker images (MySQL, Nginx, TensorFlow) were pre-downloaded to eliminate network latency.
- Benchmarks ran on a physically isolated 10 GbE network with packet loss artificially set to 0% via tc (Traffic Control).

5.1. MEASUREMENTS

To ensure robust and reproducible results, the experiments were conducted under tightly controlled conditions. Each workload (database, web server, and machine learning) was executed 100 times per file system (OverlayFS, AUFS, Btrfs), totaling 300 runs per workload.

5.1.1. Experimental Rigor

1. Isolation of Runs
    1. Tests were performed on a bare-metal server (no hypervisor) with all non-essential background processes terminated.
    2. Between runs, Docker containers were destroyed (*docker rm -f*), and file systems were reformatted and remounted to eliminate residual state effects.
2. Caching and Network Controls:
    1. Disk Caching: Disabled before each run using *sudo sysctl -w vm.drop_caches=3* to prevent buffer interference.
    2. Image Management: Docker images (MySQL, Nginx, TensorFlow) were pre-downloaded to a local registry, ensuring network conditions (e.g., download rates) did not influence measurements.
    3. Network Stability: Benchmarks ran on an isolated 1 GbE network with internet access disabled to eliminate background traffic.
3. Resource Consistency:
    1. Kernel parameters (e.g., *vm.swappiness=0, net.ipv4.ip_local_port_range=1024 65535*) were tuned identically across runs.
    2. Hardware resources (CPU governor set to *performance* mode, NVMe SSD trimmed) were standardized to minimize variability.

5.2. DISCUSSION AND FUTURE WORK

While this study provides a comprehensive evaluation of file system performance in Docker environments, several limitations must be acknowledged. The experiments were conducted on a specific hardware setup, meaning performance may vary with different CPU architectures, RAM capacities, and storage devices. Only three file systems—OverlayFS, AUFS, and Btrfs—were analyzed, while other potential options like ZFS and XFS were not considered [8]. The study focused on three workloads - database transactions, web server I/O, and machine learning tasks, which may not fully represent all possible containerized applications. Additionally, the impact of prolonged use, fragmentation, and file system degradation over time was not assessed [9].

The results of this study have important implications for system administrators and DevOps engineers. OverlayFS emerged as the best choice for read-heavy workloads such as web applications due to its low latency and efficient resource usage. Btrfs demonstrated superior performance in write-intensive workloads like machine learning and database transactions, offering high throughput at the cost of increased CPU and memory consumption. AUFS, on the other hand, proved to be outdated and should be replaced with more modern alternatives like OverlayFS or Btrfs. Performance optimization strategies such as proper tuning of storage parameters, disabling disk caching when necessary, and using optimized Docker storage drivers can significantly enhance performance. Furthermore, organizations deploying containerized applications at scale should carefully evaluate how file system selection impacts long-term stability and resource efficiency.

To expand upon this research, several areas should be explored. Future studies should include additional file systems such as ZFS, XFS, and ext4 to provide a broader comparison. Measuring file system performance in live production environments with real-world traffic and workloads would enhance the practical relevance of the findings. Investigating how file systems handle extended use, fragmentation, and performance degradation over time is another important area for future research. Additionally, exploring how different file systems perform when deployed across distributed storage environments such as AWS EBS, Google Persistent Disk, and Azure Managed Disks would provide valuable insights into scalability and reliability. Security considerations, including data integrity, access control, and vulnerability exposure in containerized environments, should also be analyzed to ensure robust and secure deployments.

6. CONCLUSION

The experimental results revealed that OverlayFS consistently outperformed AUFS and Btrfs in read-heavy workloads, such as web server I/O, due to its efficient merging mechanism and lightweight design. Its low memory usage further makes it an ideal choice for memory-constrained environments. On the other hand, Btrfs demonstrated superior performance in write-heavy and data-intensive workloads, such as machine learning tasks, leveraging its advanced features like copy-on-write and dynamic disk allocation. However, its higher CPU and memory consumption may limit its applicability in resource-constrained scenarios. AUFS, while functional, lagged behind the other file systems in most performance metrics, highlighting its diminishing relevance in modern containerized environments. Additionally, the choice of Docker storage driver significantly impacted performance, with the overlay2 driver consistently outperforming aufs across all workloads.

Despite its contributions, this study has certain limitations. The experiments were conducted using a specific set of workloads—database operations, web server I/O, and machine learning tasks—which may not fully represent the diverse range of applications running in containerized environments. Additionally, the tests were performed on a single hardware configuration, and performance may vary across different setups, such as those with slower storage devices or limited CPU resources. Furthermore, the study focused on OverlayFS, AUFS, and Btrfs, leaving out other file systems like ZFS and XFS, which could offer additional insights.

In conclusion, the performance of file systems is a critical factor in the efficiency and scalability of containerized applications. This study underscores the importance of selecting and configuring file systems based on workload requirements and resource constraints. By leveraging the insights and recommendations presented in this research, system administrators and DevOps engineers can optimize Docker deployments for improved performance, stability, and resource utilization. As containerization continues to evolve, further research and innovation in storage optimization will remain essential to meet the growing demands of modern applications.

REFERENCES

[1] W. Felter et al., "An Updated Performance Comparison of Virtual Machines and Linux Containers," *IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, pp. 171-172, 2015, doi: 10.1109/ISPASS.2015.7095802.

[2] A. P. Ferreira and R. Sinnott, "A Performance Evaluation of Containers Running on Managed Kubernetes Services," *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Sydney, NSW, Australia, 2019, pp. 199–206, doi: 10.1109/CloudCom.2019.00038.

[3] V. Tarasov et al., "In Search of the Ideal Storage Configuration for Docker Containers," *2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems (FASW)*, Tucson, AZ, USA, 2017, pp. 199–206, doi: 10.1109/FASW.2017.148.

[4] I. Cilic, P. Krivic, I. Podnar Zarko, and M. Kusek, "Performance Evaluation of Container Orchestration Tools in Edge Computing Environments," *Sensors*, vol. 23, no. 8, p. 4008, Apr. 2023, doi: 10.3390/s23084008.

[5] Y. Chen et al., "PeakFS: An Ultra-High Performance Parallel File System via Computing-Network-Storage Co-Optimization for HPC Applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 12, pp. 2578-2595, Dec. 2024, doi: 10.1109/TPDS.2024.3485754.

[6] W. A. Bhat, "Performance-Baseline Estimation of File System Operations for Linux-Based Edge Devices," *IEEE Trans. Ind. Informat.*, vol. 20, no. 5, pp. 7537-7544, May 2024, doi: 10.1109/TII.2024.3363090.

[7] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi, "Improving I/O Performance in Container with OverlayFS," *2018 IEEE Int. Conf. Big Data (Big Data)*, Seattle, WA, USA, 2018, pp. 5395-5395, doi: 10.1109/BigData.2018.8622479.

[8] V. Tarasov, L. Rupprecht, D. Skourtis, et al., "Evaluating Docker storage performance: from workloads to graph drivers," *Cluster Comput.*, vol. 22, pp. 1159–1172, 2019, doi: 10.1007/s10586-018-02893-y.

[9] N. Zhao et al., "Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 918-930, Apr. 1, 2021, doi: 10.1109/TPDS.2020.3034517.

APPENDIX A

```
#!/bin/bash
# setup_environment.sh
# Configures Docker and mounts a base filesystem (e.g., ext4/btrfs) for storage drivers.

# Exit on error and log all commands
set -e
set -x

# Install Docker dependencies
sudo apt-get update -y
sudo apt-get install -y ca-certificates curl gnupg lsb-release

# Add Docker's GPG key
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

# Configure Docker repository
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/
docker.list > /dev/null

# Install Docker
sudo apt-get update -y
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

# Verify Docker
docker --version || { echo "Docker installation failed"; exit 1; }

# Format base device (e.g., ext4, btrfs)
DEVICE="/dev/nvme0n1"
FS_TYPE="ext4"
sudo mkfs.${FS_TYPE} -f $DEVICE  # Force format

# Mount device
MOUNT_POINT="/mnt/test"
sudo mkdir -p $MOUNT_POINT
sudo mount $DEVICE $MOUNT_POINT || { echo "Mount failed"; exit 1; }

# Configure Docker storage driver (overlay2/aufs/btrfs)
STORAGE_DRIVER="overlay2"
sudo mkdir -p /etc/docker
echo "{\"storage-driver\": \"$STORAGE_DRIVER\"}" | sudo tee /etc/docker/daemon.json > /dev/null

# Restart Docker
sudo systemctl restart docker || { echo "Docker restart failed"; exit 1; }
echo "Environment setup complete."
```

Listing 1. The script for setting up the test environment

```
#!/bin/bash
# run_fio_benchmarks.sh
# Measures I/O performance using Fio with direct I/O (bypassing cache).

TEST_DIR="/mnt/test/fio_tests"
mkdir -p $TEST_DIR

# Random Reads (4K blocks)
fio --name=random-read --directory=$TEST_DIR --ioengine=libaio --rw=randread \
    --bs=4k --size=1G --numjobs=4 --runtime=60 --time_based --group_reporting \
    --output=random-read-results.txt --direct=1

# Sequential Writes (64K blocks)
fio --name=sequential-write --directory=$TEST_DIR --ioengine=libaio --rw=write \
    --bs=64k --size=1G --numjobs=4 --runtime=60 --time_based --group_reporting \
    --output=sequential-write-results.txt --direct=1

# Mixed workload (70% reads, 30% writes)
fio --name=mixed-io --directory=$TEST_DIR --ioengine=libaio --rw=randrw \
    --bs=4k --size=1G --numjobs=4 --runtime=60 --time_based --group_reporting \
    --rwmixread=70 --output=mixed-io-results.txt --direct=1

echo "Fio benchmarks completed."
```

Listing 2. The script for automating the execution of Fio benchmarks

```
#!/bin/bash
# run_sysbench_db.sh
# Simulates database transactions with MySQL.

DB_NAME="testdb"
DB_USER="root"
DB_PASSWORD="password"
TABLE_SIZE=10000
THREADS=4
DURATION=60

# Prepare database
sysbench oltp_read_write --table-size=$TABLE_SIZE --db-driver=mysql \
    --mysql-host=localhost --mysql-user=$DB_USER --mysql-password=$DB_PASSWORD \
    --mysql-db=$DB_NAME prepare

# Run benchmark
sysbench oltp_read_write --table-size=$TABLE_SIZE --db-driver=mysql \
    --mysql-host=localhost --mysql-user=$DB_USER --mysql-password=$DB_PASSWORD \
    --mysql-db=$DB_NAME --threads=$THREADS --time=$DURATION run > sysbench-results.txt

# Cleanup (even if the test fails)
sysbench oltp_read_write --table-size=$TABLE_SIZE --db-driver=mysql \
    --mysql-host=localhost --mysql-user=$DB_USER --mysql-password=$DB_PASSWORD \
    --mysql-db=$DB_NAME cleanup || true

echo "Sysbench database workload completed."
```

Listing 3. The script for automating the setup and execution of Sysbench database

```bash
#!/bin/bash
# monitor_docker_stats.sh
# Collects Docker container stats every 2 seconds for 60 seconds.

CONTAINER_ID=$1
OUTPUT_FILE="docker-stats-results.txt"

# Validate input
if [ -z "$CONTAINER_ID" ]; then
  echo "Usage: $0 <container_id>"
  exit 1
fi

# Header
echo "Timestamp,CPU %,Memory Usage,Memory %,Disk Read,Disk Write,Network I/O" > $OUTPUT_FILE

# Collect stats every 2 seconds for 1 minute
for _ in {1..30}; do
  docker stats --no-stream --format '{{json .}}' $CONTAINER_ID | \
  jq -r '[.CPUPerc, .MemUsage, .MemPerc, .BlockIO, .NetIO] | @csv' \
  >> $OUTPUT_FILE
  sleep 2
done

echo "Docker stats saved to $OUTPUT_FILE."
```

Listing 4. The script that uses Docker stats to monitor resource usage during benchmarks

```bash
#!/bin/bash
# aggregate_results.sh
# Combines benchmark results into a single file.

OUTPUT_FILE="benchmark-results-summary.txt"

# Check if result files exist
for file in random-read-results.txt sequential-write-results.txt mixed-io-results.txt sysbench-results.txt
docker-stats-results.txt; do
  if [ ! -f "$file" ]; then
    echo "Error: $file missing!"
    exit 1
  fi
done

# Aggregate results
echo "=== Fio Benchmarks ===" > $OUTPUT_FILE
cat random-read-results.txt sequential-write-results.txt mixed-io-results.txt >> $OUTPUT_FILE

echo -e "\n=== Sysbench Database Results ===" >> $OUTPUT_FILE
cat sysbench-results.txt >> $OUTPUT_FILE

echo -e "\n=== Docker Resource Usage ===" >> $OUTPUT_FILE
cat docker-stats-results.txt >> $OUTPUT_FILE

echo "Results aggregated into $OUTPUT_FILE."
```

Listing 5. The script that aggregates results from multiple benchmarks into a single file

```bash
#!/bin/bash
# cleanup_environment.sh
# Resets the environment by removing containers, unmounting devices, and resetting Docker.

# Force-stop and remove all containers
docker rm -f $(docker ps -aq) 2>/dev/null || true

# Unmount test device
MOUNT_POINT="/mnt/test"
sudo umount -l $MOUNT_POINT 2>/dev/null || true
sudo rm -rf $MOUNT_POINT

# Reset Docker configuration
sudo rm -f /etc/docker/daemon.json
sudo systemctl restart docker

echo "Cleanup complete."
```

Listing 6. The script that cleans up the environment after the tests are completed