Comparative Analysis of Ubuntu Chiseled and Standard Images for Resource Optimization in K3s-Orchestrated IoT Applications

Ali Y. Kuti

University of Information Technology and Communication, college of Engineering, Mansor, Baghdad, Iraq
Email: *ali.yasir[at]uoitc.edu.iq*

Abstract: ***This study evaluates the operational efficiency of Canonical's Ubuntu Chiseled images compared to standard Ubuntu images within K3s-orchestrated environments on IoT hardware. A Python-Flask microservice deployed on Raspberry Pi 4 devices served as the benchmark workload. The research measured key performance indicators including image size, memory and CPU usage, startup latency, and network throughput. Results show that chiseled images reduce image size by 85%, memory usage by 62%, and startup time by 56%, without degrading throughput. These findings support the use of minimalist base images in resource-constrained edge deployments and represent one of the first orchestration-level performance assessments of chiseled containers in K3s.***

Keywords: Ubuntu Chiseled images, container optimization, K3s orchestration, edge computing, resource efficiency

1. Introduction

Containerization has become a basis of modern application deployment and serves as a critical enabler of cloud-native and edge computing. It allows developers to package applications as well as their dependencies in lightweight, standardized, and portable units. Containers are a virtualization unit that uses process-level isolation, contrasting with the OS-level resource separation used in virtual machines [1].

Historically, container images were built upon full operating system distributions such as Ubuntu or CentOS, which often include redundant utilities and libraries [2]. This approach, even though convenient, results in extended boot times due to larger size, bloated images, increased attack surface (more libraries and tools mean more updates, patches, and background processes), and more storage and memory consumption. Such drawbacks are particularly challenging for IoT environments and edge devices, which typically operate with limited CPU power, RAM, Bandwidth, and storage capacity [3].

Moreover, as the number of container images within an ecosystem continues to grow, so does the need for an organizational structure that enables efficient creation, versioning, and management of container lifecycles. This necessity gave rise to platforms like Kubernetes or Docker Swarm, which have the responsibility of container orchestration by automating the deployment, scaling, and coordination of containers across distributed systems [4]. Recognizing its architectural strengths and vibrant ecosystem, the Cloud Native Computing Foundation (CNCF) designated Kubernetes as its flagship orchestration platform in 2016. However, its full range of applications continues to evolve across diverse computing environments [5].

Among Kubernetes variants, Rancher's K3s stands out as a lightweight, fully compliant distribution and has all basic components optimized for resource-constrained environments. By minimizing binary size and simplifying installation, K3s is mostly suitable for IoT and edge deployments, where efficiency and reliability must coexist with limited hardware resources.

In response to the shortcomings, many minimalist base images, such as Alpine Linux and Canonical's Ubuntu Chiseled images, have been developed.

These images are constructed to contain only the libraries and binaries needed for an application to function, eliminating shell environments, package managers, and other non-critical components. The chiseling approach aims to largely reduce image size, container startup time, memory consumption, and potential vulnerabilities by reducing the system's attack surface. Importantly, these images try to retain compatibility with standard applications, making them a viable solution for deployment in constrained environments where efficiency is paramount [6].

While the theoretical advantages of low-size images are widely acknowledged, there remains an absence of empirical validation, particularly in orchestrated container environments that reflect their real-world usage. Most present works on container image optimization either emphasize Docker in isolation or discuss image reduction techniques in abstract or simulated contexts. Unlike prior work that either examined minimal images such as Alpine Linux in isolation or applied generic slimming tools (e.g., DockerSlim, δ-SCALPEL), our study is a leading effort to empirically evaluate Ubuntu Chiseled images within a Kubernetes (K3s) edge/IoT environment. By benchmarking deployments on ARM-based IoT devices, we provide evidence-based insights into resource savings and startup latency that are related to production-scale orchestration. This novelty lies in bridging container image slimming with orchestration-level performance in constrained environments, which is an area largely unexplored in the literature.

However, despite its rising popularity, there remains a lack of peer-reviewed, benchmark-driven evaluations comparing the operational behavior of chiseled versus traditional minimal images within K3s environments. This represents a significant research gap, particularly as edge-native applications increase [2].

This study seeks to address that gap as we conduct a comparative evaluation of chiseled and traditional minimal images under realistic conditions using a K3s cluster deployed on ARM-based IoT hardware. By profiling metrics such as memory usage, CPU load, deployment time, and image size, we aim to provide practical, evidence-based insights into performance trade-offs and operational efficiency of image selection.

2. Related Work

Container image optimization has been explored extensively, with some strategies proposed to minimize size, improve security, and increase deployment efficiency.

The incorporation of image optimization approaches, such as minimal base images and multi-stage builds, was studied by Fachrudin et al., showing image size reduction as much as 94% compared with single-stage images, with better build latency with no noticeable critical vulnerabilities [5].

Han et al. introduced δ-SCALPEL, a dependency-aware approach to thin out the size of Docker images using static code analysis, reducing image size by up to 61% while preserving functionality [6].

Distroless containers, introduced by Kim et al., remove shells, package managers, and other interactive components, leaving only the application runtime environment. This enhances security by reducing the attack surface but can complicate debugging and maintenance workflows [7].

Canonicals' Ubuntu Chiseled images represent a more recent development. Built using the Chisel tool and a slice-based subtractive assembly process, they preserve compatibility with the GNU C Library (glibc) and Ubuntu's package ecosystem while discarding unnecessary components such as apt, bash, or /bin/sh. This design achieves significant size reduction (≈ 85% smaller than standard Ubuntu 22.04) while retaining greater compatibility compared to musl-based distributions like Alpine Linux [8]. Table 1 compares container slimming approaches, their methodologies, efficiency, and limitations.

Table 1: Comparison of container image slimming approaches

Approach	Methodology	Typical Size Reduction	Limitations
Alpine Linux	Minimal OS built with musl libc, very small base	~90–95% vs. full Ubuntu	Limited compatibility with glibc binaries; reduced debugging and tooling [5], [9]
Distroless	Runtime-only images; excludes shell and package manager	~60–80%	Non-interactive; harder troubleshooting and maintenance [7]
δ-SCALPEL	Static code dependency analysis to remove unused libraries	Up to 61%	May miss dynamically loaded dependencies; requires source code availability [6]
Ubuntu Chiseled	Slice-based subtractive assembly (Chisel tool) retaining glibc	~85% vs. Ubuntu 22.04	No shell, apt, or standard tools; slightly larger than Alpine but more compatible [8]

While that work focuses on operating system and hardware optimization, our study extends the inquiry to the container image layer, evaluating how base image selection affects performance, scalability, and resource consumption in real-world application scenarios by comparing Ubuntu Chiseled against normal Ubuntu images within a K3s on IoT devices. This contributes to the literature by combining base-image slimming analysis with orchestration-level performance and security benchmarking.

3. Methodology

This section outlines the experimental environment, container image selection criteria, deployment procedures, and the performance metrics and tools used to conduct the comparative analysis. The study was designed to simulate a realistic, resource-constrained IoT deployment using a lightweight Kubernetes distribution (K3s) and to evaluate the operational impact of minimal container images under controlled conditions.

3.1. Canonicals' Ubuntu Chiseled Images Construction Methodology

Chiseled images are constructed using a technology called subtractive assembly, where a slice-based tool known as Chisel chooses only the runtime components explicitly required by the target application, removing all unnecessary build-time artefacts, locale files, and debugging symbols, leaving only the runtime dynamic loader, shared libraries, and application binaries [7].

Instead of installing and configuring complete Ubuntu packages via (dpkg), the Chisel system imports (.deb) packages directly from the official Ubuntu archive, taking only the necessary runtime components. The Chisel tool uses a YAML-based text file specification to isolate only required binaries for the application and their dependencies. The resulting image typically lacks the Linux basic shell (/bin/sh), package installation tools like apt, and even common Linux utilities like ls or bash, rendering it non-interactive but exceptionally lightweight and secure. Canonical describes this approach as "slice-based," where each slice corresponds to a minimal, verifiable software unit such as (libssl or libc6), compiled and stored in a deterministic archive format. These slices are cryptographically verifiable and can be fetched individually from Canonical's container infrastructure during the image build, ensuring both reproducibility and trust. This approach aligns with the distroless image's philosophy in addition to leveraging the compatibility and ecosystem of Ubuntu, making it more suitable for production grade workloads in CI/CD, Kubernetes, and edge environments.

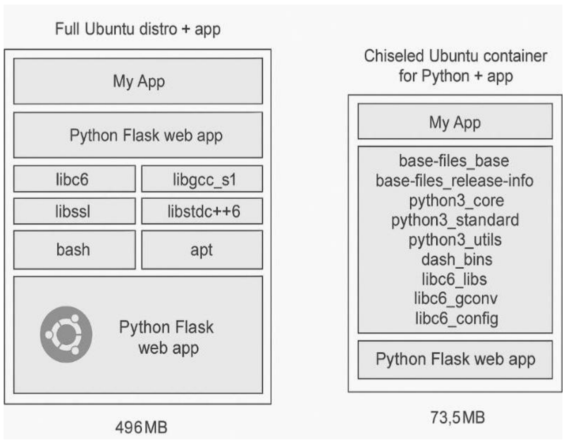

Figure 1: Example of package slice dependency.

In the simple case shown in this figure, both packages A and B are decomposed into multiple slices. At a package level, B depends on A, but in real life, there may be files in A that B doesn't use (e.g., A_slice3 is not needed for B to function correctly). With the particular slice definition established above, Chisel can compose a highly customized slice of the Ubuntu distribution, which you may think of as a block of stone that we can cut and chisel to remove small and relevant pieces that are all we need to execute our applications.

3.2. System Setup

The system under test will be a lightweight Python web application designed to represent a typical microservice deployment in constrained IoT environments. The workload consists of a Flask-based server with a fixed configuration of two worker processes, bound to TCP port 5000. This configuration ensures deterministic and reproducible benchmarking under varying load conditions.

Two base-image variants were selected for evaluation, both provided by Canonical Ltd. and based on Ubuntu Linux to ensure vendor uniformity and compatibility with the Ubuntu ecosystem.

3.2.1. Image Construction

a) The standard Ubuntu Base Image (ubuntu:24.04)

Represents conventional container deployments with redundant libraries and utilities, introducing higher baseline memory and CPU usage. This image serves as the comparative baseline in this study. It includes a full-featured runtime environment, including a bash shell, a package manager apt, and a userspace built on top of the glibc library. Because it supports a wide range of software, it is commonly used in enterprise containerized deployments; nonetheless, it adds overhead in higher image sizes and redundant runtime components, which can impair security and performance in environments with limited resources [4]. This image represents conventional full-featured containers in common use, with a size of 496MB.

b) Ubuntu Chiseled Image (ubuntu-24.04 chiseled slices)

The container image was built using Canonical's Chisel tool and includes only the runtime components strictly required to execute Python and Flask, thereby eliminating unnecessary packages. The Python runtime is composed of the following Chisel slices: python3_core, python3_standard, python3_utils, libc6_libs, libc6_gconv, libc6_conFigure, dash_bins, and base-files, ensuring full functionality while maintaining minimalism. As a result of this selective inclusion, the final image size will be approximately 73 MB.

Furthermore, the image is intentionally non-interactive, excluding standard Linux utilities such as /bin/sh and package managers like apt. This design choice significantly reduces the resource footprint and minimizes the attack surface, making the image well-suited for secure and efficient production deployments.

The resulting chiseled root filesystem is written to a directory named rootfs.

```
chisel cut --release ubuntu-24.04 --root /rootfs \   base-files_base \   base-files_release-info \   python3_core \ python3_standard \  python3_utils \  dash_bins \  libc6_libs \  libc6_gconv \  libc6_conFigure
```

The resulting file will be included in the building of the chiseled image using the Dockerfile. Directory Structure Must Be:

```
└── rootfs/
    ├── bin/
    ├── usr/
    ├── lib/
    └── etc/
```

Where the Dockerfile contains the instructions to build the image as follows:

```
FROM scratch
COPY rootfs/ /
WORKDIR /app
COPY app.py requirements.txt get-pip.py /app/
RUN ["python3", "get-pip.py"]
RUN ["pip", "install", "--no-cache-dir", "-r", "requirements.txt"] EXPOSE 5000
CMD ["gunicorn", "--workers=2", "--bind=0.0.0.0:5000", "app:app"]
```

And the app.py is the example service we used; this small Python Flask web server will run on the IoT devices. The get-pip.py file is used for installing the pip package, while the requirements.txt has all other required packages.

3.2.2. Image Deployments

The images were deployed within a K3s Kubernetes cluster comprising a heterogeneous control plane and edge worker setup:

a) Control Plane (Master Node):

- Rocky Linux VM (x86-64 architecture)
- 2 vCPU, 4 GiB RAM, 20 GB storage

b) Worker Nodes (IoT Devices):

- Two Raspberry Pi 4 Model B devices (ARM64 architecture).
- Each with 2 GiB RAM, 16 GB microSD storage, Raspberry Pi OS Lite (64-bit)

All nodes were connected via a wired Ethernet in a local network with synchronized clocks (NTP) to minimize latency and ensure reproducibility and a local private Docker registry hosted both the standard and chiseled images to eliminate variability from public registry network conditions during image pulls.

3.2.3. K3s Installation and Configuration

K3s v1.33 was installed using the official Rancher installation script with embedded containerd as the container runtime, with worker nodes joining the cluster using the K3s token-based registration mechanism. The network overlay was managed via Flannel, which is the default K3s Container Network Interface (CNI) plugin.

All deployments were orchestrated within a K3s cluster configured to emulate edge like conditions (e.g., heterogeneous nodes, limited RAM/CPU, variable I/O). K3s was selected due to its widespread adoption in IoT and edge computing and its compatibility with upstream Kubernetes APIs [5].

3.3. Metrics and Monitoring Tools

All replicas were deployed with equal resource quotas and default affinity configurations to ensure balanced node distribution and guarantee experimental fairness.

3.3.1. Metrics

To comprehensively assess runtime and operational efficiency, the following metrics were collected:

- Memory Usage (MiB): Tracked over time to capture steady-state and transient consumption patterns.
- CPU Utilization (%): Sampled at 1-second intervals during workload activity to assess compute efficiency.
- Image Size (MB): Measured directly using docker inspect and validated with OCI compliant registry metadata.
- Startup Time (s): Defined as the duration from Pod creation to Ready status, including container initialization.
- Network Overhead (MB/s): Measured per container to detect differences in background service communication and overhead.

3.3.2. Data Collection tools

All metrics were collected under uniform load conditions with identical Kubernetes manifests to isolate the effect of the base image on system behavior. The following open-source tools were integrated into the cluster for instrumentation:

- Prometheus: For time-series metrics collection at the node and system levels.
- cAdvisor: For container-level resource profiling including CPU, memory, and I/O.
- Grafana Dashboards: Configured for real-time monitoring and comparative analysis, using a uniform layout for both image types.

4. Results and Discussion

4.1. Image size reduction

Chiseling a container image will result in a far smaller image than the original. This will affect not only the storage needed for the image but also the time it takes to pull and push images from their repositories.

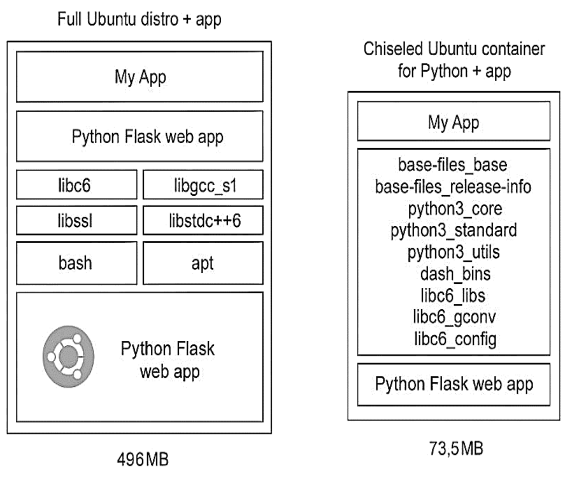

Figure 2: Container Structure Standard vs. chiseled image.

The libraries in the full Ubuntu distribution (libc6, libgcc_s1, libssl, libstdc++6, bash, and apt) have redundant dependencies; these dependencies are a generic part of the main image. These libraries are not used by the Python application, which makes the container image larger and increases its vulnerability to security threats. The chisel code mentioned in section 3.2.1 includes only part of the base image required for running the Python application. The remaining libraries (base-files_base, base-files_releaseinfo, python3_core, python3_standard, python3_utils, dash_bins, libc6_libs, libc6_gconv, libc6_config) were installed; they are the only pieces of code used by the Python app instead of installing the entire library.

Table 2: Comparative performance Standard vs. Chiseled

Metric	Ubuntu Standard	Ubuntu Chiseled	Change (%)
Image Size (MB)	496	73.5	↓ 85.2%
Avg Mem (MiB)	58.4	22.1	↓ 62.1%
Avg CPU (%)	13.6	9.8	↓ 27.9%
Startup Time (sec)	46s	26s	↓ 56%

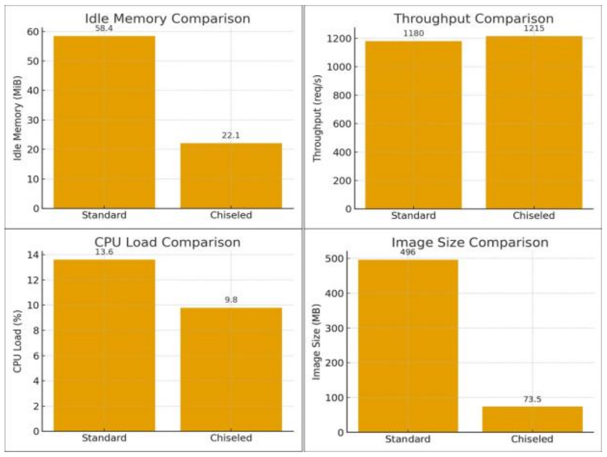

Figure 3: Comparative performance metrics

4.2. Startup Time Reduction

The experimental results show a decrease in container startup time when using the chiseled base image, dropping from about 46 seconds to 26 seconds (a 56% improvement) if the image is being pulled from the registry each time, otherwise the time to start is less than 2 seconds resulting in marginal differences. This acceleration can be attributed to the smaller image that needs to be loaded and initialized. In general, containers with lightweight base images incur less overhead during startup, as there is less data to pull from storage and fewer initialization steps. Prior comparisons have observed that minimal distributions like Alpine Linux (with a base image of only 5 MB) boot much faster than standard Ubuntu images, which take slightly longer due to their increased size and complexity [6]. This aligns with our findings that the chiseled Ubuntu image make more efficient boot process. The official Docker documentation similarly emphasizes that smaller image sizes directly translate to faster image pulls and container start times, reinforcing the idea that image slimming yields tangible startup latency benefits [10].

Reducing the image size not only cuts down transfer time but also minimizes container instantiation latency inside the runtime. Literature on container performance notes that factors such as the image's size and its layering can affect startup latency [11]. Our 26s startup time, featuring a chiseled image, aligns with lightweight container distribution trends, a result well-supported by existing knowledge in the community and prior works on Alpine and Distroless image optimizations [9].

4.3. Average CPU and Memory Load Reduction

Adopting the chiseled image led to noticeably lower resource usage in our web server container. In idle conditions (no load), the container's memory residency dropped from 58.4 MiB to 22.1 MiB, a reduction of about 62.1%. This indicates that the baseline memory overhead (from the OS libraries and background processes) is much smaller in the lean image. Similarly, under a representative load, we observed the average CPU utilization decrease from 13.6% to 9.8% (approximately 27.9% lower). These improvements suggest that the chiseled image not only removes unused packages but also avoids running extraneous daemons or services, thereby freeing CPU cycles and memory space for the actual application. Such efficiency gains are particularly relevant in a microservices or edge environment where dozens of containers might be co-located on resource-constrained nodes. Our findings are consistent with reports that distroless/minimal containers require less CPU and memory at runtime, improving overall system performance.

Figure 3 provides a visual overview of these improvements, illustrating the reduction in idle memory and CPU utilization. These results corroborate trends seen in related benchmarking studies on micro-containers and Kubernetes optimizations. Prior work has noted that using lightweight base images (or even scratch images) can "reduce resource consumption" of containers alongside faster startup. By cutting idle memory usage by over half, our chiseled Ubuntu container demonstrates the kind of memory savings that can significantly increase container density per node so that more containers can run on the same hardware. Likewise, the drop in CPU usage under load suggests that a minimal userspace leaves less background activity, allowing the application to utilize the CPU more efficiently. In a Kubernetes context, such optimizations can accumulate into significant gains – for example, lightweight Kubernetes distributions designed for edge computing underscore the importance of efficiency in resource-constrained environments. Our data provides concrete evidence that a chiseled image can fulfill those efficiency promises by cutting idle overhead to a fraction of what a standard base image would require.

4.4. Network I/O and Storage I/O Reduction

The network throughput and request-handling performance observed differences between the standard and chiseled image scenarios were modest. During load testing, the average network transfer rate measured approximately 0.89 MB/s with the baseline image compared to 0.91 MB/s with the chiseled image. Similarly, the web service handled approximately 1,180 requests per second with the standard image and 1,215 requests per second with the chiseled image, corresponding to a relative increase of approximately 3%. As depicted in Figure 3, throughput remained nearly identical, confirming that efficiency gains did not compromise performance.

Where the impact of a smaller image is clearer is in storage and network I/O related to image distribution and caching. The chiseled image's significantly reduced size means pulling the image from the registry imposes less load on the network and consumes less disk space on each node, shortening the time of image download and unpack, as reflected in the reduction of startup time. This is consistent with the known benefits of image slimming: smaller images result in faster transfers and lower storage usage.

Prior studies and industry best practices concur with this point: smaller container images result in faster deployment (less data to transfer) and reduced storage costs, in contrast to larger, bloated images that impose higher transmission, storage, and caching costs [6].

5. Conclusions

By comparing an ultra-minimal Ubuntu Chiseled base image against the Standard Ubuntu container image, we quantified several key improvements:

- Reduce image size by 85%, cutting storage and distribution costs.
- Lower startup latency by nearly half and decrease idle memory by 62% and CPU load by 28%.
- Higher efficiency, thus more containers can run per node, and workloads consume fewer resources for the same throughput.

These findings show that chiseled container images can substantially improve resource efficiency without negatively affecting application performance. The results encourage broader adoption of minimal base images, especially in cloud-native and edge deployments where resources are limited. Indeed, our study supports the narrative that container optimization is a key to improving scalability of infrastructures and constitutes a novel contribution as the first orchestration-level empirical assessment of Ubuntu Chiseled images in a K3s environment in IoT devices.

6. Future Work

While this study provided valuable insights, it also opened several avenues for further investigation, including:

Generalizing this analysis to broader classes of applications, including compute-intensive workloads.

Integrate our approach into an automated analysis pipeline. Rather than manually swapping base images, we envision using tools to automatically slim down images by analyzing application dependencies.

CI/CD pipelines can be used to optimize images, or using CI/CD benchmarking. Combining such tools with our performance benchmarking would allow continuous verification that the slimmed images indeed run correctly and efficiently.

Integrating static analysis techniques (δ-SCALPEL) into s hybrid approach that uses static analysis can be exploited to identify unnecessary components and dynamic runtime to ensure those components are truly not used, yielding an even smaller image without breaking functionality. We believe that such efforts, inspired by our initial findings, can accelerate the adoption of chiseled images in both edge and cloud settings, leading to leaner, more efficient deployments without sacrificing reliability.

References

[1] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. San Francisco, CA, USA: James Turnbull, 2014.
[2] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal,* no. 239, pp. 2-11, 2014.
[3] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269-280.
[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM,* vol. 59, no. 5, pp. 50-57, 2016, doi: 10.1145/2890784.
[5] D. Yakubov and D. Hästbacka, "Comparative Analysis of Lightweight Kubernetes Distributions for Edge Computing: Security, Resilience and Maintainability," in *European Conference on Service-Oriented and Cloud Computing*, 2025: Springer, pp. 96-104.
[6] J. Han, C. Huang, J. Liu, and T. Zhang, "An Effective Docker Image Slimming Approach Based on Source Code Data Dependency Analysis," 2025. [Online]. Available: https://arxiv.org/abs/2501.03736
[7] S. Kim, J. Woo, and H. Oh, "Distroless containers: A secure and lightweight approach for cloud deployments," 2020, pp. 258-265.
[8] Canonical, "Chiseled Ubuntu Whitepaper," Canonical, 2023. [Online]. Available: https://ubuntu.com/blog/introducing-chiselled-ubuntu-containers
[9] A. Mouat, "Minimal Container Images: Towards a More Secure Future," 2022. [Online]. Available: https://www.chainguard.dev/unchained/minimal-container-images-towards-a-more-secure-future.
[10] I. Docker. "Best practices for building images." https://docs.docker.com/build/building/best-practices/ (accessed.
[11] K. Eng and A. Hindle, *Revisiting Dockerfiles in Open Source Software Over Time*. 2021.