BLAFS: A Bloat-Aware Container File System

Citation for the original published paper (version of record):
Zhang, H., Alhanahnah, M., Leitner, P. et al (2026). BLAFS: A Bloat-Aware Container File System. Socc 2025 Proceedings of the 2025 ACM Symposium on Cloud Computing: 614-628.
http://dx.doi.org/10.1145/3772052.3772263

(article starts on next page)

RESEARCH-ARTICLE

BLAFS: A Bloat-Aware Container File System

HUAIFENG ZHANG, Chalmers University of Technology, Gothenburg, Vastra Gotaland, Sweden

MOHANNAD ALHANAHNAH, University of Wisconsin-Madison, Madison, WI, United States

PHILIPP LEITNER, Chalmers University of Technology, Gothenburg, Vastra Gotaland, Sweden

AHMED ALI-ELDIN, Chalmers University of Technology, Gothenburg, Vastra Gotaland, Sweden

Open Access Support provided by:

University of Wisconsin-Madison

Chalmers University of Technology

Published: 19 November 2025

Citation in BibTeX format

SoCC '25: ACM Symposium on Cloud Computing
November 19 - 21, 2025
Online, USA

Conference Sponsors:
SIGOPS
SIGMOD

BLAFS: A Bloat-Aware Container File System

Huaifeng Zhang
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
huaifeng@chalmers.se

Mohannad Alhanahnah
University of Wisconsin-Madison
Madison, USA
mohannad.alhanahnah@gmail.com

Philipp Leitner
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
philipp.leitner@chalmers.se

Ahmed Ali-Eldin
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
ahmed.hassan@chalmers.se

Abstract

Containers have become the standard for deploying applications in many cloud systems due to its convenience. However, this convenience leads to significant container bloat, i.e., unused files that inflate container image sizes, increase provisioning times, waste resources and introduce security vulnerabilities. Bloat is particularly problematic in serverless and edge computing scenarios, where resources are constrained, and performance is critical, and for microservice applications where rapid scaling is key to meet performance targets. However, existing container debloating tools are often limited in both effectiveness and robustness. In this paper, we propose BLAFS, a bloat-aware container filesystem that removes bloat while guaranteeing the correct operation of the debloated containers. BLAFS addresses bloat at the filesystem level by introducing new layers in the filesystem to enable debloating. During runtime, accessed files are moved to the debloating layers, and then similar to garbage collection mechanisms, BLAFS removes files that are not accessed during runtime. An optional reloading layer fetches files from a remote cloud cache on-demand if the files are mistakenly removed. We discuss how BLAFS can be used in different deployment scenarios and for different use-cases including container security-hardened and a dynamic deployment mode where the target is improved provisioning performance. We evaluate BLAFS performance using the top 20 downloaded containers from DockerHub, four ML containers, and SEBS, a Serverless Benchmark containing 10 serverless functions and compare its performance against two state-of-the-art debloating tools. Our evaluation shows that BLAFS reduces container sizes by up to 95% and cold-starts by up to 68%. In the security-hardened mode, BLAFS removes up to 89% of CVEs while the two state-of-the-art debloating tools fail on most of the workloads. We identify their limitations, and show how BLAFS provides a more principled approach to debloating. Additionally, when combined with lazy-loading snapshotters, BLAFS improves provisioning efficiency, reducing conversion times by up to 93% and provisioning times by up to 19%.

CCS Concepts

• **Software and its engineering** → *Software evolution.*

Keywords

Software Bloat, Cloud Computing, Serverless Computing, Container, File System

ACM Reference Format:
Huaifeng Zhang, Mohannad Alhanahnah, Philipp Leitner, and Ahmed Ali-Eldin. 2025. BLAFS: A Bloat-Aware Container File System. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3772052.3772263

1 Introduction

Containers have become the main deployment model for cloud workloads including traditional workloads [11, 42], machine learning [24, 31], serverless functions [9, 18], and edge deployments [14]. Container's popularity is driven by how the technology simplifies deployments. Containerized applications are distributed as container images, composed of compact, and shareable "layers" of files. These images are then stored in centralized registries, such as DockerHub [20]. A user pulls a container from a registry, and then can deploy, update, or create a new image based on the pulled image

Despite their advantages, container images have become excessively large [51, 55]. For example, it has been shown that most container images often package unnecessary files and libraries [51, 55]. This is commonly referred to as software or container bloat. Container bloat is a result of software development practices that favor rapid feature integration and dependency inclusion, even when these components are only marginally useful to end users [8, 54]. Container bloat leads to performance degradation [45], increased provisioning times [14], wasted storage and network bandwidth [54], and an expanded attack surface [13, 30].

To address these inefficiencies, prior work proposed optimizations for storage reduction [26], faster provisioning [14, 16], and improved security [23, 32]. However, most of these solutions do not address the root-cause of the problem, container bloat. For example, lazy-loading snapshotters [14, 16] speed up container startup by converting the container images into a lazy-loading format which enables the container to start before all the layers are pulled from the registry or source. However, when a container image is bloated, converting it into a lazy-loading format becomes costly, in many cases, offsetting performance benefits. In addition, snapshotters transfer the entire container, only lazily. Hence, they do not fundamentally remove the bloat.

Another less studied solution is container debloating, i.e., removing unused files from containers. For example, Cimplifier [44] and SlimToolKit [21], rely on file-tracing tools, such as `ptrace` or `strace` [5, 7], to identify and remove unused files. However, they lack awareness of the underlying filesystem structure, which limits their ability to handle more complex bloat scenarios and often result in broken debloated containers that do not function [27]. These limitations are inherent in these tools as we describe later.

We argue that the root cause of bloat is the *container layered filesystem architecture* itself; Containers are built on top of other containers often referred to as *base container images*. To facilitate this, containers rely on layered filesystems such as the UnionFS [43], the OverlayFS [38], or the Btrfs [46]. In a layered filesystem, a container inherits all the filesystem layers of its base image, creating a new layer on top with its own files. Any files (and bloat) in the base image is inherited in the new container, with possibly new bloat added with each new layer added. As a result, bloat compounds across layers and propagates transitively.

To solve this filesystem bloat, in this paper we introduce BLAFS, a *Bl*oat-*A*ware container *F*ile *S*ystem that leverages the layered structure of container filesystems to debloat containers during runtime. BLAFS introduces two new types of layers to container filesystems: a *debloating layer* and a *reloading layer*. BLAFS converts a container image to a BLAFS-image by inserting a combination of these new layers to the container image.

At runtime, BLAFS transparently migrates any accessed files from the original filesystem layers to the debloating layers. Subsequently, the original layer is pruned by removing files that remain unaccessed, effectively identifying and eliminating bloat. Debloating layers serve as composable primitives enabling a flexible container filesystem architecture that can be tailored to different deployment scenarios. Based on where and how many debloating layers are added, BLAFS supports two layer-sharing modes: (1) *no-sharing mode* for minimal per-container size where no layers are shared across containers on the same host; (2) *fully-sharing mode* which optimizes aggregate storage across containers, enabling all containers on the same host to share their common image layers.

The reloading layer is designed to address one key weakness of debloating tools; their inability to generalize a debloated container. If a file is removed and is later needed, the debloated container will fail [53]. To solve this issue, BLAFS introduces an optional reloading layer that enables containers to remote-fetch a file that is missing from the debloated container in case it is needed after debloating. Thus, debloating can be done repetitively, adapting to any workload changes that were not captured previously. Hence, in addition to the layer-sharing modes enabled by the flexibility of the debloating layers, the reloading layer enables two more deployment modes that can be combined with the layer-sharing modes; (1) *security-hardened mode* where containers are only allowed to access files in a debloated version that encompass all the functionalities the container is allowed to do, failing in case any other files that are not in the security-hardened debloated version are accessed (or raising an error sent to the operator). In this mode, the reloading layer is disabled; (2) *dynamic deployment mode* where the reloading layer is used to fetch any missing files from a remote file caching service, allowing for load changes during runtime.

We anticipate four main use cases for BLAFS: (1) Serverless functions that can be profiled and debloated prior to deployment, reducing storage, cold start latency, and cost; (2) Serverless platforms that can monitor file access in production and debloat mature containers with stable access patterns, improving efficiency and performance at scale; (3) Micro-service based applications where the application's workflow and functionality are stable; (4) Edge computing deployments where the resources are constrained. In building BLAFS, we make the following contributions:

- We design a flexible and effective filesystem-level container debloating approach. We show the flexibility of BLAFS to accommodate different scenarios. We open source BLAFS[1].
- We evaluate BLAFS with 22 popular container images and a serverless benchmarks running on OpenWhisk under different scenarios. We demonstrate reductions in container provisioning times, CVEs, and storage requirements. We also show how BLAFS reduces cold start latencies of serverless functions by up to 68%.
- We further show that BLAFS can be effectively combined with other container optimization techniques, such as lazy-loading snapshotters to enhance their performance, significantly reducing conversion time by up to 93% and provisioning time by up to 19%.

2 Background

2.1 Container File Systems

In essence, a container is a process running on the host operating system isolated via operating system primitives such as cgroups and namespaces. Containers typically use union filesystems such as OverlayFS [38], BTRFS [46] or similar filesystems [48, 56]. A container filesystem can be thought of as a *container layer* on the top and a list of (inherited or shared) *image layer*s, as illustrated in Figure 12 in the appendix. The container layer handles all write requests from the container in a copy-on-write manner with all files created or modified by a container stored in its container layer. The image layers on the other hand are read-only. When a container accesses a file, the filesystem first looks up the file in the container layer, then in the image layers one by one, top to bottom, until the file is found, or an error is raised.

The image layers can be shared across many containers, with the container layer allowing each container to have its isolated filesystem. This approach greatly reduces disk space requirements for each container. However, this architecture is also a major source of bloat [21, 44, 49]. Most containers are built on top of a base container image, where a container inherits all the filesystem layers of the base container image. Any bloat in the base image is inherited in the new container, with new bloat possibly added with each layer added to the filesystem.

2.2 Container Debloating

Bloated containers consume unneeded disk space and network bandwidth, while increasing the provisioning time of containerized

[1]https://github.com/negativa-ai/BLAFS

applications [28, 47, 54]. For serverless, this leads to unnecessary longer cold start times [34, 36, 50]. In addition, large containers often include unnecessary dependencies, libraries, and applications that can introduce and increase security vulnerabilities. Container debloating tools remove bloat from container image layers to reduce their size and improve their security and performance.

One example of such tools is Cimplifier [44]. Cimplifier uses dynamic analysis to slim and partition containers to provide better privilege separation across applications. SlimToolKit [21] is another open-source container debloating tool that also relies on dynamic analysis to generate *Seccomp* profiles and remove unnecessary files from containers. Different to these tools, some container debloating tools do not aim to reduce the size of containers but instead focus on restricting the system calls available to them, reducing the attack surface of these containers [23, 32].

All container debloating tools, sometimes referred to as debloaters, we are aware off suffer from one or several of the following three key limitations:

- All debloaters available today rely on tracing with some sample workloads. This requires users to know beforehand the exact workload that will run, otherwise, the generated containers will fail.
- Current tracing methods fail to capture some file accesses. For example, Cimplifier [44] and SlimToolKit [21] start tracing after the container has started. Hence, they do not capture files accessed during the startup phase. This is especially problematic for containers that utilize resources at startup, such as CUDA libraries in machine learning containers. Removal of these files can lead to the containers failing to start.
- Existing container debloaters trace files inside the container without awareness of the layered filesystem. As a result, they break the layered structure, eliminating layer sharing across containers deployed on the same host. In practice, this can cause the sum of debloated containers to be larger than the sum of the original non-debloated containers on the host due to redundant files being kept in each debloated container.

2.3 Lazy-Loading Snapshotters

To improve container provisioning time, there are solutions that use the *containerd* snapshotter plugin [15] to enable container lazy-loading. Snapshotter techniques need to convert the original container image to the snapshotter format. Starlight [14] is an accelerator for container provisioning that redesigns the container provisioning protocol, filesystem, and storage format. Starlight initially identifies the essential files needed for container start-up. When a container is pulled, these essential files are retrieved first allowing for a quick start-up. The remaining files are then pulled in the background.

eStargz [16] is another open-source snapshotter. eStargz estimates the order of use of files in the image layers of a container and presorts the files in each compressed layer according to the estimation. When a container starts and opens a file that has not been transferred yet, eStargz pauses the container start-up and requests the file from the registry. Similar to Starlight, the remaining files of the container are pulled in the background, resulting in the deployment of the fully bloated container.

While container debloating and lazy-loading snapshotters are orthogonal techniques that can be effectively combined to further enhance the performance of containerized applications, lazy-loading has some fundamental limitations as they require every container to be converted to the snapshotter format. The image format conversion time can be significant, taking several minutes even for simple containers, as we show later in our evaluation. Automating this process for every container can prove challenging in production.

2.4 Containers Cold-Starts

The large size of containers affect their cold-starts. This is especially problematic for serverless computing. Serverless functions are widely deployed using containers in order to provide an isolated and consistent environment for function execution [4, 6, 19]. A cold start occurs when a serverless function is invoked for the first time, or after a period of inactivity, requiring the serverless platform to provision the execution environment [29]. This process typically involves downloading the container image from the registry, initializing the runtime environment, and executing the function, all of which contribute to latency [34, 36, 50]. Serverless function containers are bloated, which exacerbates the cold start latency problem. For instance, on AWS Lambda, a widely used serverless computing platform [6], a simple serverless function that prints "Hello World" can require a container image as large as 530 MB [3].

3 BLAFS Design

The limitations of existing tools to deal with container bloat are fundamental and hence, to the best of our knowledge, limits their usage in production cloud systems. To solve these limitations, we take a radically different approach to debloating in designing BLAFS, where we seamlessly integrate debloating in the existing container filesystems with no added overheads. For production systems, this requires no-changes in the container provisioning pipeline from a user or operator perspective.

From production requirements, we identify three key design goals that BLAFS must satisfy:

- **Effectiveness.** BLAFS should remove unnecessary files in containers with no impacts on container performance.
- **Flexibility.** BLAFS should be flexible to run on different cloud, edge, and container systems accommodating different production use-cases.
- **Transparency.** BLAFS should be transparent with no special requirements to work with the existing container tools minimizing impact on existing infrastructure management and deployment processes.

BLAFS consists of the following components: A *debloating layer* that detects accessed files at the filesystem level (§3.1); An optional *reloading layer* that fetches missing files on demand to ensure the robustness of debloated containers (§3.2); A *converter* that composes the debloating and reloading layers with the original image layers in different configurations, enabling multiple operational modes for various use cases (§3.3); a mode selection strategy to help to select the deployment mode (§3.4); and finally an optional module that uses package dependency analysis to expand the set of retained files, improving the robustness of debloated containers (§3.5).

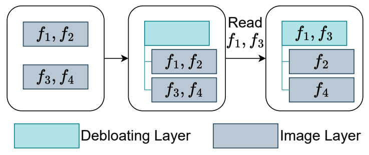


Figure 1: An example of debloating layers.

3.1 The Debloating Layer

When a container reads a file, the original container filesystem first tries to open the file from the top image layer down to the bottom layer and then reads from the file descriptor. Debloating layers are inserted above the original container filesystem layers to detect used files. A debloating layer D has a list of image layers as its child layers ($D.children$). When reading a file from a debloating layer, the file is moved from the child layers to the added debloating layer. In doing so, the debloating layer only includes used files.

Figure 1 shows an example where the original container filesystem has two image layers, with each layer having two files. In this example, we add a debloating layer to the filesystem such that the two original image layers are now child layers to the debloating layer. When the container reads files f_1 and f_3, the two files are moved to the debloating layer from the image layers. The two used files are now in the debloating layer, while the other two unused files stay in the image layers. If we delete the image layers, the container will be debloated containing only the two used files.

Algorithm 1 shows the details of the debloating process. We denote a file to be read as p. We use the notation D/p to denote concatenating the path of layer D with path p. The algorithm first opens p in D. If p exists in D, it returns its file descriptor ($fd \not\equiv -1$). Otherwise, BLAFS tries to open p from the child layers. If p exists in layer l, the file is moved from l to D. If p does not exist in the child layers, -1 is returned to indicate that the file does not exist in D, and that the file should be fetched from lower debloating layers if they exist. The lower debloating layers try to fetch the file from their child layer(s) similarly. This enables BLAFS to have a hierarchy of debloating layers. If no other debloating layers exist, and the reloading layer is enabled, then the file is remotely fetched as we describe later.

Implementation. The debloating layer of BLAFS is developed in C++ with 1k+ lines of code, reimplementing all file-related interface, such as `open`, `read`, `opendir`, etc.. To insert a debloating layer into the container's filesystem, BLAFS first parses the container configuration to retrieve the list of image layers and their corresponding locations on the host filesystem. For each debloating layer, BLAFS creates a new directory on the host, which serves as a mount point for the BLAFS filesystem. This mount point acts as the debloating layer and accepts a list of directories, which are the locations of container's image layers—as its child layers. While the original container filesystem is linear-layered where the layers are stacked sequentially, BLAFS is a tree-layered filesystem due to the debloating layers mounting the original layers as child layers. BLAFS then modifies the container's configuration to change the original images layer paths to this new mount point. As a result, when the container starts, all file accesses are transparently routed through BLAFS's mount point. Any file access is handled according

Algorithm 1 Pseudo-code of function DOPEN

```
Input: Debloating layer $D$; file path $p$
Output: File descriptor
1:  function DOPEN($D$,$p$)
2:      $fd \leftarrow$ open($D/p$)
3:      if $fd \not\equiv -1$ then
4:          return $fd$
5:      end if
6:      for $l$ in $D.children$ do                ▷ from top to bottom
7:          $fd \leftarrow$ open($l/p$)
8:          if $fd \not\equiv -1$ then
9:              move($l/p$, $D/p$)
10:             return open($D/p$)
11:         end if
12:     end for
13:     return $-1$
14: end function
```

to the logic defined in Algorithm 1, enabling separation of accessed and unaccessed files during execution.

Layer-sharing modes. Depending on the number and position of the added layers, BLAFS enables two layer sharing modes, namely, *no-sharing* and *fully-sharing* modes. Operating in the no-sharing mode results in a container with maximum per-container size reduction, while operating in the fully-sharing mode results in multiple containers with maximum total size reduction. We now describe the details of these two modes.

(1) No-sharing Mode. In this mode, BLAFS guarantees the minimum possible size of a container by removing files that are not used by the container without considering any other containers sharing the same host. Given a container with n image layers, the no-sharing mode inserts one debloating layer, which has *all* the n original image layers as its child layers. The debloated container contains only files needed by the container's workloads. This mode is equivalent to existing container debloaters, such as Cimplifier and SlimToolKit, i.e., it flattens the layers of the container file system to be a single layer. The no-sharing BLAFS is effective in producing debloated containers with a minimum size per container, but duplicate files among multiple debloated containers on the same host can result in the total size of the debloated containers being larger than the total size of the original containers which share filesystem layers. We note that all existing container debloaters have this flaw (or feature). To give an example, Figure 2 displays an example of no-sharing BLAFS. The two containers, C_1 and C_2, share the same image layers, which include two layers and four files f_1, f_2, f_3 and f_4 whose sizes are 1MB, 2MB, 3MB, and 4MB respectively. Thus, both C_1 and C_2 are 10MB. When using BLAFS, C_1 reads f_1 and f_2, so both are moved to the debloating layer from the image layer. Similarly, f_2 and f_3 are moved to the debloating layer of C_2. After debloating, C_1 has one debloating layer, which contains f_1 and f_2, while C_2's debloating layer contains f_2 and f_3. As a result, the sizes of C_1 and C_2 are reduced to 3MB and 5MB. However, f_2 is duplicated in both debloated containers.

(2) Fully-sharing mode. This mode allows sharing of common files across multiple debloated containers located on the same host. Given a container with n image layers, BLAFS inserts n debloating

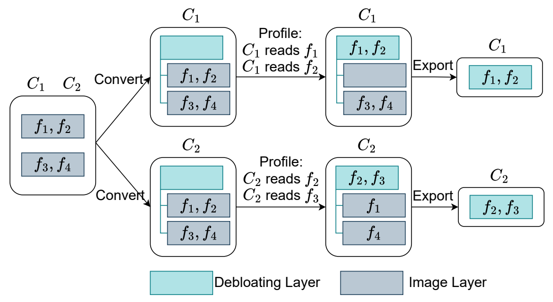


Figure 2: Example for no-sharing BLAFS

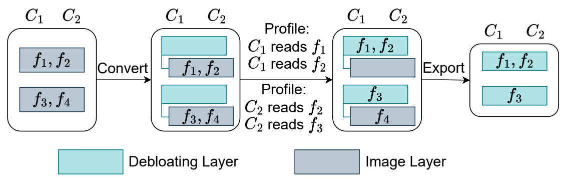


Figure 3: Example for fully-sharing BLAFS

layers, each of which has one image layer as its child layer. This means that layers shared among different containers on the same host can still be shared among their debloated versions, with the added debloating layers having the superset of all the files used by all the containers sharing the same layer. Compared with the no-sharing mode, this can lead to a larger size for each container, but the total storage used is reduced since the layers can be shared. Figure 3 shows an example with the fully-sharing mode. The input container C_1 and C_2 and the workloads are the same as in Figure 2. First, two debloating layers are inserted for each image layer, with the image layer converted to be their child layers. The inserted debloating layers are shared by C_1 and C_2. After conversion, the debloated versions of C_1 and C_2 share the two debloating layers. Therefore, the sizes of C_1 and C_2 are the same, which is 6MB. The size is larger than the sizes of 3MB and 5MB generated by the no-sharing BLAFS, because each container consists of an unnecessary file (f_3 for C_1,f_1 for C_2). However, unlike no-sharing BLAFS, the debloating layers are shared by the two debloated containers. So the total size of C_1 and C_2 is 6MB, which is smaller than the size of 8MB generated by no-sharing BLAFS.

3.2 The Reloading Layer

The reloading layer can be used to fetch files missing in a debloated container in case these files were mistakenly removed or if the workload dynamics change. Figure 4 shows an example of the reloading layer in-action. The reloading layer is appended as the bottom layer in the file system, below all the other image layers. When a container accesses a file, it searches top-to-bottom across the filesystem layers of the layered filesystem. Thus, if a file request reaches the reloading layer, it indicates that the file was not found in any of the preceding layers. The reloading layer intercepts the request for the file and connects to a remote file caching service that can be either the container registry itself, or a cheap cold storage such as AWS S3 [1]. The reloading layer is optional and is used depending on the mode BLAFS is running in as we explain later.

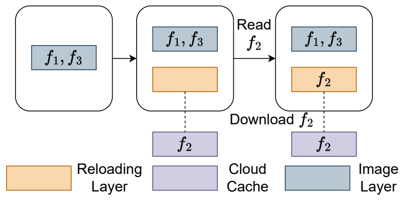


Figure 4: An example of reloading layers.

Algorithm 2 presents the pseudo-code of this reloading process. M is a mapping of the absolute file paths of files expected to exist in the image layers of the container, to their corresponding file types defined by UNIX file system [52]. M is populated by analyzing the file system of container C before debloating. If file p is not in M, it implies that p does not exist in the original file system. So -1 is returned to signify that p does not exist. BLAFS tries to open p from R. If p does not exist in $R(fd \equiv -1)$, BLAFS downloads p from the cloud cache and saves it to R. Then it opens p from R. Finally, the file descriptor is returned. The reloading layers are usually appended to the end of the image layers, to avoid unnecessary file downloading.

Algorithm 2 Pseudo-code of function ROPEN

```
Input: Reloading layer $R$; file path $p$
Output: File descriptor
1:  function ROPEN($R$,$p$)
2:      if $p$ not in $M$ then
3:          return $-1$
4:      end if
5:      $fd \leftarrow$ open($R/p$)
6:      if $fd \equiv -1$ then
7:          download($R$,$p$)
8:          $fd \leftarrow$ open($R/p$)
9:      end if
10:     return $fd$
11: end function
```

Implementation. The reloading layer component of BLAFS is developed in C++ with another 1k+ lines of code. We use AWS S3 [1] as a prototype of a remote caching service. The reloading layer employs the AWS S3 C++ SDK [2] to retrieve from remote caching service on demand.

Reloading modes. BLAFS provides two reloading modes that either enable or disable the reloading layer. When enabled, the *dynamic deployment mode* enables the reloading layer with a remote cache, guaranteeing that containers never fail; even if a file is missing, it is remotely fetched. This would enable operators to enable repetitive debloating, where file removal is run periodically. The *security-hardened mode* disables the reloading layer for container image hardening [25], enabling users to set the container to only run allowed workloads based on which the container is debloated.

3.3 The Converter

The third main component of BLAFS is the image converter. The image converter converts the original container file system to a BLAFS file system by adding debloating layers and reloading layers. Users or operators can choose one of the different BLAFS modes to convert the original filesystem to, i.e., *no-sharing security-hardened*, *fully-sharing security-hardened*, *no-sharing dynamic deployment*, and *fully-sharing dynamic deployment*. Since the converter only adds the layers based on the configuration, the total conversion time are several seconds.

Converter Implementation: The converter extracts the layer information of containers and add the debloating and reloading layer to create a BLAFS filesystem. It is implemented in Golang with 1.7k+ lines of code.

3.4 Layer-Sharing Mode Selection Strategy

Since many cloud applications run colocated on the same host, we implement a heuristic-based tool that determines which mode is better for the containers running on a given host between the two sharing modes. During runtime, and given n containers $C_1, C_2, \cdots, C_n$ running on the same host, our tool calculates the debloated size of each container using the no-sharing mode based on their running workloads $s_1, s_2, \cdots, s_n$. We denote the total disk size required by the debloated containers as t. BLAFS then computes the sizes of these containers if debloated with the fully-sharing BLAFS, s'_1, s'_2, $\cdots$, s'_n, respectively. The total size of the debloated containers in this mode is t'. Let α denote the size of duplicated files in s_i because of no-sharing; β denotes the size of unnecessary files in s'_i because of fully-sharing. α is calculated by Equation 1,

$$\alpha = \sum_1^n s_i - t' \tag{1}$$

β can be obtained using Equation 2,

$$\beta = \sum_1^n (s'_i - s_i) \tag{2}$$

Here we define θ as shown in Equation 3,

$$\theta = \frac{\alpha}{\beta} = \frac{\sum_1^n s_i - t'}{\sum_1^n (s'_i - s_i)} \tag{3}$$

If θ is small, it means there are very few files that can be shared between the debloated containers (α is small), and many unnecessary files will be packed with each container if fully-sharing(β is large). If θ is large, then many files can be shared between the debloated containers (α is large); and fewer unnecessary files will be incurred if using the fully-sharing BLAFS (β is small). In our deployments, we set 1 as a threshold for θ. No-sharing BLAFS is more appropriate for debloating if $\theta < 1$; otherwise, fully-sharing BLAFS is used.

3.5 Package-Dependency-Based Expansion

Traditional container debloaters rely on profiling sample workloads to identify files to be retained. However, these sample workloads may not cover all possible workload variations. To mitigate the risk of removing files that may be accessed by unobserved workloads, BLAFS employs a Package-Dependency-Based Expansion algorithm, PDBE, to expand the files to be retained in the debloated containers. BLAFS first analyzes the packages installed in the containers and the files included in each package. Then for each package, BLAFS calculates the necessity degree of this package(d_p), which is defined as follows:

$$d_p = \frac{\texttt{size}(F_p \cap F_{c'})}{\texttt{size}(F_p)}$$

where F_p is the set of files included in package p. $F_{c'}$ are accessed files by profiling workloads. A package of $d_p > 0$ is considered a *likely needed package* since at least some of its files are used by the observed workloads. To account for potential unseen workloads, all files within such packages are also retained in the debloated container. Two key rules are used to expand likely needed packages:

R0: If a package is a likely needed package, then all files within it will be retained.

R1: If a package is a likely needed package, its dependent packages are also likely needed packages.

Based on the above two rules, PDBE first creates a package dependency graph. Packages with a necessity degree greater than zero are selected as starting points. The package dependency graph is traversed from these starting points to find all the dependencies (R1). The result is a sub-graph that only contains likely needed packages. All the files included in the likely needed packages will be retained in the debloated container(R0).

An alternative strategy for handling files from likely needed packages is to serve them through a cloud-based caching service instead of retaining them in the debloated container, allowing the reloading layer to fetch them on demand.

Implementation. We implement PDBE for Python packages; We use `pipreqdeb` to scan the packages installed in a container and `pip` to identify files included in a package. Support for other package types is left for future work.

4 Evaluation

We compare BLAFS with debloaters and lazy-loading snapshotters on the top 20 most downloaded containers from DockerHub, several ML containers, and a serverless function benchmark. In addition, we evaluate the security hardening benefits of BLAFS by measuring the number of vulnerabilities in the debloated containers. We also evaluate the mode selection strategy, the effectiveness of PDBE, and the overheads incurred by the debloating and reloading layers.

4.1 Comparing to Debloaters

We start our experiments by comparing the effectiveness of BLAFS against two state-of-the-art container debloaters, Cimplifier and SlimToolKit. For these experiments, we ran with the no-sharing security-hardened mode as this is similar to what these debloaters provide. We select the top 20 downloaded containers from DockerHub, focusing exclusively on application containers rather than general-purpose operating system containers. For each selected container, we manually identified representative workloads to cover as many relevant use-cases as possible. The number of identified workloads for each container ranges from 2 to 7. We used these as our debloating container. Each workload may include multiple functionalities. The details of the selected containers are shown in Table 7 in the appendix. Additionally, we evaluate two machine learning (ML) training containers sourced from AWS Deep Learning Containers (DLC) [12], which offer pre-built environments for model training and inference. For these containers, we use the same training workloads from AWS DLC for debloating.

Table 1 summarizes the debloating results for successfully debloated containers. While BLAFS successfully debloated all 22 containers, Cimplifier and SlimToolKit successfully debloated 7 and 8 containers, respectively, producing broken containers for the remaining containers. The results also reveal significant bloat across most of the containers, with average size reductions of 59%.

Table 1: Number of successfully debloated containers and the average, minimum, and maximum size reduction.

Debloater	#Containers	Avg.	Min.	Max.
Cimplifier	7	51%	4%	95%
SlimToolKit	8	55%	4%	93%
BLAFS	**22**	**59%**	**4%**	**95%**

Table 8 in the appendix lists the detailed debloating results for all 22 containers using BLAFS, sorted by the reduction percentage. The results for Cimplifier and SlimToolKit are detailed in Tables 9 and 10 in the appendix. The original container sizes ranged from 14 MB to 16,822 MB, with size reductions varying between 4% and 95%. We note that 16 out of the 22 containers had size reductions more than 50%, and only two had a size reduction of less than 10%. Furthermore, for the two ML containers, `tensorflow-train` and `pytorch-train`, their original sizes are much larger than the other containers, with sizes of 10,952 MB and 16,822 MB. BLAFS successfully reduce their sizes by 85% and 83%, respectively. Our findings show the effectiveness of BLAFS in debloating containers. It also shows the significant bloat present in these widely-used containers.

Summary. BLAFS successfully debloats all 22 containers, while Cimplifier and SlimToolKit only succeed on 7 and 8 containers, respectively. The container size reduction of BLAFS ranges from 4% to 95%, with an average of 59%.

4.2 BLAFS for Serverless Computing

In this set of experiments, we evaluate BLAFS with realistic serverless function containers in both open-source and commercial serverless platforms. We test BLAFS with nine serverless containers of the *Serverless Benchmark [17]*. One of the containers in the benchmark requires special configuration on the serverless platform was excluded from the evaluation. We evaluate cold start latency improvements by deploying the function containers on two serverless platforms: an open-source serverless platform, OpenWhisk [19], and a commercial serverless platform, AWS Lambda [6].

Table 2: Container size and cold start latency of original and debloated function containers on OpenWhisk. Percentages in parentheses are reductions. Ori.=Original, Deb.=Debloated.

Function	Size/MB		Latency/ms	
	Ori.	Deb.	Ori.	Deb.
dynamic-html	1,085 (96%)	49	18,192 (68%)	5,839
uploader	1,084 (95%)	49	18,682 (68%)	6,029
thumbnailer	1,099 (95%)	55	18,854 (68%)	6,017
video-proc.	1,339 (91%)	125	21,737 (67%)	7,244
compression	1,084 (96%)	49	19,206 (67%)	6,335
image-recog.	1,802 (66%)	619	28,283 (46%)	15,392
graph-page.	1,093 (95%)	60	18,619 (62%)	7,158
graph-mst	1,093 (95%)	60	18,869 (67%)	6,162
graph-bfs	1,093 (95%)	60	18,682 (68%)	5,975

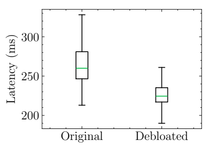
Table 3: Container size and memory usage of function containers on AWS Lambda. The percentage in parentheses is the reduction.

Function	Container Size/MB	Memory Usage/MB
dynamic-html	537 (72%)	40 (2%)
uploader	535 (67%)	79 (3%)
thumbnailer	564 (68%)	93 (2%)
video-proc.	789 (68%)	321 (0%)
compression	535 (67%)	101 (1%)
image-recog.	1,893(61%)	681 (1%)
graph-page.	553 (72%)	41 (2%)
graph-mst	553 (72%)	41 (2%)
graph-bfs	553 (72%)	41 (2%)

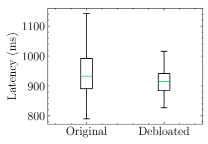
Evaluation on OpenWhisk We first evaluate the performance improvement of the debloated serverless functions using OpenWhisk. The framework was deployed on a machine with 16 CPUs and 64GB of memory. We use DockerHub as the container registry. We run the functions using the no-sharing mode. Since serverless functions are supposed to be modular, we also use the security-hardened mode. For each original and debloated container, we enforce cold starts 50 times and record the cold start latency. The results, summarized in Table 2, include the container sizes and medians of the cold start latency of the original and debloated containers. The debloated container sizes are much smaller, with reductions ranging from 66% to 96%. Similarly, cold start latencies also decrease, with reductions between 46% and 68%. The standard deviation of the cold start latency was less than 6%.

Evaluation on AWS Lambda. We perform the same evaluation on a commercial serverless platform, AWS Lambda. We again evaluate the no-sharing mode of BLAFS using the same workload we used with OpenWhisk. The results of our experiments are summarized in Table 3. The evaluation focuses on two metrics provided by the AWS Lambda platform: memory usage and cold start latency. Memory usage measures the amount of memory consumed by the function during execution. We use the billed duration reported by the benchmark as the cold start latency metric, which represents the time from when the function begins executing until it terminates, rounded up to the nearest millisecond. Table 3 shows the original container size, memory usage and their reductions after debloating. For AWS Lambda function containers, the container sizes are reduced by 61% to 72%. Since AWS Lambda hosts containers in its registry, smaller container sizes can further reduce costs associated with storage and network bandwidth. Besides the container size reduction, the memory usage is also reduced by 0% to 3%.

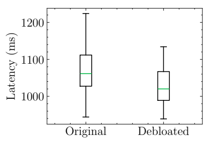
Figure 5 shows the cold start latency of the original and debloated containers for each function. For all functions, the debloated containers show lower cold start latency compared to the original containers. We calculate the relative improvement of the cold start latency using the median values. The functions `dynamic-html`, `graph-mst` and `graph-bfs` achieve the highest improvements, with reductions of 14% in cold start latency. The minimal improvement of 1% was observed for the `video-processing` and `image-recognition`. While the median cold start latencies show modest improvements, the tail cold start latency shows a reduction of up to 35%. We believe this is because AWS Lambda already employs certain lazy-loading techniques—such as lazy-loading snapshotters—to accelerate cold starts. Nevertheless, BLAFS provides additional performance gains. We further demonstrate how BLAFS can complement lazy-loading snapshotters to achieve greater improvements in § 4.6.



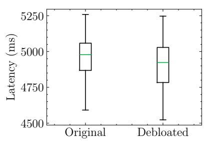
(a) dynamic-html



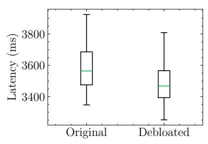
(b) uploader



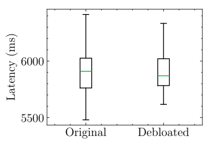
(c) thumbnailer



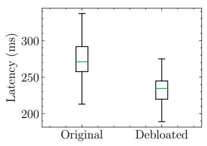
(d) video-processing



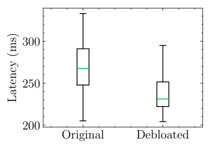
(e) compression



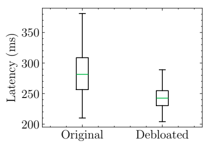
(f) image-recognition



(g) graph-pagerank



(h) graph-mst



(i) graph-bfs

Figure 5: Cold start latency of the original and debloated containers for each function on AWS Lambda.

> **Summary**. For serverless function containers in OpenWhisk and AWS Lambda, BLAFS reduces container sizes by up to 96% and 72%, and cold start latencies by up to 68% and 14%, respectively.

4.3 Evaluation of Security Impact

To better understand the security benefits of debloating, we use Grype [10], a popular container scanning tool to scan the vulnerabilities in a set of 10 debloated versus the original containers. For this experiment, we opted to use slightly older versions of the containers that are widely used in production today. We debloated these containers using the no-sharing security-hardened mode.

Table 4 displays the number of CVEs at each severity level found in the original containers and debloated containers. As can be seen, BLAFS significantly reduces the number of CVEs, with reduction percentages ranging from 20% to 97%. For `mysql:8.0.23`, `redis:6.2.1`, `golang:1.16.2` and `python:3.9.3`, the critical level CVEs are also reduced considerably. The number of CVEs found in the debloated containers is considerably lower than that in the original containers, indicating that debloating can effectively reduce the security risks associated with production containers.

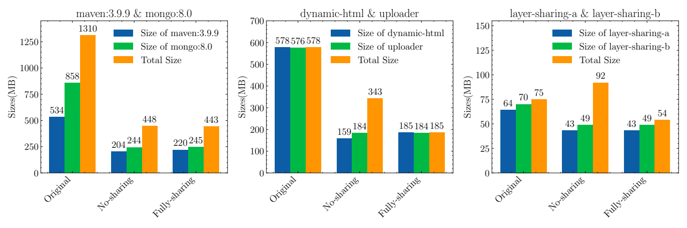


Figure 6: Container sizes under different modes.

> **Summary**. For a set of 10 widely-used containers, BLAFS reduces the number of CVEs significantly, with reductions ranging from 20% to 97%.

4.4 Evaluation of Mode Selection Strategy

To evaluate the mode selection strategy of BLAFS, we select two containers with shared layers, `maven:3.9.9` and `mongo:8.0`, and two AWS Lambda function containers `dynamic-html` and `uploader`. We also manually created two containers `layer-sharing-a` and `layer-sharing-b` with shared layers. These two created containers have a large overlap in their layers and the files they access. We debloated these containers using no-sharing and fully-sharing BLAFS. The results are presented in Figure 6.

For containers `maven:3.9.9` and `mongo:8.0`, originally sized at 534MB and 858MB (totaling 1,310MB due to shared layers), debloating with no-sharing BLAFS results in a significant size reduction, bringing the total size down to 448MB. When debloated using fully-sharing BLAFS, while the total size of both containers is slightly smaller, individual container sizes are slightly larger than their no-sharing counterparts. The θ value of these two containers is 0.3. This means that debloating these two containers using fully-sharing BLAFS will cause each container to include more unneeded files. Therefore, no-sharing BLAFS for these containers is more suitable.

In the case of containers `dynamic-html` and `uploader`, with original sizes of 578MB and 576MB, no-sharing BLAFS reduces the total size to 343MB while layer-sharing BLAFS markedly decreases the total size to 185MB. This significant size reduction, supported by a θ value of 6, indicates a clear advantage of fully-sharing BLAFS for these containers.

The experiment with `layer-sharing-a` and `layer-sharing-b` reveals an intriguing aspect of container debloating. The original total size of two containers is 75MB, debloating with no-sharing BLAFS, however, increases their total size to 92MB. This issue, that the total size of debloated containers can exceed that of the original ones, is faced by all state-of-the-art debloating tools like Cimplifier and SlimToolKit. These tools break the layer-sharing feature of container filesystems. However, fully-sharing BLAFS utilizes the layer-sharing feature and effectively reduce their total size to 54MB. A remarkably high θ value of 380,000 in this scenario indicates that fully-sharing BLAFS is more suitable.

> **Summary**. The mode selection strategy of BLAFS effectively selects the appropriate debloating mode for containers with shared layers, leading to optimal size reductions.

Table 4: Number of CVEs at each severity level found in the original containers and debloated containers. Numbers in the parenthesis represent the numbers of CVEs found in the debloated containers.

Container	Critical	High	Medium	Low	Negligible	Total	Reduction
mysql:8.0.23	26 (3)	71 (7)	44 (5)	34 (3)	90 (11)	265(29)	89%
redis:6.2.1	20 (2)	67 (12)	32 (6)	25 (3)	60 (3)	204 (26)	87%
ghost:3.42.5-alpine	14 (12)	129 (108)	67 (46)	7 (7)	63 (2)	217 (173)	20%
registry:2.7.0	4 (3)	46 (37)	9 (7)	0 (0)	0 (0)	59(43)	27%
golang:1.16.2	63 (2)	376 (19)	314 (5)	80 (3)	499 (5)	1332(34)	97%
python:3.9.3	145 (15)	863 (47)	1048 (34)	418 (3)	898 (10)	3372 (109)	20%
bert_tf2:latest*	38 (26)	358 (256)	1426 (452)	549 (210)	57 (3)	2428 (947)	61%
nvidia_mrcnn_tf2:latest*	38 (26)	358 (225)	1445 (451)	558 (210)	57 (3)	2456 (945)	62%
merlin-pytorch-training:22.04*	47(46)	166(127)	902(134)	317(15)	70(3)	1502(325)	78%
merlin-tensorflow-training:22.04*	15(14)	109(76)	939(222)	304(34)	56(3)	1423(349)	75%

* These machine learning containers are collected from NVIDIA NGC services [37].

4.5 Comparing to Lazy-Loading

Lazy-loading snapshotters are used to improve container provisioning time. They rely on the containerd snapshotter plugin as discussed earlier. In this experiment, we compare BLAFS with three state-of-the-art container provisioning approaches: the containerd baseline [15], and two lazy-loading snapshotters; eStargz [16] and Starlight [14]. We deployed an image registry on a `t3.large` instance, with ten different images using the no-sharing mode. We then pulled the containers from a `g4dn.2xlarge` instance acting as an emulated edge server. We compare deploying the debloated image (including pulling, creating and starting) to the baseline of using a containerd container and the two lazy-loading snapshotters.

To run this experiment, the original containers were converted to the eStargz and Starlight format. In addition, Starlight requires a proxy server for mediating between Starlight workers and the registry server, which we set up in the same AWS instance as the registry server. We pulled the container 30 times with each of the approaches on two different network settings, namely a network connection with 4Gbps bandwidth and 10ms latency, and another network connection with 100Mbps bandwidth and 200ms latency.

To compare the speed-ups of the different approaches, we divide the average provisioning time of each approach with the average provisioning time of containerd original file-system. Figure 7 presents the average speed-up of provisioning time using the three approaches. We note that for six containers, Starlight produces non-functional containers[2]. For these containers, we do not plot any result for Starlight. Figure 7 shows that for the 4Gbps bandwidth and 10ms latency network configuration, BLAFS outperforms eStargz for 4 out of 10 containers. For `golang:1.16.2` and `python:3.9.3`, BLAFS and eStargz have similar performance. BLAFS outperforms Starlight for 3 out of 4 containers. Under the 100Mbps bandwidth and 200ms latency configuration, BLAFS has higher performance compared to eStargz for eight containers, having a slightly worse performance for two (`nvidia_mrcnn_tf2` and `mysql`). Comparing BLAFS with Starlight, BLAFS outperforms Starlight for only one out of four Starlight working containers (`golang:1.16.2`).

We finally note that when containers are converted using eStargz and Starlight, their compressed sizes are slightly larger than the original containers, requiring larger space on (the constrained edge) host. This is a limitation that all state-of-the-art lazy-loading snapshotters, to the best of our knowledge, have. For example, SOCI, another state-of-the-art snapshotter based on eStargz, deployed in AWS Fargate only provides benefits for containers larger than 250MB [41]. For small container images, SOCI can even slow down the time taken to launch AWS Fargate Tasks.

[2]We reported our findings to the paper authors, and they are investigating the reason.

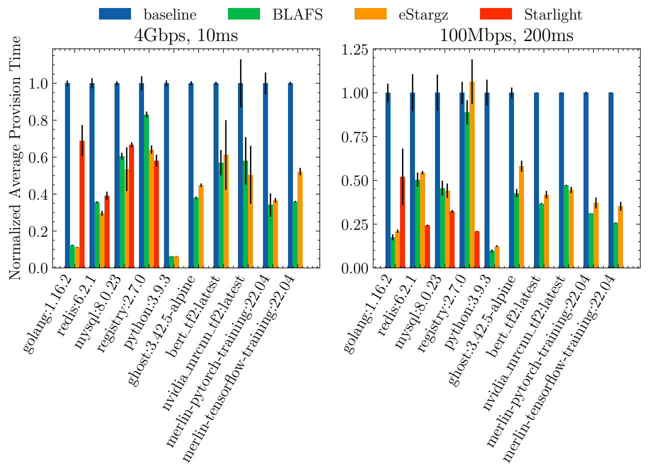


Figure 7: Provision time speed-up under different network connections. The lower, the better the provision performance is. For python:3.9.3 to merlin-tensroflow-training:22.04, Starlight fails to generate functional containers and their results are not plotted.

> **Summary.** Compared to eStargz and Starlight, BLAFS achieves comparable container provisioning performance, while also reducing container sizes significantly.

4.6 Combining BLAFS with Lazy-Loading

BLAFS can be combined with lazy-loading snapshotters to further enhance provision performance. Lazy-loading snapshotters first convert container images into a lazy-loading format, a process that can be time-consuming [14]. This conversion latency can lead to delays in updating serverless functions. Once converted, the image is pushed to and stored in a registry. During container startup, only the essential files for startup are immediately pulled, while the remaining files are fetched in the background. This mechanism allows the container to start faster. We evaluate whether debloating can improve two key metrics of lazy-loading snapshotters:(1) Image format conversion time and (2) Container pull time. We combined BLAFS with two lazy-loading snapshotters, eStargz and Starlight. We used the same nine OpenWhisk serverless containers, debloating them using the no-sharing mode. The full-sharing modes is not used, as lazy-loading snapshotters do not rely on a base image to start the container. Then we convert both the original and debloated containers into a lazy-loading format using eStargz and Starlight.

The time required to convert each container is measured, and we display the results in Table 5. Both eStargz and Starlight exhibit significant reductions in conversion time for debloated containers. For eStargz, the conversion time is reduced by 28% to 79%. For Starlight, the reduction range from 45% to 93%. The standard deviation of the conversion time is less than 3%. These results demonstrate that debloating significantly reduces the time needed to convert container images into a lazy-loading format.

Table 5: The conversion time of eStargz and Starlight. Percentages in parentheses are reductions. Time is in seconds.

Function	eStargz	Starlight
dynamic-html	73s (79%)	114s (93%)
uploader	67s (77%)	113s (93%)
thumbnailer	68s (77%)	114s (93%)
video-proc.	70s (70%)	115s (92%)
compression	67s (77%)	114s (93%)
image-recog.	111s (28%)	117s (45%)
graph-page.	68s (77%)	113s (93%)
graph-mst	63s (76%)	112s (93%)
graph-bfs	65s (76%)	114s (93%)

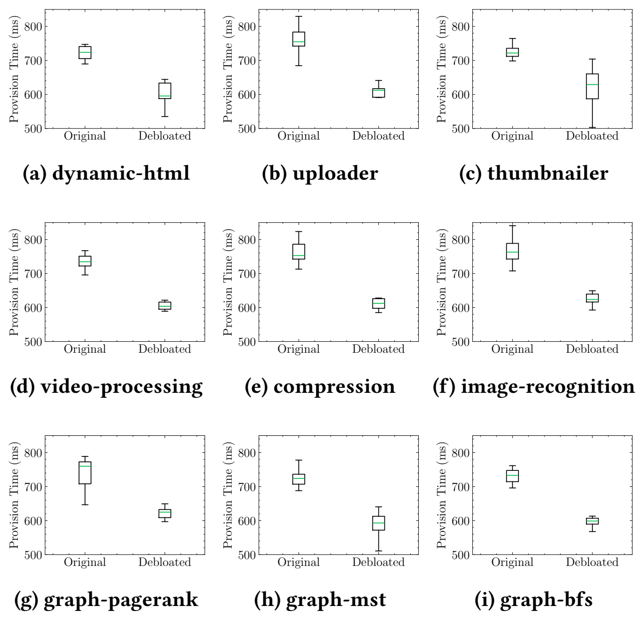


Figure 8: Provision time of the original and debloated containers for each function using eStargz.

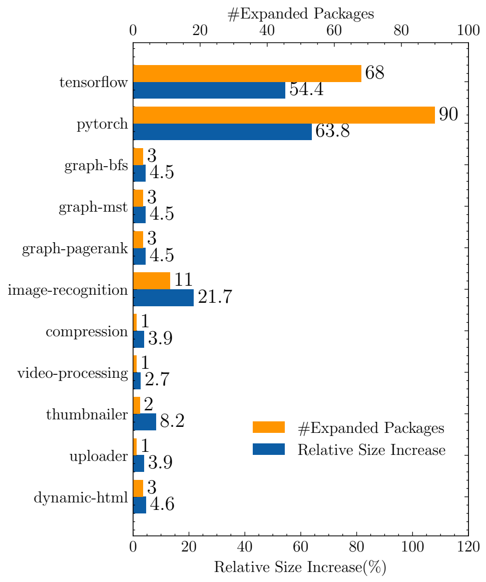


Figure 9: Relative size increase and number of expanded packages using PDBE.

Figure 8 presents the provisioning times for both the original and debloated containers. Starlight is excluded from this evaluation, as it failed to start the containers. The results show that with eStargz, the average provisioning times for debloated containers is reduced by 13% to 19%. However, the tail is reduced by up to 40% for the `compression` function. These findings confirm that container debloating can complement lazy-loading snapshotters, reducing the time required for format conversion and container provisioning, and thereby improving the overall performance of lazy-loading snapshotters. We note that running with AWS Lambda function containers had similar results.

Summary. Combining BLAFS with lazy-loading snapshotters significantly reduces the time required for image format conversion (by up to 93%) and container provisioning (by up to 19%).

4.7 Evaluation of PDBE

Table 6: Execution results of unobserved workloads for the two ML containers.

pytorch-train			tensorflow-train		
Workloads*	BLAFS	BLAFS +PDBE	Workloads*	BLAFS	BLAFS +PDBE
Torchdata	✗	✓	TensorBoard	✓	✓
PytorchRegression	✓	✓	TFAddons	✓	✓
PyTorchwithInductor	✗	✓	TFKerasHVDFP32	✗	✓
Torchaudio	✗	✓	TFKerasHVDFAMP	✗	✓

* Please refer to the appendix §A.6 for the details of the workloads.

PDBE is designed to mitigate the risk of removing necessary files by expanding the set of retained files based on package dependency analysis. While this improves coverage, it also increases the size of the debloated containers. To evaluate this trade-off, we focus on

AWS serverless function containers and two ML containers, all of which use Python packages—the package type currently supported by PDBE. We compare two configurations in no-sharing mode: debloating containers with or without PDBE. Figure 9 presents the relative size increase compared to the debloated containers and the number of expanded packages across the containers. As expected, all containers show a size increase due to the additional retained files introduced by PDBE, ranging from 2.7% to 63.8%. Notably, the two ML containers exhibit the most size increases (63.8% and 54.4%), as PDBE expanded the largest number of packages for these containers (90 and 68).

We further evaluate whether PDBE improves the robustness of debloated containers against unobserved workloads. The two ML containers (`pytorch-train` and `tensorflow-train`) are selected for this evaluation, as they are designed to perform more various workloads than the serverless function containers. The two ML containers were debloated using training workloads for a CNN model, representative of their core functionality. To test robustness, we use four additional ML workloads drawn from publicly available samples [12]. These workloads serve as unobserved workloads to evaluate robustness of the debloated containers. Table 6 summarizes the results for the two containers. Containers debloated with only BLAFS show partial success—some workloads execute successfully, while others fail due to missing files. In contrast, containers debloated with BLAFS + PDBE successfully execute all four workloads. It is important to note that this does not imply PDBE *guarantees* robustness, for that the reloading layer should be used. However, the results demonstrate that PDBE can meaningfully *improve* the robustness of debloated containers.

Summary. PDBE effectively enhances the robustness of debloated containers against unobserved workloads, but at the cost of increased container size.

4.8 Debloating and Reloading Overheads

Our final set of experiments aim to show how running with both layer sharing and dynamic mode introduces overheads. We note that our previous evaluations did not evaluate the dynamic sharing mode since it does not affect any aspects of the BLAFS performance except by adding an extra overhead to fetch the file remotely. All our previous results thus hold for the reloading layer, except for the overheads added.

Overhead of the debloating layer. To measure the performance overhead of BLAFS, we run disk benchmarks [39] using the Phoronix test suite [40]. Here we use the fully-sharing mode as it has more layers compared to the original file system. We first run the disk benchmarks in a container with the original container filesystem. Then we convert the filesystem into fully-sharing BLAFS and run the same disk benchmarks in the converted container. The Flexible IO Tester of the disk benchmark was executed in the container. Only read operations were measured as BLAFS does not affect the writing layer of a container. Both random read and sequential read operations were measured with block sizes of 4KB and 2MB, and the bandwidth and I/O per second (IOPS) were recorded. Then we compare the performance metrics of the original container with the container of BLAFS. The performance metrics of the container of BLAFS were divided by the same metrics of the original container to obtain the relative overhead. Figure 10 shows the relative overhead (*x-axis*) of the read operations of BLAFS. The results show that all the metrics are around 1, indicating that the performance of BLAFS and the original container filesystem are similar. The debloating layer does not incur much performance overhead. Although we show the results for the fully-sharing BLAFS, the no-sharing BLAFS achieves similar performance.

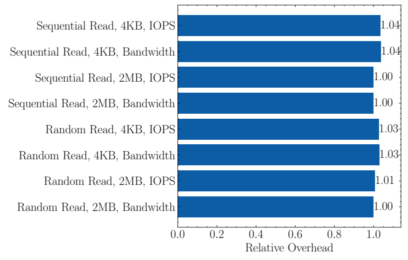


Figure 10: Relative overhead of file-system read operations. Lower is better.

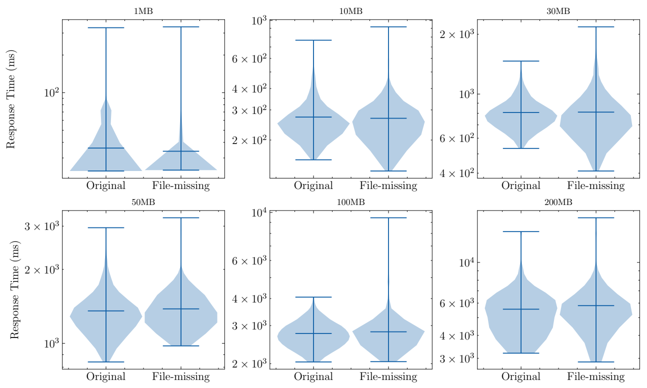


Figure 11: Violin plots of the response time distribution (in milliseconds) of file retrievals. The horizontal line in the shade indicates the mean value of the response time.

Overhead of the reloading layer. In this experiment, we combine the no-sharing mode with the dynamic deployment mode, i.e., with enabled reloading layer and remote cache service. The reloading layer incurs overhead when a file needs to be fetched on demand. In order to test the performance overhead of the reloading layer, we simulate the case when a file is missing using a Nginx server serving data from six files with sizes of 1MB, 10MB, 30MB, 50MB, 100MB and 200MB inside the container. In this experiment, we removed the files from the containers manually, triggering the reloading layer when the first time the file is accessed. While unrealistic, we choose this scenario to be able to control the size of the file-misses. We used Locust [35] for request generation. The experiment ran for 3 minutes with a locust spawn rate of 10 users per second. In total, the 1MB file was downloaded over 77000 times, and over 440 times for the 200MB file, with the other sizes having their download times in-between these two numbers. We compare the response time of file retrievals in the file-missing container with that of the original container. We used AWS S3 Standard storage as the remote file caching service, while the containers are deployed on `g4dn.2xlarge` instances.

Figure 11 shows the distribution of response times when requesting files from both the original container and the file-missing container. We notice that the file-missing container exhibits a longer tail towards higher response times for all file sizes compared to the original container. This outcome is anticipated since the reloading layer must dynamically load files when they are absent before they can be served, resulting in a significant performance overhead for the initial few requests. Nevertheless, the distributions of the original container and the file-missing container display comparable spreads around the mean, suggesting that the performance overhead imposed by the reloading layer occurs only occasionally. Once the missing files are loaded, the file-missing container's performance aligns with that of the original container, leading to nearly identical mean values.

Summary. The debloating layer incurs negligible performance overhead, while the reloading layer incurs overhead only when files are fetched on demand, with performance similar to the original container once files are loaded.

5 Discussion

Limitations. Our current implementation of reloading layer fetches files individually, while loading a bloated container fetches all files at once, often with compression. This design of the reloading layer can lead to decreased fetching performance, leading to reloading consuming more bandwidth as reloading layer cannot batch or compress multiple files in a single transfer. However, this overhead is mostly mitigated by the fact that fewer files are fetched in total, since it is unlikely that all removed files will be accessed. In many cases, even a change of workload does not lead to significant file misses. To further address this issue for fault-tolerant or horizontally scaled services, where multiple instances of the same service coexist, a practical approach is to maintain one bloated instance to which any missing-file requests can be redirected, while deploying additional instances using the debloated container. Although this strategy slightly increases the overall cost, the savings remain substantial due to the large number of replicas typically involved, and scaling or provisioning operations continue to be fast.

Integration with Existing Systems.. BLAFS can be integrated into existing CI/CD pipelines. After the container image is built, BLAFS can be used to debloat the image based on profiling workloads. The debloated image can then be pushed to a container registry for deployment. This process can optionally generate a Software Bill of Materials (SBOM) listing accessed and removed files for auditing and compliance. In production environments, BLAFS can continuously monitor file accesses and dynamically remove unused content based on actual workloads. This enables automatic adaptation to evolving execution patterns, maintaining efficiency without requiring manually crafted profiling workloads.

6 Related Work

The problem of container bloat has been well studied from both academia and industry. Apart from the lazy-loading snapshotters discussed in §2, many other techniques have been proposed to address the effects of container bloat. CNTR [49] introduces the concept of a slim and a fat container image. The three main use cases for CNTR are: Container to container debugging in production; Host to container debugging; and Container to host administration. Both the slim and fat containers run on the same host, and there are no space or bandwidth savings. Slacker [26] is a Docker storage driver designed to optimize fast container startup and reduce the time it takes to provision a container. It provisions the container quickly using backend clones and minimizes startup latency by lazily fetching container data. DADI [33] is a block-level image service for increased agility and elasticity in deploying applications by providing fine-grained on-demand transfer of remote images. FAASNET [51] is a middle-ware system designed for highly scalable container provisioning in serverless platforms, which enables scalable container provisioning via a lightweight, adaptive function tree structure and uses an on-demand fetching mechanism to reduce provisioning costs. Gear [22] is a new image format that reduces container deployment time and storage size of the image registry by separating the index that describes the filesystem structure from the files that are required for running an application. We argue that these optimizations only solve the symptoms of the problem, but not the root cause. We believe that the root cause of long provisioning time and increased resource usage is container bloat, and BLAFS can be combined with them to provide further improvements.

7 Conclusion

This paper addresses the issue of container bloat, which impacts provisioning times, resource utilization, overall system performance and security. We demonstrate that container bloat is widespread, with over 50% of the top 20 containers in DockerHub containing more than 60% of bloat. Existing debloating tools have several inherent limitations, such as breaking the layered structure of container filesystems. To overcome these limitations, we introduced BLAFS, a bloat-aware filesystem that preserves container functionality while significantly reducing container bloat. BLAFS supports multiple modes, including no-sharing, fully-sharing, security-hardened and dynamic deployment, making it adaptable to diverse use cases. Our evaluations show that BLAFS reduces cold start latency of serverless functions by up to 68% and, when integrated with lazy-loading snapshotters, enhances container provisioning performance by reducing conversion times by 93% and provisioning times by 19%. Under the security-hardened mode, BLAFS effectively reduces the number of CVEs in containers by up to 89%. BLAFS provides an effective and flexible solution to container debloating that balances the trade-off between container convenience and efficiency.

Acknowledgments

This project is supported by the Knut and Alice Wallenberg Foundation via a Wallenberg AI, Autonomous Systems and Software Program PhD grant and an SSF future research leaders grant. Mohannad was funded by the Office of Naval Research (ONR) under Contracts N68335-17-C-0558 and N00014-24-1-2049.

References

[1] Amazon S3 - Cloud Object Storage - AWS — aws.amazon.com. https://aws.amazon.com/s3/, 2025. [Accessed 2025-07-10].

[2] AWS SDK for C++ — aws.amazon.com. https://aws.amazon.com/sdk-for-cpp/, 2025. [Accessed 2025-07-10].

[3] Deploy Python Lambda functions with container images - AWS Lambda — docs.aws.amazon.com. https://docs.aws.amazon.com/lambda/latest/dg/python-image.html#python-image-instructions, 2025. [Accessed 2025-07-10].

[4] GitHub - openfaas/faas: OpenFaaS - Serverless Functions Made Simple — github.com. https://github.com/openfaas/faas, 2025. [Accessed 2025-07-10].

[5] ptrace(2) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man2/ptrace.2.html, 2025. [Accessed 2025-07-10].

[6] Serverless Function, FaaS Serverless - AWS Lambda - AWS — aws.amazon.com. https://aws.amazon.com/lambda/, 2025. [Accessed 2025-07-10].

[7] strace(1) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man1/strace.1.html, 2025. [Accessed 2025-07-10].

[8] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Large-scale debloating of binary shared libraries. *Digital Threats: Research and Practice*, 1(4):1–28, 2020.

[9] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[10] Anchore. Grype. https://github.com/anchore/grype, 2025. [Online; accessed 2025-07-10].

[11] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. Improving docker registry design based on production workload analysis. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 265–278, 2018.

[12] AWS. Aws deep learning containers. https://github.com/aws/deep-learning-containers, 2025. [Online; accessed 2025-07-10].

[13] Michael D Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Eric Kilmer, and Ronald Eytchison. A broad comparative evaluation of software debloating tools. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3927–3943, 2024.

[14] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. Starlight: Fast container provisioning on the edge and over the WAN. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

[15] containerd. Containerd. https://github.com/containerd/containerd, 2025. [Online; accessed 2025-07-10].

[16] containerd. Stargz snapshotter. https://github.com/containerd/stargz-snapshotter, 2025. [Online; accessed 2025-07-10].

[17] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 369–384, 2024.

[19] Karim Djemame, Matthew Parker, and Daniel Datsev. Open-source serverless architectures: an evaluation of apache openwhisk. In *2020 ieee/acm 13th international conference on utility and cloud computing (ucc)*, pages 329–335. IEEE, 2020.

[20] Docker. Dockerhub. https://github.com/libfuse/libfuse, 2025. [Online; accessed 2025-07-10].

[21] Slimtoolkit. https://slimtoolkit.org/, 2025. [Online; accessed 2025-07-10].

[22] Hao Fan, Shengwei Bian, Song Wu, Song Jiang, Shadi Ibrahim, and Hai Jin. Gear: Enable efficient container storage and deployment with a new image format. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 115–125. IEEE, 2021.

[23] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 443–458, 2020.

[24] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. *Proceedings of Machine Learning and Systems*, 4:779–794, 2022.

[25] Md Sadun Haq, Thien Duc Nguyen, Ali Şaman Tosun, Franziska Vollmer, Turgay Korkmaz, and Ahmad-Reza Sadeghi. Sok: A comprehensive analysis and evaluation of docker container attack and defense mechanisms. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4573–4590. IEEE, 2024.

[26] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 181–195, 2016.

[27] Muhammad Hassan, Talha Tahir, Muhammad Farrukh, Abdullah Naveed, Anas Naeem, Fareed Zaffar, Fahad Shaon, Ashish Gehani, and Sazzadur Rahaman. Evaluating container debloaters. In *2023 IEEE Secure Development Conference (SecDev)*, pages 88–98. IEEE, 2023.

[28] Shihong Hu, Weisong Shi, and Guanghui Li. Cec: A containerized edge computing framework for dynamic resource provisioning. *IEEE Transactions on Mobile Computing*, 22(7):3840–3854, 2023.

[29] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 938–953, New York, NY, USA, 2025. Association for Computing Machinery.

[30] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. Set the configuration for the heart of the os: On the practicality of operating system kernel debloating. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–27, 2020.

[31] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, 2018.

[32] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-phase execution of application containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pages 230–251. Springer, 2017.

[33] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. Dadi: Block-level image service for agile and elastic application deployment. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 727–740, 2020.

[34] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[35] Locust. An open source load testing tool. https://www.locust.io/, 2025. [Online; accessed 2025-07-10].

[36] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[37] NVidia. Nvidia NGC containers. https://catalog.ngc.nvidia.com/containers, 2025. [Online; accessed 2025-07-10].

[38] Overlay filesystem. https://docs.kernel.org/filesystems/overlayfs.html, 2025. [Online; accessed 2025-07-10].

[39] Phoronix. Disk test suite. https://openbenchmarking.org/suite/pts/disk, 2025. [Online; accessed 2025-07-10].

[40] Phoronix. Phoronix test suite 10.8.4. https://github.com/phoronix-test-suite/phoronix-test-suite, 2025. [Online; accessed 2025-07-10].

[41] Olly Pomeroy and Vaibhav Khunger. Under the hood: Lazy loading container images with seekable oci and aws fargate. https://aws.amazon.com/blogs/containers/under-the-hood-lazy-loading-container-images-with-seekable-oci-and-aws-fargate/, 2025. [Online; accessed 2025-07-10].

[42] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 805–825, 2020.

[43] David Quigley, Josef Sipek, Charles P Wright, and Erez Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[44] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 476–486, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Yuxin Ren, Kang Zhou, Jianhai Luan, Yunfeng Ye, Shiyuan Hu, Xu Wu, Wenqin Zheng, Wenfeng Zhang, and Xinwei Hu. From dynamic loading to extensible transformation: An infrastructure for dynamic library transformation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 649–666, 2022.

[46] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[47] Deepa Rajendra Sangolli, Nagthej Manangi Ravindrarao, Priyanka Chidambar Patil, Thrishna Palissery, and Kaikai Liu. Enabling high availability edge computing platform. In *2019 7th IEEE International Conference on Mobile Cloud Computing,*

Services, and Engineering (MobileCloud), pages 85–92, 2019.
[48] Yu Sun, Jiaxin Lei, Seunghee Shin, and Hui Lu. Baoverlay: a block-accessible overlay file system for fast and efficient container storage. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 90–104, 2020.
[49] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight {OS} containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 199–212, 2018.
[50] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, 2020.
[51] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.
[52] Wikipedia. Unix file types. https://en.wikipedia.org/wiki/Unix_file_types, 2025. [Online; accessed 2025-07-10].
[53] Qi Xin, Qirun Zhang, and Alessandro Orso. Studying and understanding the tradeoffs between generality and reduction in software debloating. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.
[54] Huaifeng Zhang, Fahmi Abdulqadir Ahmed, Dyako Fatih, Akayou Kitessa, Mohannad Alhanahnah, Philipp Leitner, and Ahmed Ali-Eldin. Machine learning containers are bloated and vulnerable. *arXiv preprint arXiv:2212.09437*, 2022.
[55] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.
[56] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 174–185, 2018.

A Appendix

A.1 Container File Systems

Figure 12 shows an example of a container filesystem.

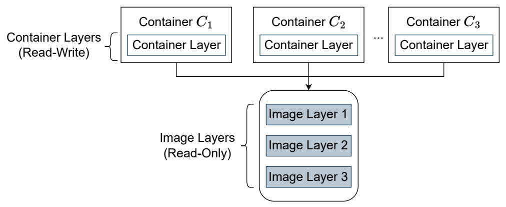


Figure 12: An example of a container filesystem.

A.2 Containers Evaluated

Table 7 shows the details of all 22 containers evaluated in this paper.

A.3 BLAFS Debloating Results

Table 8 shows the debloating results for BLAFS for all 22 containers evaluated in this paper.

Table 7: Details of containers evaluated.

Container	#Workloads[1]	Pull Times
httpd:2.4	4	1B+
nginx:1.27.2	4	1B+
memcached:1.6.32	6	500M+
mysql:9.1	7	1B+
postgres:17	4	1B+
ghost:5.101.3	5	500M+
redis:7.4.1	4	1B+
haproxy:3.0.6	4	1B+
mongo:8.0	7	1B+
solr:9.7.0	4	100M+
rabbitmq:4.0	4	1B+
maven:3.9.9	3	500M+
elasticsearch:8.16.0	4	500M+
eclipse-mosquitto:2.0.20	5	500M+
telegraf:1.30	3	500M+
nextcloud:28.0.12	2	500M+
sonarqube:9.9.7	3	1B+
registry:2.8.3	4	1B+
consul:1.15.4	4	1B+
traefik:v3.2.0	3	1B+
tensorflow-training	1	-
pytorch-training	1	-

[1] We categorized the features into workloads. Each workload may involve multiple features. For example, the `test_index_operations` workload for the `mysql:9.1` involves creating a table, inserting data, creating an index, and querying the data. For the details of the workloads, please refer https://github.com/negativa-ai/BLAFS

Table 8: Debloating results for BLAFS for the all 22 containers sorted by the reduction percentage.

Container	Original (MB)	Debloated (MB)	Reduction
httpd:2.4	141	7	95%
nginx:1.27.2	183	12	93%
memcached:1.6.32	81	9	89%
pytorch-train	16,822	2,902	83%
tensorflow-train	10,952	1,695	85%
mysql:9.1	574	99	83%
postgres:17	415	85	79%
ghost:5.101.3	547	121	78%
redis:7.4.1	112	27	75%
haproxy:3.0.6	98	27	72%
mongo:8.0	815	233	71%
solr:9.7.0	561	195	65%
rabbitmq:4.0	209	73	65%
maven:3.9.9	505	195	61%
elasticsearch:8.16.0	1,241	479	61%
eclipse-mosquitto:2.0.20	14	7	51%
telegraf:1.30	435	223	49%
nextcloud:28.0.12	1,200	761	37%
sonarqube:9.9.7	576	428	26%
registry:2.8.3	24	18	25%
consul:1.15.4	148	137	7%
traefik:v3.2.0	176	169	4%

A.4 Cimplifier Debloating Results

Table 9 shows the debloating results for Cimplifier.

Table 9: Cimplifier Debloating Results of the top 20 most downloaded containers from DockerHub. Containers failed to debloat are not included in the table. Results are sorted by the reduction percentage.

Container	Original (MB)	Debloated (MB)	Reduction
httpd:2.4	141	7	95%
nginx:1.27.2	183	12	93%
eclipse-mosquitto:2.0.20	14	7	51%
telegraf:1.30	435	223	49%
nextcloud:28.0.12	1,200	761	37%
registry:2.8.3	24	18	25%
traefik:v3.2.0	176	169	4%

A.5 SlimToolKit Debloating Results

Table 10 shows the debloating results for SlimToolKit.

Table 10: SlimToolKit Debloating Results of the top 20 most downloaded containers from DockerHub. Containers failed to debloat are not included in the table. Results are sorted by the reduction percentage.

Container	Original (MB)	Debloated (MB)	Reduction
nginx:1.27.2	183	13	93%
memcached:1.6.32	81	9	89%
haproxy:3.0.6	98	27	72%
maven:3.9.9	505	199	61%
telegraf:1.30	435	223	49%
eclipse-mosquitto:2.0.20	14	7	49%
registry:2.8.3	24	19	24%
traefik:v3.2.0	176	170	4%

A.6 Unobserved Workloads for the ML containers

The following list shows the description of the unobserved workloads of two debloated ML containers. For the code of each workloads, please refer to https://github.com/aws/deep-learning-containers/tree/master/test/dlc_tests/container_tests/bin

- Torchdata: Torchdata S3 IO datapipe tests.
- PytorchRegression: Training linear regression model using PyTorch.
- PyTorchwithInductor: Training BertForMaskedLM using PyTorch dynamo and inductor backend.
- Torchaudio: Torchaudio integration datapipe tests.
- TensorBoard: Test TensorBoard.
- TFAddons: Tensorflow addons layers normalizations example.
- TFKerasHVDFP32: Train a CNN model using FP32 type on Horovod.
- TFKerasHVDFAMP: Train a CNN model using AMP type on Horovod.