

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

# Debloating Machine Learning Systems

HUAI FENG ZHANG

Department of Computer Science and Engineering  
Chalmers University of Technology  
Gothenburg, Sweden, 2025

# Debloating Machine Learning Systems

HUAIFENG ZHANG

© Huaifeng Zhang, 2025

ISBN 978-91-8103-312-0

Doktorsavhandlingar vid Chalmers tekniska högskola

Ny serie nr 5769

ISSN 0346-718X

<https://doi.org/10.63959/chalmers.dt/5769>

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Gothenburg

Sweden

Telephone +46 (0)31-772 1000

This thesis has been prepared using L<sup>A</sup>T<sub>E</sub>X.

Printed by Chalmers Digitaltryck  
Gothenburg, Sweden 2025

*To my family*



# Debloating Machine Learning Systems

HUAI FENG ZHANG

Department of Computer Science and Engineering  
Chalmers University of Technology

## Abstract

In recent decades, Machine Learning (ML) has rapidly evolved from an academic pursuit into a cornerstone of modern industries, with applications spanning manufacturing, healthcare, finance, transportation, etc.. The emergence of large language models (LLMs) has further accelerated this growth, driving unprecedented demand for ML systems capable of supporting models of widely varying scales and data modalities. Despite the growing importance of ML systems, the problem of software bloat within them remains underexplored. Software bloat usually refers to unnecessary code, files, or dependencies. It degrades performance, increases resource consumption, and introduces security risks. With the rise of containerized applications as a common deployment method for ML, the problem is further exacerbated: containers must package not only applications but also all associated libraries and dependencies, leading to significant overhead.

This thesis investigates bloat in ML systems across multiple layers, from ML containers to ML shared libraries. This thesis introduces novel techniques to measure, analyze, and mitigate bloat in ML systems. The main contributions are: (A) *MMLB*: a framework for measuring ML bloat that quantifies container-level bloat and identifies its causes, showing that ML containers are substantially more bloated than general-purpose containers and ML shared libraries are the major source of bloat. (B) *BLAFS*: a bloat-aware file system that efficiently and effectively reduces file bloat in ML containers by detecting and removing unused files. (C) *RTrace*: a tracer that accurately identifies functions executed in shared libraries, improving the visibility of shared library execution and enabling precise detection of host-code bloat. (D) *MERGESHUFFLE*: a tool that removes unused code from shared libraries while preserving functionality and improving performance and security. (E) *Negative-ML*: This work reveals that ML shared libraries are different from generic libraries: while the latter contain only host code, ML shared libraries include both host code and device code, the latter targeting GPUs and significantly contributing to their size. *Negative-ML* presents a holistic debloating approach that targets both host and device code, representing the first systematic investigation of device-code bloat. This thesis thus offers both a systematic understanding of software bloat in ML systems and practical techniques to mitigate it, contributing to more efficient, secure, and sustainable deployment of ML in real-world environments.

**Keywords:** Machine Learning Systems, Software Debloating, Software Bloat, Software Engineering, Performance Optimization



# Acknowledgment

I would like to express my deepest gratitude to my supervisor, Ahmed Ali-Eldin Hassan, for his continuous support, guidance and encouragement throughout my Ph.D. journey. I still remember him picking me up at the airport on a cold winter day in 2022, which is the beginning of a truly inspiring academic and personal experience. I am also deeply grateful to my co-supervisor, Philipp Leitner, for his guidance, insightful advice, and unwavering support, especially during some of the most difficult times in my life. I would also like to thank my collaborator, Mohannad Alhanahnah, for the inspiring discussions we had and valuable feedback he provided on my research.

I have been fortunate to work with many wonderful colleagues. Thank you all for creating a friendly and supportive work environment, especially (though not limited to): Hashim, Francisco, Danish, Yixing, Jingyu, Qi, Kåre, Wania, Martin, Umer, Xuechen, Jing, Jacob, Vinh, Yasir, Yenan, Atmane, Philippas, Elad, Marina, Romaric, Magnus, Vincenzo, Rhouma, Masoom, Muoi, and Tomas, my former line manager, whose continuous support and encouragement I deeply appreciate. It has truly been an honor and a pleasure to work with all of you. I would also like to thank those who have left Chalmers but guided me at the beginning of my Ph.D. journey, especially, Christos, Dimitris, Georgia, Shiliang, Bastian and many others. My sincere thanks also go to the administrative staff of the department, especially Clara, Wolfgang and Sanna, for your kind help and support. Additionally, I am also grateful to my friend, Ken, for his generous help and guidance during my early days in Sweden.

Finally, I would like to express my deepest gratitude to my family, especially my parents, who raised me with limited means but endless love and encouragement. You have always supported me in pursuing my dreams. And to my partner, Jing, thank you for your endless love and support, and for sharing both the bitter and sweet moments of life with me. I could not have done this without you.

**Funding Source.** This project is supported by the Knut and Alice Wallenberg Foundation via a Wallenberg AI, Autonomous Systems and Software Program (WASP) PhD grant.

Huafeng Zhang  
Gothenburg, October 2025



# List of Publications

## Appended publications

This thesis is based on the following publications:

- [A] **H.F. Zhang**, M. Alhanahnah, F. Abdulqadir-Ahmed, D. Faith, P. Leitner, and A. Ali-Eldin  
“Machine Learning Systems are Bloated and Vulnerable”  
*Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, 2024, 8(1): 1-30.
- [B] **H.F. Zhang**, M. Alhanahnah, P. Leitner, and A. Ali-Eldin  
“BLAFS: A Bloat Aware Container File System”  
*Proceedings of the 16th ACM Symposium on Cloud Computing (SoCC)*, 2025.
- [C] **H.F. Zhang**, and A. Ali-Eldin  
“RTrace: Towards Better Visibility of Shared Library Execution”  
*Proceedings of the 33rd Network and Distributed System Security (NDSS) Symposium*, 2026.
- [D] **H.F. Zhang**, and A. Ali-Eldin  
“MERGESHUFFLE: Debloating Shared Libraries for Improved Performance and Security”  
*Under submission*.
- [E] **H.F. Zhang**, and A. Ali-Eldin  
“The Hidden Bloat in Machine Learning Systems”  
*Proceedings of the 8th Conference on Machine Learning and Systems (MLSys)*, 2025.



# Research Contribution

**Table 1:** Summary of personal contributions per paper according to the Contributor Roles Taxonomy (CRediT)<sup>1</sup>.

	Conceptualization	Data Curation	Formal Analysis	Funding Acquisition	Investigation	Methodology	Project Administration	Resources	Software	Supervision	Validation	Visualization	Writing - Original Draft	Writing - Review & Editing
Paper A	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓
Paper B	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓
Paper C	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓
Paper D	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓
Paper E	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓



# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgement</b>	<b>vii</b>
<b>List of Publications</b>	<b>ix</b>
<b>Personal Contribution</b>	<b>xi</b>
<b>Part I: Thesis Overview</b>	<b>1</b>
1 Introduction . . . . .	3
2 Software Bloat . . . . .	4
2.1 Causes of Software Bloat . . . . .	4
2.2 Impacts of Software Bloat . . . . .	6
3 Machine Learning . . . . .	6
3.1 Machine Learning: An Overview . . . . .	6
3.2 Machine Learning Systems . . . . .	7
3.3 Software Bloat in Machine Learning Systems . . . . .	8
4 Cloud Computing and Containers . . . . .	8
4.1 Cloud Computing . . . . .	8
4.2 Containers . . . . .	9
4.3 Software Bloat in Containers . . . . .	10
5 Software Debloating . . . . .	11
5.1 Source Code Level Debloating . . . . .	11
5.2 Binary Level Debloating . . . . .	12
5.3 Container Level Debloating . . . . .	13
6 Thesis Objectives . . . . .	13
7 Thesis Contributions . . . . .	15
7.1 Chapter A: Measuring Bloat in ML Containers (Paper A)	16
7.2 Chapter B: Debloating ML Containers (Paper B) . . . . .	16
7.3 Chapter C: Identifying Bloat in Host Code (Paper C) . . . . .	17
7.4 Chapter D: Debloating Host Code (Paper D) . . . . .	18
7.5 Chapter E: Debloating Device Code (Paper E) . . . . .	18
8 Conclusions and Future Directions . . . . .	19
<b>Part II: Collection of Papers</b>	<b>21</b>

<b>A</b>	<b>Machine Learning Systems are Bloated and Vulnerable</b>	<b>23</b>
1	Introduction . . . . .	24
2	Background . . . . .	26
	2.1 Software Debloating . . . . .	26
	2.2 ML Systems . . . . .	27
	2.3 ML containers . . . . .	28
3	Experiment setup and Framework . . . . .	29
	3.1 Model Selection and Container Creation . . . . .	29
	3.2 Container Debloating . . . . .	30
	3.3 Analysis Framework . . . . .	31
4	Measuring Bloat in ML Systems . . . . .	35
	4.1 RQ1: Prevalence of Bloat . . . . .	35
	4.2 RQ2: Runtime Performance Comparison . . . . .	38
	4.3 RQ3: Deployment Performance Comparison . . . . .	38
	4.4 RQ4: Sources of Bloat . . . . .	40
	4.5 RQ5: Vulnerabilities and Bloat . . . . .	45
	4.6 RQ6: Impact of Package Dependency and Package Reach . . . . .	47
5	Discussion . . . . .	50
6	Related Work . . . . .	51
7	Conclusion . . . . .	52
<b>B</b>	<b>BLAFS: A Bloat-Aware Container File System</b>	<b>53</b>
1	Introduction . . . . .	54
2	Background . . . . .	56
	2.1 Container File Systems . . . . .	56
	2.2 Container Debloating . . . . .	56
	2.3 Lazy Loading Snapshotters . . . . .	57
	2.4 Containers Cold-Starts . . . . .	57
3	BLAFS Design . . . . .	58
	3.1 The Debloating Layer . . . . .	59
	3.2 The Reloading Layer . . . . .	61
	3.3 The Converter . . . . .	63
	3.4 Layer-Sharing Mode Selection Strategy . . . . .	63
	3.5 Package-Dependency-Based Expansion . . . . .	63
4	Evaluation . . . . .	64
	4.1 Comparing to Deblockers . . . . .	65
	4.2 BLAFS for Serverless Computing . . . . .	66
	4.3 Security Hardening. . . . .	68
	4.4 Evaluation of Mode Selection Strategy . . . . .	70
	4.5 Comparing to Lazy-Loading . . . . .	71
	4.6 Combining BLAFS with Lazy-Loading . . . . .	73
	4.7 Evaluation of PDBE . . . . .	74
	4.8 Debloating and Reloading Overheads . . . . .	76
5	Discussion . . . . .	78
6	Related Work . . . . .	79
7	Conclusion . . . . .	79

<b>C</b>	<b>RTrace: Towards Better Visibility of Shared Library Execution</b>	<b>81</b>
1	Introduction . . . . .	82
2	Background and Related Work . . . . .	84
	2.1 Supply Chain Security . . . . .	84
	2.2 Shared Library . . . . .	84
	2.3 Library Call Tracing . . . . .	85
	2.4 Binary Analysis . . . . .	86
3	A Call for a Better Tracer . . . . .	86
4	RTrace: An accurate Tracer . . . . .	88
	4.1 Overview . . . . .	88
	4.2 Light Mode . . . . .	89
	4.2.1 Library Detection . . . . .	89
	4.2.2 Branch-Return Instrumentation . . . . .	90
	4.2.3 Adaptive Boundary Detection . . . . .	91
	4.2.4 Function Coverage Report . . . . .	91
	4.3 Rich Mode . . . . .	92
	4.3.1 Function Call Instrumentation . . . . .	92
	4.3.2 Call Graph Report . . . . .	94
5	Experiment . . . . .	96
	5.1 Experiment Setup . . . . .	96
	5.2 Accuracy and Performance of the Light Mode . . . . .	97
	5.3 Accuracy and Performance of the Rich Mode . . . . .	98
	5.4 Impacts of Boundary Detection Algorithm . . . . .	101
	5.5 Ablation Study . . . . .	102
	5.6 Analysis of Missed Functions . . . . .	104
	5.7 Application of RTrace in Security Domain . . . . .	105
	5.7.1 Malicious Python Package Detection . . . . .	105
	5.7.2 XZ Backdoor Case Study . . . . .	106
6	Discussion and Future Work . . . . .	108
7	Conclusion . . . . .	109
<b>D</b>	<b>MERGESHUFFLE: Debloating Shared Libraries for Improved Performance and Security</b>	<b>111</b>
1	Introduction . . . . .	112
2	Background and Related Work . . . . .	113
	2.1 Shared Libraries: the ELF format . . . . .	113
	2.2 Shared Library Debloating . . . . .	114
	2.3 Program Loading and Cold Start . . . . .	114
3	MERGESHUFFLE Design . . . . .	115
	3.1 Motivation and Problem Definition . . . . .	115
	3.2 Overview of MERGESHUFFLE . . . . .	116
	3.3 MERGE: Balance Between Memory Usage and Run Time	118
	3.4 SHUFFLE: Reduce File Size under Congruence Constraint	119
	3.4.1 Range Shuffling Problem . . . . .	119
	3.4.2 LINEARSHUFFLE . . . . .	119
	3.4.3 TSPSHUFFLE . . . . .	120
	3.4.4 SHUFFLE . . . . .	121
	3.5 Implementation . . . . .	123
4	Experiments . . . . .	123

4.1	Overview of Size, Memory and Run time Reduction . . .	124
4.2	Impact of Threshold $g$ . . . . .	125
4.3	Analysis of Size Reduction . . . . .	127
4.4	Analysis of Memory Reduction . . . . .	128
4.5	Analysis of Run Time Reduction . . . . .	129
4.6	Reconstructed Yet Shareable . . . . .	130
4.7	Analysis of Impacts on Security . . . . .	131
5	Discussion . . . . .	132
6	Conclusion . . . . .	133
<b>E The Hidden Bloat in Machine Learning Systems</b>		<b>135</b>
1	Introduction . . . . .	136
2	Background and Related Work . . . . .	137
2.1	ML Frameworks . . . . .	137
2.2	Software Bloat . . . . .	138
2.3	Software Debloating . . . . .	139
3	Methodology . . . . .	140
3.1	Kernel Detector . . . . .	140
3.2	Kernel Locator . . . . .	142
4	Experiments . . . . .	143
4.1	Overview of Bloat in ML Frameworks . . . . .	144
4.2	Shared Library Level Analysis . . . . .	145
4.3	Function and Element Level Analysis . . . . .	146
4.4	Runtime Performance Analysis . . . . .	149
4.5	Evaluation on Different GPUs . . . . .	150
4.6	Performance of Negativa-ML . . . . .	152
5	Discussion . . . . .	153
6	Conclusion . . . . .	153
<b>Bibliography</b>		<b>155</b>

# Part I

## Thesis Overview



---

# 1 Introduction

In recent decades, Machine Learning (ML) has rapidly transcended the boundaries of academia and research labs to become an integral component of numerous industries. Today, ML applications span numerous sectors, including manufacturing, healthcare, finance, transportation, and entertainment, among others [1]. The advent of large language models (LLMs) has further showcased ML's potential, driving unprecedented demand for ML solutions [2]. ML models vary significantly in size and scale, from small models with limited parameters designed for edge devices to large models with billions of parameters that operate within data centers requiring substantial computational resources. Moreover, ML is capable of handling diverse tasks across multiple data modalities, including texts, images, audio, video, and multimodal data. This increasing demand, alongside the growing complexity and variety of ML use cases, presents considerable challenges for the underlying ML systems that support these models [3].

ML systems are data-driven, where the data plays a pivotal role in the systems. This characteristic of ML systems makes them different from traditional software systems in many ways. First, ML systems prioritize improving data quality. Second, ML systems are also adaptive and iterative due to the frequent changes in data, requiring testing and versioning of both code and data. Additionally, managing massive ML models with billions of parameters requires significant memory, posing production challenges, especially for deployment on edge devices. Furthermore, the lack of explainability and interpretability in ML models requires specialized tools for monitoring and debugging in production [4]. These factors add significant complexity to ML systems, causing them to inherit all the maintenance challenges of traditional software systems along with a unique set of ML-specific issues, which makes them particularly susceptible to be bloated [5].

Software bloat generally refers to the resource overhead caused by excessive functionality, often resulting from the use of highly general components or dependency on deeply layered, standardized frameworks [6]. Software bloat is a long-standing problem in software engineering. It is exemplified by Joel Spolsky's 80/20 myth, which states that only 20% of a software's features are used by 80% of its users [7]. Software bloat can lead to performance degradation, security vulnerabilities, and high resource consumption. This problem is further exacerbated by the current trends in cloud computing, which has led to the widespread adoption of containerized applications. Containerized applications, or containers, are lightweight, portable, and isolated environments that package both the application and its dependencies. Due to their ease of deployment and scalability, containers are extensively used in modern software systems. However, they tend to be more bloated, as they must also include all related libraries and dependencies to maintain an isolated environment.

While most research focuses on adding new features to existing systems, this thesis takes a different approach by removing unnecessary elements. Specifically, this thesis addresses the problem of bloat in ML systems — a unique challenge due to their special capacities to introduce bloat, yet one that remains underexplored in current literature. This thesis thoroughly investigates sources

of bloat in ML systems and introduces novel techniques for debloating. We propose a novel, multi-level debloating framework for ML systems.

We start with an in-depth study to understand the extent of bloat in ML systems. We propose MMLB, a framework for *Measuring Machine Learning Bloat*, to quantify the bloat in ML containers and identify the main causes of this bloat. The results show that ML containers are significantly more bloated than generic containers, with larger sizes that consume substantial storage and network resources. In response, we introduce BLAFS, a *Bloat-Aware File System* designed to reduce file bloat in ML containers by identifying and removing unnecessary files.

We further investigate the root causes behind the large size of ML containers and find that a major contributor is the presence of large shared libraries within ML frameworks. Therefore, we delve into shared libraries. We analyze the structure of ML shared libraries and observe that, unlike traditional shared libraries which contain only host code (executed on CPUs), ML shared libraries include both host code and *device code* designed to run on GPUs for accelerated computing. Device code account for the majority of the large size of ML shared libraries. To tackle bloat in host code, we propose RTrace and MERGESHUFFLE for identify and remove bloat from host code. Building on these tools, we introduce Negative-ML, a comprehensive approach that identifies and removes bloat in both host and device code within ML shared libraries.

Together, MMLB, BLAFS, RTrace, MERGESHUFFLE and Negative-ML form the foundation of a comprehensive online debloating framework for ML systems, capable of performing debloating at both the container and code levels. While these approaches are targeted for ML systems, MMLB, BLAFS, RTrace, and MERGESHUFFLE can also be potentially applied to debloat traditional software systems.

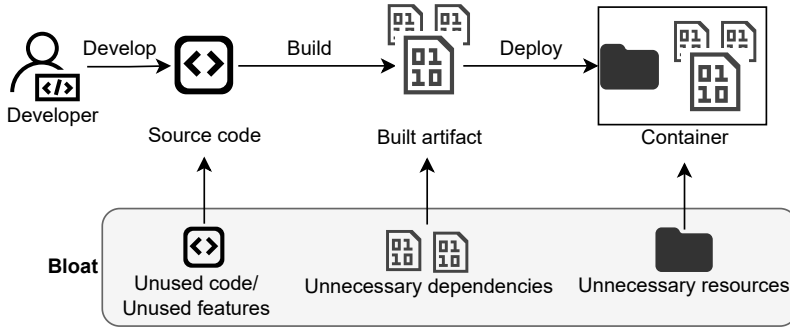
In the remainder of this overview, we dive deeper into the software bloat problem in ML systems. We first discuss the software bloat problem in §2. Then we introduce machine learning systems and discuss software bloat in the context of machine learning systems in §3. After that, we introduce the modern cloud computing technology, which is widely used in machine learning deployment in §4. Subsequently, we introduce the software debloating techniques at source code, binary and container levels in §5. Next, we present the objectives and contributions of this thesis in §6 and §7. Finally, we present conclusions of this thesis in §8.

## 2 Software Bloat

This section provides an overview of the software bloat problem, its causes, and its impacts.

### 2.1 Causes of Software Bloat

Software bloat is a long-standing problem. Although there is no strict definition of software bloat, it generally refers to unnecessary code, features, resources, or dependencies that add to the size and complexity of software systems [6]. Software bloat can be categorized into two main types: Type I bloat and Type II bloat [8]. Type I bloat is entirely unnecessary and can be eliminated



**Figure 1:** In modern DevOps practices, each stage introduces new source of software bloat.

without affecting the program’s behavior for any intended use cases. A common example of Type I bloat is dead or unreachable code, which can be effectively removed through static analysis by compilers. Type II bloat is dependent on end-use; code may qualify as Type II bloat depending on how the program is utilized by its users. Due to its context-dependent nature, Type II bloat is more challenging to identify and eliminate. Therefore, this thesis focuses on Type II bloat, as it is not only more challenging to eliminate effectively but also more prevalent.

Software bloat exists throughout the entire software stack, from operating system [9], executable binaries [10, 11], shared libraries [12, 13], web-applications [14, 15], Android applications [16], and even firmware [17]. In modern DevOps practices, software is developed, built, and deployed continuously [18], with each stage inevitably introducing software bloat as shown in Figure 1. During development, code bloat can arise from adding unnecessary features. Poor code design can also cause redundant or dead code [19]. Additionally, prioritizing faster time-to-market can lead to “technical debt”, where trade-offs in code quality and efficiency are made for quicker release cycles, contributing to long-term bloat [20].

In the build process, dependencies are often included without careful scrutiny, leading to larger application sizes. Dependency management tools may pull in entire libraries or frameworks even if only small parts are needed, leading to increased build size [21]. Furthermore, a complex dependency graph can lead to *dependency hell* [22], where managing dependencies becomes challenging due to conflicting versions or compatibility issues. This situation increase updates and maintenance overhead, often requiring extensive manual intervention to resolve conflicts [23].

In deployment, nowadays containerized applications (containers) and virtualized environments are widely used to ensure portability and isolation. However, these containers often include all dependencies and resources to create a self-contained environment, which also includes many unnecessary components, leading to bloat [24]. For example, a simple Nginx container can be 70MB in size, while the actual Nginx binary is only 1MB<sup>2</sup>.

<sup>2</sup>[https://hub.docker.com/\\_/nginx/tags?name=1.27](https://hub.docker.com/_/nginx/tags?name=1.27)

## 2.2 Impacts of Software Bloat

Software bloat has several negative impacts on software systems, including performance degradation, wasted resource, security vulnerabilities, and increased maintenance costs. As Wirth's law suggests, software performance often declines faster than hardware improvements can compensate [25]. Software bloat can make the software more challenging to run on mobile devices, embedded systems, or edge devices with limited resources. Redundant features, unused code, and extraneous libraries expand the attack surface, increasing the potential for security vulnerabilities. Moreover, outdated dependencies included for backward compatibility may lack necessary security patches, further exacerbating risk [26]. Bloated software is harder to maintain, as it requires more time and resources for updates, testing, and debugging. Larger codebases are more complex, making it challenging to identify and isolate issues. Additionally, with each new update, there is an increased likelihood of compatibility issues due to unnecessary or outdated components, resulting in longer testing cycles and higher maintenance costs [27]. The current software engineering practices lack a feedback loop to eliminate bloat. With continuous adding new features, software bloat accumulates over time. The resulting overhead and maintenance challenges lead to delayed deliveries, budget overruns, and ultimately potential project failures [28].

## 3 Machine Learning

Machine learning has been playing a key role in our daily life. This section introduces the basic concepts of machine learning, its applications, the underlying machine learning systems and the software bloat in these systems.

### 3.1 Machine Learning: An Overview

Machine Learning involves creating models that learn from data by training to minimize a loss function. Major Data-driven ML approaches include *supervised learning*, where models learn from labeled data; and *unsupervised learning*, which learns patterns in unlabeled data. Over decades of research, a variety of ML models have been developed. Traditional ML models use statistical methods to identify patterns in data, such as decision trees, and support vector machines [29, 30]. Advances in hardware have made computers significantly more powerful, while the explosion of the internet has ushered in the era of big data. With vast amounts of data and increased computational capacity, *deep learning*, which uses neural networks with multiple layers to capture complex patterns in large datasets, has emerged as the dominant approach in ML [31].

Deep learning works by using artificial neural networks with multiple layers, often called deep neural networks, to learn patterns in data. Each layer of the network consists of neurons (or nodes) that are connected to neurons in the previous and subsequent layers. During training, data is fed into the network, and each layer processes it by applying a set of weights and biases to transform the input. The model then compares its output to the actual result, calculating an error using a loss function. The network adjusts the weights and biases across layers to minimize this error, a process called *backpropagation* [32–34].

This training continues iteratively, with the network learning to make better predictions by adjusting its parameters [35]. Deep learning has excelled in areas like computer vision, speech recognition, natural language processing, and content generation. With increasing parameters, advanced model architectures, and access to larger datasets, these models have demonstrated capabilities previously considered beyond machine reach – this phenomenon, known as the *scaling law*, suggests that neural network performance improves as model size and dataset volume increase [36]. This principle has driven the development of *large language models* (LLMs), such as GPT-3, which contain billions of parameters and trained on billions of words of text data [37]. LLMs have once again expanded the boundaries of ML, enabling these models to generate human-like text, answer questions, and even write code [38].

### 3.2 Machine Learning Systems

To support the growing complexity and scalability of ML models, the infrastructure around ML models has had to evolve significantly. ML systems manage the entire lifecycle of ML models, from development to deployment and ongoing maintenance. Unlike traditional software systems, which primarily rely on code, ML systems are data-driven and iterative. They encompass not only the ML model itself but also handle various aspects of model management, including data preprocessing, model training, deployment, monitoring, and continuous updates. As a result, ML systems mainly involve the following stages [39]:

- **Data preprocessing:** Involves collecting, cleaning, and storing large volumes of data, often using distributed systems and cloud infrastructure to handle the scale efficiently.
- **Model training:** Training the ML model to refine its parameters and improve its accuracy. This process can involve massive datasets and complex models, requiring significant computational resources. To accelerate training, specialized hardware such as GPUs and TPUs is used, alongside distributed computing to scale the process [40].
- **Model deployment:** Deploying the trained model into a production environment where it can handle real-time data and deliver inference. For large models, accelerators and distributed computing are also used to optimize the inference process for speed and efficiency.
- **Monitoring and maintenance:** Continuously tracking model performance and ensuring it meets reliability standards. This step includes detecting issues like data drift, where changing data patterns can impact the model's accuracy.
- **Model updates and retraining:** Iteratively updating or retraining models based on new data to maintain relevance and accuracy over time.

Due to these processes, ML systems are generally more resource-intensive and complex than traditional software, requiring substantial computational resources and often specialized hardware. They also need robust monitoring, testing, and versioning of both the data and the model itself, which can make their maintenance more challenging [41].

### 3.3 Software Bloat in Machine Learning Systems

Machine learning systems are prone to introduce bloat, because they have all the maintenance challenges of traditional software systems along with a unique set of ML-specific issues. In ML systems, ML code is only a small fraction of the system [5, 42], while the surrounding infrastructure such as data collection, feature extraction, serving infrastructure is vast and complex. Sculley et al. [5] identify various sources of technical debt within ML systems. For instance, ML systems often struggle to maintain strict abstraction boundaries because their behavior depends heavily on external data. With many components entangled, any change in the system, such as modifications to inputs, hyperparameters, or data selection, can impact all other parts, denoted by the “CACE” principle: Changing Anything Changes Everything [43]. Furthermore, unlike traditional software that deals mainly with code dependencies, ML systems also have data dependencies, adding additional layers of complexity and potential bloat. For example, if an input signal was previously mis-calibrated, the model would likely adapt to these mis-calibrations, and a silent update correcting the signal could have sudden, significant effects on the model’s performance.

To optimize performance, ML systems often depend on high-performance yet inflexible kernels running on accelerators. While effective for efficiency, these kernels reduce the expressiveness, maintainability, and programmability of the systems, limiting the exploration of new machine learning research ideas [39]. Moreover, although advancements in neural network architectures and hardware accelerators, the actual wall-clock inference latency observed in practice is often higher than that promised by the advancements due to the overhead introduced by the ML system [44].

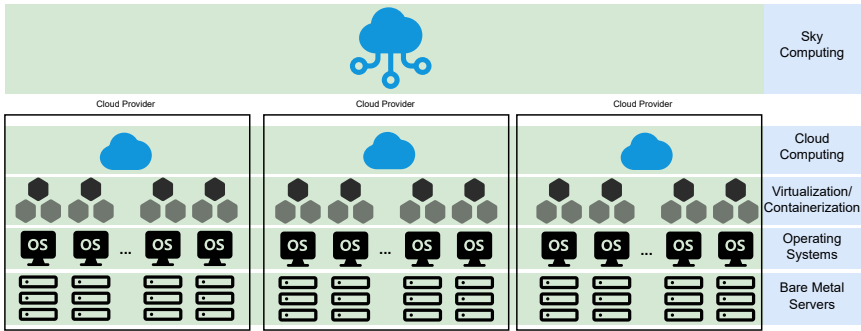
## 4 Cloud Computing and Containers

Cloud computing has gained widespread adoption due to its flexibility, scalability, and cost-effectiveness. The ease of use and scalability offered by cloud computing come from multiple layers of abstraction that conceal the underlying infrastructure complexity, as illustrated in Figure 2, with each layer potentially introducing bloat. This section first introduces cloud computing and its evolution, followed by a discussion on containers, a key technology that enables cloud computing. Finally, we explore the software bloat problem in containers.

### 4.1 Cloud Computing

Cloud computing offers on-demand access to computing resources, storage, and a variety of services over the internet. This approach allows organizations to scale their applications effortlessly, manage vast datasets, and reduce the costs and complexities associated with maintaining physical infrastructure. Cloud computing has been widely adopted and become the foundation for modern software development and deployment practices, as shown by that 98% of organizations are using some form of cloud computing [45].

Serverless computing, a new execution model in cloud computing, has gained significant popularity in recent years due to its lightweight nature and ease of management. Serverless computing abstracts infrastructure management



**Figure 2:** The modern software stack is complex and multi-layered, spanning bare metal servers, operating systems, virtualization/containerization, cloud computing, and edge computing. Each of these layers can introduce and compound bloat.

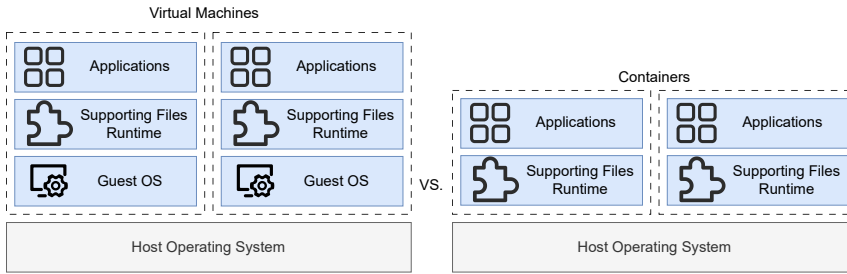
completely, allowing developers to focus solely on writing and deploying code [46]. In a serverless architecture, the cloud provider automatically provisions and scales the computing resources as needed, enabling applications to scale seamlessly in response to user demands. Serverless is often billed on a per-execution basis, making it cost-effective for applications with unpredictable or intermittent usage patterns [47]. By simply deploying discrete functions that execute in response to events (like user requests or data changes), developers can achieve rapid scalability without managing the underlying servers. However, serverless is not without challenges: cold start latency [48], execution time limits [49], and tighter integration with specific cloud providers can pose limitations in certain applications [46].

The widespread adoption of cloud computing has led many providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), to offer a range of cloud services [50]. Aiming to enable seamless interoperability across multiple cloud providers, sky computing is an emerging paradigm that builds on the foundations of cloud computing [51]. Unlike traditional multi-cloud strategies, where organizations manage and integrate services from different cloud platforms separately, sky computing envisions a unified cloud ecosystem in which resources from various providers can be used seamlessly, without the need for complex manual integration.

In reality, both public and private clouds are heavily bloated: instances can reach sizes of several gigabytes, “fast” boot times may still take minutes, and memory consumption for running even a basic service (such as a static web server) can be disproportionately high [52].

## 4.2 Containers

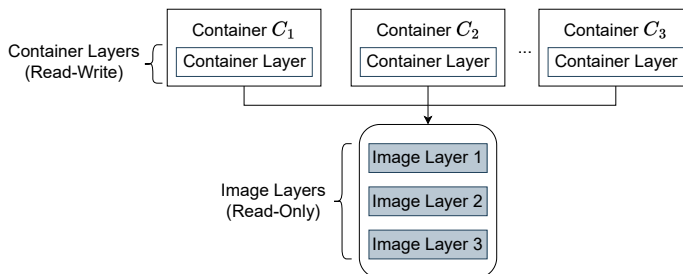
Containers are one of the key technologies that enable cloud computing [53]. Containers are lightweight virtualization technologies used to package code and dependencies in a virtual environment that can be easily migrated and redeployed across different clusters or computing environments [54]. Unlike virtual machines, which virtualize entire operating systems, containers share the host OS kernel but maintain their own isolated environment for the application and



**Figure 3:** Each virtual machine has its own guest OS, while containers share the host OS but maintain their own isolated environment for the application and its runtime. Containers are more lightweight and faster to start than virtual machines.

its dependencies. This lightweight structure allows containers to start quickly and consume fewer resources, making them the key technology that supports cloud computing environments [55]. In Linux, containers are implemented based on the techniques such as namespaces, control groups (cgroups), and union filesystems [54].

This thesis tackles the problem of container bloat from the perspective of container filesystems. Containers use union filesystems, like OverlayFS [56], to create layered, lightweight, and flexible filesystems. This layered approach enables shared images, where multiple containers can use a common base image, and any changes or updates are layered on top without modifying the original. The union filesystem of a container consists of a writable *container layer* stacked on top of multiple read-only *image layers*, as depicted in Figure 4. The container layer manages all write operations using a copy-on-write mechanism, storing any newly created or modified files. The image layers are read-only. When a file is accessed, the filesystem first searches the container layer; if the file is not found, it proceeds through the image layers in order, from top to bottom, until the file is located or an error occurs. This approach optimizes storage use and makes updates faster but also introduce bloat as discussed in the next section.



**Figure 4:** An example of a container filesystem.

### 4.3 Software Bloat in Containers

The convenience of containers comes at the cost of efficiency. To ensure consistent execution across environments, containers typically package all

dependencies, but this often results in the inclusion of unnecessary files, libraries, and code, contributing to software bloat.

The software bloat problem is further exacerbated by the use of layered filesystems, which accumulate bloat across image layers and propagate it to derived containers. Containers are typically built on top of a *base container image*, where the container inherits all layers from the base image. Any existing bloat in the base image is inherited by the new container. Moreover, additional bloat can be introduced as new layers are added. This accumulated bloat presents several challenges: it wastes resources in both cloud data centers and resource-constrained edge devices [57,58]; it also increases the attack surface [59], degrades performance, particularly in serverless computing, where bloated containers prolong provisioning times and worsen cold start latency [60–62].

## 5 Software Debloating

In this section, we present an overview of this software debloating, highlighting some limitations of previous approaches. These debloating tools either remove Type I bloat (universally unnecessary code) or Type II bloat (use-case dependent unnecessary code) from software. According to the input, we categorize the debloating techniques into three levels: source code level, binary level, and container level.

### 5.1 Source Code Level Debloating

Source code-level debloating requires access to the software’s source code. It is typically specific to certain programming languages, such as C [26,63–67], Java [66] or JavaScript [68,69]. This approach uses static analysis, combined with language syntax and semantics, to identify and eliminate unnecessary code or dependencies [21].

For Type I bloat, such as dead or unreachable code, source code-level debloating can be achieved through call graph analysis and data flow analysis to statically identify and remove such bloat [70]. Modern compilers are capable of performing this type of analysis with a high degree of accuracy [71].

For Type II bloat, however, source code level debloating aims to remove unnecessary code for specific use cases. The use cases are usually specified in the form of specific inputs, configurations or workloads. This process involves mapping the code to the corresponding use case and retaining only the code necessary for that use case. This can be accomplished by annotating the source code to tag optional or unused features [26], using partial evaluation based on specified configurations [11,63,72], or applying dynamic tracing to run the software with specific workloads to identify necessary code [73,74]. Furthermore, compiler-assisted analysis can be used for source code level debloating by compiling the source code into an intermediate representation (e.g., LLVM IR) and analyzing it to identify unnecessary code [11,63]. However, compiler-assisted analyses are often tightly coupled with specific compilers and may require users to be familiar with the toolchain configuration (e.g., compiler flags). Moreover, they may impose manual effort on users, such as explicitly identifying the boundary between input parsing and program logic [63].

Source code debloating can reduce the size, complexity, and attack surface of software applications by removing unnecessary code. The debloating results are more auditable as the changes are made directly to the source code. However, the requirement for source code access limits its applicability to proprietary software or software with complex build systems [8, 75].

## 5.2 Binary Level Debloating

Binary level debloating focuses on removing unnecessary code directly from compiled binaries without the need for source code. This approach is especially beneficial for applications where source code may not be accessible, such as proprietary software, closed-source libraries. Binary level debloating targets a variety of application types, including Android applications (APKs) [16, 76], Java packages [77, 78], shared libraries [13, 79–83], and standalone executable binaries [10, 84].

For Type I bloat, binary-level debloating utilizes static analysis. It disassembles the binary, extracts functions, and constructs a function call graph, then performs reachability analysis to identify and eliminate dead code. Code removal can be achieved by rewriting the binary to strip out unused parts [13] or by recompiling the binary to generate a new, optimized version [80].

For Type II bloat, binary-level debloating employs dynamic analysis to profile the binary’s execution and identify unused code. This execution profiling can be based on typical user interactions or predefined workloads [16, 78]. Static analysis may also be incorporated to improve accuracy and minimize the risk of removing necessary code [10, 79]. Most binary debloating tools remove unused code from the binary, producing a debloated version with smaller size. Some tools, however, do not remove the code entirely; instead, they disable it from loading at runtime, allowing it to be re-enabled if needed [79, 83].

Although a wide range of debloating tools have been proposed in the literature, most of them target debloating executable binaries [10, 11, 26, 64, 67, 72, 85]. Debloaters for executable binaries typically have a well-defined scope, as each application usually consists of a single executable. However, these approaches largely overlook the shared libraries on which such binaries depend. In fact, shared libraries are often substantially larger than the binaries themselves. For instance, in the case of Nginx, its dependent shared libraries are approximately three times larger than the binary. And training a machine learning model with TensorFlow may involve approximately 400 shared libraries with a total size more than 2 GB [86]. Existing approaches that support shared library debloating either require source code access for recompilation [12, 84], or fail to address Type II bloat [13, 79]. Importantly, bloat in a shared library has a more pronounced impact than bloat in an executable binary, since the inefficiencies and potential vulnerabilities introduced by the shared library can propagate transitively along the dependency chain to all binaries that rely on the affected library.

Compared with source code-level debloating, binary-level debloating is more challenging. Binary debloating mostly rely on profiling data to identify unused code. However, the profiling workloads may not cover the actual use cases of the software. This means that any missed code paths during profiling may lead to the removal of code that is needed in less-common scenarios, which can

cause functionality issues. However, binary debloating are more applicable to a wider range of software applications because they do not require access to the source code.

### 5.3 Container Level Debloating

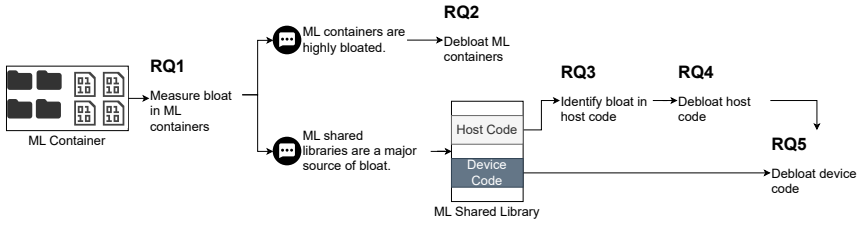
Containers bundle applications with their dependencies, making it easy to run and scale applications across various environments. However, these container images are often bloated with unnecessary files and libraries, which not only increases storage requirements, decrease performance, but also expands the attack surface, increasing security vulnerabilities. Container debloating aims to address these issues by removing unneeded components, optimizing image size, and reducing security risks [24].

Container debloating techniques can be broadly categorized into two main categories: *restricting system calls* and *reducing image size*. Confine [87] and SPEAKER [88] are two representative tools that focus on restricting system calls. Confine restricts system calls within containers to reduce the kernel’s attack surface. SPEAKER divides container execution into distinct phases—booting and running. This split-phase method applies different system call restrictions to each phase, tailored specifically to the application’s requirements in each state. By doing so, these tools reduce the potential for privilege escalation attacks and tighten security within container environments. In terms of the image size-reduction category, SlimToolKit [89] combines static and dynamic analyses to identify and remove unnecessary container image layers and files, retaining only essential elements and yielding smaller, leaner container images. This process enhances performance by cutting down storage requirements, which is advantageous for faster container deployment and data transfer efficiency. Cimplifier [90], goes further by partitioning containers into smaller, isolated containers based on functional roles, adhering to the principle of least privilege. This partitioning ensures that each container only contains the resources necessary for its designated role, minimizing cross-component dependencies and reducing the scope for potential vulnerabilities.

The container debloating methods for image size reduction face the same challenge as binary debloating: the profiling workloads may not cover the actual use cases of the software, which may lead to the removal of necessary files. Our evaluation of SlimToolkit and Cimplifier on several ML containers demonstrates that these tools often produce non-functional containers. For example, they fail to capture certain files accessed during container startup, which leads to broken containers. In addition, these two tools may break the layered structure of the container filesystem, which in some cases leads to debloated containers that are even larger in size than the original ones.

## 6 Thesis Objectives

As discussed in §3, ML systems have a unique tendency to accumulate bloat. They face all the maintenance challenges of traditional software systems, along with a distinct set of ML-specific issues. However, the bloat in ML systems has not been systematically studied, and its impact on performance, security, and operational costs remains unclear. This thesis examines the issue of bloat



**Figure 5:** Connections between the research questions in the thesis. The findings of RQ1 lead to the subsequent research questions RQ2, RQ3, RQ4 and RQ5.

in ML systems from ML containers to ML shared libraries. Figure 5 illustrates the research flow.

Firstly, this thesis starts with Research Question 1 (RQ1) as shown below. To answer RQ1, this thesis conducts an empirical study to measure ML container bloat, quantifying to what extent are these systems bloated versus traditional software, and to identify the main causes of this bloat. This empirical study leads to two key findings as shown below, which motivate the subsequent research questions.

#### Research Question 1

*To what extent are machine learning systems affected by bloat? (Measure bloat in ML containers)*

- *Finding 1:* ML containers are highly bloated, with many unnecessary files and dependencies that significantly increase their sizes, leading to performance degradation and security vulnerabilities.
- *Finding 2:* ML workloads use many large shared libraries, which are a major source of bloat even after removing unnecessary files.

Based on Finding 1, this thesis then addresses Research Question 2 (RQ2). To answer RQ2, this thesis analyzes the limitations of state-of-the-art container debloating methods in the context of ML containers and proposes a novel technique that effectively reduce bloat from ML containers.

#### Research Question 2

*Why do existing debloating approaches fail for ML containers and how can novel approaches be developed to address these limitations? (Debloat ML containers)*

Building on Finding 2, this thesis further investigates debloating ML shared libraries. ML libraries encapsulate the core functionalities of many ML frameworks and applications, and they constitute the majority of the overall framework size [86]. In practice, ML shared libraries can range from hundreds of megabytes to several gigabytes, much larger than generic shared libraries in traditional software [59]. A detailed investigation into the structure of ML shared libraries reveals that they are unique in containing both host code, which executes on CPUs, and device code, which executes on accelerators such as GPUs.

Regarding the host code, although debloating techniques for host code in shared libraries have been previously studied, the problem remains unsolved: Existing approaches often suffer from limitations such as breaking library functionality, incurring significant overhead, or requiring complex configurations, as highlighted in a recent study [8]. These limitations motivate the development of new debloating methods for host code, leading to Research Questions 3 (RQ3) and 4 (RQ4). To address RQ3, this thesis first evaluates existing techniques for tracing executed functions in shared libraries and identifies key limitations that hinder their applicability to ML workloads. Then it introduces a novel method for accurately identifying executed functions in ML shared libraries. To address RQ4, this thesis proposes a new approach for effectively removing bloat from host code, while enhancing performance and security.

#### Research Question 3

*How can bloat in the host code of ML shared libraries be identified? (Identify bloat in host code)*

#### Research Question 4

*How can host code in ML shared libraries be debloated? (Debloat host code)*

As for the device code, no prior work has addressed the problem of debloating device code in ML shared libraries. This gap motivates Research Question 5 (RQ5). To answer RQ5, this thesis explores techniques for detecting and removing bloat in device code, building upon the methods proposed for host code (RQ3 and RQ4), and evaluates the impact of device code debloating on performance, such as run time and memory usage.

#### Research Question 5

*How can device code in ML shared libraries be debloated? (Debloat device code)*

## 7 Thesis Contributions

This section outlines the contributions of this thesis, organized by the research questions each chapter addresses as shown in Table 2. First, Chapter A measures the bloat in ML containers, addressing RQ1. Building on the first finding from Chapter A, Chapter B proposes a debloating approach to remove unnecessary files in ML containers, addressing RQ2. Based on the second finding from Chapter A, Chapter C and Chapter D introduce methods to identify bloat in host code and remove unnecessary host code according to the running workload of ML shared libraries, addressing RQ3 and RQ4, respectively. Finally, Chapter E presents an approach that removes bloat from device code in ML shared libraries, addressing RQ5. Although the thesis focuses on ML systems, the approaches in Chapters A, B, C and D can be applied to other software systems as well.

**Table 2:** Research questions and the corresponding chapters that address them.

	Chapter A	Chapter B	Chapter C	Chapter D	Chapter E
RQ1	●	○	○	○	○
RQ2	○	●	○	○	○
RQ3	○	○	●	○	○
RQ4	○	○	○	●	○
RQ5	○	○	○	○	●

## 7.1 Chapter A: Measuring Bloat in ML Containers (Paper A)

The paper addresses RQ1, i.e., to what extent machine learning systems are bloated, and what are the causes of bloat in machine learning systems, with a particular focus on ML containers used in deployment. Bloat in this context refers to unnecessary code, dependencies, and resources packaged within ML containers that do not contribute to the core functionality of the ML system but still consume resources. This bloat leads to increased resource usage, reduced performance, and increased security risks due to a larger attack surface. We quantify the extent of bloat in ML containers, bridging the gap of existing research by being the first to conduct a comprehensive study of ML container bloat. Then we identify the main causes of bloat in ML containers, including redundant dependencies, large shared libraries, and the packaging of both training and serving functionalities in the same container.

We introduce MMLB, a framework for *Measuring and analyzing ML deployment Bloat* at the container level and the package level. We first select a range of ML containers, covering various frameworks, models, and tasks, ensuring the analysis is representative of typical ML use cases. Then we employ the system call tracing to identify files accessed during container operation, which allows us to determine the files that are necessary for the ML system to function. The files that are not accessed during operation are considered unnecessary, i.e., bloat. Then we calculate bloat degrees by comparing the sizes of original container (package) and its debloated version. Finally, we assess the security risks associated with bloat by scanning for vulnerabilities in excess code and examining dependency chains.

We measured 15 ML containers and found that bloat affects a large number of ML containers, with up to 80% of some containers consisting of bloat. Debloated ML containers tend to be larger than debloated generic containers, primarily due to the large sizes of ML shared libraries. APT packages are a significant source of bloat and vulnerabilities due to package dependencies. Additionally, ML packages are a main source of container bloat. Compared to the debloated version, bloat increases provisioning time by up to 370% and increases vulnerabilities by up to 99%.

## 7.2 Chapter B: Debloating ML Containers (Paper B)

This paper addresses RQ2, i.e., how to effectively debloat ML containers. Existing container debloating tools typically rely on tracing tools such as

**strace** to identify unused files from containers. However, such tracing tools lack awareness of the layered structure of the underlying container filesystems, limiting their ability to handle more complex bloat scenarios and often resulting in broken debloated containers.

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 two new types of layers in the filesystem to enable debloating, *debloating layer* and *reloading layer*. 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. To address the challenge of generalizing debloated containers across different workloads—a limitation faced by existing container debloating tools—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. We discuss how BLAFS can be used in different deployment scenarios and for different use-cases including container security hardening and a dynamic debloating 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 hardening mode, BLAFS removes up to 89% of CVEs while the 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%.

### 7.3 Chapter C: Identifying Bloat in Host Code (Paper C)

This paper addresses RQ3, i.e., identifying bloat in host code within shared libraries. Bloat in host code not only increases memory usage but also expands the attack surface, adding unnecessary vulnerabilities. To tackle this, the paper focuses on tracing shared library function calls made by the running workload, enabling the detection of unused code. However, existing library call tracers often suffer from low accuracy, high overhead, leading to false negatives that misclassify used code as unused—ultimately resulting in broken shared libraries when debloating is performed based on their output.

In this paper, we present RTrace, a novel tracer designed to accurately identify function calls in shared libraries. We identify the root causes behind the function call misses in widely used tracers, highlighting common pitfalls such as reliance on inaccurate or missing symbol information and failure to capture early-stage or indirect function calls. RTrace addresses these limitations through comprehensive runtime monitoring, accurate function boundary detection, and support for implicit and unconventional call patterns.

We compare RTrace against four state-of-the-art tracers, `ltrace`, `drltrace`, `ldaudit`, and `IntelPT`, across 21 applications and 92 shared libraries. Our results show that RTrace consistently outperforms these tools in accurately detecting function calls, achieving an F1 score of at least 0.92 on all benchmarks, while the best existing tracer achieves only 0.74. This high accuracy provides significantly improved visibility into the runtime behavior of shared libraries, enabling more accurate bloat identification in host code.

## 7.4 Chapter D: Debloating Host Code (Paper D)

This paper contributes to RQ4, i.e., removing bloat from host code in shared libraries. Tools for shared library debloating rely on either static or dynamic analysis techniques to discover which parts of the shared library is used. Once these parts are found, these tools take one of two routes, either removing parts of the library that are not used, requiring reconstruction of the library; or overwriting the unused functions. However, recent research has shown that debloating only yields negligible performance benefits or even degrades performance. In this paper, we challenge these finding, demonstrating that debloating shared libraries can yield substantial performance and security benefits for certain applications.

We propose MERGESHUFFLE, a novel tool for reconstructing shared libraries from a given set of file ranges to eliminate unused code based on user-specified requirements, without requiring access to source code. MERGESHUFFLE takes as input a shared library and a set of file ranges to retain, and performs a two-phase reconstruction: merging nearby ranges to reduce loading overhead, and shuffling them to minimize file size while preserving page-aligned memory mappings. The shuffling problem is formulated as a variant of the Traveling Salesman Problem and solved using a heuristic algorithm. MERGESHUFFLE utilizes program headers to remap retained code to its original memory addresses, preserving user-specified functionality.

We evaluate MERGESHUFFLE on 46 shared libraries from 9 realistic applications, including machine learning training and inference, command line tools, and micro-service based applications. Our result show that MERGESHUFFLE achieves up to 60% reduction in file size, 16% reduction in memory usage, 32% reduction in runtime, and 79% reduction in gadgets, while maintaining user-specified functionality, cross-application sharing, and improving security.

## 7.5 Chapter E: Debloating Device Code (Paper E)

This paper addresses RQ5, i.e., identifying and removing bloat in device code within ML shared libraries. Existing debloating research has mainly focused on host code bloat, neglecting device code, which forms a substantial part of ML shared libraries. Debloating device code is more challenging due to the lack of a public specification for device code, making it difficult to identify and remove unnecessary code. We propose an approach to debloat device code, addressing this gap in the literature.

We introduces `Negativa-ML`, a debloating tool specifically designed for ML shared libraries. `Negativa-ML` applies dynamic tracing to detect both host and device code utilized in ML workloads, allowing it to identify and remove unused

code effectively. *Negativa-ML* involves the following stages: (1) detection of used kernels via lightweight tracing, (2) locating the specific file ranges in shared libraries that contain used device code, and (3) compacting the libraries by removing unused elements to reduce overall size (proposed by Paper D). We utilize insights into the interaction between host and device code, as well as an understanding of the device code compilation process, to design an algorithm that safely removes unused device code while maintaining the functionality of the ML shared library.

We evaluate *Negativa-ML* on four widely used ML frameworks across ten ML workloads, analyzing over 300 shared libraries. The results show substantial reductions in bloat, with device code size reductions of up to 75% and host code reductions of up to 72%, leading to an overall file size reduction of up to 55%. Performance improvements include reduced memory usage for both CPU memory and GPU memory and faster execution times, particularly benefiting resource-limited or latency-sensitive applications.

## 8 Conclusions and Future Directions

In this thesis, we investigate the issue of bloat in ML systems, from file bloat in ML containers to code bloat in ML shared libraries. ML containers are widely used in modern cloud computing environments, while ML shared libraries are the standard mechanism for code sharing in modern operating systems. Our findings reveal that both ML containers and shared libraries contain a significant amount of unnecessary files and code, which bloat their size and lead to performance degradation, resource waste, and increased security vulnerabilities. To address these challenges, we propose novel debloating techniques that target debloating ML containers and ML shared libraries.

For ML containers, we propose *MMLB*, to measure the bloat in ML containers and identify the main causes of bloat. The results show that ML containers are highly bloated. In response to this finding, we introduce *BLAFS*, a novel, layer-based filesystem that automatically retains necessary files while removing unused ones. Our evaluations demonstrate that *BLAFS* effectively reduces container size by up to 94% with minimal overhead, making it feasible for deployment in live production environments. In doing so, *BLAFS* can debloat containers based on real-world usage patterns, which overcomes the limitations of existing offline debloating tools that rely on predefined workloads and may miss actual application usage.

In terms of ML shared libraries, We present *RTrace* and *MERGESHUFFLE* to identify and remove bloat in host code. We then introduce *Negativa-ML*, a novel debloating technique that debloats device code in ML shared libraries. *RTrace* achieves a more accurate and efficient detection of shared library function calls compared to widely used tracers. *MERGESHUFFLE* effectively removes bloat from ML shared libraries, leading to significant reductions in size, memory usage and run time while maintaining functionality and improving security. *Negativa-ML*, on the other hand, is the first tool to specifically address device code bloat in ML libraries.

Future research could explore integrating *BLAFS*, *RTrace*, *MERGESHUFFLE* and *Negativa-ML* into a unified debloating framework to comprehensively

address bloat in ML systems. This integrated framework would be particularly valuable for serverless ML services, where applications are packaged with specific features in containers that frequently include many shared libraries, contributing to bloat. Applying the framework to debloat these serverless services could reduce container size, library size, network bandwidth, energy consumption, memory overhead, and security risks. Additionally, existing research has focused solely on unused code, overlooking “used bloat”—code that is executed but does not contribute meaningfully to application performance or functionality. For example, training a model with a specific optimizer may involve initializing contexts with nonessential function calls. “Used bloat” is particularly harmful as it consumes memory, CPU, and GPU resources during execution and is more challenging to detect. Future work could focus on identifying and removing this “used bloat”. Finally, with the rise of large language models that show strong coding capabilities, future research could explore their potential in software debloating, potentially opening new avenues for automated and efficient debloating techniques.