

Mahdiyehsadat Mirsharifi

**DECREASING BUILD TIME IN OPEN BUILD
SERVICE THROUGH CONTAINERIZATION**
Implementation and performance analysis

Faculty of Information Technology and Communication Sciences
Master's thesis
December 2025

ABSTRACT

Mahdiyehsadat Mirsharifi

Decreasing Build Time in Open Build Service Through Containerization: Implementation and performance analysis

Master's thesis

Tampere University

Master's Programme in Information Technology

December 2025

The Open Build Service (OBS) enables reproducible software package creation but suffers from a performance bottleneck during build initialization. Each build requires extracting compressed archive files containing base systems and dependencies. When multiple builds share common dependencies, this repeated extraction of identical content creates significant time overhead, particularly in enterprise environments with frequent builds.

This thesis analyzes the OBS architecture and implements containerization to replace traditional archive extraction. The implementation modifies three core components: client enhancements for preinstallimage.info file downloads, build script containerization logic with hash-based image tracking, and worker integration supporting Podman alongside existing virtualization technologies.

Performance evaluation compares Docker and Podman container engines using preinstall images as container base images. Podman containerization delivers build time reductions ranging from 8 to 90 seconds depending on dependency complexity, translating to substantial time savings in enterprise environments executing hundreds of daily builds. Most significantly, containerization enables Podman to utilize preinstall images in rootless environments for the first time, transforming it from the slowest traditional approach to achieving optimal 5-second build times. Both engines demonstrate network efficiency improvements of approximately 45%. The containerization approach provides faster builds while maintaining backward compatibility with OBS infrastructure and security requirements.

Keywords: Open Build Service, Containerization, Docker, Podman, Build Performance, Monitoring, Network

The originality of this thesis has been checked using the Turnitin Originality service.

PREFACE

I would like to express my sincere gratitude to my mentor Islam Amer, who first introduced me to the Open Build Service and suggested this thesis topic. His continuous support and guidance throughout the entire process have been invaluable to the completion of this work.

I am deeply grateful to my university supervisor Dr. Markus Allen for his insightful advice and constructive feedback throughout my thesis work. His expertise and guidance helped shape this research into its final form.

Finally, I want to thank my family and my partner for their constant support and encouragement during the entire thesis process.

Helsinki, 9th December 2025

Mahdiyehsadat Mirsharifi

CONTENTS

1	INTRODUCTION	1
1.1	BACKGROUND AND CONTEXT	1
1.2	PROBLEM STATEMENT	2
1.3	RESEARCH OBJECTIVES	2
1.4	THESIS STRUCTURE	3
2	OPEN BUILD SERVICE (OBS)	4
2.1	INTRODUCTION TO OBS	4
2.2	SYSTEM ARCHITECTURE	5
2.2.1	BACKEND COMPONENTS	6
2.2.2	NETWORK ARCHITECTURE AND SECURITY	8
2.2.3	PROJECT ORGANIZATION	12
2.3	BUILD SYSTEM OPERATIONS	12
2.3.1	BUILD EXECUTION WORKFLOW	13
2.3.2	RPM PACKAGE FORMAT	14
2.4	BUILD TOOLS	15
2.4.1	OSC CLIENT	15
2.4.2	OBS-BUILD	15
2.5	CURRENT CHALLENGES AND BOTTLENECKS	16
3	TOOLS AND TECHNOLOGIES OVERVIEW	18
3.1	CONTAINER TECHNOLOGIES	18
3.1.1	CONTAINERS VS TRADITIONAL VIRTUALIZATION	18
3.1.2	EVOLUTION FROM EARLIER ISOLATION MECHANISMS	19
3.1.3	CONTAINER FOUNDATIONS	20
3.1.4	CONTAINER IMAGES AND STANDARDS	20
3.1.5	DOCKER	21
3.1.6	PODMAN	21

3.1.7	DOCKER VS PODMAN FOR OBS	22
3.2	MONITORING AND VISUALIZATION TOOLS	23
3.2.1	GITLAB	23
3.2.2	SUPERSET DASHBOARDS	24
4	CONTAINERIZATION IMPLEMENTATION IN OBS	25
4.1	OSC CLIENT MODIFICATIONS	25
4.2	BUILD SCRIPT MODIFICATIONS	27
4.2.1	CONTAINER IMPORT IMPLEMENTATION	27
4.2.2	CONTAINER STARTUP MODIFICATIONS	28
4.2.3	PACKAGE MANAGEMENT OPTIMIZATIONS	31
4.2.4	SUPPORTING INFRASTRUCTURE	32
4.3	OBS WORKER MODIFICATIONS	33
5	EVALUATION OF CONTAINERIZATION PERFORMANCE	34
5.1	TEST ENVIRONMENT SETUP	34
5.2	PREINSTALL IMAGE CONFIGURATIONS	35
5.3	TEST PACKAGE SPECIFICATIONS	36
5.4	EVALUATION METHODOLOGY	39
5.5	PERFORMANCE ANALYSIS	39
5.5.1	SANDBOX PACKAGE WITH BASE PREINSTALL IMAGE: EMPTY CACHE	40
5.5.2	SANDBOX PACKAGE WITH BASE PREINSTALL IMAGE: FULL CACHE	41
5.5.3	SANDBOX PACKAGE WITH OPTIMIZED PREINSTALL IMAGE	43
5.5.4	HEAVY PACKAGE WITH HEAVY PREINSTALL IMAGE	44
5.6	EVALUATION RESULTS	45
6	MONITORING FRAMEWORK FOR FUTURE PRODUCTION DEPLOYMENT	48
6.1	DATA COLLECTION	48
6.2	DATA ANALYSIS	49
6.3	VISUALIZATION	50
6.3.1	DATASET CREATION	50

6.3.2 DASHBOARD COMPONENTS	52
6.4 PERFORMANCE EXAMPLES	53
7 CONCLUSION AND FUTURE WORK	56
REFERENCES	58
APPENDIX A: GITLAB EVENTS DATA COLLECTION SCRIPT	60
APPENDIX B: DATA PROCESSING SCRIPT	65

LIST OF FIGURES

2.1	Conceptual overview of Open Build Service [6].	5
2.2	Simplified OBS component overview [7].	6
2.3	OBS network security architecture showing different trust zones and network isolation boundaries [11].	11
3.1	Comparison of traditional virtualization vs container architecture [21].	19
3.2	GitLab DevSecOps platform overview [30].	23
5.1	OBS build time comparison across preinstall image configurations (cached builds).	46
6.1	Available metrics and columns in Superset dataset.	52
6.2	Query A configuration (left) for individual build events and Query B configuration (right) for rolling average trend line. . .	53
6.3	Build performance for package A.	54
6.4	Build performance for package B.	54
6.5	Build performance for package C.	54
6.6	Build performance for package D.	55
6.7	Build performance for package E.	55
6.8	Build performance for package F.	55
6.9	Build performance for package G.	55

LIST OF TABLES

2.1	OBS component default network ports [9].	9
5.1	Sandbox package performance - base preinstall image (empty cache).	40
5.2	Sandbox package performance - base preinstall image (full cache).	41
5.3	Sandbox package performance - optimized preinstall image (full cache).	43
5.4	Heavy package performance - heavy preinstall image (full cache).	44

LIST OF CODES

4.1	Enhanced preinstall image download with info file support.	26
4.2	Podman container preinstall image detection.	27
4.3	Container preinstall image import function.	28
4.4	Docker container startup function with preinstall image support. . .	29
4.5	Podman container startup function with preinstall image support. .	30
4.6	Enhanced package filtering with info file support.	31
4.7	Optimized package metadata retrieval from preinstallimage.info. .	31
4.8	Skip preinstall packages for containers.	32
4.9	Container name randomization for concurrent builds.	32
4.10	Preinstall image build result handling for containers.	33
5.1	Test project configuration for containerization evaluation.	34
5.2	Base preinstall image configuration.	35
5.3	Sandbox preinstall image configuration.	35
5.4	Heavy preinstall image configuration with extensive dependencies. .	36
5.5	Sandbox package specification for performance evaluation.	37
5.6	Heavy package specification for performance evaluation.	38
5.7	Network monitoring wrapper script for performance evaluation. . .	39
6.1	GitLab events database schema.	49
6.2	Apache Superset dataset query for build performance analysis. . .	51
A.1	Complete GitLab Events Data Collection Script - Part 1	61
A.2	Complete GitLab Events Data Collection Script - Part 2	62
A.3	Complete GitLab Events Data Collection Script - Part 3	63
A.4	Complete GitLab Events Data Collection Script - Part 4	64
B.1	GitLab Events Data Processing Script - Part 1	66
B.2	GitLab Events Data Processing Script - Part 2	67

ABBREVIATIONS AND NOTATIONS

Apache Superset	A modern data exploration and visualization platform that provides business intelligence capabilities for analyzing and presenting data
API	Application Programming Interface
AppArmor	Application Armor, a Linux kernel security module that implements mandatory access control policies to restrict program capabilities
Build Recipe	Generic term for a recipe file for creating a package, including metadata, instructions, requirements, and changelogs. For RPM-based systems, a .spec file is used
Build Root	Directory where the osc command copies, patches, builds, and creates packages. By default, the build root is located in /var/tmp/build-root/BUILD_TARGET
BuildRequires	RPM spec file directive that specifies packages required for building the software, defining build-time dependencies that must be present in the build environment before compilation can proceed
CD	Continuous Deployment
chroot	A system operation that changes the apparent root directory for executing processes, creating an isolated environment known as a chroot jail
CI	Continuous Integration
Container Engine	Software that accepts user requests, pulls images, and manages the container lifecycle, such as Docker or Podman
Container Image	An immutable file containing a complete computing environment description, including libraries, executables, and configuration settings, used as a template for creating containers
Control Groups (cgroups)	Linux kernel feature that implements resource accounting and limiting capabilities for memory, disk space, and I/O operations
Dispatcher	OBS component that receives build jobs from schedulers and allocates them to qualified workers based on build constraints and load balancing
Docker	A container platform that uses a client-server architecture to build, run, and distribute containers through integrated tooling
FreeBSD Jails	Operating system-level virtualization mechanism that extends chroot capabilities by adding network virtualization, system user isolation, and process isolation

GitLab	A comprehensive DevSecOps platform that integrates source code management, CI/CD, project management, and deployment capabilities
I/O	Input/Output
IDE	Integrated Development Environment
KDE	International community that develops free and open-source software applications, with KDE-apps.org serving as a platform for application distribution
KIWI	System Image Creation Tool
KVM	Kernel-based Virtual Machine
libpod	Library that enables Podman to manage comprehensive container ecosystems including pods, containers, images, and volumes
LXC	Linux Containers
Namespace	Linux kernel feature that provides process isolation by creating separate instances of global system resources
OBS	Open Build Service
OCI	Open Container Initiative
OS	Operating System
osc	A command line tool to work with OBS instances. The acronym osc stands for openSUSE commander and works similarly to SVN or Git
Podman	A daemonless, open-source container management tool that provides Docker-compatible commands while supporting rootless container execution
Preinstall Image	An image containing a set of packages that can be installed in one quick step instead of individual package installations, designed to accelerate build processes
Publisher	OBS component that processes scheduler events for finished repositories by consolidating build results across all architectures into standardized directory structures
QEMU	Quick Emulator
REST	Representational State Transfer
Rootless Containers	Containers that can be created, run, and managed by non-privileged users without requiring root access, enhancing security by operating with the same privileges as the executing user
RPM	Red Hat Package Manager
RunC	OCI-compliant container runtime that serves as the reference implementation for creating and running containers according to OCI specifications

Scheduler	OBS component that continuously evaluates the need for new build jobs by monitoring changes in source code, project configurations, and binary dependencies
seccomp	Secure computing mode that provides system call filtering to restrict the kernel operations available to containerized processes
SELinux	Security-Enhanced Linux, a mandatory access control framework that enforces security policies governing filesystem access, process interactions, and network communications
SQL	Structured Query Language
Worker	Core execution units of the OBS system responsible for performing the actual package compilation process within isolated environments
WSL	Windows Subsystem for Linux

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

- Yes
 No

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
Amazon Q Developer (AWS)	1.18.1

PURPOSE OF THE USE OF AI

Amazon Q was used to provide feedback on thesis drafts and check grammar throughout the writing process. This included reviewing sentence structure, improving text clarity, and organizing information flow within sections. The tool was used solely to enhance readability and provide editorial feedback without changing or creating original research content, with all output subsequently revised and rewritten by the author.

PARTS OF THIS WORK, WHERE AI WAS USED

Amazon Q Developer was used across all chapters for grammar checking, proof-reading, and providing feedback on draft content to improve overall thesis quality and readability.

ACKNOWLEDGEMENT OF RISKS

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

1 INTRODUCTION

In modern software development, the ability to produce consistent and reproducible builds is fundamental to ensuring software reliability and security across diverse deployment environments. Reproducible builds establish development methodologies that enable independent verification of the relationship between source code and compiled binaries, ensuring software authenticity and detecting unauthorized modifications [1]. Industries such as telecommunications, health-care, finance, and aerospace require strict guarantees that rebuilding software from the same source code in similar environments will produce identical results, as software failures in these domains can have critical consequences.

The Open Build Service (OBS) addresses these requirements by providing automated build systems that ensure reproducible package creation across multiple platforms and architectures [2]. However, while OBS successfully delivers build consistency, a performance bottleneck in build processes remains a significant concern, particularly in enterprise environments where hundreds of builds are executed daily. This thesis investigates containerization as a solution to optimize OBS build performance while maintaining the reproducibility and isolation requirements essential for reliable software delivery.

1.1 BACKGROUND AND CONTEXT

The Open Build Service (OBS) is a comprehensive build and distribution system that enables the creation of software packages for multiple Linux distributions and architectures from a single source specification [2]. A key feature of OBS is its ability to construct clean, sandboxed build environments for every compilation task, ensuring build consistency and automatically handling the reconstruction of packages that depend on modified components. This multi-platform approach eliminates the complexity of maintaining separate build infrastructures for different operating systems and processor architectures.

The versatility and reliability of OBS have led to its widespread adoption across various industries to address complex software distribution challenges. Telecommunications companies like sysmocom use self-hosted OBS instances to build mobile communication software with customer-specific modifications [3]. Cybersecurity firms such as Datto leverage OBS to build software across over 25 Linux distribution targets, while network testing companies like Ostinato use OBS for multi-distribution package creation. These implementations demonstrate OBS's capability to support enterprise-scale software distribution requirements across diverse technical domains.

Given the scale of these enterprise deployments, build performance optimization becomes critical. To optimize build performance, OBS can optionally employ pre-install images that allow installing a set of packages in one quick step instead of individual package installations [4]. Depending on the build host, snapshots with

copy-on-write support may be used, which can avoid I/O operations entirely. This approach provides significant performance improvements compared to installing all dependencies from scratch for each build.

1.2 PROBLEM STATEMENT

Despite the performance improvements provided by preinstall images, a bottleneck remains in the build initialization phase. Each build job requires tar extraction of preinstall images, which involves extracting compressed tar archives containing the base system and pre-installed dependencies. While this tar extraction process is faster than downloading and installing packages individually, it still represents significant time overhead, with tar extraction and installation taking up to 2 minutes for some preinstall images in the company where this thesis was conducted.

In enterprise environments executing hundreds of builds daily, this repetitive tar extraction process represents a substantial performance optimization opportunity. The impact becomes more pronounced in high-frequency build scenarios where the same preinstall images are repeatedly extracted for multiple build jobs, with identical base system components being extracted independently for each build operation. Depending on the size and complexity of the packages included in the preinstall image, this tar extraction time can constitute a substantial portion of the total build time, especially for smaller packages where the build itself is relatively quick.

Containerization technologies offer an opportunity to address this performance bottleneck by eliminating the tar extraction process entirely. Instead of extracting preinstall images for each build, the same pre-installed system components could be utilized as container base images that start instantly without requiring extraction. This approach would maintain all the benefits of preinstall images while eliminating the time overhead of repeated tar archive processing, providing faster build initialization while preserving the isolation and reproducibility requirements essential for reliable software delivery.

1.3 RESEARCH OBJECTIVES

To address the performance bottleneck identified above, this thesis aims to implement and evaluate containerization in the Open Build Service to reduce build times beyond what preinstall images currently provide. The specific research objectives include:

1. **Implementation of containerization support:** Import preinstall images as container base images to eliminate the tar extraction process in OBS core components.
2. **Comparative evaluation of container technologies:** Determine the optimal containerization solution for OBS through performance and compatibility analysis of Docker and Podman implementations.

3. **Performance evaluation and validation:** Measure and demonstrate the build time improvements achieved through importing preinstall images as container base images compared to the current tar-based approach.

1.4 THESIS STRUCTURE

This thesis is organized into seven chapters that systematically address the containerization optimization of OBS build processes. Chapter 2 examines the Open Build Service architecture, focusing on the three-phase build process and identifying the specific preinstall phase bottleneck that containerization can address. Chapter 3 evaluates containerization technologies (Docker and Podman) as solutions for replacing tar-based preinstall image extraction, along with monitoring tools for performance validation. Chapter 4 describes the containerization implementation methodology across OBS core components. Chapter 5 provides a detailed performance evaluation comparing container technologies in the OBS context. Chapter 6 presents the performance monitoring framework developed for production environments and establishes baseline measurements for future validation. Chapter 7 concludes with research findings, contributions, and recommendations for future work in OBS performance optimization.

2 OPEN BUILD SERVICE (OBS)

This chapter demonstrates that while OBS provides a robust build infrastructure, its current chroot-based approach with preinstall image tar extraction creates a significant performance bottleneck that containerization can effectively address. Understanding OBS's architecture and current limitations reveals why containerization represents not merely an alternative approach, but a necessary evolution for improving build efficiency while maintaining security requirements.

2.1 INTRODUCTION TO OBS

The Open Build Service (OBS) is an automated system for building and distributing binary packages from source code in a consistent and reproducible manner [2]. The system supports package creation for multiple operating systems and hardware architectures, enabling software distribution across diverse platforms including various Linux distributions, different processor architectures, and complete software appliances [2].

OBS addresses the complexity of cross-platform software packaging by providing a centralized build infrastructure. Traditional software distribution requires developers to maintain separate build environments for each target platform, often necessitating "compiler farms" with different hardware architectures and operating system installations [5]. OBS eliminates this requirement by automating the entire build process in isolated environments that are created fresh for each build operation.

Beyond providing centralized infrastructure, the system incorporates automatic dependency resolution and management. When a package depends on another package, OBS automatically triggers rebuilds of all dependent packages when the underlying component changes [5]. This ensures package compatibility and maintains system integrity across the software ecosystem.

OBS operates through a project-based collaborative model that facilitates teamwork among developers and maintainers [5]. The system exposes an open API that supports integration with command-line tools like `osc` and external platforms including GitHub and KDE-apps.org. Additionally, OBS integrates with KIWI for automatic product and image creation through a web interface.

The platform demonstrates significant scale and adoption. The openSUSE project operates a public instance at `build.opensuse.org`, serving nearly 30,000 users who build over 140,000 packages across 21 base distributions and 6 architectures [2]. Beyond openSUSE, the system is utilized by various open source projects including Tizen and VideoLAN, companies such as SUSE, Dell, and Intel, as well as academic institutions operating their own OBS instances [2].

Figure 2.1 illustrates the complete OBS workflow from source code input to binary package distribution. The diagram shows how developers submit source code

to OBS, which then automatically builds packages across multiple target distributions and architectures. The key insight from this workflow is that OBS creates isolated build environments for each compilation task, ensuring reproducible builds while handling complex dependency relationships between packages.

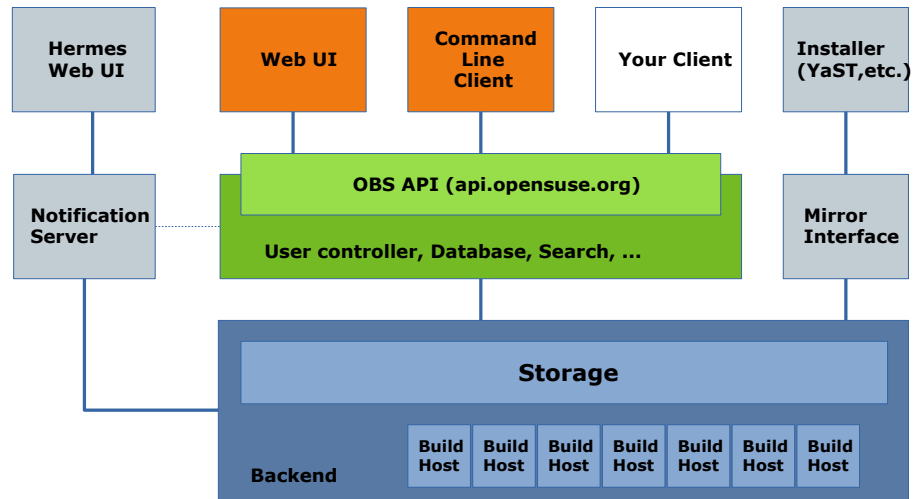


Figure 2.1: Conceptual overview of Open Build Service [6].

2.2 SYSTEM ARCHITECTURE

The OBS system employs a distributed architecture consisting of multiple interconnected services rather than a single monolithic application [7]. The system is fundamentally organized into two main components: a backend that implements all core functionality including package building, and a frontend that provides web application and XML API interfaces for backend interaction [8]. This design enables scalability and allows different components to handle specialized tasks within the build and distribution pipeline.

Since this thesis focuses on optimizing the build process, the following analysis concentrates primarily on the backend components where package compilation occurs. Figure 2.2 illustrates these distributed components and their interactions, showing how the Frontend handles user interfaces while the Backend manages build operations through several specialized components that will be examined in detail below.

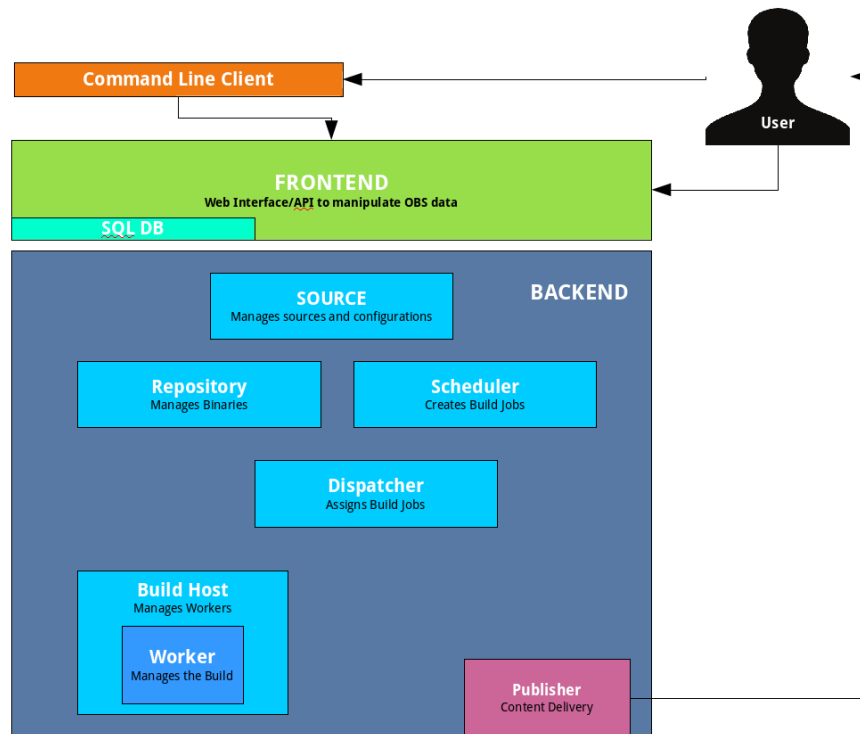


Figure 2.2: Simplified OBS component overview [7].

2.2.1 BACKEND COMPONENTS

The OBS backend infrastructure consists of multiple specialized daemons that work together to orchestrate the complete build and distribution process. These components form an interconnected system where source management, build scheduling, job distribution, and package publication operate in coordinated sequence. Understanding their interactions is essential for comprehending where containerization improvements can be integrated into the existing workflow. The following component descriptions are based on the OBS architecture guide [7].

SOURCE SERVER

The Source Server functions as the centralized repository management component, maintaining source code and project configurations while serving as the primary communication interface for frontend systems. This component implements storage optimization through MD5 hash verification with filename tracking, ensuring each unique source file exists only once in the repository while maintaining complete historical records of all source revisions. This historical tracking provides the foundation for reproducible builds by guaranteeing access to the exact source code used for any previously generated binary package. Each OBS installation operates with a single Source Server instance that manages all sources, project trees, and configuration data.

REPOSITORY SERVERS

Building upon the source management foundation, Repository Servers manage binary package distribution through HTTP-based interfaces, serving as intermediaries between frontend systems and build infrastructure. These servers coordinate with the Source Server to handle frontend access to binary packages, while workers interact directly with Repository Servers to register availability, obtain compilation dependencies, and submit completed build artifacts. Additionally, Repository Servers generate notification events that inform schedulers of binary package changes, triggering dependency analysis and potential rebuild requirements. Each OBS installation requires at least one Repository Server, with partitioned deployments maintaining dedicated instances per partition.

SCHEDULER

The coordination between source and repository management enables the Scheduler to determine build job requirements by monitoring source modifications, configuration changes, and binary dependency updates. The Scheduler orchestrates build execution sequences and integrates completed packages while maintaining build directory state, with each architecture and partition combination operating a dedicated Scheduler instance.

DISPATCHER

Following job creation by the Scheduler, the Dispatcher manages job assignment by receiving build jobs and allocating them to qualified workers based on build constraints. Fair resource distribution is achieved through load tracking per project repository, similar to Unix system load metrics, with jobs assigned to repositories having the lowest current load. Each partition operates its own dispatcher, with one designated as the master for coordination purposes.

PUBLISHER

Completing the build workflow, the Publisher handles repository completion by processing scheduler events for finished repositories, consolidating build results across all architectures into standardized directory structures. The Publisher generates required metadata and can synchronize content to external download servers while maintaining the established repository directory structure.

SUPPORTING SERVICES

Beyond the core workflow components, OBS relies on several supporting services that enhance system functionality and optimize performance. These include the Signer for cryptographic operations, the Source Service Server for continuous integration automation including version control system integration, and the Download on Demand Updater (dodup) for optimizing network efficiency in external repository management.

The dodup service is particularly relevant to performance optimization, as it addresses network bandwidth challenges through intelligent monitoring of external repositories. Rather than maintaining complete local copies, dodup continuously polls only the metadata of repositories defined as download on demand resources. When metadata changes are detected, the service selectively downloads only updated metadata and notifies the scheduler to recalculate affected build jobs. This approach significantly reduces both network bandwidth consumption and local storage requirements while ensuring builds have access to current dependency information. The efficiency gains are particularly pronounced in distributed OBS deployments where each partition can operate its own dodup service, minimizing redundant network transfers across geographic boundaries. Additional optimization services include the Delta Store, which provides source storage optimization through delta compression techniques.

WORKERS

The actual build execution is performed by Workers, which serve as the endpoint of the coordinated workflow established by the previous components. Workers represent the core execution units where containerization improvements have the most direct impact.

Workers initiate their participation by registering with repository servers and subsequently receive build job assignments from dispatchers. Upon job assignment, workers independently retrieve source code from the source server and download all required binary dependencies from the appropriate repository servers. The build process executes using build scripts within isolated environments, after which the resulting packages and build logs are transmitted back to the repository server.

While workers can operate on the same hardware as other OBS services, most production installations deploy workers on dedicated hardware to optimize build performance and resource utilization.

2.2.2 NETWORK ARCHITECTURE AND SECURITY

The backend components described above operate within a distributed network architecture that significantly influences both build performance and system security. Understanding this network foundation is essential for comprehending how containerization improvements can be integrated while maintaining OBS security requirements.

OBS operates as a distributed system where network communication patterns create both opportunities and constraints for performance optimization. The system's reliance on HTTP-based protocols creates multiple network dependencies that can become bottlenecks in high-throughput build environments [7]. These components communicate through specific TCP ports that define the network architecture, as detailed in Table 2.1.

Table 2.1: OBS component default network ports [9].

Component	Port	Purpose
Web Interface/API	443	Frontend HTTPS access
Repository Server	5252	Binary package distribution
Source Server	5352	Source code management
Source Service Server	5152	Continuous integration services
Cloud Upload Server	5452	Cloud deployment integration
Package Repository	82	Public package access

The communication flow follows a structured pattern that demonstrates the network-intensive nature of build operations [7]. User interactions with the frontend occur through HTTP/HTTPS protocols, with the frontend serving as both web interface and API gateway. The source server acts as the primary coordination point, managing communication between frontend systems and other backend components including repository servers for dependency management.

When package sources are updated, the system initiates a complex sequence of network exchanges [7]. The frontend receives source uploads via HTTP PUT operations, authenticates users, and forwards approved changes to the source server. The source server stores changes under revision control and coordinates with source service servers when automated services are required. Schedulers receive event notifications about package changes and coordinate with dispatchers to assign build jobs to available workers. Upon build completion, results are uploaded back through the scheduler to publishers, which coordinate with signers for package authentication before final distribution.

The most network-intensive operations occur during build execution phases [7]. Workers must establish connections to multiple repository servers to resolve dependency requirements, often downloading hundreds of megabytes of packages and preinstall images for each build task. This dependency resolution process creates significant bandwidth consumption, particularly when identical base system components are repeatedly transferred across the network for similar build jobs. The traditional approach of downloading complete preinstall images for each worker represents a substantial inefficiency in network resource utilization.

To mitigate network consumption, OBS addresses some efficiency challenges through Download on Demand (DoD) repositories, as described in Section 2.2.1, which optimize external dependency management by downloading packages only when required for specific builds [10]. This approach reduces both storage requirements and network bandwidth consumption compared to maintaining complete local copies of external repositories. DoD implementations leverage mirror server architectures to distribute network load away from primary repository servers, while SSL fingerprinting and checksum verification maintain security during network transfers. The system's ability to automatically detect upstream package updates through periodic metadata polling further optimizes network usage by avoiding unnecessary data transfers when repositories remain unchanged. Additionally, workers typically maintain local caches of frequently used dependen-

cies to avoid redundant transfers.

Build result distribution further compounds network demands as completed packages and associated metadata must be transmitted back through the system hierarchy. Repository updates trigger additional network activity as publication services coordinate with signing mechanisms and external distribution points [7]. In geographically distributed OBS deployments, these communication patterns are amplified by network latency, where round-trip delays between components can significantly extend build completion times.

For large-scale deployments, OBS partitions backend services across multiple networked machines to address scalability challenges [9]. This approach distributes computational and network loads across dedicated infrastructure components while employing network isolation strategies that separate backend communication from public interfaces. The partitioning model dedicates separate servers for different functional requirements, such as user home projects, release repositories, and general development projects.

Each partition operates its own set of backend services, creating multiple network endpoints that workers must coordinate with during build operations. While this partitioning strategy distributes network load, it also increases communication complexity, as workers may need to access multiple repository servers across different network locations to resolve build dependencies [9]. Network access control mechanisms enforce security through IP-based restrictions, defining which network addresses can access specific backend services.

The network architecture described above operates within strict security boundaries that define the operational constraints for any optimization approach. OBS implements a comprehensive security architecture based on network-isolated trust zones [11]. This layered security model separates system components across distinct network boundaries to minimize attack surfaces and contain potential security breaches.

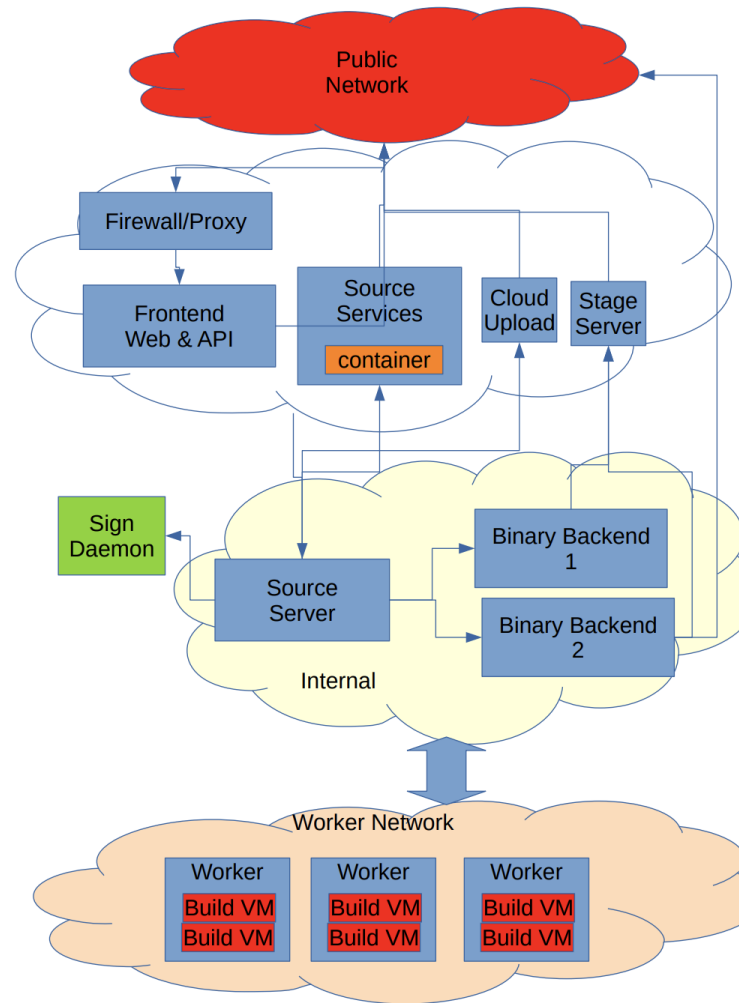


Figure 2.3: OBS network security architecture showing different trust zones and network isolation boundaries [11].

The architecture organizes components into four trust zones with progressively stricter network access controls, as illustrated in Figure 2.3. The public zone handles HTTPS connections to the API frontend while executing untrusted code under complete network isolation [11]. The demilitarized zone (DMZ) acts as a controlled bridge between public and internal networks, hosting the frontend service, background services for external integrations, and source service servers. The internal zone provides the most protected environment for core backend operations, where source servers coordinate package configurations and binary servers host build results. At the most restrictive level, the worker zone enforces the strictest isolation policies, allowing workers to access only source and binary servers through controlled internal connections.

The security model's emphasis on network isolation creates specific technical constraints for build system optimization. Build environments must execute untrusted code within virtualization boundaries that completely prohibit network access [11]. The system recognizes only KVM and XEN virtualization as fully secure isolation mechanisms, with build results extracted directly from block devices to

prevent filesystem-based security exploits.

These established security requirements define the operational framework within which any containerization improvements must function. The network isolation constraints that prevent build processes from establishing external connections represent fundamental design parameters that performance optimization strategies must respect. Understanding these security boundaries is essential for implementing containerization solutions that maintain OBS's security model while achieving performance improvements through faster build initialization and reduced preinstall image processing overhead.

2.2.3 PROJECT ORGANIZATION

Beyond the network architecture and security constraints that govern OBS infrastructure, the logical organization of software within OBS is managed through a structured project system. While the backend components provide the technical infrastructure for build operations, OBS organizes all hosted sources and binaries within projects, which define development tasks and specify authorized contributors [10]. Each project contains packages that represent specifications for building individual software components across all defined repositories.

This organizational structure follows a hierarchical framework where packages are grouped into logical collections based on functionality or development focus [6]. Each project defines access permissions, target platforms, and build configurations that apply to all contained packages. The hierarchy supports nested structures, allowing specialized subprojects to inherit from parent configurations while maintaining their own specific settings. The naming convention uses colon separators to indicate these hierarchical relationships, such as `devel:languages:python` for Python-related development packages.

The configuration of these projects is managed through metadata files that define generic descriptions, access control lists connecting users and groups with specific roles, and repository specifications [10]. The build process is controlled through various flags including build enablement, publishing control, and dependency usage settings that can be applied at either project or package levels.

To support individual development workflows within this structured environment, users operate within personal workspaces identified by their username [6]. These workspaces provide isolated areas for development and testing before contributing to shared projects, enabling both individual development workflows and collaborative team environments.

2.3 BUILD SYSTEM OPERATIONS

Having established the foundational architecture of OBS, this section examines the operational aspects of OBS build systems. The analysis begins with the build execution workflow that demonstrates where containerization improvements can be applied, followed by an exploration of the RPM package format that drives

these build operations in the company environment where this thesis was conducted.

2.3.1 BUILD EXECUTION WORKFLOW

The build execution process follows a coordinated sequence that demonstrates where containerization improvements can be applied [7]. When source changes trigger build requirements, schedulers perform dependency analysis and create build jobs when all requirements are satisfied. Dispatchers then assign these jobs to qualified workers based on build constraints and resource availability.

The critical phase for this thesis occurs when workers receive job assignments and begin build preparation. Workers independently download source code from the source server and retrieve all required binary dependencies from repository servers [7]. This dependency retrieval and environment setup phase, which traditionally involves tar extraction of preinstall images, represents the primary target for containerization optimization. Following environment preparation, workers execute the build process within isolated environments and transmit results back to the scheduler.

To ensure build consistency and reproducibility, package compilation occurs within completely isolated environments that are constructed dynamically for each build task [6]. These environments contain only the specific distribution components and dependencies required for the particular build, ensuring consistency and preventing interference from the host system. The OBS build process transforms source code into binary packages through a systematic approach that consists of three distinct phases [12].

The preinstall phase represents the critical initialization stage where OBS workers establish a minimal base system using packages designated as preinstalled components [12]. This phase installs essential system tools including file system utilities, core utilities, binary utilities, and package management tools such as rpm or debutils. The preinstall phase also transfers all required build dependencies and source code into the base system, involving the preinstall image processing that containerization aims to optimize.

Following the preinstall phase, the install phase begins where the worker may initialize virtual machines, emulators, or enter the build root environment depending on the selected build type [12]. Following successful environment activation, this phase reinstalls all base packages from the preinstall phase and additionally installs packages specified in the build recipe along with their dependencies. Upon completion, the environment contains all necessary components for build recipe execution.

The final package build phase executes distribution-specific build commands based on the package type, such as rpmbuild for RPM-based distributions or dpkgbuildpackage for Debian-based systems [12]. Source code compilation and packaging into the target format occurs during this phase, with additional quality checks performed through tools like rpmlint for RPM-based distributions.

These three build phases can be executed within different isolation environments

depending on security and performance requirements. OBS provides multiple build environment types to accommodate these different needs [12]. These include chroot-based environments for lightweight isolation, and virtualization technologies such as Xen, KVM, and QEMU for stronger isolation boundaries. The choice of build environment affects the setup process but maintains the same fundamental build phases across all types.

2.3.2 RPM PACKAGE FORMAT

Having examined the three-phase build process that OBS uses to transform source code into binary packages, it is essential to understand the RPM package format that drives these operations in the company environment where this thesis was conducted. The RPM Package Manager provides a comprehensive system for software distribution and management across Red Hat-based Linux distributions [13]. RPM packages consist of three primary components: a GPG signature for integrity verification, header metadata containing dependency information and installation instructions, and a payload comprising a cpio archive with the actual files to be installed.

OBS operations involve two distinct RPM package types: Source RPMs (SRPMs) containing source code and spec files, and Binary RPMs containing compiled executables and libraries [13]. The build process transforms SRPMs into Binary RPMs through the three-phase workflow, with RPM spec files controlling the compilation during the package build phase.

RPM spec files serve as instruction sets that the rpmbuild utility uses to build RPM packages [14]. These spec files consist of two main parts: the Preamble section containing metadata items, and the Body section representing the main build instructions. Most relevant to this thesis are the dependency-related directives, particularly BuildRequires, which specifies packages required for building the software.

The BuildRequires dependencies directly impact the preinstall phase optimization targeted by this thesis. When OBS processes a package build, it analyzes BuildRequires dependencies to determine which packages can be satisfied from preinstall images versus those requiring separate downloads.

During the build process, RPM utilizes a buildroot environment that functions as a chroot environment where build artifacts are organized according to the target system's file system hierarchy [14]. Build artifacts are placed within this buildroot using the same directory structure that will exist on the end user's system. The buildroot contents are subsequently packaged into a cpio archive that forms the core component of the resulting RPM package. The containerization improvements implemented in this thesis optimize the environment setup that occurs before this buildroot creation, maintaining compatibility with existing RPM spec file formats and buildroot functionality.

2.4 BUILD TOOLS

The three-phase build process and RPM package management described above are implemented through specific tools that manage the interaction between users, build environments, and the OBS infrastructure. This section examines the `osc` client for user interaction and `obs-build` for executing build processes to understand how containerization improvements can be integrated into existing workflows.

2.4.1 OSC CLIENT

The `osc` application functions as the primary command-line tool for Build Service interaction, developed in Python and offering both terminal interface and programmatic module capabilities [15]. It serves as the gateway to Build Service source repositories, providing functionality for metadata management and build status queries.

The client provides comprehensive functionality for project and package management, including source code checkout, status monitoring, change uploading, and build result analysis [16]. Key operations include project listing, package checkout, working copy updates, and change commitment processes that mirror traditional version control workflows.

For package compilation, `osc` employs `obs-build` to execute builds within isolated environments, defaulting to `chroot`-based isolation for enhanced build speed [15]. The tool accommodates various virtualization and containerization platforms through the `--vm-type` configuration option, supporting environments including `chroot`, `LXC`, `KVM`, `QEMU`, `Docker` with root privileges, and `Podman` with rootless operation.

Local build testing represents a critical functionality for development workflows, enabling developers to execute `osc build` commands to test package builds locally before server submission [16]. This local build capability was extensively utilized in this thesis for validating containerization improvements. The local build process replicates the same environment setup procedures used by OBS workers, including preinstall image handling, making it an ideal testing platform for containerization modifications.

2.4.2 OBS-BUILD

While `osc` provides the user interface for OBS interaction, the actual build execution is handled by `obs-build`, which serves as the core build execution engine that creates binary packages through safe and reproducible processes [17]. This tool operates both as a standalone application and as an integrated component within the Open Build Service infrastructure, providing the fundamental build capabilities that enable package compilation across the diverse build environment types described in Section 2.3.1.

`obs-build` supports multiple package formats to accommodate different Linux distributions, including RPM packages for SUSE, Fedora, and CentOS, DEB packages

for Debian and Ubuntu systems, and PKG packages for Arch Linux [17]. The system also supports various image formats such as Docker containers, KIWI appliances, and desktop application formats like AppImage, Flatpak, and Snapcraft.

Critical to this thesis is obs-build's support for preinstall images, which are designed to accelerate build processes particularly within OBS environments [17]. Preinstall images enable rapid installation of package sets through single operations rather than individual package installations [4]. Advanced build hosts may utilize snapshot functionality with copy-on-write capabilities to eliminate input/output operations entirely during this phase.

The system selects preinstall images based on package subset compatibility, choosing the largest applicable image when multiple options satisfy build requirements [4]. Implementation requires a package container within the project or associated repository containing a `_preinstallimage` file with spec file-like syntax specifying the image name and required packages through `BuildRequires` directives. Build job scheduling prioritizes preinstall image builds to maximize their effectiveness across the build infrastructure.

The current implementation processes these preinstall images through traditional tar-based extraction methods, creating the performance bottleneck that the containerization improvements developed in this thesis aim to address.

2.5 CURRENT CHALLENGES AND BOTTLENECKS

The OBS architecture, build operations, and tooling described throughout this chapter reveal a specific performance bottleneck that containerization can address. The bottleneck stems from the preinstall phase where all build environments require extracting compressed tar archives for each build operation through obs-build, regardless of whether chroot, KVM, or XEN virtualization is used.

In the production environment where this research was conducted, over 200 daily builds experience significant initialization overhead due to preinstall image tar extraction. This process can consume up to 2 minutes per build for larger preinstall images, representing a substantial portion of total build time. The inefficiency is compounded by the repetitive nature of the operation—identical base system components are extracted independently for each build job, even when the same preinstall images are reused across multiple builds.

This performance bottleneck is further exacerbated by the network architecture constraints described in Section 2.2.2. Workers must repeatedly download preinstall images and dependency packages, creating cumulative delays that scale with concurrent build operations [18]. While the repeated downloading represents an inherent network cost in distributed systems, the subsequent tar extraction process adds significant time overhead that further degrades overall system performance.

This bottleneck directly impacts the `BuildRequires` dependency resolution process outlined in Section 2.3.2 and affects both osc local builds and OBS build

execution. The containerization approach developed in this thesis primarily addresses this time overhead by utilizing preinstall image content as container base images. This eliminates repeated tar extraction operations while maintaining the isolation and reproducibility requirements essential to the OBS build system.

3 TOOLS AND TECHNOLOGIES OVERVIEW

This chapter examines the containerization technologies and monitoring tools selected to address the OBS performance bottleneck identified in Chapter 2. The analysis begins with container fundamentals, exploring how containerization can replace the traditional tar extraction process while maintaining the isolation requirements essential to OBS security architecture.

The chapter then compares Docker and Podman as container engine options, evaluating their compatibility with OBS's chroot-based build environments and security constraints. Finally, it introduces the monitoring and visualization tools—GitLab and Apache Superset—that enable rigorous performance evaluation of the containerization improvements, providing the measurement framework necessary to validate optimization effectiveness in production OBS environments.

3.1 CONTAINER TECHNOLOGIES

Containerization represents a paradigm shift in application deployment that directly addresses the preinstall image extraction bottleneck identified in OBS build environments. Containers provide lightweight virtualization solutions that enable application isolation without the overhead of traditional virtual machines or the time-consuming tar extraction processes currently used in OBS. The fundamental goal of containerization is to package applications along with all dependencies, enabling consistent execution across disparate development, test, and production environments [19].

Containerization addresses the critical problem of "dependency hell" in modern software deployment [19]. Applications assembled from existing components often rely on services with conflicting dependencies, missing dependencies in new environments, and platform differences across distributions. By packaging each component with its dependencies, containers solve these deployment challenges. This approach can potentially eliminate the repetitive preinstall image tar extraction that creates the performance bottleneck in OBS build environments where consistent, reproducible builds are essential.

3.1.1 CONTAINERS VS TRADITIONAL VIRTUALIZATION

To understand the advantages of containerization for OBS optimization, it is essential to examine how containers differ fundamentally from traditional virtualization technologies such as VMware, QEMU, Hyper-V, and Xen in their approach to isolation [20]. Traditional virtualization involves executing complete guest operating systems within isolated environments, where guest OS kernels communicate with hosts through virtualized hardware devices. In contrast, Linux containers share the host's underlying kernel and isolate processes from other host

processes by virtualizing the operating system itself rather than the hardware [19, 20]. Figure 3.1 illustrates this fundamental architectural difference.

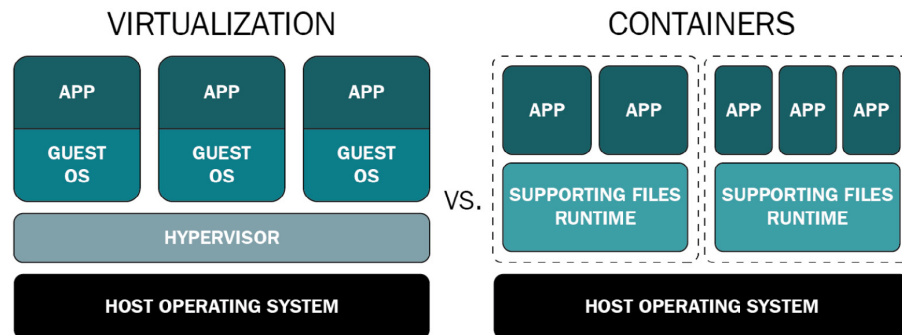


Figure 3.1: Comparison of traditional virtualization vs container architecture [21].

This architectural difference creates significant performance advantages. While traditional virtual machines load entire guest operating systems into memory, consuming gigabytes of storage and substantial resources before performing useful work [20], containers start as quickly as native applications since the host system only initiates isolated processes. This efficiency enables thousands of containers per host, with idle containers consuming minimal resources.

For build environments specifically, containers offer additional advantages beyond basic resource efficiency [19]. Container creation and destruction operations are inexpensive and fast, resembling application startup rather than operating system boot cycles. This characteristic proves particularly valuable for OBS builds, where environments must be rapidly provisioned and torn down for each build job.

3.1.2 EVOLUTION FROM EARLIER ISOLATION MECHANISMS

Modern container isolation represents a significant evolution from earlier Unix isolation mechanisms [21]. The chroot system call, introduced in Unix V7 (1979), provided basic filesystem isolation by changing the root directory for processes and their children. However, chroot was not designed with security in mind and could be easily escaped, with operating system documentation strongly discouraging its use as a security mechanism. FreeBSD jails (2000) extended chroot capabilities by adding network virtualization, system user isolation, and process isolation, but remained limited compared to modern container technologies.

Linux containers address the fundamental limitations of chroot-based isolation through comprehensive security mechanisms that extend far beyond filesystem restrictions. While chroot provides only filesystem-level isolation with shared system resources, containers implement multi-layered security through namespaces, cgroups, and additional kernel security features [21]. This evolution is particularly relevant for OBS optimization, as it enables both performance improvements through faster startup times and enhanced security isolation compared to the traditional chroot-based build environments currently employed in the company.

3.1.3 CONTAINER FOUNDATIONS

To understand how containerization can replace OBS's current tar extraction process, it is essential to examine the underlying kernel technologies that enable container isolation and resource management. Modern container implementations harness two primary kernel-level technologies for process isolation and resource management [19]. Linux Containers (LXC) utilize kernel namespaces to isolate containers from host systems through user namespaces that separate container and host user databases, process namespaces that manage only container processes, and network namespaces that provide dedicated network devices. Complementing this isolation, Control Groups (cgroups) implement resource accounting and limiting capabilities, enabling precise consumption limits for memory, disk space, and I/O operations.

These foundational isolation mechanisms are enhanced by additional kernel-level security features [22]. System call filtering through secure computing mode (sec-comp) restricts kernel operations available to containerized processes, while kernel capabilities provide granular control over privileged system operations. Access control frameworks such as SELinux and AppArmor further strengthen security by enforcing mandatory policies governing filesystem access, process interactions, and network communications.

This comprehensive approach to isolation and security directly addresses OBS requirements for secure, isolated build environments. The combination of namespace isolation and resource controls can replace the current chroot-based approach while eliminating tar extraction overhead. These kernel-level mechanisms provide the security boundaries necessary for executing untrusted build code while offering superior performance characteristics compared to traditional preinstall image tar extraction.

3.1.4 CONTAINER IMAGES AND STANDARDS

Understanding the distinction between container images and running containers is critical for OBS optimization [20]. Container images are files stored on disk containing application code, dependencies, and metadata [23]. When executed, container engines unpack these files and transfer them to the Linux kernel through API calls that create isolated processes with enhanced security mechanisms.

The Open Container Initiative (OCI) provides industry standardization for container formats and runtime specifications [23]. The OCI Container Image Format Specification defines on-disk formats and metadata including hardware architecture and operating system specifications, while the Container Runtime Specification standardizes how container engines communicate with host kernels. This standardization ensures interoperability between different container platforms and tools across diverse environments.

Base images form the foundational layer of container architectures, containing complete operating system installations without parent layers [23]. These images include essential package management tools such as rpm, yum, or apt-get, enabling subsequent package installations and system modifications. For OBS

optimization, preinstall images can be converted to container base images containing the operating system foundation and build tools necessary for package compilation, eliminating the repetitive tar extraction operations that create the current performance bottleneck.

3.1.5 DOCKER

Docker represents an open platform designed for application development, deployment, and execution [24]. The platform enables separation of applications from underlying infrastructure, facilitating rapid software delivery through consistent deployment methodologies. Docker facilitates complete container lifecycle management through integrated tooling, encompassing development, distribution, testing, and deployment phases across diverse environments.

The platform employs a client-server model where the Docker client communicates with the Docker daemon through REST API interfaces over UNIX sockets or network connections [24]. The daemon (dockerd) manages all Docker objects including images, containers, networks, and volumes while processing these API requests. The client serves as the primary user interface, translating commands such as `docker run` into daemon instructions. For actual container creation, Docker relies on OCI-compliant container runtimes, primarily RunC, to interface with the operating system and create running containers [23]. Docker registries provide centralized storage for Docker images, with Docker Hub serving as the default public registry [24].

In terms of image management, Docker utilizes a layered architecture where each Dockerfile instruction creates a distinct layer, enabling efficient rebuilds by updating only modified layers [24]. This layered approach optimizes storage and network transfer by sharing common layers between images. For container isolation, Docker leverages Linux kernel features, particularly namespaces, to provide container isolation where each container operates within separate namespaces that limit access to designated resources.

Docker's networking subsystem employs a pluggable architecture using multiple network drivers [25]. Most relevant for OBS integration is the none network driver, which completely isolates containers from both the host and other containers, ensuring no network connectivity during build execution as required by OBS security policies.

Combined with these technical capabilities, Docker's mature ecosystem includes extensive tooling, comprehensive documentation, and widespread industry adoption, making it the de facto standard for containerization.

3.1.6 PODMAN

Podman represents a daemonless, open-source container management tool designed for Linux environments, focusing on finding, executing, building, sharing, and deploying applications through Open Containers Initiative (OCI) compliant containers and images [26]. The tool provides a command-line interface that

maintains compatibility with Docker Container Engine commands, enabling users to substitute Docker commands with Podman equivalents without modification.

In contrast to Docker's client-server architecture, Podman operates without a central daemon process [26]. This daemonless design eliminates the need for a continuously running background service, reducing system resource consumption and potential security attack vectors. Instead, Podman interfaces directly with OCI-compliant container runtimes such as runc, crun, and runv to communicate with the operating system and instantiate containers. To support this direct interface approach, the libpod library enables Podman to manage comprehensive container ecosystems including pods, containers, images, and volumes.

From a security perspective, Podman's most distinctive feature is its rootless container capability, which allows container execution without requiring root access [21, 26]. Rootless containers operate with the same privileges as the executing user, eliminating elevated permissions while maintaining full functionality [21].

Podman supports the same core network modes as Docker, including the none network mode essential for OBS security requirements [27]. Additionally, Podman provides RESTful API capabilities for programmatic container management, with remote client support across Linux, macOS, and Windows platforms [26].

3.1.7 DOCKER VS PODMAN FOR OBS

Both Docker and Podman address the preinstall image bottleneck identified in OBS builds by eliminating repetitive tar extraction operations. Converting preinstall images to container images allows containers to start directly without requiring extraction for each build, preserving build isolation while significantly improving startup performance.

Docker's mature ecosystem and widespread adoption provide extensive tooling and community support, making it a reliable choice for production environments. Its client-server architecture offers centralized container management, though this requires a continuously running daemon process.

Podman's daemonless architecture and rootless execution capabilities align particularly well with security-conscious build environments. The elimination of daemon processes reduces system overhead and potential security vulnerabilities, while rootless capabilities address multi-user build environment security concerns without requiring elevated privileges.

Both platforms support the network isolation requirements essential for OBS security architecture through the none network mode, enabling complete network isolation during build execution. Both offer OCI compliance ensuring compatibility with container images converted from OBS preinstall images, and their Docker-compatible interfaces facilitate seamless integration with existing OBS infrastructure through the `--vm-type` option.

For this thesis implementation, both Docker and Podman are evaluated through practical testing to determine the optimal solution, as detailed in Chapter 5. While Podman's security advantages and reduced system overhead appear particularly beneficial for multi-user build environments, where rootless execution provides

additional security benefits without compromising functionality, the final selection is based on empirical performance results and integration compatibility with existing OBS infrastructure.

3.2 MONITORING AND VISUALIZATION TOOLS

Implementing containerization improvements requires rigorous performance measurement to validate optimization effectiveness. This section examines the monitoring and visualization tools used to collect and analyze build performance data, providing the analytical framework necessary to demonstrate containerization benefits quantitatively. The complete monitoring framework implementation is described in Chapter 6.

3.2.1 GITLAB

GitLab is a comprehensive DevSecOps platform that integrates source code management, continuous integration/continuous deployment (CI/CD), project management, and deployment capabilities within a unified environment [28]. This all-in-one approach eliminates the complexity of managing separate tools for different stages of the software development lifecycle, making it particularly suitable for containerized application development and deployment workflows [29].

The platform’s integrated design offers key advantages for development teams [28]. Complete DevOps lifecycles can be managed within a single system, while built-in security features include automated vulnerability scanning with each commit and detailed audit capabilities that track all actions from planning to deployment. All DevSecOps tools provided by GitLab can be found in the comprehensive platform overview, as illustrated in Figure 3.2.

Planning	Source Code Management	Continuous Integration	Security	Compliance	Artifact Registry	Continuous Delivery	Observability
DevOps Reports	Remote Development	Secrets Management	Container Scanning	Release Evidence	Virtual Registry	Release Orchestration	On-call Schedule Management
DORA Metrics	Source Code Management	Review Apps	Software Composition Analysis	Compliance Management	Container Registry	Infrastructure as Code	Incident Management
Value Stream Management	Web IDE	Code Testing and Coverage	API Security	Audit Events	Helm Chart Registry	Pages	Error Tracking
Value Stream Forecasting	GitLab CLI	Merge Trains	Coverage-guided Fuzz Testing	Software Bill of Materials	Package Registry	Feature Flags	Product Analytics Visualization
Service Desk	Code Review Workflow	Merge Request Summary	Dependency Management	Dependency Management	Model Registry (Beta)	Environment Management	AI Product Analytics
Wiki	Code Suggestions	Root Cause Analysis	DAST	Vulnerability Management	Dependency Proxy	Deployment Management	AI Impact Dashboard
Portfolio Management	Code Explanation	Discussion Summary	Code Quality	Security Policy Management		Auto DevOps	Metrics
Team Planning	Code Review Summary	Merge Commit Message Generation	Secret Detection	SAST			Distributed Tracing
Generate issue description	Test Generation	Pipeline Composition and Component Catalog	Vulnerability Explanation	Vulnerability Resolution			Logs
Discussion Summary	Code Refactorization		GitLab Advisory Database				
Design Management	GitLab Duo for the CLI						
Replacement for Jira	Replacement for GitHub	Replacement for Jenkins	Replacement for Snyk		Replacement for JFrog	Replacement for Harness	Replacement for Sentry

Figure 3.2: GitLab DevSecOps platform overview [30].

For this thesis, GitLab serves as the version control tool with CI/CD pipelines provided by an internally developed tool that integrates with GitLab through the commit status API for visibility purposes. GitLab's comprehensive REST and GraphQL APIs enable programmatic access to platform functionality [29], allowing developers to create automation scripts for routine maintenance tasks. The REST API, accessible through `/api/v4` endpoints, provides structured access to GitLab resources including project data and pipeline information essential for retrieving build state measurements and performance analysis in this thesis.

3.2.2 SUPERSET DASHBOARDS

Apache Superset is a modern data exploration and visualization platform that can replace or augment proprietary business intelligence tools for many teams [31]. The platform offers an interactive user experience that requires no programming knowledge, making it accessible for cross-functional teams including data analysts, business professionals, and software engineers [32].

Superset's strength lies in its broad compatibility and visualization capabilities. The platform supports nearly any SQL database or data engine through Python SQLAlchemy connectors, including PostgreSQL, MySQL, SQLite, and Snowflake [32]. It provides a wide array of visualizations ranging from simple bar charts to complex time-series analyses, while its lightweight, configurable caching layer helps ease database load during intensive performance monitoring [31].

Additional key features include a no-code interface for rapid chart creation and SQL Lab, a built-in SQL IDE for data exploration [32]. Its lightweight semantic layer enables definition of custom dimensions and metrics, facilitating standardized performance measurements [31].

For this thesis, Superset serves as the primary visualization platform, leveraging its native PostgreSQL support to analyze build performance metrics collected from GitLab pipelines and present containerization performance improvements.

4 CONTAINERIZATION IMPLEMENTATION IN OBS

The previous chapters identified preinstall image extraction as a key performance bottleneck and established containerization as the solution approach. This chapter details the practical implementation of container support in OBS to eliminate tar extraction overhead.

The implementation required coordinated modifications across three core OBS components due to the interconnected nature of the build pipeline. The OSC client was enhanced to download `preinstallimage.info` files and enable Podman to use preinstall images as container base images. The build scripts contain the core containerization logic for container import and execution. The OBS worker was extended to support Podman as a virtualization type and to provide `preinstallimage.info` file downloads. The complete implementation is available in the respective project repositories: OSC client modifications [33], build script changes [34], and OBS worker integration [35].

4.1 OSC CLIENT MODIFICATIONS

The OSC client required modifications to support containerized preinstall images through enhanced download functionality and Podman compatibility fixes [33]. While OSC operates independently of the OBS server infrastructure, it serves as the primary testing platform for containerization changes since `osc build` replicates the same build process used by OBS workers.

The core modification extended the preinstall image download functionality in `osc/build.py`. The `get_preinstall_image()` function was enhanced to download both `preinstallimage` and `preinstallimage.info` files, with the info file providing essential metadata for container-based builds. To ensure reliable concurrent operations, the implementation uses atomic file operations through `NamedTemporaryFile`, as shown in Code 4.1.

```

1 def get_preinstall_image(apiurl, arch, cache_dir, img_info,
  ↵ offline=False):
2     imagefile = ""
3     imagesource = ""
4     info_file = "preinstallimage.info"
5     # ... existing code ...
6
7     imageinfo = info_file_path
8
9     # Download preinstall image with unique temporary file
10    with NamedTemporaryFile(dir=cache_path, delete=False) as
  ↵ temp_file:
11        try:
12            gr.urlgrab(url, filename=temp_file.name,
  ↵ text="fetching image")
13            os.rename(temp_file.name, ifile_path)
14        except HTTPError as e:
15            print("Failed to download! ecode:%i reason:%s" %
16                  (e.code, e.reason))
17            if os.path.exists(temp_file.name):
18                os.unlink(temp_file.name)
19            return ("", "", "", [])
20
21    # Also download the corresponding .info file
22    if not os.path.exists(info_file_path):
23        info_url = "%s/build/%s/%s/%s/%s/%s" % (
24            apiurl, img_project, img_repository,
25            img_arch, img_pkg, info_file)
26        with NamedTemporaryFile(dir=cache_path, delete=False) as
  ↵ temp_file:
27            try:
28                gr.urlgrab(info_url, filename=temp_file.name,
29                           text="fetching image info")
30                os.rename(temp_file.name, info_file_path)
31            except HTTPError as e:
32                print("Failed to download info file! ecode:%i
  ↵ reason:%s" %
33                      (e.code, e.reason))
34
35    return (imagefile, imagesource, imageinfo, img_bins)

```

Code 4.1: Enhanced preinstall image download with info file support.

Beyond the download functionality, additional modifications were needed to enable preinstall image usage with Podman. Testing revealed that Podman builds were not receiving preinstall images due to existing restrictions on unprivileged virtualization types. The system disables preinstall images for "kvm", "podman", and "qemu" environments, assuming extraction would fail in unprivileged contexts. However, containerization eliminates the need for extraction, allowing successful preinstall image usage. The client was modified to detect container-based preinstall usage and enable preinstall images for Podman in this specific scenario,

as shown in Code 4.2.

```
# Check if using container preinstallimage for podman
using_container_preinstall = (
    vm_type == "podman"
    and opts.build_opt
    and any("--vm-use-container-preinstallimage" in opt
           for opt in opts.build_opt)
)

if build_as_user(vm_type) and not using_container_preinstall:
    # preinstallimage extraction will fail for unprivileged user
    bi.preinstallimage = None
```

Code 4.2: Podman container preinstall image detection.

4.2 BUILD SCRIPT MODIFICATIONS

The build script modifications represent the core containerization implementation, enabling preinstall images to function as container base images for both Docker and Podman environments [34]. These changes eliminate the traditional tar extraction process by importing preinstall images directly as container images while maintaining existing OBS security constraints.

4.2.1 CONTAINER IMPORT IMPLEMENTATION

The implementation began with adding a new command-line option `--vm-use-container-preinstallimage` to enable using preinstall images as container base images instead of tar files. This option is only available for Docker and Podman virtualization types, providing the foundation for all containerization functionality.

The core functionality is provided by the `preinstall_container_image()` function in the `init_buildsystem` script. This function manages the container import process by first checking if the preinstallimage has already been imported as a container image. If not found, it imports the image to avoid redundant operations in subsequent builds.

The function uses MD5 hash-based tagging for consistent image identification across builds, as shown in Code 4.3. This approach leverages existing OBS build process data to ensure reliable container image management. When a preinstall image is updated, its MD5 hash changes, causing the system to detect that the previously imported container image is outdated and automatically trigger a re-import. If the MD5 hash matches an existing imported image, the system reuses the cached container image, avoiding redundant import operations.

```

1 preinstall_container_image() {
2     check_exit
3     IMAGE_NAME="${2##*/}"
4     IMAGE_NAME="${IMAGE_NAME%% \[*]}"
5     IMAGE_TAG=$(echo "$2" | awk -F'[][]' '{print $2}')
6     echo "Using preinstall image${2:+ $2} as container image"
7     preinstall_setup
8
9     # Check if the image exists
10    if $CONTAINER images --format "{{.Repository}}:{{.Tag}}" | \
11        grep -q "${IMAGE_NAME}:${IMAGE_TAG}"; then
12        echo "Preinstall container image
13        ↪ ${IMAGE_NAME}:${IMAGE_TAG} exists."
14    else
15        echo "Importing preinstall container image
16        ↪ ${IMAGE_NAME}:${IMAGE_TAG}"
17        if ! $CONTAINER import "$BUILD_INIT_CACHE/rpms/$1" \
18            "${IMAGE_NAME}:${IMAGE_TAG}"; then
19            echo "ERROR: failed to import preinstall container
20            ↪ image."
21            cleanup_and_exit 1
22        fi
23    fi
24    export VM_CONTAINER_IMAGE="${IMAGE_NAME}:${IMAGE_TAG}"
25    echo "VM_CONTAINER_IMAGE='$VM_CONTAINER_IMAGE'" >> \
26        $BUILD_ROOT/.build/build.data
27 }

```

Code 4.3: Container preinstall image import function.

4.2.2 CONTAINER STARTUP MODIFICATIONS

Beyond the import functionality, the container startup process required fundamental changes to preserve preinstall image benefits. Traditional container builds used `chroot` operations within containers, mounting the build root at `/mnt` and executing `chroot /mnt`. However, this approach would lose access to packages provided by the preinstall image since `chroot` isolates the build environment from the container's filesystem.

The solution involved eliminating `chroot` operations when using preinstall images as container bases, instead running build scripts directly within the container environment. This allows the build process to access both preinstall image packages and build files mounted from the host. Both Docker and Podman required specific modifications to implement this approach while maintaining their respective operational characteristics.

For Docker, the `build-vm-docker` script was modified to change from `busybox` with `chroot` to preinstall image with direct execution, as shown in Code 4.4.

```

1  vm_startup_docker() {
2      source $BUILD_ROOT/.build/build.data
3      local name=$CONTAINER_NAME
4      docker rm "$name" >/dev/null 2>&1 || true
5      local docker_opts=
6      local container_root="/mnt"
7      local build_command=(chroot /mnt "$vm_init_script")
8      local mounts=( "--mount"
9          ↪ "type=bind,source=$BUILD_ROOT,destination=/mnt")
10     test -n "$VM_TYPE_PRIVILEGED" && docker_opts="--privileged
11     ↪ --cap-add=SYS_ADMIN --cap-add=MKNOD"
12     test -n "$RUN_SHELL" -o -n "$RUN_SHELL_AFTER_FAIL" &&
13     ↪ docker_opts="$docker_opts -it"
14     local docker_image="busybox"
15     if test -n "$VM_CONTAINER_IMAGE" ; then
16         mkdir -p $BUILD_ROOT/home/abuild
17         rm -rf $BUILD_ROOT/.build.packages
18         ln -s home/abuild/rpmbuild $BUILD_ROOT/.build.packages
19         docker_image="$VM_CONTAINER_IMAGE"
20         container_root=
21         mounts+=( "--mount" "type=bind,source=$BUILD_ROOT/home/ab_
22             ↪ uild,destination=/home/abuild")
23         local build_command=(bash -c "
24             for item in /mnt/.[^.]*; do
25                 [ -d "\$item" ] && ln -sf "\$item"
26                 ↪ "\/$$(basename "\$item")\"
27             done
28             $vm_init_script
29         ")
30     fi
31     mounts+=(
32         "--mount" "type=bind,source=/proc,destination=$container_
33             ↪ _root/proc"
34         "--mount" "type=bind,source=/dev/pts,destination=$contai_
35             ↪ ner_root/dev/pts"
36         "--mount" "type=bind,source=/dev/null,destination=$conta_
37             ↪ iner_root/dev/null"
38     )
39     docker run --rm --name "$name" --net=none $docker_opts \
40         "${mounts[@]}" "$@" "$docker_image"
41         ↪ "${build_command[@]}"
42     BUILDSTATUS="$?"
43     test "$BUILDSTATUS" != 255 || BUILDSTATUS=3
44     cleanup_and_exit "$BUILDSTATUS"
45 }

```

Code 4.4: Docker container startup function with preinstall image support.

Podman required similar modifications but faced a different technical challenge. Unlike Docker's chroot approach, Podman typically mounts the build root directly to the container root (/). When using preinstall images as container bases, this default behavior would overwrite the container's filesystem and lose access

to preinstalled packages. The Podman solution therefore mounts the build root at /mnt and creates symlinks to access build scripts while preserving the container's package environment, as shown in Code 4.5.

```

1 vm_startup_podman() {
2     source $BUILD_ROOT/.build/build.data
3     local name=$CONTAINER_NAME
4     podman rm "$name" >/dev/null 2>&1 || true
5     local podman_opts=
6     local container_root="/"
7     local build_command=("$vm_init_script")
8     local mounts=()
9     test -n "$VM_TYPE_PRIVILEGED" && podman_opts="--privileged
   ↪ --cap-add=SYS_ADMIN --cap-add=MKNOD"
10    test -n "$RUN_SHELL" -o -n "$RUN_SHELL_AFTER_FAIL" &&
   ↪ podman_opts="$podman_opts -it"
11    local podman_image="build-scratch:latest"
12    if test -n "$VM_CONTAINER_IMAGE" ; then
13        mkdir -p $BUILD_ROOT/home/abuild
14        rm -rf $BUILD_ROOT/.build/packages
15        ln -s home/abuild/rpmbuild $BUILD_ROOT/.build/packages
16        podman_image="$VM_CONTAINER_IMAGE"
17        container_root="/mnt"
18        mounts+=("--mount" "type=bind,source=$BUILD_ROOT/home/abj
   ↪ uild,destination=/home/abuild")
19        local build_command=(bash -c "
20            for item in /mnt/.[^.]*; do
21                [ -d "\$item" ] && ln -sf "\$item"
   ↪ "\/\$(basename "\$item")\"
22            done
23            $vm_init_script
24        ")
25    fi
26    mounts+=("--mount" "type=bind,source=$BUILD_ROOT,destination,
   ↪ =$container_root")
27    podman run --runtime=runc --rm --name "$name" --net=none
   ↪ $podman_opts \
28        "${mounts[@]}" "$@" $podman_image "${build_command[@]}"
29    BUILDSTATUS="$?"
30    test "$BUILDSTATUS" != 255 || BUILDSTATUS=3
31    cleanup_and_exit "$BUILDSTATUS"
32 }

```

Code 4.5: Podman container startup function with preinstall image support.

Both implementations preserve the `--net=none` parameter to maintain OBS security requirements, ensuring complete network isolation during build execution. The containerization design accommodates this constraint by ensuring all required dependencies are either present in the preinstall image or downloaded during the preinstall phase.

4.2.3 PACKAGE MANAGEMENT OPTIMIZATIONS

With container startup mechanisms in place, the build system required optimizations to work efficiently with preinstall images as container bases. Three key optimizations were implemented: enhanced package filtering, optimized metadata retrieval, and elimination of redundant package installation.

Package filtering was enhanced to utilize `preinstallimage.info` files when available. Previously, the system checked extracted preinstall image directories to determine available packages. With containerization, preinstall images are not extracted, but the info file provides the same package information without requiring extraction, as shown in Code 4.6.

```

1 preinstall_image_filter() {
2     # If we have preinstall image info file, use it for package
3     ↪ filtering
4     if test -n "$PREINSTALL_IMAGE_INFO" -a -f
5     ↪ "$PREINSTALL_IMAGE_INFO" ; then
6         for PKG in "$@" ; do
7             if grep -q "$PKG$" "$PREINSTALL_IMAGE_INFO"
8             ↪ 2>/dev/null ; then
9                 continue
10            fi
11            echo "$PKG"
12        done
13    else
14        echo "$@"
15    fi
16 }

```

Code 4.6: Enhanced package filtering with info file support.

Building on this info file approach, package metadata retrieval was also optimized to prioritize `preinstallimage.info` files for `PKG_HDRMD5` and `PKGID` information, avoiding the need to access extracted directories, as shown in Code 4.7.

```

1 # Try to read PKG_HDRMD5 and PKGID from preinstallimage info
2 ↪ file first
3 if test -n "$PREINSTALL_IMAGE_INFO" -a -f
4 ↪ "$PREINSTALL_IMAGE_INFO" ; then
5     PKG_INFO=$(grep "$PKG$" "$PREINSTALL_IMAGE_INFO" 2>/dev/null
6     ↪ | head -n1)
7     if test -n "$PKG_INFO" ; then
8         PKG_HDRMD5=$(echo "$PKG_INFO" | awk '{print $1}')
9         PKGID=$(echo "$PKG_INFO" | awk '{print $2}')
10    fi
11 else
12     # Fallback to reading from unpacked preinstall image
13     read PKG_HDRMD5 PKGID < $BUILD_ROOT/.preinstall_image/$PKG
14 fi

```

Code 4.7: Optimized package metadata retrieval from `preinstallimage.info`.

The final optimization leverages the fact that packages are already present in container base images by skipping traditional preinstall package installation when using container preinstall images. This avoids redundant installations, as shown in Code 4.8.

```

1 # no need to preinstall packages if we are importing them in
  ◁ preinstall container image
2 if test -z "$VM_CONTAINER_IMAGE"; then
3     progress_setup PACKAGES_TO_PREINSTALL_FILTERED
4     for PKG in $PACKAGES_TO_PREINSTALL_FILTERED ; do
5         progress_step PACKAGES_TO_PREINSTALL_FILTERED
6         preinstall ${PKG##*/}
7     done
8     if test -n "$PREPARE_VM" ; then
9         echo
10        progress_setup PACKAGES_TO_VMINSTALL_FILTERED
11        for PKG in $PACKAGES_TO_VMINSTALL_FILTERED ; do
12            progress_step PACKAGES_TO_VMINSTALL_FILTERED
13            preinstall ${PKG##*/}
14        done
15    fi
16 fi

```

Code 4.8: Skip preinstall packages for containers.

4.2.4 SUPPORTING INFRASTRUCTURE

Beyond the core containerization functionality, additional modifications were implemented to ensure reliable container operation in production environments. Container name randomization prevents conflicts during concurrent builds, as shown in Code 4.9.

```

test -n "$CONTAINER_PREINSTALLIMAGE" && echo
  ◁ "CONTAINER_PREINSTALLIMAGE='$CONTAINER_PREINSTALLIMAGE'" >>
  ◁ $BUILD_ROOT/.build/build.data
test "$VM_TYPE" = "docker" && echo
  ◁ "CONTAINER_NAME='obsbuild.${BUILD_ROOT##*/}.$(date
  ◁ +%s).${RANDOM}'" >> $BUILD_ROOT/.build/build.data
test "$VM_TYPE" = "podman" && echo
  ◁ "CONTAINER_NAME='build_${RECIPEFILE//:/-}.$(date
  ◁ +%s).${RANDOM}'" >> $BUILD_ROOT/.build/build.data

```

Code 4.9: Container name randomization for concurrent builds.

Finally, the `vm_wrapup_build()` function was modified to handle preinstall image build results for container environments, ensuring successful completion of preinstall image builds when using Docker or Podman, as shown in Code 4.10.

```

if test "$BUILDTYPE" = "preinstallimage"; then
    case "$VM_TYPE" in
        docker|podman) preinstallimage_compress
            ↪ "$BUILD_ROOT/$TOPDIR/OTHER" ;;
    esac
fi

```

Code 4.10: Preinstall image build result handling for containers.

4.3 OBS WORKER MODIFICATIONS

The OBS worker required coordinated modifications to integrate with the containerization changes implemented in the build scripts [35]. These modifications enable container-based preinstall image usage while maintaining compatibility with existing OBS infrastructure.

The primary modification implemented a new command-line option `--vm-use-container-preinstallimage` to enable using preinstall images as container base images. When the system configuration `OBS_WORKER_USE_CONTAINER_PREINSTALLIMAGE` is enabled, this option flows from the OBS worker through `bs_worker` to the build script for Docker and Podman virtualization types.

To support the build script metadata optimizations, the worker's `getpreinstallimage()` function was enhanced to detect and return `preinstallimage.info` files alongside traditional preinstall images. This enables the worker to pass metadata files to the build system, supporting the optimized package information retrieval described in the previous section. The `getbinaries()` function stores the info file path in the `preinstallimagedata` structure, and `dobuild()` includes it in the build's `rpm` list when available.

Finally, Podman was integrated as a supported virtualization type alongside existing LXC and Docker support. This required updating the worker's cleanup logic, job management, and build argument handling to recognize Podman containers. The `cleanup_job()`, `kill_job()`, and `dobuild()` functions were modified to handle Podman containers identically to Docker containers for build root management and argument passing.

5 EVALUATION OF CONTAINERIZATION PERFORMANCE

With the containerization implementation complete across all OBS components, performance evaluation was conducted to validate the optimization benefits and compare build approaches. The evaluation utilized the public openSUSE OBS instance to test real-world scenarios with varying preinstall image complexities and build requirements.

5.1 TEST ENVIRONMENT SETUP

The evaluation was conducted using Podman version 4.9.5 and Docker version 28.4.0-ce (build 249d679a6) on Windows Subsystem for Linux (WSL) running on a laptop with Intel Core Ultra 7 165H processor and 32 GB RAM. Container engine versions can affect build performance, as updates and modifications in each version may introduce optimizations or changes that impact execution efficiency. Additionally, host hardware specifications influence performance values. The performance results presented in this evaluation are specific to this configuration.

The test project employed a hierarchical repository configuration that mirrors real-world OBS usage patterns. The project metadata defined two repositories with specific build and publish configurations, as shown in Code 5.1.

```

1 <project name="home:mahdiyeh_mrs">
2   <title/>
3   <description/>
4   <person userid="mahdiyeh_mrs" role="maintainer"/>
5   <build>
6     <disable repository="base_image"/>
7     <disable repository="imx7d"/>
8   </build>
9   <publish>
10    <enable/>
11  </publish>
12  <repository name="imx7d">
13    <path project="home:mahdiyeh_mrs" repository="base_image"/>
14    <arch>x86_64</arch>
15  </repository>
16  <repository name="base_image">
17    <path project="openSUSE:Leap:15.6" repository="standard"/>
18    <arch>x86_64</arch>
19  </repository>
20 </project>

```

Code 5.1: Test project configuration for containerization evaluation.

This configuration creates a realistic testing environment with hierarchical package inheritance. The `base_image` repository inherits packages from the openSUSE Leap 15.6 standard repository, enabling it to build preinstall images using those base packages. The `imx7d` repository inherits from both `base_image` and its parent openSUSE Leap 15.6, allowing it to utilize the preinstall images built in `base_image` as dependencies while having access to the full package ecosystem. This setup enables measurement of containerization optimizations in scenarios that reflect actual OBS usage patterns, where repositories build upon each other in hierarchical relationships. Build operations were initially disabled to allow manual control over the testing sequence.

5.2 PREINSTALL IMAGE CONFIGURATIONS

Three distinct preinstall image configurations were developed to test different complexity levels and their impact on containerization performance. All preinstall images were built in the `base_image` repository, making them available for packages in the `imx7d` repository. OBS automatically selects the most suitable preinstall image for each package build by matching as many build dependencies as possible while minimizing unnecessary packages.

The base preinstall image, shown in Code 5.2, provides minimal dependencies suitable for any package build, as it contains `bash` which is a fundamental requirement for OBS build operations.

```
Name: base
BuildRequires: bash
#!BuildIgnore: brp-trim-desktopfiles
```

Code 5.2: Base preinstall image configuration.

An optimized preinstall image was created to match the specific build dependencies of the test package, as shown in Code 5.3. This preinstall image was initially disabled to allow testing with the base preinstall image, then enabled later to compare performance when OBS selects the optimized image over the base image for the `sandbox` package.

```
Name: sandbox
BuildRequires: bash
BuildRequires: python311-devel python311-pip
↳ python311-setuptools
BuildRequires: python311-wheel python311-qt5 python311-PyYAML
BuildRequires: python311-numpy python311-matplotlib
BuildRequires: zbar
#!BuildIgnore: brp-trim-desktopfiles
```

Code 5.3: Sandbox preinstall image configuration.

A comprehensive heavy preinstall image was configured to simulate complex

build environments with extensive dependency requirements, as shown in Code 5.4.

```

1 Name: heavy-base
2 BuildRequires: bash
3 BuildRequires: gcc-c++ cmake make pkg-config
4 BuildRequires: libboost_headers-devel libboost_system-devel
5 BuildRequires: libqt5-qtbase-devel libqt5-qttools-devel
6 BuildRequires: opencv-devel ffmpeg-4-libavcodec-devel
7 BuildRequires: gstreamer-devel gstreamer-plugins-base-devel
8 BuildRequires: gtk3-devel libxml2-devel libxslt-devel
9 BuildRequires: libopenssl-devel libcurl-devel sqlite3-devel
10 BuildRequires: postgresql-devel libmysqlclient-devel
11 BuildRequires: python3-devel python3-numpy-devel python3-scipy
12 BuildRequires: nodejs-common rust cargo go
13 BuildRequires: java-11-openjdk-devel maven gradle
14 BuildRequires: texlive-latex-bin ImageMagick-devel
    ◀ libreoffice-sdk
15 #!BuildIgnore: brp-trim-desktopfiles

```

Code 5.4: Heavy preinstall image configuration with extensive dependencies.

5.3 TEST PACKAGE SPECIFICATIONS

The evaluation employed two representative test packages to demonstrate different dependency complexity levels and their interaction with the preinstall image configurations described in Section 5.2. The primary test package is a Python-based application that demonstrates typical modern build requirements, as shown in Code 5.5. The optimized preinstall image shown in Code 5.3 was specifically created to match this package's dependencies.

```

1 Name:          sandbox
2 Version:      0
3 Release:      1
4 Summary:      This is a test package.
5 License:      GPLv2+
6
7 BuildRequires: python311-devel python311-pip
   ↪ python311-setuptools
8 BuildRequires: python311-wheel python311-qt5 python311-PyYAML
9 BuildRequires: python311-numpy python311-matplotlib
10 BuildRequires: zbar
11
12 %description
13 This is a test package for containerization performance
   ↪ evaluation.
14
15 %build
16 echo "Nothing to build"
17
18 %install
19 mkdir -p {buildroot}/usr/bin
20 echo "#!/bin/bash" > {buildroot}/usr/bin/sandbox
21 echo "echo 'Sandbox test package'" >>
   ↪ {buildroot}/usr/bin/sandbox
22 chmod +x {buildroot}/usr/bin/sandbox
23
24 %files
25 /usr/bin/sandbox

```

Code 5.5: Sandbox package specification for performance evaluation.

A second test package was developed to evaluate performance with extensive development dependencies, as shown in Code 5.6. The comprehensive heavy preinstall image shown in Code 5.4 was designed to match this package's extensive dependencies and test containerization performance in complex enterprise build environments.

```

1  Name:             heavy-build-test
2  Version:        1.0
3  Release:        1
4  Summary:        Test package with heavy BuildRequires
5  Source0:        %{name}-%{version}.tar.gz
6
7  BuildRequires: gcc-c++ cmake make pkg-config
8  BuildRequires: libboost_headers-devel libboost_system-devel
9  BuildRequires: libqt5-qtbase-devel libqt5-qttools-devel
10 BuildRequires: opencv-devel ffmpeg-4-libavcodec-devel
11 BuildRequires: gstreamer-devel gstreamer-plugins-base-devel
12 BuildRequires: gtk3-devel libxml2-devel libxslt-devel
13 BuildRequires: libopenssl-devel libcurl-devel sqlite3-devel
14 BuildRequires: postgresql-devel libmysqlclient-devel
15 BuildRequires: python3-devel python3-numpy-devel python3-scipy
16 BuildRequires: nodejs-common rust cargo go
17 BuildRequires: java-11-openjdk-devel maven gradle
18 BuildRequires: texlive-latex-bin ImageMagick-devel
   ↵ libreoffice-sdk
19
20 %description
21 A test package designed to have many heavy BuildRequires
22
23 %prep
24 %setup -q
25
26 %build
27 echo "Testing heavy dependencies build..."
28 gcc --version
29 cmake --version
30 python3 --version
31 java -version
32 go version
33 rustc --version
34 node --version
35 # Create a simple test program
36 cat > test.cpp << 'EOF'
37 #include <iostream>
38 #include <boost/version.hpp>
39 int main() {
40     std::cout << "Boost version: " << BOOST_VERSION <<
   ↵     std::endl;
41     return 0;
42 }
43 EOF
44 g++ -o test test.cpp -lboost_system
45
46 %install
47 mkdir -p %{buildroot}%{_bindir}
48 install -m 755 test %{buildroot}%{_bindir}/heavy-build-test
49
50 %files
51 %{_bindir}/heavy-build-test

```

Code 5.6: Heavy package specification for performance evaluation.

5.4 EVALUATION METHODOLOGY

The evaluation methodology employed `osc build` commands executed through a custom `network-time.sh` wrapper script to collect comprehensive performance and network metrics. This script, shown in Code 5.7, combines GNU `time` with verbose output (`time -v`) and network usage monitoring to capture both system performance and network traffic data.

```

1 #!/bin/bash
2 # Network monitoring wrapper with GNU time
3
4 RX_START=$(cat /sys/class/net/*/statistics/rx_bytes | awk
5   ↵ '{sum+=$1} END {print sum}')
6 TX_START=$(cat /sys/class/net/*/statistics/tx_bytes | awk
7   ↵ '{sum+=$1} END {print sum}')
8
9 /usr/bin/time -v "$@"
10
11 RX_END=$(cat /sys/class/net/*/statistics/rx_bytes | awk
12   ↵ '{sum+=$1} END {print sum}')
13 TX_END=$(cat /sys/class/net/*/statistics/tx_bytes | awk
14   ↵ '{sum+=$1} END {print sum}')
15
16 echo "=== Network Usage ==="
17 echo "RX: $((RX_END - RX_START)) bytes"
18 echo "TX: $((TX_END - TX_START)) bytes"

```

Code 5.7: Network monitoring wrapper script for performance evaluation.

The script captures network interface statistics before and after command execution, providing measurements of data transfer during build operations. The evaluation was conducted across three test configurations with different preinstall image complexities: sandbox with base preinstall image (tested with both empty and full package cache), sandbox with optimized preinstall image, and heavy package with heavy preinstall image.

Each configuration tested three build approaches: `chroot` builds (default), `Docker` builds (`--vm-type=docker`), and `Podman` builds (`--vm-type=podman`). For `Docker` and `Podman`, both traditional approaches and containerized builds using `preinstallimage` as container base (`--vm-use-container-preinstallimage`) were evaluated, including first-time import and subsequent cached runs.

5.5 PERFORMANCE ANALYSIS

The performance analysis examines containerization benefits through systematic comparison of the build approaches and test configurations described in Sections 5.2, 5.3, and 5.4. Using the network monitoring wrapper script, metrics were

collected across all test scenarios to demonstrate the impact of containerization on OBS build performance.

The performance tables present six key metrics: Total Time (complete `osc build` execution including setup, dependency resolution, build execution, and cleanup), OBS Build Time (actual package compilation and installation duration within the build environment), CPU (percentage utilization), Memory (peak consumption in megabytes), and RX/TX (network traffic in kilobytes for received and transmitted data).

The first evaluation uses an empty package cache to establish baseline performance, while all subsequent evaluations use a full package cache to focus on containerization improvements without network overhead.

5.5.1 SANDBOX PACKAGE WITH BASE PREINSTALL IMAGE: EMPTY CACHE

The initial evaluation compared five build approaches using the sandbox package (Code 5.5) with the base preinstall image (Code 5.2) when the package cache (`/var/tmp/osbuild-packagecache`) was empty. This scenario represents worst-case performance conditions where all dependencies must be downloaded from network repositories and the preinstall image is lean, containing only essential components like `bash`. Note that scenarios with pre-imported container images were not tested in this empty cache condition, as an empty package cache typically indicates either a fresh system where preinstall images would not yet be imported, or updated preinstall images that would require re-import regardless.

Table 5.1: Sandbox package performance - base preinstall image (empty cache).

Build Type	Total Time (sec)	OBS Build (sec)	CPU (%)	Memory (MB)	RX (KB)	TX (KB)
Chroot	118.70	15	5	71.3	322205	7743
Docker (traditional)	121.26	15	5	71.2	321945	7505
Docker (preinstall base, 1st)	140.14	27	4	71.2	322709	7569
Podman (traditional)	160.61	26	7	69.1	315169	7467
Podman (preinstall base, 1st)	129.50	16	6	71.6	322718	8043

Table 5.1 reveals important distinctions between total execution time and actual OBS build performance. Both `chroot` (118.70 s total, 15 s OBS build) and traditional `Docker` (121.26 s total, 15 s OBS build) achieve identical OBS build times, with `Docker` showing only 2.5 seconds additional system overhead. Traditional `Podman` exhibits significantly degraded performance (160.61 s total, 26 s OBS build) due to its inability to use preinstall images in rootless environments.

The containerization approach shows varying results depending on the engine used. `Docker` using `preinstallimage` as container base requires 140.14 s total time with 27 s OBS build time, where the 12-second increase (27 s vs 15 s) includes 8-second import overhead. Most significantly, `Podman` using `preinstallimage` as

container base achieves 129.50 s total time with only 16 s OBS build time, representing a significant 10-second improvement over traditional Podman by successfully utilizing preinstall images. This improvement was expected since traditional Podman cannot utilize preinstall images in rootless environments due to tar extraction limitations, as discussed in Section 4.1. Containerization eliminates this constraint by removing the need for tar extraction entirely, enabling Podman to access preinstall image benefits for the first time.

Network usage patterns show consistent substantial downloads (315–322 MB) across all approaches when building from empty cache, confirming that dependency resolution dominates total execution time. The similar network requirements highlight that OBS build time differences represent the true containerization performance impact, independent of network overhead.

With baseline performance established under worst-case conditions, the next evaluation examines performance when dependencies are readily available.

5.5.2 SANDBOX PACKAGE WITH BASE PREINSTALL IMAGE: FULL CACHE

The second evaluation examined the same package and preinstall image configuration as the previous test, but with the package cache populated, representing typical development scenarios where dependencies have been previously downloaded. This comparison includes traditional builds and builds using preinstall-image as container base with both first-time image import and subsequent runs using cached images.

Table 5.2: Sandbox package performance - base preinstall image (full cache).

Build Type	Total Time (sec)	OBS Build (sec)	CPU (%)	Memory (MB)	RX (KB)	TX (KB)
Chroot (traditional, cached)	22.74	16	2	41.1	507	252
Docker (traditional, cached)	19.75	15	2	41.4	542	282
Podman (traditional, cached)	29.28	24	16	51.2	288	28
Docker (preinstall base, 1st)	31.86	27	1	41.5	289	30
Docker (preinstall base, 2nd)	21.49	17	2	41.4	285	27
Docker (preinstall base, 3rd)	23.78	18	2	41.4	275	17
Docker (preinstall base, 4th)	23.11	18	2	41.4	275	16
Podman (preinstall base, 1st)	25.99	20	30	105.9	288	30
Podman (preinstall base, 2nd)	21.65	16	8	48.3	286	29
Podman (preinstall base, 3rd)	20.74	15	8	47.5	276	15
Podman (preinstall base, 4th)	20.63	15	9	48.0	276	17

Table 5.2 demonstrates the performance characteristics when package dependencies are readily available. Traditional Docker establishes the baseline with 15 s OBS build time. Traditional Podman shows degraded performance with 24 s

OBS build time due to its inability to utilize preinstall images in rootless environments.

Containerized approaches reveal a two-phase performance pattern. First-time runs incorporate the preinstall image import process, with Docker achieving 27 s OBS build time and Podman requiring 20 s OBS build time. Podman imports the preinstall image in approximately 4 seconds compared to Docker's 8-second import time, contributing to better first-run performance. Second-run builds eliminate the import overhead, with Docker achieving 17 s OBS build time and Podman achieving 16 s OBS build time.

The results reveal important insights about containerization benefits. Most significantly, Podman containerization enables preinstall image usage with substantial performance improvements. Traditional Podman cannot utilize preinstall images in rootless environments, resulting in 24 s OBS build time. Containerization eliminates this limitation, enabling Podman to achieve 16 s OBS build time (an 8-second improvement) that unlocks preinstall image capabilities previously unavailable in rootless configurations.

Docker containerization introduces a 2-second overhead in OBS build time (17 s vs 15 s traditional) when using preinstall images in steady state. Containerized Podman achieves 16 s OBS build time, outperforming containerized Docker (17s) in the preinstall image environment. While containerized Podman is 1 second slower than traditional Docker, this comparison highlights the significant benefit of preinstall image containerization for Podman specifically—transforming it from the slowest traditional approach (24s) to the second-fastest overall solution (16s).

As visible in Table 5.2, the 3rd and 4th run results show minor variations but remain stable, demonstrating that performance stabilizes after the initial import. These minor deviations likely stem from connection speed fluctuations with the openSUSE public OBS instance or system resource variations during measurement. Local OBS deployments would provide more predictable performance with reduced variation. To minimize such effects, the evaluation ensured only one build executed at a time during measurements.

CPU and memory usage reveal different patterns between Docker and Podman. Traditional and containerized Docker maintain consistent low CPU usage (2%) and memory consumption (41.4 MB) across all runs. Podman shows higher resource consumption during first-run image import (30% CPU, 105.9 MB memory), but returns to efficient levels in subsequent runs (8% CPU, 48.3 MB). This contrasts with Docker's consistent resource usage across runs. Despite the elevated first-run consumption, Podman's faster import times make this temporary resource spike acceptable.

Network efficiency improvements are evident across containerized builds. Docker containerization reduces network traffic by 47% (285 KB vs 542 KB traditional Docker) and 44% compared to chroot (285 KB vs 507 KB). Podman containerization achieves similar efficiency with 47% reduction compared to traditional Docker (286 KB vs 542 KB) and 44% compared to chroot (286 KB vs 507 KB).

The previous evaluations used the base preinstall image containing only essential components. To demonstrate the full potential of containerization, the next eval-

uation uses an optimized preinstall image specifically designed for the sandbox package.

5.5.3 SANDBOX PACKAGE WITH OPTIMIZED PREINSTALL IMAGE

The third evaluation uses the same sandbox package as previous tests, but with an optimized preinstall image (Code 5.3) specifically designed to match the package's Python-based dependencies. This optimized preinstall image includes all of the sandbox package build dependencies and was automatically selected instead of base preinstall image.

Table 5.3: Sandbox package performance - optimized preinstall image (full cache).

Build Type	Total Time (sec)	OBS Build (sec)	CPU (%)	Memory (MB)	RX (KB)	TX (KB)
Chroot (traditional, cached)	11.94	8	3	40.6	294	27
Docker (traditional, cached)	11.18	7	3	40.4	538	271
Podman (traditional, cached)	29.81	24	17	51.6	298	31
Docker (preinstall base, 1st)	25.06	20	1	40.4	309	32
Docker (preinstall base, 2nd)	9.11	5	4	40.5	291	22
Podman (preinstall base, 1st)	17.27	13	82	125.4	296	26
Podman (preinstall base, 2nd)	8.79	5	15	48.4	291	22

Table 5.3 demonstrates exceptional performance improvements when preinstall images are optimized for specific build requirements. Traditional Docker achieves 7 s OBS build time, while chroot requires 8 s OBS build time. Traditional Podman continues to show degraded performance (24 s OBS build) due to its inability to use preinstall images in rootless environments.

Containerized approaches with optimized preinstall images reveal significant performance improvements. First-time runs incorporate the preinstall image import process, with Docker achieving 20 s OBS build time and Podman requiring 13 s OBS build time. Notably, Podman imports the optimized preinstall image in approximately 8 seconds compared to Docker's 15-second import time. Second-run builds eliminate the import overhead, with both Docker and Podman achieving identical exceptional 5 s OBS build times.

Performance data demonstrates the transformative impact of optimized preinstall images on containerization benefits. Podman containerization enables preinstall image usage with remarkable performance improvements, achieving 5 s OBS build time—a 19-second improvement over traditional Podman's 24 s. The optimized preinstall image enables containerization to surpass traditional approaches for both platforms.

CPU and memory usage exhibit similar behavior to the base preinstall evaluation. Traditional Docker (3% CPU, 40.4 MB) and containerized Docker (4% CPU, 40.5 MB) show minimal differences across runs. Podman exhibits elevated resource

consumption during first-run import (82% CPU, 125.4 MB) due to the larger optimized image, but returns to reasonable levels in subsequent runs (15% CPU, 48.4 MB).

Network efficiency remains excellent across containerized builds. Docker containerization reduces network traffic by 46% (291 KB vs 538 KB traditional Docker) and maintains similar efficiency to chroot (291 KB vs 294 KB). Podman containerization achieves identical 46% reduction compared to traditional Docker (291 KB vs 538 KB) and similar efficiency to chroot (291 KB vs 294 KB).

Comparing Tables 5.2 and 5.3 reveals the substantial impact of preinstall image optimization. Docker container second runs improve from 17 s to 5 s OBS build time—a 12-second improvement. Podman container second runs improve from 16 s to 5 s OBS build time—an 11-second improvement. Traditional approaches also benefit significantly from optimized preinstall images. Docker improves from 15 s to 7 s OBS build time (8-second improvement), while chroot improves from 16 s to 8 s (8-second improvement). The optimized preinstall image delivers consistent performance gains across all build approaches, with containerized solutions achieving the best absolute performance.

To evaluate containerization performance in complex enterprise scenarios, the final evaluation tests a heavy package with extensive development dependencies using a heavy preinstall image.

5.5.4 HEAVY PACKAGE WITH HEAVY PREINSTALL IMAGE

The final evaluation tests the heavy package defined in Code 5.6 using the preinstall image defined in Code 5.4. This configuration represents complex enterprise build environments with substantial build requirements.

Table 5.4: Heavy package performance - heavy preinstall image (full cache).

Build Type	Total Time (sec)	OBS Build (sec)	CPU (%)	Memory (MB)	RX (KB)	TX (KB)
Chroot (traditional, cached)	19.33	14	2	41.6	429	34
Docker (traditional, cached)	17.06	13	2	41.6	840	441
Podman (traditional, cached)	101.80	96	6	86.4	521	93
Docker (preinstall base, 1st)	76.92	72	0	41.6	463	57
Docker (preinstall base, 2nd)	12.45	7	4	41.9	447	36
Podman (preinstall base, 1st)	41.96	37	126	224.1	437	42
Podman (preinstall base, 2nd)	10.85	6	13	48.1	425	28

Table 5.4 demonstrates the most substantial containerization performance gains across all test scenarios. Traditional Docker achieves 13 s OBS build time, while chroot requires 14 s OBS build time. Traditional Podman exhibits severe performance degradation (96 s OBS build) as it must install extensive development dependencies individually without preinstall image support in rootless environments.

Containerized approaches with heavy preinstall images reveal exceptional performance improvements. First-time runs incorporate the heavy preinstall image import process, with Docker achieving 72 s OBS build time and Podman requiring 37 s OBS build time. Podman imports the heavy preinstall image in approximately 30 seconds, demonstrating efficient container management. Second-run builds eliminate the import overhead, with Docker achieving 7 s OBS build time and Podman achieving 6 s OBS build time.

Analysis shows the transformative impact of heavy preinstall images on containerization benefits. Podman containerization delivers extraordinary performance improvements in complex enterprise environments. Heavy preinstall images enable Podman to achieve 6 s OBS build time—a 90-second improvement over traditional Podman's 96 s build time.

Docker containerization with heavy preinstall images achieves 7 s OBS build time compared to 13 s traditional, delivering a 6-second improvement. Containerized Podman achieves superior 6 s OBS build time, outperforming both traditional Docker (13s) and containerized Docker (7s). The heavy preinstall image configuration demonstrates that containerization scales effectively with complex enterprise build requirements while maintaining exceptional performance.

CPU and memory usage exhibit the same pattern as previous evaluations, scaling with image size. Traditional Docker (2% CPU, 41.6 MB) and containerized Docker (4% CPU, 41.9 MB) maintain efficient resource usage across runs. Podman exhibits the highest resource consumption observed during first-run import (126% CPU, 224.1 MB), reflecting the substantial size of the enterprise-grade image, but returns to reasonable levels in subsequent runs (13% CPU, 48.1 MB).

Network efficiency remains excellent across containerized builds. Docker containerization reduces network traffic by 47% (447 KB vs 840 KB traditional Docker) but uses slightly more than chroot (447 KB vs 429 KB). Podman containerization achieves 49% reduction compared to traditional Docker (425 KB vs 840 KB) and maintains similar efficiency to chroot (425 KB vs 429 KB).

5.6 EVALUATION RESULTS

The evaluation across three preinstall image configurations reveals distinct performance patterns between traditional and containerized build approaches.

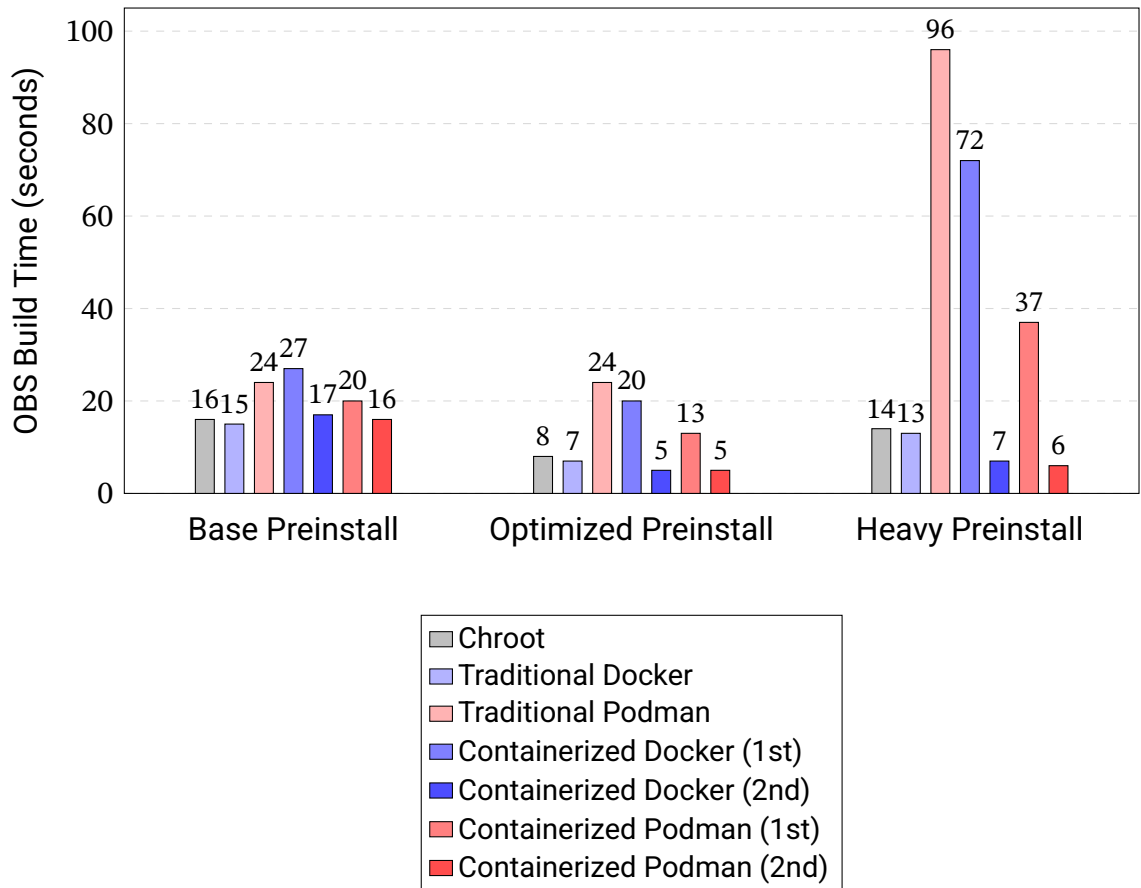


Figure 5.1: OBS build time comparison across preinstall image configurations (cached builds).

Figure 5.1 illustrates the OBS build times across different preinstall image configurations. Traditional Podman shows severe performance degradation as preinstall image complexity increases, particularly evident in the heavy preinstall scenario (96 s) due to its inability to utilize preinstall images in rootless environments. Containerized approaches demonstrate consistent performance regardless of preinstall image size, with both Docker and Podman achieving exceptional build times (5 s for optimized images, 6–7 s for heavy images) in subsequent runs after initial image import. The results demonstrate that using preinstall images as container base images proves most beneficial when preinstall images are heavier and include more package dependencies, delivering substantial performance improvements over traditional tar extraction approaches.

Comparing the two containerization platforms, Podman demonstrates superior performance compared to Docker across all evaluated scenarios. Podman consistently achieves faster image import times: 4 seconds versus Docker’s 8 seconds for basic images, 8 seconds versus 15 seconds for optimized images, and approximately 30 seconds for heavy images. These faster imports come with elevated CPU and memory consumption during first-run imports that scale with image size (30% to 126% CPU, 105.9 MB to 224.1 MB memory), but resource consumption returns to efficient levels in subsequent runs. Network efficiency gains and second-run timing remain nearly identical between both platforms.

Beyond performance metrics, the containerization approach delivers a critical capability for rootless environments. Traditional Podman cannot utilize preinstall images in rootless configurations, resulting in severe performance penalties. Containerized Podman enables preinstall image usage for the first time in rootless environments, delivering performance improvements ranging from 8 seconds (basic images) to 90 seconds (heavy images) in OBS build times.

6 MONITORING FRAMEWORK FOR FUTURE PRODUCTION DEPLOYMENT

To validate the containerization implementation and assess its potential impact in OBS production environments, a monitoring framework was developed. This chapter describes how the framework collects build performance data from GitLab CI/CD pipelines, processes it through PostgreSQL, and visualizes results using Apache Superset dashboards. The framework establishes performance baselines of current chroot-based OBS infrastructure while preparing for future measurement of containerization benefits once enterprise-scale deployment becomes feasible.

6.1 DATA COLLECTION

To analyze the performance impact of migrating from traditional chroot builds to containerization in production environments, a data collection system was developed to gather build performance metrics from existing GitLab CI/CD pipelines.

The monitoring framework integrates with GitLab's API to collect build event data from merge request pipelines that utilize OBS for package building. A custom Python script (Appendix A) systematically gathers performance data from GitLab CI/CD executions. The script performs the following data collection operations:

- **Merge Request Discovery:** Queries GitLab API to identify merge requests within specified project groups, filtering by time ranges to capture build data
- **Pipeline Event Extraction:** Retrieves commit status information for each merge request version, capturing build start times, completion times, and execution status
- **Build Metadata Collection:** Gathers contextual information including target branches, labels, repository identifiers, and merge request URLs to correlate performance with specific projects and build types
- **Automated Data Export:** Packages collected data into compressed archives and uploads to monitoring framework for analysis

The data collection system operates in two distinct modes to balance coverage with real-time monitoring capabilities:

Cumulative Collection: Runs once daily to gather all available pipeline data from GitLab, providing complete historical datasets for trend analysis and long-term performance comparison. This mode ensures no build events are missed and maintains a complete record of all OBS build activities across all monitored projects.

Incremental Collection: Executes frequently (typically every 30 minutes) to collect only recent build data from the last X minutes, where X is configurable based on

monitoring requirements. This mode enables near real-time performance monitoring and rapid detection of performance changes or issues following the chroot-to-container migration.

This dual-mode approach ensures complete data coverage while providing timely insights into build performance changes, enabling both historical analysis and immediate response to performance variations. For each build execution, the monitoring framework captures performance metrics including timing metrics (build creation, start, and completion times), success rates (build status tracking for success or failure), and contextual data (repository information, branch details, and merge request metadata for segmented analysis).

6.2 DATA ANALYSIS

The collected GitLab pipeline data is processed and analyzed to extract performance insights. A data processing pipeline transforms the raw JSON data from GitLab API into structured database records for statistical analysis and visualization.

The analysis infrastructure utilizes a PostgreSQL database with a schema to store and analyze GitLab pipeline events. The primary table `obs.gitlab_events` captures essential build performance metrics as shown in Code 6.1.

```

1 CREATE TABLE obs.gitlab_events (
2     id bigint NOT NULL,
3     name text NOT NULL,
4     git_repo text NOT NULL,
5     git_target_branch text NOT NULL,
6     status text NOT NULL,
7     pipeline_id integer NOT NULL,
8     created_at timestamp with time zone NOT NULL,
9     started_at timestamp with time zone,
10    finished_at timestamp with time zone,
11    CONSTRAINT gitlab_events_check CHECK ((started_at <=
12    ↪ finished_at)),
13    CONSTRAINT gitlab_events_check1 CHECK ((created_at <=
14    ↪ started_at))
15 );

```

Code 6.1: GitLab events database schema.

The schema includes temporal constraints and unique constraints to prevent duplicate event records, enabling efficient querying of build duration, success rates, and performance trends across different repositories and branches.

Raw GitLab API data undergoes transformation through a Ruby-based processing script (Appendix B) that handles data validation, deduplication, and database insertion. The processing pipeline performs several operations:

- **Data Validation:** Filters out incomplete events by rejecting records with

missing timestamps or invalid status values, ensuring only completed builds are analyzed.

- **Temporal Validation:** Ensures logical timestamp ordering, rejecting events where timestamps violate the $\text{created} \leq \text{started} \leq \text{finished}$ sequence.
- **Event Deduplication:** Uses composite keys to prevent duplicate records during overlapping data collections.
- **Batch Processing:** Generates SQL statements with PostgreSQL's bulk insert capabilities for efficient processing of large datasets.

6.3 VISUALIZATION

The processed GitLab pipeline data is visualized using Apache Superset to create interactive dashboards for monitoring build performance trends and analyzing the impact of the chroot-to-container migration. The visualization infrastructure enables stakeholders to track performance improvements and identify optimization opportunities.

6.3.1 DATASET CREATION

Apache Superset datasets are created using custom SQL queries that transform the raw event data into analysis-ready formats. The primary dataset focuses on build events with calculated performance metrics as shown in Code 6.2.

```

1 WITH build_events AS (
2   SELECT
3     git_repo,
4     git_target_branch,
5     pipeline_id,
6     created_at AT TIME ZONE 'Europe/Helsinki' AS created_at,
7     finished_at AT TIME ZONE 'Europe/Helsinki' AS finished_at,
8     ABS(EXTRACT(EPOCH FROM (finished_at - created_at)) / 60) AS
      ↪ build_time
9   FROM
10    gitlab.events
11  WHERE
12    AND name = 'ci/build'
13    AND finished_at > created_at
14 )
15 SELECT
16   created_at,
17   finished_at,
18   git_repo,
19   git_target_branch,
20   pipeline_id,
21   build_time,
22   AVG(build_time) OVER (PARTITION BY git_repo, git_target_branch
23                        ORDER BY created_at
24                        RANGE BETWEEN INTERVAL '1 WEEK'
25                        ↪ PRECEDING
26                        AND CURRENT ROW) AS trend
27 FROM
28   build_events
29 ORDER BY created_at DESC;

```

Code 6.2: Apache Superset dataset query for build performance analysis.

The query implements several key transformations for effective visualization:

- **Timezone Conversion:** Converts UTC timestamps to Europe/Helsinki time-zone for local time analysis, ensuring charts display times relevant to the development team's working hours.
- **Build Duration Calculation:** Computes build time in minutes using the difference between finished_at and created_at timestamps, providing intuitive time measurements for performance analysis.
- **Trend Analysis:** Implements a rolling 7-day average using window functions to smooth out daily variations and highlight longer-term performance trends across repositories and branches.
- **Data Quality Filtering:** Validates temporal consistency (finished_at > created_at) to ensure accurate performance measurements.

6.3.2 DASHBOARD COMPONENTS

The Superset dashboard provides build performance analysis through multiple Mixed Time-Series charts, with one chart per GitLab repository-branch combination. Each chart displays both individual build durations and overall trend analysis for its specific GitLab repository and branch. The chart creation utilizes the dataset metrics and columns available through Superset's interface, as shown in Figure 6.1.

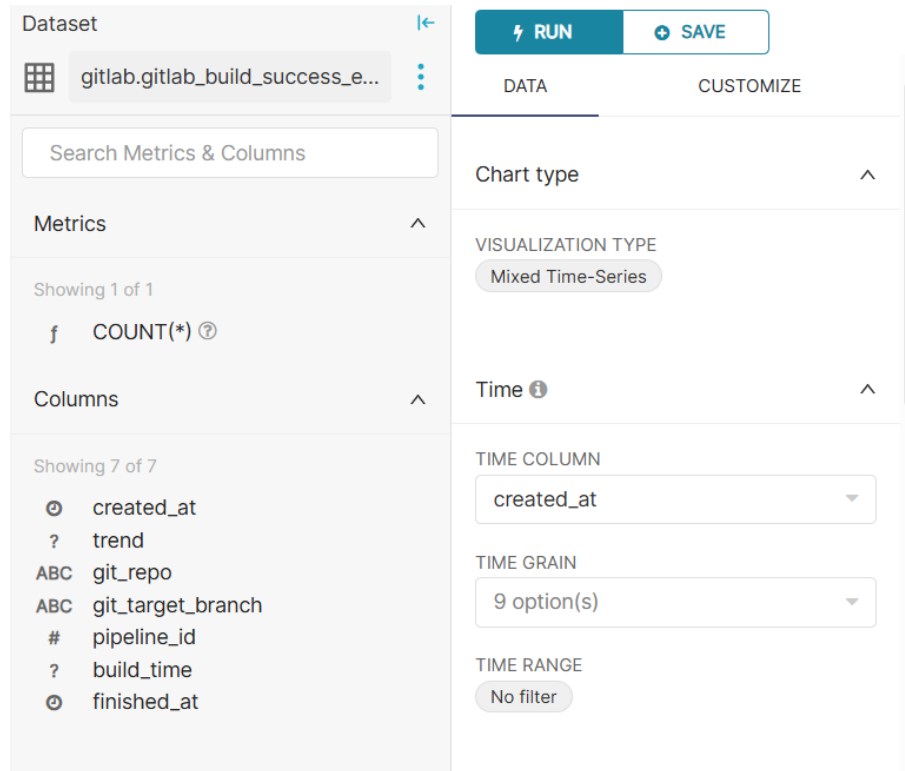


Figure 6.1: Available metrics and columns in Superset dataset.

Each Mixed Time-Series chart combines two queries to display build performance data, as illustrated in Figure 6.2. Query A displays individual build events by utilizing $\text{AVG}(\text{build_time})$ grouped by `git_repo`, `git_target_branch`, `build_time`, and `created_at`, with filters applied to focus on a specific GitLab repository and branch. This creates individual data points for every build event in the specified repository-branch combination. Query B provides trend analysis using $\text{AVG}(\text{trend})$ grouped by `git_repo` and `git_target_branch`, with the same repository and branch filters. The trend metric, calculated as described in Section 6.3.1, displays the rolling 7-day average trend line alongside the individual build durations.

The image shows two side-by-side query configuration panels for Superset. The left panel is for 'Query A' and the right panel is for 'Query B'. Both panels have a similar layout with sections for METRICS, GROUP BY, FILTERS, SERIES LIMIT, SORT BY, and ROW LIMIT.

Query A Configuration (Left):

- METRICS:** $f(x)$ AVG(build_time)
- GROUP BY:** git_repo, git_target_branch, build_time, created_at (3 option(s))
- FILTERS:** git_repo = 'xxxx', git_target_branch = 'main'
- SERIES LIMIT:** 100
- SORT BY:** + Add metric
- SORT DESCENDING:**
- ROW LIMIT:** 10000

Query B Configuration (Right):

- METRICS:** $f(x)$ AVG(trend)
- GROUP BY:** git_repo, git_target_branch (5 option(s))
- FILTERS:** git_repo = 'xxxx', git_target_branch = 'main'
- SERIES LIMIT:** 100
- SORT BY:** + Add metric
- SORT DESCENDING:**
- ROW LIMIT:** 10000

Figure 6.2: Query A configuration (left) for individual build events and Query B configuration (right) for rolling average trend line.

6.4 PERFORMANCE EXAMPLES

The Superset dashboards provide monitoring of GitLab CI/CD build performance, establishing baseline measurements for various packages built in the current chroot-based OBS infrastructure. The containerization features developed in this thesis have not been deployed to production, as enterprise-scale production evaluation was not feasible within the thesis timeframe due to organizational constraints preventing OBS version upgrades according to the thesis schedule. The monitoring framework captures performance patterns across different package types and complexities, preparing for future production deployment validation.

The monitoring data reveals diverse performance characteristics across different packages in the existing chroot-based build system. Build times vary significantly based on package complexity, dependency requirements, and compilation processes. Figures 6.3–6.9 demonstrate performance patterns across seven different packages.

In the performance charts, colored dots represent individual build events, while the line shows the 7-day rolling average trend that smooths out daily variations to highlight longer-term performance patterns. Some builds display near-zero duration due to the company's pipeline architecture: when pipelines are retriggered without source changes, the system returns the cached build status and proceeds directly to testing, avoiding redundant compilation while preserving complete job history.

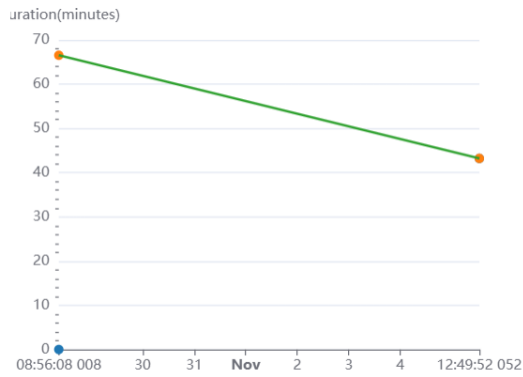


Figure 6.3: Build performance for package A.

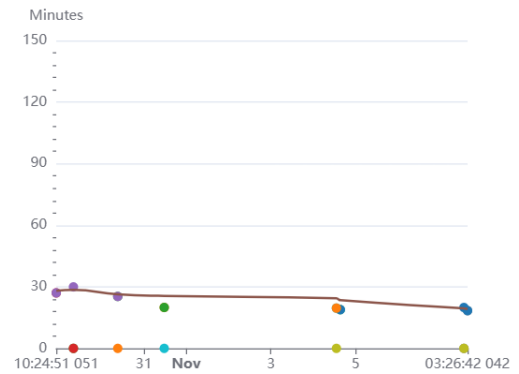


Figure 6.4: Build performance for package B.

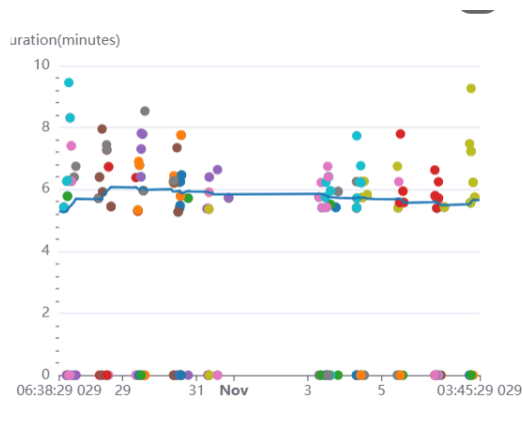


Figure 6.5: Build performance for package C.

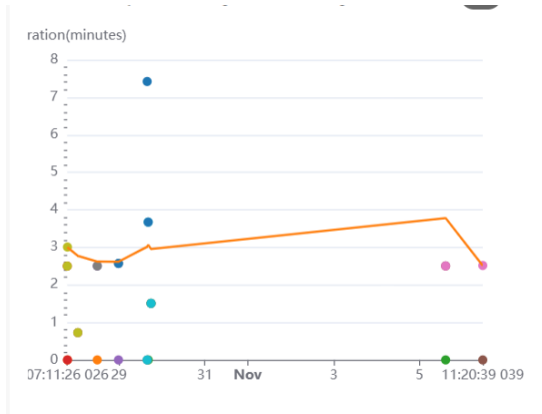


Figure 6.6: Build performance for package D.

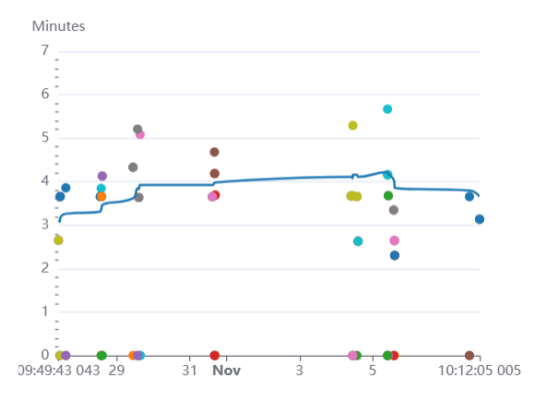


Figure 6.7: Build performance for package E.

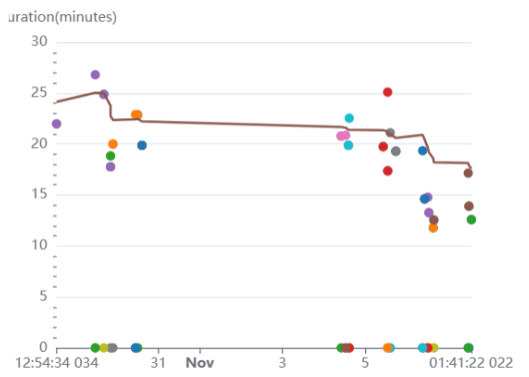


Figure 6.8: Build performance for package F.

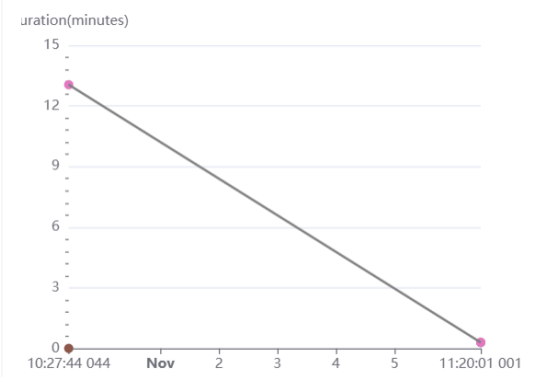


Figure 6.9: Build performance for package G.

These baseline measurements provide the foundation for future performance comparison when containerization deployment becomes feasible.

7 CONCLUSION AND FUTURE WORK

This thesis addressed the performance bottleneck in Open Build Service (OBS) build processes by implementing containerization to eliminate preinstall image tar extraction overhead. The solution resolves existing performance limitations and unlocks preinstall image functionality in rootless environments where it was previously unavailable.

The implementation required coordinated modifications across three core OBS components. The OSC client was enhanced to download preinstallimage.info files and enable Podman to use preinstall images as container base images—functionality previously impossible in unprivileged environments due to tar extraction limitations. The build scripts were modified to support containerization logic with MD5 hash-based image tagging to prevent redundant imports, prioritizing package metadata from preinstallimage.info files over tar extraction. The OBS worker was modified to support Podman as a virtualization type alongside LXC and Docker, and to provide preinstallimage.info file downloads alongside preinstallimage tar files. This coordinated implementation eliminated the tar extraction bottleneck while maintaining OBS security requirements, including network isolation during build execution.

To validate the implementation, laboratory evaluation was conducted using the public openSUSE OBS instance with specific test packages across multiple scenarios. Podman consistently outperforms Docker through faster image import speeds, delivering build time improvements ranging from 8 to 90 seconds depending on image complexity and achieving optimal 5–6 second build times with heavy preinstall images. While Podman exhibits higher CPU and memory consumption during first-run imports, resource usage returns to reasonable levels from the second run onward. Both platforms deliver network efficiency improvements of 44–49%, and Podman’s daemonless architecture provides additional security benefits, making it the preferred choice for reducing build times. The approach shows greatest benefits when preinstall images remain stable while builds occur frequently.

Enterprise-scale production evaluation was not feasible within the thesis timeframe due to organizational constraints preventing OBS version upgrade. However, a monitoring framework was developed to enable future production validation. The framework collects build performance data from GitLab CI/CD pipelines, processes it through PostgreSQL, and visualizes results using Apache Superset dashboards with charts displaying individual build events and 7-day rolling average trends. This establishes baseline performance measurements for future comparison.

Future research should extend evaluation to broader package ecosystems and investigate optimization opportunities for specific enterprise build patterns. Additional opportunities include exploring OBS deployment on orchestration platforms like Kubernetes to leverage container orchestration capabilities, and conducting

comparative analysis of different container engine versions to understand how updates affect OBS build performance.

REFERENCES

- [1] Reproducible Builds Project. *Reproducible Builds*. Accessed: 2025-10-26. URL: <https://reproducible-builds.org/>.
- [2] openSUSE Project. *Open Build Service*. Accessed: 2025-10-02. URL: <https://openbuildservice.org/>.
- [3] openSUSE Wiki. *openSUSE:Build Service installations*. Accessed: 2025-11-06. URL: https://en.opensuse.org/openSUSE:Build_Service_installations.
- [4] SUSE. *Open Build Service User Guide - Building Preinstall Images*. Accessed: 2025-10-15. URL: <https://openbuildservice.org/help/manuals/obs-user-guide/cha-obs-build-preinstall>.
- [5] openSUSE Wiki. *Portal:Build Service*. Accessed: 2025-10-16. URL: https://en.opensuse.org/Portal:Build_Service.
- [6] SUSE. *Open Build Service User Guide - Beginner's Guide*. Accessed: 2025-10-18. URL: <https://openbuildservice.org/help/manuals/obs-user-guide/art-obs-bg>.
- [7] SUSE. *Open Build Service User Guide - OBS Architecture*. Accessed: 2025-10-18. URL: <https://openbuildservice.org/help/manuals/obs-user-guide/cha-obs-architecture>.
- [8] openSUSE Project. *Open Build Service*. Accessed: 2025-10-16. URL: <https://github.com/openSUSE/open-build-service>.
- [9] SUSE. *Open Build Service Administrator Guide - Installation and Configuration*. Accessed: 2025-11-06. URL: <https://openbuildservice.org/help/manuals/obs-admin-guide/obs-cha-installation-and-configuration>.
- [10] SUSE. *Open Build Service User Guide - OBS Concepts*. Accessed: 2025-10-18. URL: <https://openbuildservice.org/help/manuals/obs-user-guide/cha-obs-concepts>.
- [11] SUSE. *Open Build Service Administrator Guide - Security Concepts*. Accessed: 2025-10-16. URL: <https://openbuildservice.org/help/manuals/obs-admin-guide/obs-cha-security-concepts>.
- [12] SUSE. *Open Build Service User Guide - Build Process*. Accessed: 2025-10-18. URL: <https://openbuildservice.org/help/manuals/obs-user-guide/cha-obs-build-process>.
- [13] Red Hat. *Introduction to RPM*. Accessed: 2025-10-05. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/packaging_and_distributing_software/introduction-to-rpm_packaging-and-distributing-software.
- [14] Red Hat. *Packaging software - About spec files*. Accessed: 2025-10-05. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/packaging_and_distributing_software/packaging_software_packaging-and-distributing-software.
- [15] openSUSE Wiki. *openSUSE:OSC*. Accessed: 2025-10-18. URL: <https://en.opensuse.org/openSUSE:OSC>.
- [16] openSUSE Project. *openSUSE Commander (osc)*. Accessed: 2025-10-18. URL: <https://github.com/openSUSE/osc>.

- [17] openSUSE Project. *obs-build: Build tool for binary packages*. Accessed: 2025-10-15. URL: <https://github.com/openSUSE/obs-build>.
- [18] SUSE. *Open Build Service User Guide - Working with Limited Bandwidth*. Accessed: 2025-11-05. URL: <https://openbuildservice.org/help/manuals/obs-user-guide/cha-obs-best-practices-howto>.
- [19] D. Merkel. "Docker: Lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (Mar. 2014), pp. 76–90. ISSN: 1075-3583.
- [20] J. S. Hale et al. "Containers for Portable, Productive, and Performant Scientific Computing". In: *Computing in Science & Engineering* 19.6 (2017), pp. 40–50. doi: [10.1109/MCSE.2017.2421459](https://doi.org/10.1109/MCSE.2017.2421459).
- [21] A. Arrichiello and G. Salinetti. *Podman for DevOps: containerization reimaged with Podman and its companion tools*. First edition. Birmingham, UK: Packt Publishing, 2022. ISBN: 9781803248967.
- [22] C. Binnie and R. McCune. "What Is A Container?" In: *Cloud Native Security*. John Wiley & Sons, 2021, pp. 3–15.
- [23] Red Hat Developer. *A Practical Introduction to Container Terminology*. Accessed: 2025-10-19. Feb. 2018. URL: <https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction>.
- [24] Docker Inc. *Docker Overview*. Accessed: 2025-10-19. URL: <https://docs.docker.com/get-started/docker-overview/>.
- [25] Docker Inc. *Docker Network Drivers*. Accessed: 2025-11-06. URL: <https://docs.docker.com/engine/network/drivers/>.
- [26] Red Hat Inc. *Podman Documentation*. Accessed: 2025-10-13. URL: <https://docs.podman.io/en/latest/>.
- [27] Red Hat. *Podman Run - Network Options*. Accessed: 2025-11-06. URL: <https://docs.podman.io/en/latest/markdown/podman-run.1.html>.
- [28] GitLab Inc. *DevOps Platform*. Accessed: 2025-10-21. URL: <https://about.gitlab.com/solutions/devops-platform/>.
- [29] J. Painter. *Practical GitLab Services: A Complete DevOps Guide for Developers and Administrators*. 1st ed. Berkeley, CA: Apress L. P, 2024. ISBN: 9798868804267.
- [30] GitLab Inc. *The GitLab Platform*. Accessed: 2025-10-21. URL: <https://about.gitlab.com/platform/>.
- [31] Apache Software Foundation. *Apache Superset Documentation*. Accessed: 2025-10-21. URL: <https://superset.apache.org/docs/intro>.
- [32] S. Shekhar. *Apache superset quick start guide : develop interactive visualizations by creating user-friendly dashboards*. 1st edition. Birmingham: Packt, 2018. ISBN: 9781788999564.
- [33] openSUSE. *Add support for preinstallimage.info and container preinstallimage*. GitHub commit. 2025. URL: <https://github.com/openSUSE/osc/commit/57bdcf2ef3e8b9bfbc108bac6899325f82b4400a>.
- [34] openSUSE. *Add support for using preinstallimage as container base image*. GitHub pull request. 2025. URL: <https://github.com/openSUSE/obs-build/pull/1105/files>.
- [35] openSUSE. *Add container preinstallimage support to OBS worker*. GitHub pull request. 2025. URL: <https://github.com/openSUSE/open-build-service/pull/18803>.

APPENDIX A: GITLAB EVENTS DATA COLLECTION SCRIPT

This appendix contains the complete implementation of the `gitlab_events.py` script used for collecting build performance data from GitLab CI/CD pipelines. The script integrates with GitLab's API to systematically gather pipeline events for performance analysis of the chroot-to-container migration.

The script implements the following key functions:

- **get()**: Handles GitLab API requests with rate limiting and error handling
- **paged_get()**: Manages paginated API responses to collect all available data
- **get_merge_requests()**: Discovers merge requests within specified groups, with optional time filtering for incremental collection
- **get_events()**: Extracts commit status events from merge request versions, capturing build timing and metadata

The script operates in two modes based on command-line arguments: cumulative collection (no arguments) for complete historical data, and incremental collection (with `minutes` parameter) for recent events only.

```

1 import os, tarfile, json, datetime, requests, pytz, sys
2 from time import sleep
3
4 GITLAB_HOST="https://gitlab.com"
5 API_SLUG="api/v4"
6 GROUPS=["mygroup"]
7 token=""
8 PROXYS={}
9
10 def get(Session, req, params={}, page=None, proxies=PROXYS):
11     url=f"{GITLAB_HOST}/{API_SLUG}/{req}"
12     response = Session.get(url, params=params, proxies=proxies)
13     response.close()
14
15     if response.status_code == 429:
16         # Handle rate limit exceeded
17         retry_after = int(response.headers.get('Retry-After',
18         ↵ 60))
19         print(f"Rate limit exceeded. Retrying after
20         ↵ {retry_after} seconds.")
21         sleep(retry_after)
22         return get(req, params, page, proxies)
23
24     else:
25         if response.status_code != requests.codes.ok:
26             response.raise_for_status()
27         else:
28             if page:
29                 return response
30             else:
31                 return response.json()
32
33 def paged_get(Session, req, params={}, page=1):
34     response = get(Session, req, params=None, page=1)
35     for i in response.json():
36         yield i
37     pages = int(response.headers.get("x-total-pages", 1))
38     while pages > page:
39         page = page + 1
40         params.update({"page" : str(page)})
41         sleep(5)
42         response = get(Session, req, params=params, page=page)
43         for i in response.json():
44             yield i

```

Code A.1: Complete GitLab Events Data Collection Script - Part 1

```

1 def get_merge_requests(Session, start_time=None):
2     all_merge_requests = []
3     keys = ["id", "iid", "project_id", "title", "description",
4           ↪ "state",
5           ↪ "created_at", "updated_at", "merged_at",
6           ↪ "closed_at",
7           ↪ "target_branch", "source_branch", "web_url",
8           ↪ "labels", "merge_status"]
9     for group in GROUPS:
10        # this would also get the subgroups
11        group_details = paged_get(Session,
12        ↪ f"groups/?search={group}")
13        for subgroup in group_details:
14            merge_requests = paged_get(Session,
15            ↪ f"groups/{subgroup['id']}/m,
16            ↪ erge_requests")
17            for merge_request in merge_requests:
18                mr = {key: merge_request[key] for key in keys}
19                if start_time is not None:
20                    if datetime.datetime.fromisoformat(
21                    ↪ mr["updated_at"].replace("Z", "+00:00")
22                    ↪ ).astimezone(finnish_tz) > start_time:
23                        all_merge_requests.append(mr)
24                else:
25                    all_merge_requests.append(mr)
26    return all_merge_requests

```

Code A.2: Complete GitLab Events Data Collection Script - Part 2

```

1 def get_events(Session, merge_requests):
2     events = []
3     for merge_request in merge_requests:
4         versions = paged_get(Session,
5                               f"projects/{merge_request['project_id',
6                               ↵ ']}")
7                               f"/merge_requests/{merge_request['iid',
8                               ↵ ']}versions")
9         for version in versions:
10            if version["head_commit_sha"]:
11                commit_statuses = paged_get(Session,
12                                              f"projects/{merge_request['project_id']}/rep_
13                                              ↵ ository"
14                                              f"/commits/{version['head_commit_sha']}/stat_
15                                              ↵ uses")
16                for commit_status in commit_statuses:
17                    commit_status["merge_id"] =
18                        ↵ merge_request["id"]
19                    commit_status["repo"] =
20                        ↵ merge_request["web_url"].split("/")[4]
21                    commit_status["target_branch"] =
22                        ↵ merge_request["target_branch"]
23                    commit_status["labels"] =
24                        ↵ merge_request["labels"]
25                    commit_status["merge_url"] =
26                        ↵ merge_request["web_url"]
27                    if (commit_status["created_at"] and
28                        commit_status["started_at"] and
29                        commit_status["finished_at"]):
30                        events.append(commit_status)
31                    else:
32                        print(f"Unusual event: {commit_status}")
33            else:
34                print(f"Not tracking {merge_request['web_url']}
35                ↵ "
36                    f"version: {version}")
37        return events
38
39 if __name__ == "__main__":
40     if not token:
41         with open("/run/secrets/token", "r") as file:
42             token = file.read().strip()
43
44     finnish_tz = pytz.timezone('Europe/Helsinki')
45     now = datetime.datetime.now(finnish_tz)
46     events_jsonfile = "gitlab_events.json"
47     merge_jsonfile = "gitlab_merge_requests.json"
48     archive = "gitlab_events.tar.bz2"
49     Session = requests.Session()
50     Session.headers.update({"Authorization": f"Bearer {token}"})
51     upload_path = "gitlab-incremental-events-1"

```

Code A.3: Complete GitLab Events Data Collection Script - Part 3

```

1  if len(sys.argv) == 1:
2      upload_path = "gitlab-cumulative-events-1"
3      print(f"Querying GitLab for all events
4          ↪ {datetime.datetime.now()} ..")
5      merge_requests = get_merge_requests(Session)
6
7  elif len(sys.argv) == 2:
8      print(f"Querying GitLab for events happened in the last
9          ↪ "
10         f"{sys.argv[1]} minutes {now} ..")
11     minutes_ago = now -
12     ↪ datetime.timedelta(minutes=int(sys.argv[1]))
13     merge_requests = get_merge_requests(Session,
14     ↪ minutes_ago)
15     if not merge_requests:
16         print(f"No new events in the last {sys.argv[1]}
17             ↪ minutes.")
18         Session.close()
19         sys.exit(1)
20
21     else:
22         print("Please run the script without any arguments for
23             ↪ all event "
24             "or with a number representing the events in last
25             ↪ X minutes.")
26         sys.exit(1)
27
28     with open(merge_jsonfile, "w") as output:
29         json.dump(merge_requests, output, indent=4)
30
31     events= get_events(Session, merge_requests)
32     with open(events_jsonfile, "w") as output:
33         json.dump(events, output, indent=4)
34
35     with tarfile.open(archive, "w:bz2") as tar:
36         tar.add(events_jsonfile)
37         tar.add(merge_jsonfile)
38
39     os.unlink(events_jsonfile)
40     os.unlink(merge_jsonfile)
41
42     print(f"GitLab events collected in {archive}, "
43         f"uploading {datetime.datetime.now()} ..")
44     os.environ['no_proxy'] = 'localhost'
45     r = requests.post(f"http://localhost:8000/data/{upload_path}",
46         ↪ "
47         ↪ files={'file': open(archive, 'rb')})
48     print(f"Gitlab events uploaded {datetime.datetime.now()}..
49         ↪ {r.text}")
50     os.unlink(archive)
51     Session.close()

```

Code A.4: Complete GitLab Events Data Collection Script - Part 4

APPENDIX B: DATA PROCESSING SCRIPT

This appendix contains the Ruby-based data processing script that transforms raw GitLab API data into structured database records for performance analysis. The script handles data validation, deduplication, and batch insertion into PostgreSQL.

The data processing script implements focused event processing with the following key features:

- **Status Filtering:** Excludes incomplete pipeline states (running, skipped, pending, created) to focus on completed builds
- **Timestamp Validation:** Ensures all required timestamps (created_at, started_at, finished_at) are present and logically ordered
- **Temporal Consistency:** Rejects events where creation or start times exceed completion times
- **Deduplication:** Uses composite keys (name, repository, branch, status, pipeline ID, merge ID) to prevent duplicate records
- **Bulk Processing:** Generates efficient SQL with PostgreSQL's bulk insert and conflict resolution capabilities

The script processes JSON data files collected by the GitLab events script and transforms them into SQL statements for database insertion, ensuring data quality and consistency for performance analysis.

```

1  #!/bin/env ruby
2
3  require 'json'
4  require 'oj'
5  require 'time'
6
7  require_relative './processor.rb'
8
9  import_loop { |dirname|
10     return puts "No event file, skipping" if not
11         ↪ File.exist?(dirname+'/gitlab_events.json')
12
13     events =
14         ↪ JSON.parse(open(dirname+"/gitlab_events.json").read)
15     merge_requests =
16         ↪ Oj.load_file(dirname+"/gitlab_merge_requests.json")
17
18     ret = <<~SQL
19         #{header(["gitlab"])}
20         DECLARE
21         BEGIN
22         SQL
23
24     # Process and validate pipeline events
25     events_to_insert = events.uniq.reject { |event|
26         ["running", "skipped", "pending",
27         ↪ "created"].include?(event["status"]) ||
28         event["created_at"].nil? ||
29         event["started_at"].nil? ||
30         event["finished_at"].nil? ||
31         Time.parse(event["created_at"]) >
32         ↪ Time.parse(event["finished_at"]) ||
33         Time.parse(event["started_at"]) >
34         ↪ Time.parse(event["finished_at"])
35     }.map { |event|
36         "\t\t(#{s event["name"]},#{s event["repo"]},#{s
37         ↪ event["target_branch"]}," +
38         "#{s event["status"]},#{event["pipeline_id"]},#{event["m_
39         ↪ erge_id"]}," +
40         "#{as event["labels"]},#{t
41         ↪ event["created_at"]}::timestampz," +
42         "#{t event["started_at"]}::timestampz,#{t
43         ↪ event["finished_at"]}::timestampz)"
44     }.join(",\n")

```

Code B.1: GitLab Events Data Processing Script - Part 1

```

1  if events_to_insert.size > 0
2      ret += <<~SQL
3          WITH to_insert (name, git_repo, git_target_branch,
4              ↪ status, pipeline_id,
5                  merge_id, labels, created_at,
6                  ↪ started_at, finished_at) AS
7                  ↪ (VALUES
8                      ↪ #{events_to_insert}
9                  )
10         INSERT INTO events (name, git_repo,
11             ↪ git_target_branch, status, pipeline_id,
12                 ↪ merge_id, labels, created_at,
13                 ↪ started_at, finished_at)
14         SELECT * from to_insert
15         WHERE NOT EXISTS(SELECT * FROM events
16             WHERE to_insert.name = events.name
17                 ↪ AND
18                     to_insert.git_repo =
19                         ↪ events.git_repo AND
20                     to_insert.git_target_branch =
21                         ↪ events.git_target_branch
22                     ↪ AND
23                     to_insert.status =
24                         ↪ events.status AND
25                     to_insert.pipeline_id =
26                         ↪ events.pipeline_id AND
27                     to_insert.merge_id =
28                         ↪ events.merge_id)
29         ON CONFLICT DO NOTHING;
30     SQL
31 end
32
33 ret += "END $$;"
34 ret
35 }

```

Code B.2: GitLab Events Data Processing Script - Part 2