VILNIUS UNIVERSITY
KAUNAS FACULTY

INSTITUTE OF SOCIAL SCIENCES AND APPLIED INFORMATICS

Study programme Information Systems and Cyber Security
State code 6121BX003

SHUBHAM BHASKER

BACHELOR'S THESIS

LINUX FORM SCRATCH - LFS

VILNIUS UNIVERSITY
KAUNAS FACULTY

INSTITUTE OF SOCIAL SCIENCES AND APPLIED INFORMATICS

SHUBHAM BHASKER

BACHELOR'S THESIS

LINUX FROM SCRATCH - LFS

Allowed to defend _____

Bachelor student _____
(signature)

Scientific advisor _____
(signature)

_____
(scientific degree of the advisor, scientific pedagogical name, name and surname)

Thesis submitted on _____

Registration No: _____

Contents

LIST OF ABBREVIATIONS

LFS	Linux from scratch
UI	User interface
ISO	International Organization for Standardization
GRUB	Grand Unified Bootloader
ISOLINUX	A boot loader for Linux that operates from a CD/DVD or USB drive
FN	Functional Requirement
NFN	Non- Functional Requirement
OS	Operating System
IoT	Internet of Things
GUI	Graphic User Interface
CI/CD	Continuous Integration/ Continuous Development
RAM	Random Access Memory
JWT	JSON web tokens
RLS	Row level security
API	Application Programming Interface
DB	Database

INTRODUCTION

The process of creating a Linux operating system from scratch-sometimes called Linux from scratch, or LFS-is very rewarding. It is known to be somewhat difficult, but it gives you a really deep understanding of how a Linux system works from top to bottom, switching frequently between a number of user contexts, which may be hard and time-consuming.

It appears that the typical Linux From Scratch build procedure often offers a somewhat delicate balancing proposition, where a single mistake or setting issue potentially will require repeated jumps between user sessions, whether it is root, the specified Linux build user, or actions within the chroot environment. This alone offers the promise of becoming a significant hindrance, certainly for a novice, such as the student Main, who will face these problems for the very first time.

To tackle these issues, my thesis focuses on Linux from scratch, a visual automation framework I designed to make the LFS build process smoother and manageable.

The traditional LFS build often feels like a tightrope, one wrong command or environment setting can result in having to constantly switch between user context (like root), the LFS user, or within the chroot environment), these demands can be a significant barrier, particularly for students like main who are attempting to overcome these challenges for the first time.

However, building LFS manually using the steps provided in the documents poses considerable hurdles for easy adoption. The entire build process consumes about 10-15 hours for compiling, and it also requires careful execution of more than 200 commands involved in various build processes, along with a properly installed Linux host environment. Many build processes are often thwarted by environment-related inconsistencies, dependencies, and compilation errors, especially for users new to Linux operating-system level functionality. This reduces the use of LFS, even though it is very informative for computer science, IS, and cyber security students learning the details of operating systems.

Additionally, the state of build automation and reproducibility that exists today and is germane to software development includes containerization, cloud-native systems, and Infrastructure as Code. However, the technological merit of LFS not with standing, it is deficient in contemporary technology that adheres to the tenets of DevOps and cloud computing. This not only presents a challenge for the adoption of LFS, but it also offers the opportunity of adopting new software development approaches within the established educational paradigm.

Research Problem

Existing solutions for the automation of Linux From Scratch, notably the Automated Linux From Scratch system (ALFS) and ALFS, handle various aspects of the automation process but clearly have several shortcomings. ALFS is defined using complex XML files and fails to leverage contemporary cloud containerization abilities. LFS provides automation for the script creation process but still demands a Linux host environment and fails to offer cloud-based processing. ALFS fails to provide a learning environment, web interface, or cloud-native hosting.

The problem underlying the research work can be stated as: In what manner can the build system for Linux From Scratch be updated using containerization and cloud computing, and also provide a learning platform for the explanation of concepts of operating systems?

This problem statement relates to technological challenges, which include build reproducibility, resource isolation, and container processes, among others. Architectural challenges include scalable cloud infrastructure, state, and storage, among others. Lastly, the problem statement also relates to pedagogical challenges that include learning environment interaction, among others.

Object of the Thesis

This thesis explores the automated build process for Linux From Scratch, specifically focusing on the build process for version 12.0 of Linux, along with the web-based learning platform.

Issues discussed include: the process of compiling Linux, compiling the system libraries and building the kernel, building the environment for Linux, cloud assistance for executing the build process, the delivery of learning material, and the processes involving user interaction for the submission of their building and the receipt of the results.

Aim of the Thesis

This work will design a cloud-based, containerized Linux From Scratch build automation system that also features an integrated educational platform. This will help achieve the following goals:

- Simplify the process of building.
- Provide reproducibility.
- Offer learning material for building operating systems.

For the achievement of these goals, the following objectives are formulated.

- To examine current approaches for the automation of Linux for Scratch (LFS) systems (ALFS), assess their limitations, and formulate the requirements for the new approach.
- To build a containerized build environment using Docker multi-stage build processes promote reproducibility, isolate dependencies, and are viable on the serverless container platform of Google Cloud Run.
- To provide build orchestration capability on cloud infrastructure using Firebase Cloud Functions and Google Cloud Run, facilitating asynchronous build processing, real-time build status monitoring, and automatically storing build results on Google Cloud Storage.
- To create an online learning platform using Next.js and React that provides interactive tutorials for LFS, terminal emulation, tracking, and documentation of the construction process.
- To assess the system for functionality (functional correctness, successful builds of Chapter 5 of the LFS book), performance (build time, resource use), and usability (ease of access of the interface, success of the learning process).

Difficulties and Limitations

Several technical challenges and limitations were encountered:

- Cloud Run Timeout Restrictions: The jobs that run on Cloud Run cannot take longer than 60 minutes, enforcing the requirement for a staged build process, since a large file system build, explained in Chapters 5-8, exceeds the time limit without using job chaining.
- Chroot Environment Complexity: To achieve the build isolation provided by the use of chroot within Docker containers, it is essential for the container to run with privileged execution and careful control of the filesystem namespaces

1. ANALYTICAL PART

The analysis phase of this thesis will explore the current state of the problem space of build automation for Linux From Scratch, assess current state-of-the art solutions, and identify the requirements for a cloud-native, modernized implementation.

This chapter will have three major parts. Section 1.1 will outline the problem definition stage, assessing the build technology of the Linux for Smartphone, the educational environment, the current challenges faced, and the current technological environment. Section 1.2 will examine the flow of data for both the manually performed Linux for Smartphone build process and the current technologies used, assessing the technology requirements. Section 1.3 will deal with requirements and technology, comparing other technologies used on the basis of defined parameters for justification of the used technology.

The analytical approach will combine the literature review of the documentation of the Literature and Facts Study (LFS) with the comparison of automation tools (ALFS, jhaLFS) and the evaluation of the technology of containerization/cloud computing. This approach will highlight the requirements of the system design defined in the following chapters.

1.1 Problem Area Characteristics

1.1.1 Linux From Scratch Build Process Overview

(Burgess, 1998)is a project that provides step-by-step guidelines for building a customized Linux operating system using the source code. This project differs from other Linux distributions, such as Ubuntu, Fedora, Debian, and so on, because other Linux distributions provide binary packages for users, but Linux From Scratch provides guidelines for compiling the operating system components manually, starting with the operating system building blocks (binutils, GCC, glibc) up to the Linux kernel. It provides total transparency for building the operating system.

The LFS 12.0 build process, which this system implements, consists of eight chapters of increasing complexity

Chapters 1 through 4 form the initial phase, which includes the identification of the requirements of the host system, forming a partition, and acquisition of the source packages, aggregating around 3.8 GB of tarballs.

Chapter 5 The temporary toolchain construction using cross-compilation methods is discussed. This chapter details the construction of a minimal, host-agnostic GCC, binutils, and glibc toolchain that consists of 15-18 packages (M4, ncurses, Bash, coreutils, and other basic tools), which will take around 4-6 hours.

The entire manual process requires 10-15 hours of compilation time (dependent on the computer hardware), the careful execution of over 200 distinct commands, and monitoring for errors. Even the expert user will often experience build errors, often because of the absence of host dependencies, incorrect syntax, or environmental mismatches.

1.1.2 Educational Context and learning objectives

LFS is a learning tool for computer science, information systems, and cybersecurity students that helps them attain familiarity with the internal workings of operating systems through hands-on building. Some of the paramount learning objectives are:

- Understanding the toolchain: The knowledge of the bootstrap process whereby the compiler compiles itself, involving the three-stage build process of the GCC compiler, differences between static and dynamic linking, and the patterns of dependencies among the system libraries.
- System Architecture: Analysis of the Linux system directory structure (for example, /bin, /lib, /etc, and /usr) for the purposes of explaining the conceptual difference between system software and user software, along with the requirements of POSIX compliance.
- Build systems: An examination of the use of autoconf/automake build systems, focusing on the construction of Makefiles, how they can be interpreted, and typical patterns of compilation used among various software packages.

- Skills for Troubleshooting: Building skill sets in compiler message interpretation, analyzing dependency-related problems, and addressing conflicts generated during the configuration phase.

The steep learning curve and time involved in performing Linux From Scratch by hand create obstacles for accessibility. Those who cannot access Linux environments, those with limited time for the time-consuming compilation phases, or persons discouraged by the command line-intensive process often do not pursue the Linux From Scratch project. An automated cloud platform that integrates learning aids will help mitigate these factors, along with other benefits.

1.1.3 External Environment and Technological Trends

The contemporary software development landscape exhibits several trends relevant to LFS automation:

- Adoption of containerization. The use of Docker containers since 2014 has grown considerably, and container technology is accepted throughout the IT industry for hosting applications, DevOps, and reproducible builds. It is estimated that Docker Hub contains over 13 million container images, and containerization has been introduced into computer science courses at the undergraduate level.
- Cloud-Native Architecture. The shift from on-premises infrastructure and the implementation of cloud computing (IaaS, PaaS, and SaaS) services introduced new concepts of scaling and deploying applications. Cloud-based serverless computing environments, such as Google Cloud Run, Amazon Web Services Lambda, and Azure Functions, allow for the processing of computationally heavy tasks without the need for infrastructure management.
- DevOps processes, such as CI/CD integration and delivery pipelines, IaC, and testing, have become mainstream. Modern software developers expect the following: the reproducibility of a build, versioning of infrastructure, and the automated process of deployment. Such expectations were not possible in the previous, non-DevOps approach, because the processes were mostly manual.

- The use of educational technology is also influenced by the growing integration of interaction features like virtual labs, browser terminals, and progress tracking systems. The evolution of learning technology on the web has also been driven by the growing effects of the COVID-19 pandemic.

These concurrent trends also offer opportunities and expectations. Such users familiar with modern development approaches will expect the automation of the Lifestyle Framework Solution to take advantage of containerization, cloud technology, and easy-to-use web interfaces, rather than requiring Linux platforms and manual setup.

1.1.4 Current Challenges and User Pain Points

Analysis of LFS community forums, mailing lists, and user feedback reveals consistent challenges:

- Host System Requirements and Compatibility: The requirement for using LFS is that the host system software must run on Linux, with particular versions of the tool chain software (GCC version 4.9+ and binutils version 2.13+), and also a compatible shell (like bash), among other requirements. For people using other systems, for example, Windows or macOS, there will also be the need for virtualization, which also brings along extra complexity. There might also be conflicts for Linux users.
- Error Recovery: In-process build errors, which are often the result of normal considerations like storage space, memory, and mismatched dependencies, rarely have well-structured recovery steps. This problem results in users performing a rebuild process.
- Environment Reproducibility: Minor variation between host environments, ranging from library versions, kernel choices, and filesystem distributions, causes irregularities in the build process. Irreproducible build processes, where a successful build on one host fails on the next, following the exact same steps, often cause confusion.

- Learning Curve: The LFS Book offers very detailed procedural discussions, but it is assumed that one knows Linux very well. Topics like "chroot", cross-compiling, and the bootstrapping of toolchains are mentioned only briefly, which can make it difficult for new users to understand the conceptual bases.
- Artifact Management: Finished LFS systems are modeled using directory trees or filesystem images. There is a void regarding the support of sharing, versioning, or other processes of the finished system artifacts.

Such challenges often trigger the need for the implementation of an automated system that aims to improve reproducibility (facilitated by containerization), accessibility (enabled by the web interface), time management (facilitated by cloud-based execution), and learning (facilitated by documentation).

1.1.5 Business and Educational Value Proposition

- For the educational institution, the benefits of using the automated Linux From Scratch (LFS) system are the following:
- Standardized laboratory environment for operating systems-related coursework.
- Lower pressure on the institutional IT infrastructure, since the build processes will take place through the cloud.
- Scalability suitable for class sizes between 20 and 100 students. Scalability suitable for class sizes between 20 and 100 students.
- Integrated progress monitoring and assessment capabilities.
- For individual learners, the system offers:
- Removal of barriers for host-system configuration.
- The ability to learn the concepts of the LFS without a prior commitment of ten or more hours per session.
- Interactive learning resources that are incorporated into the build process.
- Reusable build artifacts that support experimentation.
- For cybersecurity experts, awareness about LFS helps:

- Knowledge of system level attack surfaces
- Building personalized and secure systems
- Understanding the Concepts of the Trusted Computing Base
- A Foundation for Security Analysis of Embedded Systems
- It is the convergence of educational value, technological feasibility, and utility that forms the rationale for the thesis project.

1.2 Local LFS Build Architecture and Wizard Automation

The next parts change the information flow analysis to fit with the main idea behind the LFS Automated Build project: Local-First, Transparent Execution powered by a modern frontend wizard and strong local automation scripts. This analysis critically examines the architectural constraints of both legacy automation (ALFS/jhalfs) and the theoretical Cloud-Native Continuous Integration (CI) methodology, ultimately validating the existing local architecture.

1.2.1 Paradigm 1: The Fragility of the Manual LFS Build Flow

The Fragility of the Manual LFS Build Flow The Manual LFS process represents a very sequential, single-threaded processing of information that requires human involvement and operation. Although it retains its educational significance as a teaching exercise, this model is not very useful in a practical setup owing to its inherent statefulness and vulnerability of operations.

This means that there is a susceptibility to corruption in this process, as it is evidenced by unfinished patching, procedural inaccuracies, and a misalignment of sources. This set of properties, taken as a whole, suggests a situation that has a very high amount of Operational Risk as well as a lack of reproducibility. This lack of reproducibility translates to a set of compile results that are described as being “extremely irregular and prolonged, taking usually a whole month” in which “a lot of triaging of failures” takes place rather than actual compilation.

This large variability means that a large amount of human debugging effort needs to be exercised for determining the critical path in this software, further increasing compilation time due to heavy reliance upon very qualified personnel. Also, a normal failure mode for this compilation stage can originate from delicate dependencies of this software upon other components of this software package, including the lack of host software packages like a set of utilities including texinfo, along with improper setting of environmental variables.

This has been shown by compilation failures due to improper toolchain alignments in Glibc. To counteract a risk due to installing software packages onto a host computer, a strategy of isolation using a separate LFS user, as has been described in Chapters 5 and 6 of a standard manual for Linux From Scratch, brings in a fresh vulnerability for this software component.

1.2.2 The Architectural Limits of Legacy Automation (ALFS/jhalfs)

"Legacy automation projects, especially in the form of jhalfs, which is the official implementation of ALFS, improve computational efficiency but are limited by major architectural and stability issues."

Jhalfs gets commands from the LFS XML sources and runs them with Bash scripts that are controlled by a Makefile framework. This shows that it can compile quickly, full builds take about 30 minutes on high-performance hardware (not including testing). On the other hand, the project's official status makes it less useful as a dependency for production. It is kept up to date as a "rolling release," but there is a "lack of manpower" and it is clearly marked as "not stable."

Using Makefiles for execution and error recovery is still an important, stateful strategy when it comes to architecture. If a package doesn't work, you have to clean up the local environment by hand or with a script before you can try again. This dependence on cleaning state makes it possible for a "dirty" build to happen again, which could lead to outputs that aren't always the same.

1.2.3 Proposed System: Local-First Philosophy and Architecture

The LFS method requires that you immediately separate yourself from the libraries and tools that are already on the host system. The host environment, which can be Ubuntu, Fedora, or a custom distribution, adds environmental variables, library versions, and patches that make sure the build can't be reproduced. To fix this, the first step of cross-compilation (Pass 1) builds a temporary toolchain that is specifically for the new system architecture, which is defined by the triplet $LFS_TGT=x86_64-lfs-linux-gnu.

This process of cross-compiling follows the principle of dependency closure. When you build parts like Binutils Pass 1 and GCC Pass 1, you set them up with flags like --target=$LFS_TGT and, most importantly, --disable-shared. If core components were dynamically linked at this point, they would depend on libraries on the host system, especially the host Glibc. This dependency would cause the resulting toolchain to fail or have symbols that don't match up when run inside the newly built, minimal LFS root.

The Filesystem Hierarchy Standard (FHS) says that you should make a separate /tools directory and link it to the temporary toolchain location. This structure makes it clear where the trust line is. The files in /tools are separate, temporary binaries that are only used to start up the final system. The execution environment makes sure that the new LFS build process uses its own new, separate tools right away after installation by putting the /tools path before the standard host paths in the $PATH variable. This stops host utilities from messing up the build process by mistake. This separation is very important for achieving the level of environmental immutability that the thesis's main goals are looking for.

1.2.4 The Two-Pass Build Rationale: GCC Bootstrapping for Verification

The need for a two-pass toolchain build, which includes re-compiling GCC and Binutils (Pass 2) inside the isolated LFS environment, goes beyond just configuring the system to address serious concerns about its integrity and trust. The host operating system's compiler compiles the binary that was made in Pass 1. The host compiler may have bugs or security features that the resulting Pass 1 binaries could inherit, even though it works.

The next step, Pass 2, takes place in the newly isolated chroot environment. It makes the Pass 1 compiler, which is now running in the LFS environment, recompile itself using the LFS system's new C library (Glibc) and kernel headers. In compiler engineering, this process is called "bootstrapping," and it checks the integrity of the whole toolchain stack. If the Pass 2 compiler binary is the same as the Pass 1 binary, it means that the system libraries (Glibc, headers) are installed correctly and that the compiler can compile itself without any problems. Any difference means that there is an instability or dependency misalignment that needs to be fixed.

This whole mechanism directly meets the automated system's non-functional requirement (NFR) for reproducibility (NFR-R01). The two-pass verification ensures that the LFS kernel and userland will be consistent, no matter what the initial host environment is, which could be anything. This strict verification process turns the LFS build into a quality-assured, self-validating system instead of a series of steps. This directly supports the Cyber Security program's focus on trusted computing bases.

1.3 Isolation Models, Performance, and the PoC Justification

The main difference in the architecture of the thesis is between the stated goal of using high-isolation, short-lived Google Cloud Run containers and the actual implementation, which uses a hybrid WSL/chroot orchestration model. This necessary technical change must be backed up by a thorough look at isolation mechanisms and performance.

1.3.1 Comparative Analysis of Isolation Mechanisms

It is important to understand the architectural limits of chroot isolation in comparison to modern containerization technologies such as Docker. The LFS scripts use chroot mainly to isolate filesystems by limiting processes to a new root directory.

This is enough to keep the build process separate from the host's main binaries, but it doesn't provide much security or operational separation. Processes in the chroot use the host's kernel, PID namespace, and networking stack. Docker and cloud container services, on the other hand, use kernel features like Linux Namespaces and control groups (cgroups) to provide multi-layered isolation. This allows them to virtualize PID, network, and user environments, making them better for running code that isn't trusted or for making environments more stable.

But the best thing about chroot in the LFS context is that it doesn't slow down performance very much. Running computation-heavy compilation tasks inside a chroot gets CPU throughput that is almost the same as the host system's because it uses very few kernel layers or hypervisor functions.

This performance edge is the main reason for the PoC pivot. The thesis objectives require performance (NFR-P01), particularly concerning the compilation phase, which usually takes 10 to 15 hours for a full manual build. Cloud Run, the intended cloud platform, has a strict execution timeout that is usually set to 60 minutes. It is impossible to do the full LFS Chapter 5-8 build, which includes hundreds of configuration and compilation steps for packages like GCC, in this short amount of time.

To validate the core orchestration logic and achieve demonstrably fast compilation times (NFR-P01), the most resource-intensive compilation stage must be executed in a high-fidelity, local environment—the WSL/chroot Proof-of-Concept—prioritizing performance and process continuity over the multi-layered security isolation ideal of full cloud deployment.

1.3.2 Justification of the Hybrid WSL Architecture

The chosen architecture uses Windows Subsystem for Linux (WSL) to get around a big problem for users: they need a specific Linux host environment. Putting the Linux build environment inside WSL makes it a lot easier for people who use Windows or macOS to use Linux. This makes it easier to set up traditional virtualization.

The PowerShell wrappers, like BUILD-LFS-CORRECT.ps1, are the API gateway for the host system. They made a stable, repeatable way to start and control the build process from the host machine. They take commands from users that are at a high level and turn them into the Bash commands that run on Linux. This mixed approach meets the Non-Functional Requirement of Portability (NFR-P02) because it makes the hard build process easier to handle in different user environments. This shows that it fully answers the questions about accessibility that were raised in the introduction.

1.4 Analysis of Current Information Flow

It is very important to look at how information flows in order to figure out the system's architecture and compare the current methods with the proposed automated flow.

1.4.1 Legacy Information Flow (Manual LFS)

The manual LFS build has an information flow that goes in order and is controlled by people. This makes it very easy for mistakes and contamination to happen.

- Steps in the flow: Human input (command) $→$ Host Shell Run Compilation to Mutable State (the LFS directory) and then to Human Output (an unstructured log). → Debugging by Humans

- Vulnerability: The flow of information depends on the host environment's initial state (GCC and binutils versions) and the memory of the person operating it (correct user context, chroot execution, and correct flags). Errors in managing dependencies or environment variables can cause builds to fail, which means that state cleanup must be done by hand and cannot be repeated, and debugging takes longer. The output is a stream of text that isn't organized (standard output), which makes it very hard to monitor automatically.

1.4.2 Proposed Information Flow (Automated and Scripted PoC)

The cloud-native architecture controls and automates the flow of information so that it can be easily repeated and audited. The local PoC scripts use mounting and isolation techniques to make this flow look the same.

- Steps in the Flow: Host Orchestrator (PowerShell/Bash) Separate Environment (WSL/chroot) → Running the script Compilation →Structured Output (BUILDLOG) Managing artifacts (system image and provenance attestation).
- Setting up the environment: init-lfs-env.sh and build-lfs-complete-local.sh set the execution context, especially by setting the cross-compilation triplet and giving the temporary toolchain path priority. This flow makes sure that the right tools are used before installing system binaries, which is very important for the dependency closure principle.
- Isolation Transfer: chroot-and-build.sh takes care of the important job of transferring control, binding the host virtual filesystem, and running the final build script in the isolated environment. This is the main architectural change, which shows that it is possible to switch from cross-compilation to native, self-hosted compilation.
- Output of Information: The script /build-lfs-in-chroot.sh uses structured logging by calling the log() function and tee -a "$BUILDLOG". This changes the information

that comes out of an opaque text stream into a resource that can be used for automated monitoring and finding problems (Structured Observability).

1.4.3 Identification of Key Issues to Resolve

The system directly addresses five major problems in the Linux From Scratch (LFS) community, linking each one to a specific Non-Functional Requirement (NFR) of the thesis.

Table 1

Key Issues to Resolve (1)

Issue to Resolve	NFR Addressed	Problematic Implication
Irreproducible Toolchains	Reproducibility	Small, hard-to-track changes in the host environment cause binary output that isn't always the same, which hurts system trust.
Excessive Compilation Time	Performance	The 10–15 hours it takes to compile makes it hard to access and goes against cloud service time limits, like the 60-minute Google Cloud Run timeout.
Host System Dependency	Usability/Accessibility	The requirement for a specific Linux host environment makes it harder for people who don't use Linux (Windows/macOS) to learn virtualization first, which is not necessary.
Operational Fragility (State)	Maintainability	When a build fails, you have to clean up the state by hand (recovering from a "dirty" build), which can take weeks and raise the risk of operational failure.
Lack of System Integrity	Security/Assurance	The final artifact doesn't have any metadata about the build environment and script execution that can be checked, which makes it hard to trust, which is not acceptable in modern TCB requirements.

Table 1 (1)

(Source: Made by Shubham Bhasker) (1)

1.4.4 Synthesis of Analytical Findings and Technical Justification (Local Project)

This part brings together the theoretical analysis and the main implementation choices made in the local project scripts to meet the need for very technical detail.

Table 2

Analytical Findings and Technical Justification

Analytical Finding/Principle	Synthesis of Script Implementation	Justification
Two-Pass Toolchain Trust	Binutils Pass 1 (build-lfs-complete-local.sh) was set up with --target=$LFS_TGT and --disable-shared. GCC Pass 2 (build-minimal-bootable.sh): The target Glibc libraries were used to recompile inside the final LFS environment.	Makes sure that the whole toolchain is compiled on its own and checks for integrity, which closes dependencies and gets rid of the host compiler's unpredictable effects.
Memory Stability	Bash Configuration: Both the temporary toolchain (implied) and the final Bash package (build-lfs-in-chroot.sh) have the critical flag set: --without-bash-malloc.	Reduces the chance of segmentation faults and crashes during heavy, parallel compilation by making Bash use the stable Glibc memory functions.
Minimalism and Auditability	Python Configuration: The Python build uses the following flag: --without-ensurepip.	Follows the principle of minimal TCB by making it clear that pip and setuptools package management components cannot be combined, which keeps control over the system's package management footprint.
Improving Parallel Performance	All of the core compilation scripts (init-lfs-env.sh, /build-lfs-in-chroot.sh, build-minimal-bootable.sh) set and enforce export for MAKEFLAGS. MAKEFLAGS=-j12.	1Directly addresses I02 (time) and NFR-P01 by maximizing parallel throughput. The chosen factor ($N=12$) is an engineering balance that tries to get the most out of parallelizable compilation time while keeping the cost of managing threads as low as possible (Amdahl's Law).10
LFS Path Isolation	PATH Management: init-lfs-env.sh gives the temporary toolchain priority with export PATH=$LFS_TOOLS/tools/bin:/usr/bin:....	Enforces the FHS structure for LFS Chapter 5, making sure that the new toolchain binaries run over older host binaries that may not be compatible, which sets up the trust boundary that is needed.