# Cloud Computing Labs on Docker and Serverless Framework

*MTP 1 report*
*by*

**Soumik Dutta**
(23m0826)

Supervisor:
**Prof. Kameswari Chebrolu**


Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

13 Oct 2025

# Table of Contents

# List of Figures

# List of Tables

# Abstract

This report presents the design, implementation, and evaluation of a series of hands-on **Cloud Computing Laboratory Exercises** developed to provide practical exposure to containerization and Function-as-a-Service (FaaS) paradigms. The laboratory work is divided into two major components — the **Docker Lab** and the **Serverless Framework (FaaS) Lab** — each emphasizing foundational concepts, workflow automation, and reproducible deployment practices in modern cloud environments.

The **Docker Lab** introduces students to core containerization workflows, orchestration, and production-grade optimization.

The **Serverless Lab** focuses on developing and deploying event-driven applications using the Serverless Framework (v4) on AWS Lambda. Students learn to define infrastructure-as-code (IaC) through `serverless.yml`.

Both lab series were developed, tested, and evaluated on the **cLab/BodhiTree Learning Platform**, which provides an integrated client-server environment for containerized lab execution, version control, and automated assessment.

The report aims to document the technical and pedagogical design of these labs, highlighting key challenges faced during development and the systematic solutions adopted to ensure scalability, reliability, and student accessibility across environments.

**Keywords:** Docker, Serverless Framework, AWS Lambda, Cloud Computing, Containerization, Function-as-a-Service, cLab, BodhiTree, Autograder.

# Chapter 1

# Introduction

## 1.1 Motivation

Cloud computing has transformed the way modern applications are built, deployed, and scaled. With the widespread adoption of microservices, containerization, and serverless computing, the need for hands-on understanding of these technologies has become essential for students and professionals alike. Traditional theoretical approaches often fail to convey the depth of real-world challenges associated with deployment automation, scalability, and reliability.

The **Cloud Computing Laboratory** was designed to bridge this gap by providing a series of structured, hands-on exercises that simulate practical scenarios using industry-standard tools such as Docker and the Serverless Framework. These labs enable learners to experiment, debug, and reason about cloud-native architectures in controlled environments that closely resemble real production systems.

## 1.2 Objectives

The primary objectives of this project are as follows:

- To design and implement modular lab activities demonstrating the use of Docker and the Serverless Framework (FaaS) in cloud application deployment.
- To provide a reproducible and containerized execution environment for all exercises through the **cLab/BodhiTree Learning Platform**.
- To develop automatic evaluation (autograder) scripts ensuring consistency and fairness in assessment.
- To document the complete workflow, including setup, execution, and verification, for each activity in a systematic manner.

- To highlight key technical challenges encountered during lab creation and the approaches adopted to overcome them.

## 1.3 Related Work

In recent years, several educational initiatives have attempted to simplify the teaching of cloud computing concepts through virtualized and container-based lab platforms. Platforms such as **Katacoda**, **Play with Docker**, and **AWS Cloud Labs** provide interactive, browser-based environments that enable students to execute commands and visualize cloud workflows without local setup. While these tools are effective for demonstrations, they often lack the flexibility required for customized evaluation or offline experimentation.

The **BodhiTree Learning Platform**, developed at IIT Bombay, and its enhanced client-side extension **cLab**, represent effort to provide reproducible, containerized lab environments with integrated version control and automated grading.

This project builds upon these efforts by extending the lab framework to encompass comprehensive exercises on **Docker** and **Serverless Framework (FaaS)**.

# Chapter 2

# cLab / BodhiTree Platform

## 2.1 The cLab/BodhiTree Learning Platform

This chapter describes the underlying educational platform used to deliver, manage, and evaluate the hands-on labs developed in this project. The platform operates on a client-server model, consisting of two primary components: **BodhiTree**, a server-side Learning Management System (LMS) that serves as the central hub for instructors to create and manage lab content, and **cLab**, a client-side application for students. This integrated system is designed to facilitate hands-on learning by executing labs in isolated Docker containers directly on the student's machine.

## 2.2 Key Features of the cLab Platform

The cLab platform is built around a set of core features designed to provide a robust and seamless learning experience for both students and instructors:

- **Client-Side Labs:** Students install the Electron-based cLab app, which runs labs as Docker containers on their local machines, ensuring a consistent environment and reducing server load.
- **Auto-Evaluation:** cLab offers instant feedback via local scripts on public test cases, while final submissions are securely graded on the server with hidden test cases.
- **Integrated Submission:** Students submit their work directly through the cLab interface, which automatically packages and uploads files—eliminating the need for manual uploads.
- **Git Version Control:** Each lab environment is initialized as a Git repository, allowing students to commit, track, and manage their progress within the app.

- **IDE-like Features:** cLab includes a built-in code editor and an integrated terminal for direct shell access into the running container, providing a complete and contained development environment.

## 2.3 Instructor Workflow in BodhiTree

Through the BodhiTree web interface, instructors can design and manage complete lab activities by defining several key parameters:

- **Lab Metadata:** Setting the activity title, problem description, total marks, and submission deadlines.
- **Activity Files:** Uploading an archive of necessary files, such as application source code or configuration templates, and marking them as read-only or editable by the student.
- **Execution Environment:** Specifying the Docker image to be used for the lab, which contains all required software, dependencies, and tools.
- **Evaluation Scripts:** Providing both a client-side evaluation script for immediate formative feedback and a server-side script for final summative grading.
- **Resource Constraints:** Defining container limits such as CPU and memory to maintain stability and isolation.

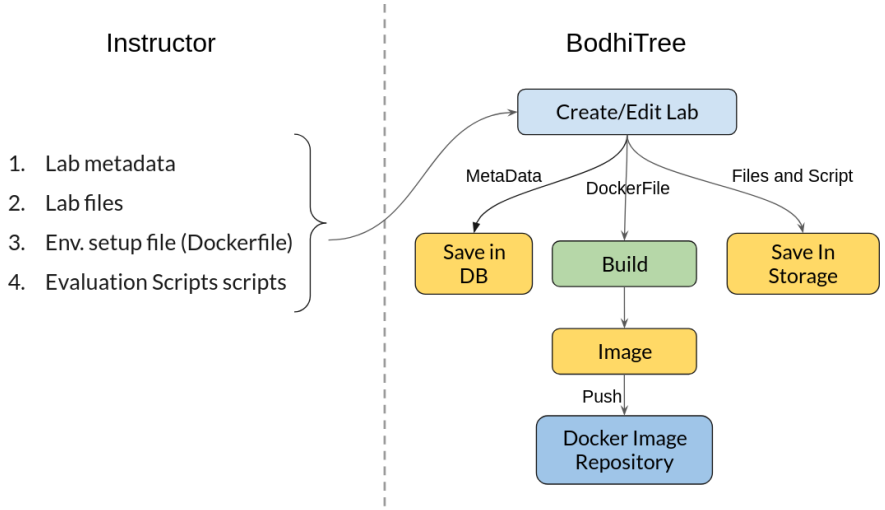


Figure 2.1: Instructor workflow in the BodhiTree system.

## 2.4 Student Workflow in cLab

The student's journey begins with the cLab client application. Upon logging in, the student can view their enrolled courses and select a lab to begin. The cLab application communicates with the backend server to download all necessary resources, including the Docker image path, lab files, and evaluation scripts.

Once initialized, cLab automatically pulls the specified Docker image and runs it locally, mounting two directories:

- `studentdir` – the student's working directory, containing editable files.
- `evalScripts` – a hidden directory containing evaluation and helper scripts.

Students interact with the environment as follows:

1. **Select lab:** Open the cLab client, authenticate, and choose the assigned lab activity from the dashboard.
2. **Develop inside container:** Use the integrated text editor or terminal to modify source code, run applications, and debug within the isolated container.
3. **Local evaluation:** Click the *Evaluate* button to execute `grader.sh` on public test cases and view instant feedback.
4. **Submit for final grading:** Use the *Submit* button to package the workspace and upload it to the BodhiTree server, where hidden test cases are executed for summative evaluation.

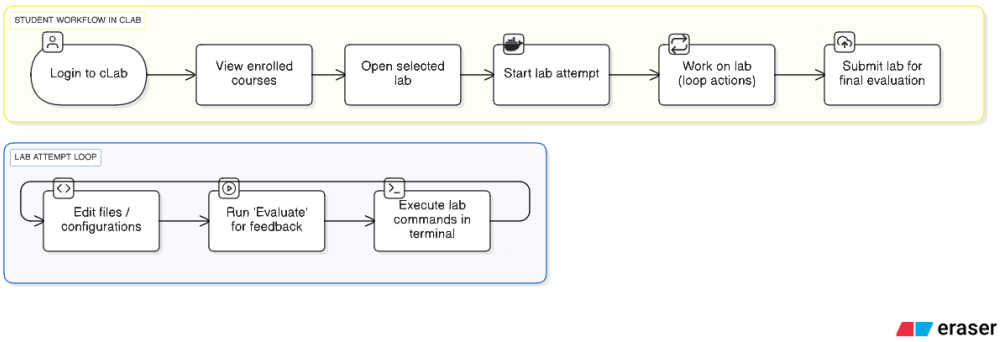


Figure 2.2: Student workflow in the cLab client application.

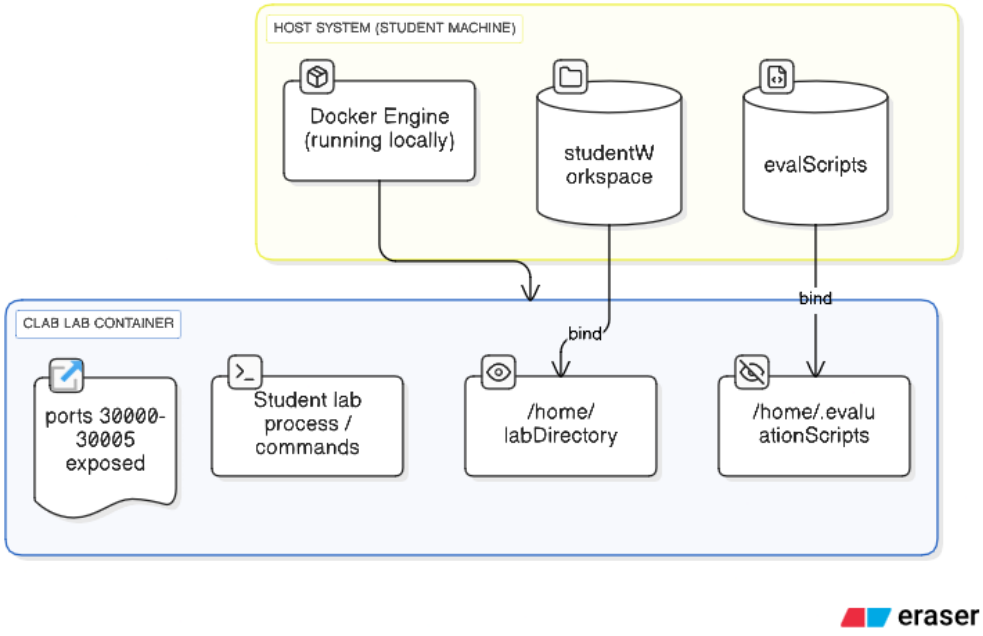


Figure 2.3: Container architecture used for executing cLab labs locally.

## 2.5 Remarks

The cLab/BodhiTree platform provides a reproducible, low-overhead environment for hands-on cloud labs by combining local Docker-based execution with centralized management and evaluation. Its integrated features—version control, IDE-like editing, and two-tiered evaluation—make it especially well suited for structured academic labs where consistency, reproducibility, and real-world skill development are critical. This hybrid approach efficiently balances student autonomy with instructor control, enabling scalable, feedback-driven learning for modern cloud computing education.

# Chapter 3

# All About Docker

## 3.1 About Docker

### 3.1.1 Introduction

Docker has revolutionized modern software development and deployment by enabling lightweight, portable, and reproducible execution environments known as **containers**. Containers encapsulate an application and all its dependencies into a single isolated unit, ensuring that the software behaves identically irrespective of where it runs—on a developer's laptop, a testing server, or a production cluster.

Before understanding Docker's internal architecture, it is essential to recognize how containers differ from the traditional **Virtual Machine (VM)** model.

### 3.1.2 Containers vs. Virtual Machines

Virtual machines emulate complete hardware, running a separate operating system kernel for each instance. In contrast, Docker containers share the same host kernel while isolating user-space processes. This lightweight isolation eliminates the need for redundant OS overhead, drastically reducing resource consumption and startup time.

- **Virtual Machine:** Includes a full guest OS, hypervisor, and virtualized hardware. Each VM runs independently, with its own kernel.
- **Container:** Runs as a process on the host OS using shared kernel primitives such as namespaces and cgroups for isolation and resource control.

This architectural distinction makes containers lightweight, fast to start, and highly scalable compared to traditional VMs.

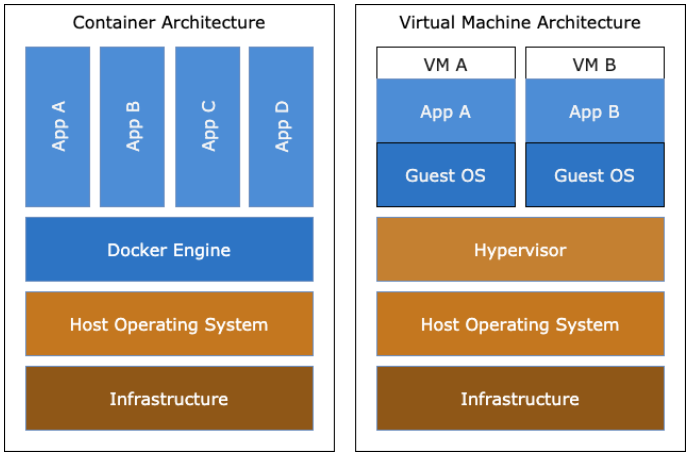


Figure 3.1: Comparison between Virtual Machines and Docker Containers (Adapted from [1]).

### 3.1.3 Docker Architecture

At its core, Docker follows a **client–server model**. The client communicates with the Docker daemon through REST APIs or CLI commands to build, run, and manage containers.

- **Docker Client:** The primary user interface through the `docker` command-line tool. It sends requests to the daemon via REST or UNIX sockets.
- **Docker Daemon (dockerd):** The background service responsible for building, running, and monitoring containers. It handles image management, networking, and volume mounting.
- **Docker Images:** Read-only templates that define a container's file system and dependencies. Each image consists of multiple layers built from Dockerfile instructions.
- **Docker Containers:** Runtime instances of images. Containers include a writable layer over the image layers and execute isolated processes.
- **Docker Registry:** Centralized repository (e.g., Docker Hub) for storing and distributing images.
- **Docker Engine:** The combination of the client, daemon, and APIs that together power Docker's complete functionality.

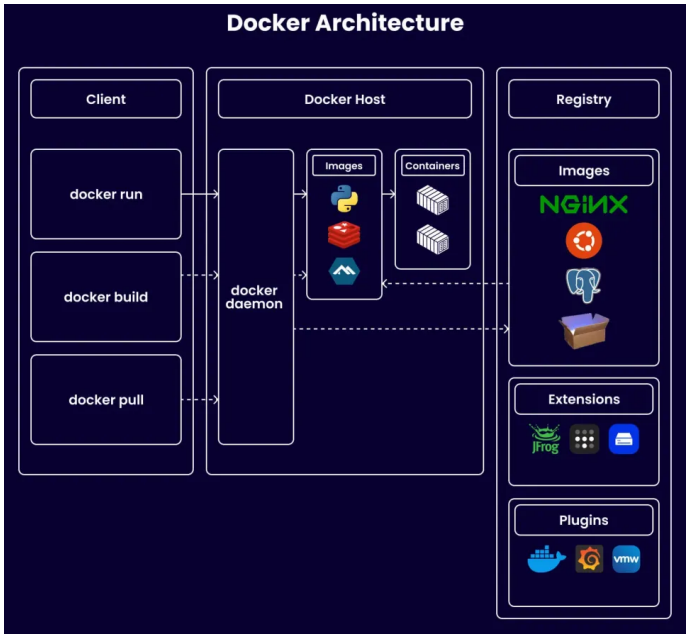


Figure 3.2: High-level architecture of the Docker ecosystem (Adapted from [2]).

### 3.1.4 Internal Implementation and Linux Primitives

Docker achieves containerization not by emulating hardware, but by leveraging built-in **Linux kernel features** for process isolation and resource control. The most critical primitives include:

- **Namespaces:** Provide isolated views of system resources for each container. For example:
  - `PID` namespace isolates process IDs.
  - `NET` namespace provides independent network interfaces.
  - `MNT` namespace ensures isolated mount points and file systems.
  - `UTS` namespace allows custom hostnames per container.
  - `USER` namespace maps user IDs for privilege separation.
- **Control Groups (cgroups):** Limit and monitor system resources such as CPU, memory, and I/O usage per container. They ensure that one container cannot monopolize the host's resources.
- **Union File Systems (OverlayFS):** Enable layered images. Each Dockerfile instruction adds a new read-only layer, while the container adds a writable top layer at runtime.

- **Capabilities and Seccomp:** Provide fine-grained control over system calls and permissions, enhancing container security.

Together, these primitives form the foundation for Docker's efficient and secure isolation mechanism without requiring a hypervisor.

### 3.1.5 Docker Ecosystem Components

The Docker ecosystem extends beyond the core engine to include several additional tools and orchestration layers that enhance usability, scalability, and maintainability:

- **Docker Compose:** Defines and manages multi-container applications using YAML configuration files.
- **Docker Swarm:** Provides built-in container orchestration and clustering.
- **Docker Desktop:** Integrates the Docker Engine with a graphical user interface for simplified local development.
- **Docker Hub / Private Registry:** Acts as a centralized image repository.
- **BuildKit:** An optimized build backend for faster, cache-aware image builds with parallel execution.

### 3.1.6 Why Docker Matters

Docker has become a cornerstone of modern DevOps and cloud-native application design for several reasons:

- **Portability:** Containers run uniformly across environments — from laptops to production clusters.
- **Efficiency:** Low overhead due to kernel sharing and layered file systems.
- **Reproducibility:** Same image yields identical runtime behavior.
- **Scalability:** Integrates seamlessly with orchestration tools like Kubernetes for large-scale deployments.
- **Automation:** Enables continuous integration and delivery (CI/CD) pipelines.

These characteristics make Docker a vital tool for microservices, edge computing, and hybrid cloud deployments.

### 3.1.7 Limitations and Considerations

Despite its strengths, Docker is not without drawbacks. Understanding its limitations is key to making informed architectural choices.

- **Security Risks:** Containers share the host kernel — a compromised container can potentially impact the host.
- **Resource Isolation Limitations:** While cgroups help, isolation is weaker than hypervisors under heavy loads.
- **Persistent Storage Management:** Requires explicit volume and bind mount management.
- **Networking Complexity:** Multi-network deployments can be non-trivial.
- **Compatibility on Non-Linux Systems:** Docker Desktop relies on lightweight VMs for Windows and macOS, reducing some performance gains.

### 3.1.8 Conclusion

Docker represents a paradigm shift in how software is developed, packaged, and deployed. By combining kernel-level isolation, layered file systems, and a simple declarative interface, it has made reproducible and scalable deployments accessible to developers at every level. Understanding its architecture, internal mechanisms, and ecosystem is crucial for designing reliable and maintainable cloud-native systems.

# Chapter 4

# Docker Lab

## 4.1 Overview

This chapter documents the Docker Lab series, comprising Activities 0 through 7, designed to build a foundational understanding of containerization, image management, networking, orchestration, and production-grade optimization. Each activity introduces one or more core Docker concepts through hands-on experimentation, structured workflows, and autograded verification.

The exercises were conducted on an AWS EC2 instance running Ubuntu 24.04 LTS, with Docker Engine (25.x) installed via the official Docker repositories. All tasks are delivered and executed via the **cLab** platform, ensuring reproducibility and isolation of each student's workspace.

Below is a brief summary of each activity in this lab:

- **Activity 0: Environment Setup** — Provision an EC2 instance, install Docker, and verify the engine's correctness via simple test containers.
- **Activity 1: Core Docker Operations** — Pull images, run containers, inspect metadata, mount volumes, and manage container lifecycle.
- **Activity 2: Image Lifecycle & Persistence** — Modify running containers, commit images, export/import images, and explore registry operations.
- **Activity 3: Custom Image Build & Publish** — Create Dockerfiles for sample applications, build, tag, and publish custom images to a registry.
- **Activity 4: Docker Networking** — Use bridge and user networks to enable inter-container communication and explore container DNS resolution.

- **Activity 5: Docker Compose — Multi-tier Stack** — Deploy a three-tier application using Docker Compose, connecting frontend, backend, and database services.
- **Activity 6: Production-Ready Dockerfiles** — Optimize Dockerfiles using multi-stage builds, health checks, non-root users, layer caching, and security best practices.
- **Activity 7: Advanced Topics — Daemon, Rootless, Docker Init** — Explore Docker's daemon internals, rootless mode, and scaffold new projects with `docker init`.

For the full problem statement and lab specification, refer to **Docker Lab Problem Statement Document**.

Through this sequence of experiments, students gradually progress from basic container invocation to robust Docker-based deployment strategies, enabling them to understand and apply containerization principles in real-world settings.

## 4.2 Activity 0: Environment Setup

**Objective:** This activity establishes the foundational test environment required for all subsequent Docker Lab exercises. Students are guided to launch an AWS EC2 instance (Ubuntu 24.04 LTS) and install the Docker Engine to ensure a consistent, cloud-based execution platform.

**Overview:** As the first activity in the Docker Lab series, this task primarily focuses on preparing the learning infrastructure rather than application-level experimentation. The main goal was to design an environment setup process that is clear, error-resistant, and reproducible across different students' systems.

To achieve this, every step of the AWS EC2 setup—right from account login and key-pair generation to instance launch and Docker verification—was documented in a highly visual and guided format. Each configuration step in the AWS Management Console was accompanied by detailed screenshots and descriptive annotations to remove ambiguity for beginners. This visual approach proved extremely effective in reducing student confusion, particularly for those new to AWS or Linux cloud environments.

Beyond basic installation, the activity also ensures that learners configure network rules, permissions, and metadata options correctly. This guarantees that subsequent lab tasks can execute smoothly without environment-related failures.

**Instructor-side Implementation:** From the instructor's perspective, this activity required configuring and publishing the setup on the **BodhiTree/cLab platform**. Since

this was the initial exercise, I had to become familiar with the BodhiTree instructor interface—uploading resource archives, setting evaluation scripts, defining metadata, and linking autograder logic. I also explored how cLab integrates client-side and server-side evaluation mechanisms to enable seamless local testing and server-grade verification.

**Autograder Design:** A robust and modular autograder was implemented to automatically validate environment correctness. The grader connects to the student's EC2 instance via SSH using the submitted key and verifies each component step-by-step:

- Instance accessibility and SSH key permissions.
- Docker Engine installation and service status.
- Successful execution of the `docker run hello-world` test container.

If any component fails, the autograder reports granular diagnostic feedback so that students can quickly correct configuration issues. This modular structure was crucial in ensuring reliability across diverse student setups and made debugging straightforward during evaluation.

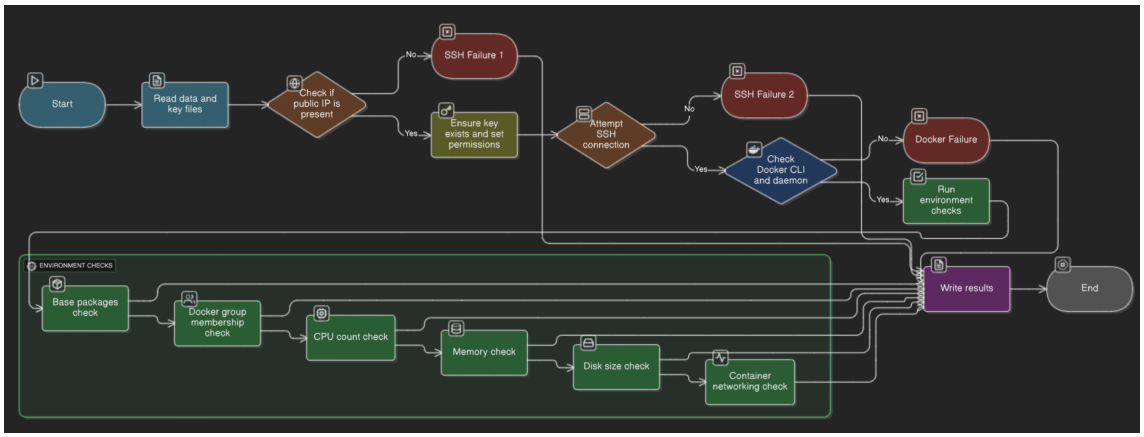


Figure 4.1: Flow of autograder test cases for Activity 0.

**Challenges Faced and Solutions:**

- **Challenge 1 – Familiarity with BodhiTree Instructor Tools:** As this was the first activity to be created, I had to thoroughly learn the BodhiTree instructor interface—activity publishing, file uploads, evaluation modes, and grading pipelines.
  **Solution:** Conducted trial deployments and dry runs to understand the instructor dashboard and ensure stable evaluation integration with cLab.
- **Challenge 2 – Designing a Modular Autograder:** The environment setup required multiple independent checks (EC2 reachability, Docker installation, permissions).

**Solution:** Implemented a modular structure in the grader where each check runs as an independent function and contributes to the cumulative score, enhancing both clarity and maintainability.

- **Challenge 3 – Simplifying Student Experience:** Initial feedback showed that students struggled with AWS console navigation.
  **Solution:** Added labeled screenshots for every EC2 configuration step, clarifying key concepts such as metadata versioning, storage configuration, and SSH permissions.

**Remarks:** This activity serves as the test-bed initialization step of the entire Docker Lab sequence. By the end of this stage, every student has a functional Docker-enabled EC2 instance connected through cLab, guaranteeing a consistent and reproducible base for the remaining activities. The emphasis on clarity, guided visuals, and resilient evaluation makes this activity a unique and high-impact starting point for the course.

For the complete problem statement and detailed step-by-step guide, refer to the official **Activity 0 Problem Statement Document**.

## 4.3 Activity 1: Core Docker Operations

**Objective:** This activity provides hands-on exposure to Docker's fundamental command-line operations, focusing on container lifecycle management, process isolation, and image inspection. Students are expected to run, inspect, and manage containers interactively while learning secure and efficient container usage practices.

**Overview:** Building on the environment prepared in Activity 0, this task introduces the essential Docker operations required for everyday container management. The activity is divided into four logical segments:

- **Task 1 – Basic Containers:** Running and managing both short-lived and long-running containers, such as `hello-world`, `nginx`, and `ubuntu`. Students learn container naming, labeling, port mapping, and interactive shell usage.

- **Task 2 – Container Lifecycle and Inspection:** Understanding container metadata through commands like `docker ps`, `docker inspect`, and `docker stats`. Students extract specific fields such as image size, creation time, and network information using Go templating.

- **Task 3 – Non-root Execution:** Exploring container security by running workloads as non-root users using the `-user` flag, verifying privileges through `whoami`, `id`, and file access checks.

- **Task 4 – Image and System Housekeeping:** Learning cleanup and optimization commands such as `docker system prune`, `docker image prune`, and `docker container prune` to maintain a lean and efficient Docker environment.

Each of these segments contributes to a progressive understanding of how Docker containers function as isolated user-space processes with manageable resources, and how effective CLI usage improves productivity and reproducibility.

**Implementation Details:** To design this activity, multiple example scripts and reference containers were prepared to demonstrate Docker's lifecycle operations in real time. I structured the tasks to simulate realistic scenarios—web services, background processes, and system-level inspections—to give students a deeper sense of container orchestration fundamentals.

For security awareness, an explicit section on non-root containers was added. This part required extensive testing, as some official images (like `ubuntu:latest`) lack default non-root users. Students were therefore instructed to run with explicit user IDs and verify permission boundaries both inside and outside the container.

The activity also emphasizes data persistence and cleanliness. By including container pruning and disk usage measurement, learners were introduced to system hygiene practices—an aspect often overlooked in beginner-level Docker tutorials.

**Testing and Evaluation Strategy:** The autograder for this activity was designed to be stateful and comprehensive, verifying each stage of container lifecycle correctness. It automatically checks:

- Presence of expected containers and labels (`lab1-hello`, `lab1-nginx`, `lab1-ubuntu`).
- Functional verification of HTTP service on port 8080.
- Proper execution of metadata extraction commands and correctness of entries in `ans.json`.
- Validation that containers running under UID 1000 do not possess root privileges.
- Post-cleanup verification ensuring no dangling images remain.

The grader's design follows a modular structure similar to Activity 0 but expanded with multiple subtests to handle dynamic container states. It connects to the student's EC2 instance, executes targeted verification commands, and reports detailed feedback for any missing or misconfigured step.

**Challenges Faced and Solutions:**

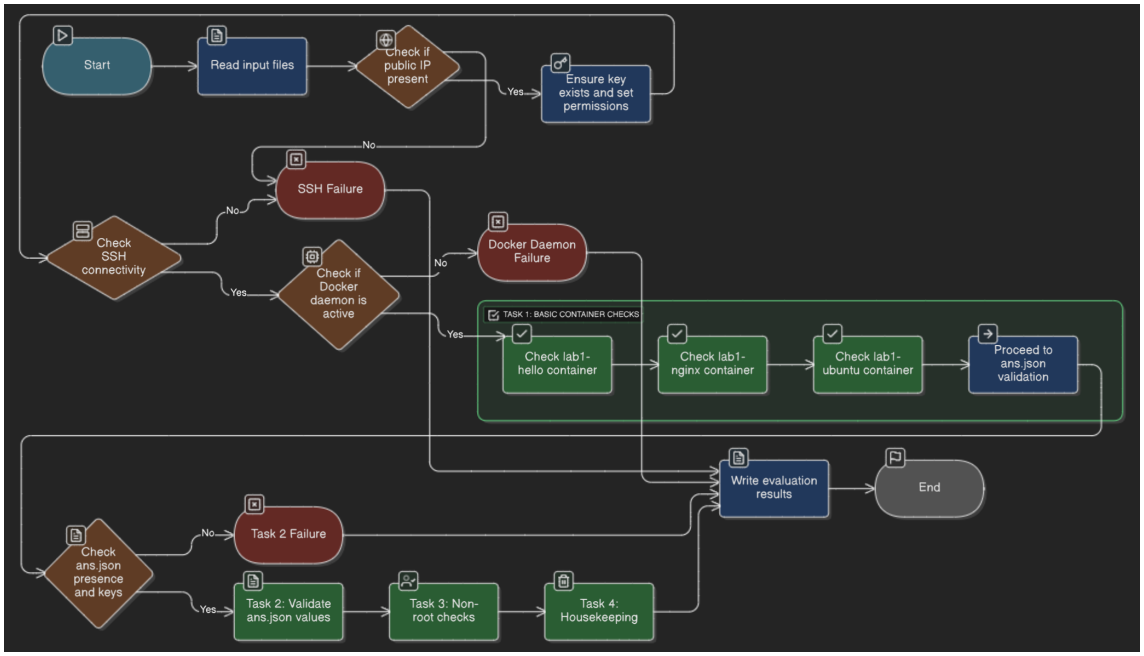


Figure 4.2: Flow of autograder verification for Activity 1.

- **Challenge 1 – Testing Dynamic Container States:** Since the grader interacts with running containers that may exit or restart unpredictably, synchronization issues initially occurred.
  **Solution:** Introduced retry logic and a validation loop with `docker inspect -format` checks to ensure containers reached a stable running or exited state before grading.

- **Challenge 2 – Security Testing for Non-root Containers:** Verifying non-root privileges required differentiating between system-level UID and in-container UID.
  **Solution:** Used a combination of `docker exec id` output parsing and host-level `docker top` comparisons to confirm privilege isolation.

- **Challenge 3 – Ensuring Robust Cleanup Validation:** Early versions of the autograder occasionally misidentified ‘<none>‘ images due to caching.
  **Solution:** Added explicit filters for `dangling=true` and validated against total image count before and after cleanup.

- **Challenge 4 – Balancing Complexity and Clarity for Learners:** As this activity involves numerous CLI commands, students initially found the documentation overwhelming.
  **Solution:** Condensed repetitive steps and added structured command summaries for clarity, while keeping essential learning points intact.

**Remarks:** This activity served as a practical foundation for students to internalize core Docker operations and container security concepts. By the end of this lab, learners gained the ability to confidently manipulate containers, analyze metadata, manage system hygiene, and understand privilege boundaries—skills directly applicable to real-world DevOps environments.

For the complete problem statement and reference command list, refer to the official **Activity 1 Problem Statement Document**.

## 4.4 Activity 2: Docker Image Lifecycle & Persistence

**Objective:** This activity introduces advanced Docker concepts—image lifecycle management, runtime controls, and data persistence. Students learn to inspect and export images, commit container changes, apply health checks and resource limits, and implement persistence through Docker volumes, bind mounts, and tmpfs. The activity concludes with a MySQL persistence case study to illustrate durable storage across container restarts.

**Overview:** Building upon the fundamentals from Activity 1, this exercise focuses on how container images evolve, how runtime resources are controlled, and how persistent storage is managed within Docker's architecture. Students practice practical DevOps operations such as:

- Inspecting image metadata and history to understand layering and size impact.
- Using `docker save/load` for image transfer and backup.
- Capturing container state via `docker commit`.
- Applying `HEALTHCHECK`, `-cpus`, `-memory`, and `-ulimit` for runtime governance.
- Comparing ephemeral, volume-based, tmpfs, and bind-mount persistence using MySQL.

**Implementation Details:** To make the lab both rigorous and learner-friendly, I designed the activity around small, verifiable experiments that showcase each concept independently. The MySQL persistence test was structured to clearly demonstrate the difference between ephemeral container filesystems and Docker-managed volumes—ensuring students could see their data survive container removal. Additional helper scripts were created to automate extraction of key fields (image IDs, architecture, largest layer size, health status, and mount paths) used later by the autograder.

**Testing and Autograder Strategy:** The autograder verifies correctness through a modular test suite covering:

- **Image operations:** Validates pulled images, metadata fields, largest-layer size, saved tar existence, and committed-image digest.
- **Runtime controls:** Confirms healthcheck status, CPU/memory limits, and ulimit values through `docker inspect` output.
- **Persistence:** Runs SQL queries inside MySQL containers to compare row counts before and after recreation for volume, tmpfs, and bind-mount cases.
- **Cleanup integrity:** Ensures no residual containers or unused images remain after completion.

The grading logic incorporates safe retry intervals for slow-starting services (e.g., MySQL) and reports detailed per-test diagnostics, improving reliability across student EC2 setups.

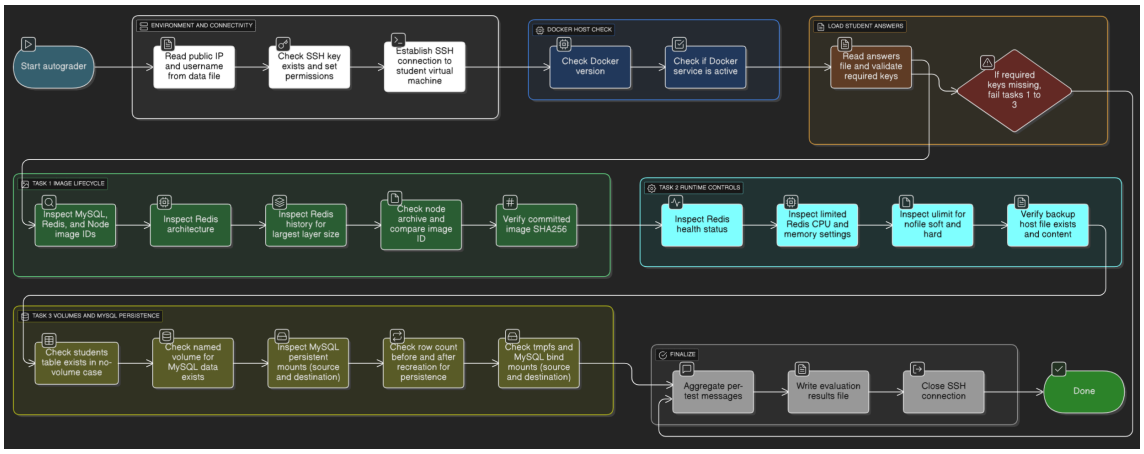


Figure 4.3: Flow of autograder verification for Activity 2.

**Challenges Faced and Solutions:**

- **Managing service startup delays:** MySQL initialization often exceeded default grading timeouts.
  *Solution:* Added a retry mechanism using container health status as readiness indicator.
- **Disk usage during image save/load:** Large image archives filled EC2 volumes.
  *Solution:* Recommended compression and ensured temporary tar files were pruned post-evaluation.
- **Parsing image history output:** Formatting differences caused inaccurate layer-size detection.

*Solution:* Used non-truncated JSON parsing with `-no-trunc` to extract exact values.

- **Ensuring consistent mount paths:** Bind-mount paths varied across instances. *Solution:* Standardized workspace under `/home/ubuntu/docker_lab` for all persistence tests.

**Remarks:** Activity 2 bridges the gap between transient container use and production-ready image and storage management. Through the combination of image lifecycle commands, runtime constraints, and persistence mechanisms, students gained a holistic understanding of Docker's operational model—how containers can be controlled, backed up, and made stateful in real deployments.

For the full problem statement and command reference, refer to the official **Activity 2 Problem Statement Document**.

## 4.5 Activity 3: Building and Publishing Custom Docker Images

**Objective:** Containerize a sample Flask application by authoring an efficient Dockerfile, build and test the image locally, and publish the final image to Docker Hub. The activity emphasizes understanding each Dockerfile instruction (FROM, RUN, COPY/ADD, ENV, WORKDIR, USER, EXPOSE, HEALTHCHECK, ENTRYPOINT/CMD), image layering, caching, and best practices for production-ready images.

**Overview:** Activity 3 guides students from a manual, environment-based workflow to a reproducible, image-driven deployment. The lab uses a small Flask REST API (endpoints: `/api/products`, `/api/products/<id>`, `/health`, `/logo`) which students containerize. Students progressively construct the Dockerfile in focused steps (base image, environment vars, system packages, application copy, dependency install, non-root user, metadata, runtime configuration, and health checks), then build, run, validate, retag, and push the image to Docker Hub.

**Implementation Details:**

- **Instructor-provided Flask application:** I developed the reference Flask service exposing the APIs used in the lab. This ensures a consistent application behavior across all student environments and simplifies grader validation.

- **Stepwise pedagogical Dockerfile**: split the build into small incremental steps (tags like `lab3/stepX`) to teach cache behavior and layer optimization.

- **‘.dockerignore‘ template**: included a curated ‘.dockerignore‘ to prevent leaking large or sensitive files into the build context.
- **‘system-clean-init.sh‘ helper**: a safe, idempotent script to reset the EC2 workspace, deploy the application archive, and ensure reproducible student environments.
- **Non-root runtime and permissions**: enforced a fixed UID/GID user, appropriate chown steps, and EXPOSE/HEALTHCHECK configuration for robust runtime behavior.
- **Push workflow guidance**: documented retagging, Docker Hub authentication, push verification, and digest extraction for reproducible distribution.

**Testing and Autograder Strategy:**

- **Image build tests:** confirm successful build for incremental tags, check presence of expected files (`src/app.py`, `requirements.txt`, logo), and validate reported versions (Python, pip, Flask).
- **Runtime tests:** run container, assert HTTP responses for `/api/products` and `/health` (HTTP 200), and verify logo endpoint serves the image.
- **Security checks:** validate process runs as non-root user (UID/GID), and permissions on application directory.
- **Production checks:** inspect labels, exposed port value, health status via `docker inspect`, and ensure healthcheck becomes `healthy`.
- **Distribution checks:** verify retagging, successful `docker push`, and that the pulled image digest matches the pushed digest.
- The grader is modular—small independent checks with retry logic for service startup—and writes explicit diagnostics to help students resolve failures.

**Challenges faced and solutions:**

- **Cache invalidation due to broad COPY**: initial Dockerfile copied the full context early, invalidating dependency cache.
  *Solution:* advised copying `requirements.txt` first (or structured steps) and using ‘.dockerignore‘ to minimize rebuilds.

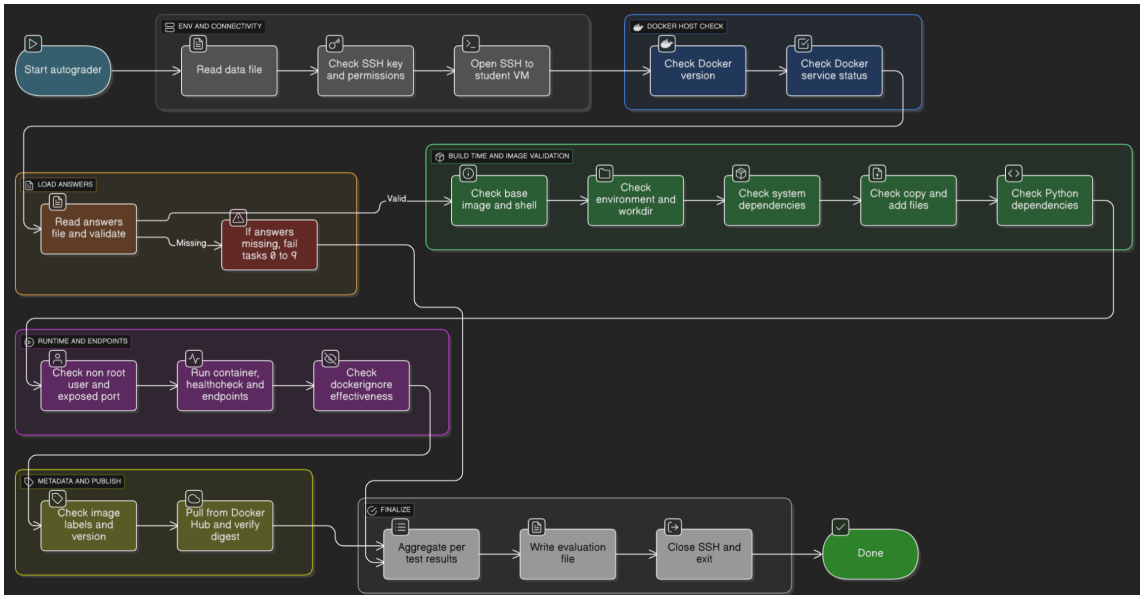


Figure 4.4: Flow of autograder verification for Activity 3.

- **Permission errors for non-root user**: some packages required root-owned directories.
  *Solution:* performed installs as root during build, then chowned application files and switched to non-root user for runtime.

- **Unreliable healthcheck timing**: short healthcheck periods caused transient failures.
  *Solution:* set sensible `-start-period`, `-interval`, and added grader retry logic.

- **Student friction in push workflow**: credential issues and tagging mistakes were common.
  *Solution:* included explicit retagging steps, sample `docker login` guidance, and example commands to verify digests.

**Remarks:** Activity 3 consolidates Dockerfile best practices and the end-to-end image lifecycle—from local build to public distribution. The incremental build approach, helper init script, instructor-provided Flask app, and explicit runtime checks reduce student setup friction and make the lab robust and reproducible.

For the full student-facing problem statement and step-by-step instructions, refer to the official **Activity 3 Problem Statement Document**.

## 4.6 Activity 4: Networking in Docker (without Compose)

**Objective:** Understand Docker's networking model and manually compose multi-container topologies without Docker Compose. Students compare default bridge, user-defined bridge, and special modes (`none`, `host`), perform service integration (MySQL + phpMyAdmin) on a user-defined network, and apply inspection and security best-practices for container networking.

**Overview:** This activity focuses on how containers communicate with each other and with the host. Rather than using Compose, students manually create networks, attach containers, publish host ports, and inspect DNS and routing behavior. Key learning outcomes include service discovery via embedded DNS on user-defined bridges, port-publishing patterns for host access, the trade-offs of `-network host` and `-network none`, and practical troubleshooting techniques (ping/curl/getent/docker inspect).

**Implementation Details:**

- **Hands-on services:** used `nginx` (web), `alpine` (lightweight client), `mysql` (database), and `phpmyadmin` (DB UI) to demonstrate real networking scenarios.
- **Default vs user-defined bridge:** demonstrated limitations of the default `bridge` (no DNS) and benefits of a user-defined bridge (`lab4-net`) which provides name-based resolution and service discovery.
- **Manual wiring:** ran containers with explicit `-network` flags, connected/disconnected networks with `docker network connect/disconnect`, and showed multi-homing via attaching a container to multiple networks.
- **Port publishing patterns:** illustrated deterministic mapping (`-p 8080:80`) and dynamic mapping (`-p :80`), and how to discover assigned host ports via `docker port` / `docker ps`.
- **Special modes:** exercised `-network none` (complete isolation) and `-network host` (shared host stack) with explicit warnings about security and port conflicts.
- **Inspection tooling:** used `docker network inspect`, `docker inspect`, `getent hosts`, and `docker events` to reveal subnets, gateways, container mappings, and runtime network changes.

**Testing and Autograder Strategy:**

- **Default bridge checks:** verify that containers on the default bridge communicate by IP but name-based resolution fails.
- **User-defined bridge checks:** ensure `lab4-net` exists, containers attached to it resolve each other by name (checked via `getent hosts` and successful `curl http://<name>`), and inspect reports correct subnet/gateway and container IPs.
- **Integration checks:** validate MySQL initialization (logs/readiness probe), confirm phpMyAdmin can connect to `mysql-db` using container name, and assert phpMyAdmin is reachable from the host on the published port (e.g., 8080).
- **Special modes checks:** confirm `none-test` shows no network interfaces/reachability and `host-test` reports `NetworkMode: "host"` and that services are reachable on the host without `-p`.
- **Robustness:** grader uses polite retries for services that take time to initialize and records concrete diagnostics (container IDs, IPs, host ports, getent outputs) into `ans.json`.

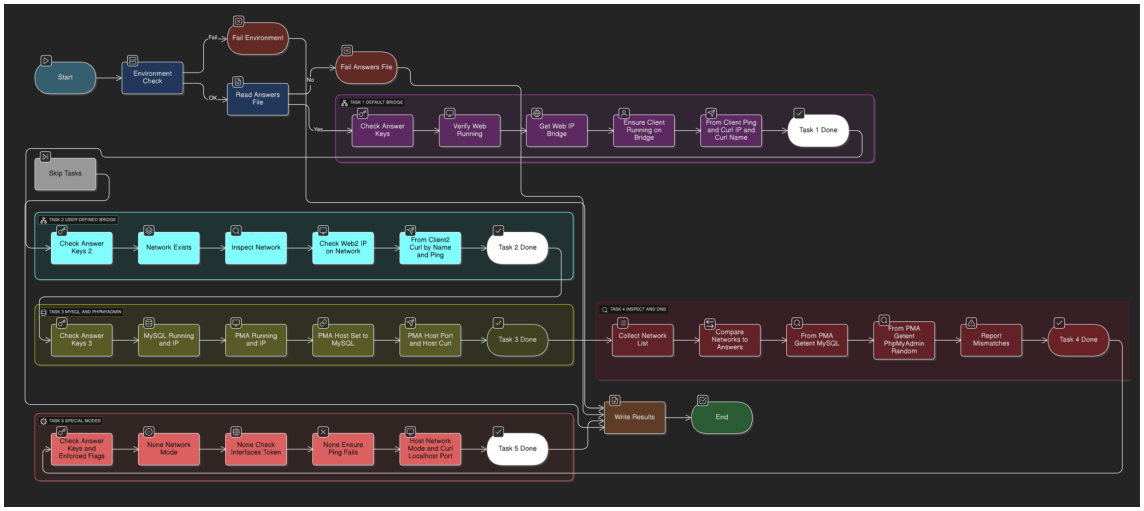


Figure 4.5: Flow of autograder verification for Activity 4.

**Challenges faced and solutions:**

- **Service readiness timing (MySQL):** phpMyAdmin often attempted connection before MySQL was ready.
  *Solution:* used readiness polling (logs/mysqladmin) and added grader retry/backoff to avoid false negatives.

- **Host-mode risks and port conflicts:** accidental binding to critical host ports caused failures during tests.
  *Solution:* documented safe test ports (e.g., 18080), added warnings, and ensured grader avoids destructive host-port assertions.

**Remarks:** Activity 4 gives learners a practical understanding of Docker networking fundamentals and the manual steps Compose automates. Emphasis on inspection commands and security best-practices helps students design safer multi-container deployments and troubleshoot connectivity issues effectively.

For the full student-facing problem statement and step-by-step instructions, refer to the official **Activity 4 Problem Statement Document**.

## 4.7 Activity 5: Multi-Container Applications with Docker Compose

**Objective:** Learn how Docker Compose simplifies orchestration of multi-container applications by defining services, networks, volumes, secrets, healthchecks, restart policies, and basic scaling. Containerize and orchestrate a production-like 3-tier demo application (React frontend, Spring Boot backend, PostgreSQL database) and produce a production-ready `docker-compose.yml`.

**Overview:** This activity walks students from manual `docker build`/`docker run` steps to a fully specified Compose stack that brings up the entire 3-tier application with a single command. The app used in the lab is a complete, containerizable 3-tier demo. The exercise covers networks, environment variables, secrets, volumes/persistence, healthchecks & startup sequencing, restart policies, scaling, logging, and override files (dev vs prod).

**Implementation Details:**

- **Instructor-provided 3-tier app:** I developed the React frontend and Spring Boot backend (with DB schema + seed scripts) so students get a consistent, realistic target to containerize and orchestrate.

- **Progressive workflow:** students first run services manually (Postgres, backend, frontend) to understand ports and logs, then recreate the same topology using Compose (single-service → networks → secrets → build from source → healthchecks → restart policies → scaling).

- **Compose primitives emphasized:** `services`, `networks`, `volumes`, `secrets`, `environment`/`.env`, `build:`, `depends_on` (with `condition:`

`service_healthy`), `healthcheck`, `restart`, logging driver options, and override files (`docker-compose.override.yml` / prod files).

- **Secrets handling:** demonstrated Compose-native secrets via files mounted at `/run/secrets`, and a lightweight wrapper approach for services (e.g., Postgres) that expect password in env.
- **Volumes:** Postgres data and backend logs persisted to named volumes to demonstrate durability across container recreation.
- **Build vs Image mode:** students build from local sources during development (using `build:`) and switch to prebuilt images (e.g., `soumik13/...:v2`) when preparing the final submission.

**Testing and Autograder Strategy:**

- **Service availability:** autograder verifies each service health by checking Postgres readiness (`pg_isready`), backend health endpoint (`/api/health`), and frontend root (`/`).
- **Network & discovery:** confirm Compose network `course-net` exists and services resolve by service name from within the network (e.g., backend resolves `course-db`).
- **Persistence checks:** validate data persists in the named DB volume after container recreation and backend logs are present in the backend logs volume.
- **Secrets & environment:** ensure `course_db_password` secret file is mounted read-only and not present in `.env`.
- **Port exposure & security:** assert only required host ports (3000, 8080) are published; course-db must not expose a host port.
- **Final composition validation:** autograder executes `docker compose up -build -d`, waits for services to become healthy, tests endpoints from host, and records digests/logs for reproducibility.
- The grader is modular and tolerant to startup delays: healthchecks are polled with backoff and clear diagnostic logs are captured on failure.

**Challenges faced and solutions:**

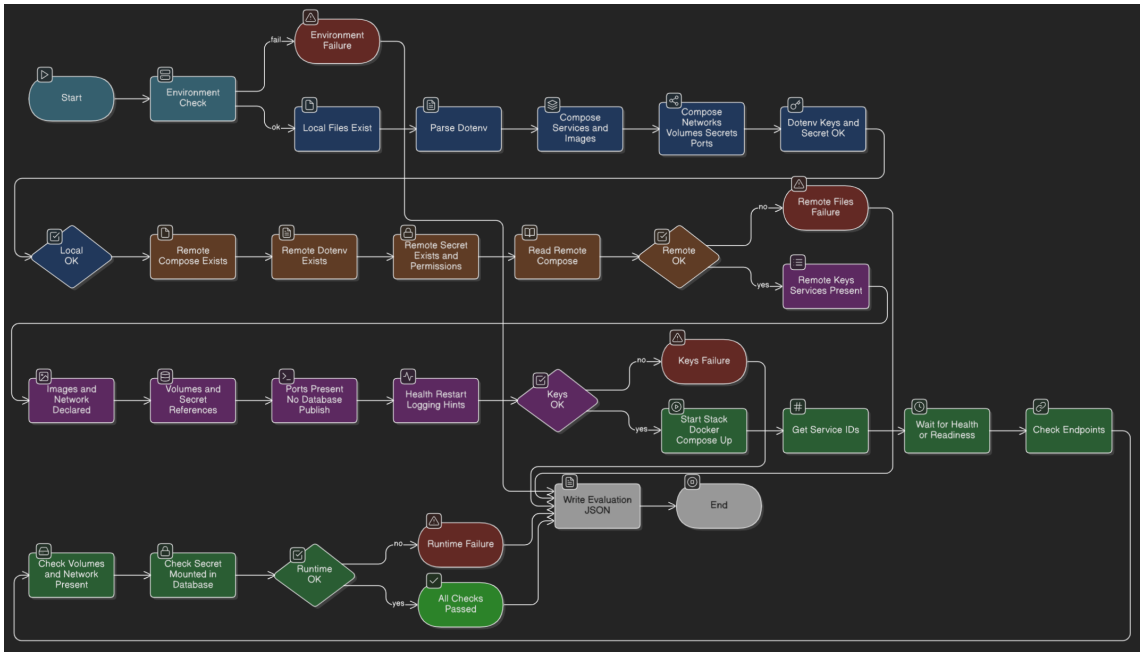


Figure 4.6: Flow of autograder verification for Activity 5.

- **Service startup races (DB → backend → frontend):** initial runs failed when backend attempted migrations before Postgres readiness.
  *Solution:* added robust `healthcheck` definitions and used `depends_on: condition: service_healthy` so Compose respects readiness before starting dependents.

- **Secret propagation to services expecting env vars:** official images (Postgres) expect `POSTGRES_PASSWORD` as an environment variable.
  *Solution:* implemented a small wrapper entrypoint that reads the mounted secret from `/run/secrets` and exports it before starting the container; this keeps passwords out of `.env` and image layers.

- **Local dev vs production parity:** developers needed fast iteration (mounts) but production requires images and no mounts.
  *Solution:* provided `docker-compose.override.yml` for dev (with `build:` and source mounts) and a base `docker-compose.yml` geared toward production (prebuilt images + logging rotation + restart policies).

- **Scaling backend complexity:** scaling revealed name/port issues (only one host port can be published).
  *Solution:* documented scaling caveats and recommended a reverse proxy (NGINX/-

Traefik) for external load balancing; in the autograder we avoid tests that require published ports for scaled replicas.

**Remarks:** Activity 5 synthesizes prior lessons (networking, images, healthchecks, secrets) into a single, reproducible Compose workflow. The instructor-provided 3-tier app (frontend + backend implemented) ensures consistent behavior across student environments and serves as a realistic target for teaching Compose best practices and production-oriented configuration.

For the full student-facing problem statement and step-by-step instructions, refer to the official **Activity 5 Problem Statement Document**.

## 4.8 Activity 6: Make Dockerfiles Production Ready

**Objective:** Learn how to build smaller, faster, and more secure container images by applying production-ready Dockerfile patterns. The activity consists of a detailed, worked Go example and a parallel Node exercise where students optimize naive Dockerfiles into efficient, production-grade builds using multi-stage builds, cache mounts, minimal base images, non-root execution, and security scanning.

**Overview:** This activity transitions students from functional container builds to production-grade image design. The lab demonstrates progressive optimization on a Go service (multi-stage builds, cache mounts, stripped static binaries, healthchecks, metadata, non-root user, and vulnerability scans). Students then apply the same principles to a Node application to create a lean, reproducible runtime image. Both tasks emphasize build efficiency, image minimality, and runtime security verification.

**Implementation Details:**

- **Instructor-built reference apps:** I implemented both the Go and Node applications to ensure controlled and testable behavior.
  - The Go app is a statically linked web service exposing `/`, `/health`, and `/metrics`, with an internal `-healthcheck` flag for scratch-based health validation.
  - The Node app is a minimal Express service exposing `/` and `/health`, built to illustrate caching, dependency layering, and least-privilege principles.
- **Progressive optimization workflow:** For the Go example, I authored incremental Dockerfiles (`step1`–`step4`, `semifinal`) showing optimization stages: cache ordering, BuildKit mounts, static builds, and multi-stage runtime extraction (scratch/distroless).

- **Final Dockerfile.prod:** Combined all optimizations — static stripped binary, metadata labels, non-root user (`USER 10001`), `HEALTHCHECK`, and exposed port — producing a fully minimal image under 10 MB.
- **Node scaffold:** Provided a guided `Dockerfile.prod` template where students apply the same principles (multi-stage or slim runtime, cache mounts, reproducible installs via `npm ci`, non-root execution, and HEALTHCHECK).
- **System setup automation:** The `system-clean-init.sh` script resets the EC2 environment and transfers lab archives for reproducible starting states.
- **Measurement and comparison:** Students measure image sizes, build times (first vs cached), layer counts, startup time, and Trivy scan results to quantify optimization effects.

**Testing and Autograder Strategy:**

- **Image build validation:** autograder builds both students developed naive and optimized images using BuildKit, verifying successful builds and checking for expected files and metadata.
- **Runtime checks:** containers for both images are run on distinct ports; the grader tests HTTP responses from `/health` and verifies correct service startup.
- **User & privilege verification:** image metadata is inspected (`Config.User`) to confirm non-root execution (UID 10001).
- **Performance & optimization metrics:** compares image size, number of layers, and build times; optimized image must show measurable improvements.
- **Security scanning:** runs containerized `trivy` scans to confirm reduced vulnerabilities in the optimized image compared to the naive baseline.
- **Modular checks:** all steps (build, run, inspect, scan) are modular, retried on transient errors, and produce clear diagnostics for student debugging.

**Challenges faced and solutions:**

- **Scratch image health validation:** no shell or `curl` available in minimal images. *Solution:* extended the Go binary with an internal `-healthcheck` flag and added an alternative distroless variant for fallback testing.

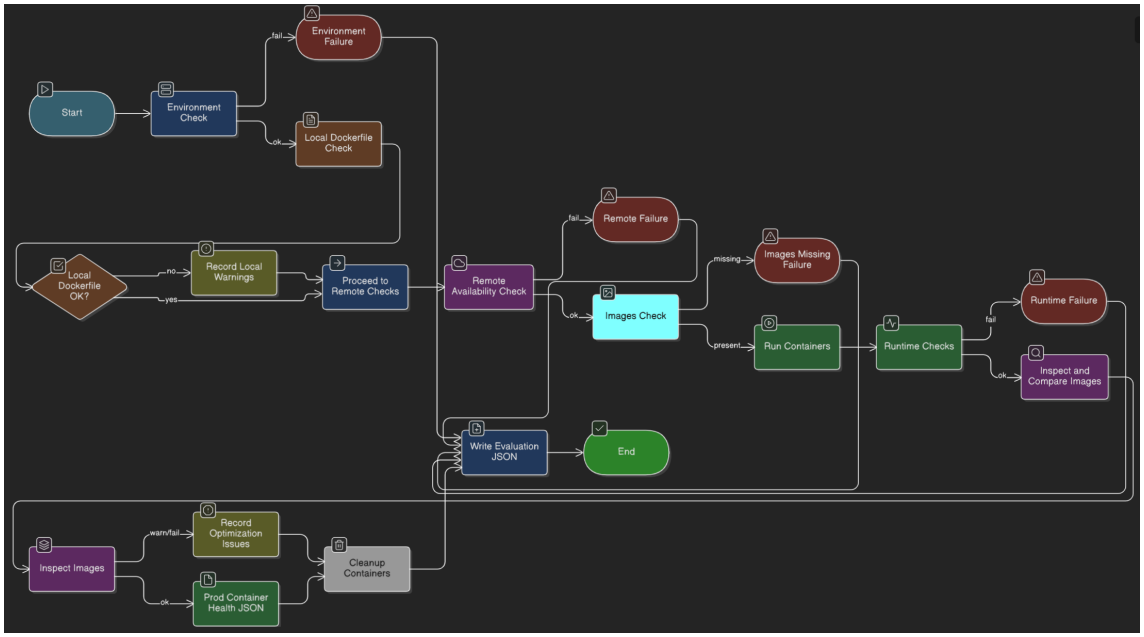


Figure 4.7: Flow of autograder verification for Activity 6.

- **Build caching inconsistencies:** different student environments caused repeated module downloads and slow builds.
  *Solution:* enforced BuildKit usage and demonstrated cache mounts for `go mod` and `npm ci`.

- **Permission issues in minimal images:** non-root users lacked correct ownership of copied files.
  *Solution:* corrected ownership during build using `chown` and verified through runtime UID inspection.

- **Large image sizes from full base images:** naive builds inherited toolchains and shells from full images.
  *Solution:* demonstrated multi-stage extraction into `scratch` or `distroless`, achieving $> 100\times$ size reduction.

- **Trivy scan overhead:** running full DB updates slowed grading.
  *Solution:* used `-skip-db-update` mode in autograder while allowing full scans for instructors.

**Remarks:** Activity 6 emphasizes practical, quantifiable optimization in container builds. Students witness firsthand how layer ordering, BuildKit caching, and minimal bases translate into faster, safer, and smaller production images. The Go example provides the conceptual depth, while the Node exercise reinforces pattern generalization across

languages. The autograder's measurable checks (size, health, CVEs) make this lab one of the most impactful in demonstrating Docker's real-world best practices.

For the full student-facing problem statement and detailed instructions, refer to the official **Activity 6 Problem Statement Document**.

## 4.9 Activity 7: Advanced Docker Concepts (Bonus)

**Objective:** Explore advanced Docker internals and ecosystem topics (daemon/API, namespaces, cgroups, storage drivers, rootless mode, OCI standards, and kernel interfaces). This is a self-study, non-graded bonus lab for students who want to understand how containers work under the hood.

**Overview:** Activity 7 is a guided deep-dive that pairs concise theory with hands-on host-level experiments on an Ubuntu EC2 instance. Students inspect the Docker daemon and Engine API, examine namespace/cgroup/capability mechanics, probe storage drivers and image layers, experiment with rootless Docker concepts, and explore low-level tooling (`containerd`, `runc`, `unshare`, `/proc` and cgroup interfaces). The lab emphasizes observation and experimentation rather than producing deliverables.

**Implementation Details:**

- **Daemon & API exploration:** Demonstrated how the Docker CLI interacts with the `dockerd` daemon through the Unix socket, and executed basic REST API calls (`GET /containers/json`, `POST /containers/create`) using `curl --unix-socket`.
- **Namespaces / cgroups / capabilities:** Launched containers, inspected their process namespaces via `/proc/$PID/ns`, viewed cgroup membership in `/proc/$PID/cgroup`, and observed privilege differences using capability drops.
- **Storage drivers & layering:** Examined the active storage driver (typically `overlay2`), viewed image histories, and visualized union filesystem layering to illustrate image immutability and container writable overlays.
- **Rootless awareness:** Introduced the concept of rootless Docker, explained user namespace remapping, and discussed the networking/storage shims (`slirp4netns`, `fuse-overlayfs`) enabling non-root operation.
- **Low-level runtimes & OCI:** Demonstrated Docker's runtime stack (Docker → containerd → runc) and ran sample images with `ctr` and minimal OCI bundle executions via `runc`.

- **Scaffold tooling (`docker init`):** Showed how Docker's new `init` command generates starter Dockerfiles for common languages, and cautioned students to treat the result as a baseline to be hardened with production best practices (multi-stage, healthcheck, non-root).

**Testing and Autograder Strategy:**

- **Observation-based verification:** As this is a non-graded lab, instructors provided self-check steps instead of automated scoring. Students validate that API calls work, namespaces are isolated, and system resources are properly controlled by cgroups.
- **Safe experiments:** All host-level commands were curated to be non-destructive, relying on temporary containers (e.g., `busybox`) for inspection. Cleanup scripts were included to restore the environment.

**Remarks:** Activity 7 is an exploratory capstone that bridges theory with Docker internals. It complements earlier labs by showing how namespaces, cgroups, and runtimes cooperate to isolate processes and resources. Students gain a conceptual foundation for understanding container engines, kernel interfaces, and emerging OCI-based runtime ecosystems.

For the full student-facing problem statement and detailed instructions, refer to the official **Activity 7 Problem Statement Document**.

# Overall Remarks

The Docker Lab as a whole provides a structured, incremental journey from foundational containerization concepts to advanced production and internal aspects of Docker. Across Activities 0 through 7, students progressively evolve from simple image builds and container runs to constructing secure, optimized, and multi-service deployments.

The lab sequence is intentionally designed to mirror real-world adoption patterns:

- **Activity 0–2:** Build confidence with the Docker CLI, images, and containers through small, self-contained examples and basic automation.
- **Activity 3–5:** Transition to production-oriented container workflows by containerizing a realistic application, optimizing Dockerfiles, and orchestrating multi-container systems with Compose.
- **Activity 6:** Introduces industry-grade build patterns such as multi-stage builds, cache optimization, image hardening, and security scanning.

- **Activity 7:** Provides conceptual depth by exposing Docker's internals—how namespaces, cgroups, and runtimes form the foundation of containerization—and situates Docker within the broader OCI ecosystem.

Pedagogically, the lab moves from "*what Docker does*" to "*how Docker does it*." By the end of this sequence, students acquire:

- the ability to design and optimize production-ready images,
- awareness of container security and runtime principles,
- and an appreciation of Docker's architecture and its role in the modern container ecosystem.

The Docker Lab thus bridges the gap between classroom demonstrations and practical DevOps engineering, preparing students to design, deploy, and reason about containerized systems confidently and responsibly.

# Chapter 5

# All About Serverless Framework V4

## 5.1 Introduction

The **Serverless Framework** is one of the most popular open-source toolkits for developing, deploying, and managing serverless applications across cloud providers such as AWS, Azure, and Google Cloud. It abstracts away the complexities of provisioning and managing cloud resources, allowing developers to focus on writing application logic instead of dealing with infrastructure setup.

Serverless computing represents a paradigm shift in cloud architecture—developers deploy *functions* that automatically scale, execute on demand, and require no explicit server management. The Serverless Framework (abbreviated as SLS) provides a unifying developer experience around this model by integrating configuration, deployment automation, environment management, and observability into a single workflow.

## 5.2 Why Serverless Framework v4 Matters

Version 4 of the framework (released in 2024) introduced a series of improvements designed for modern cloud-native development:

- **Faster deployments:** Uses incremental deployments and parallel packaging to drastically reduce deployment times.
- **Improved developer experience:** Simplified YAML syntax, smarter error handling, and enhanced CLI feedback for better readability.
- **Native support for AWS Lambda versions and aliases:** Allows smooth blue-green and canary deployments directly via configuration.
- **Extended observability:** Integrated metrics and logs via `serverless dashboard`, making it easier to monitor invocation counts, durations, and errors.

- **Modular plugin architecture:** Clean plugin ecosystem supporting framework extensions for serverless offline, packaging, secrets, or multi-provider support.
- **Improved local development:** Enhanced `serverless offline` and `invoke local` utilities to simulate Lambda + API Gateway behavior locally.

## 5.3 Core Components of the Serverless Framework

The framework consists of several coordinated components that together enable a seamless function-based deployment workflow.

### 5.3.1 Serverless CLI

The primary user interface for the framework. It enables developers to:

- Initialize new projects (`serverless create`).
- Deploy and remove services (`serverless deploy`, `serverless remove`).
- Invoke functions locally or remotely (`serverless invoke`).
- Manage logs, metrics, and stack information (`serverless logs`, `serverless info`).

It serves as the bridge between the developer's code and the underlying cloud infrastructure.

### 5.3.2 `serverless.yml`

At the heart of every project lies the `serverless.yml` configuration file. This declarative YAML defines:

- **Provider:** The cloud platform (e.g., AWS, Azure, Google Cloud).
- **Functions:** Each Lambda function, its handler file, events (HTTP endpoints, SQS triggers, S3 events), and environment variables.
- **Resources:** Additional infrastructure (DynamoDB tables, S3 buckets, SNS topics) defined as CloudFormation templates.
- **Plugins & Custom Variables:** Reusable configuration snippets and plugins extending framework functionality.

This YAML-driven configuration ensures reproducibility and infrastructure-as-code (IaC) principles.

### 5.3.3 Providers and Plugins

Serverless Framework supports multiple providers through modular plugins. The most widely used provider is AWS, where Serverless manages Lambda functions, API Gateway endpoints, SQS queues, and DynamoDB tables seamlessly. Plugins like `serverless-offline`, `serverless-python-requirements`, and `serverless-domain-manager` further enhance the developer experience by adding simulation, dependency packaging, and domain mapping capabilities.

### 5.3.4 Serverless Dashboard

An optional but powerful companion service offering:

- Real-time monitoring and alerts for deployed functions.
- Deployment history and rollback support.
- Secure parameter and secret management.
- Team-based collaboration and environment control.

### 5.3.5 Deployment Process Overview

When you run `serverless deploy`, the framework:

1. Parses and validates `serverless.yml`.
2. Packages each function with its dependencies.
3. Generates a CloudFormation template (for AWS) or equivalent provider-specific infrastructure template.
4. Uploads artifacts (code + configuration) to the cloud provider.
5. Creates or updates the necessary resources and deploys functions.

## 5.4 Serverless vs Traditional Architectures

Unlike traditional applications that run continuously on servers, serverless applications run in ephemeral, event-driven environments. Table 5.1 highlights the comparison.

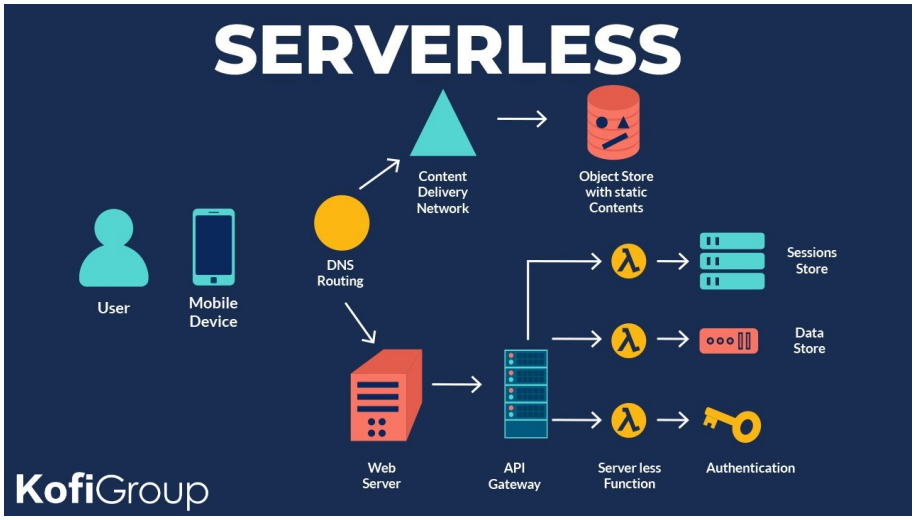


Figure 5.1: High-level architecture of the Serverless(Adapted from [3]).

Table 5.1: Comparison between Traditional and Serverless Architectures

| **Aspect** | **Traditional Server-based** | **Serverless (FaaS)** |
|---|---|---|
| Deployment Unit | Full server or container | Individual function |
| Scaling | Manual or auto-scaling groups | Automatic, event-driven scaling |
| Billing | Pay for uptime or reserved capacity | Pay only per execution |
| Management | Requires OS, patching, monitoring | Fully managed by provider |
| Deployment Time | Minutes to hours | Seconds |
| Complexity | Requires DevOps setup | Simplified deployment |

## 5.5 How It Works Internally

At its core, the framework generates provider-native deployment templates (e.g., AWS CloudFormation) under the hood. These templates define all cloud resources required to deploy and connect your serverless functions. When you execute `serverless deploy`, the following sequence occurs:

1. Functions are packaged and uploaded to a storage bucket (e.g., S3 on AWS).
2. CloudFormation (or provider equivalent) is triggered to create/update stacks.
3. Event triggers (API Gateway routes, SQS queues, etc.) are wired to the function handlers.

4. The deployment state is tracked to enable future incremental updates or rollbacks.

This architecture ensures that every deployment is declarative, version-controlled, and reversible — aligning strongly with modern DevOps and IaC practices.

## 5.6 Why Learning Serverless Framework is Important

Serverless Framework simplifies the creation of scalable, event-driven systems that are:

- **Cost-efficient:** Pay only for what executes.
- **Scalable:** Automatically scales with traffic and usage.
- **Portable:** Abstracts cloud-specific details, enabling deployment across providers.
- **Developer-friendly:** Offers CLI commands, templates, and plugins to minimize setup.
- **DevOps aligned:** Integrates CI/CD pipelines, environment promotion, and IaC.

For learners, mastering the Serverless Framework equips them with industry-relevant skills to design resilient, event-driven architectures on modern cloud platforms—skills that are foundational for cloud-native backend development and modern DevOps pipelines.

## 5.7 Remarks

The Serverless Framework v4 stands at the intersection of simplicity, scalability, and automation. It bridges the gap between developers and cloud infrastructure, making serverless architectures accessible and efficient. Understanding its design, deployment model, and best practices lays a strong foundation for the practical labs that follow, where each activity builds on these principles to create, deploy, and test serverless applications in real AWS environments.

# Chapter 6

# Serverless Lab

## 6.1 Overview

The **Serverless Framework v4 Lab** offers a hands-on, progressive introduction to developing event-driven, serverless applications on AWS using the latest version of the Serverless Framework. The lab transitions students from fundamental configuration and environment setup to deploying complete, multi-service applications integrating Lambda, API Gateway, DynamoDB, SQS, SNS, and S3 — all defined declaratively through `serverless.yml`.

**Objective:** To teach students how to design, implement, deploy, and manage scalable serverless applications using Infrastructure-as-Code (IaC) principles with the Serverless Framework. Through guided exercises, students learn not just the syntax of `serverless.yml`, but also the architectural reasoning behind each component, including IAM roles, events, and deployment automation.

**Motivation:** Traditional server-based architectures require provisioning, scaling, and maintaining servers. In contrast, serverless architectures automatically scale with demand and charge only for execution time. The Serverless Framework simplifies this paradigm by abstracting cloud infrastructure, enabling developers to define an entire application — from compute to storage to events — within a single configuration file.

This lab closely mirrors how modern cloud-native systems are built in the industry. Students will gain experience in real AWS environments using best practices in configuration, deployment, permissions, and monitoring.

**Pedagogical Design:** The lab is structured across four major activities that incrementally increase in depth and autonomy:

- **Activity 1 – Environment Setup:** Students configure their local development environment for serverless development. This includes installing and verifying the

AWS CLI, Node.js, Python, and the Serverless Framework CLI. They also create IAM users, configure profiles, and connect the Serverless CLI with their account.

- **Activity 2 – Core Concepts:** Introduces the theoretical and practical foundations of the Serverless Framework. Students explore key constructs such as `service`, `provider`, `functions`, `events`, and `resources` within the `serverless.yml`. They learn how the framework maps these abstractions to AWS resources via CloudFormation and how to run, test, and remove deployments using essential CLI commands.

- **Activity 3 – Guided Project:** A step-by-step "hand-held" project where students build a complete serverless microservice with Lambda functions, DynamoDB integration, and event-driven messaging using SNS and SQS. The project is scaffolded to ensure clear understanding of IAM permissions, function triggers, CloudFormation resource definitions, and deployment verification.

- **Activity 4 – Independent Assignment:** Students independently build and deploy a serverless image processing pipeline — integrating S3, Lambda, DynamoDB, and SNS. This activity tests their ability to design end-to-end workflows, configure IAM roles securely, and manage multi-resource stacks. It includes a self-verifiable checklist and an autograder-based evaluation mechanism.

**Learning Outcomes:** By the end of the Serverless Framework Lab, students will be able to:

- Configure and deploy AWS resources through declarative IaC using `serverless.yml`.

- Design and deploy Lambda-based applications triggered by diverse events (HTTP, S3, SQS, SNS).

- Implement and verify IAM roles with least-privilege permissions.

- Utilize Serverless Framework plugins for packaging, dependency management, and local testing.

- Interpret CloudFormation stacks and troubleshoot deployments via logs and CLI tools.

For the complete student-facing problem statement and activity instructions, refer to the official document: **Serverless Framework v4 Lab Problem Statement Document**.

## 6.2 Activity 1: Environment Setup

**Objective:** This activity establishes the foundational cloud environment required for all subsequent Serverless Framework exercises. Students configure AWS IAM credentials, the AWS CLI, Python, Node.js (via `nvm`), and the Serverless Framework CLI, completing a fully functional setup capable of deploying and managing AWS Lambda functions. The goal is to ensure every learner can authenticate programmatically to AWS, manage functions via the CLI, and access the Serverless Dashboard for later monitoring and deployment stages.

**Overview:** Activity 1 serves as the infrastructure initialization phase of the Serverless Framework Lab series. Unlike later activities that involve Lambda deployments, this task focuses purely on preparing a clean, consistent development and cloud environment. Students begin by creating a dedicated IAM user with programmatic access and attaching the `AdministratorAccess` policy (for educational simplicity). They then install and configure the AWS CLI using a named profile `serverless-lab`, which isolates lab credentials from other AWS configurations. Subsequent steps involve setting up Python ($\geq$ 3.9) and Node.js (LTS) to support Lambda development and the Serverless CLI respectively.

To keep the setup reproducible, the activity includes an *ephemeral workspace reset* command that cleans the cLab environment between attempts. This ensures each student always starts with a known baseline, avoiding contamination from earlier partial setups.

The final verification phase consolidates the entire configuration pipeline—students confirm AWS identity via `aws sts get-caller-identity`, check runtime versions for Python and Node, and validate the Serverless Framework installation with `serverless -version`. This guarantees readiness for subsequent activities involving deployments, IAM role creation, and event-driven triggers.

**Implementation Details:** From an instructional standpoint, the activity was designed to minimize configuration drift across heterogeneous student systems. Each tool installation—AWS CLI, Python, Node.js, and Serverless CLI—was documented with both direct package commands and official reference links to ensure transparency and future reproducibility.

Key design considerations included:

- **Isolated AWS profile:** enforcing use of `-profile serverless-lab` prevents interference with existing credentials and enables targeted autograder checks in later labs.

- **Language parity:** Python serves as the default runtime for Lambda functions throughout the lab, while Node.js underpins the Serverless CLI; installing both establishes a uniform toolchain.

- **Ephemeral design:** because cLab environments reset after each session, a simple `find`-based cleanup command was provided to remove all transient files except the problem statement—keeping re-runs deterministic.

- **Dashboard integration:** students authenticate once via `serverless login`, linking the local CLI with the online Serverless Dashboard for deployment tracking and environment metadata.

This step-by-step yet platform-agnostic design makes the environment preparation robust for both cloud and local Linux setups, while remaining concise enough for quick replication.

**Challenges faced and solutions:**

- **Challenge 1 – Credential misconfiguration:** Early learners often mis-entered access keys or default regions.
  *Solution:* Added explicit verification via `aws sts get-caller-identity` and detailed explanation of each returned field (UserId, Account, Arn) so students could self-diagnose configuration issues.

- **Challenge 2 – Version mismatches between Node and Serverless CLI:** Some older Node versions failed to install the latest CLI globally.
  *Solution:* Standardized installation through `nvm install -lts` ensuring all students operate on a supported LTS release.

- **Challenge 3 – Environment persistence limitations:** Since the lab environment resets after logout, progress occasionally disappeared unexpectedly.
  *Solution:* Emphasized the ephemeral warning and provided the cleanup command to enforce reproducibility across re-entries.

- **Challenge 4 – Dashboard authentication issues:** Browser-less environments caused login friction.
  *Solution:* Clarified that students could perform `serverless login` locally with GitHub SSO, then re-use the generated credentials in the cloud shell.

**Remarks:** Activity 1 lays the essential groundwork for all following Serverless Framework labs. By the end of this stage, every participant has a fully operational

AWS CLI profile, functional Python and Node.js runtimes, and a verified Serverless CLI linked to the online dashboard. This uniform baseline guarantees that later deployments—covering Lambda functions, DynamoDB tables, S3 triggers, and event integrations—can execute smoothly without setup-related discrepancies. The emphasis on reproducibility, version control, and tool alignment makes this activity an indispensable starting point for the Serverless Framework v4 Lab sequence.

For the complete student-facing problem statement and detailed step-by-step guide, refer to the official **Activity 1 Problem Statement Document**.

## 6.3 Activity 2: Serverless Framework v4 Core Concepts

**Objective:** This activity introduces the foundational concepts, configuration structure, and core CLI operations of the Serverless Framework v4. Students learn what the framework does, the key YAML keywords that define a Serverless service, and how deployments translate into AWS CloudFormation resources.

The aim is to make learners comfortable with interpreting and running Serverless commands such as `deploy`, `package`, `invoke`, `logs`, and `remove`, and to understand how Serverless Framework automates AWS infrastructure creation and function deployment.

**Overview:** Activity 2 serves as the conceptual backbone of the Serverless Framework lab series. It builds upon the environment prepared in Activity 1, introducing the declarative configuration style used by Serverless Framework and explaining its relation to AWS CloudFormation.

Students explore how a short YAML definition can describe an entire application stack, including functions, events, resources, outputs, and plugins. They also learn the difference between local and deployed execution, and how the Serverless CLI interacts with AWS services behind the scenes. The activity concludes with a guided hands-on session where students deploy small demo services and inspect the resulting CloudFormation templates, providing valuable insight into how the framework abstracts complex AWS operations.

**Implementation Details:**

- **Conceptual coverage:** The content was structured to gradually progress from idea-level understanding ("What is Serverless?") to technical YAML-level mastery. Each keyword in `serverless.yml` — such as `service`, `provider`, `functions`, `events`, `resources`, `stages`, and `outputs` — was introduced with simple definitions, YAML examples, and practical use-cases.

- **Hands-on walkthrough:** Students created a small Python-based demo service using the Serverless CLI and performed a complete lifecycle:

  1. Initialize a new project with `sls`.
  2. Edit and explore `serverless.yml`.
  3. Package and deploy using `sls deploy`.
  4. Invoke functions both locally and remotely using `sls invoke` and `sls invoke local`.
  5. Observe logs and stack outputs using `sls logs` and `sls info`.
  6. Tear down the stack using `sls remove`.

- **Generated artifacts inspection:** Students examined the CloudFormation template generated under `.serverless/` to understand how Serverless transforms high-level YAML definitions into AWS-native resources.

- **Runtimes and defaults:** The lab enforced `python3.10` as the default runtime across all subsequent exercises, ensuring uniformity and avoiding compatibility issues.

- **Plugin integration:** The two most relevant plugins—`serverless-python-requirements` and `serverless-offline`—were demonstrated for dependency packaging and local API simulation, respectively. Additional plugin examples (`serverless-deployment-bucket`, `serverless-plugin-canary-deployments`) were also briefly covered to show extensibility.

- **Local vs deployed workflows:** Students were shown the practical differences between `sls invoke local` (fast iteration) and full AWS deployments (IAM, networking, VPC constraints). Real DynamoDB local integration via Docker was also demonstrated to illustrate hybrid testing workflows.

- **Variable and stage handling:** Detailed examples explained how Serverless resolves `${self:}`, `${env:}`, `${ssm:}`, and `${file:}` variables, allowing dynamic configuration and stage-based deployments (e.g., dev/prod).

- **Internal process visualization:** The step-by-step breakdown of the `sls deploy` process (YAML parsing → packaging → artifact upload → CloudFormation execution → outputs) helped clarify the automation pipeline underlying every Serverless deployment.

**Instructor-side Implementation:** I authored the complete instructional material for Activity 2, ensuring that theoretical explanations are directly tied to practical examples. Each keyword and command was supported with a real YAML or CLI demonstration that could be run in the student workspace. In addition, I created a demo directory structure ( `/serverless_lab/Activity2_demo`) to serve as the working sandbox for all examples.

To encourage conceptual reinforcement, I also developed a short quiz at the end of the activity using the **BodhiTree Quiz Creator Tool**. This quiz assesses students' understanding of YAML structure, command semantics, and plugin configuration.

**Consistency improvement — unified CLAB environment:** To make the development experience seamless after Activity 1 and eliminate repetitive setup steps, I restructured the CLAB base Docker image used for all subsequent activities. The image was rebuilt to include all prerequisite tools — Node.js (v20), AWS CLI v2, Serverless Framework (latest), Python with pip and Paramiko, and other essential Linux utilities preinstalled and preconfigured.

This ensured that every student container launched with a ready-to-use environment, matching the same tool versions used in the instructor and autograder setups. By baking these dependencies directly into the Docker image, students no longer needed to manually install or configure software for each lab, significantly reducing setup time and environment drift across activities.

This optimization also improved reproducibility of the autograder executions, as all grading scripts could assume a consistent baseline environment without conditional setup logic.

**Testing and Verification Approach:** This activity was intentionally designed as self-verified and exploratory rather than autograded. Students validated each concept through direct hands-on commands, with key checks including:

- Correct configuration of AWS credentials and Serverless Dashboard login.
- Presence and syntax of the generated `serverless.yml`.
- Successful creation of demo Lambda via `sls deploy`.
- Function invocation results and log inspection outputs.
- Verification of the deployment S3 bucket and uploaded artifacts.
- Correct teardown of stacks via `sls remove`.

Each step reinforced a conceptual pillar of the framework and allowed students to visually confirm how Serverless translates declarative YAML definitions into deployed AWS resources.

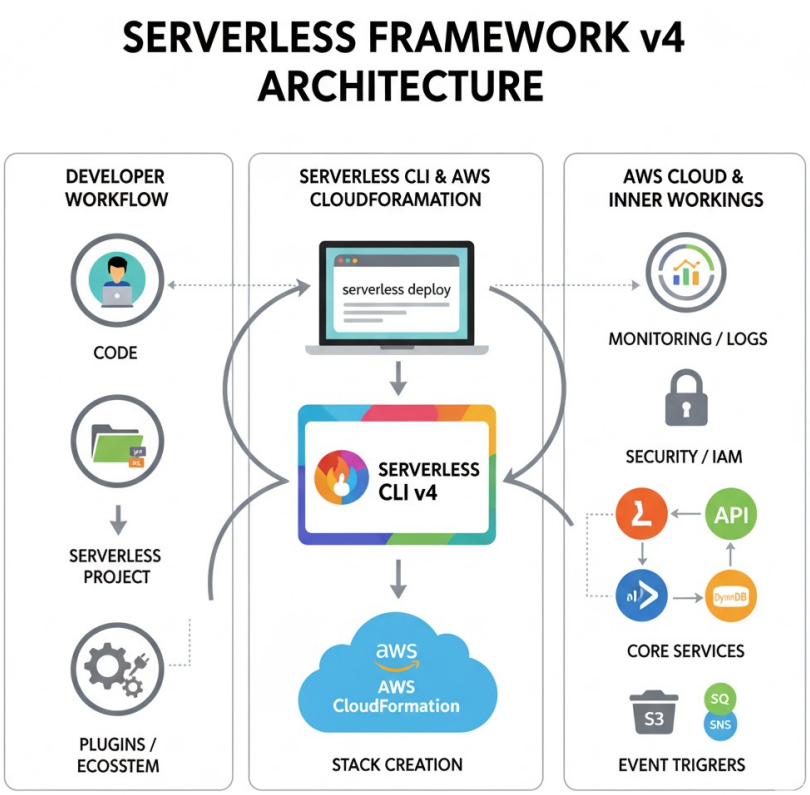


Figure 6.1: Illustration of the Serverless Framework deployment workflow.

**Challenges Faced and Solutions:**

- **Challenge 1 – Student confusion between Serverless and "serverless computing":** Many initially assume the two terms meant the same thing.
  **Solution:** Clarified through examples that "serverless" is a paradigm, while "Serverless Framework" is a tool that implements it for easier AWS deployments.

- **Challenge 2 – Plugin installation errors due to Node version mismatches:** Students encountered dependency errors while installing plugins.
  **Solution:** Standardized Node installation using `nvm` (LTS) and verified compatibility with Serverless v4.

- **Challenge 3 – Understanding deployment bucket purpose:** Students were often unsure why artifacts were uploaded to S3.
  **Solution:** Added an explicit walkthrough showing S3 artifact paths post-deploy and explained the link to CloudFormation's deployment process.

- **Challenge 4 – Variable resolution syntax complexity:** YAML interpolation using `${self:}` and `${ssm:}` syntax caused confusion.
  **Solution:** Provided a side-by-side example with both definition and resolution phases, making variable scoping clearer.

**Remarks:** Activity 2 plays a pivotal role in bridging conceptual understanding and hands-on proficiency. By mastering Serverless v4 syntax, deployment flow, and command usage, students gain the confidence to define, deploy, and manage complete serverless applications in subsequent activities.

The inclusion of a structured quiz at the end reinforces key ideas, ensuring that learners internalize not just the "how" but also the "why" behind Serverless Framework operations. This balance of guided instruction, live CLI execution, and conceptual testing makes Activity 2 a cornerstone of the Serverless lab series.

For the full problem statement and command reference, refer to the official **Activity 2 Problem Statement Document**.

## 6.4 Activity 3: Building a Serverless Application (Hand-held)

**Objective:** Guide students through building and deploying a complete serverless application using Serverless Framework v4 and AWS. Learners will initialize a Python Serverless project, implement HTTP and event-driven Lambda functions, wire DynamoDB, SNS, and SQS, apply IAM least-privilege, inspect the generated CloudFormation, and cleanly remove deployed resources.

**Overview:** Activity 3 is a hands-on, incremental project that takes students from an empty repository to a functioning multi-service serverless stack. The exercise is intentionally scaffolded: start with a minimal 'hello' function to validate credentials and tooling, then add persistent storage (DynamoDB), synchronous HTTP endpoints (API Gateway → Lambda), and an asynchronous messaging pipeline (SNS → SQS → Lambda). This sequencing builds confidence while introducing real operational concerns (IAM permissions, artifact packaging, CloudFormation stacks, and log inspection).

**Instructor-built components (to ensure a smooth student experience):**

- **Starter project skeleton:** a pre-scaffolded 'activity3-app' containing 'serverless.yml' skeleton, '.gitignore', and example 'requirements.txt' so students begin from a working baseline.

- **Reference handlers:** tested Python implementations for `hello_py`, `create_todo.py`, `publish_message.py`, and `consume_message.py` that demonstrate JSON handling, simple idempotency patterns, and correct use of AWS SDK (`boto3`).
- **IAM statement templates:** ready-to-use IAM role snippets (CloudWatch, DynamoDB, SQS, SNS) included in the example `serverless.yml` illustrating least-privilege scoping and avoiding common AccessDenied errors.
- **CLI helper snippets and Makefile:** common commands standardized (‘sls package‘, ‘sls deploy‘, ‘sls logs‘, ‘sls remove‘, ‘sls info‘) to reduce typing errors and speed iteration.
- **Verification checklist and troubleshooting guide:** concise, stepwise checks (CloudFormation events, Lambda configuration, API routes, CloudWatch log locations) and quick fixes so students can self-diagnose deployment failures.
- **Environment parity integration:** the activity assumes the unified CLAB environment (preinstalled Serverless, AWS CLI, Python) to eliminate repetitive installs and reduce environment drift.

**Implementation Details (pedagogical flow):**

1. **Initialize & validate:** scaffold service with ‘sls‘ and deploy a ‘hello‘ function to verify AWS profile and Serverless linkage.
2. **Add persistence:** declare a DynamoDB ‘TodoTable‘ in ‘resources:‘ and implement ‘create_todo‘ to ‘PutItem‘.
3. **Introduce messaging:** create ‘MyTopic‘ (SNS) and ‘MyQueue‘ (SQS), wire subscription and queue policy, and add ‘publish_message‘ and ‘consume_message‘ handlers to demonstrate asynchronous, resilient processing.
4. **IAM & least privilege:** show how to scope each permission to the service/stage and explain the operational reasons for each policy statement.
5. **Verify & debug:** inspect ‘.serverless/‘ CloudFormation template, use ‘sls info‘, stream logs with ‘sls logs -f‘, and read CloudFormation Events to resolve deployment errors.
6. **Cleanup:** teach ‘sls remove‘ and emphasise cost control and resource hygiene.

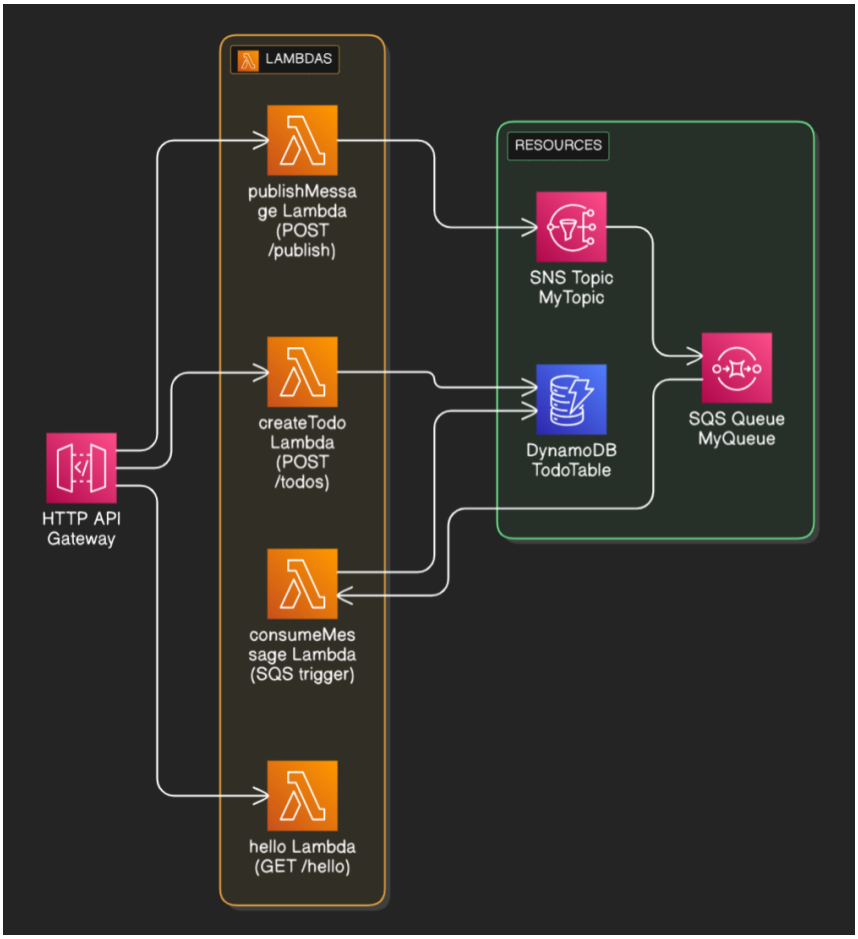


Figure 6.2: Architecture of the guided Serverless application built in Activity 3.

**Testing & Verification (student checklist):**

- ‘hello‘ endpoint responds to ‘curl‘ (HTTP 200).
- POST to ‘/todos‘ inserts an item visible in DynamoDB Console.
- POST to ‘/publish‘ returns a ‘MessageId‘ and SNS topic exists in Console.
- Messages are delivered to SQS and processed by ‘consumeMessage‘ (verify via CloudWatch logs).
- ‘sls info‘ and CloudFormation Resources show expected stack elements (Lambda, API, Table, Topic, Queue).
- ‘sls remove‘ successfully tears down the stack.

**Challenges faced and instructor solutions:**

- **Students stumbling on scaffolding and YAML minutiae:** provided the starter skeleton and filled ‘serverless.yml‘ templates to reduce syntax-related friction.

- **Runtime AccessDenied errors at test time:** included tested IAM snippets and explicit explanation of why each permission is required.

- **Environment inconsistencies across student containers:** integrated prerequisites into the CLAB base image so everyone starts from the same tool versions and avoids local install issues.

**Assessment — quiz:** A short formative quiz is included at the end of Activity 3 (concepts and configuration checks). The quiz was created and published using the BodhiTree Quiz Creator Tool and provides quick feedback on students' understanding of topics such as event wiring, IAM scoping, and the Serverless deploy lifecycle.

**Remarks:** Activity 3 serves as the first complete, guided implementation of a multi-service serverless application. Through this step-by-step, instructor-led exercise, students not only learn the declarative syntax of `serverless.yml` but also understand how individual AWS services interact in an event-driven system. Each successive step—from deploying a basic function to wiring DynamoDB, SNS, and SQS—builds on the previous one, allowing students to connect conceptual theory from Activity 2 to real AWS behavior.

By the end of this activity, students are technically and conceptually ready to proceed to **Activity 4**, where they independently design and deploy a full serverless workflow without instructor scaffolding.

For the full student-facing problem statement and step-by-step instructions, refer to the official **Activity 3 Problem Statement Document**.

## 6.5 Activity 4: Serverless Image Processing Pipeline (Student Assignment)

**Objective:** In this activity, students independently design and deploy a complete serverless image processing pipeline using Serverless Framework v4 on AWS. This exercise consolidates all concepts learned in the previous activities — integrating multiple AWS services (S3, Lambda, DynamoDB, and SNS) through declarative configuration and event-driven orchestration.

**Overview:** This is the final and evaluated activity of the Serverless Framework Lab series. Students build a serverless image processing pipeline where:

1. A user uploads an image to an S3 Uploads bucket.

2. A Lambda function (`image_processor.py`) resizes it into a thumbnail using Pillow, stores metadata in DynamoDB, and publishes a result message to SNS.

3. Another Lambda (`analytics_logger.py`) subscribes to the SNS topic and writes structured analytics logs to an Analytics S3 bucket.

This activity tests each student's ability to define AWS resources, function triggers, IAM policies, and outputs entirely from scratch. It represents the culmination of the step-by-step learning sequence started in Activity 1.

**Instructor-side enhancements for seamless experience:**

As this activity runs inside a Dockerized CLab environment, certain system-level improvements were implemented to ensure smooth persistence, plugin support, and compatibility:

- **Persistent workspace mounting:** Earlier CLab environments were ephemeral — meaning that once a student exited, all local progress was lost. To overcome this, I modified the lab's Dockerfile so that the `/home/labDirectory` (student workspace) is now mounted onto a stable, pre-existing Docker volume. This ensures that all files, configurations, and deployment artifacts remain intact even after the lab container stops and restarts, allowing students to resume seamlessly.

- **Docker-in-Docker compatibility:** The activity uses the `serverless-python-requirements` plugin, which leverages Docker internally to build dependency packages. Running Docker inside the CLab container caused permission and socket binding conflicts. As a workaround — after extensive testing and discussions — this particular lab container was configured to run with the `-privileged` flag. This grants nested Docker operations the necessary privileges to execute properly within the environment. The CLab developer team has added a beta version specifically for this configuration, and future iterations aim to streamline this approach through a controlled plugin execution model.

- **Plugin readiness:** The base Docker image now comes preinstalled with Node.js, AWS CLI v2, Python3, and the Serverless Framework CLI. Additionally, the `serverless-python-requirements` plugin can be installed per project without host-level issues.

**Implementation Summary:** Students are provided with the folder structure and Lambda function templates. They must focus on authoring a correct and complete `serverless.yml` file to define:

- Three S3 buckets — Uploads (private), Thumbnails (public-read), and Analytics (private).

- A DynamoDB table (`JobsTable`) with `jobId` as the primary key.
- An SNS topic (`ImageEventsTopic`) for communication between the processor and logger functions.
- Two Lambda functions (`imageProcessor` and `analyticsLogger`) wired via S3 and SNS triggers respectively.
- Proper IAM roles and permissions for logs, S3, DynamoDB, and SNS access.
- Outputs for easy reference of resource names and ARNs.

**Student-side tasks:**

1. Deploy the stack using `sls deploy -stage dev`.
2. Upload a test image (`LEARN.jpg`) to the Uploads bucket and observe thumbnail creation in Thumbnails bucket.
3. Verify that metadata appears in DynamoDB and analytics logs are written to the Analytics bucket.
4. Tail function logs using `sls logs -f imageProcessor` and `sls logs -f analyticsLogger`.
5. Populate `data.json` with instructor credentials and stack details for evaluation.

### 6.5.1 Testing and Autograder Strategy

**Purpose:** A lightweight, non-destructive autograder validates student submissions for Activity 4. It performs workspace sanity checks, credential validation, CloudFormation/resource verification, and an end-to-end integration test (upload → thumbnail → DynamoDB → analytics). The autograder uses instructor IAM credentials from `data.json` and outputs structured results to `evaluate.json`.

**Implementation Highlights:**

- Written in Python using `boto3`, executed from the lab host.
- Uploads one temporary test image, verifies all pipeline stages, then cleans up.
- Performs AWS STS, S3, Lambda, SNS, and DynamoDB checks in sequence.
- Reports detailed results and diagnostic hints for debugging.

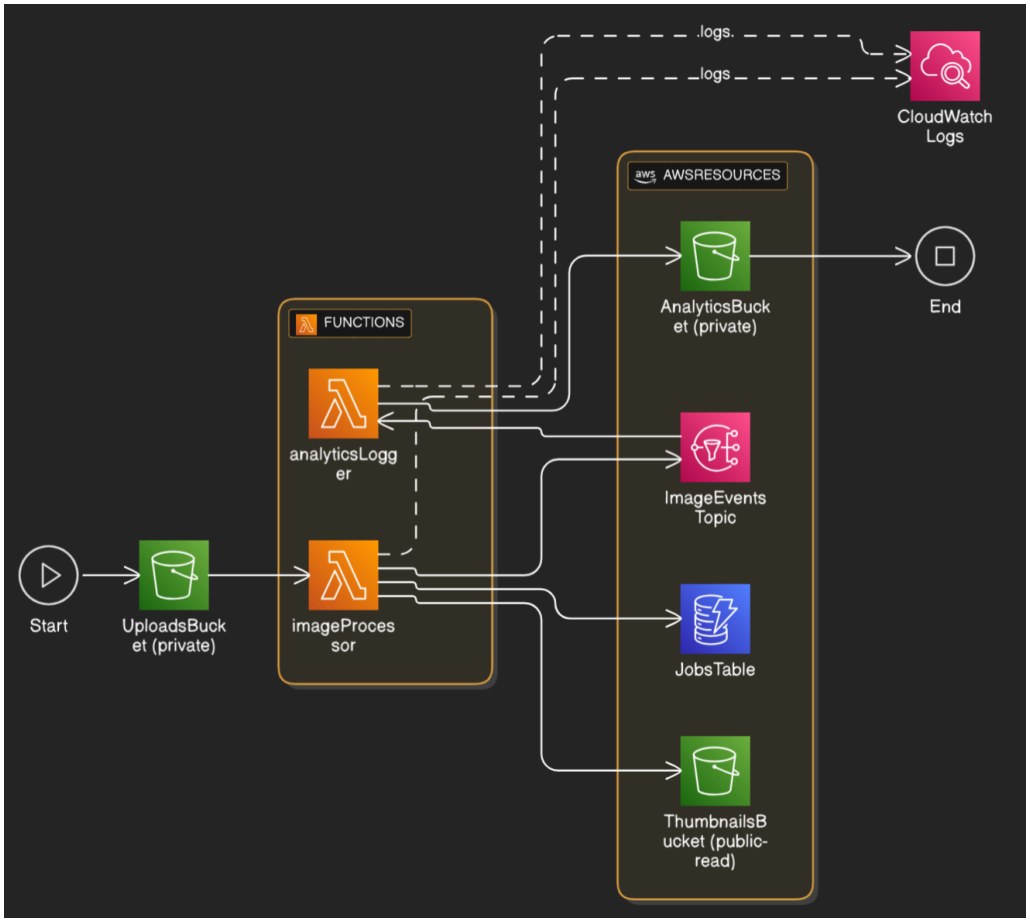


Figure 6.3: Overall architecture of the serverless image processing pipeline.

**Why this Matters:** This structured autograding process ensures both infrastructure correctness and functional validation. It helps students confirm that their deployed system truly works in the AWS environment rather than just syntactically passing configuration checks.

### 6.5.2 Challenges and Solutions

- **Ephemeral environment losses:** Solved by mounting the workspace to a persistent Docker volume.
- **Docker-in-Docker errors:** Addressed using the `-privileged` flag for isolated plugin builds.
- **Dependency inconsistency:** Resolved by preinstalling Node, Python, and AWS tools in the base image.

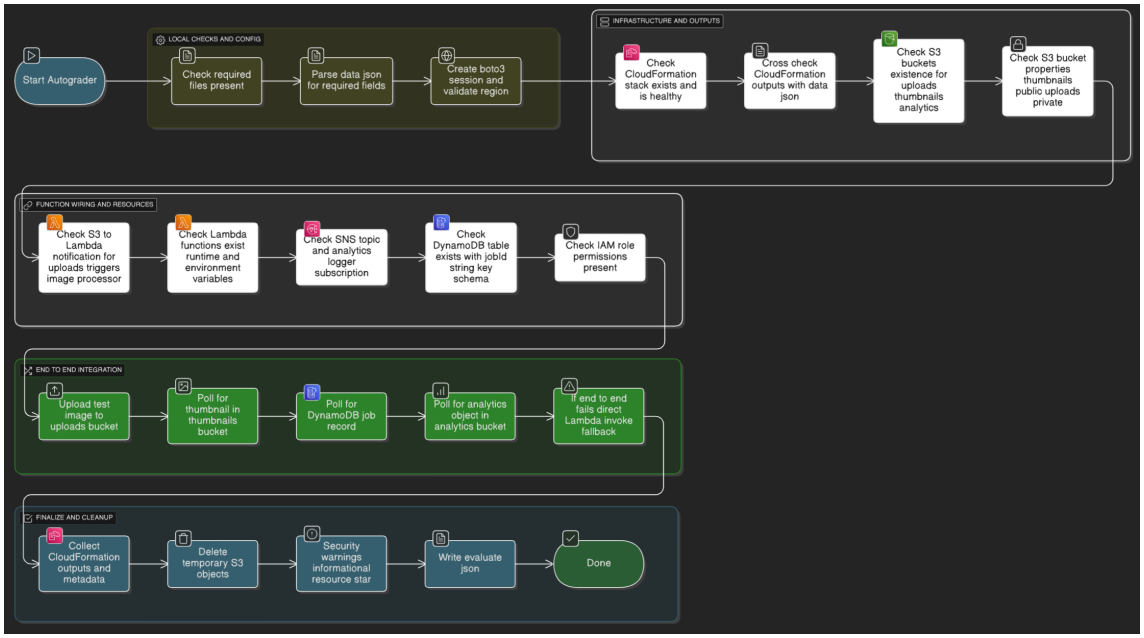


Figure 6.4: Planned flow of autograder verification for Activity 4.

### 6.5.3 Remarks

Activity 4 serves as the capstone exercise of the Serverless Framework Lab series. It transitions students from guided, instructor-led implementation to fully independent cloud deployment and validation. By building this image processing pipeline end-to-end, learners strengthen their grasp of event-driven design, Infrastructure-as-Code principles, and secure IAM configurations.

The environment refinements persistent workspaces, preinstalled toolchains, and reproducible evaluation make the learning process seamless. This approach prepares students to design, deploy, and troubleshoot real-world serverless architectures confidently.

For complete details, refer to the official **Activity 4 Problem Statement Document**.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

This guided project series — from Docker-based container fundamentals to the Serverless Framework v4 cloud applications — represents a complete, hands-on learning journey for mastering modern cloud-native architectures. Through sequential, activity-based progression, students learned how to:

- Containerize and orchestrate multi-service applications using Docker and Compose.
- Optimize image builds, apply best practices for caching, security, and deployment efficiency.
- Transition from containerized workloads to fully managed, event-driven architectures using AWS Lambda, S3, DynamoDB, and SNS.
- Automate provisioning and deployment through Infrastructure-as-Code with the Serverless Framework.

Each component of this series was designed to build upon the last, ensuring that learners not only understood the individual technologies but also how they interact cohesively within a complete system. By the final activity, students were independently deploying scalable serverless solutions — a skill directly transferable to real-world production environments.

The deliberate stepwise design — starting from foundational Docker concepts, progressing through system optimization, and culminating in serverless orchestration — ensured both conceptual depth and practical readiness. The instructor-led environment enhancements, such as preconfigured Docker images, consistent toolchains, and persistent

workspaces, played a crucial role in maintaining reproducibility and lowering setup barriers.

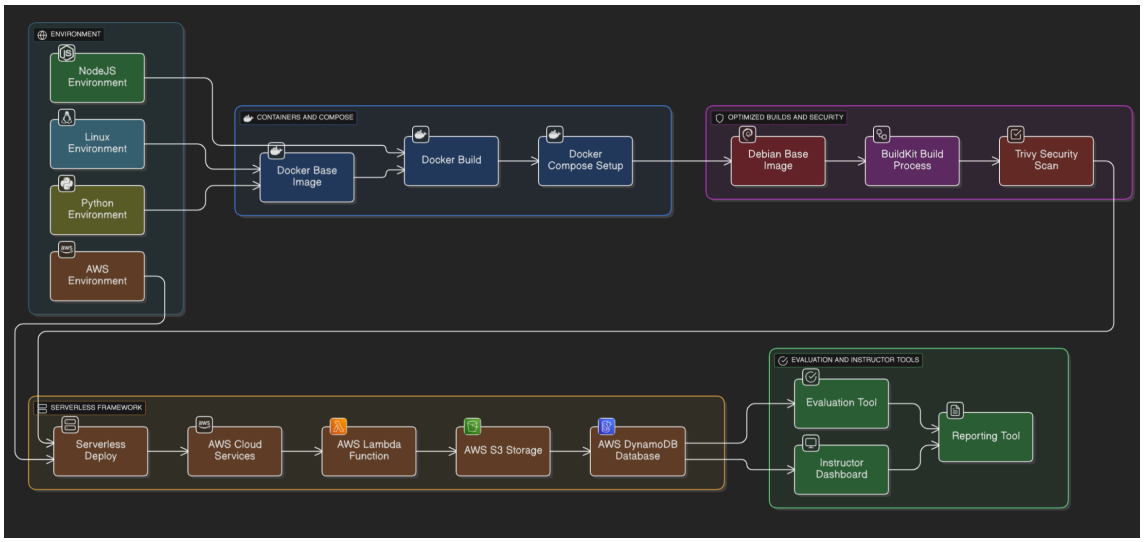


Figure 7.1: High-level summary of the end-to-end learning and implementation flow.

**Key Learning Outcomes:**

- Ability to design and deploy both containerized and serverless architectures.
- Practical understanding of AWS core services and event-driven design patterns.
- Experience in troubleshooting deployment issues, IAM permissions, and environment mismatches.
- Awareness of operational considerations such as cost-efficiency, maintainability, and scalability.

## 7.2 Future Work:

The next phase of this project aims to extend the current learning framework into more advanced domains of cloud computing, microservice architecture, and distributed systems. The objective is to study these concepts more deeply and design new hands-on laboratory exercises that make complex cloud architectures approachable and interactive for students.

- **Cloud Architecture Deep-Dive:** Extend the existing framework to cover high-availability architectures, including multi-region deployments, fault tolerance, and auto-scaling mechanisms. Students will explore how to distribute workloads efficiently using AWS Load Balancer (ALB/NLB) for routing and traffic management.

- **Microservices and API Gateways:** Develop labs around containerized microservices communicating via REST or gRPC, managed through AWS API Gateway or service mesh frameworks such as Istio. Emphasis will be placed on service discovery, distributed tracing, and secure communication between services.

- **Load Balancing and Scaling:** Introduce practical exercises using AWS Application Load Balancer (ALB) and Network Load Balancer (NLB) for routing traffic to EC2 or containerized microservices. Students will learn how to configure health checks, listener rules, and weighted routing for blue-green and canary deployments. This section will bridge the gap between container-level scaling and production-grade load management.

- **Advanced DevOps Practices:** Implement CI/CD pipelines integrating automated testing, static analysis, and canary deployment strategies using AWS CodePipeline and GitHub Actions. Future labs will emphasize observability through CloudWatch Metrics, X-Ray, and log aggregation for performance optimization.

- **Pedagogical Focus:** Each upcoming lab will continue to emphasize clarity, reproducibility, and incremental complexity, allowing students to internalize the underlying architectural rationale while gaining hands-on expertise.

## 7.3 Final Remarks:

The culmination of this guided lab series demonstrates how thoughtful environment design, progressive complexity, and structured validation can create a truly immersive learning experience. Students not only gain technical proficiency but also develop the engineering mindset required to design robust, maintainable, and scalable systems in modern cloud ecosystems.

This evolving framework — extending into advanced topics like load balancing, service orchestration, and observability — will further strengthen students' readiness for real-world cloud engineering roles. By combining theory with reproducible, hands-on practice, these labs establish a sustainable foundation for deeper exploration of distributed systems and microservice design.

# Chapter 8

# Appendix

## 8.1 Code Listings

This appendix presents the core implementation scripts and configuration files used across both the Docker Lab and Serverless Framework Lab components of the project. They illustrate environment preparation, evaluation automation, and integration of the cLab/-BodhiTree ecosystem with containerized and serverless workflows.

## 8.2 Docker Lab: Base Dockerfile

The following Dockerfile defines the consolidated lab environment used for all Docker-based activities. It ensures that all required tools such are preinstalled for seamless execution within the cLab environment.

```
1
2  # Stage 1: Use Ubuntu base image
3  FROM ubuntu:22.04 as ubuntu-stage
4
5  # Install all required packages in a single step
6  RUN yes | unminimize && \
7      apt-get update -y && \
8      apt-get install -y --no-install-recommends \
9        software-properties-common \
10       language-pack-en-base \
11       debconf-utils \
12       curl \
13       net-tools \
14       wget \
15       nano \
16       vim \
17       less \
```

```
      iproute2 \
      python3 \
      python3-pip \
      jq \
      man-db \
      openssh-client && \
    rm -rf /var/lib/apt/lists/*

# Install Paramiko via pip
RUN python3 -m pip install --no-cache-dir --upgrade pip && \
    pip3 install --no-cache-dir --break-system-packages paramiko


# Setup Environment variables
ENV INSTRUCTOR_SCRIPTS="/home/.evaluationScripts" \
    LAB_DIRECTORY="/home/labDirectory" \
    PATH="/home/.evaluationScripts:${PATH}" \
    TERM=xterm-256color

# Setup Directory Structure & Global Settings
RUN mkdir -p /home/labDirectory /home/.evaluationScripts && \
    echo "cd /home/labDirectory" > /root/.bashrc && \
    echo "alias ls='ls --color=always'" >> /root/.bashrc && \
    echo "rm -f \`find /home -type f -name \"._*\"\`" >> /root/.bashrc

# Default command: keep container alive
CMD [ "/bin/bash", "-c", "while :; do sleep 10; done" ]
```

Listing 8.1: Docker Lab Base Dockerfile.

## 8.3 Docker Lab: Sample Autograder Script (Activity 6)

The complete Python autograder script for Docker Lab Activity 6 was implemented in a moduler fashion to validate multi-stage build optimization, caching behavior, and image correctness within the cLab containerized environment. See from left to right manner.

## 8.4 Docker Lab: Environment Setup Script

This Bash script (`system-clean-init.sh`) initializes the lab workspace inside the cLab environment before each Docker-based exercise. It cleans previous data, extracts activity source archives, and prepares a consistent working directory for students.

```

```

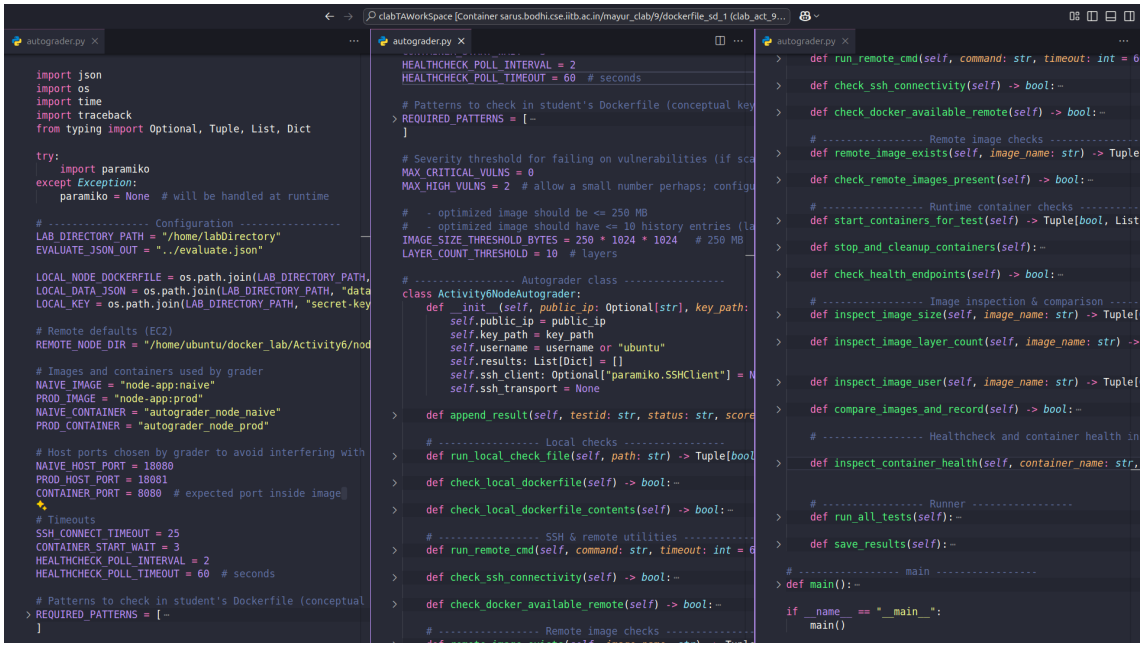

Figure 8.1: Source code modules of Activity 6 Docker Lab autograder script.

```bash
#!/bin/bash
set -euo pipefail

# --- Configuration ---
DATA_FILE="data.json"
SECRET_KEY_FILE="secret-key.pem"
EC2_USER="ubuntu"
TARGET_DIR="/home/ubuntu/docker_lab"
REMOTE_SCRIPT_PATH="/tmp/remote-cleanup.sh"

handle_error() {
  echo "ERROR: $1" >&2
  exit 1
}

[ -f "$DATA_FILE" ] || handle_error "JSON data file '$DATA_FILE' not found."
[ -f "$SECRET_KEY_FILE" ] || handle_error "Secret key file '$SECRET_KEY_FILE' not found."

if [ "$(stat -c "%a" "$SECRET_KEY_FILE")" != "400" ]; then
  echo "Fixing key permissions..."
  chmod 400 "$SECRET_KEY_FILE"
fi

```

```
25 if ! command -v jq &>/dev/null; then
26   handle_error "jq is required (install with sudo apt-get install jq)."
27 fi
28
29 echo "Reading public IP from $DATA_FILE..."
30 PUBLIC_IP=$(jq -r '."public-ip"' "$DATA_FILE")
31 [ -n "$PUBLIC_IP" ] && [ "$PUBLIC_IP" != "null" ] || handle_error "
     Invalid/missing public IP."
32
33 echo "Connecting to EC2 instance at $PUBLIC_IP..."
34
35 # --- Write the remote script locally ---
36 cat > remote-cleanup.sh <<'EOS'
37 #!/bin/bash
38 set -euo pipefail
39
40 TARGET_DIR="/home/ubuntu/docker_lab"
41
42 echo "Ensuring directory $TARGET_DIR exists..."
43 mkdir -p "$TARGET_DIR"
44
45 echo "Stopping all running containers..."
46 containers=$(docker ps -q)
47 if [ -n "$containers" ]; then
48   docker stop $containers || true
49 else
50   echo "No containers running."
51 fi
52
53 echo "Removing all containers..."
54 all_containers=$(docker ps -aq)
55 if [ -n "$all_containers" ]; then
56   docker rm $all_containers || true
57 else
58   echo "No containers to remove."
59 fi
60
61 echo "Removing all images..."
62 images=$(docker images -aq)
63 if [ -n "$images" ]; then
64   docker rmi -f $images || true
65 else
66   echo "No images to remove."
```

```
fi

echo "Pruning Docker Buildx cache..."
docker buildx prune -f || true

echo "Pruning entire Docker system (including volumes)..."
docker system prune -a --volumes -f || true
docker volume ls -q | xargs -r docker volume rm -f || true

echo "WARNING: Deleting all files in $TARGET_DIR"
rm -rf "$TARGET_DIR"/*

echo "Cleanup complete."
EOS

# --- Copy to remote ---
scp -i "$SECRET_KEY_FILE" remote-cleanup.sh "$EC2_USER@$PUBLIC_IP:
    $REMOTE_SCRIPT_PATH"

# --- Execute on remote ---
ssh -i "$SECRET_KEY_FILE" "$EC2_USER@$PUBLIC_IP" "bash
    $REMOTE_SCRIPT_PATH && rm -f $REMOTE_SCRIPT_PATH"

echo "Docker cleanup completed successfully on EC2 instance."

# --- Remove local temporary script ---
rm remote-cleanup.sh

# --- Copy product-api-flask.tar.xz to remote ---
echo "Copying Activity6.tar.xz to remote..."
scp -i "$SECRET_KEY_FILE" Activity6.tar.xz "$EC2_USER@$PUBLIC_IP:
    $TARGET_DIR/"

# --- Extract and remove tar inside remote ---
ssh -i "$SECRET_KEY_FILE" "$EC2_USER@$PUBLIC_IP" "tar -xJf $TARGET_DIR/
    Activity6.tar.xz -C $TARGET_DIR && rm -f $TARGET_DIR/Activity6.tar.
    xz"

echo "File transfer and extraction completed successfully."
```

Listing 8.2: system-clean-init.sh — Docker Lab Environment Setup Script.

## 8.5 Serverless Framework: Base Dockerfile

The Dockerfile below extends the Docker Lab image to support the Serverless Framework v4 environment. It preinstalls Node.js, AWS CLI v2, and the Serverless CLI, enabling Lambda deployments directly from within the lab container.

```
1
2  # Run (recommended, when host allows privileged containers):
3  #   docker run --rm -it --privileged -v /home/labDirectory:/home/
       labDirectory -e HOME=/home/labDirectory lab-env:latest
4  # Or (use host Docker daemon via socket, preferred if available on host
       ):
5  #   docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock -v
        /home/labDirectory:/home/labDirectory -e HOME=/home/labDirectory
       lab-env:latest
6
7  FROM ubuntu:22.04 AS base
8
9  ARG DEBIAN_FRONTEND=noninteractive
10 ARG NODE_MAJOR=20
11
12 ENV INSTRUCTOR_SCRIPTS="/home/.evaluationScripts" \
13     LAB_DIRECTORY="/home/labDirectory" \
14     HOME=/home/labDirectory \
15     TERM=xterm-256color \
16     LANG=C.UTF-8 \
17     LC_ALL=C.UTF-8 \
18     PATH="/home/.evaluationScripts:/usr/local/bin:/usr/local/aws-cli/v2
       /current/bin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
19
20 # Install base packages, Node (NodeSource), AWS CLI v2, Serverless,
       Python deps, and Paramiko
21 RUN set -eux; \
22     if command -v unminimize >/dev/null 2>&1; then yes | unminimize; fi
       ; \
23     apt-get update && apt-get install -y --no-install-recommends \
24       software-properties-common \
25       language-pack-en-base \
26       debconf-utils \
27       ca-certificates \
28       curl gnupg lsb-release unzip wget build-essential \
29       net-tools \
30       nano \
31       vim \
```

```
      less \
      iproute2 \
      python3 \
      python3-pip \
      python3-venv \
      python3-dev \
      jq \
      man-db \
      openssh-client \
      dbus \
      gnupg2 && \
    \
    # Install Node.js (NodeSource)
    curl -fsSL https://deb.nodesource.com/setup_${NODE_MAJOR}.x | bash
    - && \
    apt-get install -y --no-install-recommends nodejs && \
    npm install -g npm@latest && \
    \
    # Install AWS CLI v2
    curl -fsSL "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.
    zip" -o /tmp/awscliv2.zip && \
    unzip -q /tmp/awscliv2.zip -d /tmp && \
    /tmp/aws/install -i /usr/local/aws-cli -b /usr/local/bin && \
    rm -rf /tmp/aws /tmp/awscliv2.zip && \
    \
    # Python pip packages (including Paramiko) and other python bits
    python3 -m pip install --no-cache-dir --upgrade pip setuptools
    wheel && \
    python3 -m pip install --no-cache-dir paramiko && \
   \
    # Global npm tool (serverless)
    npm install -g serverless@latest || true && \
    \
    # cleanup apt lists (keep image small)
    apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

RUN python3 -m pip install --no-cache-dir boto3

# ---------- DELTA: install Docker Engine (daemon + CLI) and startup
    script ----------
RUN set -eux; \
  apt-get update && apt-get install -y --no-install-recommends \
    ca-certificates curl gnupg lsb-release gnupg2 dbus; \
```

```
71    mkdir -p /etc/apt/keyrings; \
72    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | gpg --
       dearmor -o /etc/apt/keyrings/docker.gpg; \
73    echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
       keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(
       lsb_release -cs) stable" \
74      > /etc/apt/sources.list.d/docker.list; \
75    apt-get update; \
76    apt-get install -y --no-install-recommends \
77      docker-ce docker-ce-cli containerd.io docker-compose-plugin; \
78    groupadd -f docker || true; \
79    mkdir -p /var/lib/docker /var/run/docker /var/log/docker; \
80    apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
81
82  # Add a simple startup script to attempt to run dockerd inside the
       container.
83  # It prints clear messages if dockerd fails (useful when runtime doesn'
       t allow starting daemon).
84  RUN cat > /usr/local/bin/start-dockerd.sh <<'EOF'
85  #!/bin/sh
86  set -eux
87
88  mkdir -p /var/log/docker
89
90  if command -v dockerd >/dev/null 2>&1; then
91    echo "Attempting to start dockerd..." >&2
92    # Start dockerd in background, log to file
93    dockerd --host=unix:///var/run/docker.sock >> /var/log/docker/dockerd
       .log 2>&1 &
94    DPID=$!
95    # wait up to 15s for socket
96    i=0
97    while [ ! -S /var/run/docker.sock ] && [ $i -lt 15 ]; do
98      sleep 1
99      i=$((i+1))
100   done
101   if [ -S /var/run/docker.sock ]; then
102     echo "dockerd started (PID=$DPID)" >&2
103     # Keep container alive for interactive use / debugging
104     exec tail -f /dev/null
105   else
106     echo "Warning: dockerd did not create /var/run/docker.sock. It may
       require --privileged or proper cgroup mounts at runtime." >&2
```

```
    echo "Inspect /var/log/docker/dockerd.log for details." >&2
    exec tail -f /var/log/docker/dockerd.log
  fi
else
  echo "dockerd binary not found" >&2
  exec tail -f /dev/null
fi
EOF

RUN chmod +x /usr/local/bin/start-dockerd.sh
# ------------------------------------------------------------------

# Create the lab directory and instructor scripts dir
RUN mkdir -p /home/labDirectory /home/.evaluationScripts && \
    echo "cd /home/labDirectory" > /root/.bashrc && \
    echo "alias ls='ls --color=always'" >> /root/.bashrc && \
    echo "rm -f \`find /home -type f -name \"._*\"\`" >> /root/.bashrc

# Optional: add same helpers to skeleton for any created user
RUN printf "cd /home/labDirectory\nalias ls='ls --color=always'\n" >> /
    etc/skel/.bashrc || true

WORKDIR /home/labDirectory

# Ensure root user (you may add non-root user later if desired)
USER root

# Attempt to start dockerd on container start (script will warn if
    runtime lacks privileges)
ENTRYPOINT ["/usr/local/bin/start-dockerd.sh"]

# Default keep-alive (ENTRYPOINT script handles keeping container up)
CMD [ "tail", "-f", "/dev/null" ]
```

Listing 8.3: Serverless Framework Lab Dockerfile.

## 8.6 Serverless Framework:Autograder Script(Activity4)

The following image shows the complete Python autograder used for the **Serverless Image Processing Pipeline (Activity 4)**. This script automatically verifies AWS resource creation, IAM permissions, CloudFormation outputs, and the end-to-end image processing flow. See from left to right.

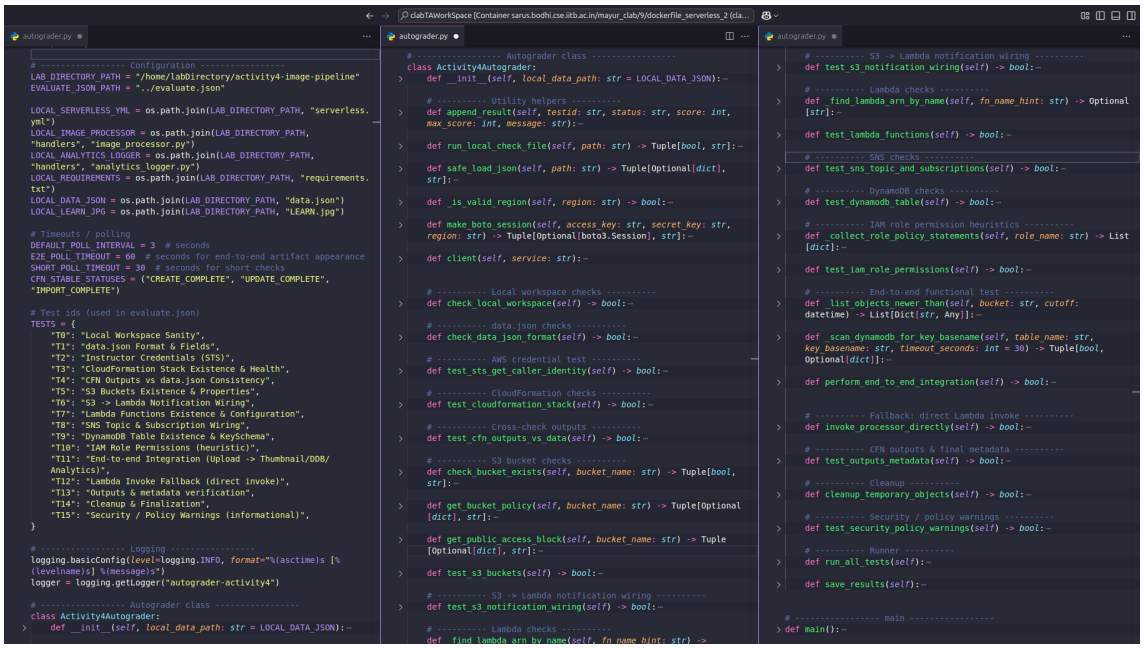

Figure 8.2: Source code modules of Activity 4 Serverless Framework autograder script.

# Acknowledgement

I would like to express my sincere gratitude to my supervisor, **Prof. Kameswari Chebrolu**, for her constant guidance, valuable feedback, and encouragement throughout the course of this project. Her insightful suggestions and clear vision were instrumental in shaping the direction and quality of this work.

I am also thankful to the **Department of Computer Science and Engineering, IIT Bombay**, for providing the academic environment and computational resources necessary for completing this project.

# References

[1] Bitovi, "What is docker." https://www.bitovi.com/academy/learn-docker/what-is-docker.html. Accessed: 10 October 2025.

[2] C. Weitz, "Kubernetes vs docker: What's the difference?." https://acecloud.ai/blog/kubernetes-vs-docker/, 2024. Accessed: 10 October 2025.

[3] K. group, "Serverless architecture explained." https://www.kofi-group.com/serverless-architecture-explained-in-10-minutes/. Accessed: 10 October 2025.