Application Management

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Application Management

Los Del DGIIM, losdeldgiim.github.io

Arturo Olivares Martos

Granada, 2025

Índice general

1. Application Lifecycle Management (ALM)

1.1. Application Lifecycle Management (ALM)

Creating an Application is not just installing it and updating it, it should cover much more aspects throughout its whole lifecycle, from the initial idea to its end of life. With that in mind, we define the following.

Definición 1.1 (Application Lifecycle Management (ALM))**.** ALM is the framework that defines the process of managing an application throughout its whole lifecycle, from the initial idea to its end of life. It integrates *people*, *processes* and *tools* to manage the application effectively and efficiently.

This framework lets manage the complexity of modern applications. Nowadays there are a lot of different people, named *stakeholders*, involved in the creation and maintenance of an application (Developers, Bussiness Analysts, Testers, Final Users, etc). ALM provides a structured approach to coordinate all these people and their tasks. This lets everyone know *what should they do at any moment*. This leads to overcome the typical "controled chaos" that usually happens in large projects.

Some aspects that are usually included in ALM are:

- Design & Development
- Continuous Integration
- Source Control & Configuration Management
- Quality Assurance
- Requirement Management

Its main goals are the following ones:

- Create fast high-quality products.
- Definition of tasks, roles and responsibilities.
- Knowing which tasks are being done, by whom and when.
- It improves the communication between teams.

It takes into account one big problem: *too much planning can have negative consequences*. If every single detail is planned, it can lead to a lack of flexibility and adaptability to changes. Therefore, when new changes are planned, they are usually not taken into account, leading to a worse final product.

In addition, the first, difficult step in ALM is to define the requirements of the application. It should be taken into account that many organizations are in a hurry to develop and release the application in order to be competitive in the market. With that in mind, a minimum set of requirements that gives them the competitve advantage should be defined. This firsy prototype should be developped and released as soon as possible. After that, new features can be added in future versions of the application.

1.2. Software Development Lifecycle (SDLC)

The Software Development Lifecycle (SDLC) is a methodology that defines the process of creating and maintaining software applications. It is a structured approach that covers all the phases of the software development process. It defines some guidelines and best practices to reduce future problems. It also helps to decide the responsibilities of each team member, so that everyone knows what they should do at any moment.

It should not be confused with the following concepts:

VS ALM SDLC is a part of ALM. While ALM covers the whole lifecycle of an Application (including retirement), SDLC focuses on the development phase.

VS System Development Lifecycle System Development Lifecycle also takes into account testing and using softwares from third parties (COTS (Commercial Off-The-Shelf)). It is really important to not blindly trust third-party softwares, as they can have security vulnerabilities or other problems that can affect the final product. There are ISO standards for Software and System Development Lifecycles.

The following subsections describe some concretes SDLC models. The number of phases of the SDLC can vary depending on the model used. Regarding documentation (which is usually not included into the phases), it is often overlooked (functional software is more important than comprehensive documentation), but it should ideally be complete.

1.2.1. Waterfall Model

The Waterfall Model is a linear and sequential approach to software development. It is the one that has been historically used the most.

Advantages It is easy to understand and manage. The phases do not overlap.

Disadvantages It is inflexible to changes. Once a phase is completed, it is difficult to go back to it. In addition, a working product is not available until the end of the process.

It should only be used when the requirements are well understood and unlikely to change during the development process. It is not suitable for complex or large projects where requirements may evolve over time.

Phases

1. Requirements & Analysis *What should the System do?*

   A good requirements list is essential for the success of the project. Test cases should be defined at this stage to ensure that the final product meets the requirements. They should be *relevant, valid and verifiable*. They are divided into Functional Requirements (what the system should do) and Non-Functional Requirements (how the system should be, e.g., performance, security, usability, etc).

2. Design & Architecture *How should the System be designed?* Online/Offline, etc.

   The system architecture is defined at this stage. It should let the requirements be implemented effectively and efficiently. One important aspect is the scalability of the system to a higher number of users or data.

3. Implementation & Coding *How is the System going to be coded?*

   The actual coding of the system is done at this stage. It should be taken into account that the knowledge of the different stakeholders can vary a lot.

4. Testing & Quality Assurance *Does the System meet the requirements?*

   Testing is so important that it should be done in parallel with the coding (Test-Driven Development and Continuous Integration). The final tests are usually done by a different team (QA Team) to ensure the objectivity of the tests. In really critical systems, formal verification techniques can be used to mathematically prove that the system meets its requirements.

5. Deployment & Maintenance *How do the deployment and updates work?*

   The system is deployed. Two aspects should be taken into account:

   - Continuous Deployment: The changes in the code should be automatically considered. Automatic tests should be done to ensure that the new code does not introduce new bugs.
   - Maintenance: A balance between new versions and bug fixing the current version should be found. Releasing new updates can be difficult depending on the type of application.

1.2.2. Agile Model

Scrum

Scrum is an Agile iterative and incremental framework for managing software development projects. There are three main roles:

1. Product Owner: Responsible for defining the product vision (client representative).
2. Scrum Master: Responsible for ensuring that the Scrum process is followed.
3. Development Team: Responsible for delivering the product increment.

The development process is divided into Sprints (usually 2-4 weeks long). Each Sprint has 4 main events:

1. Sprint Planning: Define the goals and tasks for the Sprint.
2. Daily Scrum: A short daily meeting to discuss progress and obstacles.
3. Sprint Review: Review the work completed and the work not completed.
4. Sprint Retrospective: Reflect on the past Sprint and identify improvements.

The main tools (artifacts) used in Scrum are:

- Product Backlog: List of all desired work on the project (made from the client's viewpoint).
- Sprint Backlog: List of tasks to be completed in the current Sprint.
- Increment: The sum of all the completed products, the result of the Sprint.

DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). Its main goal is to shorten the development lifecycle and provide continuous delivery with high software quality. It emphasizes collaboration, communication, and integration between development and operations teams.

Agile ALM

The agile ALM is a flexible and iterative SDLC approach that focuses on delivering value to the customer through continuous feedback and improvement. It follows the principles of Agile development.

$$\begin{gathered}\text{Individuals and interactions} \succ \text{Processes and tools} \\ \text{Working software} \succ \text{Comprehensive documentation} \\ \text{Customer collaboration} \succ \text{Contract negotiation} \\ \text{Responding to change} \succ \text{Following a plan}\end{gathered}$$

The principles of Agile ALM are:

1. Satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently.

4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals.
6. The most efficient method of conveying information is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. Everyone should maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity –the art of maximizing the amount of work not done– is essential[1].
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1.3. SLA

A Service Level Agreement (SLA) is a formal contract between a service provider and a customer that defines the level of service expected from the service provider. Some important metrics that are usually included in an SLA are:

- Operating Time: The time period during which the client has the right to use the service.
- Service Time: The time period during which the service provider is obligated work on resolving issues.
- Response Time: Maximum time for the service provider to *start* working on an issue after it has been reported[2].
- Recovery/Resolution Time: Maximum time for the service provider to *resolve* an issue after it has been reported.
- Availability: The percentage of time the service is available during the Operating Time[3].

$$\text{Availability} = \frac{\text{Total Minutes in Operating Time} - \text{Minutes of Downtime}}{\text{Total Minutes in Operating Time}} \times 100\,\%$$

[1] This principle is sometimes not understood. It just means that the simplest solution that meets the requirements should be implemented. It does not mean that the best solution should not be implemented.

[2] It should be noted that the period starts when the issue is reported only if it is during the Service Time; otherwise, it starts at the beginning of the next Service Time.

[3] It should be noted that the service can be unavailable during scheduled maintenances, which should be defined in the SLA.

There are 6 different availability levels commonly used, from VK 0 (less restrictive) to VK 5 (most restrictive). Each level defines the minimum availability percentage and the maximum allowed downtime per year.

- Regarding to failures, there are different metris that can be included in the SLA, such as:
  - Mean Time Between Failures (MTBF): The average time between two consecutive failures.
  - Mean Time To Repair (MTTR): The average time it takes to repair a failure.
  - Mean Time To Failure (MTTF): The average time until the first failure occurs.

    It is mostly used for systems that cannot be repaired (e.g., a light bulb) or that are not usually repaired (e.g., a hard drive), but replaced instead.

2. Version Control System (VCS)

Definición 2.1 (Version Control System (VCS))**.** A Version Control System (VCS) is a software tool that helps to manage changes to source code over time. It keeps track of every modification made to the code, also allowing to revert to previous versions if needed.

2.1. Types of VCS

There are three main types of VCS:

- Local VCS: It stores all the changes in a local database on the developer's computer. It is simple to use, but it does not provide collaboration features. One example is GNU RCS (Revision Control System).
- Centralized VCS: It uses a central server to store all the changes. Follows a client-server architecture.

  Advantages Clear control access, correct backup of big files and locking mechanism.

  Disadvantages Single point of failure, limited offline capabilities and the possibility of forgetting to release the locks.

  Only the original file and then each specific changes (deltas) are stored, not every version of the file. This saves space but depends on the whole set of deltas to reconstruct a specific version. One example is Subversion (SVN).
- Distributed VCS: It allows every developer to have a complete copy (including history) of the repository on their local machine. This provides better collaboration features and allows developers to work offline.

In the following section, we will focus on Distributed VCS, and specially on one of the most popular ones: Git.

2.2. Git

Git is a distributed version control system that is widely used in the software development industry. Opposite to the delta-based approach of the centralized VCS, Git uses a *snapshot-based* approach. It keeps a list of the whole snapshots of the filesystem at specific points in time. Almost every operation in Git is performed *locally*, which makes it really fast. Given that the history is stored locally, it allows

to work and to restore previous versions even when offline. It uses *Hashing* (SHA-1) to identify every single commit, ensuring the integrity of the codebase.

Git has some diffeent states for each file, which are stored in three different areas:

- Modified (Stored in the working directory): The file has been changed but not yet staged for commit (not registered in the repository).
- Staged (Stored in the staging area): The file has been modified and is marked to be included in the next commit.
- Committed (Stored in the local repository, `.git` folder): The file has been saved in the local repository.

Some important Git commands are the following ones:

- `git init`: Initializes a new Git repository (creates the `.git` folder).
- `git clone <repo_url>`: Clones an existing repository from a remote server to the local machine.
- `git add <file>`: Stages a file for commit. It also lets tracking new files and indicating when a file conflict has been resolved.

  Using the option `-A` stages all the changes (new, modified and deleted files).
- `git commit -m "message"`: Commits the staged changes to the local repository with a descriptive message.
  - `git commit --amend`: It modifies the last commit by adding the currently staged changes to it. It is useful when you forget to stage some changes before committing, or when you want to fix a mistake in the last commit (e.g., a typo in the code, a missing file, etc). It should be noted that the hash of the last commit will change after amending it. It should be used with the option `--no-edit` if you want to keep the same commit message, or with the option `-m "new message"` if you want to change the commit message.
- `git status`: Shows the current status of the working directory and staging area. Indicates which files are untracked, modified, deleted or staged for commit.

  Observación. All the files that are not tracked will always appear in red when using `git status`. Sometimes they are intended to be untracked (e.g., temporary files, build artifacts, etc). In that case, they can be added to the `.gitignore` file to avoid them appearing in the status. That will make Git ignore them.
- `git diff`: Shows the differences between files in different states (working directory, staging area, last commit). Has different options to compare specific states:
  - `git diff`: Working directory vs Staging area.

  - `git diff --staged`: Staging area vs Last commit.
  - `git diff HEAD`: Working directory vs Last commit.

- `git log`: Displays the commit history of the repository. Using the option `-p` shows the differences introduced in each commit.
- `git checkout <commit_hash>`: Switches to a specific commit, allowing to view the state of the repository at that point in time.

In addition, when referencing a commit, Git allows to use some special notations:

- `<commit_hash>^[<n>]` : Refers to the n-th parent of the specified commit (for instance, when merging, more than a parent may appear). If n is not provided, it defaults to 1, referring to the first parent.
- `<commit_hash>~[<n>]` : Refers to the n-th ancestor of the specified commit. It follows the first parent of each commit, so it is useful to refer to commits in a linear history. If n is not provided, it defaults to 1, referring to the immediate parent.

2.2.1. Git Bisect

In this section, we introduce a new command, `git bisect`, which is used to find the specific commit that introduced a bug or issue in the codebase. A whole section is needed, as it is a really useful command that can save a lot of time when debugging. It uses a binary search algorithm ($O(\log n)$) to efficiently narrow down the range of commits that may have introduced the bug. The workflow for using `git bisect` is as follows:

1. Initialize the bisect process.
   1.1 `git bisect start`: Initializes the bisect process.
   1.2 `git bisect bad`: Marks the current commit as bad (the commit where the bug is present).
   1.3 `git bisect good <commit_hash>`: Marks a specific commit as good (a commit where the bug is not present).
2. Locate the commit that introduced the bug.
   a) Git will automatically check out a commit in the middle of the range between the good and bad commits. The developer needs to test this commit to determine if it is good or bad.
   b) Based on the test result, the developer will mark the commit:
      - `git bisect good`: If the commit is good, it will narrow down the search to the range between this commit and the bad commit.
      - `git bisect bad`: If the commit is bad, it will narrow down the search to the range between this commit and the good commit.
   c) This process is repeated until Git identifies the specific commit that introduced the bug.

During the whole process, the following command are useful:

- `git bisect log`: Shows the log of the bisect process, including the commits that have been marked as good or bad.
- `git bisect visualize --oneline`: Visualizes the bisect process, showing the commits that are still not tested, and where the `HEAD` is currently located.

This whole process can be automated by using the following command:

- `git bisect run <script>`: This command automates the bisecting process by running a specified script that tests each commit. The script should return a zero exit code if the commit is good and a non-zero exit code (normally 1) if the commit is bad. This allows to quickly identify the bad commit without manual intervention.

3. End the bisect process.

   - `git bisect reset`: After finding the bad commit, this command is used to end the bisect process and return to the original state of the repository (the branch and commit that were checked out before starting the bisect). It is important to use this command to clean up the bisect state and avoid any confusion in future operations.

This command is used in the Exercise 7.4.1, so we refer to that exercise for a practical example of how to use `git bisect`.

2.2.2. Git LSF

Git Large File Storage (Git LFS) is an extension for Git that is designed to handle large files more efficiently. Given that Git is based on snapshots, storing large files directly in the repository can lead to repeated storage of the same file in different commits, which can quickly bloat the repository size. Git LFS solves this problem by replacing large files with lightweight references in the Git repository. Instead of storing the actual file content in the repository, Git LFS stores a pointer to the file in the Git repository and keeps the actual file content in a separate storage location. It is especially useful for binary files (e.g., images, videos, audio files, `pptx`, etc).

Some important Git LFS commands are the following ones:

- `git lfs install`: Installs Git LFS in the local repository.
- `git lfs track "<file_pattern>"`: Tracks files matching the specified pattern with Git LFS.
- `git lfs ls-files`: Lists the files that are being tracked by Git LFS.

2.2.3. Branching

Branching is a powerful feature in Git that allows developers to create separate lines of development within a repository. This enables multiple developers to work on different features or bug fixes simultaneously without interfering with each other's work. Each branch represents an independent line of development, allowing changes to be made in isolation. Once the changes in a branch are complete and tested, they can be merged back into the main branch (usually called `main` or `master`).

In order to explain branching, we need to define the concept of *HEAD*.

Definición 2.2 (*HEAD* Pointer)**.** The `HEAD` pointer is a reference to the current commit that the working directory is based on (therefore, it allows modifiers such as `HEAD^` or `HEAD~`). It indicates the current position in the repository's history. Some important aspects about the `HEAD` pointer are the following ones:

- `@` alone is a shorthand for `HEAD`, so it can be used interchangeably.
- `HEAD@{n}` refers to the position of `HEAD` n moves ago. For instance, `HEAD@{1}` refers to the previous position of `HEAD` before the last change (e.g., before the last checkout, merge, rebase, etc).

Some important commands related to branching are the following ones:

- `git branch <branch_name>`: Creates a new branch with the specified name that points to the current commit. With the option `-d`, it deletes the specified branch (if it has been merged).
- `git checkout <branch_name>`: Switches to the specified branch, updating the working directory to reflect the state of that branch. It then also updates the `HEAD` pointer to point to the new branch (the latest commit of that branch). With the option `-b`, it creates a new branch and switches to it in a single command.

As explained with the `HEAD` pointer, the names of branches are just pointers to specific commits (the latest commit of that branch). Therefore, modifiers such as `<branch_name>^` or `<branch_name>~` can be used to refer to commits in the history of that branch. `<branch_name>@{n}` can also be used to refer to the position of the branch pointer n moves ago.

2.2.4. Branch Integration

When the development in a branch is complete, it is often necessary to integrate the changes back into another branch (usually the `main` or `master` branch). There are two main methods for integrating branches in Git: merging and rebasing.

Observación. Given that understanding how the git tree will be after the integration is difficult, this tool can be used to visualize the effects of merging and rebasing.

Merging

The main command is the following one:

- `git merge <branch_name>`: Merges the specified branch into the current branch. It combines the changes from both branches, creating a new commit that represents the merged state.

There are two main strategies for merging branches:

- Fast-Forward Merge: If the current branch has not diverged from the branch being merged, Git simply moves the `HEAD` pointer forward to the latest commit of the merged branch. No new commit is created in this case.
- Recursive Merge: If the branches have diverged, Git creates a new commit that combines the changes from both branches. This new commit has two parent commits: one from each branch.

In the latter strategy, conflicts may arise if the same lines of code have been modified in both branches. Git will mark these conflicts in the affected files, and it is the developer's responsibility to resolve them manually before completing the merge. They can use `git status` to see which files have conflicts and need to be resolved. After resolving the conflicts, the developer can stage the changes and complete the merge by committing the changes.

Rebasing

This strategy uses the following command:

- `git rebase <branch_name>`: Reapplies the commits from the current branch on top of the specified branch. It effectively moves the entire branch to start from the latest commit of the specified branch. With the option `--continue`, it continues the rebase process after resolving conflicts.

After rebasing, the commit history appears linear, and therefore a simple fast-forward merge can be performed to integrate the changes into the target branch.

When rebasing, the commit history is rewritten, which can make it appear cleaner and more linear. However, the details about the mistakes committed during the development in the feature branch may be lost. In addition, it is dangerous to rebase branches that have already been pushed to a remote repository, as it can lead to confusion and conflicts for other developers working on the same branch, as their local copies will have a different history than the rebased branch.

2.2.5. Git Hooks

Git hooks are scripts that are automatically executed by Git before or after certain events, such as committing changes, merging branches, or pushing to a remote repository. They allow developers to customize and automate various aspects of their Git workflow. Git hooks are stored in the `.git/hooks` directory of a Git repository. Each hook is a separate script file that corresponds to a specific event. Some common Git hooks include:

- `pre-commit`: Executed before a commit is created. It can be used to perform checks on the code (e.g., run tests, check code style, etc) and prevent the commit if any issues are found.

- `post-commit`: Executed after a commit is created. It can be used to send notifications, update documentation, or trigger other actions.

- `pre-push`: Executed before changes are pushed to a remote repository. It can be used to run tests or perform other checks to ensure that the code being pushed meets certain criteria.

As a useful aspect, if any of the scripts ends with a non-zero exit code (denoting an error), if it was a pre- hook, the action that triggered the hook will be aborted. It garantees that certain conditions are met before proceeding with the action.

2.3. Distributed Git

Until this point, we have only talked about local operations. To collaborate with other developers, it is necessary to use remote repositories.

2.3.1. Remote Repositories

A remote repository is a version of the repository that is hosted on a remote server (GitHub, GitLab, Bitbucket, etc.). It allows multiple developers to collaborate on the same codebase. All developers should synchronize their local repositories with the remote repository to share changes and keep their code up to date. When more than one developer is working on the same codebase, conflicts may arise if two developers modify the same lines of code in different ways. Some important commands to interact with remote repositories are the following ones:

- `git remote add <name> <url>`: Adds a new remote repository with a specific name (e.g., `origin`).

- `git push <remote> <branch>`: Pushes the local commits to the specified remote repository and branch.

- `git fetch <remote>`: Fetches the latest changes from the specified remote repository without merging them into the local branch.

- `git pull <remote> <branch>`: Fetches and merges changes from the specified remote repository and branch into the local branch.

2.3.2. Collaboration Workflows

There are different collaboration workflows that teams can follow when using Git, depending on their preferences and project requirements. In the following, we describe three common workflows.

Centralized Workflow

The repository has a single central shared repository. All developers clone this repository, make changes in their local copies, and then push their changes back to the central repository. When more than one developer changes the same lines of code, conflicts may arise during the push operation, which need to be resolved before the changes can be successfully pushed.
This workflow is simple and easy to understand, making it suitable for small teams or projects with straightforward collaboration needs.

Integration Manager Workflow

In this workflow, there are two types of developers:

- Integration Manager: He is the responsible and manteiner of the project.
- Contributors: They suggest the integration managers the changes they want to make to the codebase.

As well as the two types of developers, there are also two types of repositories:

- Blessed Repository: It is maintained by the integration manager. It is the main repository where all the changes are eventually integrated, and it is considered the authoritative source of the codebase.
- Developper Repositories: They are maintained by the contributors. Each contributor has their own repository where they can make changes and experiment with new features. Each developper has their public repository and its local copy.

The workflow works as follows:

1. The integration manager creates the blessed repository, and make it public.
2. Each contributor clones the blessed repository to create their own developper repository (that is, a fork of the blessed repository).
3. Contributors make changes in their local copies and push them to their developper repositories.
4. When a contributor wants to suggest changes to the blessed repository, they email the integration manager asking him to make those changes (this is typically done through a pull request).
5. The integration manager adds the contributor's repository as a remote, fetches the changes, reviews them, and if everything is fine, merges them into their local blessed repository.
6. Finally, the integration manager pushes the updated blessed repository to the remote server, so that all contributors can access the latest changes.

This workflow allows for better control and review of changes, as the integration manager can carefully evaluate each contribution before integrating it into the main codebase. It is suitable for larger projects with multiple contributors.

Dictator and Lieutenants Workflow

This workflow is similar to the Integration Manager Workflow, but with a hierarchical structure. In this case, a new upper role is introduced: there are more than one integration managers, called *lieutenants*, and they report to a single *dictator* (the main integration manager). Each lieutenant is responsible for a specific area of the codebase and manages contributions related to that area.

The workflow works as follows:

1. Each contributor works on their own branch created for their feature or bug fix.
2. When needed, the lieutenants merge the branches of the contributors into their own master's branches.
3. When needed, the dictator merges the master's branches of the lieutenants into his own master's branch.
4. Finally, the dictator pushes the updated blessed repository to the remote server, so that all contributors can access the latest changes.

This workflow allows for better organization and management of contributions, as each lieutenant can focus on their specific area of expertise. It is suitable for large projects with multiple teams working on different aspects of the codebase. For instance, the Linux kernel development follows this workflow.

2.4. Git Internals

Git is built around a few fundamental concepts that enable its powerful version control capabilities. Understanding these concepts can help developers use Git more effectively and troubleshoot issues when they arise.

2.4.1. Objects

Git stores all its data in a simple key-value database, where the key is a SHA-1 hash of the content, and the value is the actual content. There are three main types of objects in Git:

- Blob: It represents the content of a file. It does not contain any *metadata* (e.g., filename, permissions, etc).
- Tree: It represents a directory. It contains references to blobs (files) and other trees (subdirectories), along with their names and *permissions*.
- Commit: It represents a snapshot of the repository at a specific point in time. It contains:
  - A reference to a tree object that represents the state of the repository at that commit (the root tree).

- References to parent commit(s) (the previous commit(s) in the history).
- Metadata such as the author, committer, timestamp, and commit message.

All of the objects are stored in the `.git/objects` directory. There, you can find subdirectories named with the first two characters of the SHA-1 hash, and inside those subdirectories, you can find files named with the remaining 38 characters of the hash. Git uses a combination of compression and delta encoding to store these objects efficiently, minimizing disk space usage. To see the objects stored in a Git repository, some useful commands are:

- `git cat-file -t <object_hash>`: Displays the type of the specified object (blob, tree, commit, etc).
- `git cat-file -p <object_hash>`: Displays the content of the specified object.
- `git ls-tree <tree_hash>`: Lists the contents of the specified tree object.
- `git show <commit_hash>`: Displays the details of the specified commit, including the commit message, author, date, and the changes introduced in that commit.

In all the cases, the `<object_hash>` is the SHA-1 hash of the object you want to inspect. Normally the first few characters of the hash are enough to uniquely identify the object.

2.4.2. Git Filesystem-Check

In the following section, a new command is presented, `git fsck`, which is used to verify the integrity of the Git repository. It checks the connectivity and validity of the objects in the repository, ensuring that there are no corrupted or missing objects. An important option is the following:

- `git fsck --lost-found`: Sometimes, files may be accidentally deleted or become unreachable due to various reasons (e.g., a commit is removed, a branch is deleted, etc). However, if those files were sometime staged, they are not completely lost, as their blob is still located in the `.git/objects` directory. However, detecting them can be difficult (there may be many objects in that folder).

  This option allows to find those lost objects and recover them. It creates two new directories, `.git/lost-found/commit` and `.git/lost-found/other`, where it places the lost commits and other objects (e.g., blobs), respectively. The files in those directories are named with their SHA-1 hash, and they can be inspected using the commands described in the previous section.

3. Build Engineering und Continuous Integration

The main aim of this chapter is to introduce the concept of Continuous Integration (CI) and its significance in modern software development. However, before diving into CI, a good understanding of build engineering is essential, as it forms the foundation for effective CI practices.

3.1. Build Engineering

Build engineering refers to the process of compiling source code into executable programs. This process should ideally be automated to ensure consistency, efficiency, and reliability. It is crucial in order to increase productivity and reduce human error during the build process.

To start with the building process, engineering teams typically start from available scripts (e.g., Ant, Maven, Make) that automate the build process. However, these scripts often need to be adapted to support Quality Assurance (QA) and deployment on production systems. This is where the role of a Build Engineer becomes vital.

There are usually two types of methologies for build engineering:

Using IDEs Integrated Development Environments (IDEs) provide built-in tools for building and managing projects. However, using them can lead to inconsistencies, as different developers may have different IDE configurations.

Command-Line Build Command-line build is usually preferred in professional environments. It allows for greater control and automation, ensuring that builds are consistent across different environments. It also lets the build scripts be version-controlled alongside the source code.

One important aspect of build engineering is the security of the build process. In order to ensure that the build process is secure, there are three concepts that need to be taken into account:

- **Automation**: The build process should be fully automated to minimize human intervention and reduce the risk of errors. These scripts should also follow the "Failing Fast" principle, which means that if an error occurs during the build process, it should stop immediately and report the error.

- **Secure Supply Chain**: The build process should ensure that all dependencies and components used in the build are secure and trustworthy. This includes verifying the integrity of third-party libraries and tools. Isolated build environments (e.g., using containers) can help mitigate risks associated with compromised dependencies.
- **Secure Trusted Base**: In the event of cyberattacks, it is crucial to accurately identify the software that has been compromised. This includes knowing which version of the software was deployed and whether it was deployed correctly. Methods for achieving this include using version numbers, hash functions, and creating a Manifest file that contains all configuration parameters.

3.1.1. GitHub Actions

In thuis building process, CI tools are used to automate the build and testing of code changes. One popular CI tool is GitHub Actions, which allows developers to create custom workflows that are triggered by specific events in a GitHub repository. In a repository, workflows are defined in YAML files located in the `.github/workflows/` directory. The triggers are specified using the `on` keyword, and include events such as `push`, `pull_request`, etc. These workflows can include various jobs, such as building the code, running tests, and deploying the application. An example of a simple GitHub Actions workflow that builds and tests a `C++` project using `CMake` is shown in the Source Code 1.

It should be noted that GitHub Actions and Git Hooks are different concepts. While Git Hooks are scripts that run locally in a developer's machine before or after certain Git events, GitHub Actions are workflows that run in the cloud on GitHub's servers in response to events in a GitHub repository. Git Hooks are not suitable for CI/CD processes, as they are not shared among team members and cannot be easily integrated with other tools and services.

3.2. Continuous Integration (CI)

The concept of Continuous Integration (CI) revolves around the idea of frequently integrating code changes into a shared repository. During the normal development process, this integration is not done frequently enough, leading to integration problems and conflicts when multiple developers work on the same codebase. The aim is to continuously integrate code changes with every commit, ensuring that the code is compilable and that the executable passes all tests.

In order to achieve CI, apart from the agreement among developers to follow this practice, the following are required:

- A version control system (e.g., Git) to manage code changes and facilitate collaboration among developers.
- An automated build script that compiles the code and runs tests.
- CI server (e.g., Jenkins, Bamboo) that monitors the version control system for changes, triggers the build process, and stores the results.

```yaml
name: C++ CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Set up CMake
      uses: jwlawson/actions-setup-cmake@v1
      with:
        cmake-version: '3.18.4'
    - name: Build
      run: |
        mkdir build
        cd build
        cmake ..
        cmake --build .
    - name: Test
        run: |
            cd build
            ctest --output-on-failure
    -
```

Código fuente 1: Example of a GitHub Actions workflow for building and testing a C++ project using CMake.

- An automated deployment of the software to a test environment.

A lot of the stakeholders in a software project benefit from CI, including developers (who get immediate feedback on their changes), QA teams (who can run automated tests) and project managers (who can get stadistics on build health and code quality).

For a good CI practice, really frequent commits are necessary. With CI, with each commit, the code is integrated, built, and tested automatically[1]. This helps to identify and fix integration issues early, reducing the risk of conflicts and bugs. A good practice is to firstly pull the latest changes from the main branch, resolve any conflicts locally, and then push the changes to the shared repository. In addition, the changes should be small and with a low complexity, making it easier for the other developers to solve the possible conflicts.

All this building process should be carried out in the "Build Farm", which is a dedicated environment for building and testing the software. It should be administrated separately from the development environment to ensure consistency and reliability. The build farm should also be scalable to handle multiple builds simultaneously, especially in large projects with many developers. If desired, developers should also be allowed to build and test their changes locally to reduce the load on the build farm and get faster feedback.

3.2.1. CI and Branches

CI and branching strategies do not fit well together, as branches are by nature changes in the code that should not yet be integrated into the main codebase. In order to mitigate this, the number of branches should be minimized and the changes in the master branch should at least once a day be merged into the feature branches. In addition, the lifetime of branches should be considered:

- Most branches should be short-lived, lasting only a few days to a week. This minimizes the risk of conflicts and integration issues.

- However, sometimes long-lived branches are necessary, for example, when it is not still clear which features will be included in the software release, and one will be later merged into the main branch. It should be clear from the beginning that late-binding always carries risks.

3.2.2. Testing

Without testing, CI can only ensure that the code compiles successfully. To ensure that the code behaves as expected, automated tests should be included in the CI process. Appart from specific types of tests (e.g. code quality tests, security tests), there are three main types of tests that should be considered:

[1]This can mean an overload in the early stages. A possible solution is "Nightly Builds", where the build process is run once a day, usually at night.

Unit Tests : These tests focus on small units of code, such as functions or methods, to ensure they work correctly in isolation, and are usually from a developer's perspective. No database or external systems should be involved. The whole application should not be started. Usually less than 10 minutes are required to run all unit tests.

Component Tests : More than one unit is tested together, possibly involving databases or external systems. The whole application is still not started.

Acceptance Tests : These tests validate the entire application against the requirements, and are usually from an end-user's perspective. The whole application is started. Usually more than a day is required to run all acceptance tests.

The CI process should also consider the time all this testing takes. If it takes too long, developers may have to wait too much time to get feedback on their changes, or while the tests are running, they may continue working on other tasks, leading to more integration issues later. A possible solution is to have a "Smoke Test Suite" that runs a subset of the tests that can give quick feedback on the most critical functionalities of the application, and only run the full test suite at specific times (e.g., nightly).

4. Deployment Strategies and DevOps

To deploy an application means to make it available for use. This process is critical, as it ensures that the application is accessible to users in a reliable and efficient manner.

Afterwards we will explore the Deployment Pipeline, but first it should be considered the risky *first deployment*, because it has some particularities that will not be present in the rest of deployments. As it has already been explained in previous chapters, only a small prototype should be deployed at this stage, to show the basic functionality of the application to the users. An IT-environment should be prepared for this deployment, being as similar as possible to the final production environment (same operating system, same installed software, similar hardware, etc.). This will help to identify potential issues that may arise during the deployment process.

As it was with the integration phase, *automatization* is key to ensure a smooth and efficient deployment process. Every step should be scripted and with self-testing capabilities, to minimize human errors and ensure consistency across deployments. Documentation and verification that the deployment process is working as expected is also crucial. Therefore, some aspects to avoid are:

- Manually performing deployment steps.
- Deploying only when the entire development is complete.
- Manual configuration management of production environments.

4.1. Deployment Strategies

There are different strategies for deploying applications, and the choice of strategy depends on various factors such as the size and complexity of the application, the frequency of updates, and the tolerance for downtime. There are two important metrics that should be taken into account:

- Down Time: The period during which the application is unavailable to users due to deployment activities. It is important to minimize downtime to ensure a good user experience and maintain service availability.

- Rollback Time: The time it takes to revert to a previous stable version of the application in case of issues during deployment. A fast rollback time is crucial to minimize the impact of failures and ensure service continuity (disaster planning).

  It is important to firstly backup the status (database, data systems, etc.) that the application has changed before rolling back, in order to avoid data loss.

The deployment strategies can be categorized based on their approach to downtime during the deployment process.

4.1.1. Non-Zero Downtime Releases

In this strategy, the application is taken offline during the deployment process, resulting in downtime for users.

Recreate Deployment

In this strategy, the existing version of the application is completely stopped and removed before deploying the new version. This approach is simple and straightforward, but it results in downtime for users during the deployment process.

4.1.2. Zero-Downtime Releases

Zero-downtime releases, also known as Hot Deployment, should change instantly between application versions without interrupting the service for users. Easyly changing the resources (Databases, Servers, etc.) that the application is using is key to achieve this, which can be achieved by changing the URI (Uniform Resource Identifier) that the application is pointing to. Some strategies to achieve zero-downtime releases are the following.

Ramped Deployment

The new version is gradually deployed in every server, one by one. The old version is still running in the servers that have not been updated yet, and therefore there is no downtime for users. Once the new version is deployed in a server, the old version is stopped and removed from that server. This process continues until all servers are updated to the new version.

Blue-Green Deployment

There are two versions of the application: the current production (green) and the new version to be deployed (blue). These can be hosted on separate environments (e.g., different servers or cloud instances) or in the same environment (e.g. two different processes running on the same server). Switching between versions is done by simply switching in the router (in less than a second).

Problems can however be caused by databases, because the new version may require a different database schema. To avoid this, during the migration the application is set to read-only.

Canary Releasing

In this strategy, the new version of the application is rolled out to a small subset of users (the canary group) while the majority of users continue to use the old version. This allows for monitoring the performance and stability of the new version in a real-world environment before fully deploying it to all users. If any issues are detected, the deployment can be halted or rolled back without affecting the entire user base. It also allows to gather information about the new version from real users (e.g. if it generates more revenue).

A/B Testing

This strategy involves deploying two different versions of the application (version A and version B) to different subsets of users. This allows for testing and comparing the performance, user experience, and other metrics of the two versions in a real-world environment. Based on the results, the better-performing version can be fully deployed to all users. This strategy differs from canary releasing in the way the users are selected, as in A/B testing the users are carefully selected (depending on country, age, etc.) to ensure that the results are statistically significant, while in canary releasing the users are randomly selected.

Shadow Deployment

The new version of the application is deployed alongside the old version, but it does not receive any user traffic. Instead, it receives a copy of the user traffic that the old version is receiving, allowing for testing and monitoring the new version in a real-world environment without affecting users. This strategy allows to gather information about the new version from real users, while minimizing risks.

4.1.3. Emergency fixes

Despite the amount of testing and precautions taken, it is possible that some bugs or vulnerabilities are discovered after deployment. In such cases, the fixes should also go through the deployment pipeline to ensure that they are properly tested and validated before being released to production. This is usually not done, just fixing the issue directly in production, but this can lead to:

- Introducing new bugs while fixing the issue, known as *Regression Bugs*.
- The system could be in an unknown state after the fix (e.g., inconsistent data), as it was neither tested nor committed properly.

Therefore, having short deployment times is even more important. In addition, when a bug is detected the severity of the issue should be evaluated, and rolling back to a previous version should also be considered.

4.2. Deployment Pipeline

A Deployment Pipeline is an automated process that takes code changes from development to production. Before analysing its phases, it is important to explain

some good practices that should be followed when implementing a deployment pipeline:

1. Binary files should only be built once and then promoted through the different stages of the pipeline (e.g., from testing to staging to production). They should be secured with hashes.
2. The deployment process should be similar in every environment (development, testing, staging, production).
3. Smoke-Tests which verify that the application, database, and external services are running correctly should be performed after each deployment.
4. Testing environments should closely resemble the production environment.
5. Each code change should go through the entire pipeline to avoid regression bugs.
6. If any phase in the pipeline fails, the entire process should stop and the team should address the issue immediately.

4.2.1. Phases of a Deployment Pipeline

A typical deployment pipeline consists of the following phases:

1. Commit Stage.
2. Automated Acceptance Test Stage.
3. Manual Test Stage.
4. Release Stage.

Commit Stage

In this phase, where the code is builded and some automated tests are performed (usually unit tests and some acceptance tests), the principles of CI are applied. It should ideally last 5-10 minutes. This phase is crucial, and its implementation leads to significant improvements in software quality and team productivity.

Automated Acceptance Test Stage

The unit tests are usually not enough, and therefore more extensive automated acceptance tests should be performed in this phase. They should be described without technical details or terms, but from the user's perspective. They have an specific structure:

- **Given** some initial context (the preconditions).
- **When** an event occurs (the user action).
- **Then** ensure some outcomes (the postconditions).

Manual Test Stage

Some tests cannot be automated, and therefore they should be performed manually in this phase. Examples of such tests are the following:

- *Look & Feel Testing*: Ensures that the application meets the desired aesthetic and usability standards.
- *Worst Case Testing*: Evaluates the application's performance and stability under extreme conditions (e.g., for instance, the application is closed while performing a critical operation).

Release Stage

In this final phase, the application is deployed to the production environment. It should be as automated and as easy as possible, to minimize human errors and ensure consistency across deployments. Monitoring and alerting mechanisms should be in place to detect any issues that may arise after deployment.

4.2.2. Deployment of User-Installed Software

This process differs from the one described above, as the software is installed and updated by the users themselves. Some important aspects to consider are the following:

- Crash Reporting from the users should be implemented.
- Roll back should be also possible.
- Mantaining old versions is time-consuming, so ideally everyone should have the same version. In order to achieve that, updates should be downloaded and installed in the background automatically.

4.2.3. Modern Deployment Practices

In these last years, new practices have emerged to improve the deployment process even further. Some of these practices are the following:

- Progressive Delivery: This practice involves gradually (1 %, 5 %, 25 %, 50 %, 100 %) rolling out new features to users, allowing for monitoring and feedback at each stage before a full release. Canary Releasing and Blue-Green Deployment are combined, and the deployment is automatically paused or rolled back according to the monitoring results.
- Feature Flags: This technique allows developers to enable or disable specific features in an application without deploying new code. It allows dark launches (where a feature is deployed but not yet visible to users) and A/B testing (where different users see different versions of a feature to evaluate its performance) while minimizing risks. The *Flag debt* (the accumulation of unused or outdated feature flags) should be managed properly to avoid code complexity. There are some variations:

  - Release Flags: Used to control the release of new features.
  - Kill Switches: Used to quickly disable a feature in case of issues.
  - Permission Flags: Used to enable features for specific user groups.
  - Experiment Flags: Used for A/B testing.

- GitOps: This practice is based on the fact that the entire system's desired state is stored in a Git repository (*Git is the single source of truth*). Instead of deploying (manually or automatically) the application, the server directly pulls the changes from the Git repository and deploys them.

- Monitory and Observability: Appart from the typical monitoring (tracking predefined metrics as CPU, RAM, etc.), observability focuses on understanding the internal state of the system based on the data it produces (logs, metrics, traces). This allows not only to understand that an error has occurred, but also why it has occurred and, hopefully, how to fix it.

4.3. Continuous Deployment (CD)

Continuous Deployment (CD) is a software development practice that lets the software be constantly deployed to production automatically, without human intervention. In order to achieve CD, CI is required, and a robust deployment pipeline with extensive automated testing is essential to ensure that only high-quality code reaches production. As happened with CI, CD prefers the changes to be small and incremental, as they are easier to test and deploy.

Some of its benefits include an increased reliability, fast deployment times and major competitiveness, as time is usually a critical factor in the software industry.

4.3.1. Continuous Delivery

Although CD is useful, it focuses on deploying every change to production, which may not be suitable or desired in all scenarios. Therefore, Continuous Delivery is often preferred, where software sould always be in a deployable state, but the actual deployment to production is a manual decision. This allows for more control over when and how changes are released, allowing Feature Toggles to be used to enable or disable features as needed.

Observación. Both CD and Continuous Delivery focus on automating the deployment process on the production environment, while CI focuses on automating the build and testing processes in the earlier stages of development, in a testing environment.

4.3.2. Rapid Incremental Deployment

Rapid Incremental Deployment is a strategy that uses the Agile Principles also to the deployment process, starting with a small and simple deployment, and then iteratively improving and expanding it based on feedback and learning. It makes

possible not stopping the development process until the deployment process is fully implemented, as it allows to deploy a small prototype at the beginning, and then iteratively improving it. This also helps convinging pessimistic stakeholders, as they can see the benefits of the deployment process early on and provide feedback to improve it.

4.4. DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the development lifecycle and provide continuous delivery with high software quality. It aims to improve collaboration and communication between development and operations teams[1].

This approach is needed, because both teams have different goals and viewpoints.

- **Development Team**: Focuses on delivering new features and updates quickly to meet user needs and stay competitive in the market.
- **Operations Team**: Prioritizes system stability, reliability, security...

DevOps practices are based on the following principles:

- **Two Pizza Theory**: Teams should be small enough to be fed with two pizzas (8-10 people), to enhance communication and collaboration. This is not always possible, and it should also be taken into account that too many small teams can lead to coordination issues.
- **Experts Silos are eliminated**: Instead of having separate teams for development, testing, deployment, and operations; cross-functional teams are formed where members have diverse skills and responsibilities. This promotes collaboration and shared ownership of the entire software lifecycle. This way, bottlenecks caused by waiting for expert teams to perform specific tasks are avoided.
- **Avoiding Volleyball Games**: In traditional development processes, tasks are often passed between the development and operations teams, blaming each other for issues. In DevOps, the focus is on collaboration and shared responsibility, avoiding this back-and-forth blaming.
- **Employees are trusted and empowered**: Team members are given the autonomy to make decisions and take ownership of their work, not having to ask for permission for every little change and avoiding delays.

4.4.1. RACI Method

The RACI method is a responsibility assignment matrix that helps to clarify roles and responsibilities within a project or organization. It has four categories:

[1]This is usually used to justify bad practices, as letting developers manage the production environment.

- **Responsible**: The person or team responsible for completing a task or making a decision.

- **Accountable**: The person who is ultimately accountable for the task or decision (specially in the comercial or juridical aspects). Only one person can be accountable for each task or decision, and without the accountable person, the task or decision will not be completed.

- **Consulted**: The person or team that provides input or expertise for a task or decision. They are usually consulted before a decision is made or a task is completed. Two-way communication is required.

- **Informed**: The person or team that needs to be kept informed about the progress or outcome of a task or decision. They are usually informed after a decision is made or a task is completed. One-way communication is sufficient.

There are usually two matrices, the initial matrix (that covers the first configuration of the system) and the ongoing matrix (that covers the maintenance and updates of the system). Usually the four categories are divided between the Clients, the Development Team and the Operations Team.

4.5. Deployment with Containers Technology

During the last years, containers have become a popular technology for deploying applications. A *container* is a process that runs in a host operating system, isolated from other processes and containers, with its own filesystem, network interfaces, and resource limits. In a container, the application and its dependencies are packaged together, ensuring that it runs consistently across different environments. This increases the portability of applications and helps developping software. Some of the most popular containerization platforms are Docker and Kubernetes.

4.5.1. Containers VS Virtual Machines

Containers and virtual machines (VMs) are both technologies that provide isolation and resource management for applications, but they do so in different ways. VMs run a full operating system on top of a hypervisor, which abstracts the underlying hardware. Each VM has its own kernel and resources, making them more resource-intensive and slower to start compared to containers. On the other hand, containers share the host operating system's kernel and resources, allowing them to be more lightweight and faster to start.

4.5.2. Isolation Measures

In this section, topics as `chroot`, namespaces, or `cgroups` will be briefly explained, as they are the basis of containerization technology.

Command `chroot`

The Unix `chroot` command changes the apparent root directory for the current running process and its children. This creates a confined space (a *chroot jail*) where the process can operate, isolating it from the rest of the file system. However, it is not a complete isolation mechanism, as they can still see all the processes running in the host system, no network isolation is provided, and if the process has root privileges, it can escape the jail. It is used as follows: `chroot <new_root_directory> <command>`. For example, `chroot /home/user/jail /bin/bash` would start a bash shell with the root directory set to `/home/user/jail`.

Command `pivot_root`

The `pivot_root` command is a Linux system call similar to `chroot`, but it provides a more complete isolation mechanism. It moves the current root filesystem to a new location and mounts a new root filesystem in its place. This allows for better isolation, as the process cannot see the original root filesystem. It is what Docker uses. It is used as follows: `pivot_root <new_root_directory><old_root>`. For example, `pivot_root /home/user/new_root /home/user/old_root` would move the current root filesystem to `/home/user/old_root` and set `/home/user/new_root` as the new root filesystem.

Namespaces

Namespaces are a feature of the Linux kernel that provides isolation for various system resources. There are several types of namespaces, each isolating a specific resource:

- **PID Namespace**: Isolates process IDs, allowing processes in different namespaces to have the same PID. A child process will have its PID in its father namespace and a different one in its own namespace.

Control groups - `cgroups`

Control Groups (`cgroups`) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. It is needed in a containerization environment to ensure that containers do not consume more resources than allocated, which could lead to performance degradation or system instability. Let's remark some important folders and files related to `cgroups`:

- `/sys/fs/cgroup/`: This is the main directory where `cgroups` are organized. Inside this directory, apart from some general files, there are subdirectories for each type of processes. The two main subdirectories are:
  - `/sys/fs/cgroup/system.slice/`: This directory contains `cgroups` for system services managed by the init system (e.g., systemd). For instance, the `docker.service` `cgroup` defines the resource limits for the Docker service and is located in `/sys/fs/cgroup/system.slice/docker.service`.

  - `/sys/fs/cgroup/user.slice/`: This directory contains `cgroups` for user processes.

- `cpu.max`: This file defines the maximum CPU time that a `cgroup` can use. It consists of two values: the first one is the quota (the total amount of CPU time that the `cgroup` can use in a given period), and the second one is the period (the length of the time period in microseconds). For example, if `cpu.max` contains "50000 100000", it means that the `cgroup` can use up to 50 % of the CPU time.

- `memory.max`: This file defines the maximum amount of memory that a `cgroup` can use. If the processes in the `cgroup` exceed this limit, they will be killed by the kernel. For example, if `memory.max` contains "512M", it means that the `cgroup` can use up to 512 megabytes of memory.

4.5.3. Docker

Docker is a popular containerization platform that simplifies the process of creating, deploying, and managing containers. It has become a standard tool in the DevOps world due to its ease of use and powerful features. Some important aspects should be defined:

- **Docker Image**: A Docker image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files.

- **Docker Container**: A Docker container is a runtime instance of a Docker image. It is an isolated environment where the application runs, sharing the host operating system's kernel but with its own filesystem and resources.

- **Docker Registry**: A Docker registry is a storage and distribution system for Docker images. It allows users to store and share their images with others. The most popular public registry is Docker Hub, but there are also private registries available.

- **Dockerfile**: A Dockerfile is a text file that contains a set of instructions to build a Docker image. It defines the base image, the application code, dependencies, and any necessary configuration.

- **Docker Compose**: Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to use a YAML file to configure your application's services, networks, and volumes, making it easier to manage complex applications with multiple containers.

Some important Docker commands are the following:

- `docker build <path>`: Builds a Docker image from a Dockerfile. With the `-t` option, a name can be given to the image.

- `docker run <image>`: Runs a Docker container from a specified image.

  - `-d`: Runs the container in detached mode (in the background).
  - `-p <host_port>:<container_port>`: Maps a port from the host to a port in the container, allowing access to the application running inside the container.
  - `--name <container_name>`: Assigns a name to the container for easier management.
  - `--rm`: Automatically removes the container when it exits, keeping the system clean.

- `docker ps`: Lists all running containers. With the `-a` option, it lists all containers, including those that are stopped.
- `docker stop <container_id>`: Stops a running container.
- `docker rm <container_id>`: Removes a stopped container.
- `docker rmi <image_id>`: Removes a Docker image.

Some aspects are different in the integration and deployment phases when using Docker:

- After the CI server, an app would normally just be deployed as explained in this chapter.
- With Docker, after the CI server builds and tests the application, it creates a Docker image that is pushed to a Docker registry. In the deployment phase, the Docker image is pulled from the registry and run as a container in the target environment.

5. Secure Deployment and CA Case Study

As it has already been explained in the previous chapters, testing is a key part of the software development lifecycle. However, it is useless if an atacker can easily compromise the deployed application. Therefore, it is essential to ensure that the deployment process is secure and that the application is protected against common threats. There are two main types of threats that need to be considered:

- **Malicious Adversaries:** They are whether malicious insiders or external attackers that impersonate legitimate users to gain unauthorized access to the system. They are indeed malicious.
- **Benign Insiders:** They are legitimate users that may unintentionally compromise the system's security due to lack of knowledge or carelessness. They are not malicious, but their actions can still pose a threat to the system.

There are some best practices that can be followed to ensure a secure deployment:

- Code Reviews: The code should be reviewed by multiple developers to ensure that it is secure and free of vulnerabilities. It also helps to share the knowledge and improve the overall quality of the code. Moreover, the concept "treat configuration as code" should be applied, so configuration files are also reviewed.
- Secrets: Secrets such as passwords, cryptographic keys, and authorization tokens should be stored securely using Key Management Systems (KMS), and should never be hardcoded in the source code or pushed to version control systems.
- Automatization: It should be done in order to remove human error from the deployment process. It also improves security, as helps avoiding attackers to introduced malicious code during the deployment.
- Shift Left: Security should be integrated into the development process from the very beginning, rather than being an afterthought. This will save time and money.
- Builds: The process of building the application should have three main properties:
    - Hermetic: All the inputs (source code, compiler, libraries, etc.) should be specified and controlled. The external dependencies should be fetched

from trusted sources and should be versioned and hashed to ensure their integrity.

- Reproducible: The same inputs should always produce the same (bit by bit) outputs. Hermetic builds help achieving this property, and it helps verifying the origins of the deployed code.
- Verifiable: The origin of the build should be verifiable, ensuring that it was built from the intended source code.

5.1. Binary Provenance

In order to achieve builds with these properties, the concept of "binary provenance" is used. It lets trace back the binary to its source code, build process, and environment. A first aproach to achieve this is:

- There is a build system that, after building the application, produces a *build recipe* that contains all the information needed to reproduce the build, including the source code version, compiler version, build flags, and dependencies. The build recipe is then signed with a private key to ensure its authenticity and integrity, and the output is both the binary and the signed build recipe.

However, in some organizations the build system may execute all types of commands, so attackers could exploit this to introduce malicious code during the build process. To mitigate this risk, user-commands should be executed in an environment with limited privileges (no access to the keys) and a secured HTTP connection should be used to avoid man-in-the-middle attacks. Therefore, a more robust aproach to achieve binary provenance is:

- The build system is divided into two parts: a *orchestrator* and a *worker*. The orchestrator is responsible for issuing a top-level command to the worker, which is in charge of executing the build commands. The worker outputs the binary data and returns the artifact identifier to the orchestrator, which then produces the signed build recipe (if the artifact identifier matches the expected one).

5.2. Certificate Authorities (CA)

For asymetric cryptography, a proof of the authenticity of the public key is needed. This is done through certificates, which are mode of:

- The Public key of the entity with its identity information.
- All that signed by a trusted third party, called Certificate Authority (CA).

The public key of the CA is widely distributed and trusted by all parties, and are usually pre-installed in web browsers and operating systems. A PKI (Public Key Infrastructure) is a system that manages the issuance, revocation, and validation of digital certificates.

5.2.1. CA Creation

Google wanted to have their own CA to issue certificates for their internal services. That way, they did not have to rely on external CAs, which could be compromised or unavailable. They also decided to use their own software because of the flexibility and control it provided. Even though they obviously had to use third-party libraries for some parts, they tested them thoroughly and made sure they were secure.

Programming Languages

They used two programming languages:

- Go: It is memory-safe, important to work with unknown inputs such as certificate signing requests.
- C++: offers more interoperability with existing Google infrastructure, and offers a sandboxed environment to run untrusted code.
- They were both used because of performance reasons and the amount of good and safe libraries available.

Complexity vs Understandability and Ease of Use

Most comercial CA are really complex, as they have to offer a lot of features and support a wide range of use cases. However, Google wanted to have a CA with the minimum set of features needed for their internal use cases, so they could have a better understanding of the system and make it easier to use and maintain. It is continously improved, as they realised that they initially used too many microservices.

Security of their Private Keys

A great risk for a CA is the compromise of its private keys, as it would allow an attacker to issue fraudulent certificates. To mitigate this risk, Google uses Hardware Security Modules (HSM) to store their private keys. HSMs are tamper-resistant devices that provide a secure environment for key storage and cryptographic operations. They also use multi-factor authentication and strict access controls to limit access to the HSMs, which are offline most of the time to prevent remote attacks. Intermediate Keys are also used, so the root key is only used to sign the intermediate keys, which are then used to sign the certificates. This way, if an intermediate key is compromised, the root key remains secure.

5.3. Human Factors in Secure Deployment

In order to avoid human errors during the deployment process, there are some general guidelines available in internet:

- **NIST**: National Institute of Standards and Technology (NIST) provides the "Secure Software Development Framework", which includes guidelines for secure deployment and references to other relevant standards.
- **OWASP SAMM**: Open Web Application Security Project (OWASP) provides the "Software Assurance Maturity Model" (SAMM), which are some open-source guidelines for secure software development, including deployment. They propose that a company should be divided into 5 business functions:
  - Governance: Strategy and metrics.
  - Design: Threat modeling and secure architecture.
  - Implementation: Secure coding and code review.
  - Verification: Security testing and vulnerability management.
  - Operations: Incident detection and response.
- **BSIMM**: Building Security In Maturity Model (BSIMM) provides anual reports of security activities and trends. It is based on the observation of around 130 companies.

5.3.1. Vulnerability Management

Vulnerability management is the process of identifying, assessing, and mitigating vulnerabilities in software applications. Before addressing how are they identified, some concepts should be explained:

- Bug: General term for a flaw in the software that can cause it to behave unexpectedly or incorrectly. Not all bugs are security-related, but they can still have an impact on the security of the application.
- Weakness: A type of bug that, under certain conditions, can lead to a vulnerability. A weakness is a potential security issue that may or may not be exploitable, depending on the context and the presence of other factors.

  The CWE (Common Weakness Enumeration) is the world's most widely adopted list of weaknesses types. It is developed by the community.
- Vulnerability: An specific instance of a weakness that has been identified in a particular application. A vulnerability implies that the system can be exploited by an attacker to cause harm. It is similar to an exposure.

  The CVE (Common Vulnerabilities and Exposures) is a list of publicly disclosed vulnerabilities and exposures. It is maintained by MITRE Corporation, and each vulnerability is assigned a unique identifier (CVE ID) and a description of the vulnerability, its impact, and its severity.

Identifying Bugs: Bug Bounty Programs

Appart from using the CWE and CVE databases, companies can also identify vulnerabilities through bug bounty programs, which are crowed-sourced systems whose aim is discovering vulnerabilities in their applications. Vulnerability huters

find vulnerabilities and report them to the company, which then rewards them with money/credit/reputation. Some important Bug Bounty Programs are `bugcrowd` and `hackerone`. Some important questions arise:

- Why should bug hunters be paid?:

  There are several reasons for this:

  - The "zero price" fallacy: If they were not paid, researchers would have zero opportunity cost, so they would not have any incentive to find vulnerabilities.
  - Competing with the black market: If they are not paid, they may sell the vulnerabilities to the dark web, where they can get a lot of money for them.
  - Professionalism: Paying bug hunters is a way to recognize their work and encourage them to continue finding vulnerabilities.
  - Incentiving Depth: If they are not paid, they may only look for low-hanging fruit, while if they are paid, they may be incentivized to look for more complex and critical vulnerabilities.
  - Safe Harbor: It provides a legal framework for researchers to report vulnerabilities without fear of legal repercussions.

- What benefits do bug hunters get?:

  Apart from money, they can get several benefits as learning, reputation, enjoyment, legal safe harbor, and even job opportunities.

- What challenges do bug hunters face?:

  They may face several challenges, such as poor communication with the company, duplicated reports...

Adressing Bugs

Once the bugs are identified, they need to be addressed. However, it is impossible to fix all vulnerabilities at once, so they should be prioritized based on their severity and impact. This can be beasured by several metrics (or even combining them):

- **CVSS**, Common Vulnerability Scoring System.

  Method used to supply a qualitative measure of severity (not risk). Each vulnerability is scored from 0 (least severe) to 10 (most severe) based on several factors, and then they are categorized into four severity levels:

  - Critical $(9{,}0 - 10{,}0)$: $< 3\,\text{h}$.
  - High $(7{,}0 - 8{,}9)$: $< 3\,\text{day}$.
  - Medium $(4{,}0 - 6{,}9)$: $< 1\,\text{month}$.
  - Low $(0{,}1 - 3{,}9)$: $< 3\,\text{month}$.

- **EPSS**, Exploit Prediction Scoring System.

  It uses the chance of a vulnerability being exploited in the wild, and the most likely to be exploited are the ones that should be fixed first.

- Known Exploited Vulnerabilities (KEV) catalog from CISA.

- Business impact / Asset criticality.

Supply Chain Security

Modern software is dependent on many different components. Each dependency has the potential to introduce security risks into the end-product. When a vulnerability is found, most of the time a solution to it is reached in just a few days: the problem is finding all the systems that are vulnerable.

To solve this problem, an idea from the automotive industry is used: the Bill of Materials (BOM), which is a list of all the components used in a car, so that when a component is defective, all the cars that are affected can be easily identified. In software, a Software Bill of Materials (SBOM) is used, which is a list of all the components used in a software application, including their versions and licenses. Some important standards for SBOM are:

- NTIA's minimum elements (2021): It defines the minimum set of information that should be included in an SBOM, such as the component name, version and supplier. The current "minimum" requirements are higher than this initial proposal.

- CycloneDX (OWASP)

  It also provides a tool (OWASP Dependency Track) to make SBOM part of the software development lifecycle. When the source code is finished, the SBOM is generated and signed, and then it is sent to an analytics server that checks for vulnerabilities and license compliance. It also provides a standard format for SBOMs, which makes it easier to share and analyze them.

- SPDX (Linux Foundation)

- SWID (NIST)

5.3.2. Security Champion

When developing secure software, a common problem is the lack of organization. A common solution to this problem is to have a *security champion* in each development team. In addition, all security champions should form a community to share knowledge and best practices, so the whole company is improved. A security champion should at least have the following responsibilities:

- Being the source of security knowledge in the team. They should increase security awareness and promote best practices.

- Identify security risks and vulnerabilities in the team's code and processes.

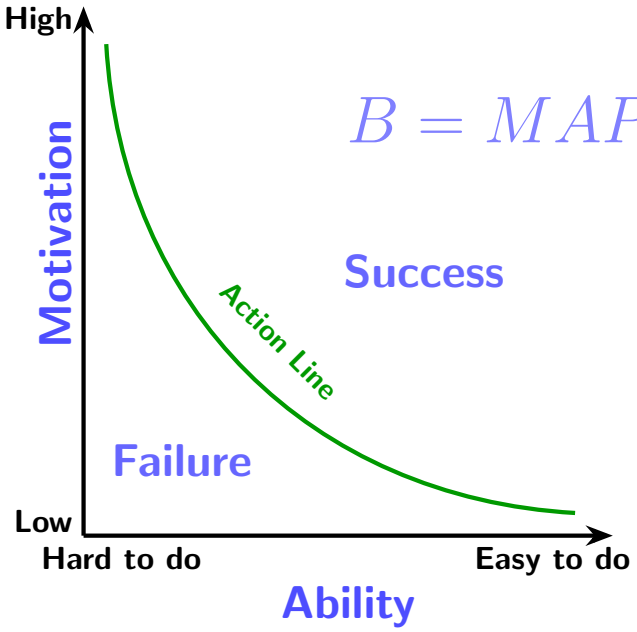

Figura 5.1: Fogg Behavior Model

- Review and escalation.

However, security champions must also be organized and supported by the company, as they cannot do everything alone. Some of the problems that may arise are:

- Shift in responsibilities: Developers may think that the security champion is responsible for all security-related tasks, leading to a lack of ownership and accountability among team members.
- Lack of time: Security champions may struggle to balance their security responsibilities with their regular development tasks, leading to burnout and decreased effectiveness.
- Insufficient training: Security champions may not have the necessary skills or knowledge to effectively identify and mitigate security risks. They all wish they had a security team backing them up.
- Selection: Choosing the right person for the role is crucial, and sometimes no one from the team wants to take the responsibility.
- Security Champion Skills: They need to have both technical and soft skills, such as communication and leadership. The Fogg Behavior Model (figure 5.1) explains that only when the three factors (motivation, ability and trigger) are present, a behavior will occur.
- Communication with PM: They need to effectively communicate security issues and risks to project managers and other stakeholders, as security is not always a priority for them.

In order to address these challenges, in the OWASP Security Champions Guide there is a manifesto that outlines the principles and values that security champions should uphold:

- Be passionate about security.
- Start with a clear vision.
- Secure management support.
- Nominate a dedicated captain.
- Trust your champions.
- Create a community.
- Promote knowledge sharing.
- Reward responsibility.
- Invest in your champions.
- Anticipate personnel changes.

As previously explained, Security Champions should promote knowledge sharing within the organization. With that aim is the OWASP Juice Shop, which "is probably the most modern and sophisticated insecure web application". It is an intentionally insecure web application for security training purposes, as it can be used in security trainings or awareness demos.

6. Software Testing

As discussed in previous chapters, testing is a crucial aspect of software development and maintenance. It is known that in complex systems, usually more time is needed to write the tests than to write the actual code. However, even with the best coding practices, bugs and issues are inevitable. Unexpected inputs may lead to:

- Confidentiality Violations: Unauthorized access to sensitive data.
- Integrity Violations: Unauthorized modification of data.
- Availability Violations: Disruption of service.

6.1. Test Types

There are two main categories of tests: unit tests and acceptance tests. In this section, we will explore these types of tests in detail.

6.1.1. Unit Tests

An unit test is a type of software test that focuses on verifying the functionality of a specific section of code, typically at the function or method level. The main goal of unit testing is to ensure that individual components of the software work as intended in isolation, without dependencies on other parts of the system. There are a lot of tools for unit testing, as `xUnit` or `GoogleTest`. There are two main approaches to develop the unit tests:

- Classic Approach: Write the code first, then create unit tests to verify its functionality. Code and tests should be commited together, and when the code is reviewed, the tests should be reviewed as well.
- Test-Driven Development (TDD): Write the unit tests before writing the actual code. The tests will fail until the code is implemented correctly. This approach encourages developers to think about the requirements and design of the code before implementation.

It should also be noted that unit tests may sometimes require refactoring of the code to make it more testable. This can lead to better code quality and maintainability.

6.1.2. Acceptance Tests

Acceptance tests, also known as end-to-end tests or functional tests, are designed to verify that a software application meets the specified requirements and behaves as expected from the user's perspective. These tests focus on validating the overall functionality of the system, ensuring that all components work together seamlessly to deliver the desired user experience. They usually need much more time to be developed and run than unit tests, but they are crucial to ensure that the software meets the user's needs. If an acceptance test fails but all of the unit tests pass, the error is usually difficult to locate.

Regarding the acceptance tests, it should be noted that *flakiness* is more common than in unit tests. A flaky test is a test that can pass or fail non-deterministically, without any changes to the code being tested. This can be due to various factors, such as timing issues, external dependencies, or environmental factors. Flaky tests can lead to false positives or false negatives, making it difficult to determine the actual state of the software. Therefore, it is important to identify and address flaky tests to ensure the reliability of the testing process.

6.2. Program Analysis

Analyzing code can help to identify potential issues and improve code quality. There are two main types of program analysis: static analysis (analyzing code without executing it) and dynamic analysis (analyzing code during execution).

When analyzing code, both the source and the binary code can be considered. There are two aspects that should be taken into account:

- Dynamic Binary Instrumentation (DBI): Using kind of a virtual machine to analyze the binary code during execution. Examples of DBI tools are `Valgrind` or `Intel Pin`.
- Dynamic Analysis based on compiler support: The compiler inserts additional code to perform the analysis during execution. They are often required to detect memory errors.

In order to analyze code during execution, it is common to use `sanitizers`, which are tools that detect various types of errors at runtime. A really common sanitizer is `Address Sanitizer`, which is described in the next section.

6.2.1. Adress Sanitizer

`Address Sanitizer (ASan)` is a fast memory error detector. It usually detects:

- Use-after-free: Accessing memory after it has been freed.
- Out-of-bounds access: Accessing memory outside the allocated bounds.

ASan uses a technique called *shadow memory* to keep track of the state of each byte of memory. For each 8 bytes of application memory, ASan maintains 1 byte of

shadow memory. The shadow memory is used to store metadata about the state of the corresponding application memory. When a programm is compiled with ASan, additional instrumentation code is added with two main aims:

- Before every memory access, ASan checks the corresponding shadow memory to determine if that memory adress is "poisoned" (i.e., invalid or unsafe to access).
- When memory is allocated, 32 bytes of "red zones" are added before and after the allocated memory to detect out-of-bounds accesses.

An important aspect to consider when using ASan is that it increases both memory usage and execution time. Typically, ASan increases memory usage by about 2-3 times and slows down program execution by a factor of 2-3. Given that overhead, ASan is primarily used during development and testing phases rather than in production environments.

6.3. The Quest for Coverage

The goal of testing is to cover as much code as possible, in order to detect potential bugs and issues. This should be done with the minimum number of test cases, to reduce the time and effort needed to run the tests. Typically, one test case explores one path through the program. In order to achieve this goal, several techniques can be used, such as symbolic execution and fuzzing.

6.3.1. Symbolic Execution

Symbolic execution is a program analysis technique that explores program paths by treating input values as symbolic variables rather than concrete values. This allows the analysis to reason about multiple execution paths simultaneously, enabling the detection of potential bugs and vulnerabilities that may not be easily discovered through traditional testing methods.

When the symbolic execution engine finishes, all the possible paths through the program have been explored, and a set of path constraints has been generated for each path. These path constraints can be used to generate test inputs that will exercise specific paths through the program, helping to achieve better code coverage and identify potential issues.

However, symbolic execution has some limitations in practice, as systems can be really complex:

- Path Explosion: The number of possible execution paths can grow exponentially with the size of the program, making it infeasible to explore all paths.
- Handling of External Dependencies: Symbolic execution may struggle to accurately model interactions with external libraries, system calls, hardware components, or user inputs.

Therefore, the scope is usually limited to small and critical parts of the code, or it is combined with other techniques, such as fuzzing.

SMT (Satisfiability Modulo Theories) Solvers

SMT solvers are tools that determine the satisfiability of logical formulas with respect to certain background theories, such as arithmetic, bit-vectors, arrays, and more. They are commonly used in symbolic execution to solve the path constraints generated during the analysis. By solving these constraints, SMT solvers can generate concrete input values that will exercise specific paths through the program, helping to achieve better code coverage and identify potential issues. Some popular SMT solvers include `Z3`, `CVC4`, and `Yices`.

SMT `Z3`

`Z3` is a high-performance SMT solver developed by Microsoft Research. It is usually used in `python`, importing the `z3` module. It provides its own data types, and the more relevant ones are:

- `x = z3.Int('x')`
- `x = z3.Bool('x')`
- `x = z3.Real('x')`
- `x = z3.BitVec('x', <number_of_bits>)`

In addition to those data types, `z3` also lets using the plural form of the data types to create multiple variables at once. For example, `x, y = z3.Ints('x y')` creates two integer variables, `x` and `y`.

To solve a set of constraints, the following steps are usually followed:

1. Create a solver instance: `solver = z3.Solver()`
2. Add constraints to the solver: `solver.add(<constraint>)`. When creating them, the operators used are the same as in regular Python code, but they are overloaded to work with `z3` data types. However, the logic operators `not`, `and`, and `or` are represented as `z3.Not()`, `z3.And()`, and `z3.Or()`, respectively.
3. Try to solve the constraints with `solver.check()`. It can have two possible outputs:
   - `z3.sat`: The constraints are satisfiable, and a solution can be found using `solver.model()`.
   - `z3.unsat`: The constraints are unsatisfiable, meaning that there is no solution that satisfies all the constraints.

6.3.2. Fuzzing

Fuzzing is an automated software testing technique that involves providing different types of inputs to a program in order to identify potential bugs, vulnerabilities, or unexpected behavior. The main goal of fuzzing is to explore the program's input space and uncover edge cases that may not have been considered during development. There are several types of fuzzing techniques, including the following:

- Random Fuzzing: Randomly generates inputs without any specific knowledge of the program's structure or behavior. The inputs for the following test cases are also generated randomly. This technique is simple to implement but may not be very effective in finding deep or complex bugs.
- Mutation-based Fuzzing: Starts with a set of valid inputs, and the inputs for the following test cases are generated by making small modifications (mutations) to these valid inputs.
- Cover-guided Fuzzing: Starts with a set of initial inputs and, after running each test case, it analyzes the code coverage achieved. If new paths are discovered, the inputs that led to those paths are mutated to generate new test cases, and if no new paths are found, that input is discarded. This technique is more effective in exploring the program's input space and finding bugs.

A really common tool for fuzzing is `Atheris`.

Atheris

`Atheris` is a coverage-guided fuzzer for Python. It is designed to be easy to use and integrate into existing testing workflows. It needs to instrument the code to be tested, which is done by using the following python functions:

- `atheris.instrument_imports()`: This function is used to instrument the imports in the code, allowing Atheris to track the coverage of the imported modules.
- `atheris.instrument_func(func)`: This function is used to instrument a specific function, allowing Atheris to track the coverage of that function.
- `atheris.instrument_all()`: This function is used to instrument all the code in the module, allowing Atheris to track the coverage of the entire codebase.

In order to improve efficiency, only the most critical parts of the code are usually instrumented, as instrumenting the entire codebase can lead to a significant increase in execution time and memory usage. The funcions used to run the fuzzer are:

- `atheris.Setup(<args>, <test_function>)`: This function is used to set up the fuzzer, specifying the command-line arguments and the test function to be executed. Usually, `<args>` is set to `sys.argv`, as it allows the fuzzer to accept command-line arguments for configuration.

  In the arguments, the following options can be specified:

  - `--runs=N`: Specifies the number of test cases to be executed before the fuzzer stops. If not specified, the fuzzer will run indefinitely until it is manually stopped.
  - `--timeout=N`: Specifies the maximum time (in seconds) that the fuzzer will run before it stops. If not specified, the fuzzer will run indefinitely until it is manually stopped.

  - `--seed=N`: Specifies the seed for the random number generator used by the fuzzer. This can be useful for reproducibility, as it allows you to generate the same sequence of test cases by using the same seed.
  - `<corpus_file>`: Specifies a file containing a corpus of inputs to be used as the initial seed for the fuzzer. The fuzzer will use these inputs to generate new test cases through mutation. If not specified, the fuzzer will start with an empty corpus and generate test cases randomly.

- `atheris.Fuzz()`: This function starts the fuzzing process, running the specified test function with different inputs generated by the fuzzer. It will continue to run until it is manually stopped or until a certain condition is met (e.g., a specific number of test cases have been executed).

There are different types of lines in the output of Atheris:

- `INITED`: Indicates that the fuzzer has been initialized and is ready to start fuzzing.
- `NEW`: Indicates that a new path has been discovered during fuzzing. This means that the fuzzer has generated an input that has led to a new execution path in the program.
- `pulse`: Indicates that the fuzzer is still running and has not yet found any new paths. This is a normal part of the fuzzing process, as it may take some time for the fuzzer to discover new paths.

In each output, the following information is provided:

- `cov`: The current code coverage achieved by the fuzzer, expressed number of nodes in the control flow graph that have been covered.
- `corp <x>/<y>b`: The size of the corpus, expressed as the number of inputs in the corpus (`x`) and the total size of those inputs in bytes (`y`).
- `exec/s`: The number of executions per second, which indicates how quickly the fuzzer is generating and testing new inputs.
- `L`: The length of the input that led to the discovery of a new path. This can provide insight into the complexity of the input that triggered the new path.
- It also indicates how was the new path discovered (`Change-Byte`, `Cross-Over` (combining two existing inputs), `CopyPart`, `EraseBytes`, `InsertByte`, etc.).

7. Übungen

7.1. Application Lifecycle Management (ALM)

Ejercicio 7.1.1.

1. Erklären Sie den Begriff "Application Lifecycle Management" (ALM). Geben Sie an, welche Phasen im ALM typischerweise enthalten sind und warum das Verständnis dieser Phasen für das App Management wichtig ist.

   The Application Lifecycle Management (ALM) is the framework that defines the process of managing an application throughout its whole lifecycle, from the initial idea to its end of life. It integrates *people*, *processes* and *tools* to manage the application effectively and efficiently. There are no fixed phases in ALM, but the following ones are usually included:

   - Requirements & Planning
   - Development
   - Testing
   - Deployment
   - Maintenance
   - Retirement

   Understanding these phases is important for App Management because it allows to manage the complexity of modern applications. Nowadays there are a lot of different people, named *stakeholders*, involved in the creation and maintenance of an application (Developers, Bussiness Analysts, Testers, Final Users, etc). ALM provides a structured approach to coordinate all these people and their tasks. This lets everyone know *what should they do at any moment*. This leads to overcome the typical "controled chaos" that usually happens in large projects.

2. Beschreiben Sie die Bedeutung der Sicherheit im Application Management. Nennen Sie mindestens drei Sicherheitsaspekte, die bei der Entwicklung und Verwaltung von Anwendungen zu berücksichtigen sind.

   Security is a critical aspect of Application Management because applications often handle sensitive data and are exposed to various threats. Neglecting security can lead to data breaches, loss of user trust, and legal consequences. Here are three security aspects to consider:

- Authentication and Authorization: Ensuring that only authorized users can access the application and its data.
- Data Encryption: Protecting sensitive data both in transit and at rest to prevent unauthorized access.
- Automation: In order to avoid manual errors, which are a common source of security vulnerabilities, it is important to automate security testing and deployment processes.

3. Vergleichen Sie die Phasen des Application Lifecycle Management (ALM) mit den Phasen des Software Development Lifecycle (SDLC). Identifizieren Sie mindestens zwei Gemeinsamkeiten und zwei Unterschiede zwischen diesen beiden Ansätzen.

   The Software Development Lifecycle (SDLC) is a subset of ALM that focuses specifically on the development phase of an application. Specially, ALM also includes maintenance and retirement phases, which are not typically part of SDLC. Here are two similarities and two differences between ALM and SDLC:

   - Similarities:
     - Both ALM and SDLC include phases for requirements gathering, design, development, testing, and deployment.
     - They both describe a structured approach to managing the development of software applications.
   - Differences:
     - ALM encompasses the entire lifecycle of an application, including maintenance and retirement, while SDLC focuses primarily on the development phase.
     - ALM emphasizes the integration of people, processes, and tools across the entire lifecycle, while SDLC is more focused on the technical aspects of software development.

Ejercicio 7.1.2. Erklären Sie die Vor- und Nachteile der folgenden Entwicklungsmethoden:

1. Agile Entwicklung

   - Advantages: Flexibility, faster delivery, better customer collaboration, and improved quality through iterative development.
   - Disadvantages: Can lead to scope creep, requires strong team collaboration, and may not be suitable for projects with well-defined requirements.

2. Scrum

   - Advantages: Provides a clear framework for managing complex projects, promotes transparency and accountability, and encourages continuous improvement.
   - Disadvantages: Can be challenging to implement in teams that are not used to agile practices, requires a high level of discipline, and may lead to burnout if not managed properly.

3. Wasserfall-Modell

    - Advantages: Provides a clear and structured approach, easy to understand and manage, and works well for projects with well-defined requirements.
    - Disadvantages: Inflexible to changes, can lead to long development cycles, and may result in a final product that does not meet user needs if requirements are not accurately defined at the beginning.

4. DevOps-Ansatz

    - Advantages: Promotes collaboration between development and operations teams, enables faster delivery and deployment, and improves the overall quality of applications through automation.
    - Disadvantages: Requires a cultural shift in organizations, can be complex to implement, and may require significant investment in tools and training.

Ejercicio 7.1.3. Nehmen Sie an, Sie sind der Manager eines kleinen Softwareentwicklungsteams, das eine Echtzeit-Messaging-App für den Campus der "Universität der Zukunft" entwickelt. Diese App ermöglicht Studierenden und Professoren eine einzigartige Kommunikation, die den Alltag auf dem Campus einfacher und unterhaltsamer macht. Erklären Sie, warum es wichtig ist, von Anfang an ein effektives Application Management in Ihre Projekte zu integrieren. Geben Sie konkrete Beispiele für mögliche Probleme, die vermieden werden könnten, wenn Sie sich frühzeitig auf das Application Management konzentrieren, um sicherzustellen, dass Ihre App im Universitätsalltag reibungslos funktioniert.
Integrating effective Application Management from the beginning of the project is crucial for several reasons. It helps to ensure that the development process is organized, efficient, and aligned with the goals of the project. Here are some specific examples of potential problems that could be avoided by focusing on Application Management early on:

- Scope Creep: Without proper management, the project could suffer from scope creep, where new features and requirements are added without proper evaluation. This can lead to delays and increased costs.
- Resource Allocation: Effective Application Management helps to allocate resources efficiently, ensuring that the team has the necessary tools and personnel to complete the project on time.
- Quality Assurance: By integrating testing and quality assurance processes early in the development cycle, potential issues can be identified and addressed before they become major problems, ensuring that the app functions smoothly in the university environment.

Ejercicio 7.1.4. Ihr Team entwickelt weiterhin die Echtzeit-Messaging-App. Beschreiben Sie, wie Scrum den Entwicklungsprozess strukturiert.

Scrum is an agile framework that structures the development process into iterative cycles called sprints, typically lasting 2-4 weeks. There are three main roles in Scrum:

- Product Owner: Responsible for defining the product backlog (a prioritized list of features and requirements). This person could be a representative of the university, such as a student or professor, who understands the needs of the users.
- Scrum Master: Facilitates the Scrum process, ensuring that the team follows the framework and removes any obstacles that may arise. This person would help the team stay focused and organized throughout the development process.
- Development Team: A cross-functional group responsible for delivering the product increment at the end of each sprint. This team would consist of developers, testers, and designers who work together to create the app.

The development process in Scrum is structured around the following events:

- Sprint Planning: At the beginning of each sprint, the team plans the work to be done, selecting items from the product backlog to be completed during the sprint.
- Daily Scrum: A short daily meeting where the team discusses progress, plans for the day, and any obstacles they are facing.
- Sprint Review: At the end of each sprint, the team demonstrates the completed work to stakeholders and gathers feedback.
- Sprint Retrospective: After the sprint review, the team reflects on the sprint and identifies areas for improvement in the next sprint.

By structuring the development process with Scrum, the team can ensure that they are delivering value to the users in a timely manner, while also allowing for flexibility and continuous improvement throughout the project.

7.2. Version Control System (VCS)

Ejercicio 7.2.1. Beschreiben Sie die grundlegenden Unterschiede zwischen Git und SVN hinsichtlich ihrer Arbeitsweise und ihres Datenmodells. Erläutern Sie, was ein verteiltes Versionskontrollsystem (Git) und ein zentrales Versionskontrollsystem (SVN) sind.

Both Git and SVN are version control systems, but they differ in their architecture and how they manage data. Both architectures have a central repository in a server, however:

- In a centralized version control system like SVN, the central repository is the only repository, and all operations (commits, updates, etc.) are performed directly on this central repository. This means that developers need to be connected to the central repository to perform most operations, and the history of changes is stored in this central location.
- In a distributed version control system like Git, each developer has also a complete copy of that main repository, including its entire history. This allows developers to work offline and perform operations such as commits, branching, and merging locally. Changes can be shared between repositories through push and pull operations, but the local repositories are fully functional on their own.

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.2.2.

1. Nehmen Sie an, Sie arbeiten mit einem kleinen Team an der Echtzeit-Messaging-App für die „Universität der Zukunft“. Sie haben noch lokale Änderungen, die noch nicht committet sind. Erstellen Sie bitte einen Commit und schließen Sie somit die Entwicklung des neuen Features ab.

   The code needed to solve this exercise is shown in Listing 2.

2. Da das neue Feature nun fertig implementiert ist, möchten Sie dafür sorgen, dass es auch in den aktuellen master aufgenommen wird. Der übliche Prozess in Ihrem Team ist, einen Merge Request (MR) zu erstellen, der den Feature-Branch in den master übernimmt. Es gibt die Richtlinie, dass MRs nur dann akzeptiert werden, wenn sie mit dem master aktuell sind und keine Konflikte erzeugen. Sorgen Sie dafür, dass der Feature-Branch `cool_stuff` mit dem master-Branch aktuell ist.

   The code needed to solve this exercise is shown in Listing 3.

Ejercicio 7.2.3. Ihr MR wurde akzeptiert und das neue Feature ist im master. Parallel dazu haben Sie im `other_cool_stuff`-Branch noch an einem ähnlichen Feature gearbeitet. Bereiten Sie auch diesen Branch für den MR in den master vor.

At first it is needed to execute the code shown in Listing 4. After manually solving the conflict, the code shown in Listing 5 is executed.

```
$ git status
On branch cool_stuff
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   text.py

no changes added to commit (use "git add" and/or "git commit -a")
$ git add text.py
$ git commit -m "New Feature"
[cool_stuff 00c5b2a] New Feature
 1 file changed, 3 insertions(+), 2 deletions(-)
```

Código fuente 2: Lösung zu Übung 7.2.2.1

```
$ git branch
* cool_stuff
  master
$ git checkout master
Switched to branch 'master'
$ git merge cool_stuff
Merge made by the 'ort' strategy.
 text.py | 12 ++++++++++-
 1 file changed, 11 insertions(+), 1 deletion(-)
```

Código fuente 3: Lösung zu Übung 7.2.2.2

```
$ git branch
  master
* other_cool_stuff
$ git checkout master
Switched to branch 'master'
$ git merge other_cool_stuff
Auto-merging text.py
CONFLICT (content): Merge conflict in text.py
Automatic merge failed; fix conflicts and then commit the result.
```

Código fuente 4: Erster Teil der Lösung zu Übung 7.2.3.

```
$ git add text.py
$ git commit
[master 9abac85] Merge branch 'other_cool_stuff'
```

Código fuente 5: Zweiter Teil der Lösung zu Übung 7.2.3.

```
#!/bin/sh

git diff --cached --name-only --diff-filter=A \
| while read -r name; do
    git rm --cached "$name" > /dev/null 2> /dev/null
done

exit 0
```

Código fuente 6: `pre-commit` Hook that prevents committing new files.

Ejercicio 7.2.4. Ein Kollege von Ihnen ist erst seit Kurzem im Team und bittet Sie um Hilfe, da er lokale Änderungen vorgenommen hat, die er jedoch nicht committen kann. Helfen Sie ihm, alle geänderten und neu hinzugefügten Dateien zu committen.

The problem here lies in the fact that there is a `pre-commit` hook, the shown in Listing 6, that prevents committing new files. The solution is just to delete that hook from the `.git/hooks` directory.

```
$ git log --graph --oneline --all
* e777451 (HEAD -> cool_stuff) New Commit
* fbdb062 new motivating phrases
| * a5fdf5d (master) updated main.py
|/
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 7: Git-Graph nach dem Committen der Änderungen im `cool_stuff`-Branch.

```
$ git rebase master
Successfully rebased and updated refs/heads/cool_stuff.
$ git checkout master ; git merge cool_stuff
Switched to branch 'master'
Updating a5fdf5d..baae187
Fast-forward
 text.py | 12 ++++++++++-
 1 file changed, 11 insertions(+), 1 deletion(-)
```

Código fuente 8: Lösung zu Übung 7.2.2.2 mit Rebase.

7.3. Distributed Git und Internals

Ejercicio 7.3.1. In Aufgaben 7.2.2.2 und 7.2.3 des vorherigen Aufgabenblatts sollten Sie die Änderungen des master-Branches in den aktuellen Feature-Branch übernehmen. Überlegen Sie sich eine weitere Möglichkeit, die Änderungen zu übernehmen.

1. Aufgabe 7.2.2.2

   After committing the new changes to the `cool_stuff`-Branch, the git graph is the one shown in Listing 7. The other proposed solution is to rebase the `cool_stuff`-Branch on top of the `master`-Branch. The code needed to do that is shown in Listing 8. The result of the rebase is a linear history, which can be seen in the git graph shown in Listing 9.

```
$ git log --graph --oneline --all
* baae187 (HEAD -> master, cool_stuff) New Commit
* 9c087cd new motivating phrases
* a5fdf5d updated main.py
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 9: Git-Graph nach dem Rebase des `cool_stuff`-Branches auf den `master`-Branch.

```
$ git log --graph --oneline --all
* ff02d9a (HEAD -> other_cool_stuff) fixed unterminated string literal
and added text2 to main
* 13061d3 new text2
| * 07216d8 (master) finalized cool stuff
| * 36e8466 new motivating phrases
|/
* a5fdf5d updated main.py
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 10: Git-Graph vor dem Rebase des `other_cool_stuff`-Branches auf den `master`-Branch.

```
$ git add text.py
$ git rebase --continue
[detached HEAD fe644de] new text2
 1 file changed, 4 insertions(+), 9 deletions(-)
Successfully rebased and updated refs/heads/other_cool_stuff.
$ git checkout master ; git merge other_cool_stuff
Switched to branch 'master'
Updating 07216d8..7027614
Fast-forward
 main.py |  4 ++--
 text.py | 13 ++++---------
 2 files changed, 6 insertions(+), 11 deletions(-)
```

Código fuente 11: Lösung zu Übung 7.2.3 mit Rebase.

2. Aufgabe 7.2.3

   The initial git graph is the one shown in Listing 10. When trying to rebase the `other_cool_stuff`-Branch on top of the `master`-Branch, a conflict is generated. After manually solving the conflict, the code shown in Listing 11 is executed. The result of the rebase is a linear history, which can be seen in the git graph shown in Listing 12.

Ejercicio 7.3.2. Überlegen Sie sich mögliche Vor- und Nachteile der drei vorgestellten Distributed Workflows.

1. Dictator and Lieutenants Workflow

   - Advantages: Clear hierarchy and control over the codebase; easier to manage contributions from multiple developers; more code reviews before integrating everything in the blessed repository.
   - Disadvantages: Slower development process due to the need for code reviews and approvals; potential bottleneck if the dictator is unavailable or overwhelmed with contributions; less autonomy for developers, which may lead to lower motivation and creativity.

```
$ git log --graph --oneline --all
* 7027614 (HEAD -> master, other_cool_stuff) fixed unterminated string
literal and added text2 to main
* fe644de new text2
* 07216d8 finalized cool stuff
* 36e8466 new motivating phrases
* a5fdf5d updated main.py
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 12: Git-Graph nach dem Rebase des `other_cool_stuff`-Branches auf den `master`-Branch.

2. Integration-Manager Workflow
    - Advantages: The bottleneck is reduced compared to the Dictator and Lieutenants Workflow, as there are multiple integration managers; more autonomy for developers, which can lead to higher motivation and creativity; faster development process due to less need for code reviews and approvals.
    - Disadvantages: Potential for conflicts between integration managers; less control over the codebase compared to the Dictator and Lieutenants Workflow; potential for lower code quality if integration managers do not perform thorough reviews.
3. Centralized Workflow
    - Advantages: Simple and straightforward workflow; easier to manage for small teams; faster development process due to less need for code reviews and approvals.
    - Disadvantages: Higher risk of conflicts and code quality issues due to lack of code reviews.

Ejercicio 7.3.3. Der Chef des Teams, zuständig für die Entwicklung der Echtzeit-Messaging-App hat wenig Ahnung von Softwareentwicklung. Er hat damals einfach irgendwelche Regeln bezüglich des Merge-Prozesses festgelegt, weiß aber nicht wirklich, was diese bedeuten, und bittet Sie, einen sinnvollen Workflow für das Projekt zu wählen. Wählen Sie einen passenden Workflow aus und begründen Sie Ihre Wahl.

Even though the choice of a workflow depends on the specific context and needs of the project, a good option for a real-time messaging app could be the Integration-Manager Workflow.

- Centralized Workflow: This workflow is too simple, as no code reviews would be performed before merging changes into the main branch. This could lead to conflicts and code quality issues, which are especially problematic in a real-time messaging app where reliability and performance are crucial.

```
$ git fsck --lost-found
Checking object directories: 100% (256/256), done.
Checking objects: 100% (28/28), done.
dangling blob acbc45bdb82b84a3df80a69659ad672c2791f632
Verifying commits in commit graph: 100% (8/8), done.
$ git cat-file -p acbc > lost.py
$ git add lost.py ; git commit -m "File recovered"
[master c00c6ec] File recovered
 1 file changed, 16 insertions(+)
 create mode 100644 lost.py
```

Código fuente 13: Lösung zu Übung 7.3.4 mit dem Befehl `git fsck --lost-found`.

- Dictator and Lieutenants Workflow: While this workflow provides more control over the codebase, it could lead to slower development due to the need for code reviews and approvals. In addition, it involves too many different roles, which could complicate the development process.

Therefore, the Integration-Manager Workflow strikes a good balance between control and autonomy. It allows for multiple integration managers to review and approve changes, which helps maintain code quality and consistency. At the same time, it provides developers with more autonomy, which can lead to higher motivation and creativity. This is particularly important in a real-time messaging app, where innovation and responsiveness to user needs are key factors for success.

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.3.4. Ihr Kollege bittet Sie erneut um Hilfe. Dieses mal sei es ernst, er hat all seine Arbeit der letzten Wochen verloren. Da er sich nicht gut mit Git auskennt, committet er selten. Als er fertig war, wollte er committen. Dafür hat er seine Änderungen mit `git add -A` in den Staging-Bereich gepackt. Doch da ist ihm eingefallen, dass er vorher noch Änderungen vom Remoteserver herunterladen muss. Also führt er git fetch aus und sieht, dass Remote-Änderungen übernommen wurden. Doch als er die Änderungen dann endlich in seinem lokalen Branch hat, sind alle seine eigenen Änderungen verschwunden. Helfen Sie Ihm die verlorenen Dateien wiederherzustellen.

The solution to this exercise involves using the Git command `git fsck --lost-found` to recover lost objects. The solution is shown in Listing 13.

Ejercicio 7.3.5. Nehmen Sie an, Sie haben ein Git-Repository, welches die in Abbildung 7.1 dargestellte Commit-Graphen hat. Nehmen Sie nun an, Sie führen die untenstehenden Git-Befehle aus. Zeichnen Sie den Commit-Graphen nach jedem Befehl. Wenn ein neuer Commit-Hash berechnet wurde, wählen Sie bitte eine eindeutige zufällige Nummer.

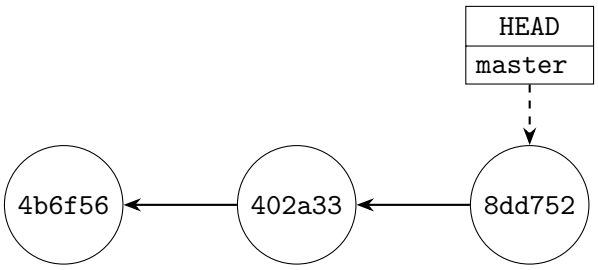

Figura 7.1: Git-Repository mit drei Commits und einem Branch `master`.

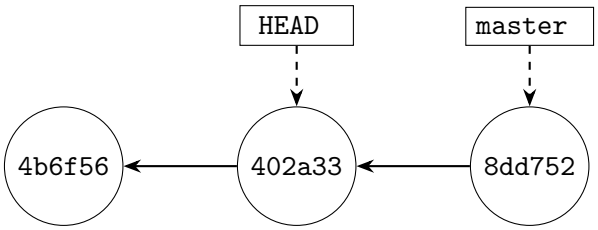

Figura 7.2: Git-Graph after the command of the Exercise 7.3.5.1.

1. `git checkout HEAD~1`

   The result of this command is shown in the git graph in Figure 7.2.

2. `git checkout -b 'feature_branch'`

   The result of this command is shown in the git graph in Figure 7.3.

3. `git commit -m 'new feature'`

   The result of this command is shown in the git graph in Figure 7.4.

4. Zeichnen Sie bitte beide Graphen

   a) `git merge master`
   The result of this command is shown in the git graph in Figure 7.5.

   b) `git rebase master`
   The result of this command is shown in the git graph in Figure 7.6.

5. Führen Sie abschließend für beide Graphen einen Merge von `feature_branch` in den `master` aus und löschen Sie den obsoleten Branch.

   In both cases, the command needed to merge the `feature_branch` into the `master` branch and delete the `feature_branch` is shown in Listing 14.

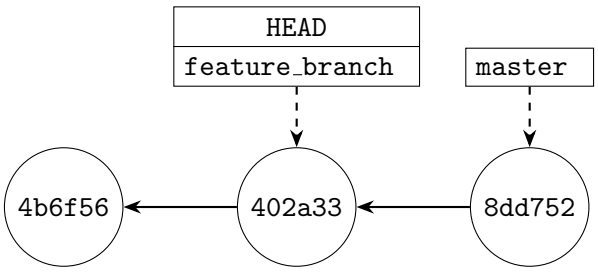

Figura 7.3: Git-Graph after the command of the Exercise 7.3.5.2.

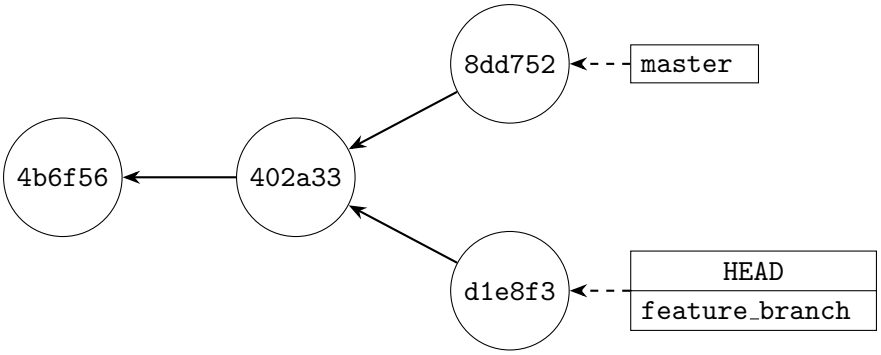

Figura 7.4: Git-Graph after the command of the Exercise 7.3.5.3.

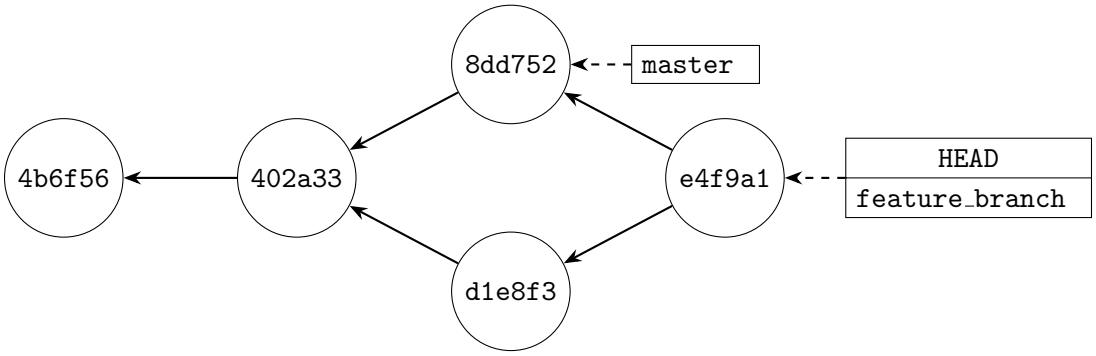

Figura 7.5: Git-Graph after the command of the Exercise 7.3.5.4*a*.

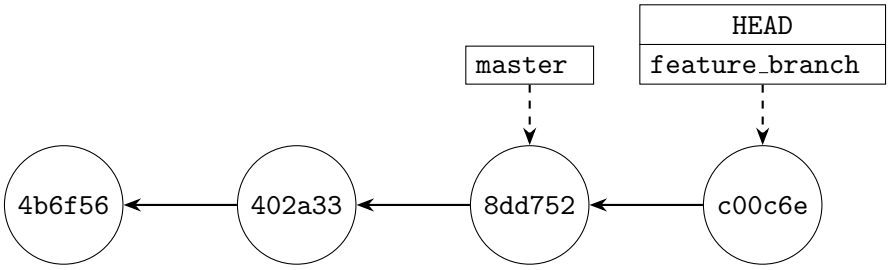

Figura 7.6: Git-Graph after the command of the Exercise 7.3.5.4*b*.

```
1 $ git checkout master ; git merge feature_branch ; git branch -d
  feature_branch
```

Código fuente 14: Command to merge the `feature_branch` into the `master` branch and delete the `feature_branch` for the graph in Figure 7.5.

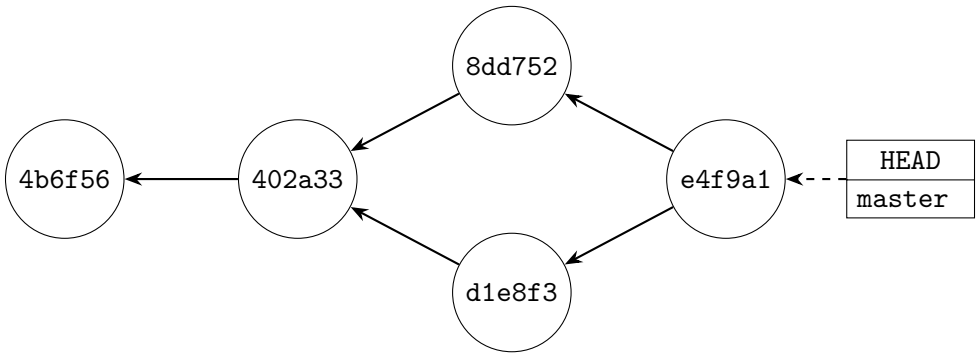

Figura 7.7: Git-Graph after merging the `feature_branch` into the `master` branch and deleting the `feature_branch` for the graph in Figure 7.5.

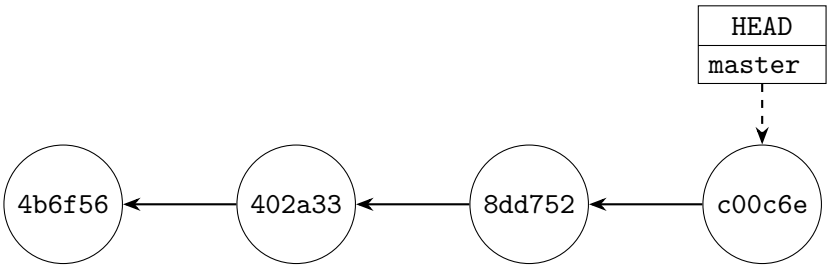

Figura 7.8: Git-Graph after merging the `feature_branch` into the `master` branch and deleting the `feature_branch` for the graph in Figure 7.6.

a) For the graph in Figure 7.5, after applying the command in Listing 14, the resulting graph is the one shown in Figure 7.7.

b) For the graph in Figure 7.6, after applying the command in Listing 14, the resulting graph is the one shown in Figure 7.8.

Ejercicio 7.3.6.

1. Angenommen, Sie haben gerade Ihre neuesten Änderungen committet und möchten vor dem Pushen testen, ob alles noch funktioniert. Dabei fällt Ihnen auf, dass ein Anführungszeichen fehlt. Sie haben es bereits hinzugefügt, finden es jedoch unnötig, dafür einen neuen Commit anzulegen. Fügen Sie die Änderung Ihrem lokalen Commit hinzu.

   The command needed to add the change to the last commit is shown in Listing 15.

```
$ git add text.py
$ git commit --amend --no-edit
[other_cool_stuff e9274fd] new text2
 Date: Thu Oct 26 14:38:02 2023 +0200
 2 files changed, 8 insertions(+), 3 deletions(-)
```

Código fuente 15: Lösung zu Übung 7.3.6.1.

```
$ git commit --amend -m "new text2 and added it to main"
[other_cool_stuff 9c087cd] new text2 and added it to main
 Date: Thu Oct 26 14:38:02 2023 +0200
 2 files changed, 8 insertions(+), 3 deletions(-)
```

Código fuente 16: Lösung zu Übung 7.3.6.2.

2. Nachdem Sie die Änderung vorgenommen haben, stellen Sie fest, dass die Commit-Nachricht nicht alle Änderungen widerspiegelt. Sie möchten der Nachricht hinzufügen, dass die neue Funktion auch in `main.py` aufgenommen wurde.

   The command needed to change the commit message of the last commit is shown in Listing 16.

```
$ git bisect start
status: waiting for both good and bad commits
$ git bisect bad
status: waiting for good commit(s), bad commit known
$ git bisect good b6763c2
Bisecting: 184 revisions left to test after this (roughly 8 steps)
[45d0499cacc9082b426292f53b63e24f5cb87a1e] Commit 215
```

Código fuente 17: Starting the bisecting process

7.4. Continuous Integration

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.4.1. Ihre Tests zeigen, dass eine Funktion nicht mehr korrekt funktioniert. Identifizieren Sie den Commit, ab dem der Fehler eingeführt wurde.

As we can see when executing `get_pi.py`, the output is not correct. We will use `git bisect` to find the commit that introduced the error. First of all we need to detect a commit where the output is correct (a good commit). We will check out to some of the first commits and execute the script to check that the output is correct. In this case, commit `b6763c2` is a good commit. Then, the Listing 17 shows how to start the bisecting process by marking the current commit as bad and the good commit as good.

At this point, Git has automatically checked out to a commit in the middle of the history (in this case, commit `45d0499`). The checking process is shown in the Listing 18. After the checking process, we can see that the first bad commit is `46917b5` (commit 173), which is the commit that introduced the error.

Finally, we can see that the first bad commit is `46917b5` (commit 173), which is the commit that introduced the error. In the Listing 19 we can see the details of the bad commit.

The checking process can be automated using the `git bisect run <script>` command, where `<script>` is the shown in the Listing 20. That way, Git will automatically execute the script in each commit and determine if it is good or bad based on the exit code of the script (0 for good, 1 for bad), simplifying the bisecting process.

Ejercicio 7.4.2. Für die Echtzeit-Messaging-App der "Universität der Zukunft" soll eine CI/CD-Pipeline aufgebaut werden.

1. Welche Schritte sollte die Pipeline umfassen und welche Werkzeuge könnte man dafür nutzen?

2. In Section 7.2 haben Sie bereits Git-Hooks kennen gelernt. Wie könnten Sie diese in einer CI- bzw. CD-Pipeline benutzen?

3. Welche Branching-Strategie für die Echtzeit-Messaging-App würden Sie vorschlagen?

```
            $ python3 get_pi.py
2.77
$ git bisect bad
Bisecting: 91 revisions left to test after this (roughly 7 steps)
[0ed8e90afea45388b98e7caa74475cf1da7ba614] Commit 123
$ python3 get_pi.py
3.14
$ git bisect good
Bisecting: 45 revisions left to test after this (roughly 6 steps)
[534ee36e8acfa1e8112b21c559cc4e390455113d] Commit 169
$ ...   # We continue the process by checking out to the next commit suggested
by Git and executing the script to check the output. We repeat this process
until we find the first bad commit.
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[53b336b7d7cbfc0069951d7cf1988b3c44c603f9] Commit 172
$ python3 get_pi.py
3.14
$ git bisect good
46917b552f8df592e2d86becbba6a26d7be1da36 is the first bad commit
commit 46917b552f8df592e2d86becbba6a26d7be1da36
Author: Alice <alice@example.com>
Date:   Mon Dec 9 18:25:28 2024 +0100

    Commit 173

 get_pi.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Código fuente 18: Bisecting process

```
$ git bisect reset
Previous HEAD position was 53b336b Commit 172
Switched to branch 'master'
$ git show 46917b5
commit 46917b552f8df592e2d86becbba6a26d7be1da36
Author: Alice <alice@example.com>
Date:   Mon Dec 9 18:25:28 2024 +0100

    Commit 173

diff --git a/get_pi.py b/get_pi.py
index 3325ae8..9b12bce 100644
--- a/get_pi.py
+++ b/get_pi.py
@@ -5,7 +5,7 @@ def berechne_pi(n_terms):
     for i in range(2, 2 + 2 * n_terms, 2):
         term = 4.0 / (i * (i + 1) * (i + 2))
         if add:
-            pi += term  # Berechnungsschritt 172
+            pi -= term  # Berechnungsschritt 172
         else:
             pi -= term  # Berechnungsschritt 172
         add = not add
```

Código fuente 19: Details of the bad commit

```
#!/bin/bash

RESULT=$(python3 get_pi.py)

if [ "$RESULT" == "3.14" ]; then
  exit 0
else
  exit 1
fi
```

Código fuente 20: Script to automate the bisecting process

4. Welche Unit-, Component- und Acceptance-Tests würden zur Messaging-App passen?

```
FROM ubuntu:22.04
COPY server_linux_x64 /server
RUN chmod +x /server
CMD ["/server"]
EXPOSE 8080
```

Código fuente 21: Dockerfile used to create the Docker image for the second instance of the server.

```
$ docker build -t server .
# ...
 => => naming to docker.io/library/server
 0.0s
$ docker run -p 8081:8080 server
Starting Webserver at: localhost:8080
```

Código fuente 22: Terminal commands to build the Docker image and run the Docker container for the second instance of the server.

7.5. Docker

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.5.1. Das `zip`-Datei enthält ausführbare Dateien für verschiedene Architekturen. Wenn Sie die Ubuntu-VM nutzen, ist die Datei `server_linux_x86_64` die richtige. Alle anderen Dateien sind ungetestet und nicht garantiert zu funktionieren. Sie können die Datei mit `$ ./server_linux_x86_64` ausführen und den Webserver im Browser unter `localhost:8080` erreichen. Ihre Aufgabe ist es, zwei Instanzen des Webservers parallel auf Ihrem System auszuführen.

The problem here is that the server always run on port 8080. The first instance will start without problems, but the second one will fail because the port is already in use. To solve this problem, Docker will be used to create an isolated environment for the second instance of the server, allowing it to run on the same port without conflicts. The Dockerfile used is shown in the Listing 21. The code executed in the terminal to build and run the Docker container is shown in the Listing 22.

Ejercicio 7.5.2. Sie sind Teil des Entwicklerteams für die Echtzeit-Messaging-App der “Universität der Zukunft”. Da Ihr Team in einer heterogenen Umgebung arbeitet und sicherstellen muss, dass das entwickelte Python-Skript unter verschiedenen Python-Versionen ordnungsgemäß ausgeführt wird, ist es Ihre Aufgabe, Docker-Container für diese Tests zu erstellen. Ihr Manager hat Sie gebeten, zu testen, ob die App mit allen offiziell unterstützten Python-Versionen ausführbar ist.

In order to accomplish this task, different Docker images for each Python version will be created, and they will be used to run the Python script in an isolated environment. In order to automate this process, the Dockerfile with recieve the Python version as a build argument, and a script will be created to build and run the Docker

```
# Slim -> Optimized image with only the necessary dependencies to run Python

ARG PYTHON_VERSION=3.12
FROM python:${PYTHON_VERSION}-slim
WORKDIR /app
COPY prime.py .
ENTRYPOINT ["python", "prime.py"]
CMD ["512"]
```

Código fuente 23: Dockerfile used to create the Docker images for each Python version.

```
#!/bin/bash
VERSIONS=("3.9" "3.10" "3.11" "3.12" "3.13")

for VER in "${VERSIONS[@]}"; do
    echo "-----------------------------------------------"
    echo "Testing Python version: $VER"
    echo "-----------------------------------------------"

    docker build --build-arg PYTHON_VERSION=$VER -t "test-python-$VER" . -q
    docker run --rm "test-python-$VER" 512

    echo -e "Test completed for $VER\n"
done
```

Código fuente 24: Script used to automate the building and running of Docker containers for each Python version.

containers for each Python version. The Dockerfile used is shown in the Listing 23. The script used to automate the process is shown in the Listing 24. The executed commands in the terminal to run the script are shown in the Listing 25.

Ejercicio 7.5.3.

1. Was ist Docker und wie unterscheidet es sich von Hypervisor-basierten Virtualisierungstechnologien?

   Even though both Docker and Hypervisor-based virtualization technologies provide isolation for applications, they do so at different levels. Docker uses containerization, which allows multiple applications to run directly in the host operating system, sharing the same kernel, while Hypervisor-based virtualization creates separate virtual machines with their own operating systems and kernels. The hypervisor abstracts the underlying hardware and allows multiple virtual machines to run on a single physical machine, while Docker abstracts the application and its dependencies, allowing it to run in an isolated envi-

```
$ chmod +x versions.sh ; ./versions.sh
$ ./versions.sh
-----------------------------------------------
Testing Python version: 3.9
-----------------------------------------------
sha256:4085d618dbeb1044b7eabcc8caa11805c27b681f5c31374931ddc9112eb0fb32
Traceback (most recent call last):
  File "/app/prime.py", line 4, in <module>
    from typing import Self
ImportError: cannot import name 'Self' from 'typing'
(/usr/local/lib/python3.9/typing.py)
Test completed for 3.9
# ...
# ...
-----------------------------------------------
Testing Python version: 3.13
-----------------------------------------------
sha256:7a4f73d714b81124ea19168bde4f6f39922aa0b679bbc5565e64ef8d09f0e885
Found prime number:
860948537517470561470852374872580873359899570093860724557 4...188797
Test completed for 3.13
```

Código fuente 25: Terminal commands to run the script that tests the Python script with different Python versions using Docker containers.

ronment without the need for a full virtual machine. This makes Docker more lightweight and efficient compared to Hypervisor-based virtualization.

2. Erläutern Sie den Begriff “Container” im Kontext von Docker.

   In the context of Docker, a container is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. Containers are created from Docker images, which are read-only templates that define the contents and configuration of the container. When a container is run, it provides an isolated environment for the application to execute, ensuring that it runs consistently across different environments and platforms. Containers share the host operating system’s kernel but have their own filesystem, network interfaces, and process space, allowing for efficient resource utilization and fast startup times.

3. Wie kann Docker in einer Continuous-Integration/Continuous-Deployment-(CI/CD)-Pipeline eingesetzt werden?

   After the CI server builds the application and runs the tests, a Docker image can be created with the application and its dependencies. This image can then be pushed to a Docker registry, which is a repository for storing and distributing Docker images. In the CD stage, the Docker image can be pulled from the registry and deployed to the production environment, ensuring that

```
$ sudo chroot . /bin/sh
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ # pwd
/
/ # ps
PID   USER     TIME  COMMAND
/ #
```

Código fuente 26: Command used to start a shell within the chroot environment.

the same image that was tested in the CI stage is the one that is deployed. This allows for consistent and reliable deployments, as well as easy rollbacks if any issues arise.

4. Erläutern Sie den Begriff "Docker Registry" und erläutern Sie, warum er für CI/CD wichtig ist.

   A Docker Registry is a repository for storing and distributing Docker images. It allows developers to share their images with others and provides a central location for managing and versioning images. In the context of CI/CD, a Docker Registry is important because it allows for the storage and distribution of Docker images that are built during the CI stage. This means that the same image that was tested in the CI stage can be easily pulled and deployed in the CD stage, ensuring consistency and reliability in the deployment process. Additionally, a Docker Registry can also provide features such as access control, image scanning, and vulnerability management, which are important for maintaining the security and integrity of the images used in the CI/CD pipeline.

Ejercicio 7.5.4. In dieser Aufgabe lernen Sie einen der Grundmechanismen der *historischen* Containerisolierung kennen. Dazu verwenden Sie das UNIX-Werkzeug `chroot`, das den sichtbaren Root-Ordner eines Prozesses ändert. In der `zip`-Datei finden Sie die Datei `alpine-rootfs.tar`. Diese Datei enthält ein minimales Linux-Dateisystem, das ursprünglich aus einem Docker-Container (`alpine`) exportiert wurde.

1. Entpacken Sie das Root-Dateisystem in ein Verzeichnis Ihrer Wahl.
2. Starten Sie eine Shell innerhalb dieses Dateisystems mithilfe von `chroot`.

   After unzipping the root filesystem, the command shown in the Listing 26 is used to start a shell within the chroot environment.

3. Untersuchen Sie das Verhalten innerhalb der Umgebung:
   - Führen Sie Befehle wie `ls`, `pwd` und `ps` aus. Was fällt Ihnen auf?

     As observed in the Listing 26, the root directory is now the chroot environment, and given that there are no other processes running within the chroot, the `ps` command shows no processes.

```
#!/bin/bash

ROOTFS="$HOME/alpine-rootfs"
ARCHIVE="alpine-rootfs.tar"

mkdir -p $ROOTFS
tar -xf $ARCHIVE -C $ROOTFS

echo "Entering chroot environment..."
sudo chroot $ROOTFS /bin/sh -c "echo 'Inside chroot: '; pwd; ls; exec
/bin/sh"
```

Código fuente 27: Script used to automate the process of unzipping the root filesystem and starting a shell within the chroot environment.

- Starten Sie in einem separaten Terminal `ps -ef` aus. Können Sie den Prozess aus der `chroot`-Umgebung sehen?

  Yes, the process running within the `chroot` environment can be seen from the host system using the `ps -ef` command.

4. Schreiben Sie ein kleines Programm/Script, das die Schritte des Entpackens sowie das Starten eines Befehls in der `chroot`-Umgebung automatisiert.

   The script used to automate the process is shown in the Listing 27.

5. Begründen Sie, warum `chroot` *keine vollständige Isolation* bietet.

   This command only changes the apparent root directory for the process, but it does not provide isolation in terms of processes, network, or users. The process running within the `chroot` environment can still see all the processes running in the host system, no network isolation is provided, and if the process has root privileges, it can escape the jail.

7.6. Deployment

Ejercicio 7.6.1. Die *Echtzeit-Messaging-App* für den Campus der "Universität der Zukunft" soll bereitgestellt werden. Die App soll zunächst auf Android-Telefonen verfügbar gemacht werden. Es gibt außerdem einen Server, der Anfragen der App verarbeitet und Nachrichten speichert.

1. Wie sollte man das erste Deployment der App gestalten?

   For the first deployment of the app, it is important to ensure that the app is stable and does not have any major bugs. It is also important to ensure that the app is easy to use and has a good user interface. One way to achieve this is to do a beta release of the app to a small group of users, and gather feedback from them before doing a full release. This way, any issues can be identified and fixed before the app is released to a larger audience.

2. Eine neue Version der App ist fertig entwickelt. Lohnt sich ein Zero-Downtime-Release?

   Yes, a zero-downtime release would be beneficial for the app, especially if it is already being used by a large number of users. Given that the app is a real-time messaging app, it is important to minimize downtime as much as possible, as users may rely on the app for communication. A zero-downtime release would allow users to continue using the app without interruption while the new version is being deployed.

3. Was sollte man bei diesem Release beachten?

   Apart from ensuring that it is a zero-downtime release, monitoring the app during and after the release is important to ensure that there are no issues. It is also important to have a rollback plan in case any issues arise during the release. Additionally, it is important to communicate with users about the release and any changes that may affect them.

4. Wann sind Blue-Green Deployments sinnvoll?

   Blue-Green Deployments are useful when you want to minimize downtime and reduce the risk of deployment failures. This allows instant rollback to the previous version if any issues arise during the deployment. However, the hardware requirements for this type of deployment can be high, as it requires maintaining two separate environments (blue and green) or a single environment with enough resources to run both versions of the application simultaneously. Therefore, it may not be suitable for all projects, especially those with limited resources.

5. Wann sind Canary Releases sinnvoll?

   Canary Releases are useful when you want to test a new version of the application with a small subset of users before rolling it out to the entire user base. This allows you to identify any issues or bugs in the new version before it affects all users. It is particularly beneficial for applications with a large user base, as it helps to minimize the impact of any potential issues that may arise during the deployment.

6. Was passiert in unserer App während der Commit Stage?

   During the Commit Stage, the code changes are committed to the version control system (e.g., Git), and the CI/CD pipeline is triggered. This stage typically includes building the application, running unit tests, and performing static code analysis to ensure that the code meets quality standards. If any issues are found during this stage, the pipeline will fail, and the developers will need to address the issues before proceeding to the next stage.

7. Was passiert in der Automated Acceptance Stage?

   During the Automated Acceptance Stage, automated tests are run to verify that the application meets the acceptance criteria defined for the release. This may include functional tests, integration tests, and end-to-end tests. The goal of this stage is to ensure that the application behaves as expected and that any new features or changes do not introduce regressions or break existing functionality.

8. Was passiert in der manual Test Stage?

   During the Manual Test Stage, human testers manually test the application to identify any issues that may not have been caught by automated tests. This may include exploratory testing, usability testing, and user acceptance testing. The goal of this stage is to ensure that the application is user-friendly and meets the needs of the end-users.

9. Was passiert in der Release Stage?

   During the Release Stage, the new version of the application is deployed to production. This may involve deploying to a staging environment first for final testing before deploying to production. The goal of this stage is to ensure that the deployment process goes smoothly and that any issues that arise during deployment can be quickly addressed.

10. Würden Sie eher Continuous Deployment oder Continuous Delivery für das Projekt nutzen? Argumentieren Sie.

    For this project, I would recommend using Continuous Delivery. This is because Continuous Delivery allows for more control over the release process, as it requires a manual approval step before deploying to production.

Ejercicio 7.6.2.

1. Warum lohnt es sich, Container in der Entwicklung und in der Deployment-Pipeline zu verwenden?

   Using containers in development and deployment pipelines can provide several benefits. Containers allow for consistent environments across different stages of the development and deployment process, which can help to reduce issues related to environment differences. They also allow for easier scaling and management of applications, as containers can be easily deployed and orchestrated using tools like Docker and Kubernetes. Additionally, containers can help to improve resource utilization, as multiple containers can run on a single host without the overhead of a full virtual machine.

```
$ cat /sys/fs/cgroup/system.slice/docker.service/cpu.max
max 100000
$ cat /sys/fs/cgroup/system.slice/docker.service/memory.max
max
```

Código fuente 28: Example of the content of the `cpu.max` and `memory.max` files of a `cgroup`.

2. Welche Eigenschaften von `cgroups` sind bei der Containerization nützlich?

   Control Groups (`cgroups`) provide several useful features for containerization. They allow for resource allocation and limitation, which means that containers can be restricted in terms of CPU, memory, and other resources they can use. This helps to ensure that no single container can consume all the resources of the host system, which can lead to performance issues.

3. In ihrer Ubuntu-VM sollten sie unter `/sys/fs/cgroup/system.slice/docker.service` die Dateien finden die die "docker.service"-`cgroup` definieren. Schauen Sie sich `cpu.max` und `memory.max` an. Was sagen sie über diese `cgroup` aus?

   In the Listing 28 you can see an example of the content of these files. In this case, the `cgroup` is configured to use as much CPU and memory as needed, but it could be configured to use only a percentage of the CPU or a fixed amount of memory.

4. Welche Eigenschaften haben Namespaces, die bei der Containerisierung nützlich sind?

   Namespaces provide several useful features for containerization. They allow for isolation of resources, which means that each container can have its own view of the system resources, such as process IDs, network interfaces, and file systems. This helps to ensure that containers do not interfere with each other and can run independently. Additionally, namespaces can help to improve security, as they can limit the visibility of resources to only those that are necessary for the container to function.

5. Welche Eigenschaften hat `chroot` bzw. `pivot_root`, die bei der Containerization nützlich ist, und was unterscheidet die beiden?

   Both `chroot` and `pivot_root` are used to change the root directory, but they have different use cases. `chroot` changes the root directory for the current process and its children, while `pivot_root` changes the root directory for the entire system. In containerization, `pivot_root` is more commonly used, as it allows for a complete change of the root file system for the container, while `chroot` is more limited in scope.

6. Was unterscheidet Containerization von Virtual Machines?

   As already explained, containerization uses directly the host operating system's kernel, while virtual machines run a full guest operating system on top of a hypervisor. This means that containers are generally more lightweight

and have lower overhead than virtual machines, as they do not require a full operating system to be running. Additionally, containers can be started and stopped more quickly than virtual machines, as they do not require the same level of initialization.

7.7. Secure Deployment

Ejercicio 7.7.1. Die *Echtzeit-Messaging-App* für den Campus der "Universität der Zukunft" soll regelmäßig aktualisiert werden, dabei aber sichere Deployments garantiert werden.

1. Erklären Sie kurz die Begriffe hermetic build, reproducible build und verifiable build.

   The building process should be hermetic, reproducible, and verifiable.

   - Hermetic: The build process is isolated from the outside environment, ensuring that it does not rely on any external factors that could introduce variability or security risks.
   - Reproducible: The build process can be repeated with the same inputs and produce the same outputs, allowing for consistent and reliable builds.
   - Verifiable: The build process can be independently verified to ensure that it has not been tampered with and that the resulting build artifacts are trustworthy.

2. Nennen Sie zwei typische Fallstricke, die reproducible Builds verhindern.

   - Timestamps: If the build process includes timestamps, it can lead to non-reproducible builds, as the output will differ each time due to the changing timestamps.
   - Non-deterministic inputs: If the build process relies on non-deterministic inputs, such as random numbers or environment variables, it can also lead to non-reproducible builds.

Ejercicio 7.7.2. Die Universität überlegt, eine eigene CA für interne Services einzurichten.

1. Erklären Sie, wofür eine CA benötigt wird, und welche Rolle Zertifikate beim sicheren Schlüsselaustausch spielen.

   A CA (Certificate Authority) is needed to issue digital certificates that verify the identity of entities (such as servers, users, or devices) and facilitate secure communication. Certificates play a crucial role in secure key exchange by providing a way to establish trust between parties, as they garantee that the public key contained in the certificate belongs to the entity it claims to represent. This allows for secure communication through encryption and authentication.

2. Skizzieren Sie den Ausstellungsprozess eines Zertifikats.

   The certificate issuance process typically involves the following steps:

   a) The entity (e.g., a server) generates a public-private key pair and creates a Certificate Signing Request (CSR) that includes the public key and identifying information about the entity.

b) The CSR is sent to the CA, which verifies the identity of the entity through various means (e.g., email verification, domain ownership verification).

c) Once the CA is satisfied with the verification, it signs the CSR with its private key, creating a digital certificate that binds the public key to the entity's identity.

d) The signed certificate is then returned to the entity, which can use it for secure communication and authentication.

3. Nennen Sie drei Risiken beim Betrieb einer CA und mögliche Gegenmaßnahmen.

   - Compromise of the CA's private key: If an attacker gains access to the CA's private key, they can issue fraudulent certificates. To mitigate this risk, the CA's private key should be stored securely, such as in a hardware security module (HSM), and access should be strictly controlled.
   - Misissuance of certificates: If the CA issues certificates to unauthorized entities, it can lead to security breaches. To prevent this, the CA should implement strict verification processes for certificate requests and regularly audit issued certificates.
   - Revocation of certificates: If a certificate is compromised or no longer valid, it needs to be revoked. The CA should have a robust certificate revocation mechanism in place, such as a Certificate Revocation List (CRL) or Online Certificate Status Protocol (OCSP), to ensure that revoked certificates are not trusted.

Ejercicio 7.7.3. Angenommen, ein Angreifer hat Zugriff auf einen CI-Runner erlangt, der Builds ausführt.

1. Welche zwei Angriffe sind dadurch besonders naheliegend?

   - Injection of malicious code: The attacker could inject malicious code into the build process, which could then be included in the final build artifacts and potentially distributed to users.
   - Exfiltration of sensitive information: The attacker could access sensitive information such as API keys, credentials, or proprietary code that is used during the build process.

2. Welche Sofortmaßnahme (erste 24h) würden Sie einleiten?

   The immediate response would be to isolate the compromised CI-Runner to prevent further damage. This could involve taking the runner offline, revoking any credentials that were used on that runner, and conducting a thorough investigation to determine the extent of the breach and identify any potential vulnerabilities that were exploited. Additionally, it would be important to communicate with stakeholders about the incident and implement measures to prevent similar attacks in the future.

Ejercicio 7.7.4. Schauen Sie sich SLSA an. Worum handelt es sich dabei?

SLSA (Supply-chain Levels for Software Artifacts) is a security framework that provides a set of guidelines and best practices for securing the software supply chain. It defines different levels of security assurance for software artifacts, ranging from basic to advanced, based on the measures taken to ensure the integrity and authenticity of the software throughout its lifecycle. The framework aims to help organizations improve the security of their software supply chain and reduce the risk of vulnerabilities and attacks.

7.8. Secure Deployment 2

Ejercicio 7.8.1. Erläutern Sie, inwiefern die folgenden Maßnahmen vor einem Benign Insider oder einem Malicious Adversary absichern. Gegen welche der beiden Arten von Akteuren sind die Maßnahmen effektiver und warum?

1. Code Reviews
2. Geheimnisse schützen (durch Key-Management-Systeme, Zugriffskontrollen etc.)
3. Automatisierung der CI/CD-Pipeline
4. Verifiable Builds

Ejercicio 7.8.2.

1. Welche Daten sollten in einer Binary Provenance für unsere Echtzeit-Messaging-App des Campus der "Universität der Zukunft" enthalten sein?
2. Welche Verbindung besteht zwischen Hermetic, Reproducible und Verifiable Builds und Binary Provenance?

7.9. Secure Development

Ejercicio 7.9.1.

1. Erläutern Sie, was ein Security Champion ist.
2. Welche Vor- und Nachteile hat dieses Konzept?

Ejercicio 7.9.2. Nehmen Sie wieder an, Sie seien Entwickler im Team der Echtzeit-Messaging-App der "Universität der Zukunft". Ihr Team denkt über das Thema Sicherheit nach und möchte das `OWASP SAMM` einführen.

1. Schauen Sie sich das `OWASP SAMM` an.
2. Überlegen Sie sich verschiedene Maßnahmen auf unterschiedlichen Ebenen, um eine sichere Benutzerauthentifizierung sicherzustellen. (Mögliche Hilfestellung kann das `OWASP CheatSheet` bieten.)
3. Überlegen Sie, zu welcher Domäne des `OWASP SAMM`s die Maßnahmen passen und inwiefern Sie den Reifegrad verbessern können.

```python
def func(a, b):
    x = 0
    y = 0
    if a > 0:
        x = 1
    if b == 0:
        y = 2
    assert x + y != 3
```

Código fuente 29: Beispielcode für Übung 7.10.4

7.10. Fuzzing & Z3

Ejercicio 7.10.1. Schauen Sie sich das Fuzzingbook an. Sie können den benötigten Code mittels `$ pip install fuzzingbook` installieren. Beschreiben Sie die folgenden Ansätze und erklären Sie die Unterschiede sowie die Vor- und Nachteile.

1. Random Fuzzing
2. Mutation Fuzzing
3. Coverage-guided Fuzzing

Ejercicio 7.10.2.

1. Schauen Sie sich an, welche SMT-Solver es gibt.
2. Nennen Sie einige Anwendungsgebiete, in denen SMT-Solver eingesetzt werden können.
3. Welche Herausforderungen können bei der Anwendung von SMT-Solvern auftreten und wie können sie bewältigt werden?
4. Wie können SMT-Solver zur Sicherheitsanalyse von Softwareanwendungen beitragen?

Ejercicio 7.10.3.

1. Warum sind AddressSanitizer (ASan) nicht für den Produktivbetrieb geeignet?
2. Was sind Redzones?
3. Warum wird Shadow Memory benötigt?

Ejercicio 7.10.4. Schauen Sie sich das folgende Python-Programm an (Abbildung 29).

1. Zeichnen Sie den Kontrollflussgraphen.

   The graphic can be seen in the Figure 7.9.

2. Leite alle möglichen Pfadbedingungen her.

   This can also be seen in the Figure 7.9, where the path constraints are shown in green boxes.

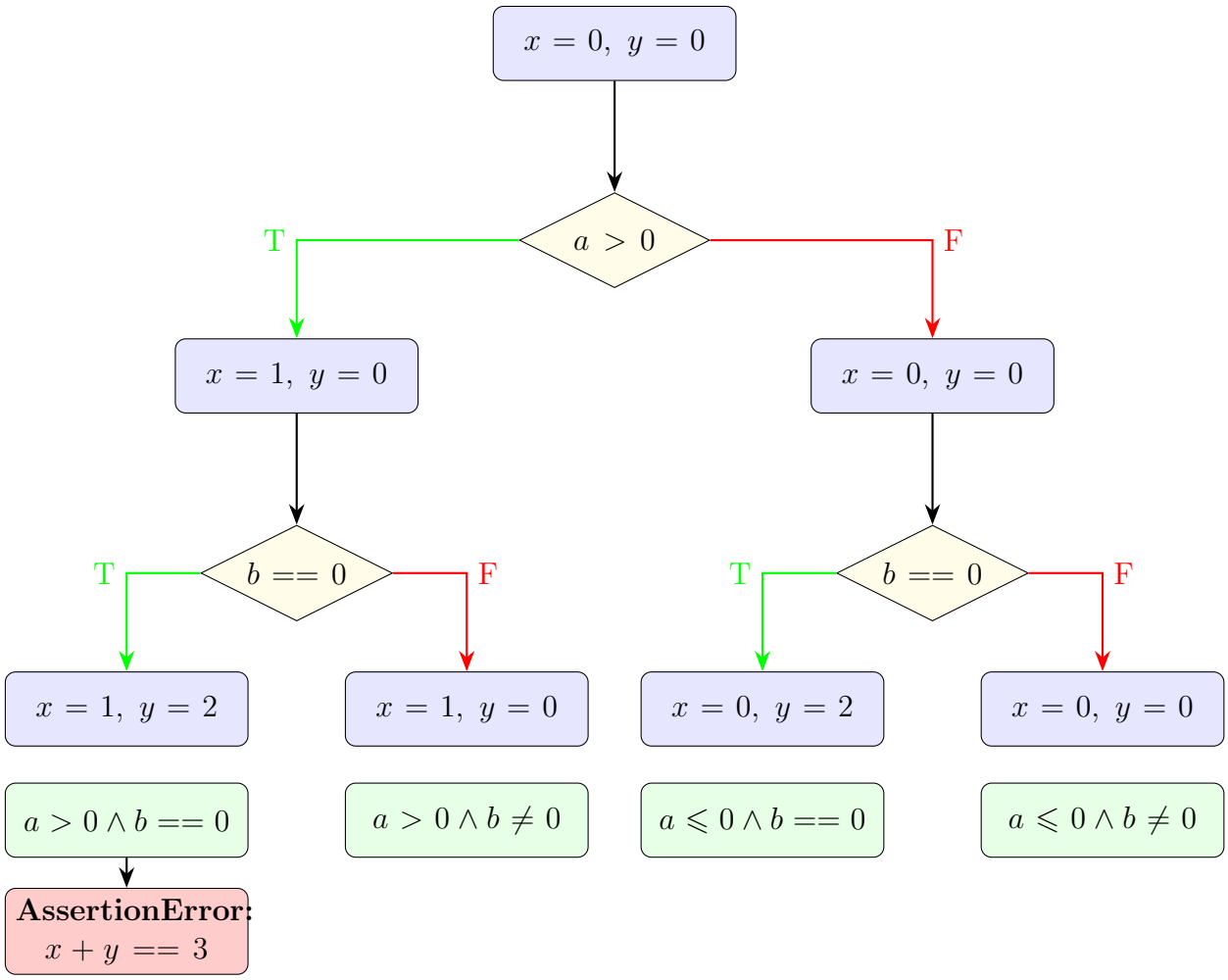

Figura 7.9: Kontrollflussgraph für das Beispielprogramm

3. Gibt es eine Belegung von `a` und `b`, die das Assert verletzt? Wenn ja: Welche?

   As shown in Figure 7.9, there is a path that leads to the assertion failure, which is the path where `a > 0` and `b == 0`. Therefore, any values of `a` and `b` that satisfy these conditions will violate the assertion. For example, `a = 1` and `b = 0` would lead to the assertion failure.

7.11. Fuzzing & Z3 2

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen. Sie müssen auch ein paar Python-Pakete installieren, um die Aufgaben zu lösen.

1. Öffnen Sie das Terminal (`ctrl + Alt + T`)
2. Installieren Sie den Package Installer for Python
   (`$ sudo apt install python3-pip python3-venv`)
3. Erstellen Sie ein neues Virtual Environment
   (`$ python3 -m venv ./venv`)
4. Konfigurieren Sie die aktuelle Shell, damit sie das `venv` verwendet.
   (`$ source ./venv/bin/activate`)
5. Installieren Sie `z3`
   (`$ pip3 install z3-solver`)
6. Atheris können Sie in einem Ubuntu 24.04 Container installieren
   (`$ pip3 install atheris`)

Ejercicio 7.11.1.

1. Nutzen Sie `atheris` und instrumentieren Sie die `validate`-Funktion in `fuzz.py`.

   The new code should look like Listing 30.

2. Zeichnen Sie einen Kontrollflussgraphen für die Funktion. Nutzen Sie als Knotennamen die Zeilen-nummern.

   The control flow graph for the `validate` function is shown in Figure 7.10.

3. Bestimmen Sie die prozentuale Coverage, wenn der Code mit `[246,63,103,121]` aufgerufen wird. Markieren Sie außerdem die erreichten Knoten im Kontrollflussgraphen.

4. Instrumentieren Sie die `validate`-Funktion in `fuzz2.py`

   The new code should look like Listing 31.

5. Suchen Sie mittels `z3` nach einer Lösung für `validate`.

   The solution for the `validate` function is shown in Listing 32. It should be noted that the code has been "translated" so that `z3` can know the constraints that need to be satisfied.

Ejercicio 7.11.2. Manche Programmierer nutzen Bit-Tricks, um auf spezifische Hardware zu optimieren. Nutzen Sie `z3`, um zu zeigen, dass die Tricks auf einem 64-Bit-System dasselbe Verhalten haben wie eine naive Implementierung.

1. Tauschen zweier Variablen mit `XOR`

   The code for swapping two variables using XOR is shown in Listing 33, while the output of the code is shown in Listing 34.

```python
#!./venv/bin/python3

from typing import List
import atheris
import sys

# PASSCODE = [getrandombits(8) for _ in range(4)]
PASSCODE = [64, 63, 121, 119]


def validate(data: List[int]):
    if len(data) != 4:
        return

    if PASSCODE[0] == data[0]:
        if PASSCODE[1] == data[1]:
            if PASSCODE[2] == data[2]:
                if PASSCODE[3] == data[3]:
                    raise RuntimeError("passcode match")
    return


atheris.instrument_func(validate)
atheris.Setup(sys.argv, validate)
atheris.Fuzz()
```

Código fuente 30: Fuzzing mit Atheris

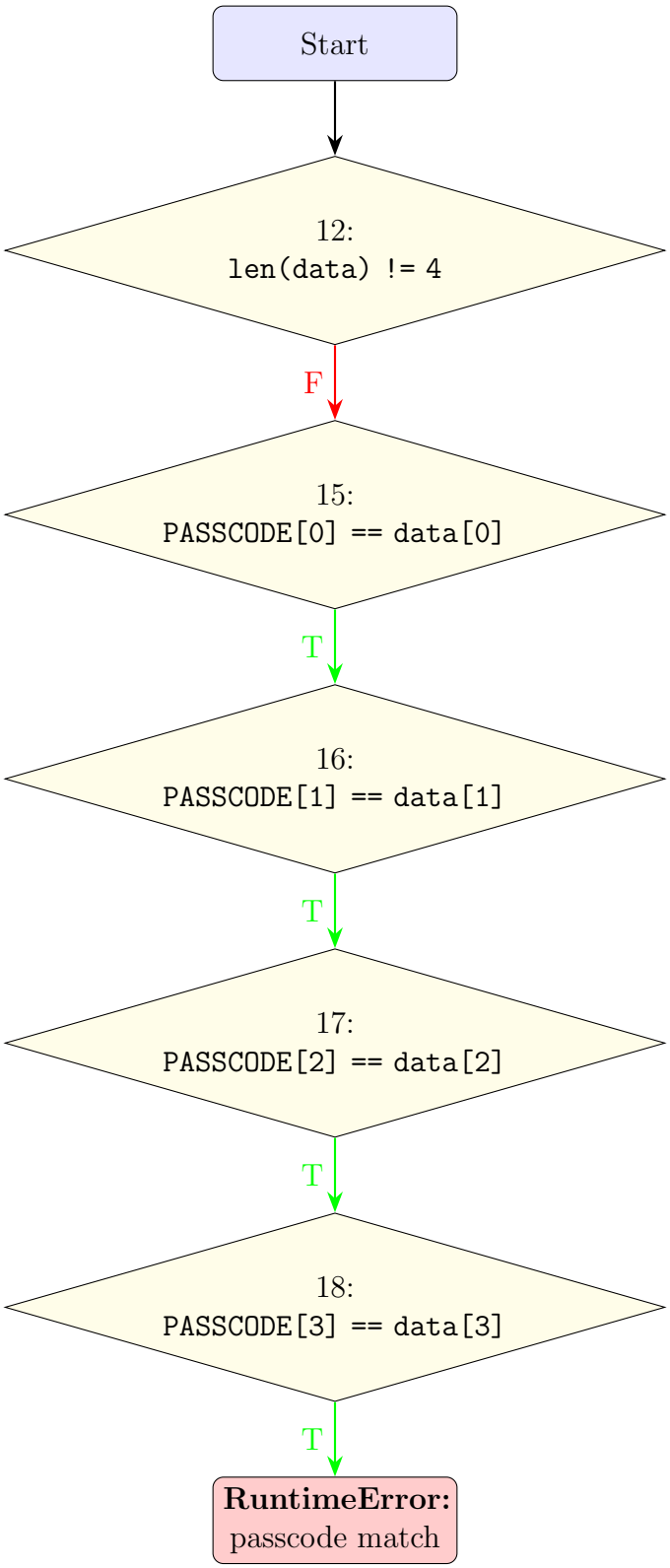

Figura 7.10: Kontrollflussgraph für das `validate`-Programm

```python
#!./venv/bin/python3
from typing import List
import atheris
import sys

# PASSCODE = [getrandombits(8) for _ in range(4)]
PASSCODE = [64, 63, 121, 119]


def validate(data: List[int]):
    if len(data) != 4:
        return

    if PASSCODE[0] == data[0]:
        if PASSCODE[1] % 3 == data[1]:
            if PASSCODE[2] * 2 + 4 == data[2]:
                if PASSCODE[3] + PASSCODE[0] == data[3]:
                    raise RuntimeError("passcode match")
    return


atheris.instrument_func(validate)
atheris.Setup(sys.argv, validate)
atheris.Fuzz()
```

Código fuente 31: Fuzzing mit Atheris (veränderte `validate`-Funktion)

```python
#!./venv/bin/python3
from typing import List
import z3

# PASSCODE = [getrandombits(8) for _ in range(4)]
PASSCODE = [64, 63, 121, 119]

solver = z3.Solver()
data = [z3.Int(f"data_{i}") for i in range(4)]

solver.add(len(data) == 4)
solver.add(PASSCODE[0] == data[0])
solver.add(PASSCODE[1] % 3 == data[1])
solver.add(PASSCODE[2] * 2 + 4 == data[2])
solver.add(PASSCODE[3] + PASSCODE[0] == data[3])

if solver.check() == z3.sat:
    model = solver.model()
    solution = [model[data[i]].as_long() for i in range(4)]
    print("Solution found:", solution)
else:
    print("No solution found.")
```

Código fuente 32: Lösungssuche mit Z3

```python
#!./venv/bin/python3

import z3

x, y = z3.BitVecs('x y', 64)
x0, y0 = x, y

x = x ^ y
y = x ^ y
x = x ^ y

solver = z3.Solver()
solver.add(z3.Not(z3.And(x == y0, y == x0)))

if solver.check() == z3.sat:
    model = solver.model()
    print("Counterexample found:", model)
elif solver.check() == z3.unsat:
    print("No counterexample found, the property holds.")
else:
    print("Solver returned unknown result.")
```

Código fuente 33: Tauschen zweier Variablen mit XOR

```
$ ./test_xor.py
No counterexample found, the property holds.
```

Código fuente 34: Output des XOR-Codes

```
#!./venv/bin/python3

import z3

x, y = z3.BitVecs('x y', 64)
x0, y0 = x, y

x = x - y
y = x + y
x = y - x

solver = z3.Solver()
solver.add(z3.Not(z3.And(x == y0, y == x0)))

if solver.check() == z3.sat:
    model = solver.model()
    print("Counterexample found:", model)
elif solver.check() == z3.unsat:
    print("No counterexample found, the property holds.")
else:
    print("Solver returned unknown result.")
```

Código fuente 35: Tauschen zweier Variablen mit Subtraktion und Addition

2. Tauschen zweier Variablen mit Subtraktion und Addition

   The code for swapping two variables using subtraction and addition is shown in Listing 35, while the output of the code is shown in Listing 36.

3. Die Bitreihenfolge innerhalb eines Bytes umkehren

Ejercicio 7.11.3. Nutzen Sie `z3`, um generische Sudokus zu lösen.

Observación. Sollten Sie nicht weiterkommen, kann eine Internetsuche weiterhelfen.

The code for solving generic Sudokus using `Z3` is available in this repository.

Ejercicio 7.11.4. Ein Freund von Ihnen behauptet, dass er sein eigenes sicheres Krypto-System entwickelt hat. Zum System gehören zwei Komponenten: eine asym-

```
$ ./test_add_sub.py
No counterexample found, the property holds.
```

Código fuente 36: Output des Subtraktion-Addition-Codes

metrische und eine symmetrische Chiffre. Zeigen Sie, dass beide Verfahren nicht sicher sind.

7.12. SLA

Ejercicio 7.12.1. Beschreiben Sie, worum es sich bei MTTF, MTTR und MTBF handelt, und stellen Sie die Unterschiede dar.

They are described in Section 1.3.1.3.

Ejercicio 7.12.2. Nehmen Sie an, Sie sind ein Teammitglied des Netzwerkzentrums der "Universität der Zukunft". Die Echtzeit-Messaging-App soll von Ihnen gehostet werden und die Rektorin möchte, dass das System hochverfügbar (VK3) ist. Erstellen Sie eine Liste möglicher Prozesse, die die Ausfallzeit minimieren.