AI DOCKER LINTER AND OPTIMISER

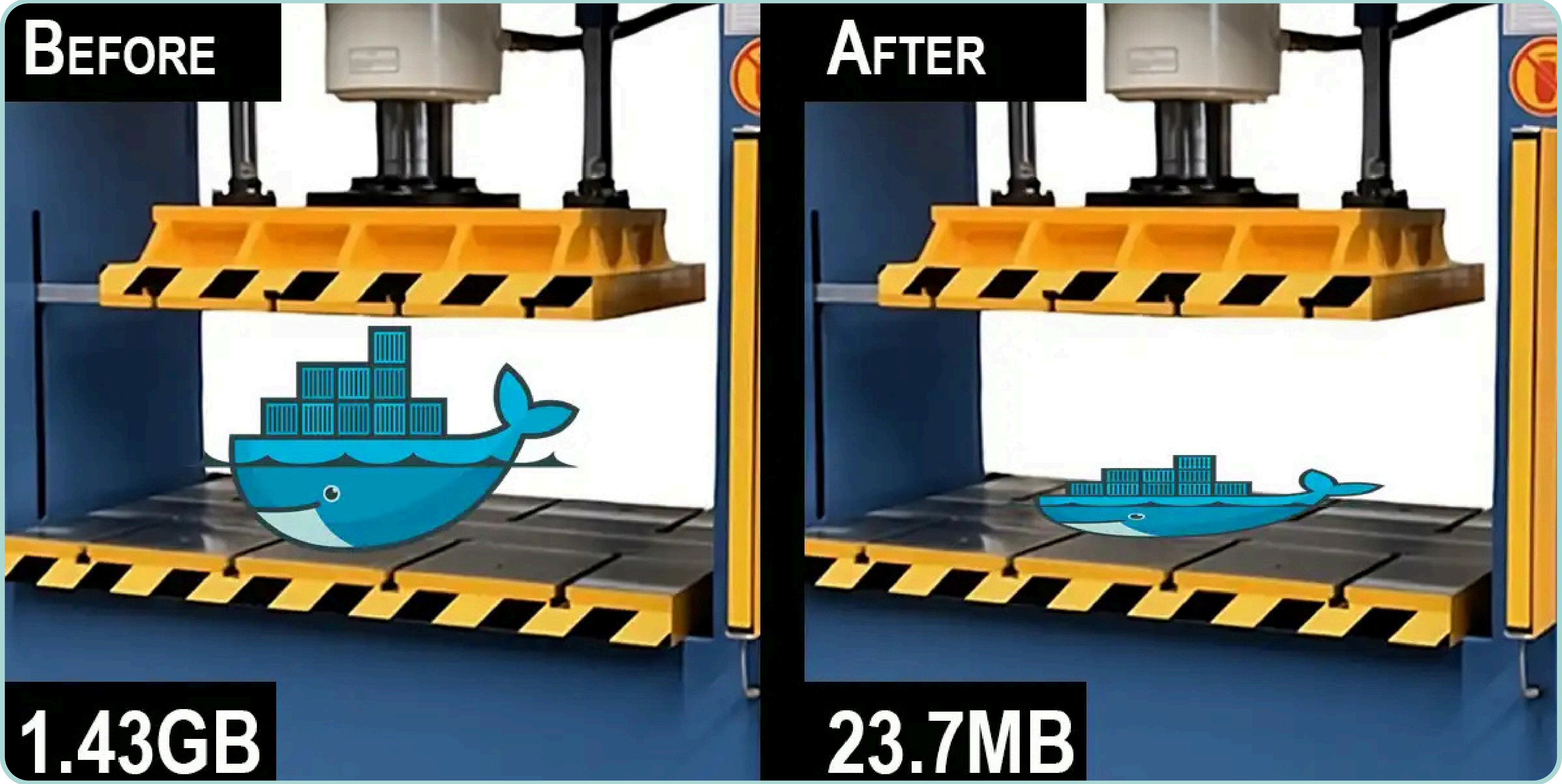
PRESENTED BY PETER SHEEHAN

Origin of the Idea

As an intern at IBM, my colleague highlighted an issue that they have to face almost every week. They had to manually check dockerfiles for issues. This was time consuming and tedious for the developer leading me to create this tool; a tool designed to analyse and optimise dockerfiles for security, scalability and efficiency.

Exploring creativity

The Problem: The Hidden Costs of Unoptimised Dockerfiles



Key Challenges

- Massive Image Sizes
- Painfully Slow Build Times
- Critical Security Vulnerabilities
- Poor Scalability

Project Overview

- The Dockerfile AI Optimiser & Linter automates the traditionally manual and error-prone task of reviewing Dockerfiles.
- This tool combines a custom linter with GPT-4o based AI to analyse Dockerfiles for security, efficiency, and scalability.
- It provides actionable linting feedback and intelligent optimisation suggestions, aiming to produce smaller, faster, and more secure Docker images, thereby freeing up developer time and improving overall containerisation practices

Enhanced Developer Productivity

Automates tedious manual Dockerfile reviews, freeing developers to focus on core tasks and accelerating development cycles."

Optimised & Cost-Efficient Image

Delivers smaller, more efficient Docker images, reducing storage costs, speeding up deployments, and improving CI/CD pipeline performance.

Improved Quality & Security Standards

Enforces best practices and helps identify security vulnerabilities early, leading to more robust, secure, and maintainable containerised applications

KEY FEATURES

Automated Analysis & Optimisation
Rapidly scans Dockerfiles with an automated linter and leverages AI for intelligent, efficiency-focused optimization suggestions

Insights & Actionable Outputs
Delivers clear explanations for AI-driven changes and provides optimized Dockerfile versions, alongside comprehensive console, JSON, and CSV reports

Seamless Developer Workflow & Collaboration
Integrates smoothly via a versatile CLI for local/GitHub analysis and facilitates teamwork with automated GitHub Pull Request generation for suggested improvements

Functional Requirements

Dockerfile Linting & Analysis:

AI-Powered Optimisation

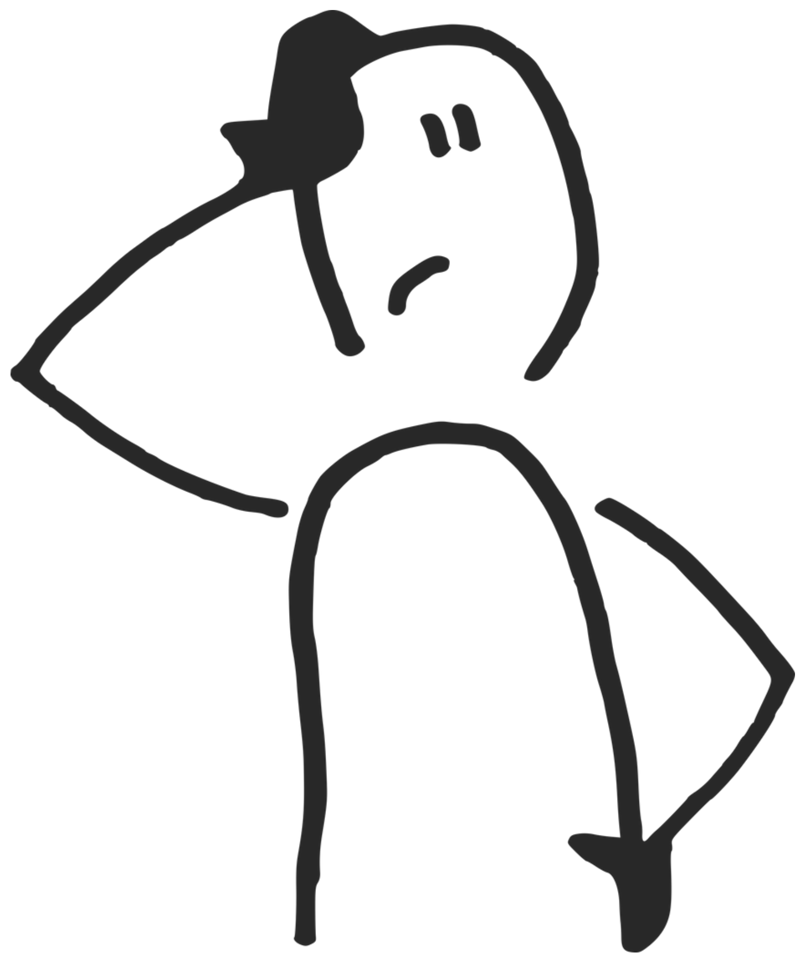
Versatile Input & Output

User Interaction

Non-Functional Requirements

- Accuracy & Reliability
- Maintainability & Extensibility
- Compatibility:
- Resource Management:

Challenges



Managing Package Version Pinning Across Distributions

A major difficulty is providing reliable suggestions for pinned package versions

Translating Linter Rules from Natural Language to Precise Logic

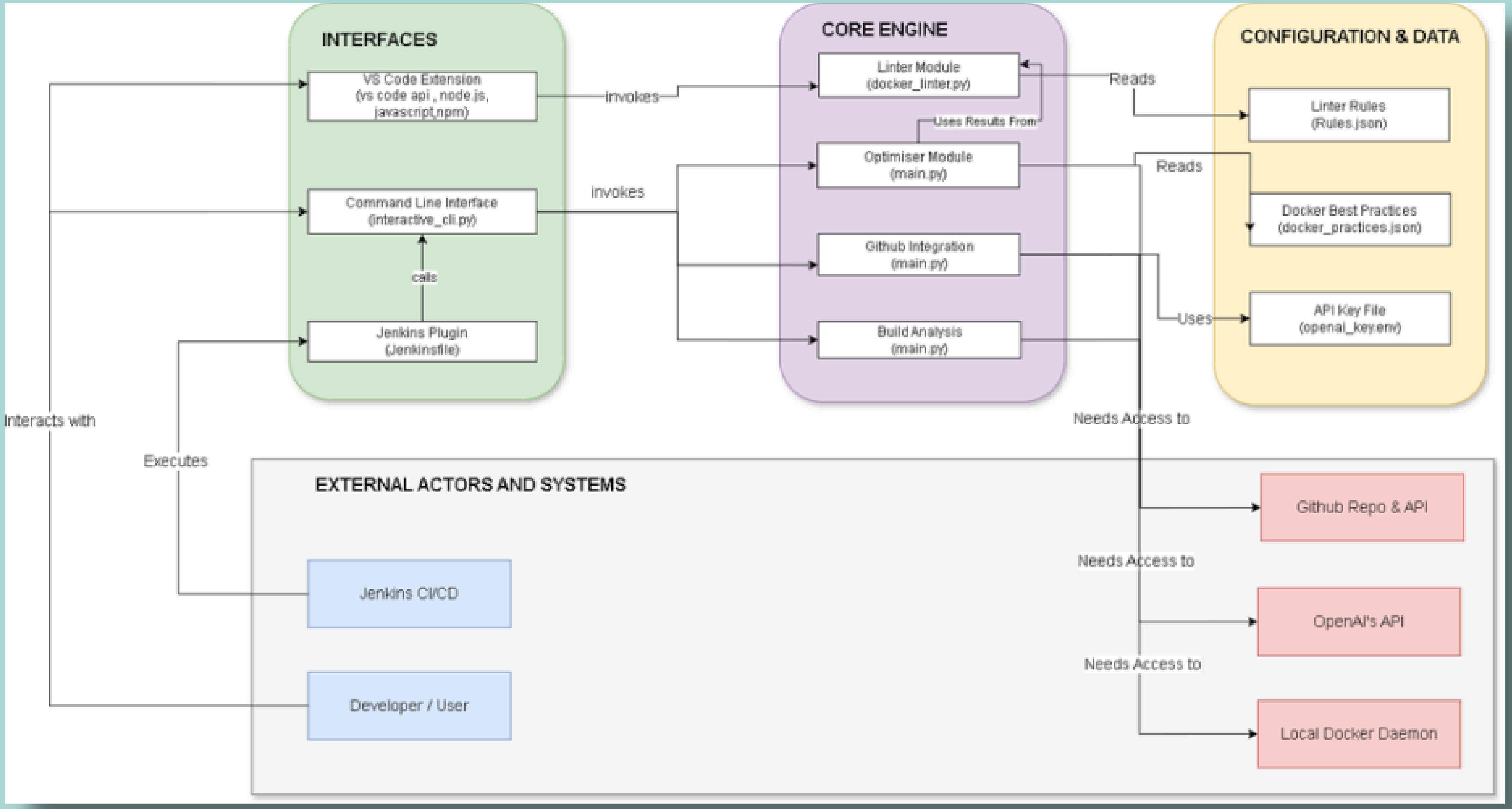
natural language (English), translating them into a formal, machine-interpretable format like regular expressions that the AI can use for analysis is a complex challenge

Implementing Dockerfile Optimisations

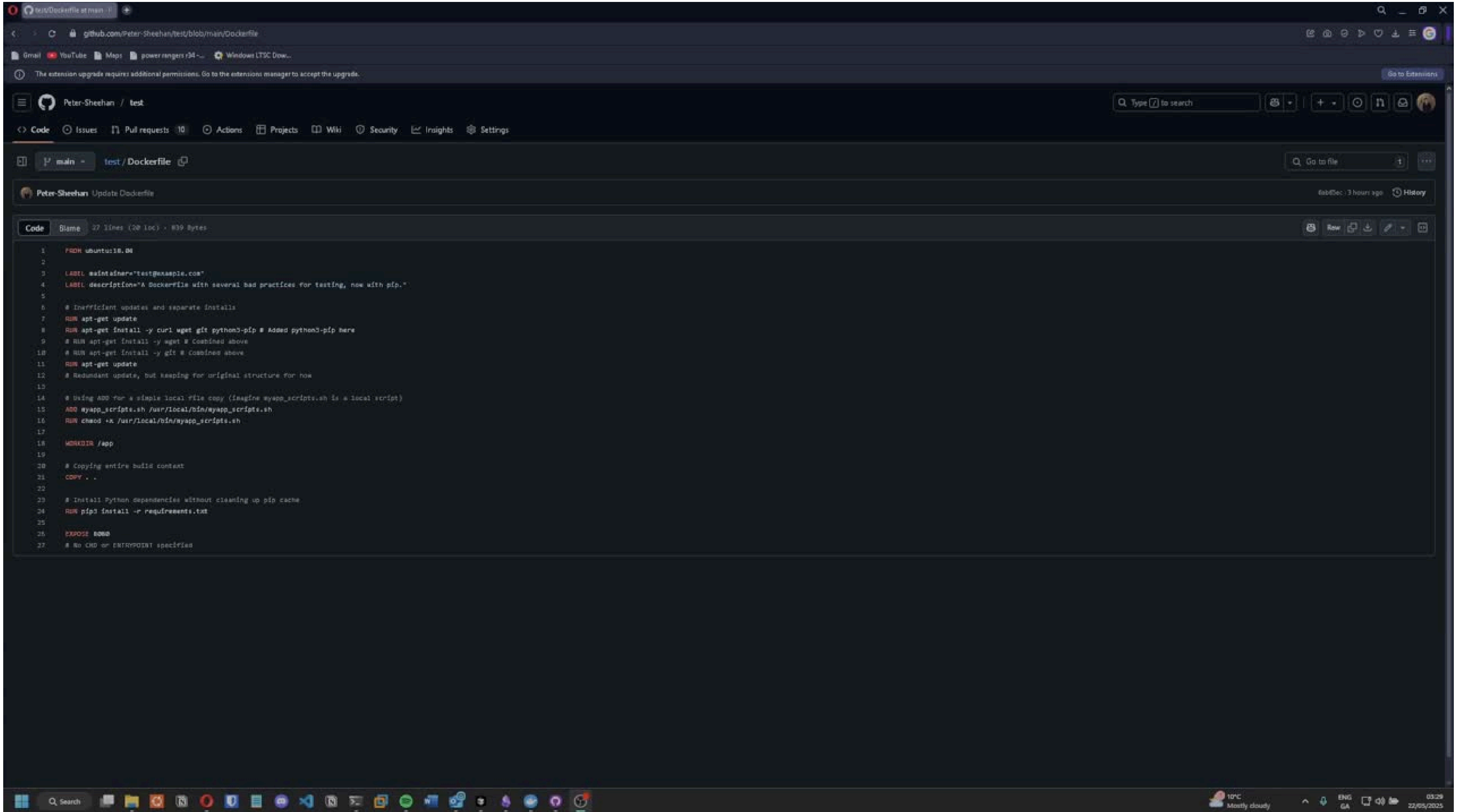
Applying linter recommendations (multi-stage builds, non-root users, combining RUN steps, using COPY instead of ADD, pinning versions) to our own Dockerfile required careful testing to avoid build or runtime breakages

Project Design

- **Interfaces (User & System Interaction):**
  - Command Line Interface (CLI)
  - VS Code Extension
  - Jenkins.
- **Core Engine (Central Processing Logic):**
  - Linter Module
  - AI Optimiser Module
  - GitHub Integration
  - Build Analysis Module
- **Configuration & Data (Knowledge Base):**
  - Linter Rules (Rules.json)
  - Docker Best Practices (docker_best_practices.json):.
  - API Keys (.env)
- **External Dependencies (Essential Services):**
  - GitHub API
  - OpenAI API
  - Local Docker Daemon.



Demo

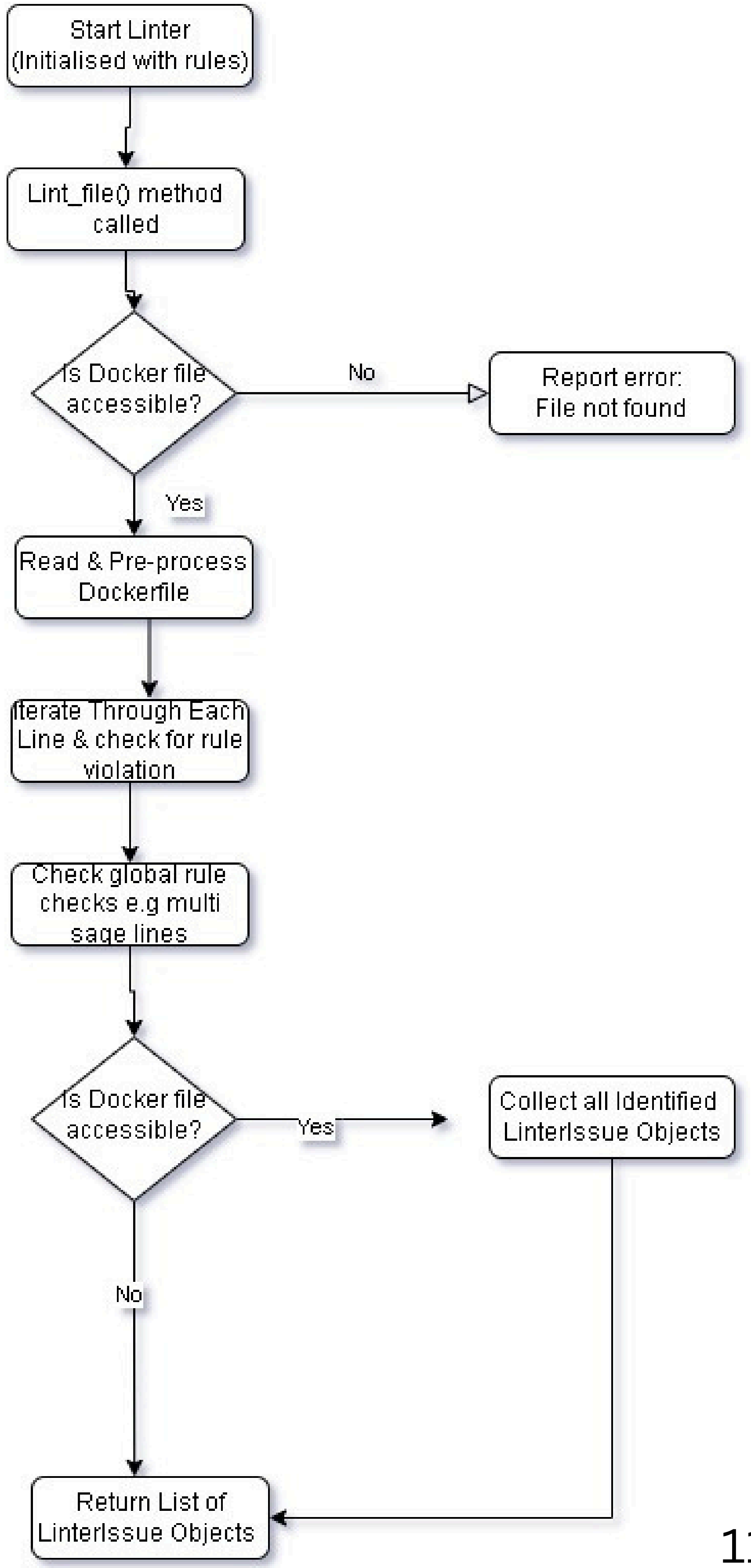

github.com/Peter-Sheehan/test/blob/main/Dockerfile
Peter-Sheehan / test
Code
Issues
Pull requests 10
Actions
Projects
Wiki
Security
Insights
Settings
main
test / Dockerfile
Peter-Sheehan Update Dockerfile
History
Code
Blame
27 lines (20 loc) · 839 Bytes
FROM ubuntu:18.04
LABEL maintainer="test@example.com"
LABEL description="A Dockerfile with several bad practices for testing, now with pip."
Inefficient updates and separate installs
RUN apt-get update
RUN apt-get install -y curl wget git python3-pip # Added python3-pip here
RUN apt-get install -y wget # Combined above
RUN apt-get install -y git # Combined above
RUN apt-get update
Redundant update, but keeping for original structure for now
Using ADD for a simple local file copy (imagine myapp_scripts.sh is a local script)
ADD myapp_scripts.sh /usr/local/bin/myapp_scripts.sh
RUN chmod +x /usr/local/bin/myapp_scripts.sh
WORKDIR /app
Copying entire build context
COPY . .
Install Python dependencies without cleaning up pip cache
RUN pip3 install -r requirements.txt
EXPOSE 8080
No CMD or ENTRYPOINT specified

Under the hood: Linter Engine

The Linter analyses Dockerfiles by:

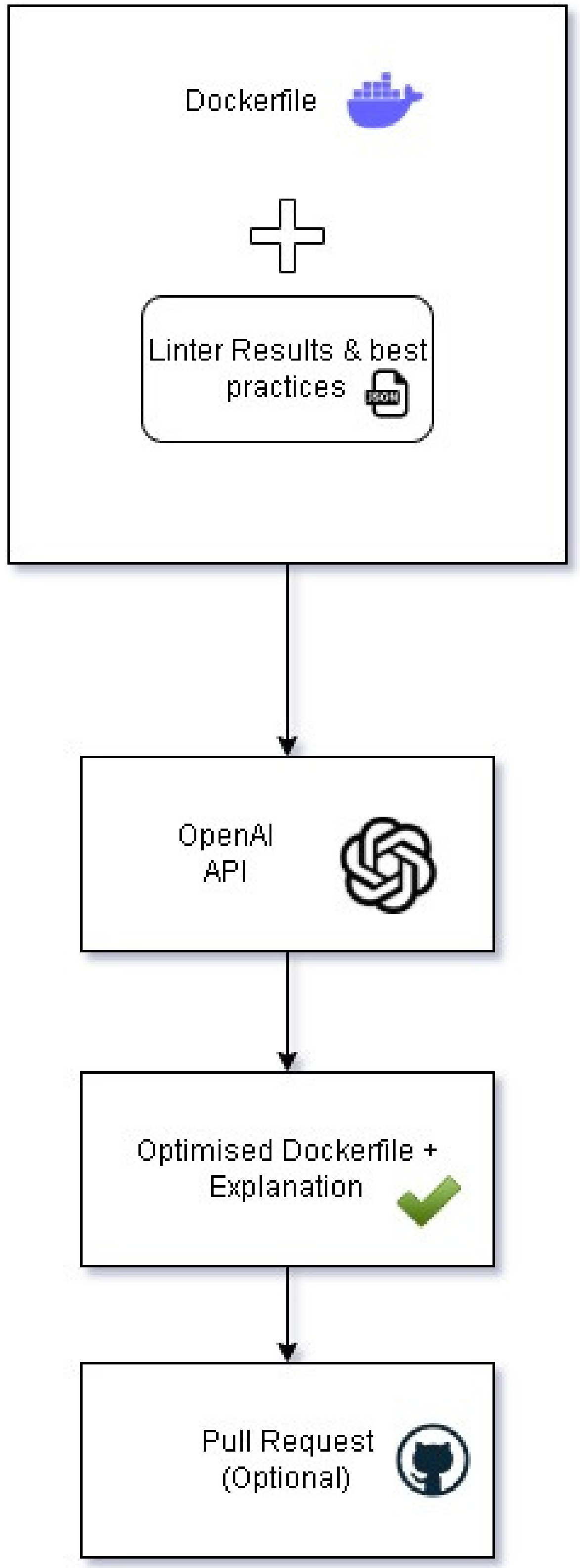
- Loading set of rules from a JSON file
- Reading a Dockerfile line by line
- Performing global checks since cannot be defined in regex pattern
- Reporting any violations to those rules

```python
for index, rule_data in enumerate(rules_data):
    category = rule_data.get("category", "Maintainability")  # Default if missing
    severity_str = CATEGORY_SEVERITY.get(category, "LOW")    # Assign severity based on category
    # Precompile the regex pattern
    compiled_pattern = re.compile(
        rule_data["regex_pattern"],
        flags=re.IGNORECASE | re.MULTILINE
    )

    rules.append(LinterRule(
        id=f"DOCKER_{index:03d}",
        title=rule_data["title"],
        description=rule_data["description"],
        severity=Severity[severity_str],
        regex_pattern=compiled_pattern,
        suggestion=rule_data["suggestion"]
    ))      You, 4 months ago • added core linter functionality
```



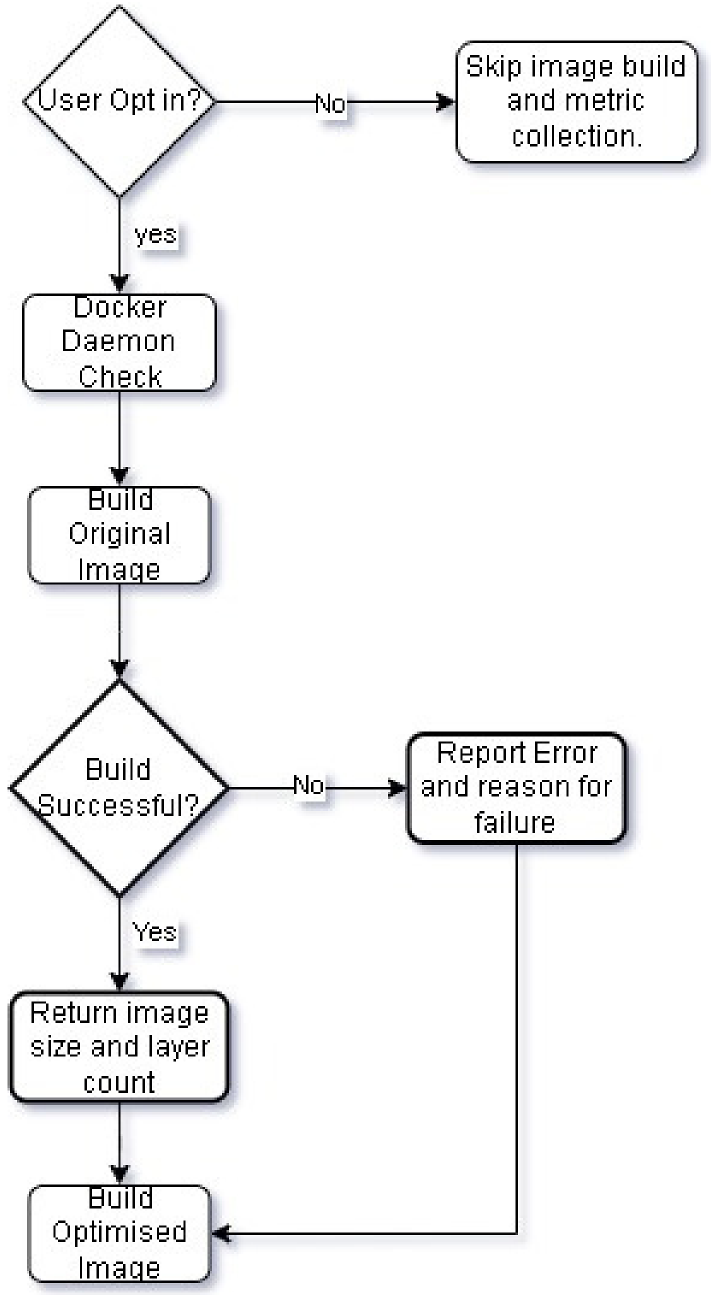
Under the hood: AI Optimisation

```
# Snippet from _create_optimization_prompt()
# prompt = f"""
# You are a Platform Engineer with deep knowledge of Dockerfiles...
# Assignment: Analyse and optimise the Dockerfile ...
#
# Important Context:
# 1. Linter Analysis:
# {linter_summary}  # Summary of issues found by our linter
#
# 2. General Best Practices:
# {self.best_practices_text} # Loaded from docker_best_practices.json
#
# **Your Task:**
# Based on the Dockerfile content, linter analysis, and best practices,
# provide an optimized version... and an explanation.
#
# **Original Dockerfile:**
# ```dockerfile
# {content} # The user's Dockerfile content
# ```
# """
```



Docker Build & Analysis Module

```python
def _get_image_info(self, dockerfile_path: str, tag: str = "temp_image_optimizer"): #returns size and layer count as a tuple.
    """Get Docker image size and layer count."""
    if not self._ensure_docker_client():
        self.console.print(f"[yellow]Skipping Docker build and analysis for '{tag}' due to Docker settings or connection status.[/yellow]")
        return 0, 0

    self.console.print(f"Building image from: {dockerfile_path} with tag {tag}")
    try:
        # Build the image with a simple progress bar (spinner + time elapsed)
        with Progress(
            SpinnerColumn(),
            TextColumn("[progress.description]{task.description}"),
            TimeElapsedColumn(),
            console=self.console,
            transient=True # Remove progress bar on completion
        ) as progress:
            build_task = progress.add_task(f"[cyan]Building Docker image ({tag})...", total=None)

            # Build the image - DO NOT iterate over build_log here
            image, build_log_stream = self.client.images.build(
                path=str(Path(dockerfile_path).parent),
                dockerfile=Path(dockerfile_path).name,
                tag=tag,
                rm=True,  # Remove intermediate containers
                forcerm=True, # Always remove intermediate containers
                timeout=300 # Add a 5-minute timeout
            )
            progress.update(build_task, completed=True, description="[green]Build finished.")

        self.console.print(f"[green]Image built successfully: {image.id}[/green]")

        # Get layer count
        layer_count = len(image.history())
        size = image.attrs['Size']

        # Clean up the temporary image
        try:
            self.client.images.remove(tag, force=True)
            self.console.print("[green]Temporary image removed.")
        except docker.errors.ImageNotFound:
            self.console.print("[yellow]Temporary image removal skipped (not found).")
        except Exception as e:
            self.console.print(f"[yellow]Warning: Could not remove temporary image: {e}[/yellow]")

        return size, layer_count
```



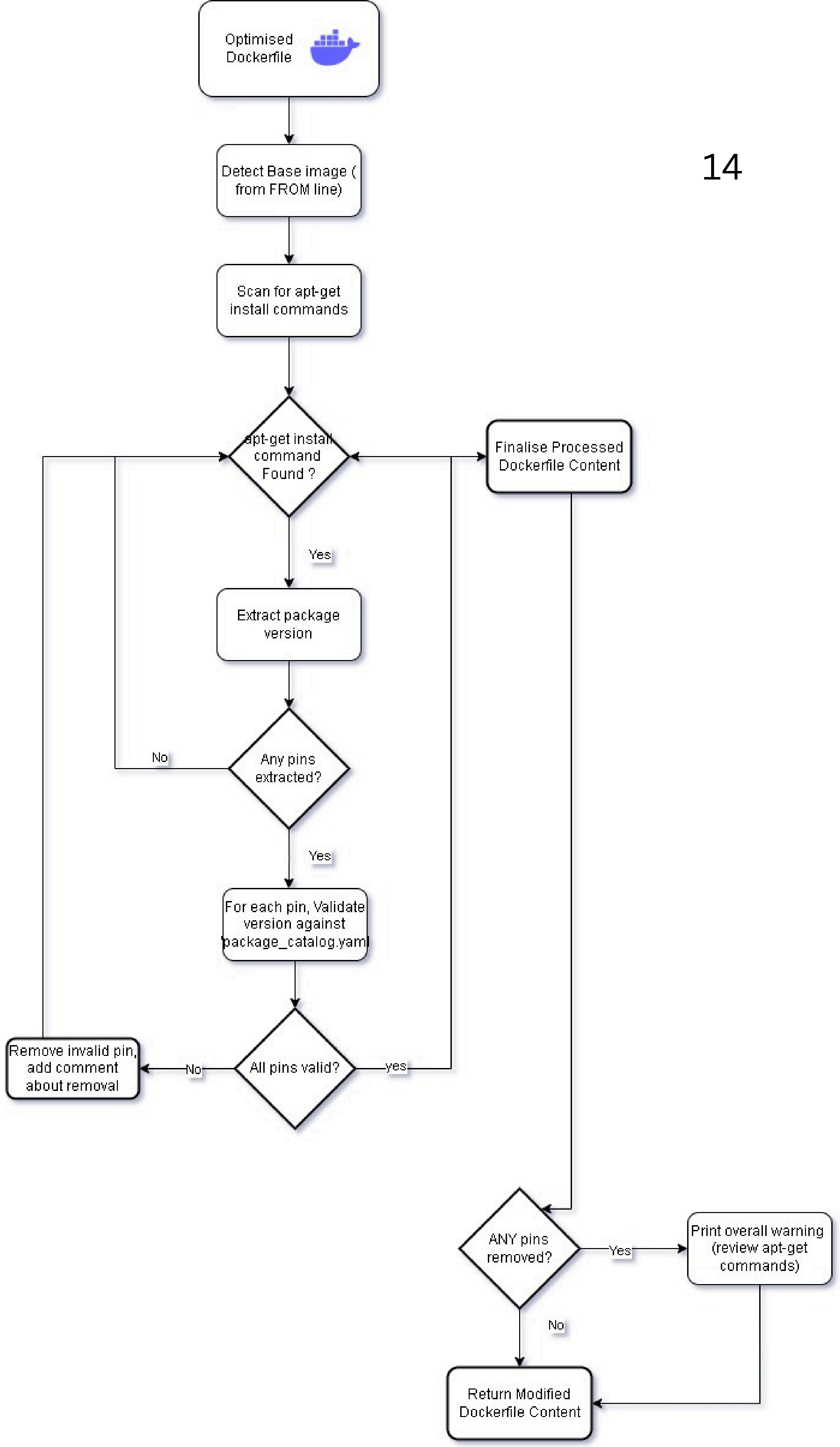
Post-Processing of AI Output (Distro-Aware Pinning)

Problem with LLMs

One of the main problems I had was AI suggesting incorrect versions of packages which might not be compatible with base images

FIX

- Detects Base OS
- Validates Pins
- Corrects Incompatibilities



CLI Interface

Interactive Mode

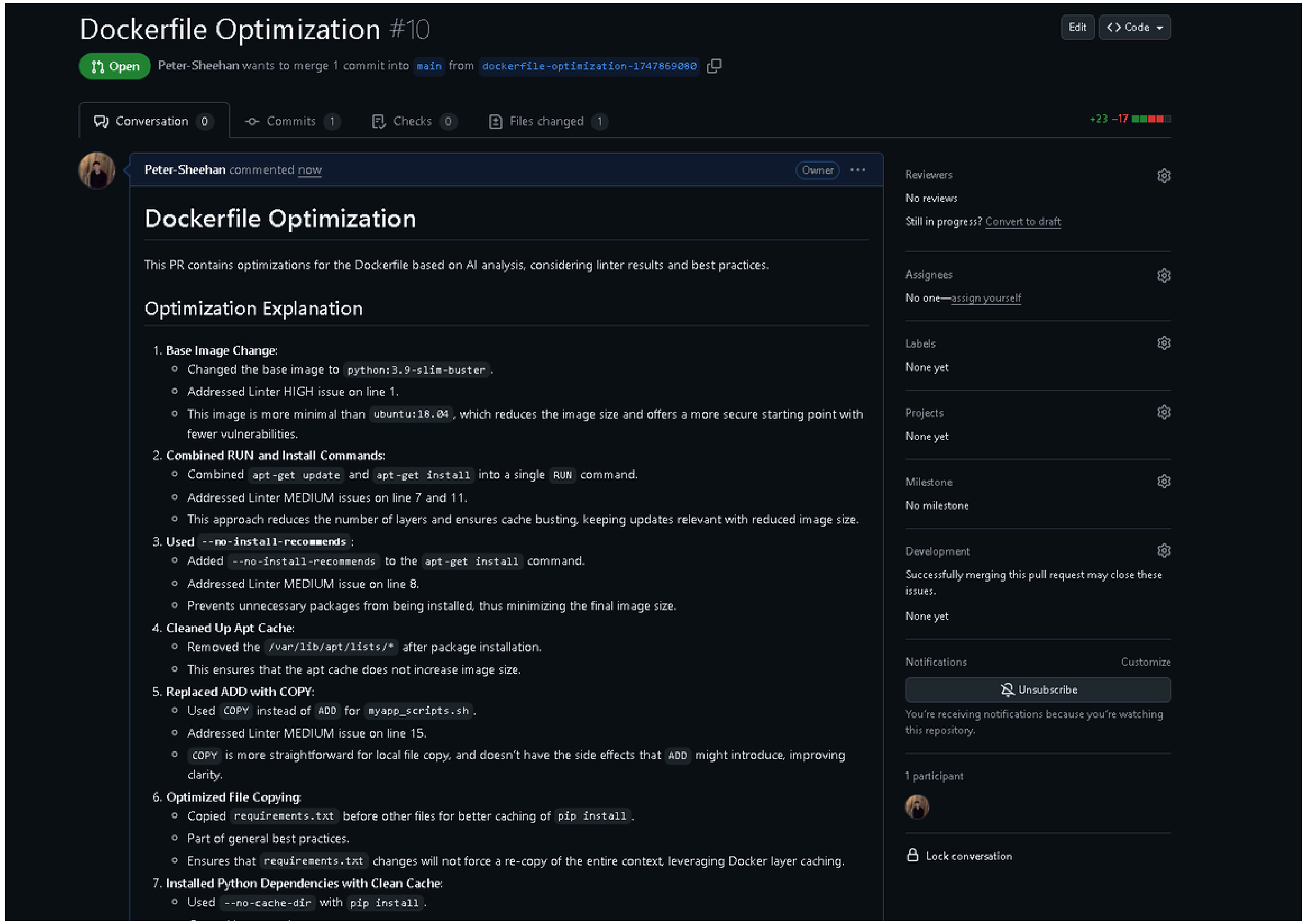
- Users are prompted to choose the analysis type (Local Dockerfile or GitHub Repository).
- They can opt-in to perform Docker image builds for size and layer count comparisons (if Docker is available and running).
- The CLI clearly communicates next steps and requirements (e.g., path to Dockerfile, Docker daemon status).

```
PS C:\Users\Peter\Desktop\Fourth Year\FYP\Code> dockerAI

Choose analysis type:
1. Local Dockerfile
2. GitHub Repository
3. Quit
Enter choice (1/2/3): 1

Attempt to build images for size comparison? (Requires Docker to be running) [y/n]: y
Docker image builds enabled. Will attempt to connect to Docker daemon when build analysis is performed.

Enter the path to your local Dockerfile: C:\Users\Peter\Desktop\Fourth Year\FYP\Code\dockerfiles\test_pinning_dockerfile
Analyzing local Dockerfile...
Attempting to connect to Docker daemon...
Warning: This operation requires Docker Desktop (or your Docker daemon) to be running. If it's not, this step will fail or be
skipped.
Successfully connected and pinged Docker daemon.
Building image from: C:\Users\Peter\Desktop\Fourth Year\FYP\Code\dockerfiles\test_pinning_dockerfile with tag temp_original_image
⠼ Building Docker image (temp_original_image)... 0:00:01
```

Non-interactive Mode

- Analyse <dockerfile_path> [options]:
  - Analyses a specified local Dockerfile.
  - **Options** include **--output-csv**, **--output-optimized-dockerfile**, **--output-explanation**, and **--perform-docker-builds**.
- optimize-github <repo_url>:
  - Analyses a Dockerfile from a GitHub repository and offers to create a Pull Request with optimizations.

```
@cli.command()
@click.argument('dockerfile_path', type=click
@click.option('--output', '-o', type=click.Ch
@click.option('--output-csv', type=click.Path
@click.option('--output-optimized-dockerfile'
@click.option('--output-explanation', type=cl
@click.option('--perform-docker-builds', is_f
```

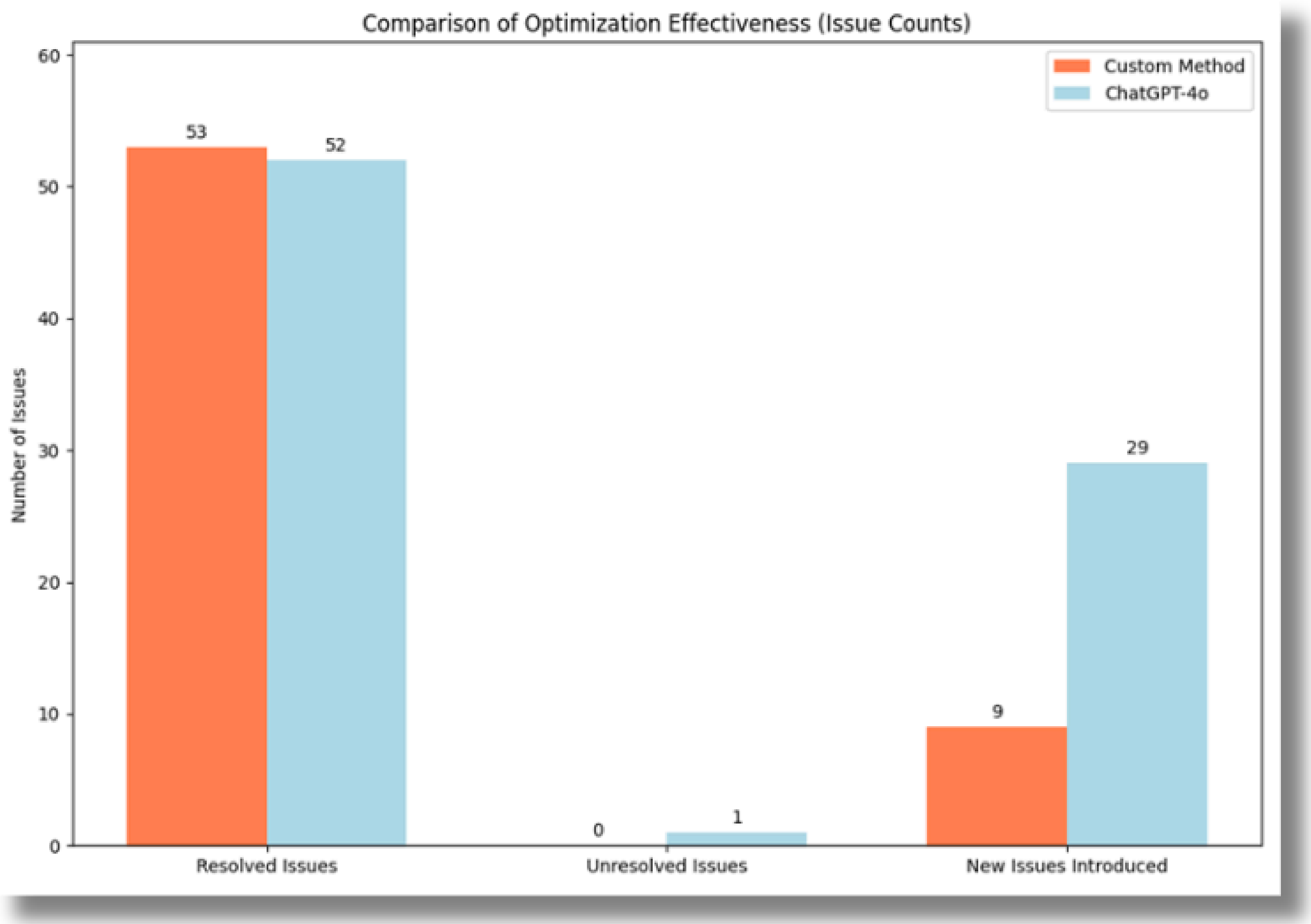


GitHub Integration

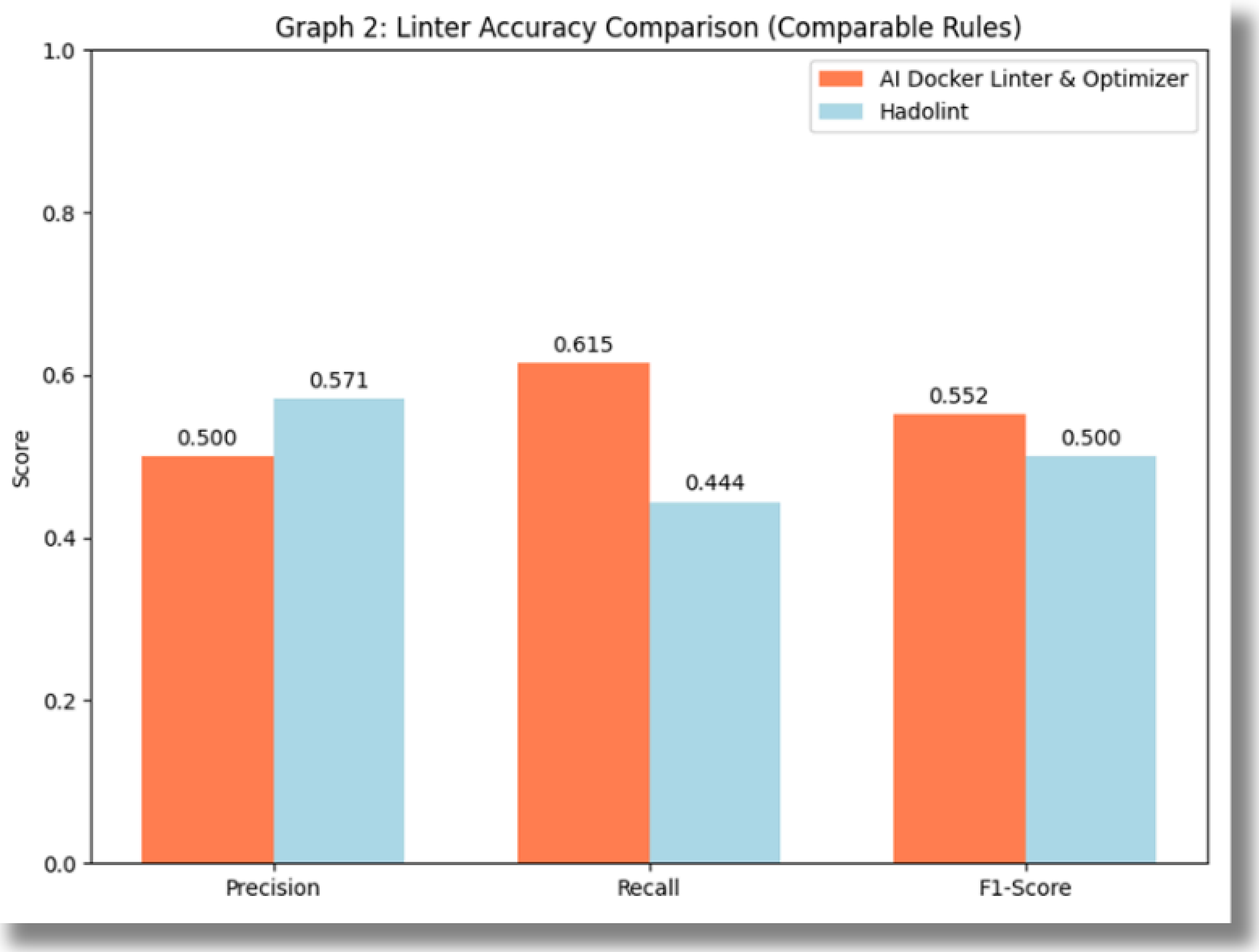
Flow:

- **Initiation & Authentication:**
- **Code Acquisition & Local Analysis**
- **PR Confirmation**
- **Branching & Committing**
- **Pull Request Generation**

Graph 1: Optimization Effectiveness)

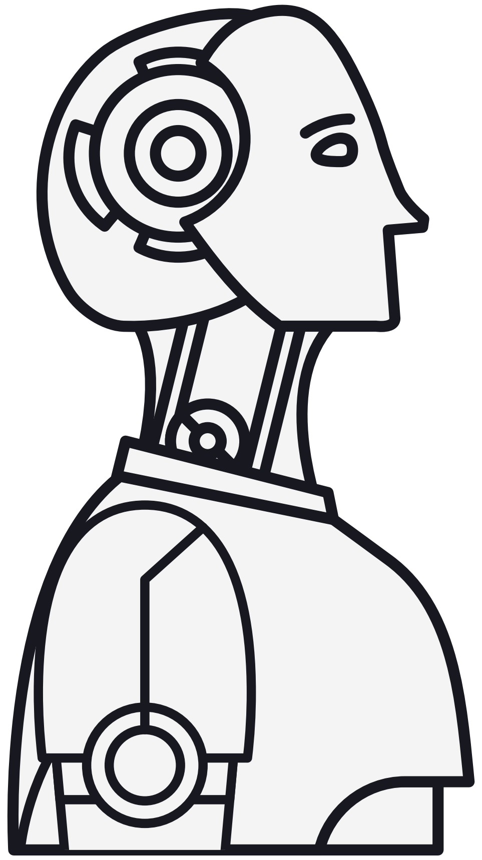


Graph 2: Linter Accuracy Comparison (Comparable Rules)



- **Precision = TP/ (TP+FP)**
- **Recall = TP / (TP + FN)**
- **F1-Score = Precision * Recall/ Precision + recall**

Future Enhancements



Expand Rule Set

Significantly increase the number and scope of manually created linting rules to cover a wider range of Dockerfile best practices and security vulnerabilities

Base Image/Package Compatibility API:

Implement a feature that checks for known compatibility issues between specified base images and pinned package versions. This could involve integrating with external vulnerability databases

Refine Custom Optimiser:

Investigate the 9 new issues introduced by the custom optimiser and enhance its logic to prevent these,

Project Conclusion

learning Outcomes	Achievements
Deepened Understanding of Docker	Effective Dockerfile Optimisation
Static & Dynamic Analysis Techniques:	Measurable Improvements
Benchmarking & Evaluation	Practical Utility

Any Questions