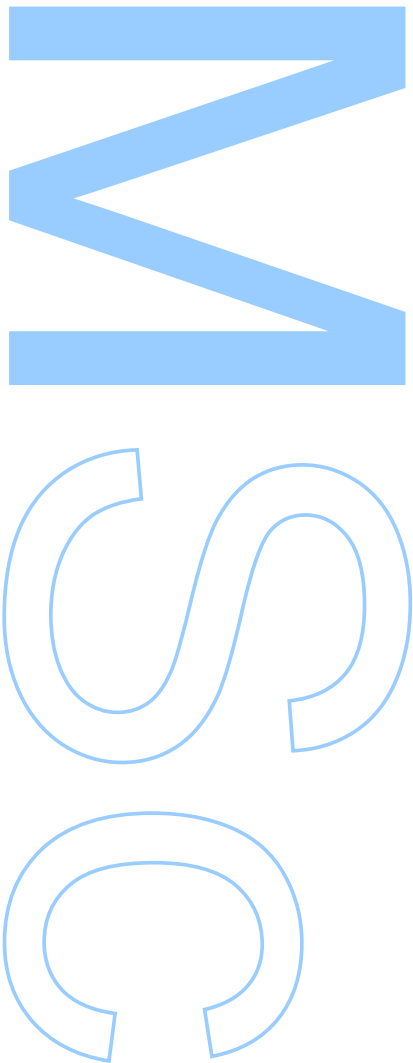
System For Pratical Evaluations of Network Administration Course

Diogo Nunes
Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2025

Orientador
Prof. Rui Prior, Faculdade de Ciências

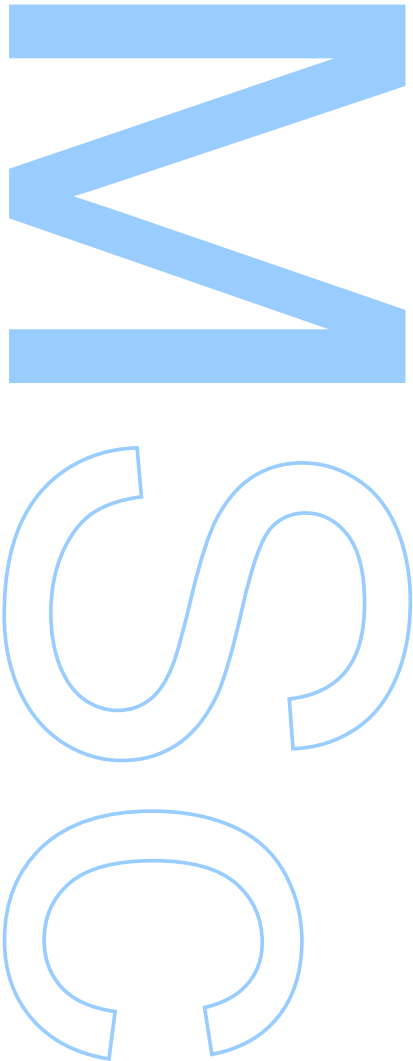


U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

Acknowledge ALL the people!

Resumo

Este tese é sobre alguma coisa

Palavras-chave: física (keywords em português)

Abstract

This thesis is about something, I guess.

Keywords: Computer Science

Table of Contents

List of Figures

List of Abbreviations

API Application Programming Interface. 6, 8, 17

ASGI Asynchronous Server Gateway Interface. 11, 13– 16

CS Computer Science. 1

DCC Department of Computer Science. 4

GNS3 Graphical Network Simulator-3. 6, 14, 16– 18

HTTP Hypertext Transfer Protocol. 6, 10, 14, 16– 18

IOS Internetworking Operating System. 7

IOU IOS on Unix. 7

JSON JavaScript Object Notation. 6, 15, 17

KVM Kernel-based Virtual Machine. iv, 9

LXC Linux Containers. iv, 9

OAS OpenAPI Specification. 15

Proxmox VE Proxmox Virtual Environment. 8, 9, 14, 16– 18

QEMU Quick Emulator. 7, 9

REST Representational State Transfer. 6, 8

VM Virtual Machine. 8, 9

VPCS Virtual PC Simulator. 7

WSGI Web Server Gateway Interface. 10, 11, 14

1. Introduction

In today's digital age the need for qualified Computer Science (CS) professionals is growing. The CS field is vast and has many areas of expertise, one of which is network administration. It is a crucial part of any organization, as it is responsible for the maintenance and management of the organization's network infrastructure. Proper training for network administrators is crucial for preparing them for real-world situations. One way to provide this training is through practical evaluations, allowing students to apply the knowledge they have acquired in a real-world scenario, helping them to develop the skills they will need in their future careers.

Creating a physical network environment for practical evaluations may be costly and challenging to scale for large student populations. Emulation and virtualization technologies can help to simplify and cost-effectively create practice environments for students. These technologies alone do not address the issue of manually reviewing a network topology's setup. Manually reviewing each student's network configuration can be time-consuming and prone to human error, rendering it challenging for their instructors. Automating the evaluation process may substantially alleviate the burden on educators and guarantee uniform and fair assessments.

1.1. Aims and Objectives

This dissertation continues the work of a previous student, who carried out research and first steps of development of a system for automated evaluation system for network topologies[1]. The main goal is to design and implement a scalable system capable of automatically evaluating evaluating network topologies that make use of different vendors and device types.The support for different vendors and device types is crucial, as it allows students to practice with a variety of networking equipment, preparing them for the real-world scenarios they will face in their future careers. Automating the evaluation process will help educators dedicate more time to other tasks such as supporting students, and would also provide a more consistent and fair evaluation, eliminating the possibility of human error.

Talk about the end goal

The main steps of this project are as follows:

taken from the proposal, probably change to be more in line with the gantt chart

- Study the bases for the system already developed

- Requirements gathering
- Identification of the main problems that need to be solved
- Proposal of solutions for these problems
- System design
- Implementation of a prototype
- Testing with volunteers to validate the system and identify possible limitations.

2. Background

The main focus of this chapter is to provide the reader with the necessary background to understand the context and technical foundations of this project. The goal of the system is to automatically evaluate network topologies by validating configurations and executing tests across various devices within a virtual network.

Achieving this requires the integration of multiple technologies, as the system must support a wide range of features to deliver a fully automated, end-to-end solution. Key concerns include not only functionality but also scalability, since multiple students may interact with the platform concurrently, each requiring an isolated working environment.

To support these requirements, this chapter introduces core concepts and tools such as virtualization, network automation, web frameworks and task processing. These components form the foundation upon which the system is built.

2.1. Virtualization

Virtualization is the process of creating a virtual version of physical resources, such as routers, switches, or even entire computers. In the context of this project, it is used to create virtual machines to provide students with a work environment and virtual networks, comprised of various types of virtualized devices. This approach enhances scalability and reduces costs, as it allows multiple virtual machines to be run on a single physical machine.

Virtualization can be categorized into **emulation** and **simulation**.

- Emulation is the process of creating a virtual version of a physical device in software, replicating its behavior exactly—including any bugs and limitations. This is useful for various things like testing software on different platforms, running legacy software on modern hardware and even running potentially harmful software in a safe isolated environment. Emulation will be used to provide students with a work environment to test their network configurations, as well as to emulate certain network devices.
- Simulation models the behaviour of a device, without replicating the underlying hardware or software. This results in a simpler less resource intensive model, though it

may not fully capture the real device's behavior. Simulation will be used to simulate the behaviour of certain, simpler and generic, network devices.

2.2. GNS3

Graphical Network Simulator-3 (GNS3) is an open-source graphical network emulator software that allows the user to create complex network topologies and interact with the various devices in it. It is widely used for educational purposes and is often used in preparation for professional network certifications like the Cisco Certified Network Associate (CCNA).

GNS3 employs a simple drag and drop interface to allow users to add new devices, make links between them and even add textual annotations. The software allows users to interact with the devices by way of a console or even a GUI if the device supports it. The software also allows users to export their topologies to be shared with others, which can be useful for teachers to provide students with a pre-configured topology to work on.

Additionally, the software supports packet capturing which is essential for students to develop their debugging and troubleshoting skills. Finally it can also be interacted with via a Representational State Transfer (REST) Application Programming Interface (API) which is of particular interest for this project.

2.2.1 Architecture

The software can be employed in a variety of ways due to its architecture [3] that separates the user interfaces that it offers, namely the locally installed gns3-gui as well as the browser accessible gns3-web, from the gns3-server that runs the emulations and the controller who orchestrates everything.

2.2.1.1 Controller

The controller is integrated in the gns3-server project and is responsible for communicating with all the other components of the software. The controller is a singleton, meaning there should only be one instance of it running at any given time, and it does not support concurrent requests. It is able to control multiple compute instances if so desired, each capable of hosting one or more emulator instances, varying depending on their complexity. The controller also exposes the REST API allowing the ability to interact with the software programatically. All communication is done over Hypertext Transfer Protocol

(HTTP) in JavaScript Object Notation (JSON) format and there is support for basic HTTP authentication as well as notifications via websockets.

2.2.1.2 Compute

The compute is also integrated in the gns3-server project and controls the various emulators required to run the nodes in the topology. The list of currently supported emulators is:

- **Dynamips** - Used to emulate Cisco routers and basic switching.
- **IOS on Unix (IOU)** - Used to emulate Cisco Internetworking Operating System (IOS) devices.
- **Quick Emulator (QEMU)** - Used to emulate a wide variety of devices.
- **Virtual PC Simulator (VPCS)** - A basic program meant to simulate a basic PC.
- **VMware/VirtualBox** - Used to run virtual machines with nested virtualization support.
- **Docker** - Used to run docker containers.

2.2.1.3 GUI

The GUI is composed of two separate but with mostly identical functionality, namely the gns3-gui and the gns3-web projects. The gns3-gui project is a desktop application that is used to to interact with a local or remote gns3-server instance. It is written in Python and uses the Qt framework for the graphical interface. The gns3-web is a web application that is accessed via a web browser it is still in a beta stage but is already capable enough to be used as a substitute for the gns3-gui.

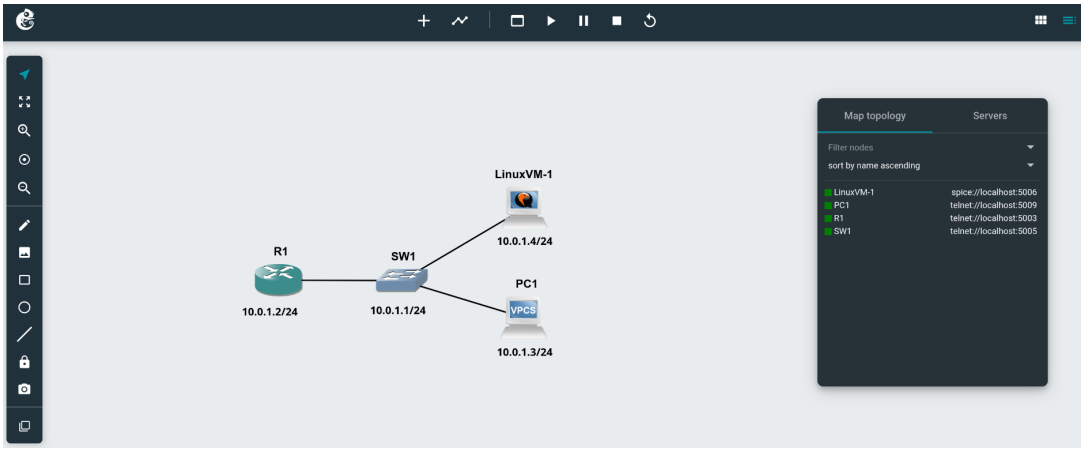


Figure 2.1: A simple network topology example in the GNS3 Web UI

2.3. Linux

Linux is a core component of this project, as it is the kernel of all operating systems used to host all the services provided.

- **Containerized Web Application** - The web application runs inside an LXC container on Proxmox. Since LXC containers share the host kernel, the containerized web application benefits from efficient resource usage while maintaining isolation from the host system.
- **Virtual Machines hosting GNS3** Proxmox also hosts virtual machines running Linux-based Ubuntu GNS3 servers. These VMs provide students with isolated environments to configure and test network topologies, benefiting from the flexibility of full virtualization through KVM, giving them the ability to virtualize any type of network device. Students only interact with these Virtual Machine (VM)s via the GNS3 web Interface. Students interact with these VMs using a browser, accessing the gns3-server instance running on them. Thus the students never have to interact with the underlying operating system, and so these machines do not require a desktop environment, which helps freeing up resources leading to better scalability.
- **Proxmox Virtual Environment (Proxmox VE)** - Proxmox is a Linux-based open-source platform for enterprise-level virtualization. It is based on the Debian Linux distribution.

2.4. Proxmox Virtual Environment

Proxmox VE is an open-source platform designed for enterprise-level virtualization [4]. It is based on the Debian distribution of Linux and provides a web-based interface for managing virtual machines and containers. It is widely used in data centers and cloud environments, as it provides a scalable and reliable solution for virtualization.

Proxmox VE bundles several core services that can be interacted with via shell commands, a web interface or even by using the Proxmox VE REST API. These allow the user to interact with every service provided by Proxmox VE, in a plethora of ways, depending on the user's needs, skills and preferences. The web interface is the most user-friendly way to interact with the platform, as it provides a graphical interface for managing the cluster. The shell commands provide a more direct way to interact with the platform, allowing for more complex operations to be performed and opening the doors to scripting and automation. Finally, the Proxmox VE REST API allows for programmatic interaction with the platform, enabling users to create custom applications that can interact with the platform.

2.4.1 Virtualization Technologies

Proxmox VE supports the the deployment and management of two distinct types of virtualization, namely, Kernel-based Virtual Machine (KVM)- based VMs and Linux Containers (LXC)-based containers.

Users can interact with these virtualized environments via NoVNC, a simple web-based VNC client or SPICE which is a more feature-rich protocol that provides better performance and more features than VNC. Both of these protocols support the use of a console-based interface, aswell as a full desktop graphical interface.

2.4.1.1 KVM

KVM is a virtualization solution provided by the Linux kernel. It leverages the hardware virtualization extensions of modern processors to provide a full virtualization experience at near-native speeds. Supports a wide range of guest operating systems making it a good choice for general purpose virtualization.

In Proxmox VE, KVM is used as the core component for running virtual machines and is used alongside QEMU.

2.4.1.2 LXC

Containerization is an operating system-level virtualization method that packages an application and its dependencies together into an isolated environment. Contrary to tradional VMs, containers dont emulate hardware or require a guest operating system relying instead on the host's kernel. This approach leads to a faster and more lightweight virtualization solution, as they consume less memory and cpu resources.

LXC creates full system containers, capable of simulating a complete Linux distribution providing users with an environment that behaves like a traditional VM but with the speed and efficiency of a container. LXC start much faster than VMs making them ideal for scenarios requiring rapid deployment and/or scaling.

However, it's important to note that while containers offer a degree of isolation, they do not provide the same level of security as VMs. This means that while they may not always be a suitable replacement for VMs.

2.5. Python

Python is a high-level, interpreted programming language renowned for its readability and versatility. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming, making it suitable for a wide array of applications. In the context of this project, Python serves as the primary programming language as its extensive standard library and supportive community contribute to efficient development and maintenance of the project's codebase.

2.5.1 WSGI

The Web Server Gateway Interface (WSGI) is a pivotal standard for Python web application deployment, defining a consistent interface between web servers and Python applications or frameworks.

Prior to WSGI's introduction, Python web frameworks were typically written against various server-specific APIs such as CGI, FastCGI, or mod_python. This diversity led to compatibility issues, limiting developers' choices of web servers and frameworks, as not all frameworks supported all web servers and vice-versa. To address this fragmentation, WSGI was created as a standardized interface, promoting portability and flexibility in deploying Python web applications.

WSGI serves as a bridge, enabling web servers to communicate with Python applications. It specifies a simple and universal interface for web servers to forward requests to Python applications and for those applications to return responses. This standardization allows developers to choose from a variety of web servers and Python frameworks without compatibility concerns[5].

Introduced in 2003 as PEP 333, WSGI was later updated to PEP 3333 in 2010 to accommodate Python 3. These specifications outline how web servers and Python applications should interact, ensuring a consistent and reliable deployment environment across different platforms.

The WSGI standard consists of two main components:

- **Server/Gateway Side** - Responsible for receiving HTTP requests from clients and passing them to the Python application. Then receives the response from the application and forwards it to the client.
- **Application** - The Python application that processes requests and returns responses.

Additionally WSGI has support for middleware components. WSGI middleware is a Python callable that wraps another WSGI application to observe or modify its behavior. Middleware can perform various functions, including request preprocessing, response postprocessing, session management, and security checks. This modularity allows developers to add functionality to their applications in a reusable and maintainable manner.

The separation defined by WSGI allows for flexibility and scalability in deploying Python web applications.

During development, Python WSGI applications often use built-in servers provided by frameworks like Flask. However, these servers typically aren't fully featured and aren't suitable for production environments. In production, WSGI servers act as intermediaries between web servers (e.g., NGINX or Apache) and Python applications, handling incoming requests and serving responses efficiently.

2.5.1.1 Flask

Flask is a web application micro framework written in Python, adhering to the WSGI standard, designed to facilitate the development of web applications by providing essential

tools and features. Classified as a microframework, Flask does not require particular tools or libraries, instead choosing to focus on simplicity and extensibility[6].

Initially, Flask served as the backbone for the entire project, providing the necessary infrastructure to handle HTTP requests, render templates, and manage application routing. Its flexibility and minimalistic approach allow for the integration of various extensions and libraries as needed, ensuring the application remains lightweight yet functional. Flask's comprehensive documentation and supportive community further enhance its suitability by the creation and support of community-driven extensions speeding up development and reducing the need to reinvent the wheel.

An example of how easy it is to develop a basic web application with flask is provided in the following small piece of code.

Algorithm 1 Flask Hello World

```
1: from flask import Flask
2: app = Flask(__name__)
3:
4: @app.route('/')
5: def hello_world():
6:     return 'Hello, World!'
7:
8: if __name__ == '__main__':
9:     app.run()
```

While Flask remained suitable for early development, emerging requirements—particularly those involving asynchronous processing and more scalable I/O operations—eventually led to an architectural shift, in part due to Flask's limited async support. This transition is discussed in detail in the following sections, where the adoption of Asynchronous Server Gateway Interface (ASGI)-compatible frameworks and asynchronous tools is explored.

2.5.2 Celery

Celery is an open source distributed task queue focused on real-time processing but also offers support for task scheduling. It is implemented in Python, but the underlying protocol can be implemented in any language. Celery requires a message broker to function, such as Redis or RabbitMQ, which are responsible for queuing and distributing tasks from producers (clients) to consumers (workers).

A Celery client running on a given machine will have allocated a set amount of workers to work on tasks. Workers are independent processes that listen to the broker and execute incoming tasks concurrently. Tasks are Python functions that are decorated with provided Celery decorators such as `@app.task`, causing them to be registered as Celery tasks within the Celery application.

Algorithm 2 Calling a Celery Task and Getting the Result

```
1: from celery import Celery
2:
3: app = Celery('tasks', broker='redis://localhost:6379/0', backend='redis://local-
   host:6379/0')
4:
5: @app.task
6: def hello():
7:     return 'hello world'
8:
9: result = hello.delay()
10: print(result.get())
```

To execute a task, a Celery task function must be called using the *delay()* method, which will return a result object. This result object can be used to check the status of the task and to retrieve the result once it is available.

Celery supports horizontal scaling by design, allowing multiple worker pools to run on separate physical or virtual machines. This makes it especially effective for handling growing workloads—for example, processing email newsletters for an expanding user base.

In addition to basic task execution, Celery provides advanced features such as retry policies, task chaining, prioritization, and timeouts. However, these benefits come with added complexity in deployment and maintenance, especially regarding broker reliability, result backend persistence, and worker supervision.

Furthermore, Celery clients and workers introduce a non-negligible overhead in terms of CPU and memory usage, even when idle, as they must maintain persistent connections to the broker and periodically perform health checks or heartbeats. This can be a concern in resource-constrained environments or during development.This overhead became especially evident during early integration tests.

As the project evolved, it became increasingly clear that Celery's benefits did not outweigh its resource and architectural costs for the current use case. This realization prompted an exploration of more lightweight asynchronous alternatives, eventually culminating in a

migration to FastAPI—an ASGI-compliant framework with native async capabilities and simpler concurrency management.

2.5.3 Asyncio

Asyncio is Python library for writing concurrent code. It provides a foundation for asynchronous programming by enabling the creation and management of event loops, coroutines, and asynchronous tasks.

An *event loop* is a central component of asynchronous programming—it continuously runs in the background, managing the execution of asynchronous tasks. When a task reaches a point where it would normally block (e.g., waiting for a network response), it yields control back to the event loop, which can then continue running other ready tasks. This model of cooperative multitasking contrasts with traditional multithreading or multiprocessing, as it operates in a single thread and does not require locking or context switching between OS threads.

A *coroutine* is a special kind of function defined with `async def`. When called, it does not run immediately, but instead returns a coroutine object. This object can be scheduled by the event loop, and when awaited, it runs until it hits a pause point (e.g., another `await`)—at which point it yields control back to the event loop, allowing other coroutines to execute.

This approach is particularly well-suited for I/O-bound operations—such as network communication, file access, or database queries—where tasks spend a significant amount of time waiting for external operations that are outside of our control to complete. Rather than blocking the entire application during such waits, Asyncio allows other tasks to execute in the meantime, leading to more efficient resource utilization and improved throughput.

In the context of this project, `asyncio` plays a critical role. The application often performs multiple concurrent HTTP requests to interact with services like GNS3 and Proxmox VE. By utilizing asynchronous functions and running them through the event loop, the application can efficiently manage these concurrent tasks without spawning additional threads or processes, thus keeping overhead low.

For example, in FastAPI, declaring an endpoint as `async def` ensures that the underlying logic is non-blocking. If that logic includes `asyncio`-compatible I/O operations—such as using a library for asynchronous HTTP calls then the request can proceed in a truly

asynchronous manner. This allows the web server to observe massive speedups when multiple HTTP calls must be made to external services.

Additionally, `asyncio` supports the orchestration of multiple tasks using constructs such as `asyncio.gather()`, which allows multiple coroutines to be executed concurrently and awaited collectively. This has been especially useful in scenarios within the project where multiple devices or services must be queried or configured simultaneously.

Overall, `asyncio` provides the concurrency model that underpins FastAPI's high performance. By embracing this model, the project benefits from improved responsiveness, lower latency, and better scalability—especially under workloads that involve heavy interaction with external services.

2.5.3.1 ASGI

ASGI is an interface specification for Python web servers and applications. It is considered a spiritual successor to WSGI, designed to provide a standard interface for asynchronous communication. ASGI was developed to address the limitations of WSGI, which was primarily designed for synchronous applications. Unlike WSGI, ASGI supports handling multiple requests concurrently, making it suitable for modern web applications that require real-time features such as WebSockets, long-lived connections, background tasks or the use of Python's async features.

As development progressed, asynchronous task handling became a more central requirement, initially addressed by integrating Celery. However, due to its resource overhead and deployment complexity, Celery was eventually phased out. This shift prompted an evaluation of frameworks that offered native support for asynchronous operations.

Quart—a reimplementation of Flask compatible with the ASGI standard—was initially considered due to its high degree of code compatibility with Flask. It promised an easy migration path while granting access to the benefits of asynchronous execution. However, several critical issues emerged concerning project stability, ecosystem maturity, and extension compatibility.

In particular, Flask extensions that were used in the project, proved incompatible with Quart, and their Quart-specific replacements were often incomplete, lacking in proper documentation, unmaintained or even abandoned. This lack of compatibility and maturity

posed significant development challenges, as many Flask extensions were essential for the project's functionality, and finding suitable replacements in Quart proved difficult.

As a result, adopting Quart would have required extensive reimplementation effort with limited long-term confidence in the ecosystem.

2.5.3.2 FastAPI

FastAPI is a modern, high-performance web framework adopting the ASGI standard. It leverages open standards, such as OpenAPI Specification (OAS), for defining path operations, parameters, and more, which in turn is based on the JSON schema. FastAPI relies entirely on Python type declarations, making it more intuitive and lowering the barrier to entry to new developers. This approach also simplifies the understanding and maintenance of the codebase.

Built on top of Starlette, a lightweight ASGI framework, and Pydantic, a data validation library. FastAPI combines the strengths of both to provide a powerful and flexible framework for building APIs with automatic data validation, serialization and documentation generation, all of which significantly enhance developer productivity.

Another key feature of FastAPI, being ASGI-compliant, is its built-in support for asynchronous programming, allowing developers to write non-blocking code using Python's *async* and *await* keywords. This is particularly useful for I/O-bound operations, such as database queries or network requests, as it allows the application to handle multiple requests concurrently without blocking the application which is essential in projects such as this one where multiple concurrent HTTP calls are made to interact with multiple devices and services concurrently, such as GNS3 and Proxmox VE.

Another powerful feature of FastAPI is its dependency injection system, that is very easy to use as it is automatically handled by the framework itselft. This allows for a clean and modular codebase, as dependencies can be easily injected into the various components of the application. This is especially useful in larger applications, where managing dependencies can become complex and cumbersome. This can be of particular importance for maintainabilty and extensibility as we have already seen in the previous sections with Flask extensions that were already integrated into the project, and that would have to be reimplemented from scratch in Quart, were the project to go down that path.

Ultimately, the project transitioned to FastAPI. While the migration to FastAPI involved a fair amount of effort, this was anticipated from the outset—unlike Quart, which had initially seemed easier but presented unforeseen difficulties, it resulted in better runtime performance, improved concurrency handling, and a cleaner overall structure, when compared to the previous Flask implementation with Celery integration.

This change laid the groundwork for more efficient handling of I/O-bound operations—such as network interactions with Proxmox or GNS3, which will be of importance in future iterations of the project while also streamlining endpoint development thanks to FastAPI's built-in request parsing, background task support, and integrated dependency injection system.

2.5.4 Nornir

Nornir is an open-source automation framework written in Python, designed to provide a flexible and efficient approach to network automation tasks[7]. Unlike other automation tools that utilize customized pseudo-languages, Nornir leverages pure Python code, offering developers the full power and versatility of the Python ecosystem.

Nornir supports multi-threaded task execution, allowing operations to run parallel across multiple devices. This capability enhances efficiency and reduces the time required enabling easy scaling to a large number of devices.

The framework provides a robust inventory management system, enabling the organization of devices into groups and the assignment of specific tasks to these groups. This structure facilitates targeted automation and simplifies complex network operations.

Finally, thanks to Nornir's architecture, it is highly extensible through its plugin system, allowing users to create custom plugins for inventory management, task execution, and result processing. This modularity ensures that Nornir can adapt to a wide range of network automation scenarios.

Nornir is particularly well-suited for tasks such as configuration management and state validation which makes it highly desirable in the context of this project. Its ability to handle concurrent operations will also ensure it can scale alongside the rest of the project.

2.5.5 Requests

The Requests[8] library is a popular and user-friendly HTTP library for Python, used to send HTTP requests to web services. It simplifies interactions with APIs by simple to use methods for the various HTTP verbs, as well as providing support for cookies, sessions, authentication, JSON and exception handling for network failures and invalid responses.

Requests was initially used in the project to handle all HTTP requests to the various services, such as GNS3 and Proxmox VE. Its simplicity and ease of use made it a natural choice for the initial implementation, allowing for quick development and testing of the various endpoints.

Algorithm 3 Making a Synchronous HTTP Request Using Requests

```
1:  import requests
2:
3:  url = ”https://api.example.com/data”
4:  response = requests.get(url)
5:
6:  if response.status_code == 200 then
7:      data = response.json()
8:      print(data)
9:  else
10:     print(”Request failed with status code”, response.status_code)
11: end if
```

However, as the project evolved and the need for asynchronous processing became more apparent, with Requests being a synchronous library only, there was a need to transition to an alternative that support asynchronous operations.

2.5.6 HTTPX

HTTPX[9] is a modern HTTP client library for Python. HTTPX retains a similar structure to Requests, while providing built-in support for asyncio.

In contrast to Requests, which blocks the current thread while waiting for a response, HTTPX enables non-blocking HTTP communication when used in asynchronous mode. This is particularly beneficial in scenarios involving multiple concurrent network operations, such as querying multiple GNS3 devices or cloning virtual machines in Proxmox VE, where synchronous requests would otherwise serialize execution and lead to performance bottlenecks.

Algorithm 4 Making an Asynchronous HTTP Request Using HTTPX

```
1: import httpx
2: import asyncio
3:
4: async def fetch():
5:     url = ”https://api.example.com/data”
6:     async with httpx.AsyncClient() as client:
7:         response = await client.get(url)
8:         if response.status_code == 200:
9:             data = response.json()
10:            print(data)
11:        else:
12:            print(”Request failed with status code”, response.status_code)
13:
14: asyncio.run(fetch())
```

HTTPX was adopted in the project to replace Requests for both asynchronous and synchronous use cases. Thanks to its full support for `async` and `await`, HTTPX integrates seamlessly into the FastAPI application, allowing concurrent HTTP requests to be awaited collectively using constructs like `asyncio.gather()`. This significantly improved the application's throughput under concurrent workloads.

Overall, HTTPX provides a robust and flexible foundation for asynchronous networking in Python, making it an ideal fit for the needs of this project.

Maybe talk about the choosen asgi server

3. Related Work

This chapter focuses on placing the current project within the context of existing solutions and related work. The primary goal of this project is to develop a system capable of automatically evaluating network topologies by validating device configurations and executing tests across a virtual network.

While automated assessment systems are well established in the field of programming education—receiving student-submitted code and running it against predefined test cases—equivalent systems for network exercises are far less common. Tools like Mooshak and similar platforms have proven effective for evaluating programming assignments and are widely adopted in academic settings.

At first glance, adapting these approaches to network topologies might seem straightforward. However, network evaluation introduces unique challenges such as the need for per-student virtual environments, real-time communication with multiple devices, and stateful, distributed configurations. This chapter explores existing tools like Mooshak and Packet Tracer, highlighting their capabilities, limitations, and how this project builds upon or diverges from them.

3.1. Programming Evaluation Systems

While not directly related, they are the main inspiration for this project. Programming evaluation systems are widely deployed in universities and other educational institutions. These systems receive as input code from students and subsequently run tests on it, outputting a score and even being configurable to provide students the first test case that they failed in, guiding students to the solution without handing it out.

These tools typically provide a structured approach to test coding and problem solving skills. They begin by offering a problem statement coupled with an optional image and an example test case, normally in the form of input and expected output. Users can interact with the system by use of an online code editor, where they can write their solution and submit it for evaluation, or by uploading a file with their solution. The system then evaluates the provided solution against multiple pre-defined test cases, and validating the output against the know-good output, outputting a score based on the number of

test cases passed. The system may also be configured to have time and/or memory constraints, to ensure that temporal and spatial complexity are also taken into account.

All of these, serve to provide a thorough evaluation of the student's solution, which can help guide a student to better their coding and problem solving skills.

In the context of the Department of Computer Science (DCC), Mooshak and Codex are commonly deployed to be used in the context of classes and even exams and programming contests.

The main differentiator between these systems and the one proposed in this project is the ability to solve a network exercise using multiple configurations across multiple devices, while programming evaluation systems will expect the same output every time, given the same input. Another key difference is the fact that programming evaluation systems dont always provide a working environment for the students to test their code, owing to the fact that students might prefer to user their own development environment for initial development and testing. This project aims to provide a working environment for students, as setting up a networking lab can be a daunting task for students, especially when they are just starting out. By providing a pre-configured environment, students can focus on learning the concepts and skills they need to succeed in their studies, rather than spending time troubleshooting their setup.

3.1.1 Mooshak

Mooshak is a web-based system for managing programming contests and also to act as an automatic judge of programming contests [2]. It supports a variety of programming languages like Java, C, etc. Under each contest students will find one more problem definitions each containing varying sets of test cases in input-output pairs. After submiting their solution, the system will compile and run the code against the test cases giving a score based on the the amount of test cases passed.

The system can also differentiate between differing types of errors, such as not giving the expected output, poorly formatted output, failure to compile or even exceeding the time limits. Mooshak also includes some features designed to drive competition between students, like a real time leaderboard and the ability to have more than 100% of the score for a given contest.

The system however is not without its limitations as it uses plain text files for its test cases and validates the output of student's code character by character, which can lead to false negatives if the output is not formatted exactly as expected.

3.2. Cisco Packet Tracer

Cisco Packet Tracer is a network **simulation** tool developed by Cisco Systems, widely used in academic environments to teach networking concepts and prepare students for certifications such as the Cisco Certified Network Associate (CCNA). It offers a visual interface for building and simulating virtual network topologies using a variety of Cisco devices, including routers, switches, and end devices.

While Packet Tracer is highly accessible and effective for introducing networking fundamentals, it is a closed-source, proprietary tool limited to simulating Cisco hardware and IOS features. Its functionality is optimized for teaching purposes rather than for flexibility, extensibility, or integration into larger automated workflows.

In contrast, this project aim to allow for a more realistic and extensible lab environment. The use of real operating systems and support of a wide range of vendor platforms for routers and switches, aswell as Linux-based virtual machines is highly desirable. This allows for a more realistic experience, as students will be able to work with the same tools and operating systems that they will encounter in real-world scenarios.

Therefore, while Cisco Packet Tracer remains a valuable educational tool, the needs of this project called for a more flexible and open architecture, which is better addressed by other tools.

4. System Architecture & Design

Delivering reliable, scalable automated assessment of virtualized networks requires a system built on solid principles of virtualization and automation. This chapter outlines the architecture of the proposed system, detailing the key components and how they interact to enable seamless evaluation of student-submitted network exercises.

The system is designed to provide each student with an isolated working environment where custom network topologies can be deployed, configured, and tested. To achieve this, the platform integrates several technologies—such as GNS3 for network emulation, Proxmox VE for virtualization, and Nornir for configuration testing—alongside an asynchronous web-based API layer for user interaction and system communications.

This section provides a high-level overview of the system, the rationale behind its design choices, and the fundamental components that make up its architecture.

4.1. System Architecture Overview

The system architecture is designed to provide a robust and scalable solution for the automated assessment of network topologies. The architecture is divided into several key components, each responsible for a specific aspect of the system's functionality. The main components of the system architecture are as follows:

- **Web App:** The web app serves as the main interface for users to interact with the system. It provides endpoints for evaluation, creation and viewing available exercises. The app is designed to be asynchronous where possible, allowing for efficient handling of multiple requests simultaneously.
- **Proxmox VE:** Proxmox VE is responsible for creating and managing VMs that host the network devices used in the exercises. This layer interacts with the web app to create and manage the VMs based on the creation of new exercises or students. All communication with Proxmox VE is done asynchronously through the Proxmox REST API, which allows for efficient communication, keeping the web app responsive, while also keeping the components decoupled.

- **GNS3:** GNS3 is used to emulate all the components of the virtual networks to be configured by students, using various types of virtualization detailed earlier. Communication with GNS3 is done through the GNS3 REST API by the web app, due to a lack of robustness of the GNS3 API.

synchronously (will it stay this way)

- **Nornir:** This automation framework is used for validating device configurations. It connects to the virtualized devices, executes commands, and compares the output to expected results to determine correctness.

4.2. Component Breakdown

4.2.1 Web Application

The web application serves as the primary interface through which users interact with the system. It is built using the FastAPI framework and follows an asynchronous-first, modular architecture that scalable interactions with other system components.

The application exposes a REST API that supports endpoints for user authentication, exercise creation, virtual machine management, and configuration validation. It acts as the coordinator for the entire system, triggering operations in Proxmox VE, Proxmox VE, and Nornir based on user actions.

Wherever possible, asynchronous I/O is employed to prevent blocking during operations such as API calls to Proxmox. Multiprocessing is also utilized to handle configuration validation. This keeps the system responsive and performant, especially when handling multiple simultaneous requests from different users.

Internally, the application is designed to be stateless and maintain minimal runtime state. All essential information—such as user accounts, defined exercises, and student-to-VM mappings—is persisted in a relational database rather than stored in memory. This design improves reliability, supports concurrent usage, and enables horizontal scalability if deployed across multiple instances.

To ensure maintainability and modularity, interactions with external services like Proxmox VE and GNS3 are isolated in dedicated modules. These serve as abstraction layers between the application logic and third-party APIs, exposing clean, reusable interfaces while hiding low-level implementation details. For example, Proxmox-related operations such as VM creation and deletion are handled in a separate module (e.g services/proxmox.py), as are all GNS3-related tasks. This separation of concerns improves the structure of the codebase and simplifies future maintainability by being more readable.

4.2.2 Proxmox Virtual Environment (PVE)

4.2.3 GNS3 Network Emulator

4.2.4 Nornir Automation Framework

4.2.5 Celery and Asynchronous Task Management

4.2.6 Storage and Data Model (Optional)

References

[1] P. M. C. Santos, “Environment for practical evaluations in network administration,” Master’s thesis, Faculdade de Ciências da Universidade do Porto, Porto, Portugal, 2024, accessed: March 21, 2025. [Online]. Available: https://hdl.handle.net/10216/164899

[2] J. Leal and F. Silva, “Mooshak: a web-based multi-site programming contest system,” *SOFTWARE-PRACTICE & EXPERIENCE*, vol. 33, no. 6, pp. 567–581, MAY 2003.

[3] GNS3 Documentation, “Architecture,” n.d., accessed: March 15, 2025. [Online]. Available: https://docs.gns3.com/docs/using-gns3/design/architecture/

[4] Proxmox Server Solutions GmbH, *Proxmox VE Administration Guide*, 2025, accessed: March 21, 2025. [Online]. Available: https://pve.proxmox.com/pve-docs/pve-admin-guide.html

[5] P. J. Eby, *PEP 333 – Python Web Server Gateway Interface v1.0*, Online, 2003, accessed: March 21, 2025. [Online]. Available: https://peps.python.org/pep-0333/

[6] P. Projects, “Flask documentation,” Online, 2025, accessed: March 21, 2025. [Online]. Available: https://flask.palletsprojects.com/en/stable/

[7] D. Barroso, “Nornir: The python automation framework,” Online, 2025, accessed: March 21, 2025. [Online]. Available: https://nornir.readthedocs.io/en/latest/

[8] K. Reitz and R. Contributors, “Requests: Http for humans,” Online, 2025, accessed: March 18, 2025. [Online]. Available: https://requests.readthedocs.io/en/latest/

[9] T. Christie and E. Contributors, “Httpx: A next generation http client for python,” Online, 2025, accessed: April 04, 2025. [Online]. Available: https://www.python-httpx.org/

Appendix Title Here

Write your Appendix content here.