# System For Pratical Evaluations in Network Administration Courses

Diogo Nunes (up202007895) - M:ERSI - FCUP

Rui Prior - Supervisor

# The Problem

- Physical networking environments demand dedicated hardware and lab space
  - Making it hard to scale for bigger student populations
- Some institutions skip pratical evaluation due to it being very time consuming
- Existing tools (e.g., Cisco Packet Tracer) are limited:
  - Closed source
  - No support for heterogeneous multi-vendor topologies
  - Is a simulator

# The Idea

- Build a backend system that can:
  - Deploy virtual lab environments for students
  - Automatically validate their configurations made by students
  - Support instructor-defined exercises and grading criteria
- Make it fully on-demand and scalable
  - Students trigger evaluations themselves
- Inspired by systems like Mooshak, but for networking
  - Automated, on-demand assessment works well in programming education
  - Our goal: bring similar automation to networking environments
- Integrate with existing infrastructure
  - Proxmox VE for virtualization, GNS3 for emulation, LDAP for auth

# Challenges Presented

- Study Proxmox VE and GNS3 APIs
  - Understand their models, limitations, and how to control them remotely
  - Each student gets a separate work environment per exercise
- Implement authentication & authorization
  - Investigate integration with LDAP
- Achieve reliable communication between components
  - Build a backend that orchestrates VMs, GNS3, validation capabilities

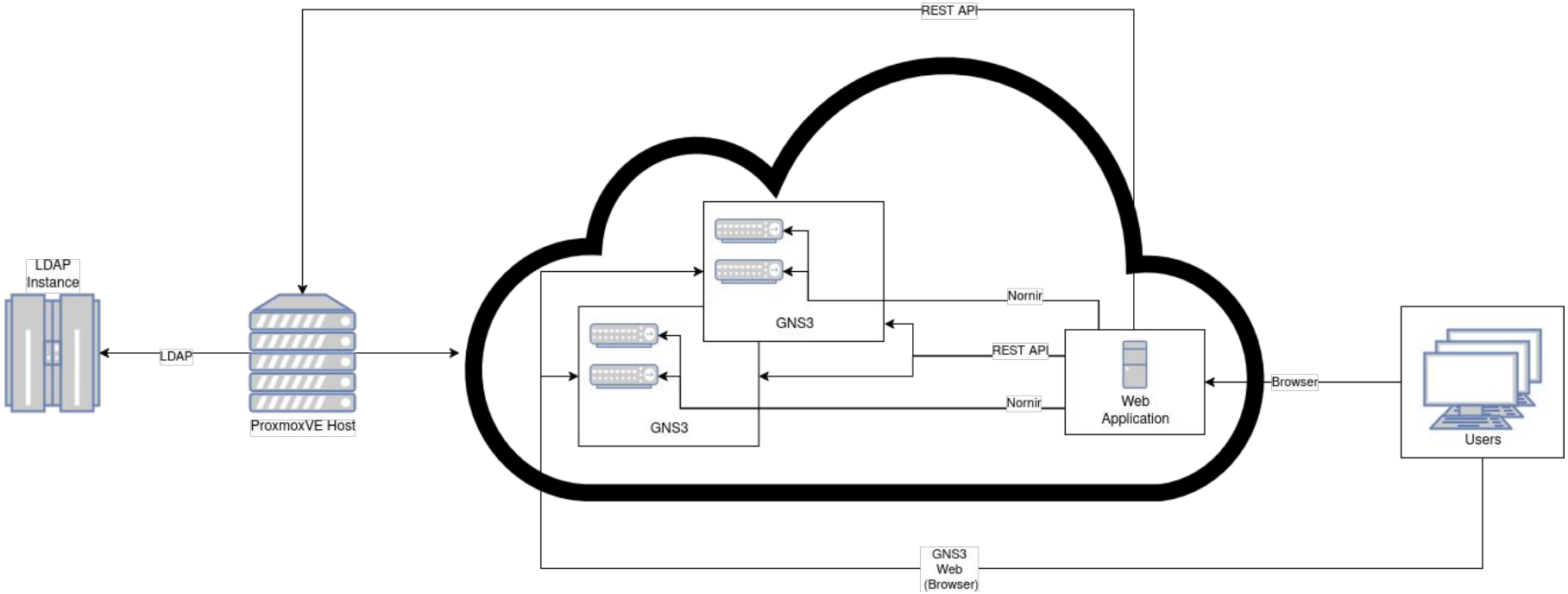
# The Starting Point

- Inherited basic building blocks:
  - Proxmox deployment
  - VMs with GUI running GNS3
  - Bash scripts for cloning and managing VMs on Proxmox
  - A Nornir ping module for validation
- All code had to be run manually
- Proxmox scripts had to be run directly on the host machine
- Tools were not connected — no system, no scalability

# Objectives/Contributions

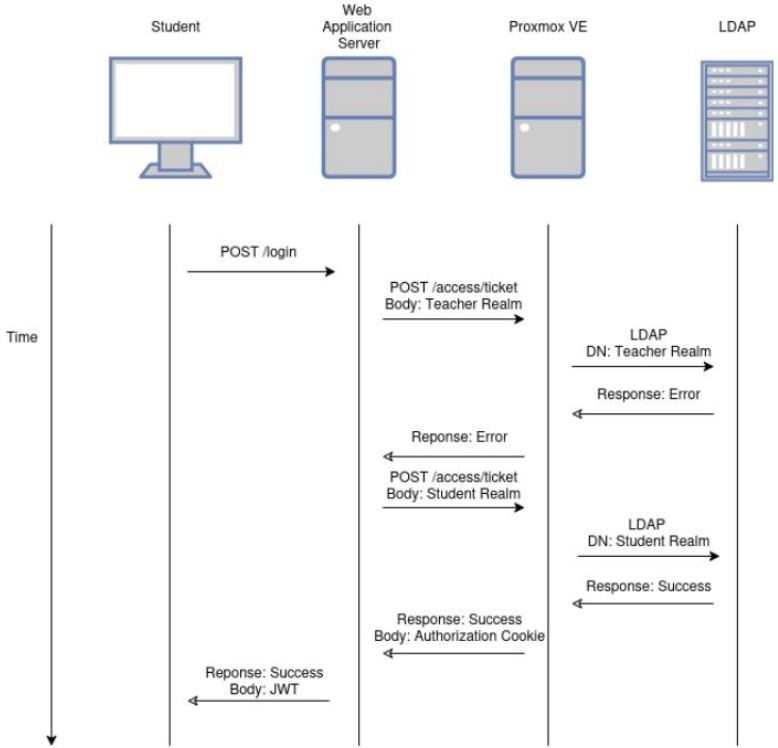
- Unify and extend disconnected tools into a cohesive backend
  - Replace manual scripts with Python code capable of remote communication
- Automate lab provisioning and validation
  - VM + GNS3 setup and run grading tasks on demand
- Enable real-time, usage for students
- Integrate institutional authentication
- Deliver a reusable, extensible grading system
  - Capable of adding new commands and device types

# Components


REST API
LDAP Instance
LDAP
ProxmoxVE Host
GNS3
Nornir
REST API
Nornir
GNS3
Web Application
Browser
Users
GNS3 Web (Browser)

# Authentication/authorization

- JWT-based
- LDAP-backed Proxmox login flow
  - Web app attempts login via Proxmox API, first to the teacher realm
  - If it fails, retries in the student realm
- Role determined dynamically by realm
  - Login to teacher realm → elevated access
  - Login to student realm → restricted access
- Web app uses role to control feature access
  - Teachers can create, manage, and view all exercises
  - Students can only interact with their assigned environment
- All authorization handled on the web app side


Student
Web Application Server
Proxmox VE
LDAP
Time
POST /login
POST /access/ticket
Body: Teacher Realm
LDAP
DN: Teacher Realm
Response: Error
Reponse: Error
POST /access/ticket
Body: Student Realm
LDAP
DN: Student Realm
Response: Success
Response: Success
Body: Authorization Cookie
Reponse: Success
Body: JWT

# Communication With Proxmox

- Fully remote interaction via Proxmox's REST API
  - No shell scripts or direct access to the Proxmox host required
- Authentication ticket cookie issued on user login
  - Retrieved via Proxmox API and cached in memory by the web app
- User-specific ticket cookie handling
- Cookie reused for all API calls
  - Allows fast and persistent access during a session
  - Ensures every action is traceable to the authenticated user
- Retry mechanism with exponential backoff
  - Automatically retries failed Proxmox API calls
  - Handles intermittent failures, rate limits, or network hiccups
- Improved reliability and responsiveness under load
  - Ensures critical tasks like VM deletion or disk cleanup eventually succeed
- Centralized session + error handling
  - All Proxmox interaction wrapped in fault-tolerant logic for consistency

# Initial Design Decisions

- Chose Flask for web app
  - Lightweight, familiar, fast to prototype
  - Strong documentation and ecosystem
- Wrapped Proxmox API endpoints in Python
  - Fully replaced shell scripts with HTTP-based automation
- Derived a modular evaluation system using Nornir
  - Based on the original implementation (Santos, 2024)
- Very minimal frontend

# First roadblocks

- Stress tests exposed scalability limits
  - Retry mechanism not built at this point in time
  - System struggled as # of HTTP calls were made to Proxmox
- Flask’s synchronous model (WSGI) became a bottleneck
  - Could only process one request per thread
  - Which is suboptimal for heavy I/O workloads
- Blocking I/O couldn't handle concurrent API calls
  - A single slow Proxmox/GNS3 call could cause major stalls
- Conclusion: Needed a concurrency-friendly architecture
  - One that could handle dozens of overlapping I/O requests efficiently

# Exploration of Celery

- Celery added parallelism via background workers
  - Offloaded slow tasks from Flask’s request thread
- But brought significant architectural overhead
  - Required Redis, worker pool, persistent connections, heartbeats
  - High idle resource usage even during low traffic
- Mismatch for short-lived, I/O-heavy tasks
  - Broker communication overhead > task duration
  - Delayed simple operations like VM deletion
- Led to exploring alternatives to Flask

# Exploration of ASGI

- WSGI --> ASGI transition was essential for concurrency
  - Needed async to support many I/O-bound operations in parallel
  - Flask’s WSGI model couldn't be adapted to fully support Python's async capabilites
- Tried Quart: an ASGI drop-in Flask replacement
  - Promised compatibility + async/await syntax
  - Migration was not easy, and Quart's ecosystem isn't mature
  - Critical Flask extensions lacked stable Quart ports, forcing reimplementation
- Chose FastAPI for long-term stability and async-native design
  - Confidence in the ecosystem and async-first architecture
  - Migration meant rewriting some components — but enabled cleaner async logic

# Migration to FastAPI

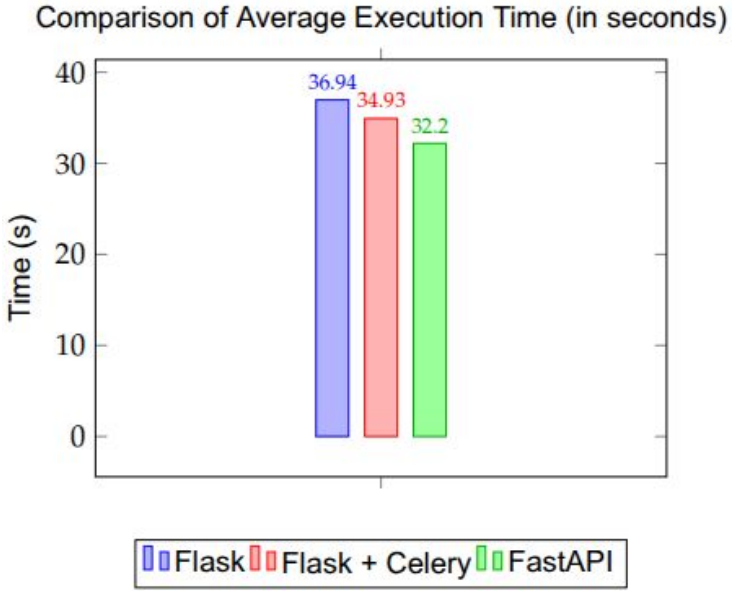
- Replaced Flask with FastAPI
  - Fully async, ASGI-native framework
  - Clean and explicit async-first architecture
- Removed Celery + Redis
  - Replaced background tasks with async routines
  - Reduced system complexity and idle resource usage
- Simplified internals
  - No separation between Flask code and Celery code
  - Fewer moving parts
  - Easier debugging and better observability

# Evaluation Strategy

- Designed 3 representative tasks to compare:
  a. Template VM Creation
     - Sequential flow: clone → boot → wait for IP → import GNS3 → shutdown
  b. Batch VM Cloning
     - Parallel clones for multiple users
     - Emphasizes concurrency and I/O stress
  c. Batch VM Deletion
     - Simple, short-lived deletions — low complexity, high frequency
- Tested 3 implementations side-by-side:
  a. Flask (synchronous)
  b. Flask + Celery
  c. FastAPI (async-native)
- Metrics collected:
  a. Average execution time
  b. Variance/stability across runs
  c. Scaling behavior with batch size

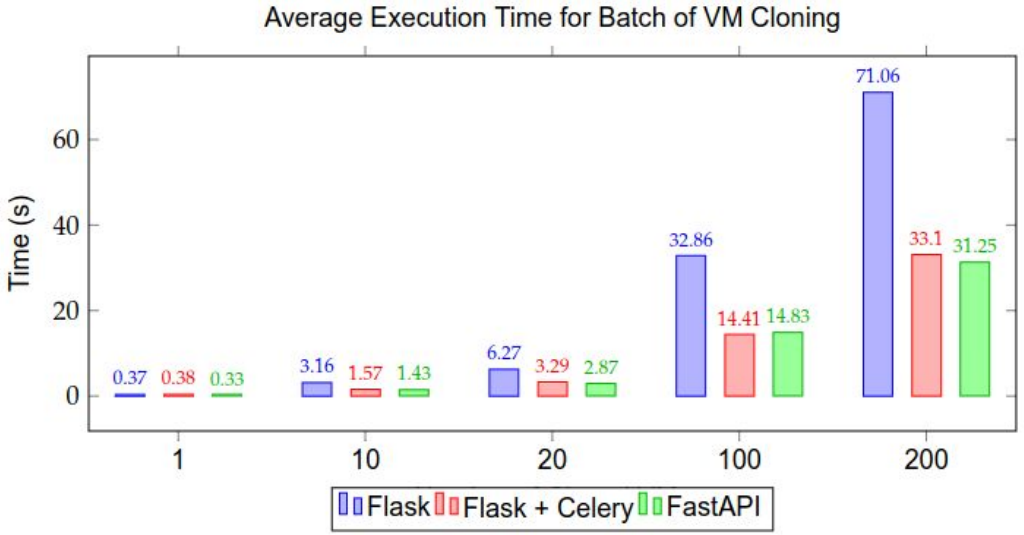
# Task 1 - Template VM Creation

- Sequential workflow
- No concurrency in the task itself: each step depends on the previous one



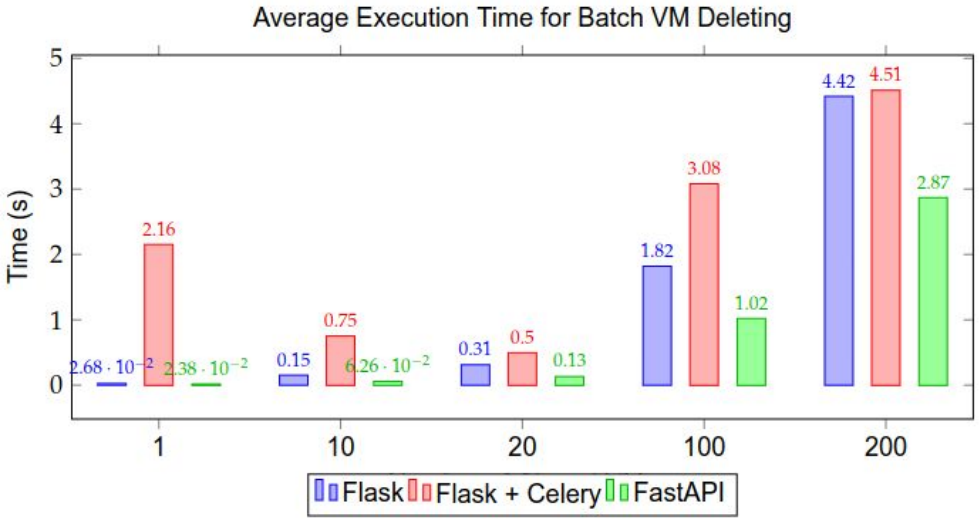
# Task 2 – Batch VM Cloning

- Parallel task: each VM clone is independent
- Scales linearly with batch size (100–200 VMs tested)



# Task 3 – Batch VM Deletion

- Lightweight task: one API call per VM to delete
- Fully parallelizable with minimal logic



# Key Findings

- Flask was the most consistent — but also the slowest
  - Predictable linear scaling with batch size
  - Long blocking times are undesirable
- Flask + Celery improved concurrency, but
  - Introduced overhead (brokers, idle workers)
  - “Moody” and erratic performance
  - Performed worse on short-lived tasks (e.g., deletion)
- FastAPI outperformed all other implementations
  - Slightly faster in sequential tasks (Template VM creation)
  - 2x faster on average in parallel workloads (Batch cloning & deletion)

FastAPI was the best balance of speed, stability, and simplicity

# Real-World Challenges – Proxmox Under Stress

- Deleting large batches of VMs revealed hidden infrastructure issues
- Proxmox failed silently during batch deletion tasks
  - Orphaned VM disks were left behind
  - Errors appeared only in server logs, not API responses
- Performance degraded over time as storage became congested
  - Locking mechanism couldn’t keep up with concurrent deletions
- Implemented:
  - Request throttling in the backend to avoid overloading Proxmox

# Final System Snapshot

- Web backend written in Python using FastAPI
- Minimal web interface built, only meant for functionality testing
- Orchestration capabilities with Proxmox VE for VM lifecycle automation
- GNS3 for real-world network topology emulation
- Nornir-based evaluation validates device configurations via command output
  - Designed for extensibility
- Students trigger assessments via web interface
- Teachers can define, and assign network exercises
- Internal concurrency, retry logic, and authorization ticket caching implemented

# Showcase

...

# Lessons Learned

- Concurrency needs to be built into architecture, not retrofitted
  - Async I/O is essential for handling external APIs at scale
  - Flask’s WSGI model was fundamentally limiting
- Modular design enables scale and maintainability
  - Separating concerns made testing and extending easier
- APIs don’t always reflect real-world state
  - APIs may report success while backend operations fail silently (e.g., Proxmox disk deletions)
  - Infrastructure-level monitoring and error handling are critical
- Testing isn't just for correctness — it’s also for detecting performance issues
  - Periodic load testing uncovered silent failures

# Future Work

- Access control improvements
  - Integrate Proxmox VE's native role-based permissions
  - Move beyond backend-only role checking
- Assignment & grading enhancements
  - Support versioning and complex grading logic
  - Add sub-lines, conditionals (e.g. boolean expressions)
- Frontend upgrades
  - Migrate to a client-side rendering framework
  - Enable real-time progress updates and feedback
- Multi-node support & load balancing
  - Proxmox clustering for horizontal scalability
  - Dynamic VM placement based on resource usage
- Containerized environments
  - Use LXC for lighter assignments
  - Especially for labs where no KVM is needed
- Test coverage & CI
  - Add unit/system tests to support long-term maintainability

# Closing Thoughts

- Started from low-level scripts and isolated components
- Designed and implemented a backend system for automated lab hosting and evaluation
- Prototyped with Flask; identified scalability issues
  - Researched alternatives and migrated to FastAPI for native async support
  - Achieved measurable performance and scalability improvements
- Built a foundation for  further development