UCLouvain

epl

École polytechnique de Louvain

PANTHER User Interface Development

Author: **Muhammad Umer Malik**
Supervisor: **Charles Pecheur**
Readers: **Christophe Crochet, Kim Mens**
Academic year 2025–2026
Master [60] in Computer Science

Abstract

Modern protocol testing frameworks are often powerful but difficult to use, especially when experiment definition and execution rely on low-level configuration files and command-line workflows. This thesis addresses that problem in the context of PANTHER, a research-oriented framework for reproducible protocol experimentation. The goal is to make experiment setup, inspection, and monitoring more accessible without sacrificing the flexibility and rigor of the underlying system.

To this end, the thesis presents the design and implementation of a web-based graphical user interface for PANTHER. The interface supports the creation and editing of experiment configurations, visual inspection of protocol topologies, execution and monitoring of experiments, and exploration of results and plugins. It preserves YAML as the canonical configuration format while adding structured forms, validation, and synchronized visual feedback to reduce errors and improve usability.

The proposed solution is implemented as a browser-based application integrated with the existing PANTHER architecture. Its effectiveness is assessed through automated testing and validation benchmarks covering configuration handling, topology visualization, and experiment workflow support. The main contribution of this work is a usable and extensible interface that lowers the entry barrier to PANTHER while remaining compatible with its plugin-based design and existing reproducibility methods.

Declaration of AI Assistance

Large language models were used during this thesis as a secondary development aid, mainly to help understand parts of the existing PANTHER codebase, explore implementation alternatives, debug code, and refine changes. They were not used as a substitute for the design, implementation, validation, or technical judgement presented in this thesis. The final decisions, code integration, testing, and written report remain the author's responsibility.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context, Problem, and Motivation

Modern network protocols such as QUIC have become increasingly complex, integrating sophisticated mechanisms for security, congestion control, and performance optimization. Ensuring their correctness, reliability, reproducibility, and interoperability requires systematic and automated testing methodologies.

PANTHER (Pluginizable Testing Environment for Network Protocols) is a research-oriented framework designed to orchestrate protocol implementations, formal testers, network environments, and execution environments in order to conduct reproducible protocol experiments. It combines containerized execution, formal verification using Ivy, and controlled network execution through plugins such as Shadow.

Before the work described in this thesis, PANTHER experiments were primarily configured and executed through YAML configuration files, command-line interfaces, and Docker-based workflows. This design favors flexibility and reproducibility, but it also introduces usability challenges for new users of PANTHER. Since the framework is plugin based, it is intended to allow developers and researchers to test their own implementations against available testers and environments. The difficulty lies in the need to understand PANTHER's own plugin architecture, configuration schemas, and execution model.

Defining and managing protocol experiments in PANTHER requires direct interaction with low-level configuration files and an in-depth understanding of how the framework connects services, protocols, testers, network environments, and execution settings. This steep learning curve limits PANTHER's accessibility for:

- Students using the framework for educational purposes,
- Researchers who are new to PANTHER or to formal protocol testing,
- Developers who wish to rapidly prototype experiments without deep knowledge of the framework's internal configuration structure.

PANTHER originally lacked a dedicated graphical interface that would support

visual design of experiment scenarios, interactive validation of configurations, and monitoring of experiment execution and results. As a result, experiment design was time consuming, difficult to scale for teaching or exploratory research use cases, and potentially error-prone.

Providing a graphical user interface (GUI) for PANTHER can lower the barrier to entry and broaden the framework's adoption beyond expert users. A well-designed GUI can abstract low-level details while preserving the expressive power of the underlying configuration system. This is especially important for PANTHER because experiment setup requires users to coordinate several related concepts at once. Visual feedback and guided configuration can make these relationships easier to inspect, reduce errors in complex YAML files, improve experiment reproducibility through structured templates, and accelerate early-stage prototyping and experimentation.

1.2 Objectives and Approach

The primary objective of this master thesis is to design and implement a frontend extension for PANTHER in the form of a graphical user interface. The GUI is intended to:

- Allow users to configure, run, and monitor PANTHER experiments without manually editing YAML files in the common case, while still exposing the underlying configuration for advanced users,
- Provide visual building blocks representing protocol implementations, testers, and network environments at the level of topology diagrams and structured forms, rather than requiring direct manipulation of raw configuration files,
- Ensure bidirectional consistency between the GUI and PANTHER's configuration format, including importing existing configurations and exporting valid experiment files,
- Support the full experiment lifecycle for typical workflows, including creation, execution, inspection of logs and results, and reuse of experiment templates.

The GUI is intended to complement, not replace, the existing PANTHER Command Line Interface (CLI), preserving full control for advanced users while improving accessibility.

The development of the PANTHER frontend follows a structured and iterative approach:

1. Analyze the existing PANTHER architecture, including its plugin system,

configuration schemas, and execution workflow, to identify all elements that must be represented in the GUI.

2. Design a modular frontend architecture capable of interacting with the PANTHER backend through its configuration files, service layer, and APIs rather than by duplicating the CLI workflow.
3. Implement a prototype GUI using an iterative, agile development process with short feedback loops.
4. Validate the prototype through expert reviews or small user tests, focusing on usability, correctness of generated configurations, and support for common PANTHER workflows.

UI/UX decisions are informed by existing literature on scientific and technical interface design to minimize cognitive load and support complex workflows intuitively.

1.3 Contributions and Roadmap

The main contributions of this thesis are as follows:

- A structured analysis of the usability problem in PANTHER's existing CLI/YAML workflow, identifying the need for a graphical layer that reduces configuration friction while preserving the framework's existing reproducibility and plugin-based design.
- The design and implementation of a browser-based PANTHER Web Interface that supports the main experiment lifecycle, including configuration authoring, topology inspection, experiment execution and monitoring, result inspection, and plugin discovery.
- An architectural integration layer that connects the web interface to PANTHER's existing configuration models, plugin system, execution logic, and result artifacts without replacing the underlying CLI-oriented framework.
- A testing and validation effort that evaluates both the functional correctness of the web interface and its workflow-level differences compared with the original CLI/YAML approach.

Due to unforeseen circumstances, the intended second author of this thesis was unable to continue the project, and the scope was adjusted accordingly. In addition, the assistant supervisor made key contributions to the development of the APIs used to connect the existing PANTHER framework to the web interface, as well as some of the skeleton code on which to build the interface (51 files, 6,054 lines of code).

The work presented in this thesis therefore focuses on the design, implementation, integration, testing, and validation of the user-facing web interface built on top of that supporting structure (74 files, 9,263 lines of code and 451 lines of tests).

The remainder of this thesis is organized around the development, testing, and validation of a graphical web interface for PANTHER in the context of protocol experiment management. It first establishes the technical and conceptual background needed to understand the framework, its configuration model, and the role of formal testing and controlled execution environments in reproducible protocol experimentation. It then reviews the most relevant existing approaches and highlights the gaps that motivate the proposed solution.

The thesis next introduces the design of the web interface and explains how the user-facing pages support the main experiment workflow, from configuration and topology inspection to execution, monitoring, and result analysis. This is followed by a discussion of the system architecture and technology choices that make the interface compatible with PANTHER's existing plugin-based and YAML-driven design. The testing chapter then evaluates the functional correctness of the implementation through unit, integration, and end-to-end tests. The validation chapter separately assesses whether the interface improves selected workflows compared with the original CLI/YAML approach, focusing on configuration effort, context switching, error recovery, topology comprehension, and plugin discovery.

Chapter 2

State of the Art and Design Foundations

This chapter brings together the state of the art, the user interface considerations, and the technology choices that guide the PANTHER Web Interface. It answers the design problem from three directions: existing work, interface design, and implementation constraints.

The chapter follows four questions:

1. What constraints already exist in PANTHER?
2. What does existing work suggest for tools like this?
3. What UI principles follow from that?
4. What technologies best fit those principles and constraints?

2.1 PANTHER Background and Inherited Constraints

This section introduces the technical context needed to understand the PANTHER Web Interface. It summarizes the underlying PANTHER framework, the role of declarative configuration in experiment execution, and the protocol-testing setting in which the interface operates. It also separates inherited PANTHER choices from technologies selected specifically for this thesis.

2.1.1 PANTHER framework and experiment workflow

PANTHER is a modular and extensible framework for automated testing and verification of network protocols. It allows researchers to combine protocol implementations, formal testers, network environments, and execution environments into a single experiment definition. The framework supports reproducible protocol experiments by orchestrating these components in a structured workflow [4].

A central characteristic of PANTHER is its plugin-based architecture. Rather than embedding support for a single protocol or fixed execution scenario, the framework defines separate plugin types for implementations under test, testers, protocol metadata, service components, network environments, and execution environments. This allows the same core workflow to be reused across different protocols and experimental settings, while preserving a common execution model.

PANTHER experiments are defined declaratively through YAML configuration files. These files describe the participating services, the protocol settings, the selected plugins, the network environment, and the execution environment. Once loaded, the configuration is validated against structured schemas and then used to instantiate the components required for the experiment. The execution workflow follows a sequence of phases: validate the configuration, load the requested plugins, prepare the services, deploy the selected environment, run the test logic, and collect logs, metrics, and output artifacts.

2.1.2 Configuration, validation, and plugin constraints

YAML is widely used for structured configuration because it is both human-readable and well suited to automation. In PANTHER, YAML files define experiment structure in a form that can be stored in version control, shared across users, and reused across executions. This makes the configuration not only an input format but also an important artefact of reproducible experimentation.

PANTHER couples YAML with schema-based validation. Before an experiment is launched, the framework checks that the configuration is structurally correct and that required parameters are present and well-typed. This validation step reduces the likelihood of invalid experiment definitions reaching the execution phase and provides a more reliable basis for protocol testing.

PANTHER integrates Ivy as a source of formal, stateful protocol tests. Ivy allows protocol behavior to be described at the specification level and supports the generation of tests that exercise an implementation against that formal model. Within PANTHER, Ivy-based testers are represented through the plugin mechanism as service-type components that can be combined with protocol implementations and execution environments. This makes formal testing part of the normal experiment workflow rather than an isolated verification activity.

PANTHER also supports experiments in controlled network environments, including Docker-based deployments, localhost execution, Docker Compose setups, and Shadow-based simulation. Shadow is one network environment plugin rather than the only supported environment. By providing deterministic execution under simulated network conditions, it supports experiments in which latency, topology,

and distributed interactions must be controlled precisely. Execution environments are separate from network environments: they describe how experiment components are launched and managed, while network environment plugins describe the communication context in which those components interact [11].

2.1.3 Inherited constraints for the web interface

Some technologies that shape the web interface are inherited from the main PANTHER framework. PANTHER already provides the protocol-testing foundation, and the web interface simply extends it. This relationship strongly influences the technology choices. The selected technologies allow the interface to reuse PANTHER's existing models, services, configuration files, and execution infrastructure. The result is a web layer that remains technically aligned with the original project.

These inherited choices act as constraints on the webapp implementation. Reusing PANTHER's Pydantic models avoids the creation of a separate UI-specific configuration schema. It also keeps form generation, default values, validation, and error reporting aligned with the core framework. The existing YAML configuration format also affects the interface design. The webapp must preserve compatibility with experiment files used by the rest of PANTHER and with CLI-based workflows, so it does not introduce a separate persistent project format.

The inherited execution and observer mechanisms influence the runtime structure of the webapp as well. Experiment execution can be long-running and produces progress information incrementally. This motivates the service-oriented integration layer, the `WebObserver` adapter, and asynchronous UI update patterns. The existing Python-centred architecture also favours a Python-based UI framework, because the graphical layer can remain close to PANTHER's services and configuration logic instead of duplicating them in a separate frontend application.

PANTHER is therefore powerful, but it is also structurally complex. A user has to understand YAML files, schema validation, plugin types, service definitions, execution environments, network environments, generated logs, metrics, and output artifacts. This complexity is manageable for expert users, but it creates friction for users who want to configure, inspect, or modify experiments quickly. The web interface is motivated by this gap: it provides a graphical layer that exposes the same PANTHER model through guided forms, topology views, plugin discovery, validation feedback, and result inspection.

2.2 State of the Art in Experimentation Interfaces

The technology choices are informed by the broader state of the art in cyber range, experimentation, and network simulation platforms. Modern cyber range systems often expose complex infrastructure through web-based interfaces that combine scenario definition, orchestration, monitoring, and result inspection. This is visible in platforms such as KYPO Cyber Range Platform, which is an open-source cyber range aimed at training, development, and execution [14], and EDURange, which provides cloud-based cybersecurity exercises for educational use [15].

These systems show that web-based access is an established pattern for complex technical environments. Users are not expected to interact directly with every low-level infrastructure component. Instead, the interface provides a task-oriented layer over the orchestration system. PANTHER operates in a different domain, focusing on protocol testing rather than general cybersecurity training, but the same design lesson applies: a web interface can make complex experimental infrastructure easier to configure, observe, and reuse. The main PANTHER framework already provides protocol testing, formal tester integration, controlled environments, and configuration-based automation [4]. The web interface exposes these capabilities without replacing them.

2.2.1 Lessons from network simulation and cyber range tools

Existing network simulators provide useful lessons about usability in technically complex research tools. A comparative survey of open-source and commercial simulators notes that tools such as NS-2, NS-3, and J-Sim are powerful and feature-rich, but can suffer from weak graphical interfaces, limited visualization support, and difficult installation or configuration procedures [7]. This creates a gap between technical capability and practical accessibility.

This observation is relevant to PANTHER because protocol experimentation also requires users to coordinate multiple technical concepts, including configuration files, services, protocol roles, runtime environments, and test artifacts. If these concepts are exposed only through scripts and command-line interaction, the entry barrier remains high. The same survey also emphasizes that documentation quality and surrounding tooling, such as scenario editors, trace viewers, and debugging aids, are important factors in adoption [7]. For PANTHER, this suggests that the UI should not merely expose controls. It should organize information in a way that helps users understand what they are configuring and why it matters.

Cyber range platforms provide a related reference point for interface expectations. Cyber ranges are typically interactive environments that mirror realistic networked systems, including virtual machines, services, traffic, and scenarios, to support hands-on training, testing, and experimentation in controlled conditions [16]. Their interfaces often emphasize scenario management, observability, and guided interaction over direct manipulation of low-level infrastructure. PANTHER is not a cyber range, but it faces a similar interface problem: making a complex technical infrastructure understandable and operable through a coherent user-facing layer [16].

2.2.2 Topology and scenario representation

Topology visualization is a specific instance of the broader need for scenario representation. In PANTHER, topology-related information can be distributed across test definitions, service entries, protocol roles, targets, and environment settings. A user reading the YAML file must mentally combine these fields to understand the resulting experiment structure.

The interface therefore treats topology as a design concern rather than only as an implemented page. A graph representation can make relationships between services explicit. It also follows the lesson drawn from cyber range and network simulation tools: scenario structure should be visible, inspectable, and connected to the underlying configuration. The later implementation chapter describes the concrete Topology Editor page. The design rationale belongs here because it follows from the state of the art.

2.2.3 Implications for PANTHER

The state of the art suggests three main implications for PANTHER:

1. Web-based access is suitable when a tool exposes complex infrastructure.
2. Scenario and topology representations help users understand how services, roles, and environments relate to one another.
3. Observability matters because experiment execution produces runtime information that users need to inspect.

These implications connect the state of the art to the interface design. The PANTHER Web Interface therefore emphasises browser access, guided scenario construction, topology inspection, structured feedback, and result exploration. The technology stack is selected to support these ideas while staying close to the original PANTHER architecture.

2.3 User Interface Design Principles

This section presents the user interface considerations that guide the design of the PANTHER Web Interface. These include lowering the access barrier, reducing configuration complexity, supporting visual understanding, improving validation feedback, handling errors gracefully, and helping users discover relevant information without relying entirely on command-line knowledge or external documentation.

2.3.1 Access model and overall usability

The introduction identifies the central usability problem: PANTHER is powerful, but its existing CLI/YAML workflow assumes that users understand the framework's configuration structure, plugin model, and execution process. This can make the tool difficult to approach for students, new researchers, or users who only need to run or modify experiments occasionally.

The interface is therefore designed around a low-friction access model. By using a browser-based interface, users can interact with PANTHER through standard web technologies rather than through a dedicated desktop application or a purely command-line workflow. This is especially relevant in educational and research settings, where users may work on shared machines, managed lab computers, or remote infrastructure. This design choice also follows practices seen in cyber range portals, where frictionless access is important for supporting large and diverse user groups such as students, trainees, and external collaborators.

2.3.2 Configuration complexity and cognitive load

A major design concern is the cognitive load created by manual configuration. PANTHER configurations are expressive, but they can also be deeply nested and sensitive to field names, indentation, and valid combinations of parameters. YAML is well suited to reproducible experimentation, but writing and maintaining it manually can be difficult when users must remember the structure of tests, services, protocols, environments, and execution settings.

The interface therefore follows a guided configuration approach. Instead of requiring users to construct every configuration element from raw YAML, the UI can present structured fields, grouped sections, constrained inputs, and validation-aware controls. This reduces the amount of information the user must keep in memory and shifts attention from syntax management toward experiment design. This does not remove the importance of YAML. As discussed earlier in this chapter, YAML remains the canonical configuration representation. The interface is intended to

provide a safer and more accessible layer over that representation, not to replace the underlying configuration-as-code model.

2.3.3 Experiment visualization and analytic presentation

PANTHER experiments produce and depend on structured information such as services, protocol relationships, runtime events, logs, metrics, and result artifacts. If all of this information is presented only as text, users must spend additional effort interpreting relationships and identifying relevant patterns. Visual presentation is therefore an important design consideration.

The interface uses visualization as a way to support comprehension rather than as decoration. Graphs, tables, charts, and summary indicators can help users move from high-level understanding to detailed inspection. Interactive visualization is also useful because experiment data is often exploratory: users may need to filter, inspect, compare, and drill into specific parts of an experiment. This mirrors analytic practices in cyber range interfaces, where visualization helps users identify anomalies, understand system behaviour, and reflect on outcomes rather than merely consume static numerical summaries. More advanced visualization patterns, such as customizable dashboards or more complex multidimensional visual encodings, are outside the scope of the current prototype but remain possible extensions.

2.3.4 Validation feedback and error recovery

The interface must support not only configuration input, but also recovery from mistakes. In a CLI/YAML workflow, errors are often discovered after running a validation or execution command. The user must then interpret the error message, return to the configuration file, locate the relevant field, and try again. This workflow can be slow, especially when the error concerns nested configuration structures.

For this reason, the UI design emphasizes validation feedback close to the point of correction. Form fields and related dialogs should surface issues such as missing required values, invalid formats, or inconsistent combinations as directly as possible. Inline messages and visual indicators help users understand what needs attention without losing the broader context of the task. Empirical work on web form usability has shown that immediate, inline validation improves completion rates and reduces user frustration compared with designs that only report errors after full form submission [8]. Because the interface's forms are grounded in Pydantic models, validation feedback can remain consistent with the same schema constraints used by the underlying system.

2.3.5 Resilience and feedback mechanisms

Because PANTHER interacts with files, plugins, execution environments, and long-running processes, the interface must be designed for partial failure. A graphical interface should not collapse entirely when one component fails to load or when an external dependency is unavailable. Instead, failures should be localized, visible, and recoverable.

The design therefore favours error containment and graceful degradation. If a view or component cannot complete its operation, the user should receive a clear message and a possible next action, while the rest of the interface remains usable. This aligns with Don Norman's recommendations on designing for error and recovery [9], which emphasize that resilient systems preserve user trust by failing gracefully rather than catastrophically. A consistent notification system also supports this goal. Messages should distinguish between success, warnings, errors, and neutral information using predictable visual patterns. Research in visual communication and Human Computer Interaction (HCI) suggests that structured, colour-coded notifications help users quickly interpret the severity and intent of system feedback, reducing ambiguity and improving task performance [6].

2.3.6 Information discovery and guided exploration

Another important design objective is to help users discover relevant information without forcing them to search manually through files, logs, or command output. PANTHER contains many different kinds of information, including configuration fields, plugin metadata, experiment states, events, metrics, and artifacts. If this information is not organized carefully, the interface can become as difficult to navigate as the command-line workflow it is meant to improve.

The interface therefore follows a layered information design. High-level information should be visible first, while more detailed information should remain available through tables, expandable regions, filters, or detail views. Visual hierarchy is important in this structure. Grouping information into clear sections, using consistent typography, and presenting high-level status before secondary details helps users build a mental model of the system without scanning large undifferentiated blocks of text or data [1]. Search and filtering are also important for guided exploration. When users search within structured information, the interface should make clear which items match and why they match. Highlighting matched terms and showing result-state feedback follows information foraging theory, where visible cues help users decide where to inspect next [2].

2.4 Technology Choices

This section explains the main technology choices behind the PANTHER Web Interface. The selected stack has to remain close to the existing PANTHER codebase, preserve compatibility with the configuration model, and support structured validation, browser rendering, visualisation, and long-running experiment feedback without adding unnecessary architectural complexity.

2.4.1 Python-centred web interface development

The primary UI framework used for the interface is *NiceGUI*. NiceGUI is a Python-based framework for creating browser-based user interfaces, with a declarative component model for layouts, inputs, buttons, tables, dialogs, charts, and other interactive elements [17]. NiceGUI also describes this approach as "web-based user interfaces with Python" [12]. This makes it a good fit for the PANTHER frontend because most of the application logic can remain in Python, close to the existing configuration and validation services.

NiceGUI avoids much of the overhead of a separate JavaScript or TypeScript frontend, such as a separate build pipeline, client-side state-management layer, and duplicated API contracts. At the same time, it builds on established web technologies such as FastAPI, Starlette, Vue, and Quasar [12]. The result is a practical compromise: the implementation benefits from browser-based rendering and modern UI components while keeping the prototype aligned with PANTHER's Python-centred architecture.

2.4.2 Backend and API foundation

The backend foundation is based on *FastAPI*, which is also part of the underlying NiceGUI stack. FastAPI is an ASGI web framework designed around Python type hints and asynchronous request handling [10]. This fits PANTHER because configuration handling and validation are naturally type-oriented tasks, and because the web interface needs structured backend operations when direct UI callbacks are not sufficient.

The implementation follows a service-oriented internal structure. UI components do not directly manipulate lower-level PANTHER internals. Instead, they call services for configuration loading, validation, plugin discovery, topology extraction, result parsing, and experiment control. This keeps rendering concerns separate from domain operations and makes the interface easier to maintain and test.

2.4.3 Browser-side visualisation and code editing

Although the implementation is Python-centred, some interface features depend on browser-side components. For visualisation, the interface uses *Apache ECharts*, an open-source JavaScript visualisation library for interactive browser charts and graphs [13]. This is suitable for PANTHER because experiment configurations and results often contain relationships, metrics, events, and structured data that are easier to understand visually than as raw text.

For textual configuration editing, the interface uses a CodeMirror-based editor. CodeMirror is a browser-based code editor component designed for embedding structured text editing into web applications [5]. This matters because textual configuration remains an important representation in PANTHER, and a code editor is better suited than a plain text area for indentation-sensitive YAML. These browser-side tools are therefore used as targeted components, while Python remains the main implementation language.

2.4.4 Asynchronous execution and event handling

Protocol experiments may involve long-running operations such as environment preparation, service startup, test execution, logging, and artefact collection. A web interface for such a system must remain responsive while these operations are in progress. The implementation therefore uses asynchronous and event-driven patterns.

The code separates long-running backend work from the interactive UI loop. Background execution prevents the browser-facing application from blocking, while event propagation allows the interface to receive updates as the system state changes. This matches the nature of PANTHER experiments, where useful feedback is often produced during execution rather than only at the end.

Chapter 3

PANTHER User Interface

This chapter presents the implemented PANTHER Web Interface and the concrete functionality of the system, including how the interface is organized, how users move through the experiment lifecycle, and how the individual parts of the interface support configuration, topology inspection, execution, monitoring, result analysis, and plugin discovery.

The source code for the web interface implementation is available at `https://github.com/umer901/PANTHER/tree/production/panther/webapp`. The file `USER_GUIDE.md` gives instructions to run the webapp, run the tests and a basic overview of the pages.

The interface is designed around the same underlying experiment model as the existing PANTHER workflow. YAML remains the canonical representation of an experiment, plugins remain the mechanism through which protocols, testers, implementations, and environments are selected, and the backend continues to perform validation and execution. The role of the web interface is to expose these capabilities through connected visual and form-based views, reducing the need for users to manually reconstruct experiment state from separate files, commands, logs, and output directories.

3.1 Overview

The PANTHER Web Interface is organized into six main pages, as shown in table 3.1: Dashboard, Configuration Builder, Topology Editor, Experiments, Results, and Plugin Browser. These pages correspond to the major activities involved in experiment management. A persistent sidebar provides navigation between pages, while shared status and activity elements help maintain continuity across the interface.

Page	Purpose
Dashboard	Provides a high-level overview of the current system state, recent activity, and entry points into common actions.
Configuration Builder	Allows users to create, edit, validate, import, export, load, and save experiment configurations through structured forms synchronized with YAML.
Topology Editor	Displays services and protocol relationships as a graph, supporting both aggregated and per-test inspection of experiment structure.
Experiments	Supports experiment selection, launch, stopping, progress tracking, live logs, and structured runtime event monitoring.
Results	Provides access to completed experiment outputs, including summaries, logs, events, metrics, per-test information, and generated artifacts.
Plugin Browser	Displays available plugins and metadata so that users can understand which protocol implementations, testers, environments, and related components are available.

Table 3.1: Main pages of the PANTHER Web Interface and their respective purposes.

The overall workflow supported by these pages is cyclical rather than strictly linear. A user may begin by inspecting available plugins, creating or loading a configuration, inspecting its topology, running an experiment, monitoring progress, viewing results, and then returning to configuration editing. This reflects the iterative nature of protocol experimentation where experiments are often refined over several cycles rather than defined and executed only once.

The interface also supports two levels of interaction. At the higher level, it presents task-oriented pages that guide users through common actions. At the lower level, it preserves access to the underlying YAML and artifacts, allowing advanced users to inspect, export, and reuse the same files that remain compatible with the CLI workflow.

3.2 Dashboard

The Dashboard acts as the landing page of the web interface. Its purpose is to provide an immediate overview of the current PANTHER environment without requiring the user to run exploratory commands or manually inspect project

directories. It displays summary information such as available plugins, experiment-related status, recently used or available configurations, and recent activity.

The dashboard workflow begins when the user opens the application. Instead of starting from an empty terminal prompt, the user is presented with the current state of the system and a set of common next actions. These actions may lead to configuration editing, experiment execution, result inspection, or plugin exploration.

The Dashboard also supports the information-discovery principle discussed in the previous chapter. High-level status is presented first, while detailed work is delegated to specialized pages. This prevents the landing page from becoming overloaded while still giving users enough context to decide what to do next.

3.3 Configuration Builder

The Configuration Builder implements the configuration-authoring part of the experiment lifecycle. It is one of the most important parts of the interface because PANTHER experiments are defined through structured YAML files, and manual YAML editing is one of the main sources of friction in the original workflow.

The page provides a form-based editor, shown in figure 3.1 that represents the major parts of an experiment configuration: global settings, metadata, test definitions, services, protocols, environments, execution options, and related parameters. These sections are grouped visually so that users can work on one part of the configuration at a time. This follows the cognitive-load principle introduced earlier, ensuring that users do not need to remember the entire nested YAML structure while editing a specific part of an experiment.

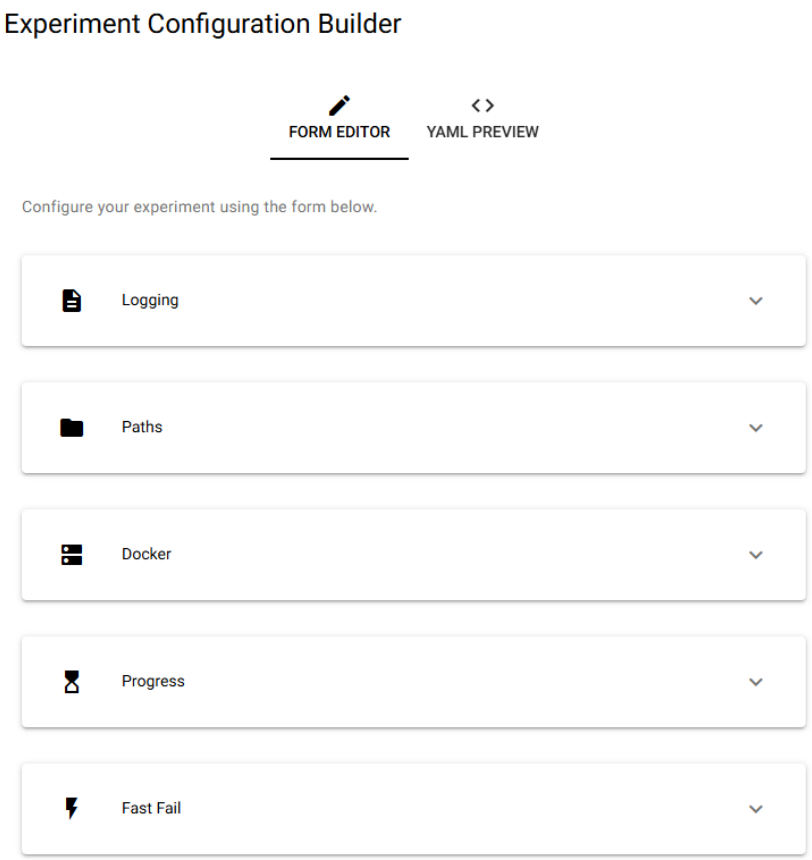

Figure 3.1: Configuration options

A typical configuration workflow starts with either creating a new configuration or loading an existing one. When creating a new configuration, the user can fill in structured fields, add test definitions, define services, select protocols, and specify environment or execution settings. When loading an existing configuration, the interface parses the YAML and populates the corresponding form fields. This allows an existing CLI-compatible configuration to become editable through the graphical interface.

The form editor and YAML representation remain connected. Users can inspect the YAML preview to see the textual configuration generated from the form, and they can also import YAML directly when needed. This is important because the interface does not introduce a separate configuration language. Instead, it provides a structured editing layer over the same YAML format used elsewhere in PANTHER.

The page also supports validation and persistence. After editing a configuration,

the user can validate it before running an experiment. Validation feedback is shown inside the same interface, so the user does not need to switch to a terminal command only to discover whether the configuration is structurally valid. If validation fails, errors can be shown with field or path information, helping the user identify the relevant section. Once the configuration is valid, it can be exported as YAML or saved for later use.

The validation chapter quantifies the difference between the YAML and form-based authoring effort. The qualitative point here is that the UI does not remove configuration work entirely. Users still choose services, parameters, plugins, and environments. The difference is that the interface shifts much of the work away from syntax, indentation, and schema recall, and toward guided field editing.

3.4 Topology Editor

The Topology Editor provides a visual representation of the services and relationships described in a configuration. In a YAML file, topology-related information is distributed across test definitions, service entries, protocol roles, protocol targets, and environment settings. Understanding the resulting experiment structure may require mentally combining several fields. The Topology Editor makes these relationships explicit.

A typical topology workflow begins with selecting a configuration file. Once loaded, the interface parses the configuration and extracts graph information. Services are represented as nodes, and relationships such as client-to-server targets are represented as edges, as shown in figure 3.2. The page can show an aggregated view across the full configuration as well as views focused on individual tests.

The topology graph supports interactive inspection. Users can pan, zoom, drag nodes, and inspect service metadata through hover or selection interactions. Summary information such as the number of tests, services, connections, and environments helps users understand the scale and structure of the loaded configuration before inspecting individual details.

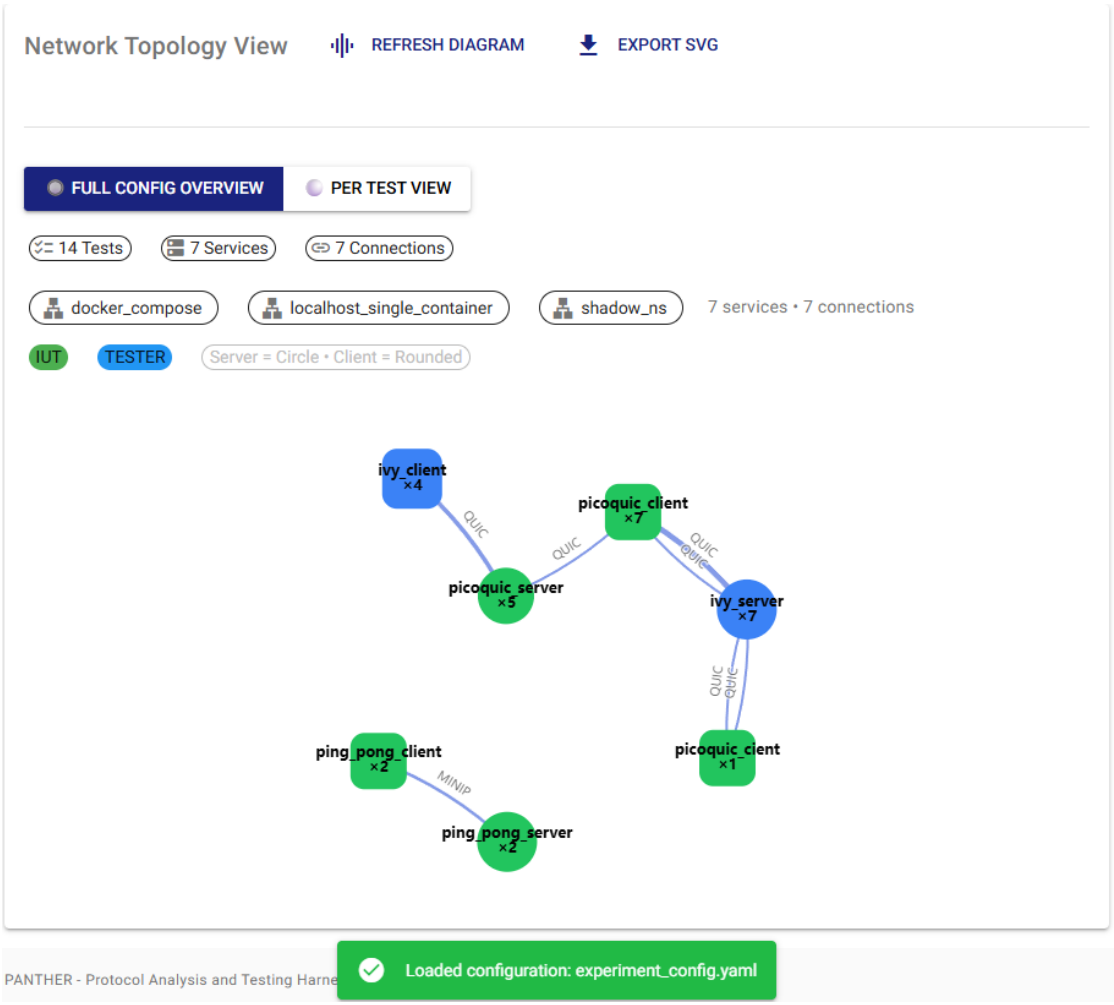

Figure 3.2: Topology view

The Topology Editor also connects back to configuration editing. When a user identifies a service or relationship that needs modification, the interface can route the user back to the corresponding configuration context. This connection reflects one of the key design decisions of the interface: topology inspection and configuration editing should not be isolated activities. The graph helps users understand the experiment, while the Configuration Builder remains the place where the canonical configuration is edited.

The topology can also be refreshed from disk and exported as an SVG image. This makes it useful for documenting experiment structure for purposes such as reports or presentations.

3.5 Experiments

The Experiments page supports the execution and monitoring phase of the lifecycle. Its purpose is to connect a selected configuration to an experiment run and then keep execution state visible while the run progresses. This is important because PANTHER experiments may involve several phases, such as validation, plugin loading, environment preparation, service startup, test execution, log collection, and result generation.

A typical workflow begins with selecting a configuration. The user can browse available configurations, search or filter them, and inspect summary information before launching. After a configuration is selected, the interface displays relevant details about it. This may include path information, environment data, protocols, services, and test-related information. The goal is to present enough context for the user to understand what will be executed without forcing them to open the raw YAML file separately.

When the experiment is launched, the page shifts from selection to monitoring. The interface displays current status, progress information, raw logs, structured log tables, and event information. This reflects the feedback and resilience principles discussed in the previous chapter that state long-running operations should not appear as opaque terminal processes, but as observable activities with visible state changes.

The raw log view preserves access to detailed execution output. This is useful for debugging and for users who are already familiar with terminal-style logs. However, the interface also provides structured views so that users do not need to interpret all runtime information from a single stream of text.

The progress timeline gives a higher-level view of execution phases as shown in figure 3.3. This helps users distinguish between normal waiting, active execution, completed phases, and possible failures.

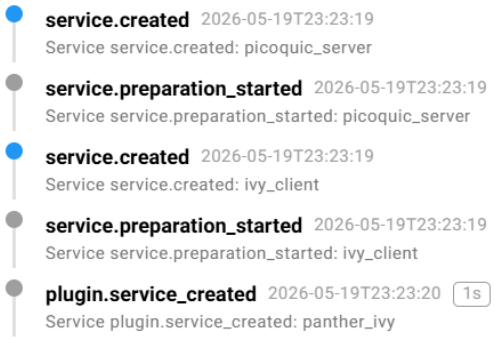

Figure 3.3: Experiment progress timeline

The page also supports stopping a running experiment. If a stop action is triggered, the interface continues to display state changes and shutdown-related events until termination is complete. This avoids the ambiguity that can arise when a process is interrupted but its final state is not visible.

3.6 Results

The Results page supports post-execution inspection. Once experiments have completed, users need to locate outputs, inspect logs, review metrics, understand per-test outcomes, and retrieve artifacts. In a file-oriented workflow, these tasks may require navigating directories and opening multiple output files. The Results page presents these outputs through a unified interface.

A typical results workflow begins by locating a completed experiment. Users can search or filter results and then open a detailed view for a selected run. The detailed view organizes the output into multiple categories, allowing the user to move from summary information to more specific data. An overview of the summary page is shown in figure 3.4.

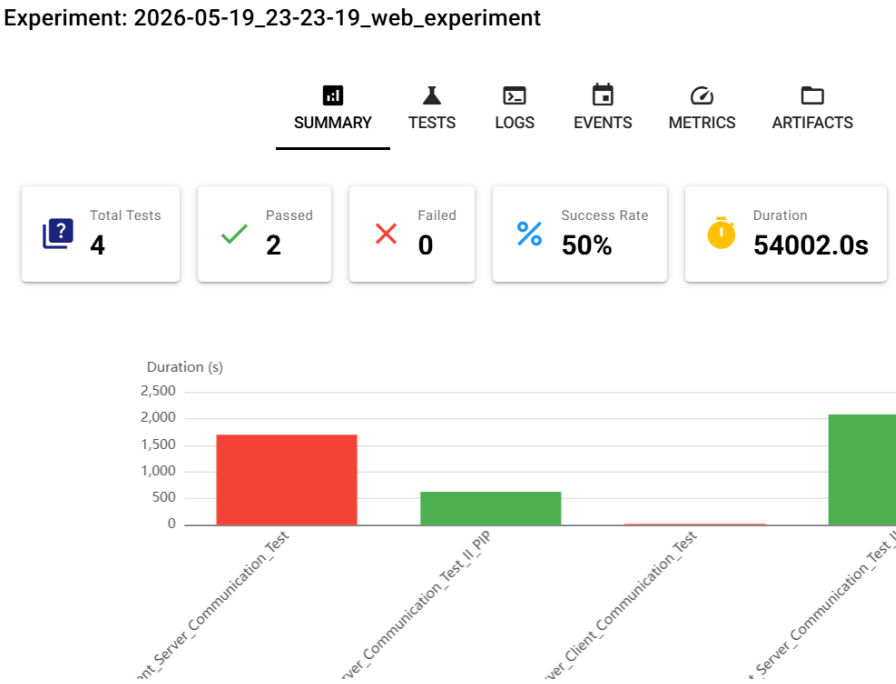

Figure 3.4: Experiment test results

The page supports different levels of analysis. A user may first check whether an experiment succeeded or failed, then inspect test-level information, raw logs, structured events, metrics, and generated files. This layered presentation follows the guided-exploration principle from the previous chapter: users should be able to begin from an overview and move toward detail when necessary.

The Results page also supports reuse. Since experiment outputs are artifacts of reproducible research, the interface must make them discoverable after the run has completed. Search, filtering, and structured detail views help users return to previous outputs without reconstructing the context entirely from filenames or terminal history.

3.7 Plugin Browser

The Plugin Browser exposes the plugin ecosystem that underlies PANTHER's extensible architecture. PANTHER relies on plugins to represent protocol implementations, testers, network environments, execution environments, protocols, and other extensible components. For users to construct valid experiments, they must

be able to discover what components are available and understand how they relate to experiment configuration.

The Plugin Browser presents installed plugins as browsable entries with associated metadata. Users can inspect plugin type, description, protocol association, dependencies, configuration information, and other manifest-derived details. This supports discovery without requiring the user to rely only on command help, documentation, or manual source-code inspection.

The main plugin workflow begins with browsing or filtering the available plugin set. Since larger PANTHER installations may contain many plugins, search and category-based organization are important. They allow users to narrow the set of visible components and focus on the plugin types relevant to the experiment they are constructing.

A second workflow concerns detailed inspection. When a plugin is selected, the interface displays additional information in a detail area. This allows the user to understand a plugin without leaving the current context.

The Plugin Browser is not only a catalogue. It also supports the broader configuration workflow because plugin discovery is connected to experiment authoring. Understanding which implementations, testers, and environments are available is a prerequisite for constructing meaningful experiment configurations.

3.8 Comparison with the CLI/YAML Workflow

The web interface differs from the original CLI/YAML workflow mainly in how it distributes user effort. In the CLI workflow, all activities are combined into text editing and terminal interaction. In the web interface, these activities are separated into more explicit views.

This does not mean that the web interface removes configuration work. Users still make the same domain decisions of which protocol implementation to use, which tester to select, which services to configure, which environment to run, and which parameters to set. The difference is that these decisions are expressed through structured forms, visual topology inspection, selectable plugins, and integrated validation rather than only through raw YAML and command-line output. The validation chapter quantifies some of these differences with concrete metrics such as configuration authoring effort, context switching, invalid-configuration recovery, topology lookup effort, and plugin discovery.

Chapter 4

System Architecture

The web interface is continuous and must keep state visible, react to events, and let users move between configuration, topology, execution, and results without rebuilding context manually. This chapter describes the architecture of the PANTHER Web Interface and how it achieved these goals, while maintaining good software practices.

4.1 Architectural Overview

Figure 4.1 presents the system as a set of connected layers. The frontend contains the six user-facing pages of the web interface: Dashboard, Configuration Builder, Topology Editor, Experiments, Results, and Plugin Browser. These pages are the entry points for the experiment lifecycle, but they do not directly implement the PANTHER domain logic.

Between the frontend and the backend, the architecture includes a UI services and state-management node. This part is important because it turns the web interface from a set of isolated pages into a continuous application. It stores active context such as the selected configuration, selected experiment, validation status, runtime events, and current result. It also routes page actions to backend services. Without this layer, each page would have to reconstruct context from files or direct backend calls.

The backend side exposes the main operations needed by the interface. It includes experiment running, configuration validation, and plugin management. These operations are wrappers around existing PANTHER functionality. This keeps the graphical layer aligned with the original framework.

The storage layer contains logs, results, metrics, artifacts, and configuration files. It is accessed through services, not directly by page code. This keeps the UI independent from the exact physical layout of the output directories and makes the presentation layer easier to change later.

The high-level architecture therefore has four responsibilities:

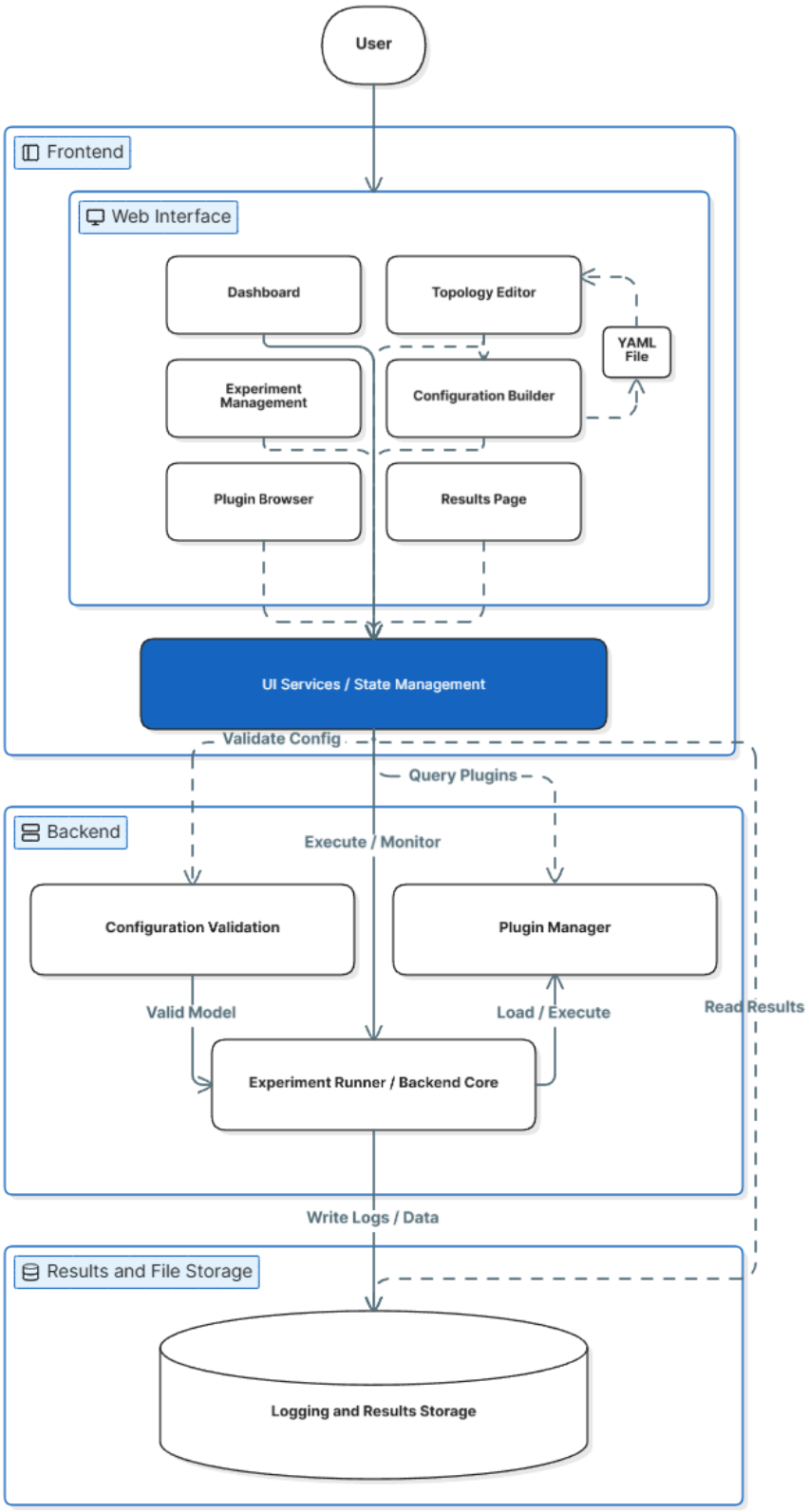

Figure 4.1: High-level architecture of the PANTHER Web Interface

1. The frontend presents the experiment lifecycle to the user.
2. The UI services and state-management layer preserves context across pages.
3. The backend adapts existing PANTHER functionality to web-facing operations.
4. The storage layer keeps experiment inputs and outputs available after execution.

4.2 Frontend Layer and Shared State

The frontend layer provides the browser-based interface through which users interact with the system. It contains the pages described in Chapter 3, shared layout elements, reusable UI components, navigation logic, and browser-side visualisation elements.

The six pages have separate user-facing purposes, but they share context. A configuration selected in the Configuration Builder may also be used by the Topology Editor and the Experiments page. Runtime status may appear in the Experiments page and later lead to the Results page. Plugin information may be inspected in the Plugin Browser and then used while creating a configuration.

Shared state allows the interface to keep track of the current workflow without forcing the user to repeatedly select the same files or rediscover the same information. The state-management layer also supports navigation paths such as moving from a topology node back to the corresponding test or service in the Configuration Builder.

The frontend uses reusable components where repeated behaviour appears across pages. Examples include cards, tables, dialogs, notifications, form sections, YAML editing elements, log viewers, progress indicators, and graph components. This reduces duplication in the implementation and makes visual behaviour more consistent.

4.3 Backend and Service Layer

The backend and service layer adapts PANTHER's core functionality to the needs of the web interface. Page code does not directly manipulate low-level experiment logic or storage structures. Instead, pages call dedicated services that provide stable operations for configuration handling, plugin discovery, topology extraction, experiment execution, and result retrieval.

The service layer has several responsibilities:

1. Load, save, import, and export YAML configuration files.
2. Validate configurations against PANTHER's schema model.
3. Convert configuration data into form-friendly structures.
4. Discover installed plugins and expose plugin metadata.
5. Extract topology graphs from experiment configurations.
6. Launch, stop, and monitor experiments.
7. Collect logs, events, metrics, and result artifacts for display.

This boundary separates presentation from domain logic. Pages handle rendering and interaction. Services handle files, models, plugins, event handling, and experiment outputs. The separation also helps with testing. A service such as topology extraction or configuration validation can be tested without rendering a full browser page.

The service layer also translates backend outcomes into user-facing feedback. A raw exception or validation error is often too technical to show directly. The service layer can convert it into a status, message, path, or structured result that the frontend can display consistently.

4.4 Code Structure

Figure 4.2 shows the main classes and dependencies involved in the PANTHER Web Interface. The diagram is organized around four parts: UI components, webapp services, the event bridge, and existing PANTHER core dependencies. It uses UML notation to distinguish inheritance, composition, association, and dependency relationships.

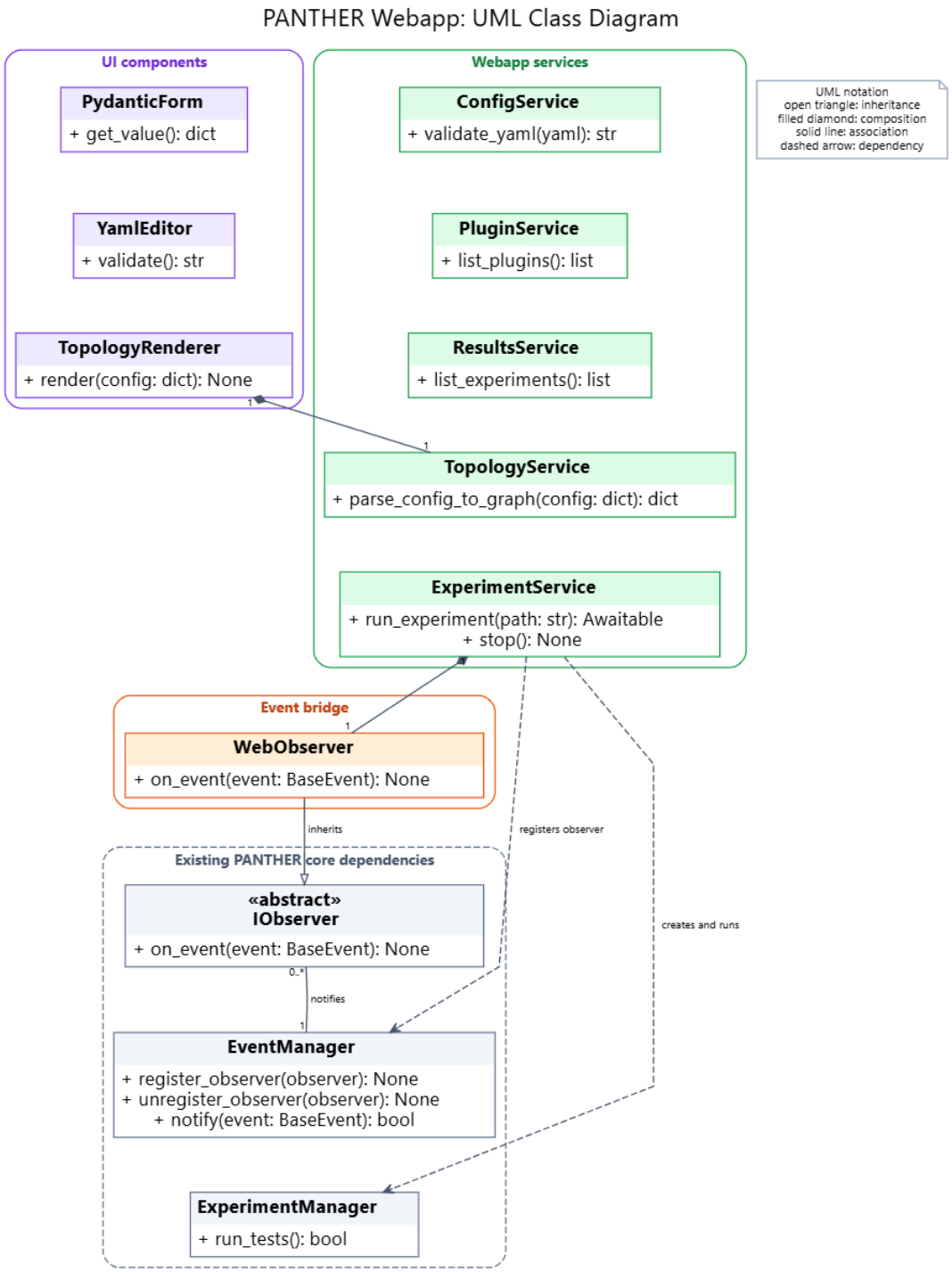

Figure 4.2: UML class diagram of the PANTHER Web Interface

The UI component group contains the main reusable interface elements. `PydanticForm` extracts and returns structured form values, `YamlEditor` supports textual YAML validation, and `TopologyRenderer` renders topology information from configuration data. These classes present and collect user-facing data, but they do not execute PANTHER experiments directly.

The webapp service group contains the main operations exposed to the interface. `ConfigService` validates YAML, `PluginService` lists available plugins, `ResultsService` lists experiment outputs, and `ExperimentService` starts or stops experiment execution. `TopologyService` transforms a parsed configuration into graph data through `parse_config_to_graph`. This keeps page-level code separate from configuration handling, plugin lookup, topology extraction, and experiment control.

The diagram also shows how the topology view is structured. `TopologyRenderer` depends on `TopologyService`, which means that rendering and graph extraction are separated. This is important because the visual topology should remain derived from the underlying configuration rather than becoming an independent source of truth.

The event bridge connects the web interface to the existing PANTHER event system. `WebObserver` inherits from `IObserver` and implements `on_event`. It is registered with `EventManager`, which can notify multiple observers when experiment events occur. This allows the web interface to receive progress and status updates while an experiment is running.

The existing PANTHER core dependencies remain responsible for experiment execution and event management. `ExperimentService` creates and runs experiments through `ExperimentManager`, while `WebObserver` receives events through the existing observer mechanism. The web interface therefore wraps the PANTHER core rather than reimplementing it.

4.5 Configuration, Validation, and Topology Flow

Configuration handling is central to the architecture because YAML remains the canonical experiment representation. The web interface therefore supports both structured editing and textual configuration fidelity.

The configuration flow begins when a YAML file is loaded, imported, or created from a template. The backend parses the YAML and validates it against the relevant Pydantic models. The resulting structured data can then be rendered in the form editor. When the user edits fields, the interface updates the corresponding

configuration state and can regenerate or preview YAML. When the user validates or saves, the data is converted back into the canonical form expected by PANTHER.

This supports bidirectional movement between YAML and UI forms. Existing CLI-authored configurations can be opened in the web interface. Web-authored configurations remain usable by the CLI. This matters for reproducibility because the interface does not introduce a hidden project format.

Validation is integrated into this flow. Instead of discovering configuration errors only during experiment execution, the interface can call validation services earlier. Pydantic-based validation helps detect missing fields, invalid types, malformed structures, and schema violations before execution. The resulting errors can then be mapped back to the UI for correction.

The topology flow builds on the same configuration source. The Topology Editor does not maintain an independent graph model as the source of truth. When a configuration is selected for topology inspection, the topology service parses the relevant test and service structures. It identifies services, protocol roles, targets, environments, and relationships. These are transformed into graph nodes, edges, labels, and metadata for the frontend.

This design avoids divergence between the topology view and the actual configuration. If the configuration changes, the topology can be regenerated from the same source. If the user identifies something in the topology that needs to be changed, the interface can route the user back to the relevant configuration context.

4.6 Experiment Workflow

Figure 4.3 shows the activity flow of the web interface. It starts with opening the PANTHER webapp and either loading an existing YAML file or starting from a default template.

The first part of the flow covers configuration editing and validation. The user edits the experiment through Pydantic forms or the YAML editor. The configuration is then validated. If it is invalid, the interface shows field or YAML errors and returns the user to editing. If it is valid, the workflow continues to topology inspection.

The second part of the flow covers topology inspection. The user opens the topology page and selects the configuration. The system renders an aggregated graph or a per-test graph, with services represented as nodes and protocol targets represented as edges. If the user needs to edit a service, the topology page stores the navigation context and returns to the Configuration Builder with the target test or service open.

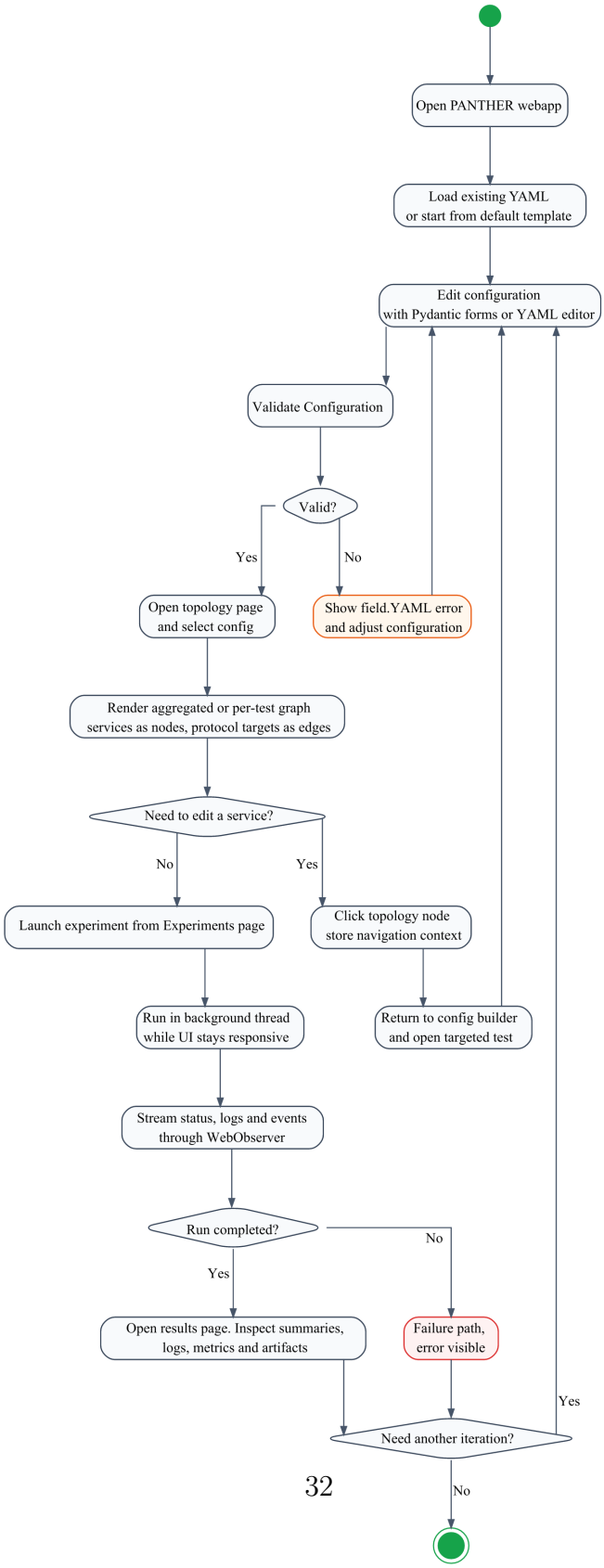

Figure 4.3: Activity flow of the PANTHER Web Interface

The third part of the flow covers experiment execution. The user launches the experiment from the Experiments page. The run is handled in the background so the UI remains responsive. Status, logs, and events are streamed through the `WebObserver`. When the run completes, the user can inspect summaries, logs, metrics, and artifacts in the Results page. If another iteration is needed, the workflow returns to configuration editing.

This activity flow captures the main architectural difference between the webapp and a one-shot command. The user does not move through isolated commands. The interface keeps the workflow connected across editing, validation, topology inspection, execution, and results.

4.7 Experiment Execution, Events, and Storage

Experiment execution introduces a runtime challenge. Experiments may involve environment preparation, service startup, test execution, logging, and artefact collection. These operations can take time. The browser interface must remain responsive while they run.

The architecture therefore separates experiment execution from UI responsiveness. Long-running execution is handled in the backend, while the frontend receives status updates, logs, and events through the event bridge. This allows users to monitor execution without blocking the rest of the interface.

The event flow follows the observer-oriented structure of the main PANTHER system. As the experiment progresses, events are emitted for relevant state changes. These may include initialization, environment setup, service actions, test execution, log output, warnings, errors, or completion. The web interface subscribes to these events through the `WebObserver` and converts them into visible UI state.

The storage layer contains the persistent artifacts generated or consumed by the system. This includes configuration files, logs, result summaries, metrics, event records, generated outputs, and other experiment artifacts. The backend writes to this layer during or after execution. The frontend reads from it through services when presenting results or historical information.

Loose coupling between storage and presentation is important. The frontend does not depend directly on the physical layout of every file or output directory. Result and configuration services interpret stored artifacts and expose them in a consistent form. This makes the UI more robust if storage conventions change.

4.8 Architectural Rationale and Quality Attributes

The architecture follows a separation of concerns between presentation, service logic, core execution, event handling, and persistence. The frontend handles interaction and rendering. The service layer adapts PANTHER capabilities to web-facing operations. The existing core remains responsible for experiment semantics and execution. The storage layer preserves configurations and outputs.

This structure supports maintainability. UI changes can be made without rewriting core experiment logic. Backend services can be tested independently from browser rendering. Storage formats can be interpreted through service adapters rather than hard-coded into every page. The plugin-based nature of PANTHER also supports extension. New protocol implementations, testers, or environments can be exposed through the same general mechanisms if their metadata and configuration schemas are available.

The architecture also supports responsiveness. Experiment execution, log generation, and event emission can continue while the browser updates status and output views. Instead of treating execution as a single blocking request, the system treats it as a sequence of observable state changes.

Reliability is supported through validation, state management, and error localization. Invalid configurations can be detected before execution. Errors can be shown near the relevant field or workflow step. Services can be tested at the right level, and browser tests can verify that rendered behaviour remains correct.

Chapter 5

Testing

This chapter documents the testing activities carried out to verify the functional correctness and robustness of the PANTHER Web Interface. Here, testing means checking whether the implemented web-facing layer behaves correctly. This includes page rendering, form behaviour, configuration validation, topology extraction, browser interactions, service boundaries, and route-level workflows.

The broader question of whether the interface improves the user workflow compared with the original CLI/YAML workflow is treated separately in Chapter 6. This keeps correctness evidence separate from workflow-improvement evidence.

5.1 Testing objectives

The testing effort aims to demonstrate the following claims:

1. The Configuration Builder preserves the structure and meaning of experiment definitions when editing, validating, importing, exporting, loading, and saving configurations.
2. Pydantic-based validation detects malformed or incomplete configurations and returns actionable feedback to the interface.
3. The Topology Editor derives graph information from YAML configurations, including services, tests, connections, network metadata, and navigation payloads.
4. Core page-level behaviours, including navigation, tab switching, dialogs, buttons, filtering, status panels, and result views, behave correctly in a rendered browser environment.
5. Shared service layers and UI helper components remain stable enough to provide regression protection for future changes.

5.2 Methodology

The test suite is organised into three levels: unit tests, browser integration tests, and end-to-end tests. These levels follow the layered architecture described in Chapter 4. Isolated service and transformation logic is tested at unit level. Rendered browser behaviour is tested through integration tests. Route-level user journeys are tested through end-to-end tests.

Unit tests. Unit tests target internal logic without requiring a running browser. They cover service-layer contracts, form model helpers, dictionary and list widget selection, Pydantic form behaviour, configuration service behaviour, plugin-form metadata extraction, topology transformation, topology renderer navigation, app shell behaviour, shared components, and web observer logic. These tests are fast and deterministic, so they are useful for early regression detection during development.

Browser integration tests. Browser integration tests validate rendered UI behaviour through NiceGUI and Selenium in a real browser session. They verify that pages load correctly, visible UI elements are present, dialogs open, tabs switch, search and filter controls behave correctly, and mocked backend data is displayed as expected. These tests provide stronger confidence than unit tests because they exercise the browser-rendered interface rather than only isolated functions.

End-to-end tests. End-to-end tests validate higher-level route and navigation workflows across the web application. They check whether the main pages can be reached and whether basic cross-page user journeys execute successfully. These tests are broader than integration tests, but they intentionally avoid full experiment execution because experiment runs may require Docker, runtime dependencies, and longer execution time.

The following are some examples of tests run on the main functional areas of the interface:

1. Configuration tests check whether form helpers classify complex fields, extract sections, map metadata, handle optional and enum values, produce serialization-safe output, and some more enhanced configuration service tests.
2. Topology tests verify that YAML configurations are transformed into graph structures correctly. They check service aggregation, deduplication, scaling thresholds, default handling, network metadata extraction, generated ECharts options, click-to-navigation payloads, test-index resolution, and URL encoding.

A complete mapping of automated test coverage is provided in Appendix A.

5.3 Results and analysis

Table 5.1 summarises the collected automated test inventory by test level. The webapp-facing test suite contains 217 collected unit tests, 79 collected browser integration tests, and 8 collected end-to-end tests.

Test level	**Scope**	**Collected tests**
Unit tests	Isolated logic in services, form models, rendering helpers, plugin metadata extraction, observer behaviour, topology transformation, and navigation payload generation.	217
Browser integration tests	Rendered UI behaviour through NiceGUI and Selenium, including page rendering, dialogs, tabs, buttons, filtering, status panels, and mocked service output.	79
End-to-end tests	Higher-level route loading and cross-page navigation workflows across the main web application pages.	8

Table 5.1: Collected automated test inventory for the PANTHER Web Interface.

The browser integration suite and end-to-end suite complete successfully in the reported environment. A broader unit-test run includes one known failing expectation connected to the original PANTHER application shell rather than to the webapp functionality developed in this master thesis. Since that failure concerns pre-existing application-level behaviour outside the direct scope of the web interface contribution, it is noted here but not treated as evidence against the correctness of the implemented webapp features.

Unit tests validate the deterministic logic such as configuration handling, form helpers, plugin metadata extraction, topology transformation, and observer behaviour. Browser integration tests add confidence that these behaviours remain visible and usable in the rendered interface. End-to-end tests provide a smaller but useful check that the main routes and cross-page navigation paths work together.

The testing evidence supports the functional correctness of the main web interface behaviours. Unit tests provide confidence in deterministic service and transformation logic. Browser integration tests confirm rendered UI behaviour in a real browser session. End-to-end tests validate route-level user journeys.

Chapter 6

Validation

This chapter validates whether the PANTHER Web Interface improves selected workflows compared with the original CLI/YAML workflow. The validation focuses on workflows where the interface is designed to reduce manual effort: creating configuration files, reviewing configuration files, understanding topology, and finding correct inputs.

We use an automated benchmark that compares the CLI/YAML workflow and the web interface on the same sample configuration files. The benchmark reports average effort values and percentage reductions across the samples. This makes the validation less dependent on a single example and gives a more stable basis for comparing the two workflows.

The evaluation remains intentionally scoped. It does not claim that the web interface is better than the CLI for every PANTHER task. The purpose is simpler: to show whether the web interface makes selected configuration-heavy and inspection-heavy workflows more efficient.

6.1 Method

The validation uses a scripted benchmark. The script reads existing PANTHER YAML configuration files and computes paired CLI/UI effort values for each workflow. A paired comparison means that the CLI and UI values are calculated from the same sample configuration wherever possible. This avoids comparing an easy UI example against a harder CLI example.

The benchmark uses eight different YAML files for the comparison.

For each workflow, the benchmark reports:

1. the number of sample files;
2. the average CLI effort;
3. the average UI effort;
4. the mean percentage reduction.

For count-based metrics, the mean reduction is computed as:

$$Meanreduction = \frac{CLImean - UImean}{CLImean} \times 100.$$

The values are not timings from a user study. They are structural effort measurements computed from files and service-level calculations. In other words, the benchmark does not measure how many seconds a user needs. It measures how much visible work each workflow requires according to explicit counting rules.

6.1.1 Compared workflows

The benchmark compares four workflows:

1. **Creating configuration files:** compares the amount of work needed to express the same experiment configuration in YAML or through UI form actions.

   For the CLI workflow, the benchmark serialises each sample configuration and counts the number of non-empty YAML lines. This represents the amount of YAML a user would need to write or review manually. For the UI workflow, the same parsed configuration is traversed recursively. Scalar values count as one form edit or select action. List items count as add actions plus their child edits. Service dictionaries count as one add/name action per service plus nested field edits.

   A YAML line and a form edit are not identical units. A YAML line can also require indentation, key naming, and correct placement. A form edit usually means changing a labelled field or selecting an option. The comparison should therefore be read as an effective approximation, not as a precise measure.

2. **Reviewing configuration files:** compares how much information a user must inspect to understand an existing configuration.

   For the CLI workflow, the benchmark uses the number of non-empty YAML lines because a user reviewing a configuration must scan the raw YAML structure. For the UI workflow, the benchmark counts visible structured objects such as top-level sections, test panels, service entries, and execution-environment entries. This captures the difference between reading a full YAML file and reviewing the same configuration through grouped interface sections.

3. **Understanding topology:** compares the effort needed to identify services, connections, and network-related structure.

For the CLI workflow, the benchmark counts graph-relevant YAML fields that a user must inspect to infer topology. These include test boundaries, services, protocol roles, protocol targets, network environment entries, and execution environments. For the UI workflow, the same configuration is passed through `TopologyService.parse_config_to_graph()`. The UI value counts the graph facts surfaced by the topology view, such as tests, services, connections, and network-environment types.

4. **Finding correct inputs:** compares the work needed to find valid configuration choices such as implementations, protocols, roles, and environment types.

   For the CLI workflow, the benchmark counts repeated option assignments in YAML. These include implementation names and types, protocol names, protocol versions, protocol roles, network environment types, and execution-environment types. For the UI workflow, the benchmark counts unique option values in the same categories. This reflects the fact that the web interface can expose repeated choices through select controls or plugin metadata rather than making the user rediscover them in different YAML locations.

6.2 Results and analysis

Table 6.1 presents the aggregate benchmark results. The web interface has a lower average effort value than the CLI/YAML workflow in all four measured workflows.

Workflow	**n**	**CLI mean**	**UI mean**	**Mean reduction**
Creating configuration files	8	139.12	120.00	13.7%
Reviewing configuration files	8	139.12	13.75	90.1%
Understanding topology	6	29.67	9.00	69.7%
Finding correct inputs	8	24.38	10.25	57.9%

Table 6.1: Aggregate UI-versus-CLI workflow benchmark results.

6.2.1 Claim 1: the UI makes existing configurations easier to review

The strongest result is for reviewing configuration files. The CLI/YAML workflow has an average of 139.12 non-empty YAML lines to scan, while the UI workflow

has an average of 13.75 structured objects to inspect. This gives a mean reduction of 90.1%.

This supports the claim that the interface makes existing configurations easier to understand. In the CLI workflow, the user reviews one raw YAML structure. In the web interface, the same information is grouped into sections, test panels, service entries, and execution-environment entries. The improvement comes from changing the review task from line-by-line scanning to structured inspection.

6.2.2 Claim 2: the topology view reduces manual reconstruction

Understanding topology also shows a strong reduction. The CLI/YAML workflow has an average of 29.67 graph-relevant YAML fields to inspect, while the UI exposes an average of 9 graph facts through the topology view. This gives a mean reduction of 69.7%.

This supports the claim that the Topology Editor reduces manual reconstruction. In the CLI/YAML workflow, topology is implicit. The user has to inspect services, roles, targets, network environments, and execution environments, then assemble the relationship mentally. The UI computes graph facts from the same configuration data and exposes them visually. The result is fewer separate facts for the user to locate manually.

6.2.3 Claim 3: the UI reduces repeated option lookup

Finding correct inputs shows a mean reduction of 57.9%. The CLI/YAML workflow has an average of 24.38 repeated option assignments, while the UI workflow has an average of 10.25 unique option choices.

This supports the claim that the Plugin Browser and guided form controls reduce repeated lookup. A user still needs to choose the correct implementation, protocol, role, or environment. The difference is that the UI can expose recurring choices through select controls and plugin metadata, instead of requiring the user to rediscover repeated values across YAML files or external references.

6.2.4 Claim 4: the UI reduces part of the configuration creation burden

Creating configuration files shows the smallest reduction, at 13.7%. The CLI/YAML workflow has an average of 139.12 non-empty YAML lines, while the UI workflow has an average of 120 form edits or add/select actions.

This supports a narrower claim than the other results. The UI does not remove the work of defining an experiment. The user still has to choose services, protocols, environments, parameters, and execution settings. The measured improvement comes from reducing some surrounding structural work, such as manual nesting, repeated syntax handling, and searching for valid options. This benefit may be more visible for less experienced users, who would otherwise need to spend more time checking YAML structure or looking up acceptable values while editing the file directly.

Taken together, the benchmark results support the main usability claim of this master thesis. The interface helps most when the task involves inspecting, understanding, or discovering information, but will still be beneficial for tasks involving creation.

6.3 Threats to validity

Structural metrics instead of user timing. The benchmark measures structural effort metrics, not completion time. It does not replace a user study that measures time, errors, confidence, and subjective workload. The reason for this was the inability to find non-expert users of PANTHER for a fair comparison. The current benchmark is still useful because it is reproducible and based on explicit calculation rules.

Workflow scope. The benchmark evaluates workflows where the UI is expected to help: creating configuration files, reviewing configuration files, understanding topology, and finding correct inputs. It does not evaluate tasks where the CLI may be more suitable, as the purpose was to justify the original motivation for the thesis.

Repository-specific samples. The benchmark uses existing PANTHER configuration files. This improves relevance for the project, but it may not cover every future protocol, plugin, or environment type. New plugins may introduce configuration structures that change the authoring or review effort.

Chapter 7

Conclusion

7.1 Main findings

This master thesis presents the design, implementation, testing, and validation of a graphical web interface for PANTHER. The work addresses the difficulty of managing protocol experiments through complex YAML files and command-line workflows by introducing a structured, browser-based interface for configuration, topology inspection, experiment monitoring, result viewing, and plugin discovery.

The testing process provides confidence that users can configure experiments, receive validation feedback, inspect generated topology information, navigate the browser interface, and move through the main application routes without breaking the intended workflow. This supports the thesis objective of building a usable web layer on top of PANTHER rather than a separate or inconsistent tool.

The validation results show that the interface also supports the thesis objective of reducing friction in selected workflows. Strongest measured advantages appear in reviewing configuration files, understanding topology, and finding correct inputs, which directly correspond to the goals of improving configuration inspection, visual comprehension, and plugin or option discovery. This supports the intended role of the interface: it does not remove experiment design work, but it reduces part of the manual YAML scanning, topology reconstruction, and repeated lookup around it.

7.2 Limitations

The current evaluation has some limitations. The validation is based on an automated structural benchmark rather than a full external user study. This makes the comparison reproducible, but it does not capture user satisfaction, learning curve differences, or variation between novice and expert users.

Some parts of the system also depend on runtime infrastructure such as Docker, browser drivers, and host environment setup. These dependencies can affect reproducibility across machines, especially for full experiment execution and browser-based testing.

Finally, the current prototype focuses on the main web-facing experiment workflow. More advanced research workflows, collaborative use, richer analytics, and attack-oriented protocol vulnerability discovery remain outside the implemented scope.

7.3 Future work

Several directions could extend the current prototype and strengthen its usefulness as a complete experiment management environment.

Attack-oriented protocol vulnerability workflows. PANTHER-related work on formally discovering and reproducing network protocol vulnerabilities introduces Network Attack-centric Compositional Testing, a method for discovering vulnerabilities and reproducing attack scenarios through attacker models, formal specification mutations, and generated test cases [3]. This thesis does not implement such attack-oriented workflows. A future version of the web interface could expose this functionality through guided attacker-model configuration, vulnerability-reproduction templates, and result views focused on attack traces and reproduced failures.

Editable topology graph. The current Topology Editor focuses on visual inspection of the configuration. A future version could make the graph directly editable, allowing users to add, remove, or modify nodes and edges in order to change the underlying experiment configuration. For example, adding a service node could create a corresponding service entry, while adding an edge could define or modify a protocol target relationship.

Configuration templates and guided experiment creation. Future versions could provide a template system for common experiment patterns, such as single client-server tests, multi-client scenarios, Shadow-based experiments, or Ivy-based protocol tests. A guided wizard could help users select protocols, implementations, testers, and environments step by step, generating an initial valid configuration that can then be refined manually.

Multi-user and collaborative use. A future version could support authentication, user roles, shared projects, and collaborative experiment editing. This would be particularly useful in educational settings where instructors prepare experiments and students run or modify them, or in research teams where several users need access to the same experiment history.

Bibliography

[1] Temitope Awodiji. Interactive dashboard design for manager, data analyst and data scientist perspective. pages 175–185, 11 2021.

[2] Panel Chairs, Marc Resnick, and Jennifer Bandos. Best practices in search user interface design. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 46, 09 2002.

[3] Christophe Crochet, John Aoga, and Axel Legay. Formally discovering and reproducing network protocols vulnerabilities. In Leonardo Horn Iwaya, Liina Kamm, Leonardo Martucci, and Tobias Pulls, editors, *Secure IT Systems*, pages 424–443, Cham, 2025. Springer Nature Switzerland.

[4] Christophe Crochet, John Aoga, and Axel Legay. Panther: Pluginizable testing environment for network protocols. *Science of Computer Programming*, 248:103389, 09 2025.

[5] Marijn Haverbeke. *CodeMirror Reference Manual*, 2026.

[6] Feiyan Jiang and Guoyong Yang. Research on color design of human-computer interaction interface of mobile app. In *Proceedings of the International Conference on Image Processing, Machine Learning and Pattern Recognition*, IPMLP '24, page 421–425, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Atta ur Rehman Khan, Sardar Bilal, and Mazliza Othman. A performance comparison of open source network simulators for wireless networks. 11 2012.

[8] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, page 152–158, New York, NY, USA, 1994. Association for Computing Machinery.

[9] Donald A. Norman. *The design of everyday things*. Basic Books, [New York], 2002.

[10] Sebastián Ramírez. *FastAPI Framework Documentation*, 2026.

[11] Tom Rousseaux, Christophe Crochet, John Aoga, and Axel Legay. Network simulator-centric compositional testing. In Valentina Castiglioni and Adrian Francalanza, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 177–196, Cham, 2024. Springer Nature Switzerland.

[12] Falko Schindler and Rodja Trappe. Nicegui: Web-based user interfaces with python. the nice way., 2026.

[13] The Apache Software Foundation. *Apache ECharts Documentation*, 2026.

[14] Jan Vykopal, Radek Ošlejšek, Pavel Celeda, Martin Vizváry, and Daniel Tovarňák. Kypo cyber range: Design and use cases. pages 310–321, 01 2017.

[15] Richard Weiss, Jens Mache, and Michael Locasto. Edurange: hands-on cybersecurity exercises in the cloud. *J. Comput. Sci. Coll.*, 30(1):178–180, October 2014.

[16] Muhammad Mudassar Yamin, Basel Katt, and Vasileios Gkioulos. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers Security*, 88:101636, 2020.

[17] Zauberzeug GmbH. *NiceGUI: Let Python Become Your Frontend*, 2026.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**