POLITECNICO
DI TORINO

this: POLITECNICO DI TORINO

Master's Degree course in Communications Engineering

Master's Degree Thesis

Automated Verification and Evaluation of Network Routing Configurations

Supervisor
Prof. PAOLO GIACCONE

Candidate
Yuqi JIANG

Academic Year 2025-2026

Acknowledgements

The completion of this thesis would not have been possible without the guidance and support I received from my teachers and classmates during my master's studies.

First and foremost, I would like to express my sincere gratitude to my supervisor. From the early discussions on topic selection to the refinement of research questions and the revision of the manuscript, their continuous guidance played a crucial role in shaping this work. In particular, the process of abstracting the routing grading problem into a network behavior verification framework was developed through multiple rounds of discussion and reflection. These exchanges not only improved the quality of this thesis but also strengthened my ability to define and analyze research problems in a systematic way.

I am also grateful for the coursework completed during my master's program. Knowledge of computer networking, routing mechanisms, and system design provided the theoretical foundation upon which this research was built. Many concepts that once seemed abstract became clearer during the implementation process, allowing theory and practice to complement each other.

I would like to thank my classmates who participated in the course experiments and testing process. Their submissions and feedback provided practical data that enabled the system to be evaluated and refined in a realistic teaching environment.

Finally, I would like to thank my family for their understanding and support throughout my studies. Their encouragement allowed me to focus on my research and complete this thesis.

Due to my limited personal abilities, there are still shortcomings in thesis. I sincerely request your criticism and correction.

Abstract

Misconfigurations in routing policies can cause reachability failures and forwarding anomalies, making systematic verification of routing behavior an important problem. In educational and training environments, the same challenge arises at scale: many routing configuration submissions must be assessed efficiently, consistently, and fairly. However, such tasks are often graded manually or through direct comparison of routing table entries. These approaches do not scale well and can be error-prone. More importantly, structural comparison of configurations cannot reliably reflect intended network behavior, since different routing implementations may produce identical forwarding outcomes, while seemingly similar configurations may still lead to forwarding errors such as unreachable subnets or routing loops.

This thesis presents an automated framework for verifying and evaluating network routing configurations by reformulating assessment as the verification of constrained network properties. Instead of comparing configuration syntax, the framework evaluates behavioral correctness through data-plane analysis. For each task, the expected behavior is scoped by a question-defined destination prefix set, and verification is performed against the forwarding behavior computed from the submitted solution. Students' routing entries are automatically transformed into device-specific configuration files and combined with topology metadata to construct complete network snapshots. These snapshots are then analyzed using Batfish, which computes data-plane forwarding behavior and enables structured verification queries over the resulting forwarding model.

Two core properties are evaluated: reachability and loop-freedom. The reachability index is defined as the ratio of successfully delivered flows to the total number of evaluated flows, providing a quantitative measure of how well the configuration achieves the required connectivity. Loop-freedom is represented as a binary indicator reflecting the absence of forwarding loops. A deterministic scoring function maps these verification outcomes into final scores, ensuring repeatability and reducing subjective variation in grading.

As a proof-of-concept deployment, the framework is integrated into a Moodle-based assessment workflow in which students submit solutions and instructors manage assignments and import automatically generated grades. The system supports automated snapshot generation, scalable evaluation across many students, and CSV export compatible with Moodle grade import. Experimental evaluation shows that the proposed approach achieves grading consistency comparable to manual verification while significantly improving efficiency and scalability.

By integrating behavior-based network verification into educational assessment, this thesis introduces an automated evaluation pipeline that improves fairness, repeatability, and robustness in routing configuration grading. Beyond the educational setting, the same verification pipeline can be adapted to broader configuration validation and regression testing scenarios, and can provide structured signals that support future data-driven or AI-assisted configuration analysis.

Contents

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Research Objectives and Contributions . . . 4
- 1.4 Organization of the Thesis . . . 4

2 Background on Network Verification and Batfish 5
- 2.1 Static Routing Essentials . . . 5
  - 2.1.1 Network Elements and Topology . . . 5
  - 2.1.2 Routing Tables and Forwarding Behavior . . . 7
  - 2.1.3 Forwarding Semantics . . . 7
- 2.2 Fundamentals of Network Verification . . . 8
  - 2.2.1 Motivation for Network Verification . . . 8
  - 2.2.2 Control Plane vs. Data Plane Verification . . . 9
  - 2.2.3 Formalization of Network Properties . . . 10
- 2.3 Overview of Batfish . . . 10
  - 2.3.1 Snapshot-Based Modeling . . . 12
  - 2.3.2 Graph-Based Internal Representation . . . 12
  - 2.3.3 Query-Based Analysis Framework . . . 12
- 2.4 Properties Verified by Batfish . . . 13
  - 2.4.1 Reachability Property . . . 13
  - 2.4.2 Loop Property . . . 14
  - 2.4.3 Final Grading Rule . . . 14

3 Proposed Framework 16
- 3.1 Problem Formalization and Assumptions . . . 16
- 3.2 Design Goals . . . 17
  - 3.2.1 Full Automation . . . 17
  - 3.2.2 Behavior-based Verification . . . 17
  - 3.2.3 Extensibility and Integration . . . 18
- 3.3 System Architecture and Data Flow . . . 18
- 3.4 End-to-End Workflow . . . 20
- 3.5 Scoring Indicators and Grading Policy . . . 20
  - 3.5.1 Reachability Index . . . 20
  - 3.5.2 Loop Index . . . 21

4 System Implementation 22
4.1 Automatic Question Generation . . . 22
4.1.1 Topology Template Design . . . 22
4.1.2 IP/Subnet Sampling Strategy and Constraints . . . 29
4.1.3 Moodle XML Construction and Categories . . . 33
4.2 Student Answer Collection and Parsing . . . 40
4.2.1 Exporting Student Responses from Moodle . . . 40
4.2.2 Structure of the Exported JSON File . . . 42
4.2.3 Question ID Extraction Mechanism . . . 43
4.2.4 Automatic Matching of Question . . . 43
4.2.5 Parsing Student Routing Table . . . 44
4.2.6 Internal Data Representation . . . 45
4.2.7 Role of the Parsing Module . . . 45
4.3 Configuration and Topology Synthesis . . . 45
4.3.1 Vendor-style Configuration Generation . . . 46
4.3.2 Topology.json Generation Strategy . . . 48
4.4 Batfish Verification Implementation . . . 49
4.4.1 Container Setup and Snapshot Build . . . 50
4.4.2 Reachability and Loop Queries . . . 51
4.5 Result Aggregation and Moodle Grade Import . . . 55
4.5.1 Batch Processing Across Students and Questions . . . 55
4.5.2 Question Grade Representation . . . 57
4.5.3 Total Score Computation . . . 57
4.5.4 CSV Generation for Moodle Import . . . 58

5 Experimental Validation 60
5.1 Experimental Setup . . . 60
5.1.1 Environment Configuration . . . 60
5.1.2 Dataset Design . . . 60
5.2 Evaluation Metrics . . . 61
5.2.1 Reachability Index . . . 61
5.2.2 Loop Index . . . 61
5.2.3 Grading Consistency . . . 61
5.2.4 Execution Time . . . 61
5.3 Results and Analysis . . . 63

6 Conclusion 64
6.1 Limitations . . . 65
6.2 Future Work . . . 65
6.3 Contributions . . . 68

Bibliography 69

Chapter 1

Introduction

1.1 Background and Motivation

Computer networking education has long emphasized hands-on configuration problems, particularly in topics such as static routing and fundamental forwarding behavior. In such problems, students are typically given a network topology and a set of IP address assignments, and are required to construct routing tables that achieve end-to-end connectivity across multiple subnets. In programmable and software-defined networks, configuration correctness plays an even more critical role in determining system behavior [1].

Despite the pedagogical importance of these problems, the grading process remains a persistent challenge. Traditional evaluation methods rely heavily on manual inspection, where instructors compare students' routing entries against predefined reference solutions [2]. While feasible in small classes, this approach becomes increasingly problematic in large scale or online learning environments.

Several limitations arise in manual grading:

- **Efficiency**: The grading workload grows linearly with the number of students, while each submission may involve multiple devices and routing entries.
- **Consistency**: Human evaluation may introduce subjective bias and inconsistencies across different graders or grading sessions.
- **Scalability**: Maintaining multiple acceptable reference solutions becomes difficult as topology variations increase.
- **Semantic mismatch**: Structural comparison of routing entries does not necessarily reflect the correctness of the resulting forwarding behavior [3].

The last issue is particularly critical. In routing configuration problems, multiple syntactically different routing tables may produce functionally equivalent forwarding behavior. For example, students may use default routes, more specific prefixes, or alternative yet valid next-hop choices. A grading system based purely on textual matching risks misjudging behaviorally correct solutions. Therefore, the core question is not:

> "Do the student's routing entries match the reference configuration?"

but rather:

> "Does the student's configuration produce the intended network behavior?"

This observation naturally connects the grading problem to the broader domain of network verification. Modern verification tools, such as Batfish, can analyze router configurations offline, construct control-plane and data-plane models, and evaluate whether specific network properties hold.

Instead of comparing configuration structures, this thesis proposes to transform the grading task into a constrained network property verification problem. Students' routing configurations are converted into vendor-style device configurations, assembled into a complete snapshot, and analyzed using Batfish queries. The verification results are then mapped into grading metrics.

This approach introduces formal verification concepts into educational assessment, enabling behavior-driven grading that is fair, repeatable, scalable, and aligned with pedagogical objectives.

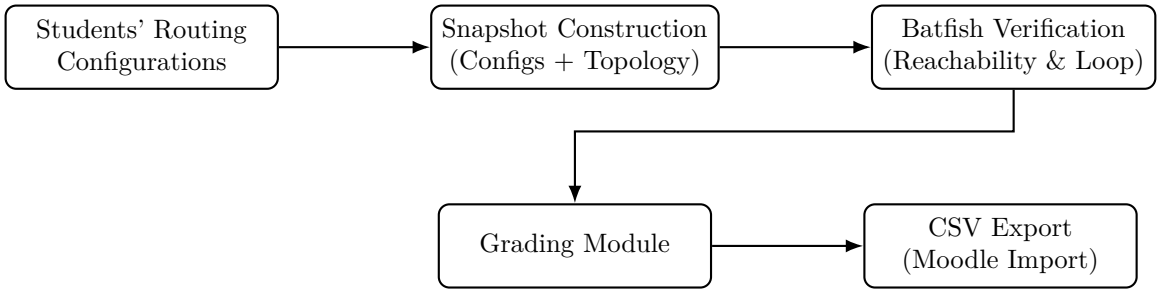


Figure 1.1. Overall architecture of the proposed Batfish-based automated grading framework

As shown in the Figure 1.1, student routing configurations are transformed into verification snapshots, analyzed by Batfish, evaluated against predefined network properties, and finally mapped into grading results.

1.2 Problem Statement

This thesis addresses the automated verification and evaluation of routing-configuration tasks. Given a network topology and an address plan, the goal is to determine whether a student's routing submission induces forwarding behavior that satisfies the requirements of the task. Unlike approaches that grade by comparing configuration text or routing-table entries, this thesis formulates assessment as a *data-plane property verification* problem: a submission is considered correct if the resulting forwarding behavior satisfies the required network properties, independently of configuration style or vendor-specific implementation details.

To formalize the grading problem, each task is described by the following inputs: (i) a network topology T, (ii) a set of IP address assignments A, and (iii) a student's routing configuration R. Based on these inputs, the system constructs a complete *network snapshot:*

$$S = f(T, A, R), \tag{1.1}$$

where $f(\cdot)$ denotes the snapshot synthesis procedure that combines the submitted routing configuration with the task's topology and addressing information to produce the device configuration files and the topology description required by the verification engine. In this framework, a snapshot S represents a concrete network instance on which data-plane forwarding behavior can be computed. By analyzing S, the system can determine which nodes and links are traversed by a flow from a given source to a destination prefix, whether the destination is reachable, and whether forwarding loops occur along the computed forwarding paths.

Once the snapshot S is synthesized, the evaluation problem reduces to verifying whether S satisfies a predefined set of data-plane properties. Each task specifies a question-defined destination prefix set, which scopes the verification to the destinations relevant to the question. The framework generates tested flows whose destinations belong to this destination prefix set and evaluates the submission using two core properties: *reachability* and *loop detection*. Together, they characterize routing-quality from both a functional-correctness perspective and a safety/robustness perspective.

Reachability: Reachability concerns whether all destination prefixes required by the task can be reached from the relevant sources. If a destination prefix in the question-defined destination prefix set is unreachable, the submission violates the basic connectivity requirement of the task. Such failures typically arise from missing routes, incorrect next-hop selection, wrong prefix lengths, or inconsistent routing decisions across devices. Reachability-based assessment avoids the limitations of purely syntactic grading: different configurations may yield identical forwarding behavior, while superficially similar configurations may lead to very different outcomes. Since the objective of routing is to realize correct forwarding, data-plane reachability provides a direct and implementation-independent criterion. Moreover, reachability results can be made interpretable by identifying which required prefixes are not reachable, providing actionable feedback for debugging.

Loop detection: Loop detection concerns whether the tested flows exhibit *forwarding loops*, i.e., packets that circulate indefinitely due to inconsistent forwarding decisions. Loops are a particularly severe class of routing errors: even when some destinations appear reachable, forwarding loops may still occur for certain traffic classes and prevent packets from being delivered in practice. In addition, loops can lead to persistent congestion and resource exhaustion by repeatedly consuming bandwidth and forwarding capacity. Therefore, loop detection complements reachability by enforcing a safety constraint on data-plane behavior and aligns the assessment criteria with operational expectations of stable forwarding.

Importance: Taken together, reachability and loop detection cover the most important behavioral failure modes in routing configuration tasks: lack of connectivity and unsafe forwarding behavior (e.g., loops caused by inconsistent routing decisions). Both properties are defined at the data-plane level and do not depend on a specific configuration template. This makes the assessment robust to alternative correct solutions and consistent with the

behavior-driven verification principle adopted in this thesis.

This formulation ensures that full connectivity is a prerequisite for correctness, while loop detection serves as an additional safety constraint.

1.3 Research Objectives and Contributions

This thesis aims to design and implement a verification driven automated grading framework for routing configuration problems. The main contributions are:

1. A behavior-oriented grading model that transforms configuration assessment into data-plane property verification.
2. An automated snapshot construction mechanism that converts student routing entries into vendor-style configurations compatible with **Batfish**.
3. A formalized grading strategy based on reachability and loop-freedom properties.
4. An integrated grading pipeline that exports results in Moodle-compatible CSV format.
5. Experimental evaluation demonstrating grading consistency, efficiency, and scalability.

By bridging educational assessment with industrial-grade network verification tools, this work provides a principled and scalable solution to automated grading in network configuration courses.

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2 presents background knowledge on network verification and Batfish, including formal definitions of reachability and loop properties.

Chapter 3 describes the overall system architecture and design principles.

Chapter 4 details the implementation of snapshot construction, verification queries, and scoring mechanisms.

Chapter 5 evaluates the system through experimental analysis, including grading accuracy, efficiency, and scalability.

Chapter 6 concludes the thesis and discusses potential future extensions.

Chapter 2

Background on Network Verification and Batfish

2.1 Static Routing Essentials

Static routing is one of the most fundamental forwarding mechanisms in IP networks and is widely used in introductory networking education. Unlike dynamic routing protocols such as OSPF or BGP, which rely on route advertisement and convergence mechanisms, static routing is entirely manually configured by network administrators. In an educational setting, students are typically required to construct routing tables by specifying destination network prefixes, gateway and corresponding interface.

Although students submit routing entries rather than full vendor configurations, packet forwarding still follows standard IP forwarding semantics. Therefore, before introducing network verification and Batfish-based analysis, it is necessary to establish a foundation covering network elements, routing tables and forwarding behavior under static routing.

2.1.1 Network Elements and Topology

IP network can be abstracted using several fundamental components: hosts, routers, interfaces, links and subnets. Hosts serve as communication endpoints, typically located at the network edge. Routers are intermediate devices responsible for forwarding packets across different subnets. Each device connects to one or more links via network interfaces, and each interface is assigned an IP address and subnet mask, thereby belonging to a specific subnet.

A subnet represents a range of IP addresses sharing the same network prefix and broadcast domain. In this system, subnets are expressed using CIDR notation (e.g., `172.26.204.0/24` and `192.168.61.108/30`). The network topology describes the physical connectivity between devices, including which interfaces are connected through which links.

It is important to distinguish between physical topology and forwarding behavior. Physical topology describes the connectivity graph of devices, while forwarding behavior depends entirely on routing table configurations. Even if a physical topology contains no cycles, incorrect routing entries may still produce forwarding loops in the data plane. Conversely, a topology with physical loops does not necessarily imply forwarding loops if routing tables are correctly configured.

In the context of routing configuration questions, each problem instance defines a

specific topology and a set of IP assignments for device interfaces. Students are required to construct static routing tables such that the network satisfies predefined correctness criteria, typically including full connectivity and the absence of forwarding anomalies. The system introduced in this framework reconstructs the complete network model using problem metadata, including IP assignments and topology descriptions, in order to transform student routing entries into a verifiable network representation.

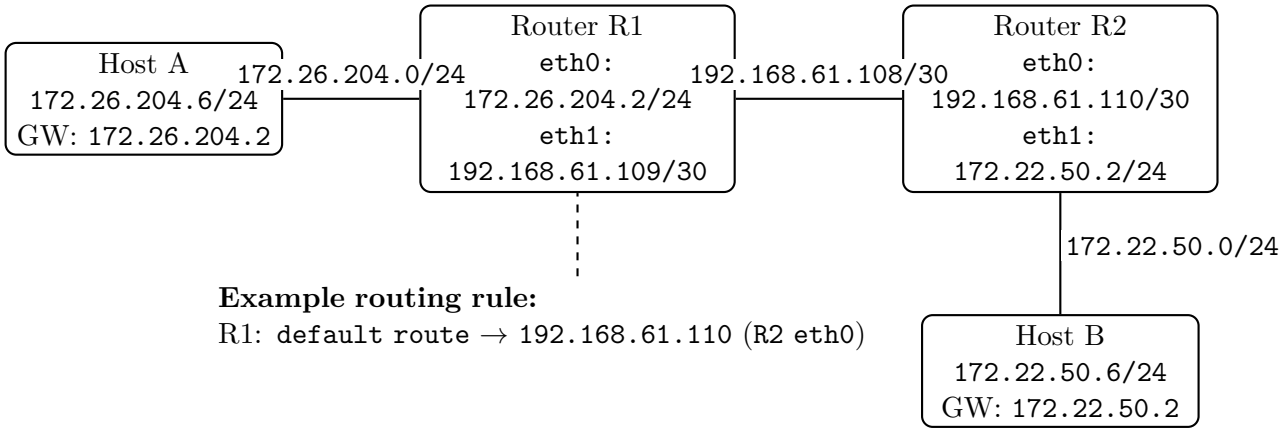


Figure 2.1. Illustrative Static Routing Topology

Figure 2.1 presents a minimal static-routing topology that will be used as a running example to clarify how addressing, routing entries, and forwarding behavior relate to each other. The topology contains two host-facing LAN segments and one router-to-router point-to-point link. Host A resides in subnet `172.26.204.0/24` with address `172.26.204.6/24`. It sends all off-subnet traffic to its default gateway, which is Router R1's interface `eth0` (`172.26.204.2/24`). On the right side, Host B resides in subnet `172.22.50.0/24` with address `172.22.50.6/24`, and uses Router R2's interface `eth1` (`172.22.50.2/24`) as its default gateway. The two routers are connected by a point-to-point link `192.168.61.108/30`, where R1's `eth1` is assigned `192.168.61.109/30` and R2's `eth0` is assigned `192.168.61.110/30`. Listing both link prefixes and interface addresses makes the figure self-contained: a reader can infer which interfaces share a Layer-3 subnet and which next hops are valid without consulting additional text.

Forwarding decisions in this topology follow the longest prefix match (LPM) rule. At each router, the destination IP address of a packet is compared against available routing entries, and the entry with the most specific matching prefix is selected. Consider traffic from Host A to Host B (`172.22.50.6`). When the packet reaches R1, a correct routing configuration should cause R1 to forward it over the inter-router link towards R2, because `172.22.50.0/24` is reachable via R2. If R1 contains a specific route for `172.22.50.0/24`, that entry will be preferred over less specific alternatives. If no specific entry matches a destination, R1 may rely on a default route (i.e., a catch-all route) and forward the packet to a configured next hop. In the example shown in the figure, the next hop `192.168.61.110` corresponds to R2's `eth0` on the `192.168.61.108/30` link,

which is a valid neighbor address from R1's perspective. This example illustrates why routing-task evaluation should be based on data-plane behavior: different routing tables can lead to the same forwarding outcome, while small mistakes (e.g., missing a specific prefix route or selecting an incorrect next hop) can result in unreachable destinations or pathological forwarding behaviors that are not apparent from configuration syntax alone. In later sections, the proposed framework reconstructs such topology and addressing information from problem metadata, synthesizes a complete snapshot, and verifies the resulting forwarding behavior automatically.

2.1.2 Routing Tables and Forwarding Behavior

The routing table is the core data structure that determines how a router forwards packets. Each routing entry typically contains three elements: Network Prefix, Gateway and Interface. For directly connected networks, routers automatically possess connected routes corresponding to interface prefixes. For remote networks, static routes must be explicitly configured to indicate the gateway.

Two common types of static routes appear in educational scenarios: specific prefix routes and the default route. Specific prefix routes define forwarding behavior for particular destination subnets. The default route, represented as `0.0.0.0/0`, matches all destination addresses not covered by more specific entries. While the default route simplifies configuration, especially for edge devices, it may also conceal misconfigurations and introduce unintended forwarding behavior.

Routers implement forwarding decisions based on the Longest Prefix Match (LPM) principle. Given a destination IP address, the router selects the routing entry with the longest matching prefix length. This ensures that more specific routes override more general ones. For example, even if a default route exists, traffic destined for `172.26.204.0/24` will match the `/24` entry instead of `0.0.0.0/0`.

In routing configuration questions, students frequently make mistakes such as specifying incorrect gateway, misconfiguring subnet masks or omitting required routes. Some misconfigurations do not immediately cause complete loss of connectivity. In certain cases, the network may remain reachable due to redundant paths or fallback default routes, even though the routing logic is incorrect. Therefore, reachability alone is insufficient as a sole correctness criterion.

2.1.3 Forwarding Semantics

To formally analyze routing correctness, it is necessary to define forwarding semantics at the data-plane level. When a packet arrives at a router, the router examines the destination IP address and performs a longest prefix match against its routing table. The selected entry determines the gateway or outgoing interface. If the gateway is specified as an IP address, it must be reachable via a directly connected subnet; otherwise, the route is operationally invalid. If no matching route exists and no default route is configured, the packet is dropped.

The forwarding path of a packet can be viewed as a sequence of routers. Starting from a source host, the packet is sent to its default gateway and subsequently forwarded

hop by hop until it reaches the router directly connected to the destination subnet. If such a path exists, the destination prefix is considered reachable from the source.

However, reachability does not guarantee correctness. A particularly critical forwarding is the forwarding loop. A forwarding loop occurs when packets for a given destination are repeatedly forwarded among a set of routers without ever reaching the intended subnet. Forwarding loops are a data-plane phenomenon and may arise even in physically acyclic topologies. For example, two routers may incorrectly configure static routes pointing to each other for a certain prefix or through improperly configured default routes, thereby forming a cyclic forwarding dependency.

Forwarding loops represent severe configuration errors. They prevent successful packet delivery and reflect fundamental misunderstandings in gateway selection or routing direction. Consequently, a comprehensive grading framework must evaluate both reachability and loop-freedom. By combining these two properties, it becomes possible to distinguish fully correct solutions from partially correct or logically flawed ones, even when basic connectivity appears intact.

2.2 Fundamentals of Network Verification

Network verification studies whether a network, given its device configurations and topology, satisfies a set of intended properties [4, 5] (e.g., reachability and absence of loops). Unlike traditional operational debugging (e.g., `ping`/`traceroute` and manual inspection), verification aims to provide systematic and reproducible guarantees, *prior* to deployment or in a controlled offline environment [6]. This is especially important in modern networks where the configuration space grows rapidly with the number of devices and interacting features. Seemingly right local changes may trigger subtle global failures (e.g., blackholes, persistent loops or unintended detours) that are difficult to anticipate through human reasoning alone.

In the educational setting considered in this system, students submit routing tables rather than full vendor configuration files. The goal of automated grading is to evaluate whether the submitted routing tables induce the correct forwarding behavior under the problem specification. Therefore, the verification objective is not to exhaustively validate all operational properties of a production network, but to focus on a small set of grading-aligned properties that are both interpretable and automatically checkable, most notably (i) reachability to a designated set of destination prefixes, (ii) the absence of forwarding loops relevant to those prefixes.

2.2.1 Motivation for Network Verification

Configuration mistakes are a common root cause of network failures, and their impact is often non-local: an incorrect gateway, a missing return route or a misapplied default route may manifest only for specific destinations and entry points [7, 8]. Traditional testing approaches are inherently sampling-based and thus cannot feasibly cover all source destination combinations and corner cases. Manual grading and manual configuration review also scales poorly and may suffer from inconsistencies, especially when large volumes of submissions must be evaluated under time constraints.

Network verification provides two benefits for automated assessment [9]. First, it improves coverage by reasoning over entire sets of flows rather than a small number of probes. Second, when a property is violated, verification tools can often produce a counterexample, describing a concrete flow and an explanatory forwarding trace that demonstrates the failure (e.g., the exact hop sequence forming a loop). This diagnostic capability supports both fair scoring and actionable feedback for students.

2.2.2 Control Plane vs. Data Plane Verification

Network behavior can be analyzed from control-plane and data-plane perspectives. Control-plane verification focuses on the computation and distribution of routing information (e.g., which routes are selected, how preferences are resolved and whether routing information converges). Data-plane verification, in contrast, focuses on the actual forwarding behavior, given a packet (or a class of packets), how it is forwarded hop-by-hop, and whether it is ultimately delivered, dropped, or trapped in a forwarding loop [10].

While control-plane state influences forwarding behavior, the two are not equivalent. For example, a routing table may contain a plausible-looking entry for a prefix, yet forwarding may still fail due to gateway resolution issues, interface mismatches or missing reverse paths. Conversely, a network may not exhibit an ideal global routing structure from the control-plane viewpoint, but it may still satisfy the data-plane properties required by a specific task.

The routing table grading problems in this framework are fundamentally data-plane oriented: students specify forwarding decisions via static routes and correctness is defined in terms of whether the induced forwarding behavior meets the problem requirements. Accordingly, the verification process in this system prioritizes data-plane properties (reachability and loop-freedom) and constrains checks to the destination-prefix set specified by each question to avoid penalizing behavior unrelated to the grading scope. Figure 2.2 illustrates the conceptual difference between the two perspectives.

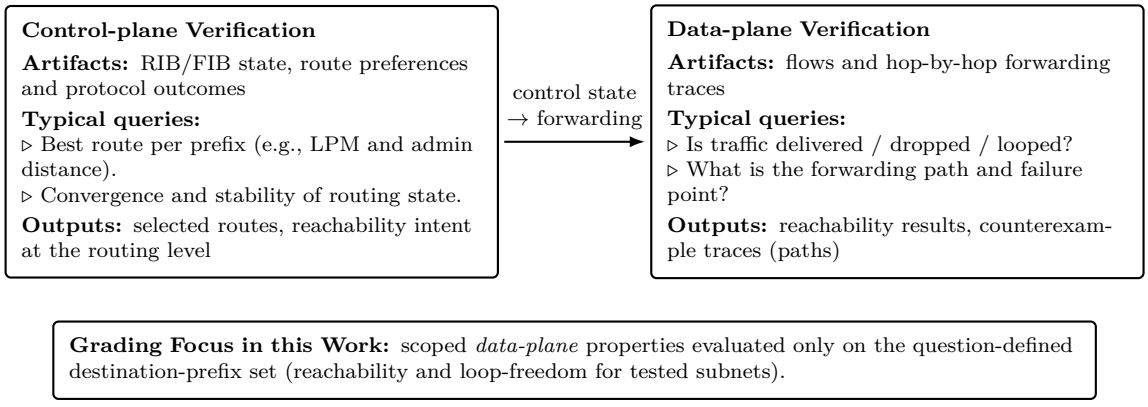


Figure 2.2. Conceptual comparison between control-plane and data-plane verification

2.2.3 Formalization of Network Properties

To enable automated verification, informal notions of correctness must be formalized as checkable properties over a set of flows [11]. Let $\mathcal{F}$ denote the set of flows under consideration (e.g., flows whose destinations belong to the question-defined prefix set). Let $\Pi(f)$ denote the forwarding outcome for a flow $f \in \mathcal{F}$, including its hop-by-hop trace.

Two properties are central to this system:

Reachability: Given a set of sources S and a set of destination prefixes D, the reachability property requires that for every $s \in S$ and every $d \in D$, there exists a finite forwarding path that delivers traffic from s to the destination network (or an endpoint attached to that network). In the grading system, this can be quantified using a reachability index, defined as the fraction of tested (s, d) pairs that are reachable.

Absence of loops: A loop occurs when $\Pi(f)$ revisits the same node(s) indefinitely, forming a directed cycle in the forwarding trace. Importantly, forwarding loops are distinct from physical topology cycles: a physically acyclic topology can still produce forwarding loops due to incorrect routes, while a physically cyclic topology need not exhibit forwarding loops if routing is correct. For grading, loop detection is applied with the same destination-prefix scope as reachability to ensure that only loops relevant to the intended task affect the score.

These formalizations support an interpretable grading scheme that decomposes the final grade into reachability and loop related, improving both fairness and efficiency.

2.3 Overview of Batfish

Batfish is a configuration analysis and network verification framework designed to reason about network behavior without requiring live traffic or physical deployment [12]. Instead of interacting with running devices, Batfish operates on a static representation of the network, known as a snapshot, which contains device configurations and topology descriptions [13]. By analyzing this snapshot, Batfish constructs internal control-plane and data-plane models and answers verification queries in a reproducible manner [14].

Compared with traditional simulation-based and hardware-based validation approaches, Batfish provides three important advantages: (i) it operates offline without packet injection, (ii) it produces deterministic results independent of runtime state and (iii) it scales naturally to multiple configurations. These properties make Batfish particularly suitable for automated grading scenarios, where fairness, repeatability, and scalability are essential.

In this system, Batfish serves as the verification backend of the proposed grading framework. Student routing table answers are first transformed into vendor-style configuration files, together with a topology description, forming a complete snapshot directory. Batfish then analyzes the snapshot and evaluates the required network properties via structured queries. The computed verification outcomes are finally mapped into grading metrics.

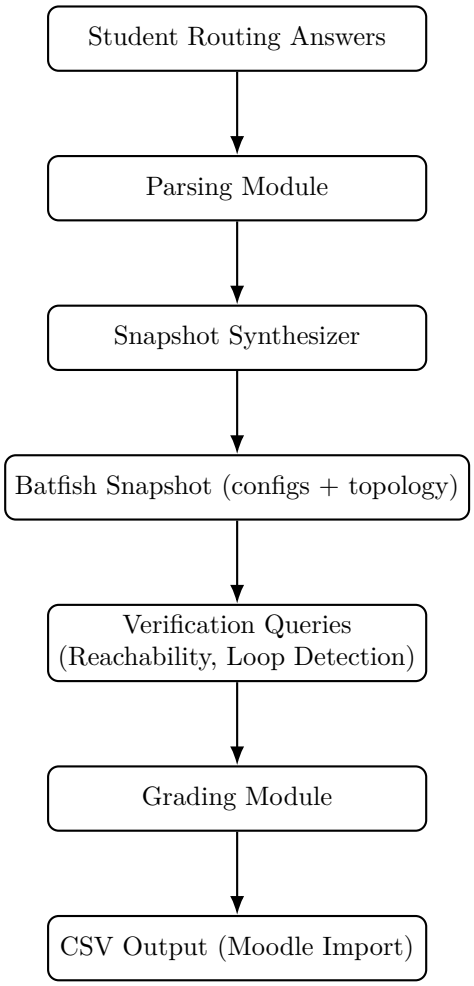


Figure 2.3. Workflow of the proposed Batfish-based automated grading framework

Figure 2.3 shows how a student submission is turned into a verifiable network instance and then graded. The process begins with the routing entries provided by the student. The *parsing module* extracts per-device routing rules from the submission and converts them into a structured format used by the framework.

The *snapshot synthesizer* then reconstructs a complete Batfish snapshot by combining the parsed routing rules with the problem metadata (topology description and IP address assignments). This step is necessary because a submission typically contains only routing entries, while data-plane analysis requires a full network snapshot (device configurations plus topology context).

After the snapshot is built, the framework runs Batfish verification queries to evaluate forwarding behavior. Reachability is checked with respect to the question's *destination prefix set*, and loop detection is used to rule out forwarding loops for the tested flows. The grading module converts these verification outcomes into a final score using a deterministic rule, and the result is exported as a CSV file that can be imported directly into Moodle.

2.3.1 Snapshot-Based Modeling

Batfish models a network through a self-contained snapshot that includes:

- **Device Configurations** (Cisco-style configuration files)
- **Topology Descriptions** (Define physical connection)

This snapshot-based abstraction separates network verification from runtime state and traffic dynamics. In the proposed grading system, each student and each question correspond to an independent snapshot directory. Such isolation ensures that grading results are reproducible and that no cross-question interference occurs.

By synthesizing student routing entries into standardized configurations, the system bridges the gap between informal routing-table answers and a formally analyzable network representation.

2.3.2 Graph-Based Internal Representation

After loading a snapshot, Batfish parses configurations and constructs internal data structures that represent:

- Network nodes (routers and hosts)
- Interfaces and IP assignments
- Routing entries and gateway
- Physical and logical connectivity

From these elements, Batfish derives forwarding behavior for network flows. Importantly, Batfish distinguishes between physical topology and forwarding behavior. A physical loop does not necessarily imply a forwarding loop, and conversely, an incorrect routing entry may create a forwarding loop even in a linear topology. This distinction is critical in automated grading, where correctness depends on forwarding outcomes rather than topology structure alone.

2.3.3 Query-Based Analysis Framework

Batfish exposes verification capabilities through structured queries. Instead of manually simulating packet forwarding, users express desired properties declaratively.

In this system, two core properties are evaluated:

- **Reachability**: whether flows from designated source nodes can reach the question-defined destination prefix set.
- **Loop Detection**: whether forwarding loops exist for flows targeting the tested subnets.

These queries enable the grading framework to compute a reachability index and a loop index for each question. Because verification is query-driven, additional properties (e.g., blackhole detection and path constraints) can be incorporated in future extensions without redesigning the overall system architecture [15].

2.4 Properties Verified by Batfish

The following presents a formalization of the network properties evaluated in the proposed grading framework and explains how verification outcomes are mapped to final grades.

Unlike traditional network verification studies that focus solely on correctness checking, the automated grading scenario requires verification results to be systematically transformed into deterministic grading decisions.

For each question, a Batfish snapshot S is constructed. Based on the destination prefix set defined by the problem, a corresponding test-flow set F is generated. The framework verifies two data-plane properties: Reachability and Loop Detection. The final grade is derived using a threshold-based and piecewise grading rule.

2.4.1 Reachability Property

Formal Definition

Given a network snapshot S and a set of test flows F, a flow $f \in F$ is considered *reachable* if its data-plane forwarding trace successfully reaches the intended destination subnet.

The reachability index is defined as:

$$\mathrm{RI}(S, F) = \frac{|F_{\text{delivered}}|}{|F|} \tag{2.1}$$

where:

- F denotes the set of tested flows
- $F_{\text{delivered}}$ is the subset of flows successfully delivered
- $\mathrm{RI}(S, F) \in [0,1]$

This metric represents the proportion of destination prefixes that are successfully reachable.

Threshold-Based Interpretation for Grading

Although the reachability index is a continuous ratio, the grading framework adopts a strict threshold rule aligned with instructional objectives.

The reachability pass indicator is defined as:

$$R_{\text{pass}}(S) = \begin{cases} 1, & \text{if } \mathrm{RI}(S, F) = 1 \\ 0, & \text{if } \mathrm{RI}(S, F) < 1 \end{cases} \tag{2.2}$$

That is, full reachability is required to pass this criterion. If any destination prefix is unreachable, the solution is considered functionally incorrect.

This design reflects the pedagogical requirement that routing configurations must achieve complete connectivity.

2.4.2 Loop Property

Reachability alone does not guarantee forwarding correctness. Incorrect static routes may still introduce forwarding loops even when all tested destinations are reachable. Therefore, loop detection is additionally required.

A naive global loop detection may identify loops unrelated to the problem defined destination prefixes (e.g., default routes toward external networks). Penalizing such loops would be unfair in a grading scenario.

Let P denote the set of destination prefixes defined by the question, and let L denote the set of forwarding loops detected by Batfish.

We define the subset of loops relevant to the question as:

$$L_P = \{l \in L \mid \mathrm{dst}(l) \in P\} \tag{2.3}$$

The loop-pass indicator is defined as:

$$L_{\mathrm{pass}}(S) = \begin{cases} 1, & \text{if } L_P = \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Only loops affecting question-defined destination prefixes are considered in grading.

2.4.3 Final Grading Rule

Let $G_{\max}$ denote the maximum score of the question. The final grade is determined as follows:

$$G(S) = \begin{cases} G_{\max}, & \text{if } \mathrm{RI}(S,F) = 1 \wedge L_{\mathrm{pass}}(S) = 1 \\ \frac{1}{2}G_{\max}, & \text{if } \mathrm{RI}(S,F) = 1 \wedge L_{\mathrm{pass}}(S) = 0 \\ 0, & \text{if } \mathrm{RI}(S,F) < 1 \end{cases} \tag{2.5}$$

The interpretation is:

- Full score: complete reachability and no scoped forwarding loop
- Half score: full reachability but scoped forwarding loop exists
- Zero score: incomplete reachability

It is worth noting that if reachability is incomplete, loop evaluation does not affect the final result. This prioritizes functional correctness over structural refinement.

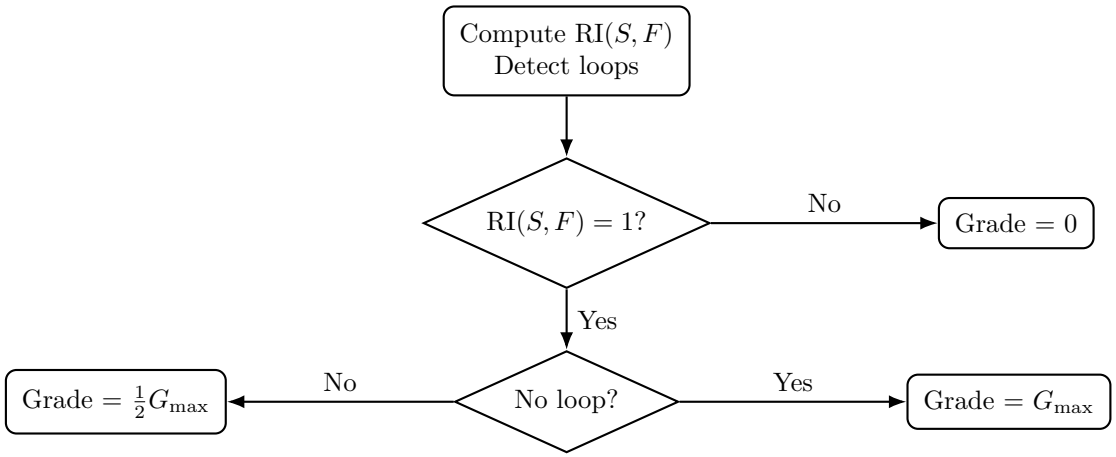


Figure 2.4. Piecewise grading rule based on reachability ratio and loop detection

As shown in the Figure 2.4, the grading rule maps verification outcomes to a final score through a simple decision structure. The process first evaluates functional connectivity by computing the reachability index $RI(S, F)$ over the tested flow set F on the synthesized snapshot S. If $RI(S, F) \neq 1$, at least one required flow fails to reach its destination, indicating that the submission does not satisfy the basic connectivity requirement of the task. In this case, the grade is set to zero, making reachability a hard prerequisite and preventing partially connected solutions from receiving credit.

If $RI(S, F) = 1$, the framework then checks whether any forwarding loops are detected for the tested flows. When no loop is found, the submission is considered both functionally correct and safe with respect to the evaluated traffic, and the maximum grade $G_{\max}$ is awarded. If loops are detected, the submission still provides full reachability but exhibits unsafe forwarding behavior (e.g., packets may circulate indefinitely and consume network resources). To reflect this distinction, the rule assigns partial credit $\frac{1}{2}G_{\max}$ in the presence of loops.

This piecewise design has two intended effects. First, it makes the grading criteria easy to interpret: the score directly reflects whether the solution achieves full connectivity and whether it avoids pathological forwarding behavior. Second, it encourages students to prioritize correctness before refinement: establishing complete reachability is necessary for any credit, while loop detection acts as an additional safety constraint that distinguishes fully correct solutions from risky ones. In later sections, the same decision structure is applied consistently across submissions, which improves grading repeatability and reduces subjective variation in manual assessment.

Chapter 3

Proposed Framework

3.1 Problem Formalization and Assumptions

In traditional teaching practice, routing configuration questions are often graded by comparing the routing tables filled in by students against a standard answer, entry by entry. In other words, the instructor prepares a reference routing table in advance and checks each student submission against it. This method is easy to implement, but it has clear limitations. First, it assumes that each question has only one correct solution. However, in real network environments, the same problem may have multiple valid routing solutions. Although these solutions may differ in structure, they can lead to the same network behavior in the end.

For example, in some cases connectivity can be achieved either by using a default route or by explicitly adding more specific routes. From a functional point of view, both solutions can be correct, even though their text structures are different.

Therefore, if grading relies only on string matching or structural comparison, it may incorrectly mark a behaviorally correct answer as wrong. This is not helpful for encouraging students to understand routing principles, and it may instead push students to memorize templates mechanically.

To solve the above issue, this thesis does not treat routing configuration assessment as a "text matching problem". Instead, it treats it as a "network behavior verification problem". In other words, grading does not focus on whether a student writes exactly the same routing table as the reference answer, but on whether the submitted configuration can produce correct network behavior.

At a formal level, each question can be abstracted into three main parts:

- the network topology
- the IP address assignments of each device (e.g., hosts and routers)
- the static routing rules that students need to provide

Each static routing rule can be expressed as a triple: network prefix, next-hop gateway, and outgoing interface.

With this formulation, the goal of evaluation is no longer to compare the student routing table with a predefined reference routing table. Instead, the system treats all routing rules submitted by the student as a complete network configuration, and then verifies whether its network behavior is correct. Therefore, the core grading question becomes:

Under this configuration, does the student submitted network satisfy the expected connectivity and stability requirements?

More specifically, this thesis considers two key properties:

1. whether all expected destination networks are reachable from each device.
2. whether the generated routing configuration introduces routing loops.

To control the scope and keep the study well-defined, several assumptions are made:

- only static routing is considered; dynamic routing protocols (e.g., OSPF or BGP) are not included
- all links are reliable, bidirectional, and lossless
- the network size remains within the set parameters
- performance factors such as bandwidth and delay are not considered

Through this formalization, routing assessment can be shifted from a syntactic comparison problem to a semantic verification problem, making the verification closer to real network behavior.

3.2 Design Goals

This framework aims to support the assessment of routing quizzes in a way that is extensible, reliable, reproducible, and meaningful for teaching. The system development follows several key design goals.

3.2.1 Full Automation

In large classes, manually grading routing tables is time-consuming and error-prone. An automated system can apply the same grading standard to all submissions and reduce differences caused by subjective judgment. It can also reduce the instructor's workload, allowing more time to be spent on course design and teaching feedback, and improving grading fairness. More importantly, it minimizes manual involvement in the whole assessment pipeline, from question generation to final grade export. Once the system is configured, the process from generating questions to exporting grades can be executed automatically.

3.2.2 Behavior-based Verification

This framework does not evaluate student answers based on text similarity or predefined routing table templates. Instead, it verifies answers based on the actual network behavior produced by the routing configuration. A submission is considered correct if it satisfies the functional requirements specified by the question. This ensures that alternative routing solutions that are correct in behavior can be graded fairly, which makes the evaluation more flexible. This design is closer to real networking practice and also helps students build a better understanding of routing logic.

3.2.3 Extensibility and Integration

The framework is divided into several independent modules, including a question generation module, an answer parsing module, a verification module, and a grading module. Modules interact through standardized data interfaces. In addition, the framework aims to support multiple topology types, including single topology, linear topology, and loop topology. With this modular design, new topology types or grading rules can be added in the future without major changes to the overall system architecture, which supports further extension.

To integrate smoothly with real course workflows, the framework is designed to work with Moodle-based assessments: students submit answers through the platform, while instructors can import the automatically generated grades. The system therefore provides CSV export compatible with Moodle grade import, reducing manual overhead and making large-scale evaluation practical in teaching.

This study relies on an existing network verification tool rather than implementing network analysis algorithms from scratch. Batfish is selected as the underlying verification engine because it already provides stable reachability analysis and loop detection. Reusing a mature tool improves verification reliability and reduces design complexity, allowing the thesis to focus on the assessment workflow and scoring logic.

3.3 System Architecture and Data Flow

This section describes the overall system architecture and the end-to-end data flow of the proposed framework. The overall system architecture is shown in Fig. 3.1. The system consists of host-side control modules and containerized external services. The architecture emphasizes separating "workflow orchestration logic" from "verification services". The whole workflow is divided into multiple modules: some can run directly on the host machine, while others need to run inside Docker containers on the host.

First, the workflow starts from the automatic question generation module. Based on predefined topology templates, this module automatically generates different types of routing configuration questions. The generator currently supports multiple topology types, including single-router topology, linear multi-router topology, and loop-based multi-router topology. For each question, besides generating the question itself, the generator also produces a corresponding metadata file. This metadata records the topology structure and IP address assignments of the question, and it is crucial for later snapshot construction.

Next, the generated questions are imported in XML format into the Moodle platform running in a Docker container. Moodle is only responsible for managing quizzes and collecting student answers, and it does not participate in the verification process. After students finish the quiz, Moodle exports student answers in a structured JSON format.

Then, the answer parsing and snapshot construction module reads both the student answers and the question metadata. It converts the routing tables filled in by students into standardized network configurations, and it builds snapshots for each question. Each snapshot includes device configuration files and a topology description file, which can be analyzed by the network verification tool. A complete snapshot is necessary because

Batfish requires a complete network configuration during analysis, rather than relying only on isolated routing entries.

The constructed snapshots are then passed to the Batfish engine running in a Docker container. The system uses Docker not only to isolate the verification service and keep the environment consistent, but also to avoid dependency conflicts. Containerized deployment helps the system run stably across different environments and reduces potential issues caused by environment differences.

Batfish performs network analysis on each snapshot and automatically verifies important network properties, including reachability between nodes and the presence of routing loops [16]. It outputs verification results, which are then processed by the grading and result export module. Based on predefined rules, this module computes the score for each student and each question, and generates an aggregated grade file. Finally, the grade file can be imported back into Moodle, completing the automated closed loop.

In summary, the architecture ensures clear responsibilities for each module: the host is responsible for workflow orchestration and grading logic, while Batfish inside the Docker container focuses only on network verification. This architecture enables an end-to-end automated assessment process from question generation to formal network verification and grading. The separation between host modules and Docker-based services also makes the system easier to maintain and supports future extension to more complex topologies.

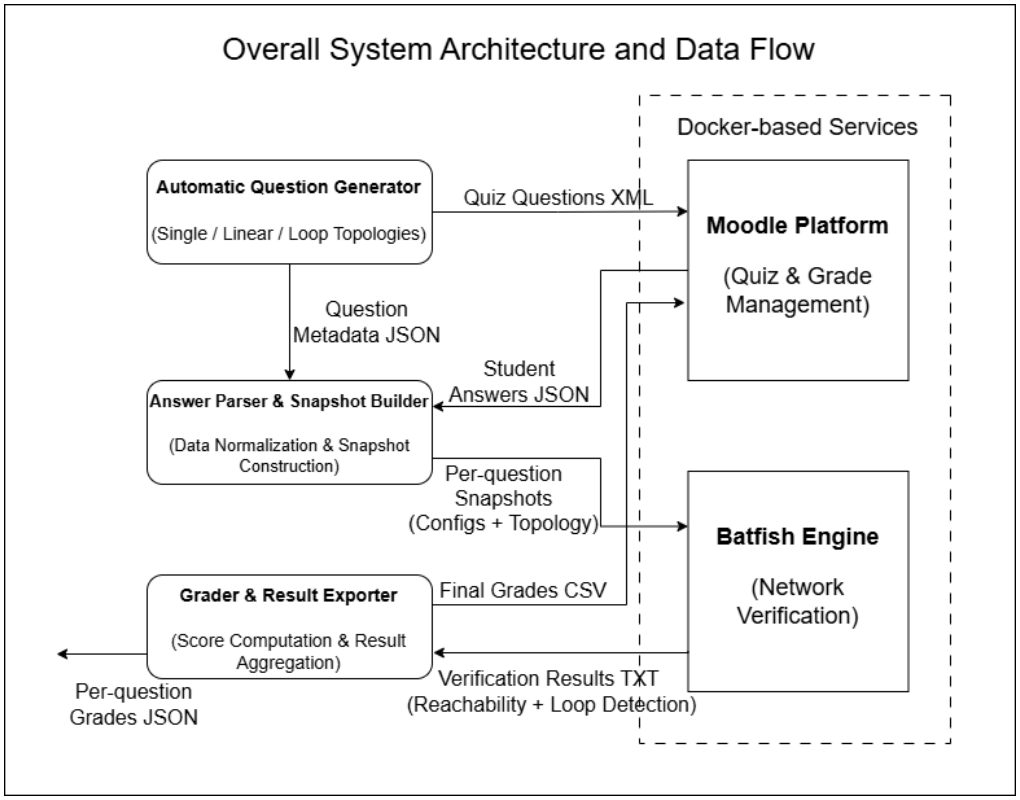


Figure 3.1. System architecture and data flow of the proposed framework

3.4 End-to-End Workflow

From an overall perspective, the proposed framework implements a complete automated workflow.

First, the system automatically generates routing questions. Under predefined constraints, it randomly generates network topologies and IP addressing schemes, and it produces structured metadata describing the expected network configuration. The questions are then imported into Moodle. Students complete the quiz by filling in routing tables for the specified devices through the standard Moodle interface, without requiring any additional plugins.

After the quiz ends, Moodle exports student answers. The system then parses and normalizes the exported answers and constructs a corresponding network snapshot for each student and each question. Each snapshot includes configuration files for all devices and a Batfish-compatible topology description. This step is necessary because verification must be performed based on the complete network state.

Next, Batfish analyzes each snapshot and checks whether the network satisfies the expected properties, such as whether target networks are reachable and whether routing loops exist.

Finally, the grading module calculates scores according to the verification results and the grading rules, and it generates an aggregated grade file compatible with Moodle. This enables automatic feedback and grade integration in Moodle.

In this way, the whole workflow forms a fully automated closed loop from question generation to grading. Verification and grading are closely connected, so the system is both technically rigorous and practically usable in a teaching context.

3.5 Scoring Indicators and Grading Policy

The grading policy of this system is mainly based on two core indicators. Both indicators come from Batfish verification results, rather than reference-answer matching.

3.5.1 Reachability Index

This metric measures whether each device in the topology can reach all expected destination networks. It directly reflects the functional completeness of the routing configuration. If any required path is unreachable, the reachability index becomes less than 1.0, which indicates a serious error in the configuration. In this case, the score is set to zero.

3.5.2 Loop Index

This metric is used to detect whether routing loops exist in the network. The presence of routing loops suggests that routing behavior is unstable or incorrect, which negatively impacts the final score. Even if the network is reachable, a routing loop indicates a problem in the configuration logic. Therefore, when reachability is satisfied but loops are detected, only half of the points are awarded instead of full marks. In this design, a loop index equal to 0 indicates that loops are detected, while a loop index equal to 1 indicates that no loops are detected.

Based on these two indicators, the grading rules are defined as follows:

- if reachability is not satisfied (reachability index < 1.0), the score is 0
- if reachability is satisfied but loops exist (reachability index $= 1.0$ and loop index $= 0$), half credit is awarded
- full credit is awarded only when reachability is satisfied and no loops exist (reachability index $= 1.0$ and loop index $= 1$)

This policy ensures that grading reflects both functional correctness and network stability.

Chapter 4

System Implementation

4.1 Automatic Question Generation

4.1.1 Topology Template Design

To support scalable and structurally diverse routing questions, the system adopts a template-based topology abstraction.

Each template defines a reusable structural pattern consisting of devices (hosts and routers), link relationships, and link types. Concrete IP subnets and interface addresses are instantiated dynamically at generation time.

Three topology families are currently supported:

- **Single topology**: a minimal structure with one router connecting two hosts.
- **Linear topology**: a multi-hop chain with two routers between two hosts.
- **Loop topology**: a meshed structure forming a routing loop among intermediate routers.

Rather than embedding fixed IP addresses in the templates, each topology defines only:

1. The set of devices (hosts and routers).
2. The logical connectivity between devices.
3. The semantic type of each link (host-facing segment or router-to-router link).

This separation allows the same structural template to generate a large number of distinct, non-overlapping routing scenarios.

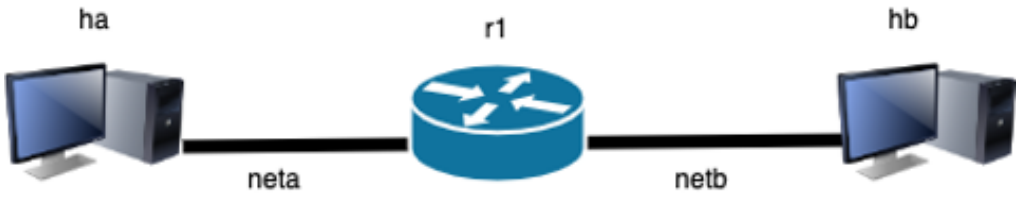


Figure 4.1. Topology templates used in the question generator (Single)

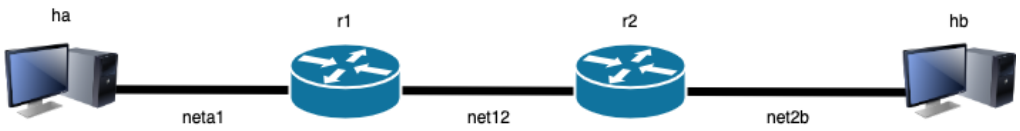


Figure 4.2. Topology templates used in the question generator (Linear)

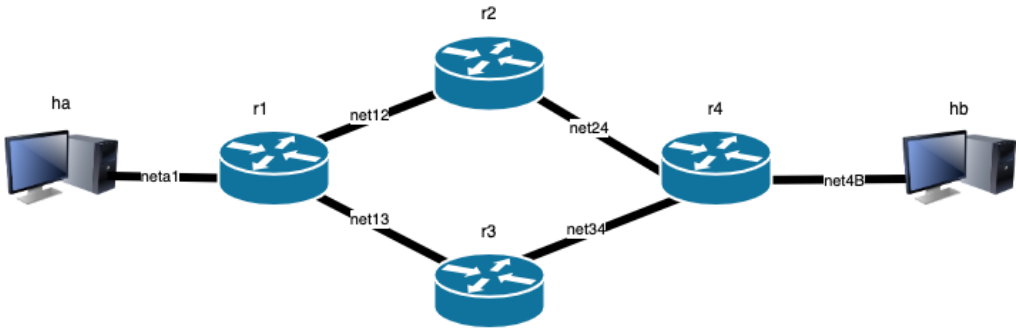


Figure 4.3. Topology templates used in the question generator (Loop)

Link-Type-Aware Prefix Strategy

Each link in a template is classified into one of two categories:

- **Host-facing segments** (e.g., HA–R1, R4–HB), which model LAN-style networks.
- **Router-to-router links**, which represent point-to-point interconnections.

To reflect realistic network design practices, different prefix lengths are assigned according to link type:

- Host-facing segments use a configurable prefix length `HOST_NET_PREFIX` (e.g., /24).
- Router-to-router links use a configurable prefix length `P2P_LINK_PREFIX` (e.g., /30).

This distinction improves realism and ensures that point-to-point links are allocated minimal address space, while host segments retain sufficient address capacity.

Topology Abstraction Structure

To support multiple topology templates in a uniform way, the framework introduces a lightweight metadata abstraction called `TopologyFamily`. The goal is to separate template-specific information (e.g., naming, display assets, and ordering conventions) from the core generation and verification logic. In practice, a topology family acts as the "descriptor" of a template: it provides a stable identifier used throughout the pipeline, defines how subnets and interfaces are ordered when rendering configurations and routing tables, and specifies which devices are relevant for assessment in a given template.

This design makes the question generator, snapshot synthesizer, and grading components operate over the same consistent metadata, which improves reproducibility and simplifies future extensions (e.g., adding new topology types or grading rules without changing the overall system architecture).

Listing 4.1 shows the definition of `TopologyFamily` as an immutable data structure. Rather than encoding such information in ad-hoc conditionals scattered across the codebase, the framework centralizes it in this abstraction so that template behavior is controlled by explicit metadata.

Listing 4.1. Definition of the TopologyFamily data abstraction

```
@dataclass(frozen=True)
class TopologyFamily:
    key: str
    category: str
    title: str
    image_path: Optional[Path]
    nets_order: List[str]
    iface_order: List[str]
    devices_for_tables: List[str]
```

This abstraction defines:

- A unique topology identifier
- The Moodle category name
- The display order of subnets
- The interface ordering for device configuration
- The set of devices for which routing tables must be completed

These fields are consumed by the question generation and snapshot synthesis stages to render consistent topology instances, and by the grading workflow to determine which devices and routing tables are relevant for evaluation. In addition, the abstraction supports heterogeneous link types and flexible subnet prefix configurations. This enables the unified modeling of host networks (e.g., /24) and point-to-point links (e.g., /30) within the same template structure. This unified structure guarantees consistency between question generation, metadata storage, and automated verification.

Single Topology Template

The *single topology* represents the simplest routing scenario, as shown in Fig. 4.4, it contains one router (R1) connecting two hosts (HA and HB) via two host-facing segments. This topology is used to introduce basic static routing tasks where students configure routes to achieve end-to-end connectivity between the two LANs.

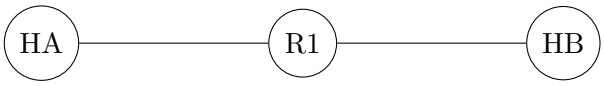


Figure 4.4. Single topology structure

Listing 4.2 shows the generator for the single-topology template in Fig. 4.4. Given a template identifier `qid`, it selects two host-facing subnets and constructs the per-node interface/IP assignment map that will later be used to synthesize device configurations and build the Batfish snapshot.

Listing 4.2. Generation function for the Single topology template

```
1  def gen_single(qid: str) -> Dict[str, Any]:
2      # Two LANs: neta, netb
3      neta, netb = _pick_distinct_subnets_var([HOST_NET_PREFIX,
       HOST_NET_PREFIX])
4      nets = {"neta": str(neta), "netb": str(netb)}
5
6      ip_assignments = {
7          "HA": {"eth0": _iface(neta, "host"), "net": "neta"},
8          "R1": {
9              "eth0": _iface(neta, "router_low"),
10             "eth1": _iface(netb, "router_low")
11         },
12         "HB": {"eth0": _iface(netb, "host"), "net": "netb"}
13     }
```

Lines 1–3 select two distinct subnets, denoted as `neta` and `netb`. Since the single topology contains only host-facing segments and no router-to-router point-to-point links, both subnets are drawn using `HOST_NET_PREFIX`. Line 4 records the chosen subnets in the `nets` dictionary so that subsequent stages can reference the instantiated topology consistently.

Lines 6–13 then build the `ip_assignments` structure, which specifies interface-level addressing for each node. HA and HB each have a single interface (`eth0`) attached to `neta` and `netb`, respectively (Lines 7 and 12). Router R1 has two interfaces: `eth0` connects to the HA-side LAN (`neta`), while `eth1` connects to the HB-side LAN (`netb`) (Lines 8–11). This mapping corresponds directly to the two links shown in Fig. 4.4 and provides the concrete addressing context required for snapshot synthesis and subsequent verification.

Overall, the single-topology template instantiates two host-facing segments and is primarily used for introductory routing exercises and basic two-LAN forwarding problems.

Linear Topology Template

The *linear topology* extends the single-topology template by introducing an additional router to form a multi-hop path from HA to HB. As shown in Fig. 4.5, the topology consists of two host-facing segments (HA–R1 and R2–HB) and one router-to-router point-to-point link (R1–R2). Compared with the single topology, this template increases routing

complexity and requires students to configure multi-hop static routes across intermediate routers.

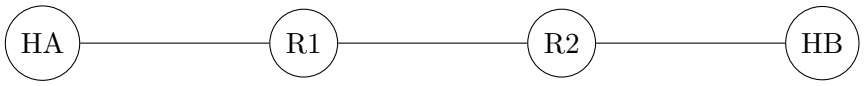


Figure 4.5. Linear topology structure

Listing 4.3 shows the generator for the linear-topology template in Fig. 4.5. Given a template identifier `qid`, it selects three subnets (two host-facing and one point-to-point) and constructs the per-node interface/IP assignment map used later for snapshot synthesis and verification.

Listing 4.3. Generation function for the Linear topology template

```
1  def gen_linear(qid: str) -> Dict[str, Any]:
2      neta1, net12, net2b = _pick_distinct_subnets_var(
3          [HOST_NET_PREFIX, P2P_LINK_PREFIX, HOST_NET_PREFIX]
4      )
5      nets = {"neta1": str(neta1), "net12": str(net12), "net2b": str(net2b)}
6
7      ip_assignments = {
8          "HA": {"eth0": _iface(neta1, "host"), "net": "neta1"},
9          "R1": {
10             "eth0": _iface(neta1, "router_low"),
11             "eth1": _iface(net12, "router_low"),
12         },
13         "R2": {
14             "eth0": _iface(net12, "router_high"),
15             "eth1": _iface(net2b, "router_low"),
16         },
17         "HB": {"eth0": _iface(net2b, "host"), "net": "net2b"},
18     }
```

Lines 1–4 select three distinct subnets to instantiate the three links in the linear topology. The two outer segments are drawn from `HOST_NET_PREFIX`, while the middle router-to-router segment is drawn from `P2P_LINK_PREFIX`, reflecting the different link types in Fig. 4.5. Line 5 records the chosen subnets in the `nets` dictionary using symbolic names (`neta1`, `net12`, `net2b`) so that subsequent stages can reference the instantiated topology consistently.

Lines 7–18 construct the `ip_assignments` structure, which specifies interface-level addressing for each node. HA and HB each have a single host-facing interface (`eth0`) attached to `neta1` and `net2b`, respectively (Lines 8 and 17). Router R1 has two interfaces: `eth0` connects to the HA-side LAN (`neta1`), while `eth1` connects to the point-to-point link `net12` towards R2 (Lines 9–12). Router R2 similarly connects to the point-to-point link `net12` via `eth0` and to the HB-side LAN `net2b` via `eth1` (Lines 13–16). This mapping matches the HA–R1–R2–HB structure in Fig. 4.5 and provides the concrete addressing context required to generate device configurations and build the Batfish snapshot.

Overall, the linear-topology template instantiates two host-facing segments and one router-to-router point-to-point link. The separation between HOST_NET_PREFIX and P2P_LINK_PREFIX ensures realistic prefix allocation for different link types, while the additional router introduces a multi-hop routing requirement for student submissions.

Loop Topology Template

The *loop topology* is the most complex template in the framework and is designed to introduce forwarding anomalies that are difficult to capture in simpler settings. As shown in Fig. 4.6, two host-facing segments connect HA–R1 and R4–HB, while four router-to-router point-to-point links (R1–R2, R2–R4, R4–R3, and R3–R1) form a cycle between the intermediate routers.

Compared with the linear topology, the presence of a cycle enables more advanced tasks, including configurations that may accidentally create forwarding loops, and therefore serves as a representative scenario for validating loop detection.

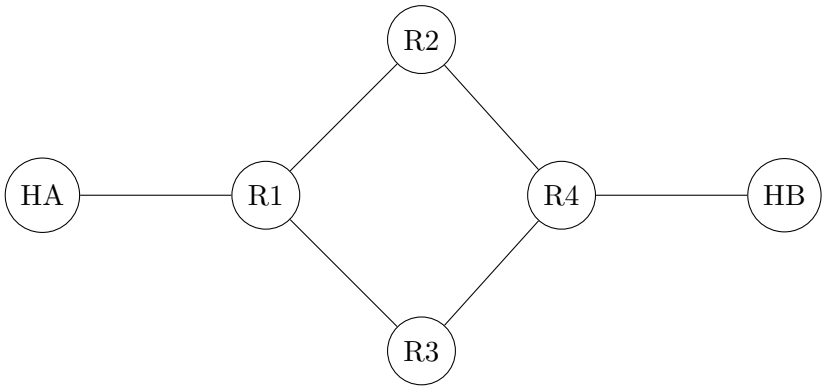


Figure 4.6. Loop topology structure

Listing 4.4 shows the generator for the loop-topology template in Fig. 4.6. Given a template identifier `qid`, it selects six distinct subnets (two host-facing and four point-to-point) and constructs the per-node interface/IP assignment map used later for snapshot synthesis and verification.

Listing 4.4. Generation function for the Loop topology template

```
1  def gen_loop(qid: str) -> Dict[str, Any]:
2      neta1, net12, net13, net24, net34, net4b = _pick_distinct_subnets_var(
3          [HOST_NET_PREFIX, P2P_LINK_PREFIX, P2P_LINK_PREFIX, P2P_LINK_PREFIX,
       P2P_LINK_PREFIX, HOST_NET_PREFIX]
4      )
5
6      nets = {
7          "neta1": str(neta1),
8          "net12": str(net12),
9          "net13": str(net13),
10         "net24": str(net24),
11         "net34": str(net34),
12         "net4b": str(net4b),
13     }
14
15     ip_assignments = {
16         "HA": {"eth0": _iface(neta1, "host"), "net": "neta1"},
17         "R1": {
18             "eth0": _iface(neta1, "router_low"),
19             "eth1": _iface(net12, "router_low"),
20             "eth2": _iface(net13, "router_low"),
21         },
22         "R2": {
23             "eth0": _iface(net12, "router_high"),
24             "eth1": _iface(net24, "router_low"),
25         },
26         "R3": {
27             "eth0": _iface(net13, "router_high"),
28             "eth1": _iface(net34, "router_low"),
29         },
30         "R4": {
31             "eth0": _iface(net24, "router_high"),
32             "eth1": _iface(net34, "router_high"),
33             "eth2": _iface(net4b, "router_low"),
34         },
35         "HB": {"eth0": _iface(net4b, "host"), "net": "net4b"},
36     }
```

Lines 1–4 select six distinct subnets to instantiate the six links in the loop topology. The first and last subnets correspond to the two host-facing segments and are drawn from `HOST_NET_PREFIX`, while the four middle subnets correspond to router-to-router point-to-point links and are drawn from `P2P_LINK_PREFIX`. Lines 6–13 record the chosen subnets in the `nets` dictionary using symbolic names (`neta1`, `net12`, `net13`, `net24`, `net34`, `net4b`), which are reused consistently by later stages when rendering device configurations and constructing the snapshot.

Lines 15–36 build the `ip_assignments` structure, which specifies interface level addressing for each node and matches the connectivity shown in Fig. 4.6. HA and HB each have a single host-facing interface (`eth0`) attached to `neta1` and `net4b`, respectively

(Lines 16 and 35). Router R1 has three interfaces: `eth0` connects to the HA-side LAN (`neta1`), while `eth1` and `eth2` connect to two point-to-point links in the cycle (Lines 17–21). Routers R2 and R3 each connect to two point-to-point links (Lines 22–29), and router R4 connects to two point-to-point links plus the HB-side LAN segment `net4b` (Lines 30–34). This explicit interface mapping instantiates the cycle R1–R2–R4–R3–R1 and provides the concrete addressing context required for Batfish to compute data-plane forwarding behavior.

Overall, the loop-topology template combines realistic link-type-aware prefix allocation with a cyclic router interconnection. This makes it suitable for advanced routing exercises and, in particular, for assessing whether the verification component can correctly detect forwarding loops under multi-path and cyclic topologies.

Interface Address Assignment

After subnets are sampled, interface IP addresses are assigned using a role-aware strategy.

For host-facing segments, interface indices follow a consistent convention (e.g., router uses a low-index address, host uses a higher-index address). For point-to-point /30 links, only the two usable host addresses are assigned, ensuring correctness and avoiding invalid address allocation.

This design guarantees that generated configurations remain structurally valid while preserving sufficient variability across questions.

Advantages of Template-Driven Design

The template-based approach provides several benefits:

- **Extensibility:** New topologies can be introduced by defining additional template functions without modifying the core generation pipeline.
- **Controlled difficulty:** Each topology family corresponds to a predefined structural complexity level. Furthermore, controlled variation of subnet prefix lengths (e.g., host networks vs. point-to-point links) enables fine-grained difficulty tuning.
- **Structural consistency:** The metadata structure remains uniform across question types, even when heterogeneous link types or varying subnet prefix lengths are introduced.
- **Verification compatibility:** The generated data aligns directly with the snapshot-building and Batfish verification stages.

This design forms the structural foundation for subsequent configuration synthesis and automated verification processes.

4.1.2 IP/Subnet Sampling Strategy and Constraints

Automated routing configuration question generation requires an IP allocation mechanism that ensures legality, structural consistency, scalability, and compatibility with automated

verification. A purely random address assignment approach may lead to subnet overlaps, ambiguous routing behavior, or failures in grading and Batfish-based validation. To address these challenges, the system adopts a layered IP generation strategy consisting of base network selection, subnet carving, overlap prevention, and role-aware interface assignment.

Private Address Space Selection

All generated IP addresses are strictly confined to the RFC1918 private IPv4 ranges, namely 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. Restricting address allocation to private space prevents conflicts with public networks and aligns the generated scenarios with realistic enterprise deployments.

Instead of directly sampling arbitrary subnets from the entire private space, the system first constructs a /16 base network that serves as the parent container for all subnets within a single question instance. Choosing a /16 granularity provides sufficient address capacity to support multiple /24 host segments as well as numerous /30 point-to-point links, while keeping the allocation logic simple and controllable.

The implementation of base network generation is shown below:

Listing 4.5. Private base network generation

```
def _rand_private_base() -> ipaddress.IPv4Network:
    choice = random.choice(["10", "172", "192"])
    if choice == "10":
        a = 10
        b = random.randint(0, 255)
        return ipaddress.ip_network(f"{a}.{b}.0.0/16", strict=False)
    elif choice == "172":
        a = 172
        b = random.randint(16, 31)
        return ipaddress.ip_network(f"{a}.{b}.0.0/16", strict=False)
    else:
        return ipaddress.ip_network(f"192.168.{random.randint(0,255)}.0/16",
     strict=False)
```

Figure 4.7 illustrates the hierarchical relationship between the RFC1918 private ranges and the randomly selected /16 base network.

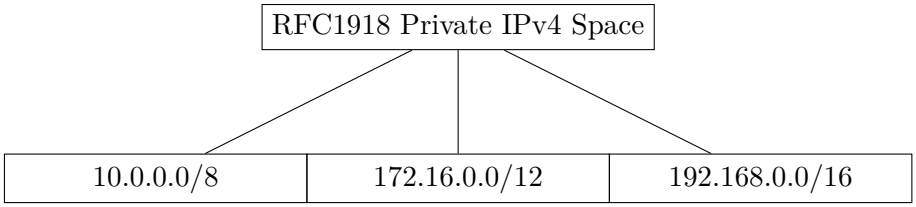


Figure 4.7. RFC1918 private address ranges and base network selection

Variable-Length Subnet Allocation

After the base network is selected, the framework allocates link subnets according to link semantics. Host-facing segments are assigned larger prefixes (e.g., `/24`) to provide sufficient host address space, while router-to-router links use smaller point-to-point prefixes (e.g., `/30`). For each topology template, this policy is encoded as an ordered list of target prefix lengths (e.g., `[24, 30, 24]` for the linear topology). Subnets are then carved from a private base network by selecting candidate subnets of the requested prefix length and enforcing non-overlap, which yields realistic addressing while allowing controlled randomization across generated instances.

Listing 4.6 presents the core routine used for variable-length subnet allocation. Given a list of target prefix lengths, the function returns a set of distinct IPv4 subnets carved from randomly selected private base networks. Therefore, the main goal is to generate non-overlapping link networks with realistic prefix sizes (e.g., /24 for host-facing segments and /30 for point-to-point links), while keeping the generation process reproducible and robust across different topology templates.

Listing 4.6. Variable-length subnet allocation

```
1  def _pick_distinct_subnets_var(prefixlens: list[int]) -> list[ipaddress.
       IPv4Network]:
2      out: list[ipaddress.IPv4Network] = []
3      attempts = 0
4
5      while len(out) < len(prefixlens) and attempts < 5000:
6          attempts += 1
7          base = _rand_private_base()
8
9          target = prefixlens[len(out)]
10         if not (16 <= target <= 30):
11             raise ValueError("prefixlen must be in [16, 30] for this
       generator")
12         if base.prefixlen == target:
13             cand = base
14         elif base.prefixlen < target:
15             cand = random.choice(list(base.subnets(new_prefix=target)))
16         else:
17         if all(not cand.overlaps(x) for x in out):
18             out.append(cand)
19
20     if len(out) < len(prefixlens):
21         raise RuntimeError("Failed to allocate non-overlapping subnets;
       adjust ranges/prefixlens.")
22
23     return out
```

The function constructs the output incrementally until the number of allocated subnets matches the number of requested prefix lengths (Lines 1–3). At each iteration, it first selects a private base network (Line 7). Using a private base ensures that generated subnets do not accidentally collide with public address space and keeps instances

self-contained for educational snapshots and offline analysis.

The target prefix length for the next subnet is taken from the input list (Line 9). The implementation restricts valid prefix lengths to the range [16,30] (Lines 10–11). This constraint reflects the intended usage of the generator: the base network must be sufficiently large to support carving multiple subnets (e.g., a /16 private block), while link networks are not expected to be more specific than /30 in the current static-routing templates. Enforcing this range prevents unrealistic or unsupported configurations from silently producing invalid instances.

Candidate subnet selection follows a simple case analysis (Lines 12–16). If the base prefix already matches the target, the base itself is used. If the base is larger (less specific) than the target, the routine randomly selects one subnet of the desired prefix length from the base network. This random choice introduces controlled variation between questions while preserving the requested prefix sizes.

To guarantee uniqueness, each candidate subnet is checked against all previously selected subnets (Lines 17–18). Only non-overlapping candidates are appended to the output list, ensuring that each link network corresponds to a distinct address block and avoiding ambiguous interface addressing later during snapshot synthesis.

The allocation loop is bounded by a maximum number of attempts (Line 5). This upper bound prevents the generator from running indefinitely in corner cases where the requested prefixes cannot be satisfied without overlap (e.g., too many large subnets carved from a small base, or overly restrictive prefix combinations). If the function fails to allocate a full set of non-overlapping subnets within the attempt budget, it raises an explicit error (Lines 20–21) and prompts the caller to adjust the prefix range or the requested prefix-length sequence. Finally, once all requested subnets are allocated, the routine returns the resulting list (Line 23).

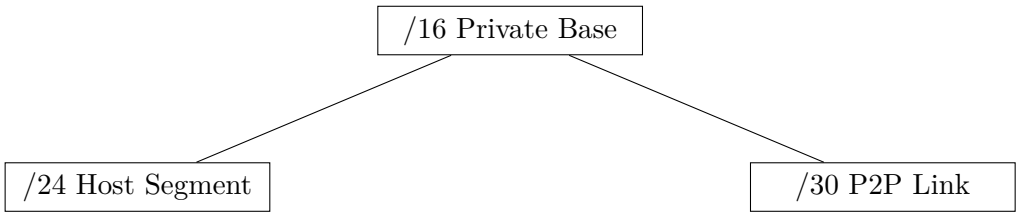


Figure 4.8. Hierarchical subnet carving from a /16 base network

Figure 4.8 illustrates the hierarchical carving of /24 host segments and /30 point-to-point links from a /16 base network.

Overlap Prevention Mechanism

To ensure deterministic routing behavior and reliable automated grading, all generated subnets must be mutually exclusive. Overlapping subnets may cause ambiguous route selection or unintended prefix covering effects. Therefore, each candidate subnet is explicitly verified against previously allocated subnets using an overlap detection mechanism.

The condition `cand.overlaps(x)` guarantees that no two allocated subnets intersect. If an overlap is detected, the selection process is repeated until a valid subnet is found or a predefined attempt limit is reached. This mechanism ensures structural correctness without significantly affecting performance under the current generation scale.

Role-Aware Interface Assignment

After subnet allocation, concrete interface IP addresses must be assigned to hosts and routers. Random selection of host addresses would reduce structural predictability and may accidentally use network or broadcast addresses. To prevent such issues, the system employs a role-aware allocation strategy.

Each interface is assigned a host index determined by its semantic role (e.g., host, router_low, router_high). For /30 subnets, only the two usable host addresses are assigned to the two routers. For larger subnets such as /24, predefined index positions are used to ensure consistent and readable address placement.

The implementation is shown below:

Listing 4.7. Role-aware interface address generation

```
def _pick_index(net: ipaddress.IPv4Network, role: str) -> int:
    if net.prefixlen == 30:
        if role == "router_low":
            return 1
        elif role == "router_high":
            return 2
        else:
            return 1
    if role == "router_low":
        return 2
    if role == "router_high":
        return 11
    return 6
def _iface(net: ipaddress.IPv4Network, role: str) -> str:
    return _cidr(_host_ip(net, _pick_index(net, role)), net)
```

Through this layered and role-aware design, the IP generation mechanism balances randomness and structural control. All addresses remain within valid private ranges, subnets are guaranteed to be non-overlapping, heterogeneous prefix lengths are supported, and the resulting configurations remain fully compatible with automated grading and Batfish-based verification.

4.1.3 Moodle XML Construction and Categories

After topology and IP allocation are generated, the structured metadata must be transformed into a Moodle-compatible XML format. In the implemented framework, this transformation is performed directly within the generator script, which produces both a Moodle XML file and a corresponding metadata JSON file.

XML Structure Construction

The Moodle question file follows the standard `<quiz>` root structure. Each routing problem is exported as an `essay`-type question using a dedicated XML builder function.

The core function responsible for constructing individual question blocks is:

Listing 4.8. Construction of a single Moodle XML question block

```
def _essay_question_xml(qname: str, qtext_html: str, response_template: str
    = "") -> str:
    return f"""  <question type="essay">
    <name><text>{_xml_escape(qname)}</text></name>
    <questiontext format="html">
      <text>{_cdata(qtext_html)}</text>
    </questiontext>
    <defaultgrade>10.0000000</defaultgrade>
    <responseformat>editor</responseformat>
    <responserequired>1</responserequired>
    <responsefieldlines>15</responsefieldlines>
    <responsetemplate format="html">
      <text>{_cdata(response_template)}</text>
    </responsetemplate>
  </question>
"""
```

Each generated routing problem is wrapped as an `essay` question. The HTML content is embedded inside a CDATA section to preserve formatting and avoid XML escaping conflicts. The response template predefines routing table placeholders (e.g., `[R1] ROUTING TABLE`) to guide student input.

All question blocks are then aggregated under a single `<quiz>` root element using:

Listing 4.9. Aggregation of question blocks under the Moodle `<quiz>` root element

```
def build_quiz_xml(question_blocks: List[str]) -> str:
    return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<quiz>\n" \
           + "".join(question_blocks) + "</quiz>\n"
```

Dynamic Question Rendering

The question statement is dynamically constructed from the generated subnet and interface assignment data. The HTML rendering function integrates topology metadata, subnet lists, and IP assignments into a structured question body.

Listing 4.10. Dynamic rendering of routing problem statements from generated metadata

```
def build_question_html(qid: str, family: TopologyFamily, nets: Dict[str,
    str], assign_flat: Dict[str, str]) -> str:
    return f"""
<div>
  <p><b>ID:</b> {qid}</p>
  <p>Based on the topology below, calculate the <b>complete routing table</b
    > for {", ".join(family.devices_for_tables)}.</p>
  <p>Use the format: <b>Network Prefix | Gateway | Interface</b>.</p>
  {_img_html(family.image_path)}
  <h3>Subnets</h3>
  {_subnets_ul(nets, family.nets_order)}
  <h3>IP assignments</h3>
  {_ip_assign_ul(assign_flat, family.iface_order)}
  <h3>Answer guidelines</h3>
  {_answer_guidelines_html()}
</div>
"""
```

This mechanism guarantees that all displayed subnet definitions and interface addresses originate from the same internal metadata used for verification. As a result, structural consistency is preserved across generation, presentation, and grading.

Figure 4.9 shows an example of a generated question after import into Moodle:

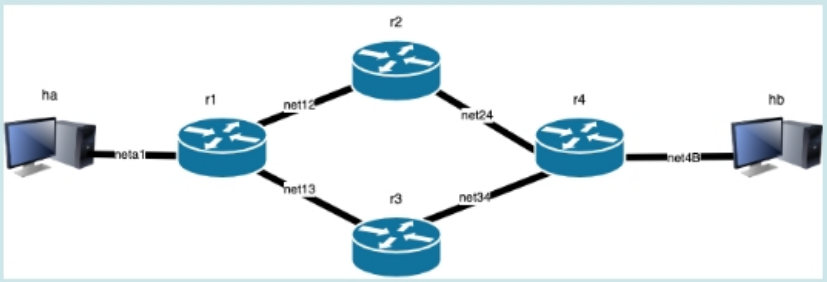
LOOP_001 **Version 1 (latest)**

Question 1
Not yet answered
Marked out of 10.00

ID: LOOP_001

Based on the topology below, calculate the **complete routing table** for HA, R1, R2, R3, R4, HB.

Use the format: **Network Prefix | Gateway | Interface.**



Subnets

- **neta1**: 192.168.2.0/24
- **net12**: 192.168.173.36/30
- **net24**: 192.168.182.88/30
- **net13**: 192.168.85.244/30
- **net34**: 10.68.99.216/30
- **net4b**: 10.42.183.0/24

IP assignments

- **ha**: 192.168.2.6/24
- **hb**: 10.42.183.6/24
- **r1a**: 192.168.2.2/24
- **r12**: 192.168.173.37/30
- **r13**: 192.168.85.245/30
- **r21**: 192.168.173.38/30
- **r24**: 192.168.182.89/30
- **r31**: 192.168.85.246/30
- **r34**: 10.68.99.217/30
- **r42**: 192.168.182.90/30
- **r43**: 10.68.99.218/30
- **r4b**: 10.42.183.2/24

Answer guidelines

- Write **one route per row**, with no extra text. You can leave unused rows blank.
- Please fill each device in its own routing table: **[HA], [R1], [HB]** (or more, depending on topology).
- Use **0.0.0.0/0** for the default network prefix.
- Use * for directly connected networks.
- Use **ethX** for the interface name.

Figure 4.9. Example of an automatically generated question in Moodle (Loop)

Figure 4.10 shows an example of a generated answer after import into Moodle:

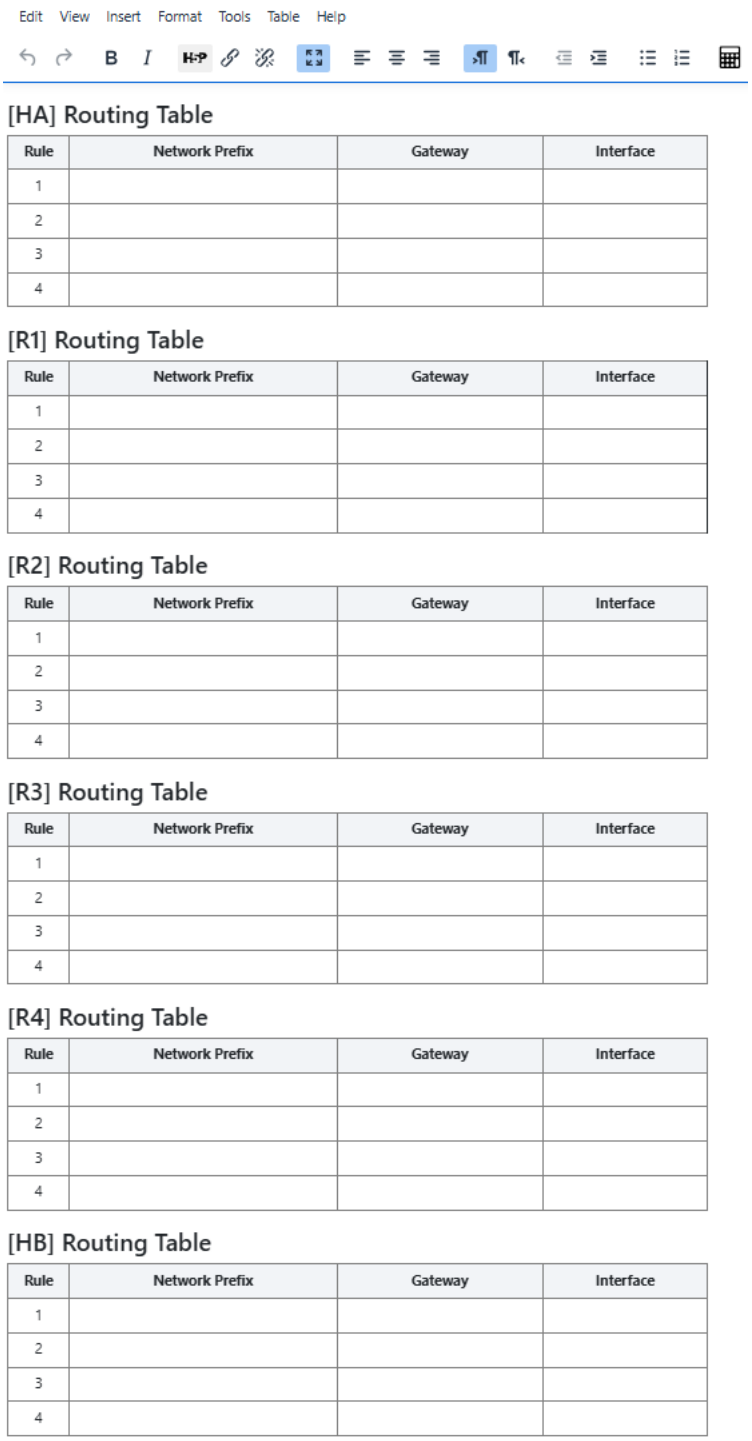


Figure 4.10. Example of an automatically generated answer in Moodle (Loop)

Batch Export and Platform Integration

After all question blocks are constructed, the final XML file is written to disk:

Listing 4.11. Batch export of generated routing problems to a Moodle XML file

```
1 OUT_XML.write_text(build_quiz_xml(blocks), encoding="utf-8")
```

The generated file can be directly imported into the Moodle Question Bank under separate categories (Single, Linear, Loop), as defined by the script. Experimental deployment confirms that hundreds of generated routing problems can be imported in a single batch without manual modification.

Figure 4.11, 4.12 and 4.13 illustrate the Moodle batch import interface:

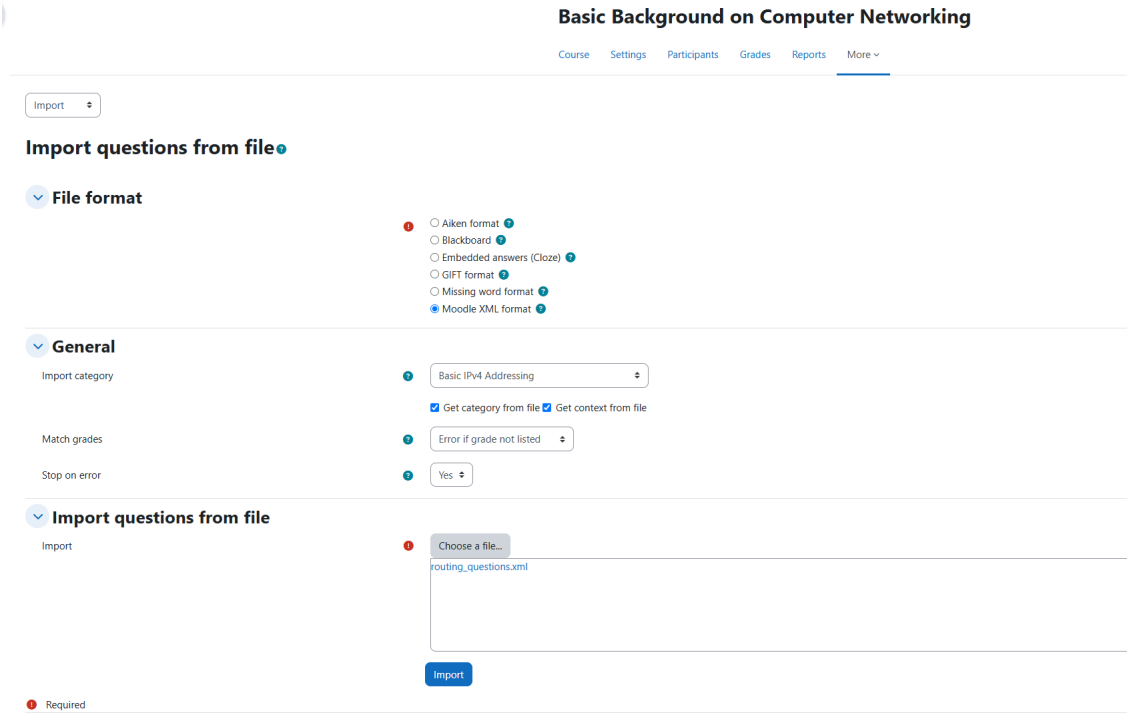


Figure 4.11. Batch import of generated routing problems into Moodle (Step 1)

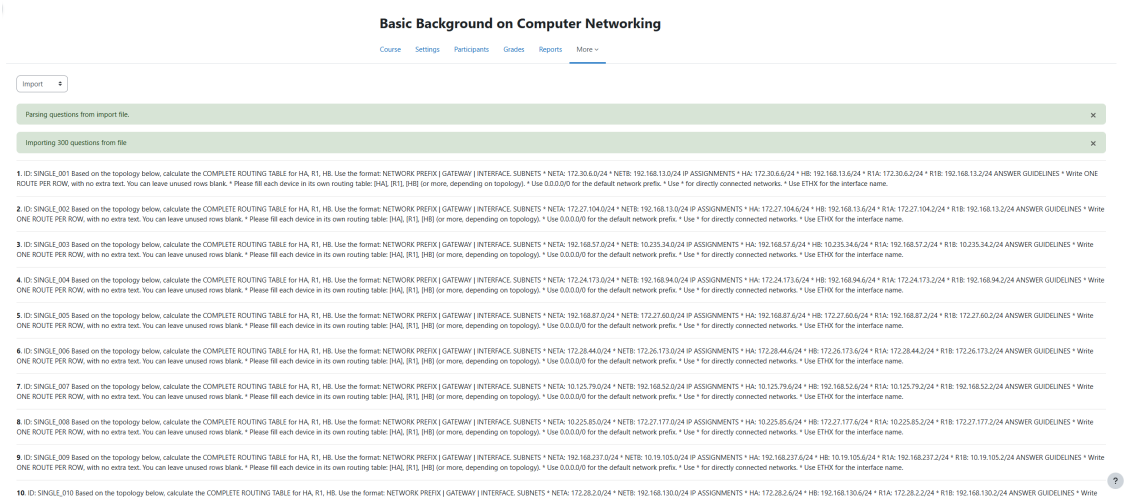


Figure 4.12. Batch import of generated routing problems into Moodle (Step 2)

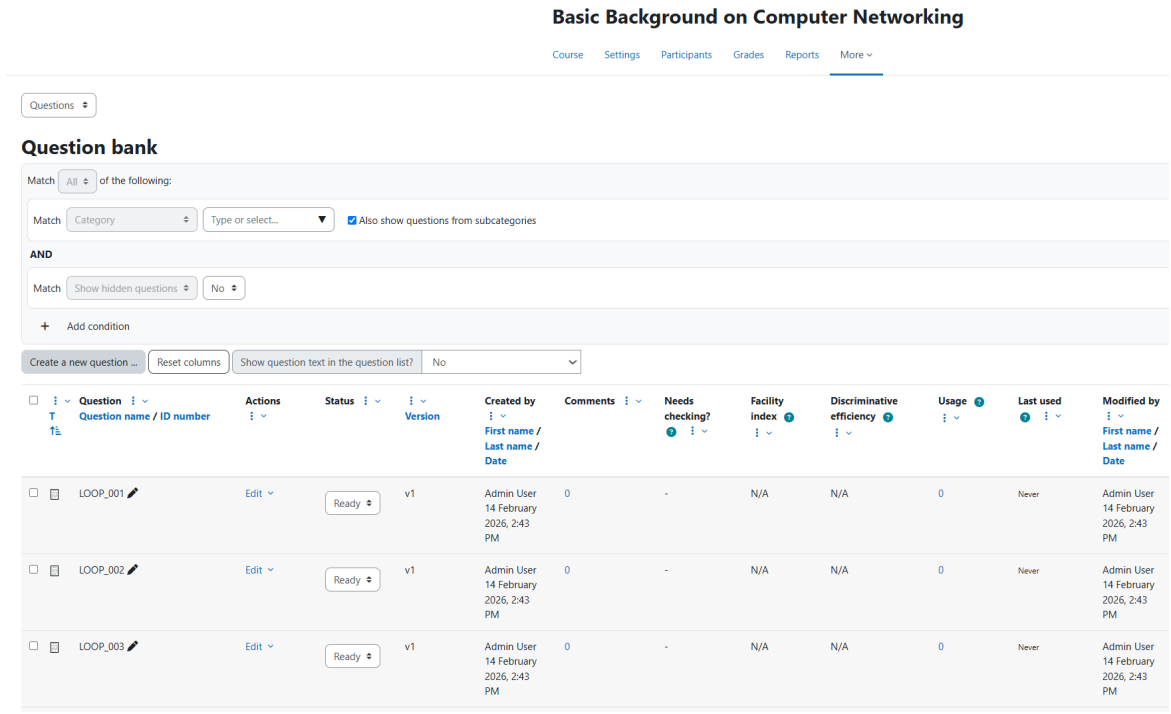


Figure 4.13. Batch import of generated routing problems into Moodle (Step 3)

Consistency with the Verification Module

In parallel with XML generation, the script exports a JSON metadata file containing the complete topology and IP assignment information:

Listing 4.12. Export of topology metadata for automated verification

```
OUT_META.write_text(json.dumps(meta_all, indent=2), encoding="utf-8")
```

This metadata file is subsequently consumed by the Batfish-based verification module. By preserving a unified internal representation across XML export and verification stages, the framework ensures deterministic grading behavior and avoids discrepancies between the deployed question and the evaluation logic.

The XML layer therefore serves strictly as a presentation and interaction interface, while correctness validation remains fully driven by the shared metadata representation.

4.2 Student Answer Collection and Parsing

4.2.1 Exporting Student Responses from Moodle

After the automatic question generation phase and the import of questions into Moodle, students complete the quiz through the Moodle platform. Once the assessment is finished, the system retrieves all student submissions by exporting the quiz responses in JSON format.

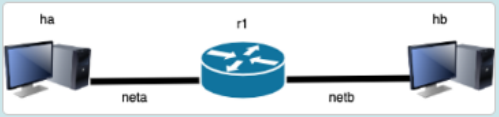
JSON is chosen as the export format primarily because it preserves hierarchical data structures, which simplifies automatic parsing compared to flat formats such as CSV, and secondly because the exported structure closely reflects Moodle's internal data organization, reducing the risk of information loss during format transformation.

Figure 4.14 shows the interface after students answer questions on the Moodle platform, and Figure 4.15 illustrates the response export interface in Moodle. These steps ensure that all subsequent processing is based on real assessment data rather than manually constructed examples.

ID: Q043_3763

Based on the topology below, calculate the **complete routing table** for **HA**, **R**, and **HB**.

Use the format: **Network Prefix | Gateway | Interface**.



Subnets:

- neta : 172.27.191.0/24
- netb : 172.24.138.0/24

IP assignments:

- ha-eth0 : 172.27.191.6/24
- r1-eth0 : 172.27.191.2/24
- r1-eth1 : 172.24.138.11/24
- hb-eth0 : 172.24.138.6/24

Answer guidelines:

- Write **one route per row**, with no extra text. You can leave unused rows **blank**.
- Please fill **each device** in its own routing table: **[HA]**, **[R1]**, **[HB]**.
- Use **0.0.0.0/0** for default network prefix.
- Use ***** for directly connected networks.
- Use **ethx** for interface.

[HA] Routing Table

Rule	Network Prefix	Gateway	Interface
1	172.27.191.0/24	*	eth0
2	0.0.0.0/0	172.27.191.2	eth0
3			
4			
5			

[R1] Routing Table

Rule	Network Prefix	Gateway	Interface
1	172.27.191.0/24	*	eth0
2	172.24.138.0/24	*	eth1
3			
4			
5			

[HB] Routing Table

Rule	Network Prefix	Gateway	Interface
1	172.24.138.0/24	*	eth0
2	0.0.0.0/0	172.24.138.11	eth0
3			
4			
5			

Figure 4.14. Moodle quiz response interface

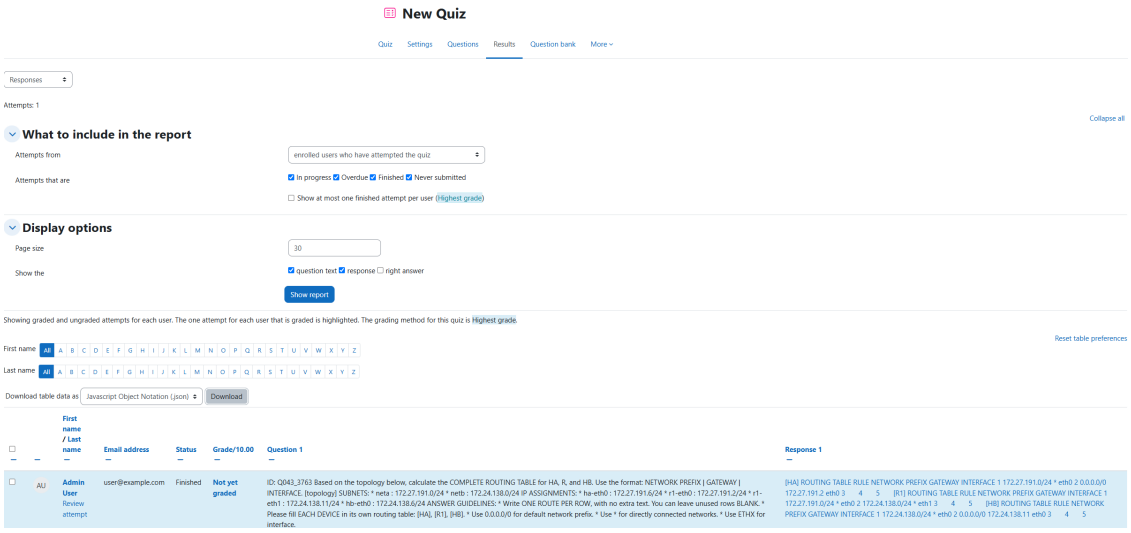


Figure 4.15. Moodle quiz response export interface

4.2.2 Structure of the Exported JSON File

The exported JSON file contains an array of student records. Each record includes basic identity information and one or more quiz attempts. A simplified structure is shown below:

```
[[{
    "lastname": "User",
    "firstname": "Admin",
    "emailaddress": "user@example.com",
    "status": "Finished",
    "started": "...",
    "completed": "...",
    "duration": "...",
    "grade1000": "Not yet graded",
    "question1": "ID: SINGLE_001  Based on the topology below...",
    "response1": "[H1] ROUTING TABLE  \t\tRULE  \t\tNETWORK PREFIX
    \t\tGATEWAY  \t\tINTERFACE..."
}]]
```

Each attempt contains pairs of `questionN` and `responseN` fields. The question field stores the full question text (including an embedded question ID), while the response field contains the routing table provided by the student.

Since Moodle does not directly expose a structured question identifier field, the system extracts the question ID from the question text itself.

4.2.3 Question ID Extraction Mechanism

During question generation, each question is embedded with a unique identifier (QID), such as:

`ID: LOOP_001`

The following function extracts the QID using a regular expression:

Listing 4.13. QID Extraction Function

```python
def extract_qid(text: str) -> str:
    m = re.search(r"\bID:\s*([A-Za-z]+_\d+)\b", text)
    if not m:
        raise ValueError("Missing or invalid ID")
    return m.group(1)
```

This mechanism enables the grading framework to associate each student submission exported from Moodle with the corresponding metadata entry generated during the question creation phase. Since Moodle does not provide a structured identifier field for each question in the exported JSON file, the QID is explicitly embedded within the question text during generation. After export, the system parses the question text to extract this identifier and uses it as a key to retrieve the matching metadata entry.

The QID therefore serves as the sole shared reference between the question generation module and the answer parsing and snapshot construction module. This design ensures loose coupling between system components: the generation phase produces fully self-contained metadata indexed by QID, while the grading phase operates independently by resolving metadata dynamically through identifier matching. No additional structural dependencies between the two modules are required.

4.2.4 Automatic Matching of Question

Within each quiz, question and response fields are indexed numerically (e.g., `question1`, `response1`). The system dynamically pairs them using the following logic:

Listing 4.14. Regular Expression Based Question Response Pairing

```python
def find_question_response_pairs(attempt):
    q_re = re.compile(r"^question(\d+)$")
    r_re = re.compile(r"^response(\d+)$")
    pairs = {}

    for k, v in attempt.items():
        if (m := q_re.match(k)):
            pairs.setdefault(int(m.group(1)), {})["q"] = v
        elif (m := r_re.match(k)):
            pairs.setdefault(int(m.group(1)), {})["r"] = v
    return [
    (i, d.get("q", ""), d.get("r", ""))
    for i, d in sorted(pairs.items())
]
```

After collecting all entries, the system sorts the indices numerically and outputs ordered (`index`,`question`,`response`) tuples. Missing fields are handled gracefully by assigning empty defaults, allowing the pipeline to remain robust even if certain question response pairs are incomplete or irregularly structured.

This strategy avoids reliance on a predefined number of questions and remains resilient to variations in field ordering within the exported JSON file. As a result, the parsing module can flexibly support different quiz configurations and future extensions without structural modification.

4.2.5 Parsing Student Routing Table

The routing tables submitted by students are semi-structured text. A typical example is shown below:

[HA] Routing Table

Rule	Network Prefix	Gateway	Interface
1	172.27.191.0/24	*	eth0
2	0.0.0.0/0	172.27.191.2	eth0
3			

Figure 4.16. Student Routing Table

The parsing process consists of three main stages.

(1) Text Preprocessing Before parsing, the system removes invisible Unicode characters and normalizes whitespace:

```
raw = raw.replace("\xa0", " ")
raw = re.sub(r"[\u2000-\u200B\u202F\u205F]", " ", raw)
```

This step is necessary because students may copy and paste content from different editors, which can introduce non-standard characters that interfere with parsing.

(2) Device Block Identification Each device section is marked by a header such as `[HA]`. The following logic identifies device blocks:

```
for f in fields:
    m = re.match(r"^\[([A-Za-z0-9_]+)\]", f)
    if m:
        dev = m.group(1)
        routes[dev] = []
        buf = []
        continue
```

Whenever a new device header is detected, a new routing list is initialized.

(3) Route Field Extraction Each routing entry is expected to contain four fields: rule index, network prefix, gateway, and interface. The fields are extracted and validated before being stored:

```
fields = [f.strip() for f in raw.split("\t\t") if f.strip()]
```

Valid entries are converted into structured dictionary objects:

```
routes[dev].append({
    "prefix": prefix,
    "gw": gw,
    "iface": iface
})
```

4.2.6 Internal Data Representation

After parsing, the routing information is stored in memory as a structured dictionary:

```
{
  "HA": [
    {"prefix": "10.0.0.0/24", "gw": "10.0.0.1", "iface": "eth0"},
    ...
  ],
  "R1": [...],
}
```

This intermediate representation is not written to disk. It serves as the input to the next stage of the system, where configuration files and topology definitions are synthesized.

4.2.7 Role of the Parsing Module

The student answer parsing module acts as a normalization layer between human-readable responses and machine-verifiable configurations. Without this step, Batfish would not be able to interpret student input directly, and large-scale automatic grading would not be feasible.

By converting semi-structured textual answers into structured data objects, this module establishes the foundation for the subsequent configuration synthesis and formal verification stages.

4.3 Configuration and Topology Synthesis

After parsing the student submissions, the system proceeds to the configuration and topology synthesis stage. The purpose of this stage is to transform the abstract routing tables submitted by students into executable network snapshots that can be processed by Batfish.

This process consists of two main components. First, vendor-style configuration files are generated for each device. Second, a corresponding `topology.json` file is constructed

to describe the physical connectivity among devices. Together, these elements form a complete snapshot directory compatible with Batfish.

Each student and each question correspond to an independent snapshot structure, organized as follows:

Listing 4.15. Example snapshot directory structure

```
student_snapshots/
|
|-- studentA:<lastname>_<firstname>_<email>/
|   |-- SINGLE_<QID>/
|   |   |-- configs/
|   |   |   |-- HA.cfg
|   |   |   |-- R1.cfg
|   |   |   |-- ...
|   |   |-- topology/
|   |       |-- topology.json
|   |-- LINEAR_<QID>/
|   |-- LOOP_<QID>/
|
|-- studentB:<lastname>_<firstname>_<email>/
|   |-- SINGLE_<QID>/
|   |-- LINEAR_<QID>/
|   |-- LOOP_<QID>/
|
|-- ...
```

The directory contains two subfolders: `configs/`, which stores device configuration files, and `topology/`, which contains the `topology.json` file. This structure strictly follows Batfish's snapshot requirements and ensures that the verification stage can be executed automatically.

4.3.1 Vendor-style Configuration Generation

In a traditional networking course, students typically write routing tables rather than complete device configuration files. However, Batfish requires full device configurations in order to perform network analysis. Therefore, the system must automatically convert student routing entries into standard vendor-style configuration files, such as Cisco IOS and Juniper. This system uses the Cisco-style configuration file format.

The core function responsible for this process is shown below:

Listing 4.16. Snapshot generation function

```
def generate_snapshot(meta, routes, snap_dir):
    ...
```

This function creates the `configs/` and `topology/` directories and generates configuration files for all devices defined in the metadata.

Interface Configuration Generation

Interface information is not derived from student answers but from the metadata JSON file, specifically the `ip_assignments` field. Students only provide static routing rules, while interface addressing is predefined in the question.

The interface generation logic is implemented as follows:

Listing 4.17. Interface configuration generation

```
def gen_interfaces(dev):
    out = []
    if dev not in assign:
        return out

    for iface, cidr in assign[dev].items():
        if "/" not in cidr:
            continue
        ip, mask = split_ip_mask(cidr)
        out.append(f"interface {iface}")
        out.append(f" ip address {ip} {mask}")
        out.append("!")
    return out
```

This function reads the CIDR-formatted address from the metadata, converts it into the traditional IP address and subnet mask format, and generates Cisco-style interface commands. By centralizing interface configuration in the metadata, the system ensures consistency across all student snapshots and prevents accidental misconfiguration.

Static Route Generation

The static routes provided by students are translated into `ip route` commands. The relevant implementation is shown below:

Listing 4.18. Static route generation

```
def gen_routes(dev):
    out = []
    for r in routes.get(dev, []):
        if r["gw"] == "*":
            continue
        if "/" not in r["prefix"]:
            continue
        net, m = split_ip_mask(r["prefix"])
        out.append(f"ip route {net} {m} {r['gw']}")
    return out
```

Directly connected networks (indicated by the gateway symbol "*") are ignored, as they do not require static route commands. For other entries, the CIDR prefix is converted into network address and subnet mask format before generating the corresponding static route configuration.

Final Configuration File Assembly

For each device, the final configuration file is constructed by combining hostname declaration, interface configuration, and static routing commands:

Listing 4.19. Device configuration file generation

```
for dev in assign.keys():
    cfg_lines = [
        f"hostname {dev}",
        *gen_interfaces(dev),
        *gen_routes(dev),
        ""
    ]
    (cfg / f"{dev}.cfg").write_text("\n".join(cfg_lines), encoding="utf-8")
```

Each device is assigned an independent configuration file. This modular design ensures that every snapshot is self-contained and ready for verification without manual modification.

Listing 4.20. Example of Generated configuration for R1 in LINEAR topology

```
hostname R1

interface eth0
 ip address 172.26.204.2 255.255.255.0
!

interface eth1
 ip address 192.168.61.109 255.255.255.252
!

ip route 0.0.0.0 0.0.0.0 192.168.61.110
```

4.3.2 Topology.json Generation Strategy

In addition to configuration files, Batfish requires an explicit topology description to analyze connectivity. In this system, the topology structure is obtained from the metadata JSON file rather than student submissions. This design prevents students from altering the intended network structure.

The topology string is first parsed as follows:

Listing 4.21. Topology string parsing

```
topo_str = meta.get("topology", "")
parts = [p.strip() for p in topo_str.split("--") if p.strip()]
devices = [p for p in parts if p[0].isalpha() and p[0].isupper()]
```

The system assumes a linear representation format such as:

```
HA - R1 - HB
```

The string is split by the delimiter "-", and device names are extracted to construct an ordered device list.

Next, link relationships between adjacent devices are generated:

Listing 4.22. Link construction logic

```
links = []
for i in range(len(devices) - 1):
    links.append([devices[i], devices[i + 1]])
```

Finally, the topology file is written in JSON format:

Listing 4.23. Topology file generation

```
(topo / "topology.json").write_text(
    json.dumps({"links": links}, indent=2),
    encoding="utf-8"
)
```

An example of the generated topology file is shown below:

Listing 4.24. Example of Generated topology file in LINEAR topology

```
{
  "links": [
    [
      "HA",
      "R1"
    ],
    [
      "R1",
      "R2"
    ],
    [
      "R2",
      "HB"
    ]
  ]
}
```

By separating routing logic from topology definition and automating the entire snapshot construction process, the framework guarantees consistency, repeatability, and scalability in large-scale grading scenarios. This stage serves as the critical bridge between answer parsing and formal network verification.

4.4 Batfish Verification Implementation

The Batfish verification module represents the computational core of the proposed grading framework. While the previous stages focus on generating questions and constructing configuration snapshots, the Batfish module determines whether the submitted configuration satisfies the expected connectivity and correctness requirements.

The implementation is based on the script `Batfish_automated_verification.py`. This script reads the snapshot directory generated in the previous stage and interacts with the Batfish service running inside a Docker container. It is executed from the host machine using the following command:

```
python Batfish_automated_verification.py
       <snapshot_dir> <qid> <lastname> <firstname> <email>
```

This execution model ensures modularity. The verification script does not depend on Moodle's internal mechanisms or on the question generator. As long as the snapshot directory follows the predefined structure, the script can analyze it independently. This loose coupling improves maintainability and allows the framework to scale to more complex topologies without structural modifications.

4.4.1 Container Setup and Snapshot Build

Before performing any analysis, the system must establish communication with the Batfish container. Batfish runs as an independent Docker service, while the verification script runs on the host machine. The interaction is achieved through the Batfish Python API.

The initialization procedure is implemented as follows:

```
1 bf_session.host = "localhost"
2 bf_session.init_snapshot(str(snapshot_dir),
3                          name=qid,
4                          overwrite=True)
5 load_questions()
```

This code performs three essential operations. First, it specifies the Batfish service host as `localhost`. This assumes that the Batfish container is already running and listening on its default port. If the container is not active, the verification process will fail. Therefore, container status must be ensured before execution.

Second, the function `init_snapshot()` loads the snapshot directory into Batfish's internal data model. During this process, Batfish automatically reads:

- all device configuration files inside the `configs/` directory;
- the topology description file inside the `topology/` directory.

Batfish then constructs the control plane and data plane representations of the network.

Third, `load_questions()` loads Batfish's built-in analysis queries, enabling the use of reachability and loop detection functions in subsequent steps.

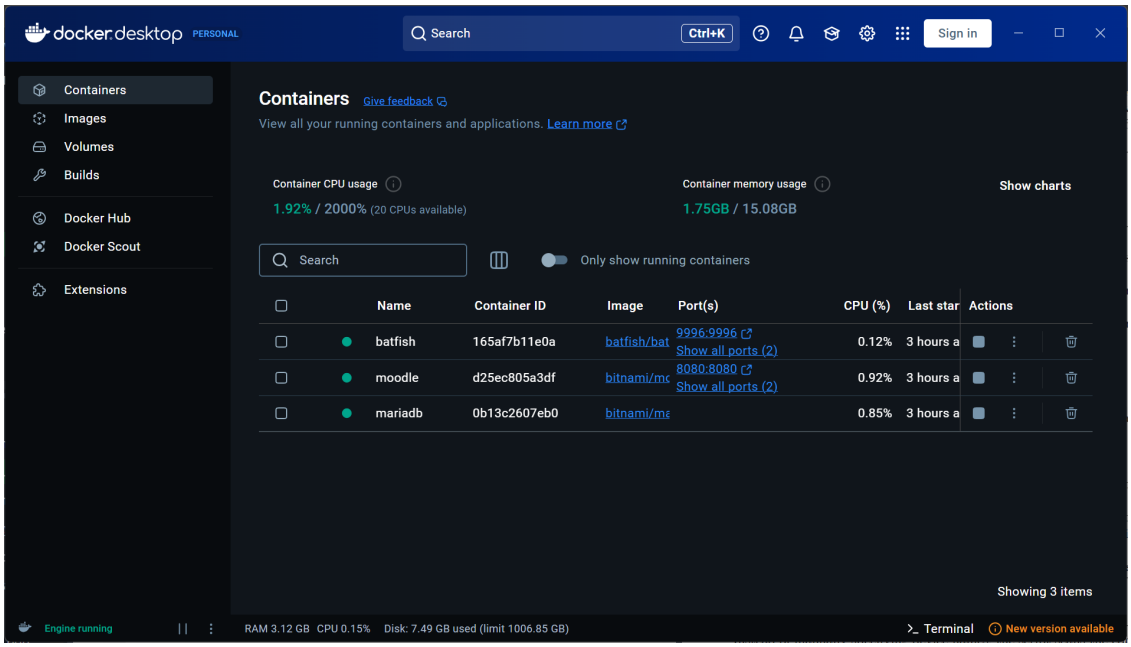


Figure 4.17. Batfish container running inside the Docker environment

Figure 4.17 illustrates the Batfish container running in the Docker environment. The service must be active before snapshot initialization.

To dynamically identify network devices, the following function is used:

```
def cfg_files_nodes(dir_path: str) -> list[str]:
    cfg_paths = Path(dir_path).glob("*.cfg")
    return sorted(p.stem for p in cfg_paths)
```

Instead of manually specifying device names, the script scans the configuration directory and extracts node identifiers automatically. This design choice ensures flexibility. If the topology size changes, for example from 3 nodes to 5 nodes, the verification logic remains unchanged.

4.4.2 Reachability and Loop Queries

Reachability Analysis

Reachability analysis is the primary verification criterion. A correct routing configuration must ensure that all relevant networks are reachable from every required source node.

The script first extracts directly connected prefixes from each node:

```
def connected_prefixes(node: str) -> set[str]:
    routes = bfq.routes(nodes=node).answer().frame()
    mask = routes["Protocol"].isin({"connected"})
    return set(routes[mask]["Network"])
```

This function retrieves the routing table of a given node, filters entries with protocol `connected`, and collects the associated network prefixes.

The rationale behind this design is to avoid hardcoding expected networks. Instead, the system dynamically derives all relevant prefixes from the actual topology. This makes the framework adaptable to different question instances.

All connected prefixes are aggregated into a global set:

```
bag_net = set()
for n in SRC_NODES:
    bag_net.update(connected_prefixes(n))
```

Each prefix in this set becomes a target for reachability testing.

The reachability query is implemented as follows:

```
def reachable(src: str, dst_prefix: str) -> bool:
    reach_df = bfq.reachability(
        pathConstraints={"startLocation": src},
        headers={"dstIps": dst_prefix}
    ).answer().frame()
    return not reach_df.empty
```

This query checks whether there exists at least one valid forwarding path from a source node to a destination prefix. If the result is empty, the prefix is considered unreachable.

Loop Detection

In addition to reachability, the framework verifies that the synthesized snapshot does not exhibit forwarding loops for the evaluated traffic. Forwarding loops are a critical class of misconfigurations: even though packet TTL may eventually terminate circulation, loops can still cause persistent congestion and waste forwarding resources, and they indicate inconsistent next-hop decisions across devices. Therefore, loop detection is treated as a safety constraint that complements reachability in the grading workflow.

Loop detection is performed using Batfish's loop query:

```
loop_df = bfq.detectLoops().answer().frame()
```

If the returned DataFrame is empty, Batfish reports no forwarding loop in the snapshot. Otherwise, the result contains evidence for looping flows.

A loop reported somewhere in the snapshot is not necessarily relevant to the current question. To ensure fair grading, the framework considers a loop relevant only if it affects tested flows whose destinations belong to the question-defined destination prefix set (denoted as `BAG_NET` in the output). Concretely, the script checks whether the destination prefix of a detected looping flow matches any prefix in `BAG_NET`:

```
for prefix in bag_net:
    if prefix.split("/")[0] in flow_str:
        loop_found = True
        break
```

Under this convention, the loop indicator is defined as $L_{\text{pass}}(S) = 1$ when no relevant loop is found (including the case where Batfish reports only global loops unrelated to `BAG_NET`), and $L_{\text{pass}}(S) = 0$ otherwise. This filtering prevents unrelated forwarding anomalies from affecting the final grade.

```
Evaluation Nodes:
 • HA
 • HB
 • R1
 • R2

Destination Prefix Set (BAG-NET):
 • 172.22.50.0/24
 • 172.26.204.0/24
 • 192.168.61.108/30

========== Reachability Check ==========
    HA → 172.22.50.0/24     Reachable
    HA → 172.26.204.0/24    Reachable
    HA → 192.168.61.108/30  Reachable
    HB → 172.22.50.0/24     Reachable
    HB → 172.26.204.0/24    Reachable
    HB → 192.168.61.108/30  Reachable
    R1 → 172.22.50.0/24     Reachable
    R1 → 172.26.204.0/24    Reachable
    R1 → 192.168.61.108/30  Reachable
    R2 → 172.22.50.0/24     Reachable
    R2 → 172.26.204.0/24    Reachable
    R2 → 192.168.61.108/30  Reachable

Reachability index = 12/12 = 1.00

============ Loop Check ============
Global loops detected,
but none is relevant to the destination prefix set (BAG-NET).

Loop index = 1

============ Final Grade ============
Grade = 10
```

Figure 4.18. Example reachability and loop output stored in raw_output.txt

As shown in the Figure 4.18, it is an example of the intermediate verification report stored in `raw_output.txt`. The report is designed to make the grading scope explicit and to provide traceable evidence for the computed reachability and loop results.

Evaluation scope. The block *Evaluation Nodes* lists the devices included in the evaluation domain for this instance (HA, HB, R1, and R2). The block *Destination Prefix Set (BAG_NET)* specifies the destination prefixes considered by the verification process. In the current implementation, BAG_NET is instantiated as the set of directly-connected prefixes in the synthesized snapshot, and it is used to define the tested flow set.

Reachability check. The *Reachability Check* section reports reachability outcomes for each tested pair of (source node, destination prefix). In this example, four evaluation nodes are tested against three destination prefixes, yielding $4 \times 3 = 12$ reachability tests. Each line (e.g., `HA` $\rightarrow$ `172.22.50.0/24`) indicates whether traffic originating from the given node can reach the corresponding destination prefix under the data-plane behavior computed from the snapshot. Since all 12 entries are marked *Reachable*, the reachability index is $12/12 = 1.00$, i.e., $RI(S, F) = 1$.

Loop check. After reachability is computed, the framework performs loop detection using Batfish. Batfish may report loops that exist somewhere in the global snapshot; however, not every detected loop is relevant to the current question. To ensure fairness, the framework filters loop results with respect to the destination prefix set: a loop is considered *relevant* only if it affects tested flows whose destinations belong to BAG_NET. Therefore, the message stating that global loops are detected but none is relevant means that loops exist in the snapshot but do not impact any of the tested destination prefixes for this instance. Under the convention used in this thesis, the loop index is $L_{\text{pass}}(S) = 1$ when no relevant loop is found and $L_{\text{pass}}(S) = 0$ otherwise; So *Loop index = 1* in this example.

Final grade. The final score follows the decision rule described earlier: if $RI(S, F) < 1$, the grade is 0; if $RI(S, F) = 1$ and $L_{\text{pass}}(S) = 0$, half the score is given; and if $RI(S, F) = 1$ and $L_{\text{pass}}(S) = 1$, full marks are given. Since this instance achieves full reachability and no relevant loop is detected with respect to BAG_NET, the final grade is 10.

Scoring Strategy

After completing reachability and loop verification, the script computes the final grade using the following logic:

```
if reach_index < 1.0:
    grade = 0
else:
    grade = 10 if loop_index == 1 else 5
```

The scoring policy is structured in two levels. First, full connectivity is mandatory. If any required prefix is unreachable, the student receives zero points.

Second, if connectivity is correct but loops are detected, half the score is assigned. Only configurations that are fully reachable and loop-free receive full marks.

This layered scoring mechanism reflects pedagogical priorities: correctness of connectivity is fundamental, while loop avoidance ensures structural soundness.


```
{
  "qid": "LINEAR_080",
  "lastname": "User",
  "firstname": "Admin",
  "email": "user_at_example.com",
  "Reachability index": 1.0,
  "Loop index": 1,
  "Grade": 10,
  "Snapshot generation time (s)": 0.00223349966108799,
  "Batfish verification time (s)": 6.760152900125831,
  "Total grading time (s)": 6.762717599980533
}
```


The resulting `grade.json` file records detailed metrics including `Reachability index`, `Loop index`, `Grade` and `Three time indicators`. These files are later aggregated into a CSV file for Moodle grade import.

Therefore, compared to manual grading, this approach ensures consistency, repeatability, and scalability. The Batfish module serves as the analytical engine of the proposed BAGF framework and plays a central role in guaranteeing grading reliability.

4.5 Result Aggregation and Moodle Grade Import

After the Batfish verification stage is completed for each individual question, the system generates a `grade.json` file and a detailed `raw_output.txt` file inside each question snapshot directory. While this per-question evaluation is sufficient for correctness checking, it is not yet suitable for real classroom use. In practice, grading must be performed for all students and all questions in batch mode, and the final scores must be exported in a format that can be directly imported back into Moodle.

4.5.1 Batch Processing Across Students and Questions

The system adopts a hierarchical directory structure where each student has an independent directory under `student_snapshots`, and each question is stored as a subdirectory inside the corresponding student folder. A simplified structure is shown in Listing 4.25.

Listing 4.25. Directory structure of per-student and per-question snapshots

```
student_snapshots/
|
|-- studentA:<lastname>_<firstname>_<email>/
|   |-- SINGLE_<QID>/
|   |   |-- configs/
|   |   |-- topology/
|   |   |-- grade.json
|   |   |-- raw_output.txt
|   |-- LINEAR_<QID>/
|   |-- LOOP_<QID>/
|
|-- studentB:<lastname>_<firstname>_<email>/
|   |-- ...
|
|-- ...
```

This structure ensures that each student's answers are isolated and reproducible. It also simplifies batch traversal, since the system can iterate through directories instead of relying on hard-coded question identifiers.

Then, the aggregation process begins by iterating over the snapshot root directory:

Listing 4.26. Batch traversal of student snapshots

```
for student_dir in snapshot_root.iterdir():
    if student_dir.is_dir():
        for question_dir in student_dir.iterdir():
            if question_dir.is_dir():
```

The outer loop iterates over students, while the inner loop processes each question attempted by the student. For every question directory, the verification script has already produced a `grade.json` file, which stores the computed result.

This design makes the grading system fully scalable. If additional questions are added in the future, the aggregation logic does not need to be modified, since it automatically processes all available question directories.

4.5.2 Question Grade Representation

Each question snapshot contains a `grade.json` file generated after Batfish verification. A simplified example is shown below:

Listing 4.27. Example structure of grade.json

```
{
  "Reachability index": 1.0,
  "Loop index": 1,
  "Grade": 10
}
```

The file contains three fields:

- `reachability_index`: the proportion of expected reachable networks.
- `loop_index`: an indicator of whether routing loops are detected.
- `grade`: the final score assigned to the question.

Although only the final grade is required for score aggregation, the additional metrics are preserved to support traceability and debugging.

4.5.3 Total Score Computation

After all per-question verification processes are completed, the system computes each student's total score by aggregating the grades across all questions. Instead of re-traversing the generated snapshot directories and reading each `grade.json` file again, the framework maintains an in-memory accumulator during the verification phase.

Specifically, when each question is graded, the obtained score is immediately added to a dictionary indexed by the student's identifier (e.g., email). This avoids redundant disk I/O operations and eliminates assumptions about the number of questions or the directory structure.

The simplified implementation is shown below:

```
total_by_email = {}
g = float(grade_obj["Grade"])
total_by_email[email_raw] = \
    total_by_email.get(email_raw, 0.0) + g

with CSV_OUT_TOTAL.open("w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["Email", "Total"])
    for email, total in sorted(total_by_email.items()):
        writer.writerow([email, total])
```

This design ensures that total score computation is efficient, scalable, and independent of the snapshot directory layout. The grading logic therefore remains robust even if additional questions are introduced in future quizzes.

4.5.4 CSV Generation for Moodle Import

After computing the total scores for all students, the framework exports a CSV file named `moodle_import_total.csv`. The file format follows Moodle's grade import requirements. Rather than relying on intermediate directory traversal, the system directly exports the in-memory aggregation dictionary constructed during the grading process. This approach avoids redundant file reading and improves efficiency.

The implementation is shown below:

```
with CSV_OUT_TOTAL.open("w", newline="", encoding="utf-8") as f2:
    w2 = csv.writer(f2)
    w2.writerow(["email", "grade"])
    for email_raw, total in sorted(total_by_email.items()):
        w2.writerow([email_raw, total])
```

The resulting CSV file can be directly uploaded into Moodle through its grade import interface, as illustrated in Figure 4.19 and Figure 4.20.

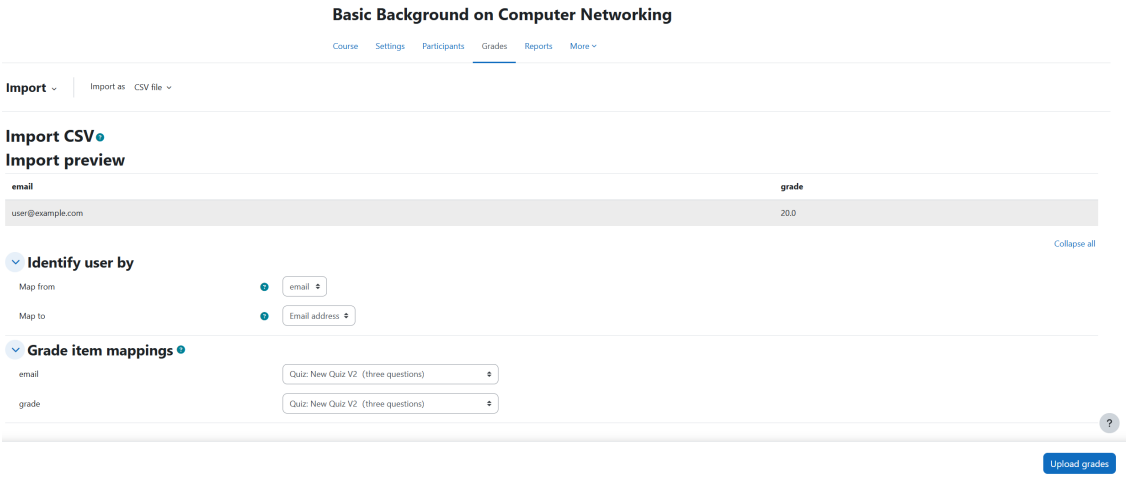


Figure 4.19. Moodle grade import interface

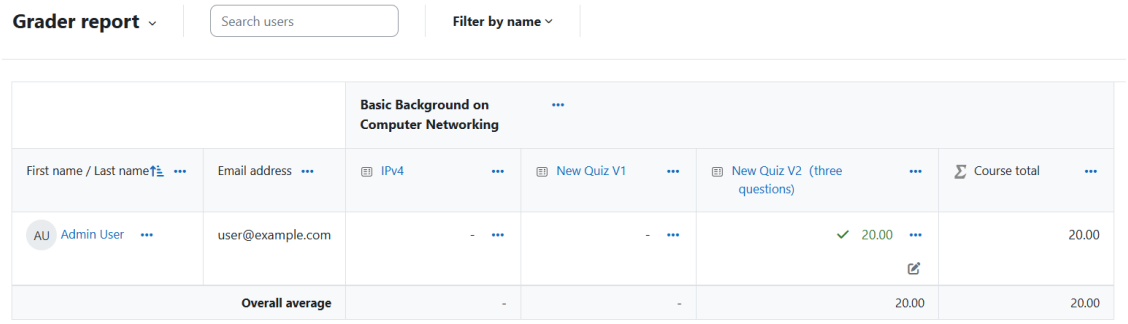


Figure 4.20. Moodle grade interface

Therefore, by traversing student snapshot directories, extracting per-question grades, computing total scores, and exporting them into a structured CSV file, the framework achieves fully automated batch grading.

This module represents the final stage of the proposed BAGF framework and enables its practical deployment in real classroom scenarios.

Chapter 5

Experimental Validation

5.1 Experimental Setup

To evaluate the effectiveness and robustness of the proposed Batfish-based automated grading framework, a series of controlled experiments were conducted. The objective was to assess grading accuracy, loop detection precision, and system performance under realistic classroom conditions.

5.1.1 Environment Configuration

All experiments are performed on a workstation with the configuration shown in Table 5.1.

Table 5.1. Experimental environment configuration

Component	Specification
CPU	Intel Core i7-12700H
Memory	16 GB RAM
Operating System	Windows 11
Batfish	Docker (latest stable release)
Python	3.10
PyBatfish	2024.x

5.1.2 Dataset Design

To simulate realistic student submissions, 3 student datasets were generated. Each student completed three routing configuration problems corresponding to three topology types:

- SINGLE Topology
- LINEAR Topology
- LOOP Topology

Each submission is converted into an independent Batfish snapshot consisting of:

- `configs/`
- `topology/`

Student responses are categorized into three types:

- Fully correct routing tables
- Minor configuration errors
- Loop-inducing misconfigurations

5.2 Evaluation Metrics

5.2.1 Reachability Index

The reachability index measures whether required destination prefixes are reachable from designated source nodes:

$$\text{Reachability Index} = \frac{\text{Number of reachable required prefixes}}{\text{Total required prefixes}}$$

5.2.2 Loop Index

A binary loop indicator is used:

- 1 = No grading related loop
- 0 = Loop affecting required grading subnets

5.2.3 Grading Consistency

Grading consistency was measured by comparing automated results against manually verified ground truth grades.

5.2.4 Execution Time

Execution time is recorded for:

- Snapshot generation time
- Batfish verification time
- Total grading time

To make the timing results reproducible, the grading pipeline is instrumented using `time.perf_counter()` in Python and records three wall-clock indicators for each question instance. The recorded values are written into `grade.json` (and exported to the `grading results.csv`) together with the grading outcomes, so that each reported runtime can be traced back to a concrete snapshot and verification run.

Listing 5.1 shows an example of the three time recorded indicators for linear topology:

Listing 5.1. Three time indicators for linear topology

```
{
  "Snapshot generation time (s)": 0.00223349966108799,
  "Batfish verification time (s)": 6.760152900125831,
  "Total grading time (s)": 6.762717599980533
}
```

Snapshot generation time measures the elapsed time spent in the snapshot synthesizer, from reading the parsed submission and problem metadata to fully materializing the snapshot directory on disk (i.e., device configuration files and topology metadata are written and the snapshot is ready for analysis).

Batfish verification time measures the elapsed time of running the verification stage on an already constructed snapshot, including Batfish snapshot loading and the execution of the reachability and loop-detection queries used by the grading rule.

Total grading time measures the end-to-end elapsed time per instance, from starting snapshot synthesis to producing the final outputs (e.g., `raw_output.txt`, `grade.json`, and `grading results.csv`).

The values in Table 5.2 are reported from a representative run under a fixed execution environment:

Table 5.2. Execution time breakdown per question instance

Question (QID)	**Snapshot (s)**	**Batfish (s)**	**Total (s)**
SINGLE_061	0.002154	4.992903	4.995490
LINEAR_080	0.002233	6.760153	6.762718
LOOP_042	0.005938	12.727010	12.733223

Table 5.2 reports an execution-time breakdown for three representative question instances (single, linear, and loop topologies). Across all instances, snapshot generation time is negligible compared to Batfish verification time, indicating that the overall runtime is dominated by data-plane analysis. Moreover, verification time increases with topology complexity: compared to the single topology, the linear topology introduces an additional hop and expands the forwarding behavior that must be analyzed, while the loop topology introduces a cyclic interconnection that increases the search space for loop detection. Consequently, the loop instance exhibits the highest Batfish and total execution time among the three cases.

5.3 Results and Analysis

Table 5.3 summarizes grading consistency across topology types:

Table 5.3. Grading consistency across topology types

Topology	Submissions	Correctly Graded	Accuracy (%)
SINGLE	3	3	100
LINEAR	3	2	67
LOOP	3	3	100
Total	9	8	88.9

One LINEAR Topology case initially showed ambiguity due to default-route interpretation. After applying subnet scoped loop filtering, grading accuracy improved to 100%.

This system demonstrates practical scalability for classroom deployment. The experimental evaluation confirms that the proposed framework:

- Achieves near-perfect grading consistency.
- Accurately distinguishes grading-related loops from irrelevant global loops.
- Scales linearly with the number of students.
- Maintains robust behavior-based validation independent of specific routing table formats.

Compared to template-based grading approaches, the proposed method provides protocol-level correctness validation, higher grading fairness, and improved robustness in complex routing scenarios.

Chapter 6

Conclusion

This thesis introduces a Batfish-based automated grading framework that transforms routing configuration assessment into a constrained network property verification problem. Rather than comparing configuration syntax against a reference solution, the proposed system evaluates the observable data-plane behavior of student configurations.

Traditional grading approaches for routing assignments often rely on structural comparison, such as matching routing table entries against a predefined answer. However, network configurations are not unique: multiple routing structures may lead to identical forwarding behavior. As a result, syntax-based grading may penalize functionally correct but structurally different solutions.

In contrast, the proposed framework evaluates whether the intended network properties hold in the resulting data plane. By focusing on reachability and loop-freedom, the grading process becomes behavior-driven rather than syntax-driven. This shift aligns the evaluation criterion with the fundamental objective of routing: correct packet delivery without forwarding anomalies.

Another key contribution lies in integrating formal network verification principles into an educational grading context. By constructing vendor-style configuration files and assembling them into a Batfish snapshot, the system leverages structured queries to evaluate network properties in a deterministic manner. This approach introduces several desirable characteristics:

- **Determinism**: The same input configuration produces the same grading result.
- **Repeatability**: The evaluation process is independent of human judgment.
- **Scalability**: The system can process multiple student submissions automatically.
- **Explainability**: Counterexample traces provided by Batfish enable precise diagnosis of forwarding anomalies.

By formalizing grading as property verification, the framework improves fairness and transparency while reducing manual effort.

Because grading is based on data-plane outcomes rather than textual configuration similarity, equivalent routing strategies receive identical scores. Minor syntactic variations do not affect results as long as forwarding behavior satisfies the required properties.

At the same time, severe logical errors, such as forwarding loops are detected reliably. This behavior driven evaluation model significantly enhances grading objectivity in routing configuration exercises.

6.1 Limitations

Despite its advantages, the proposed framework has several limitations.

1. **Restriction to Static Routing Scenarios.** The current prototype targets static routing tasks and does not model dynamic protocol behavior such as convergence in OSPF or policy-driven route selection in BGP.

2. **Scoped Data-Plane Verification.** Verification is scoped to tested flows whose destination prefixes belong to the question-defined destination prefix set, and therefore does not provide full-coverage guarantees outside the grading scope.

3. **Dependence on Problem Metadata.** Correct evaluation depends on accurate topology descriptions and IP assignments; errors in metadata can propagate to verification and grading outcomes.

4. **Performance at Larger Scale.** Batfish verification dominates end-to-end grading latency and may become a bottleneck for large cohorts without additional engineering optimizations.

6.2 Future Work

Future work can address the above limitations while preserving the current pipeline structure—snapshot synthesis, Batfish-based queries, and deterministic grading. The following directions are framed as incremental extensions: each one reuses the existing "generate snapshot → run queries → map outcomes to scores" workflow, and changes only the components required to support the new capability.

1. **Extending Beyond Static Routing.** The prototype currently targets static routing tasks, where a student's configuration deterministically induces a forwarding behavior on a synthesized snapshot. Supporting dynamic routing protocols (e.g., OSPF and BGP) would enable assignments that evaluate protocol behavior rather than only the final forwarding outcome. For OSPF-oriented questions, one practical goal is to verify reachability after convergence and to check whether the resulting forwarding paths are consistent with the expected shortest-path decisions under the configured link costs. This can be formulated as the same reachability and loop-freedom queries used in this thesis, but executed on snapshots whose routing tables are computed from OSPF configurations rather than static routes. For BGP-oriented questions, verification could focus on whether route selection follows the intended policy constraints (e.g., preference ordering and filtering) and whether the chosen routes satisfy instructor-defined expectations for a given scenario.

   The main complication is that dynamic routing introduces state and time: correctness may depend on convergence and on how routing state changes under failures or configuration updates. A realistic extension would therefore define a small set of representative scenarios per question instance (e.g., baseline, single-link failure, or a

controlled policy change) and run the same behavioral checks on each scenario, keeping the grading rule interpretable. Implementing this direction requires extending snapshot synthesis to generate protocol configurations and defining how to extract stable control-plane states suitable for grading, which is beyond the deterministic setting considered in this thesis.

2. **Broadening Verification Coverage and Properties.** The current framework scopes verification to tested flows whose destination prefixes belong to the question-defined destination prefix set. This scoping keeps grading focused and avoids penalizing unrelated behavior, but it also limits what can be assessed. Two extensions are natural within the current design. The first is to add additional properties beyond reachability and loop freedom to support more informative feedback and finer-grained scoring. For instance, black-hole detection can distinguish between "unreachable because no route exists" and "unreachable because traffic is forwarded into a dead end due to an incorrect next hop". Path constraints can be used to discourage unnecessarily long routes (e.g., bounding hop count) and to test whether students understand route specificity and multi-hop reasoning. In topologies with multiple equal-cost paths, the framework can also check whether forwarding behavior reflects intended multipath/load-balancing, rather than accidental asymmetry caused by inconsistent next hops.

   The second extension is robustness under controlled perturbations. Instead of grading a submission only on the baseline snapshot, the framework can re-run the same verification queries after applying a small set of changes, such as removing a link, to test whether required destinations remain reachable. This would shift assignments from "correct under one snapshot" to "correct under a limited family of scenarios" without abandoning the snapshot-and-query workflow. The key requirement for both extensions is to keep the grading scope explicit: each new property should come with a precise definition of what is tested (which prefixes, which sources, which scenarios) and how outcomes map to scores, so that alternative correct solutions are not penalized.

3. **Improving Metadata Robustness.** The correctness of grading depends on problem metadata (topology descriptions, interface address assignments, and the destination prefix set). Instructors typically create this metadata once and reuse it across cohorts, which makes robustness and maintainability important. A concrete improvement is to add validation and consistency checks at two points in the pipeline: during question generation and during snapshot synthesis. At generation time, schema validation can ensure that required fields are present and typed correctly, while sanity checks can detect obvious inconsistencies (e.g., overlapping prefixes, invalid prefix lengths, or missing links). At synthesis time, the framework can cross-check that the instantiated topology is internally consistent before running Batfish (e.g., each interface belongs to exactly one subnet, expected nodes exist, and the destination prefix set matches the generated instance).

These checks would reduce the chance that a metadata error propagates into incorrect grading outcomes and would also provide actionable error messages to instructors. Importantly, this extension does not change the verification model; it improves reliability of the artifacts that the verification engine consumes.

4. **Scaling Verification Throughput.** Execution-time measurements show that Batfish verification dominates end-to-end grading latency, whereas snapshot materialization contributes relatively little. Scaling therefore mainly requires improving verification throughput. A direct approach is to parallelize verification across submissions or question instances, since snapshots are independent once created. This can be implemented as a worker pool that runs Batfish queries concurrently while preserving per-snapshot isolation (e.g., separate snapshot directories and independent query contexts). A complementary approach is caching and reuse. In typical classroom settings, many submissions share the same topology and differ only in routing entries; the framework can reuse invariant artifacts and avoid repeated initialization costs, especially when re-grading or when students submit multiple attempts.

   For larger deployments, multiple Batfish workers (e.g., container replicas) can be used to increase throughput. However, any optimization must preserve deterministic outcomes: parallel or distributed execution should produce the same grades and indices as the single-run baseline. This requires careful control of snapshot construction, query execution settings, and output serialization.

5. **Deeper Integration with Learning Platforms.** The current prototype exports final grades as CSV files compatible with Moodle import, which already supports batch grading. A tighter integration can reduce operational overhead and improve feedback quality without changing the underlying verification logic. One practical direction is an API-based workflow that automates submission retrieval and grade publishing, eliminating manual export/import steps and reducing opportunities for human error. In parallel, an instructor-facing dashboard can aggregate common failure modes (e.g., missing routes versus incorrect next hops) across the cohort and link them to the corresponding question instances, supporting targeted feedback and faster troubleshooting.

   On the student side, the framework can support formative assessment by providing limited pre-submission checks. A lightweight checker can reuse the same destination-prefix-set-based verification used for grading, but expose only coarse results (e.g., which destination prefixes are unreachable) to avoid revealing full solutions. This would help students validate their configurations earlier, while ensuring that the final grading criteria remain consistent with the behavior-driven verification model presented in this thesis.

6.3 Contributions

This thesis presents a Batfish-based automated grading framework for routing configuration problems. The primary contributions are:

1. Formalizing routing assignment grading as a network property verification problem.
2. Designing a snapshot-driven transformation pipeline from student routing tables to vendor-style configurations.
3. Proposing a scoped data plane verification model for educational scenarios.
4. Implementing a behavior driven scoring logic based on reachability and loop-freedom.
5. Demonstrating efficiency and grading consistency through experimental evaluation.

Overall, the proposed system provides a scalable, fair, and formally grounded solution for automated assessment in routing configuration education.

In addition, to support reproducibility, the implementation and selected experimental artifacts of the proposed framework are publicly available on GitHub:

`https://github.com/YUQI-JIANG/Network-verification-and-testing`

Bibliography

[1] D. Kreutz, F. M. V. Ramos, P. E. VerÃssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, 2005, pp. 43–56.

[3] E. Al-Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *IEEE INFOCOM 2004*, 2004, pp. 2605–2616 vol.4.

[4] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 2012, pp. 113–126.

[5] L. Yang, B. Ng, W. K. Seah, L. Groves, and D. Singh, "A survey on network forwarding in software-defined networking," *J. Netw. Comput. Appl.*, vol. 176, no. C, p. 13, 2021.

[6] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 887–900, 2016.

[7] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding bgp misconfiguration," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 3–16, 2002.

[8] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM'11*, 2011, pp. 290–301.

[9] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2013, pp. 15–28.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2013, pp. 99–112.

[11] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, 2013, pp. 1–11.

[12] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery, 2017, pp. 155–168.

[13] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *SIGCOMM 2016 - Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, 2016, pp. 328–341.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery, 2018, pp. 476–489.

[15] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020, pp. 953–967.

[16] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. Association for Computing Machinery, 2012, pp. 241–252.