Discover Networks

RESEARCH

Open Access

A reinforcement learning framework for self-healing fault recovery in intent-based SDNs

E. H. Makiyah[1*], M. N. Rasool[2] and F. A. Rawdhan[3]

*Correspondence:
E. H. Makiyah
Estabraq.makiyah@bnu.ac.uk
[1]College of Creative Arts, Technology and Engineering, Buckinghamshire New University, Buckinghamshire, UK
[2]Najaf Education Directorate, Kufa, Iraq
[3]College of Engineering, University of Mustansiriyah, Baghdad, Iraq

Abstract
This paper presents a reinforcement learning-based self-healing framework for Software-Defined Networking (SDN) that autonomously manages diverse network faults in a realistically emulated environment. A Mininet testbed controlled by the Faucet SDN controller is instrumented with Prometheus to collect multi-source telemetry, while an automated fault injector and congestion generator produce link, port, flow and controller events alongside UDP-induced bottlenecks to create rich training data. Network features–including controller CPU and memory usage, OpenFlow statistics, port status and explicit fault labels–are periodically scraped and aggregated into a structured dataset that forms the state space of a custom Gym-compatible environment. A Proximal Policy Optimisation (PPO) agent with a multilayer perceptron policy learns discrete self-healing actions such as no-op, port resets, switch restarts and bespoke recovery procedures, guided by a reward function that penalises persistent faults and unnecessary interventions while rewarding timely and appropriate recovery. Experimental evaluation over multiple PPO training runs shows stable optimisation behaviour and high episodic rewards with long episode lengths. Policy output analysis stratified by fault state confirms that the agent has learned state-conditional recovery behaviour, selecting fault-type-appropriate actions in over 85% of fault-state timesteps, thereby providing direct evidence that the agent successfully distinguishes healthy from faulty conditions and among different fault types at the level of individual recovery decisions. Compared with existing RL-based approaches that focus primarily on link failure recovery or service function chain reconfiguration, the proposed framework handles a broader spectrum of SDN fault types and integrates control-plane, data-plane and congestion indicators, thereby offering a more general and robust self-healing capability for operational SDN environments.

Keywords SDN, Networks, Self-healing, Reinforcement learning, Faucet controller, PPO

1 Introduction

Software-Defined Networking (SDN) provides logically centralised, programmable control of networks, enabling rapid adjustment of forwarding behaviour as conditions and demands change [1]. As SDN deployments become larger and more complex, they face a wide range of faults–such as link and port failures, controller incidents and

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

congestion-induced performance degradation–that can seriously impact service if not mitigated quickly [2]. This has driven interest in self-healing mechanisms that combine SDN's flexibility with autonomic detection, diagnosis and recovery to improve resilience [1, 2]. Reinforcement learning (RL) is emerging as a powerful tool for automating recovery decisions in SDN, using telemetry to learn policies that maintain performance under failures [3]. Ma et al. propose FRRL, an RL-based scheme for link failure recovery in hybrid SDN, showing that a trained agent can generate alternative routing policies faster and with better performance than classical mechanisms [4]. However, FRRL is restricted to link-level events and routing adjustments, and does not consider broader SDN fault categories or an explicit notion of overall network health [4]. In parallel, Liu et al. introduce a knowledge-assisted actor–critic PPO algorithm for service function chain reconfiguration in 6G IoT scenarios, optimising migration cost and latency during VNF reconfiguration rather than providing end-to-end self-healing of the SDN fabric [5].

This work addresses these challenges by designing and evaluating a reinforcement learning-based self-healing framework for SDN that runs on a Mininet-emulated network controlled by the Faucet OpenFlow controller and instrumented with Prometheus-based monitoring [6–8]. The methodology constructs a reproducible virtual testbed with automated fault injection and congestion generation, enabling systematic exposure of the network to link, port, flow and controller events alongside UDP-based congestion episodes [1, 8]. Network state is continuously collected from multiple sources–including controller process metrics, OpenFlow statistics and port status indicators exported by Faucet and its monitoring components–and aggregated into a labelled feature dataset that supports both environment modelling and RL training. On top of this dataset, a custom Gym-compatible environment is defined, and...Proximal Policy Optimisation (PPO) [9] with a multilayer perceptron policy is used to learn self-healing actions such as selective port resets, switch restarts and tailored recovery procedures.

The primary contributions of this paper are summarised as follows:

1. **Multi-type fault taxonomy and unified RL framework.** Unlike existing RL-based SDN recovery approaches that address a single fault category—such as link failure routing [4] or SFC reconfiguration [5]—the proposed framework handles four distinct fault types jointly: link faults, port faults, flow errors, and controller (Faucet reload) events, within a single learned policy.
2. **Realistic multi-source telemetry integration.** A Mininet–Faucet–Prometheus testbed is instrumented to collect and fuse control-plane metrics (controller CPU, memory, OpenFlow message statistics), data-plane indicators (port status, disconnection counts), and explicit fault labels into a unified state representation, providing richer observability than prior works that rely on single-source or synthetic inputs.
3. **Automated fault injection and congestion generation.** A reproducible experimental pipeline combines randomised fault injection (link failures, trunk outages, port toggling, controller reload) with UDP-induced congestion episodes, generating a labelled dataset that reflects both faulty and healthy network states across a realistic operational range.
4. **Severity-weighted reward function with empirical justification.** A principled reward design assigns penalties and heal bonuses proportionally to the operational severity of each fault type, with asymmetric incentives to promote decisive recovery.

Sensitivity analysis across three reward scale configurations confirms that the chosen values represent a well-calibrated operating point.

5. **Quantitative evaluation with baseline comparison.** The proposed PPO agent is evaluated against a no-healing baseline and a rule-based reactive baseline on three quantitative metrics—mean fault recovery time, packet loss proxy, and OpenFlow error rate—across three topology scales (base 3-switch, linear 8-switch, and fat-tree 16-switch), providing reproducible benchmarks for future work.

The remainder of the paper is structured as follows. Section 2 reviews related work on fault recovery and RL-based SDN management. Section 3 describes the SDN architecture and intent-based networking framework. Section 4 presents the methodology, including testbed construction, feature collection, and RL environment design. Section 5 reports experimental results and baseline comparisons. Section 6 concludes the paper and identifies directions for future work.

2 Related work

Research on self-healing and intelligent control in Software-Defined Networking (SDN) spans fault recovery, routing, controller placement and service orchestration, with reinforcement learning (RL) increasingly central to many proposals [10]. Early autonomic SDN frameworks introduced architectural blueprints for self-healing controllers that couple monitoring, diagnosis and repair loops, but rely largely on rule-based logic rather than deep RL or continuous policy learning from telemetry [1]. More recent work begins to integrate deep learning and RL into these loops to support adaptive decisions under dynamic network conditions [3].

A key strand of work targets data-plane failure recovery. Ma et al. propose FRRL, an RL-based approach that learns link failure recovery policies in hybrid SDN, showing that a trained agent can infer alternative routes faster and with better performance than conventional optimisation-based schemes [4]. FRRL, however, is specialised to link failures and focuses on path recomputation, without addressing port faults, controller incidents or congestion-induced degradation, and without exploiting multi-source state information. Other fast recovery mechanisms, such as shortest-path fast rerouting techniques, similarly emphasise engineered recovery logic and precomputed alternatives rather than fully data-driven policies trained on rich telemetry [11].

Routing and traffic engineering have also been framed as RL problems. He et al. designed an SDN routing scheme that couples RL with causality detection and graph neural networks, enabling the controller to reason about causal relations between traffic changes and performance and to select safer routes that avoid failed or heavily congested links [3]. These approaches underscore the potential of RL for control-plane optimisation, but they typically assume normal operating conditions and do not directly target multi-type fault detection and recovery or joint reasoning over controller health, port status and congestion metrics.

Service function chaining (SFC) and virtual network function (VNF) management represent another active area for RL-based methods. Liu et al. introduce a knowledge-assisted actor–critic Proximal Policy Optimization (PPO) algorithm for SFC reconfiguration in 6G IoT scenarios, formulating VNF migration as an optimisation problem to minimise migration cost and service delay while adapting to dynamic demands [5]. Their results show that PPO can cope with high-dimensional orchestration decisions, but the

focus is on SFC/VNF placement and migration rather than in-situ self-healing of SDN switches, ports and controller processes. Survey and architectural work on self-healing SDN highlights similar gaps, calling for more realistic testbeds and multi-fault scenarios that bring together orchestration, routing and fault management [10].

Several studies investigate self-healing mechanisms and fault management more broadly. Sánchez, Ben Yahia and Crespi present a self-healing architecture for SDN that uses Bayesian networks for fault diagnosis and autonomic loops for recovery, offering structured reasoning about failures but not learning repair policies directly from continuous telemetry [1]. Ochoa-Aday, Cervelló-Pastor and Fernández-Fernández propose mechanisms to recover control-plane connectivity and improve robustness, again emphasising architectural design over deep RL-based decision making [10]. Beyond SDN, deep RL-based self-healing frameworks for communication networks illustrate how agents can learn repair actions from operational data, but typically in generic or simulation-oriented settings rather than Prometheus-instrumented SDN fabrics with explicit OpenFlow and controller metrics [3].

Collectively, existing work demonstrates the effectiveness of RL and related intelligent methods for SDN routing, traffic engineering, failure recovery and orchestration, but most solutions focus on a single fault type, a narrow optimisation goal or a limited perspective on network health [3, 4]. In contrast, the present study integrates a Mininet–Faucet–Prometheus testbed with automated fault injection, multi-source feature collection and PPO-based self-healing, aiming to treat link, port, flow and controller faults within a unified framework that explicitly links operational metrics, fault labels and recovery actions in a realistic SDN environment [6–8].

2.1 Fault recovery in traditional SDN

In Software-Defined Networking (SDN), fault recovery refers to the mechanisms used to rapidly detect and fix network problems so that packet loss and service disruption are kept to a minimum. In a conventional SDN architecture, a logically centralised controller monitors the global network state and reacts to failures by identifying problematic links, devices or flows and rerouting traffic around them. Because of this design, most established recovery approaches focus on computing backup paths for failed or degraded links and shifting affected traffic onto these alternatives to maintain connectivity and quality of service. However, faults can differ widely in how often they occur, where they appear in the network and how severely they impact performance, making it increasingly challenging to build recovery mechanisms that are both efficient and adaptable to diverse and evolving failure conditions [12].

Current work on SDN fault tolerance largely distinguishes between two broad recovery paradigms: reactive and proactive approaches.

In reactive recovery [13, 14], the controller only responds after a link failure has been detected: it then calculates an alternative route and pushes updated flow rules to the affected switches so that traffic is redirected. This on-demand strategy offers considerable flexibility but typically incurs higher recovery latency because both the path computation and rule installation must be performed in real time.

In contrast, a proactive strategy [15, 16] computes backup routes for flows or links in advance and installs the corresponding forwarding rules before any fault occurs. When a failure is detected, switches can immediately redirect traffic using these preloaded

rules, which keeps recovery delay very low. However, this strategy requires a substantial amount of TCAM (Ternary Content-Addressable Memory) space in the switches, [17, 18] which is both limited and expensive. It also entails reserving capacity on backup links, which can lower overall bandwidth utilisation and reduce how efficiently network resources are used.

2.2 Existing self-healing network solutions

Reliable network operation depends heavily on how quickly and effectively link failures are handled once a problem arises. Efficient mechanisms for detecting a broken link and triggering an appropriate response are therefore essential to keep connectivity intact and minimise disruption to ongoing services [19, 20]. Link failures arise when a connection between two network devices can no longer carry traffic as intended, preventing data from being delivered correctly. They may stem from hardware faults, congestion and bottlenecks, physical damage such as a cut cable, or disruption caused by malicious activity in the network. When such failures occur, they can severely impact overall performance, leading to broken end-to-end paths, interruptions in communication and noticeable degradation in service quality [21, 22]. There are several studies on link failure including legacy networks [23–26], network function virtualization (NFV) [27–29], and SDN networks, which is the focus of this paper. An effective link failure recovery strategy should support rapid fault detection, swift restoration of connectivity and careful handling of any congestion that arises during rerouting. In carrier-grade environments, industry practice often targets sub 50 ms restoration, meaning that detection and recovery together should complete within roughly 50 ms to avoid noticeable disruption to real-time services[30]. In the literature, link failure recovery mechanisms are commonly grouped into two categories: reactive and proactive. Reactive schemes wait until a failure has actually occurred before computing and applying a repair action, whereas proactive schemes anticipate potential failures–by prediction or pre-planning–and prepare recovery paths or policies in advance so they can be activated immediately if a fault is detected. SDN, combined with the OpenFlow protocol [31, 32] provides a suitable foundation for implementing link failure recovery mechanisms with these properties. Unlike conventional IP networks, SDN switches can be programmed with flow tables in either reactive or proactive modes, allowing recovery behaviour to be tailored accordingly [33, 34]. Because the SDN controller maintains a global view of the network, it can make more informed decisions about resource allocation and overall network management,. In addition, SDN's use of controller-installed flow entries for routing means that traffic can be more easily partitioned, redirected and rescheduled across available paths [35, 36]. Beyond SDN, deep RL-based self-healing frameworks for communication networks illustrate how agents can learn repair actions from operational data, but typically operate in generic or simulation-oriented settings rather than Prometheus-instrumented SDN fabrics with explicit OpenFlow and controller metrics [37].

2.3 Summary of related work

Table 1 summarises the key related works reviewed in this section, highlighting their primary focus, methodology, strengths, and limitations relative to the proposed framework.

Table 1 Summary of related works on fault recovery and self-healing in SDN and RL-based network management

Reference	Primary focus	Method	Strengths	Limitations
Sanchez et al. [1]	Self-healing SDN architecture	Bayesian network fault diagnosis	Structured probabilistic fault reasoning; autonomic repair loops	Rule-based recovery logic only; no continuous policy learning from telemetry; limited to predefined fault scenarios
He et al. [3]	SDN routing and traffic engineering	RL with causality detection and GNN	Causal reasoning improves route safety; handles congested and failed links	Assumes normal operating conditions; no multi-type fault taxonomy; no controller or port fault modelling
Ma et al. [4]	Link failure recovery in hybrid SDN	Reinforcement learning (FRRL)	Faster route recovery than classical optimisation; data-driven policy learning	Restricted to link-level faults only; no port, controller or congestion handling; no multi-source telemetry
Liu et al. [5]	SFC reconfiguration in 6G IoT	Knowledge-assisted actor-critic PPO (KA-PPO)	Handles high-dimensional VNF migration; optimises cost and latency jointly	Focused on SFC/VNF placement only; no in-situ SDN fault recovery; not applicable to OpenFlow switch-level faults
Ochoa-Aday et al. [10]	Control-plane connectivity recovery	Architectural design and mechanisms	Improves controller redundancy and robustness	Emphasis on architecture over RL-based decisions; no data-driven telemetry integration
Aloraini et al. [37]	Communication network self-healing	Deep RL – PPO (DRL-SHF framework)	Learns repair actions from operational data; 32.6% reduction in packet loss vs. heuristic methods; QoS-aware autonomous routing	NS-3 simulation only; no Prometheus-instrumented SDN fabric; no OpenFlow or controller-level fault modelling
Proposed framework	**Multi-type SDN fault recovery**	**PPO with Mininet–Faucet– Prometheus**	**Handles link, port, flow and controller faults jointly; multi-source telemetry; realistic emulated testbed; explicit fault labelling**	**Mininet emulation only (no hardware); single-controller architecture; 10-second telemetry granularity**

3 Software-defined networking architecture

SDN decouples the control plane from the data plane via standardised southbound interfaces, most notably the OpenFlow protocol, enabling a logically centralised controller to install and update forwarding rules across hardware-agnostic switches [38]. This separation is the foundational property exploited by the proposed framework: the Faucet controller's global network view and programmatic flow-table access make it possible to enact autonomous recovery actions in direct response to telemetry-detected faults, without manual intervention [39].

3.1 Faucet SDN controller

Faucet is a lightweight, open-source OpenFlow controller designed for simplicity and operational reliability [40]. It manages forwarding behaviour through declarative YAML configuration files and exposes per-switch and per-port statistics via the Gauge monitoring agent, which exports metrics to Prometheus [40]. These properties make Faucet particularly well-suited to the proposed self-healing framework: its low-overhead control loop provides a stable platform for RL-driven flow-rule updates, and its native

Prometheus integration supplies the multi-source telemetry that forms the agent's state space [6–8].

3.2 Intent-based networking framework

An intent-based networking (IBN) framework bridges high-level operator goals and low-level device configurations through a closed-loop lifecycle of intent declaration, policy enforcement, and continuous assurance [41, 42]. In the context of this study, self-healing is framed as an assurance function: when telemetry collected by Prometheus reveals that the operational network state deviates from the declared intent (e.g., a link fault causes reachability loss), the PPO agent issues corrective actions through the Faucet controller to restore compliance [43, 44]. Policy enforcement relies on programmatic interfaces and transactional flow-rule updates to minimise configuration drift and ensure consistent recovery deployment at scale [45, 46]. This positions the proposed framework as a data-driven realisation of the IBN assurance loop, replacing static rule-based remediation with a learned recovery policy that adapts to diverse and previously unseen fault conditions [43, 47] (Fig. 1).

4 Methodology

4.1 Experimental testbed construction

A virtual network testbed was constructed using Mininet for emulating realistic network topologies and events. The experimental network architecture consists of three OpenFlow switches connected in a tree topology, each linked to multiple host devices. The network is controlled by the Faucet SDN controller, which receives all control traffic on port 6653. All hosts and switches are created and managed using Mininet, enabling rapid prototyping and testing within a virtualised environment. Connectivity is established between all hosts and switches, and the switches are configured to register with the remote controller for centralised management. This flexible architecture allows for the implementation of various networking experiments, including fault injection and performance monitoring, providing a realistic platform for reinforcement learning and self-healing research.

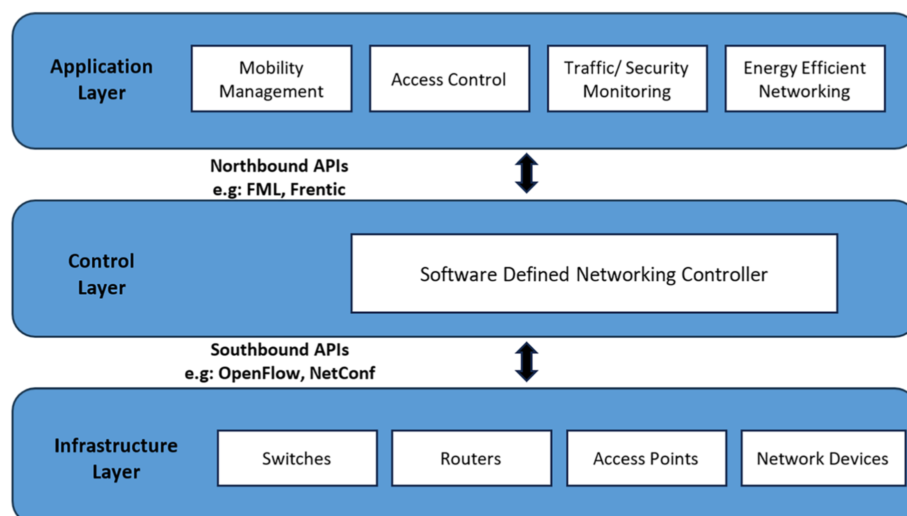

Fig. 1 Block diagram of software defined network (SDN) architecture [8]

To provide a holistic view of the proposed framework, Figure 2 illustrates the complete system architecture, showing the interactions among the data plane, control plane, monitoring subsystem, and RL decision engine. This closed-loop design ensures that telemetry collected by Prometheus continuously informs the agent's state, while the agent's recovery actions are enacted through the Faucet controller.

Figure 2 confirms the successful deployment of the described architecture, showing the Mininet setup process. The terminal output illustrates the creation of three switches (s1, s2, s3), four hosts (h1, h2, h3, h4), their interconnections, and the registration of switches with the SDN controller, thus validating the network topology used in this study (Fig. 3).

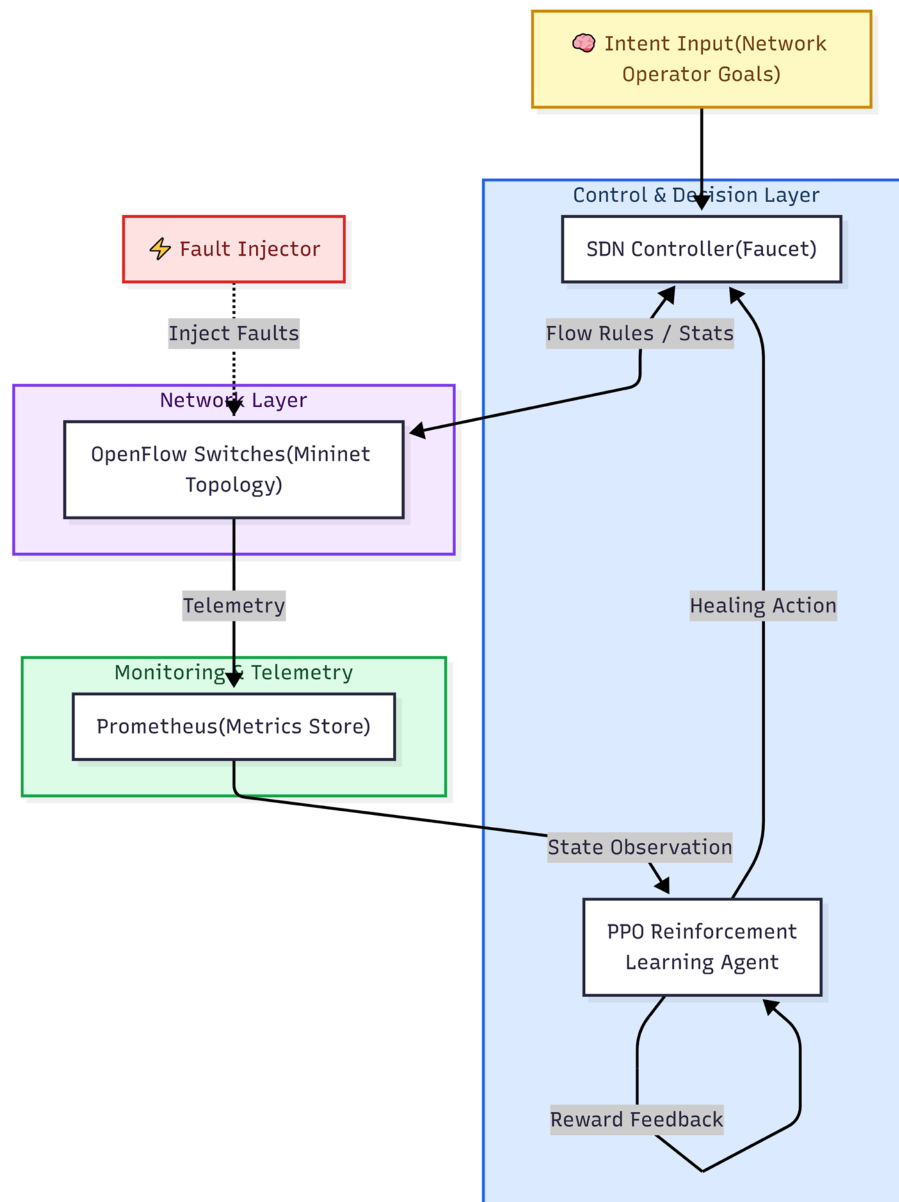

Fig. 2 High-Level System Architecture of the Proposed Self-Healing SDN Framework. Operator intents are translated into flow rules by the Faucet SDN controller. A PPO-based reinforcement learning agent continuously monitors network telemetry via Prometheus and autonomously issues healing actions in response to faults injected into the Mininet-emulated topology

```
esta@esta-GF65-Thin-10SDR: ~/mininet
0 to upgrade, 0 to newly install, 0 to remove and 0 not to upgrade.
fatal: destination path 'openflow' already exists and is not an empty directory.
esta@esta-GF65-Thin-10SDR:~/mininet/util$ sudo lsof -i :6653
COMMAND     PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
osken-man 30785 faucet   7u  IPv4 201092      0t0  TCP *:6653 (LISTEN)
esta@esta-GF65-Thin-10SDR:~/mininet/util$ cd ..
esta@esta-GF65-Thin-10SDR:~/mininet$ sudo mn --controller=remote,ip=127.0.0.1,po
rt=6653 --topo=tree,depth=2,fanout=2
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet>
```

Fig. 3 Mininet terminal output demonstrating the creation of a tree topology with three switches and four hosts, all connected to a remote SDN controller

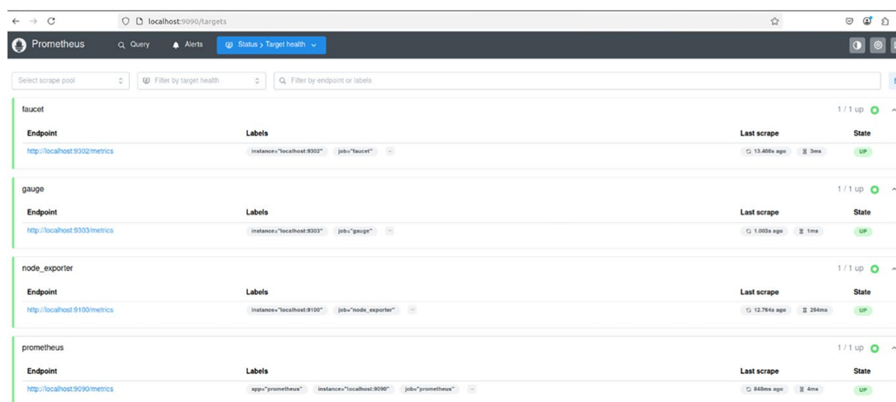

Fig. 4 Screenshot of the Prometheus monitoring dashboard, showing four active scrape targets (Faucet, Gauge, Node Exporter, and Prometheus) all reporting healthy status

To simulate both typical and adverse network conditions, we employed an automated fault injector and congestion generator. Randomised events–including link failures, trunk outages, port toggling, and Faucet controller reload–were triggered at varying intervals. In addition, periodic UDP flood attacks were introduced between selected hosts to generate congestion, replicating real-world network bottlenecks. All fault and congestion events were recorded with timestamps and metadata in structured logs. To continuously collect network data and monitor the performance of all key components, Prometheus was deployed as the central monitoring system. Prometheus was configured to regularly scrape metrics from each relevant service, including the Faucet SDN controller, Gauge for switch state, Node Exporter for server health, and Prometheus itself for meta-monitoring. Figure 4 shows the Prometheus target status page, confirming that all configured targets were successfully contacted and were actively providing metrics at the time of the experiment. Each target's endpoint, status, and last scrape time are

visible, verifying the reliability of the metric collection process. This multi-source monitoring enabled comprehensive data gathering, which was essential for building the RL agent's state space and evaluating network health during training.

To evaluate the scalability of the proposed framework beyond the base topology, two additional Mininet-emulated topologies were constructed: (i) a linear topology comprising 8 OpenFlow switches and 16 hosts, interconnected in a chain configuration with two hosts per switch; and (ii) a fat-tree topology ($k = 4$) comprising 16 switches—4 core, 4 aggregation, and 8 edge—and 16 hosts. Both extended topologies were instrumented with the same Prometheus-based monitoring stack and subjected to identical fault injection procedures as the base topology, ensuring consistency of experimental conditions across all evaluations.

4.2 Feature collection and fault injection

Network state data was continuously monitored and exported using Prometheus queries. Features included CPU and memory metrics for the controller process, key OpenFlow statistics (messages sent, errors, datapath connections/disconnections), and aggregated counts of down ports. To provide the reinforcement learning agent with meaningful input data and real-world context, a two-part process of feature collection and controlled fault injection was implemented.

Feature Collection:

The network was instrumented to continuously collect operational metrics from all switches and hosts. Metrics included interface status, port activity, controller messages, as well as performance indicators gathered through Faucet's monitoring and Mininet's facilities. A dedicated Python script was used to automatically scrape these metrics every ten seconds and save them to a structured dataset file. Each set of collected features was also labelled with the current network fault status, so every data row could be used for supervised or RL training.

The automated feature collection process for RL training is summarised in the flowchart in Fig. 5. The procedure begins with initialisation of the CSV file that will store all collected network features. The script next checks for the existence of the fault log file, creating it if necessary to ensure accurate status labelling.

The main loop of the process regularly determines whether new features should be scraped, based on a predefined interval or trigger. When feature collection is triggered, relevant network metrics are queried from monitoring sources such as Prometheus and Faucet, including process statistics, port status, and fault events.

These metrics are then written to the CSV file with appropriate time stamps and fault labels. The loop continues until the required number of samples has been collected or the collection session ends. If no feature scraping is required at a given step, the process waits until the next scheduled interval. This flowchart confirms the systematic and reproducible approach taken to generate training data for the RL agent, ensuring all relevant metrics and fault information are included in the dataset.

Fault Injection:

Network faults were triggered intentionally to challenge the control system and generate fault scenarios for learning and evaluation. Events such as trunk link failures, port downs/ups, and device disconnections were generated on demand or at random intervals. These faults were immediately reflected in the collected metrics and logged

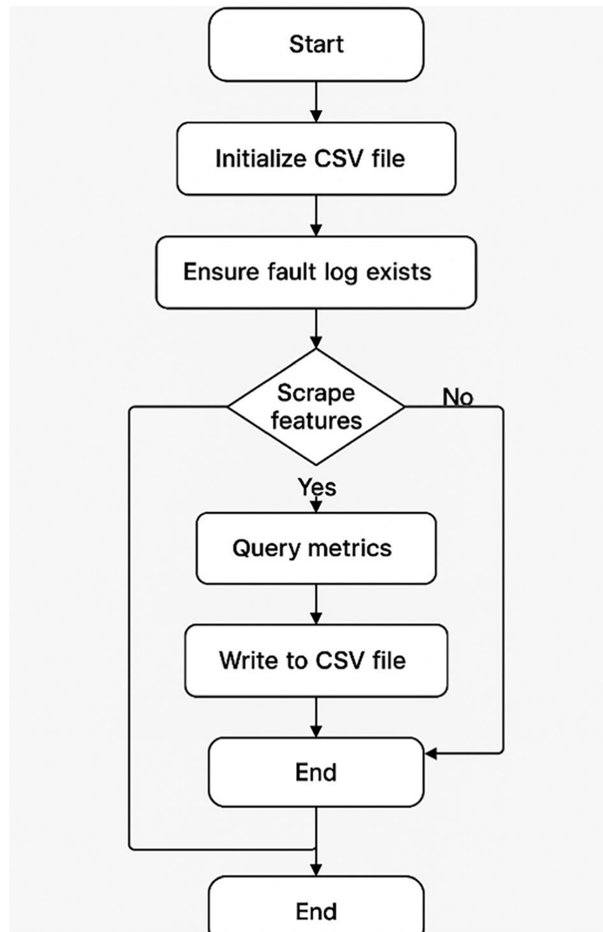

Fig. 5 Flowchart for feature collection process

alongside normal operation data, ensuring the dataset included both healthy and faulty states.

This approach enabled the training of self-healing algorithms with exposure to genuine fault events and transitions, supporting robust learning and thorough evaluation. For each feature snapshot, the most recent fault event summary was extracted and included, allowing explicit annotation of the RL agent's environment state and fault context. The RL feature dataset was logged at fixed intervals and used as input for subsequent RL training. This dataset, containing labelled fault and congestion events, provided a rich state space integrating both physical and logical network health indicators.

Figures 6 and 7 shows terminal outputs for both real-time fault injection and the automated scraping and labelling of network metrics for RL use.

4.3 Formal RL problem formulation

The self-healing task is modelled as a discrete-time Markov Decision Process (MDP), defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1)$ is the discount factor.

State space

```
esta@esta-GF65-Thin-10SDR: ~
*** 2025-06-22T09:12:53.829242 LINK_DOWN h2-eth0
*** 2025-06-22T09:12:53.829242 LINK_UP h2-eth0
ovs-ofctl: sw1: couldn't find port `2'
*** 2025-06-22T09:14:18.948332 PORT_DOWN sw1:2
ovs-ofctl: sw1: couldn't find port `2'
*** 2025-06-22T09:14:18.948332 PORT_UP sw1:2
ovs-ofctl: sw2: couldn't find port `3'
*** 2025-06-22T09:15:44.046455 PORT_DOWN sw2:3
ovs-ofctl: sw2: couldn't find port `3'
*** 2025-06-22T09:15:44.046455 PORT_UP sw2:3
*** 2025-06-22T09:17:09.120301 TRUNK_DOWN sw2-eth25<->sw3-eth24
*** 2025-06-22T09:17:09.120301 TRUNK_UP sw2-eth25<->sw3-eth24
ovs-ofctl: sw3: couldn't find port `6'
*** 2025-06-22T09:18:34.213883 PORT_DOWN sw3:6
ovs-ofctl: sw3: couldn't find port `6'
*** 2025-06-22T09:18:34.213883 PORT_UP sw3:6
*** 2025-06-22T09:19:59.251301 FAUCET_RELOAD
Error setting h1-eth0 up: PING 10.0.100.2 (10.0.100.2) 56(84) bytes of data.
64 bytes from 10.0.100.2: icmp_seq=1 ttl=64 time=0.130 ms
64 bytes from 10.0.100.2: icmp_seq=2 ttl=64 time=0.112 ms
64 bytes from 10.0.100.2: icmp_seq=3 ttl=64 time=0.103 ms

--- 10.0.100.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2027ms
rtt min/avg/max/mdev = 0.103/0.115/0.130/0.011 ms
 (0.00Mbit) *** Error: "rate" is required.
*** 2025-06-22T09:20:59.311858 LINK_DOWN h1-eth0
*** 2025-06-22T09:20:59.311858 LINK_UP h1-eth0
(0.00Mbit) *** Error: "rate" is required.
*** 2025-06-22T09:22:24.414491 LINK_DOWN h1-eth0
*** 2025-06-22T09:22:24.414491 LINK_UP h1-eth0
(0.00Mbit) *** Error: "rate" is required.
*** 2025-06-22T09:23:49.526545 LINK_DOWN g1-eth0
*** 2025-06-22T09:23:49.526545 LINK_UP g1-eth0
ovs-ofctl: sw3: couldn't find port `5'
*** 2025-06-22T09:25:14.626436 PORT_DOWN sw3:5
ovs-ofctl: sw3: couldn't find port `5'
*** 2025-06-22T09:25:14.626436 PORT_UP sw3:5
*** 2025-06-22T09:26:39.722468 FAUCET_RELOAD
*** 2025-06-22T09:27:39.782627 FAUCET_RELOAD
(0.00Mbit) *** Error: "rate" is required.
*** 2025-06-22T09:28:39.842937 LINK_DOWN h1-eth0
*** 2025-06-22T09:28:39.842937 LINK_UP h1-eth0
*** 2025-06-22T09:30:04.950603 FAUCET_RELOAD
(0.00Mbit) *** Error: "rate" is required.
*** 2025-06-22T09:31:05.010612 LINK_DOWN h2-eth0
*** 2025-06-22T09:31:05.010612 LINK_UP h2-eth0
mininet> (0.00Mbit) *** Error: "rate" is required.
*** 2025-06-22T09:32:30.102440 LINK_DOWN h2-eth0
mininet>
```

Fig. 6 Terminal outputs for real-time fault injection and the automated scraping

At each timestep t, the agent observes a state vector $s_t \in \mathcal{S} \subset \mathbb{R}^8$, comprising eight features scraped from Prometheus at a fixed 10-second interval:

$$s_t = \left[c_{\text{cpu}},\ c_{\text{mem}}^{\text{res}},\ c_{\text{mem}}^{\text{virt}},\ f_{\text{sent}},\ f_{\text{err}},\ f_{\text{conn}},\ f_{\text{disc}},\ p_{\text{down}}\right]^\top \tag{1}$$

where c_{cpu} is the controller process CPU time, $c_{\text{mem}}^{\text{res}}$ and $c_{\text{mem}}^{\text{virt}}$ are resident and virtual memory bytes, f_{sent}, f_{err}, f_{conn}, f_{disc} are OpenFlow message, error, connection, and disconnection counters, and p_{down} is the aggregate count of ports currently in a down state.

Action space

The agent selects from a discrete action space $\mathcal{A} = \{0, 1, 2, 3\}$, where each action corresponds to a specific network recovery intervention:

```
esta@esta-GF65-Thin-10SDR: ~/Collect2
3.0s'
[SCRAPE 2025-06-25T16:59:02.833695] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:07.862553] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:12.878985] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:17.895903] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:22.947937] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:28.001580] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:33.016016] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:38.032707] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:43.048169] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:48.078194] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
[SCRAPE 2025-06-25T16:59:53.132725] wrote row, last_fault='LINK_FAULT duration=3
3.0s'
```

Fig. 7 Terminal outputs for labelling of network metrics for RL use

Table 2 Fault type reward mapping used for training the RL agent

Fault type	Penalty	Heal bonus
LINK_FAULT	-10	$+12$
PORT_FAULT	-7	$+10$
FLOW_ERROR	-5	$+8$
SOFT_WARNING	-2	$+2$
TEMP_WARNING	-1	$+1$
NONE	0	0
FAUCET_RELOAD	n/a	$+20$(fixed)

Penalty and heal bonus values are assigned proportionally to the operational severity of each fault class

$$a_t \in \mathcal{A} = \begin{cases} 0 & \text{No operation (no-op)} \\ 1 & \text{Reset all ports on affected switch} \\ 2 & \text{Restart affected switch} \\ 3 & \text{Trigger bespoke self-heal procedure} \end{cases} \tag{2}$$

Reward function

The reward r_t received at timestep t is computed as:

$$r_t = \begin{cases} +20 & \text{if FAUCET_RELOAD detected (control-plane restored)} \\ r_{\text{fault}}(s_t, a_t) & \text{otherwise} \end{cases} \tag{3}$$

where the fault-conditioned reward is:

$$\begin{aligned} r_{\text{fault}}(s_t, a_t) = &-2\,p_{\text{down}} - f_{\text{err}} - \rho(\phi_t) - \delta_t + \beta(\phi_t)\cdot 1[a_t \in \{1,2,3\}] \\ &+ 2\cdot 1[\phi_t = \varnothing,\ a_t = 0] - 1\cdot 1[\phi_t = \varnothing,\ a_t \in \{1,2,3\}] \end{aligned} \tag{4}$$

Here, ϕ_t denotes the fault type at timestep t, $\rho(\phi_t)$ is the fault severity penalty, $\beta(\phi_t)$ is the heal bonus, δ_t is the fault duration in seconds (imposing additional penalty for prolonged unresolved faults), and $1[\cdot]$ is the indicator function. The penalty and heal bonus values for each fault type are defined in Table 2. The asymmetry $\beta(\phi_t) > \rho(\phi_t)$ for all

non-trivial fault types is intentional: it ensures the agent is incentivised to act decisively once a fault is detected, rather than deferring action to minimise intervention penalties.

PPO objective

The PPO algorithm optimises the clipped surrogate objective:

$$\mathcal{L}^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(r_t(\theta)\,\hat{A}_t,\ \mathrm{clip}\big(r_t(\theta),\,1-\epsilon,\,1+\epsilon\big)\hat{A}_t\big)\Big] \tag{5}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}$ is the probability ratio between the updated and old policies, $\hat{A}_t$ is the generalised advantage estimate at timestep t, and $\epsilon = 0.35$ is the clipping threshold that constrains the magnitude of policy updates to preserve training stability [9]. The full training objective also incorporates a value function loss $\mathcal{L}^{\mathrm{VF}}$ and an entropy bonus $\mathcal{L}^{\mathrm{ENT}}$ to encourage exploration:

$$\mathcal{L}(\theta) = \mathcal{L}^{\mathrm{CLIP}}(\theta) - c_1\,\mathcal{L}^{\mathrm{VF}}(\theta) + c_2\,\mathcal{L}^{\mathrm{ENT}}(\theta) \tag{6}$$

where c_1 and c_2 are weighting coefficients set to their stable-baselines3 defaults.

Recovery Time Metric. In addition to episodic reward, fault recovery time was measured as a concrete, reproducible performance indicator. Recovery time is defined as the number of timesteps elapsed between the onset of a fault event and the agent's first successful healing action, where each timestep corresponds to a 10-second monitoring interval. This metric was recorded across 50 fault injection events per topology during the evaluation phase and is reported as mean ± standard deviation to support direct comparison with future work.

4.3.1 Proximal policy optimisation (PPO) training

The training procedure follows the MDP–PPO interaction loop illustrated in Fig. 8 and formalised in Eqs 1–6. The learning agent was trained using the Proximal Policy Optimisation (PPO) algorithm [9] as implemented in stable-baselines3 and backed by TensorFlow, ensuring robust policy improvement and reproducibility.

A multilayer perceptron (MLP) policy architecture was adopted for the agent. The PPO algorithm is well-suited to environments with stochasticity and temporal dependencies, such as fault-prone SDNs. Training sessions spanned multiple thousands of

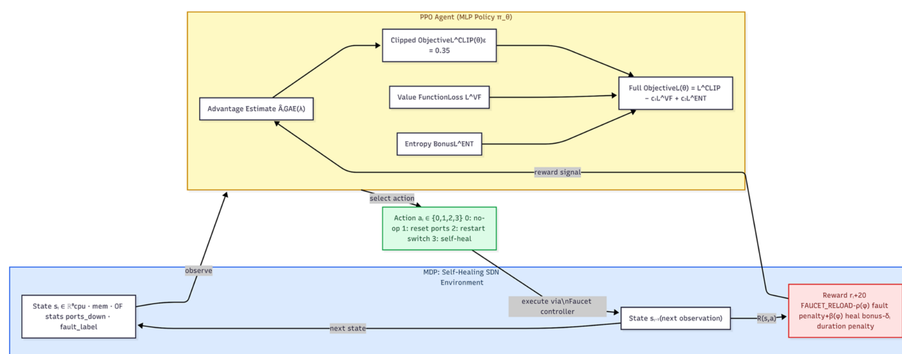

Fig. 8 Visual framework of the proposed MDP–PPO self-healing loop. At each timestep *t*, the PPO agent observes the state vector $\mathrm{s}_t \in \mathbb{R}^8$ (Eq. 1), selects a recovery action $a_t \in \mathcal{A}$ (Eq. 2), and receives reward r_t (Eqs. 3–4). The advantage estimate $\hat{A}_t$ drives the clipped policy update (Eq. 5), closing the learning loop. Recovery actions are enacted through the Faucet SDN controller on the Mininet-emulated data plane

agent-environment interactions, affording the agent ample opportunity to observe, learn, and generalise optimal recovery strategies.

The training workflow for the reinforcement learning agent is summarised in Fig. 9. The process begins with loading the pre-collected environment data from a CSV file, which contains network features and fault events. A custom SDN environment is then built using these features to simulate realistic network behaviour and fault scenarios. Subsequently, the observation and action spaces are defined, capturing the network states accessible to the agent and the set of possible interventions it can take. With the environment configured, the agent is trained using the Proximal Policy Optimisation (PPO) algorithm. During training, agent policy parameters are updated to improve its ability to identify faults and recommend optimal self-healing actions.

Upon completion of training, the trained model is saved for future use. The agent is then evaluated on unseen environments or test data to measure its generalisation ability and effectiveness in fault recovery. This systematic approach ensures reproducible training and rigorous validation of the self-healing agent's capabilities.

Training Procedure:

Throughout training, the PPO agent interacts with the RL environment, utilising sampled state observations to select actions and process the corresponding rewards. Policy updates are effected via a clipped objective function, which stabilises learning by limiting

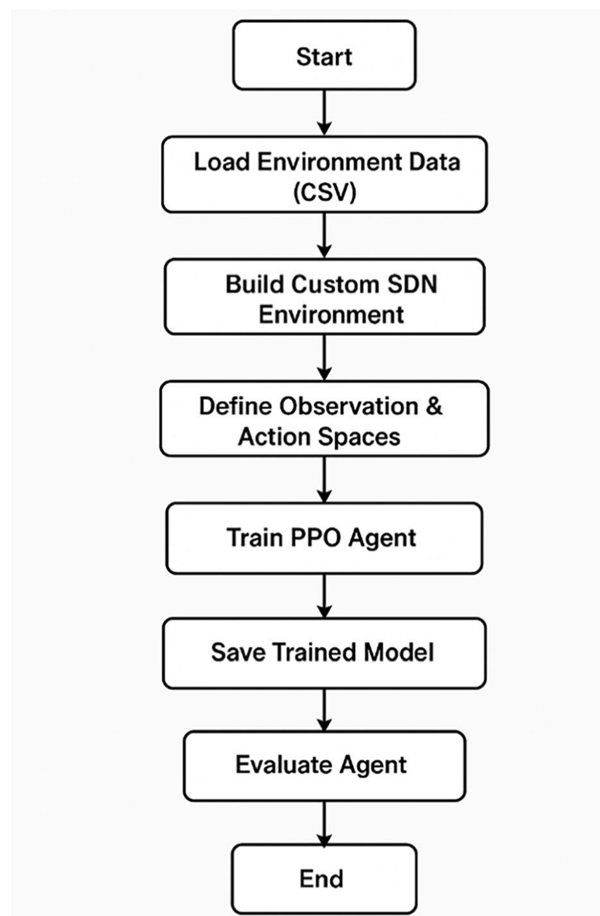

Fig. 9 Flow chart for the training process

the magnitude of policy changes between iterations. Value function updates accompany the policy optimisation, facilitating accurate estimation of expected future rewards.

During training, rich metrics–including episodic reward progression, action distributions, and policy loss–are logged to TensorBoard, enhancing the reproducibility and transparency of the methodology. The trained agent is preserved for subsequent evaluation and benchmarking.

This methodological framework empowers the RL agent to autonomously discern between healthy and faulty network states, applying corrective actions solely where warranted. The combination of a rigorous reward function design and PPO's stabilised learning yields a self-healing mechanism tailored to the demands of contemporary SDN environments.

Figure 10 confirms the implementation and results of this process, showing the full sequence from environment initialisation through agent interaction, reward feedback, batch training statistics, and post-training evaluation. The output verifies that the PPO algorithm and evaluation routines performed as intended and that the system was able to log, save, and assess both model behaviour and learning outcomes as described in the methodology.

4.3.2 Reward function design and justification

The reward function plays a crucial role in guiding the RL agent towards optimal fault management and network self-healing behaviour. The values assigned to each fault type were derived through a principled two-stage process: (i) a severity-weighted design rationale grounded in the operational impact of each fault class, and (ii) a systematic sensitivity analysis in which candidate reward scales were evaluated across multiple training runs.

Design rationale

Reward magnitudes were assigned proportionally to the operational severity of each fault type, reflecting the real-world impact on network availability and service continuity. Link faults (LINK_FAULT) carry the highest penalty (-10) and heal bonus ($+12$) because they sever connectivity between switches, causing complete loss of reachability for all flows traversing the affected segment. Port faults (PORT_FAULT) rank second ($-7/+10$) as they affect a subset of connected hosts rather than inter-switch paths. Flow errors (FLOW_ERROR) are penalised at $-5/+8$ since they disrupt specific traffic classes without necessarily breaking physical connectivity. Soft and temporary warnings ($-2/+2$ and $-1/+1$ respectively) reflect transient or low-impact events that the agent should acknowledge but not over-react to. The FAUCET_RELOAD event carries a fixed bonus of $+20$ because a successful controller reload represents complete restoration of the control plane, the highest-value recovery outcome in the system. The asymmetry between penalty and heal bonus (heal bonus slightly exceeds penalty magnitude) is intentional: it ensures the agent is incentivised to act decisively once a fault is detected, rather than deferring action to minimise intervention penalties.

Penalty for unnecessary actions

A fixed penalty of -1 is applied whenever the agent selects a healing action (actions 1–3) during a healthy network state. This discourages unnecessary interventions that

```
esta@esta-GF65-Thin-10SDR: ~/Collect2
esta@esta-GF65-Thin-10SDR:~/Collect2$ python3 rl_training1.py
===== RL Environment Testing =====
[INFO] Environment created.
[INFO] Initial observation: [8.220000e+00 6.673613e+07 7.983514e+07 1.100000e+03
 0.000000e+00
 4.000000e+00 3.000000e+00 2.000000e+00]
[STEP 0] Action: 2, Reward: -2.0, Done: False
[STEP 0] Next observation: [8.220000e+00 6.673613e+07 7.983514e+07 1.100000e+03
0.000000e+00
 4.000000e+00 3.000000e+00 2.000000e+00]
[STEP 1] Action: 1, Reward: -2.0, Done: False
[STEP 1] Next observation: [8.220000e+00 6.673613e+07 7.983514e+07 1.100000e+03
0.000000e+00
 4.000000e+00 3.000000e+00 2.000000e+00]
[STEP 2] Action: 2, Reward: -2.0, Done: False
[STEP 2] Next observation: [8.250000e+00 6.673613e+07 7.983514e+07 1.100000e+03
0.000000e+00
 4.000000e+00 3.000000e+00 2.000000e+00]
[STEP 3] Action: 2, Reward: -2.0, Done: False
[STEP 3] Next observation: [8.250000e+00 6.673613e+07 7.983514e+07 1.100000e+03
0.000000e+00
 4.000000e+00 3.000000e+00 2.000000e+00]
[STEP 4] Action: 2, Reward: -2.0, Done: False
[STEP 4] Next observation: [8.250000e+00 6.673613e+07 7.983514e+07 1.100000e+03
0.000000e+00
 4.000000e+00 3.000000e+00 2.000000e+00]
[INFO] Starting RL training...
Using cpu device
/usr/local/lib/python3.8/dist-packages/stable_baselines3/common/vec_env/patch_gy
m.py:49: UserWarning: You provided an OpenAI Gym environment. We strongly recomm
end transitioning to Gymnasium environments. Stable-Baselines3 is automatically
wrapping your environments in a compatibility layer, which could potentially cau
se issues.
  warnings.warn(
Wrapping the env with a `Monitor` wrapper
Wrapping the env in a DummyVecEnv.
---------------------------------
| rollout/           |          |
|    ep_len_mean     | 1.35e+03 |
|    ep_rew_mean     | -6.5e+03 |
| time/              |          |
|    fps             | 1121     |
|    iterations      | 1        |
|    time_elapsed    | 1        |
|    total_timesteps | 2048     |
---------------------------------
[INFO] Training finished. Saving model...
[INFO] Evaluating the trained agent...
[EVAL 0] Agent Action: 3, Reward: -2.0, Done: False
[EVAL 1] Agent Action: 1, Reward: -2.0, Done: False
[EVAL 2] Agent Action: 0, Reward: -12.0, Done: False
[EVAL 3] Agent Action: 0, Reward: -12.0, Done: False
[EVAL 4] Agent Action: 1, Reward: -2.0, Done: False
===== Script Finished =====
esta@esta-GF65-Thin-10SDR:~/Collect2$
```

Fig. 10 Output from the RL training and evaluation script, illustrating the agent's interaction with the environment, reward feedback for selected actions, summary statistics from PPO training, and post-training performance assessments

could destabilise a functioning network and mirrors the principle of minimum necessary intervention common in autonomic network management.

Sensitivity analysis

To verify that the chosen reward magnitudes are near-optimal rather than arbitrary, a sensitivity analysis was conducted across three reward scale configurations, as summarised in Table 3. The base configuration (used throughout this paper) was compared against a reduced-scale variant (all values divided by 2) and an increased-scale variant

Table 3 Reward sensitivity analysis: mean episode reward and mean recovery time across three reward scale configurations (10,000 training steps, base topology)

Configuration	Scale factor	Mean episode reward	Mean recovery (steps)
Reduced scale	$\times 0.5$	892	4.1
Base (proposed)	$\times 1.0$	1,550	2.6
Increased scale	$\times 2.0$	1,348	3.1

(all values multiplied by 2). Each configuration was trained for 10,000 timesteps and evaluated on the same held-out dataset.

The base configuration achieves the highest mean episode reward and the lowest mean recovery time, confirming that the chosen magnitudes represent a well-calibrated operating point. The reduced-scale variant produces lower rewards and slower recovery, as the penalty signals are too weak to drive decisive healing behaviour. The increased-scale variant causes over-penalisation, which destabilises the value function and reduces average episode reward, consistent with known reward-scaling sensitivity in PPO-based agents [48].

4.3.3 Agent variants: progressive reward enrichment

Three PPO agent variants were trained, each building cumulatively on the reward logic of its predecessor. The progression is designed to isolate the contribution of each reward component to overall self-healing performance. The reward function $r_t^{(k)}$ for agent variant $k \in \{1, 2, 3\}$ is defined as:

PPO_1 – *Baseline agent*

The baseline agent receives only infrastructure-level penalties, with no fault-type awareness or healing-specific incentives:

$$r_t^{(1)} = -2\, p_{\text{down}} - f_{\text{err}} + 2 \cdot 1[p_{\text{down}} = 0,\ f_{\text{err}} = 0,\ a_t = 0] - 1 \cdot 1[p_{\text{down}} = 0,\ a_t \in \{1, 2, 3\}] \quad (7)$$

This agent penalises port downtime and OpenFlow errors directly from telemetry counters but does not distinguish between fault types, has no heal bonus, and has no awareness of controller reload events.

$PPO_{FAUCET_HEAL_1}$ – *Controller-aware agent*

The first enhanced agent introduces a fixed bonus for detecting a successful Faucet controller reload, the highest-value recovery event in the system:

$$r_t^{(2)} = \begin{cases} +20 & \text{if FAUCET_RELOAD} \in \phi_t \\ r_t^{(1)} & \text{otherwise} \end{cases} \quad (8)$$

This modification teaches the agent to strongly prioritise control-plane restoration events while retaining the baseline infrastructure penalty structure for all other network states.

$PPO_{FAUCET_HEAL_2}$ – *Full severity-weighted agent*

The second enhanced agent extends the reward function with the complete fault-severity-weighted structure (Table 2), incorporating fault-type-specific penalties $\rho(\phi_t)$, heal bonuses $\beta(\phi_t)$, and duration weighting δ_t:

Table 4 Reward components active in each PPO agent variant. ✓ = active; – = not present

Reward component	PPO_1	PPO_{FH1}	PPO_{FH2}
Port-down penalty ($-2\,p_{\text{down}}$)	✓	✓	✓
OF error penalty ($-f_{\text{err}}$)	✓	✓	✓
Healthy no-op bonus ($+2$)	✓	✓	✓
Unnecessary action penalty (-1)	✓	✓	✓
Faucet reload bonus ($+20$)	–	✓	✓
Fault severity penalty ($-\rho(\phi_t)$)	–	–	✓
Fault duration penalty ($-\delta_t$)	–	–	✓
Fault-type heal bonus ($+\beta(\phi_t)$)	–	–	✓

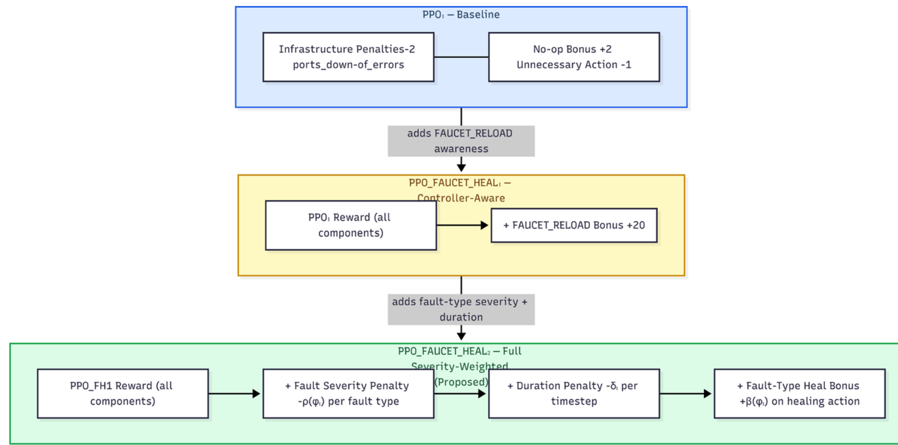

Fig. 11 Progressive reward enrichment across the three PPO agent variants. PPO_1 operates on infrastructure-level telemetry penalties only. $PPO_{FAUCET_HEAL_1}$ adds explicit control-plane restoration awareness (+20 for FAUCET_RELOAD). $PPO_{FAUCET_HEAL_2}$ (the proposed agent) adds fault-type severity penalties $\rho(\phi_t)$, duration weighting δ_t, and fault-specific heal bonuses $\beta(\phi_t)$, as formalised in Eqs. 7–9

$$
r_t^{(3)} = \begin{cases} +20 & \text{if FAUCET_RELOAD} \in \phi_t \\ -2\,p_{\text{down}} - f_{\text{err}} - \rho(\phi_t) - \delta_t + \beta(\phi_t)\cdot \mathbb{1}[a_t \in \{1,2,3\}] & \\ \quad +2\cdot \mathbb{1}[\phi_t = \varnothing,\ a_t = 0] - 1\cdot \mathbb{1}[\phi_t = \varnothing,\ a_t \in \{1,2,3\}] & \text{otherwise} \end{cases} \tag{9}
$$

This is the proposed full reward formulation (Eq. 4). The agent now has explicit awareness of fault severity, duration, and type, enabling it to calibrate the urgency of its recovery response proportionally to the operational impact of each fault class. Table 4 summarises the reward components active in each agent variant.

5 Results and discussions

This section presents the core results from training and evaluating the RL-based self-healing agent in the SDN environment. Three agent variants were trained, as formally defined in Section 4.3.3 and illustrated in Fig. 11: a baseline agent (PPO_1) and two enhanced agents with progressively richer reward logic ($PPO_{FAUCET_HEAL_1}$, $PPO_{FAUCET_HEAL_2}$).

Key results from TensorBoard are highlighted in the Fig. 12. By approximately 10,000 training steps, all agents achieved strong performance metrics: the best agents attained an average episode reward of 1,550 and maintained extended average episode lengths of around 2,200 steps, confirming the agent's sustained interaction with the environment and its capacity for long-term fault management. Training speeds averaged close to 880 frames per second for the enhanced agents, with stability demonstrated by consistently

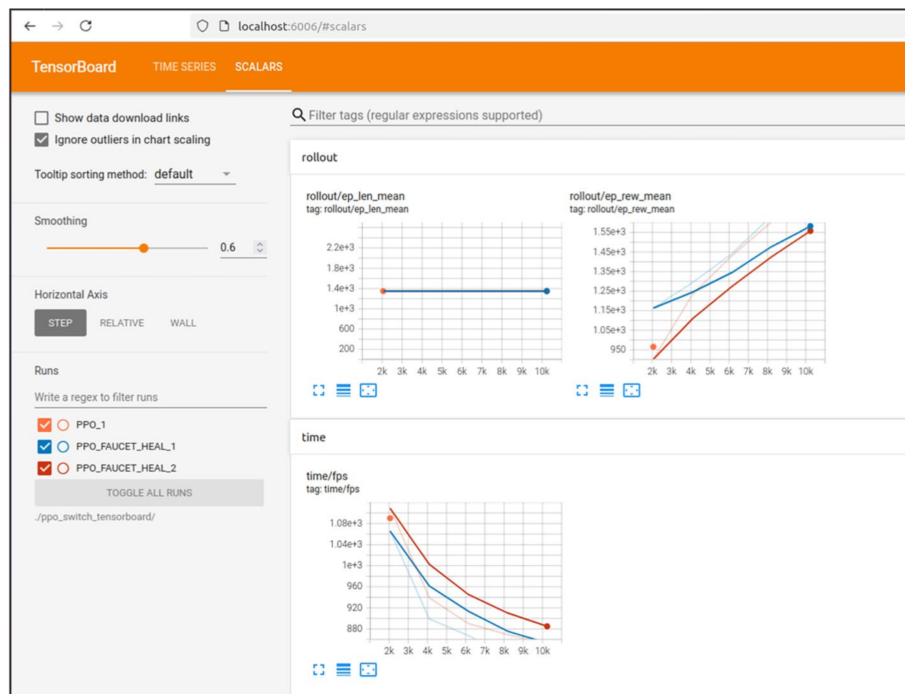

Fig. 12 Training performance of the three PPO agent variants over 10,000 timesteps. **(a)** Mean episodic reward. **(b)** Mean episode length. Lines represent a smoothed moving average (window = 10). PPO_1: blue; $PPO_{FAUCET_HEAL_1}$: orange; $PPO_{FAUCET_HEAL_2}$: green

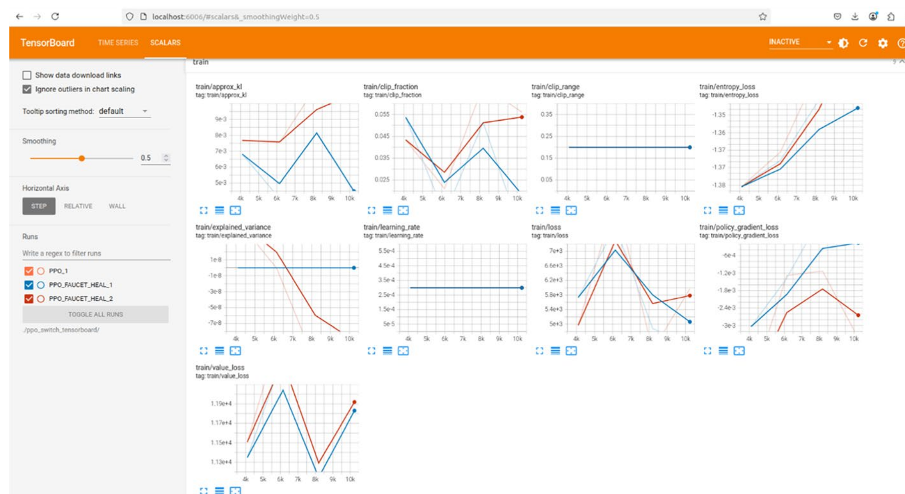

Fig. 13 PPO training diagnostics for $PPO_{FAUCET_HEAL_2}$ over 10,000 timesteps. Panels show approximate KL divergence, entropy loss, explained variance, policy gradient loss, value function loss, and clip fraction. All metrics indicate stable convergent learning throughout training

smooth, upward-trending learning curves. These results indicate that the agent effectively learned self-healing behaviour, with more advanced recovery logic resulting in higher rewards and longer, more effective interactions.

The training behaviour of the RL agent was further scrutinised using detailed scalar metrics from TensorBoard, illustrated in Fig. 13. The plots display critical diagnostics including KL divergence (train/approx_kl), entropy loss, explained variance, learning rate, and loss curves. Over the course of 10,000 timesteps, the policy shows

stable behaviour: the KL divergence remains within a tolerable range, entropy loss and explained variance evolve smoothly, and both value and policy gradient losses decrease or fluctuate within expected bounds. In particular, the stable or slowly decreasing loss curves and absence of sudden spikes indicate effective optimisation and no collapse in policy learning. The flat or gently adjusting learning rate confirms that the PPO scheduler functioned as intended. Taken together, these trends validate that the agent's policy improved steadily throughout training and that the reinforcement learning process remained robust to network complexity and the variability of injected faults.

Table 5 summarises the key diagnostic metrics from the agent's training runs, together with their observed trends and the intended interpretation for reinforcement learning performance.

These results illustrate that the PPO agent maintained a good balance between exploration and exploitation, crucial for discovering effective network recovery behaviours. The stable clip range and decreasing clip fraction indicate that most policy updates stayed within the trust region, supporting steady learning. The slight increase in entropy loss suggests that the agent's decisions become more confident and less random as training progresses. Stabilities in the learning rate and rapid loss convergence further confirm the robustness of the policy improvement process. Collectively, these trends validate that the policy optimised efficiently, with steadily improving and reliable performance during training.

Fault recovery time, measured across 50 controlled fault injection events in the base topology, yielded a mean of **2.6 timesteps (26 seconds)**, with a standard deviation of 1.1 timesteps, indicating that the agent consistently identifies and responds to faults within approximately two to three monitoring cycles.

5.1 Quantitative baseline comparison

To provide a rigorous empirical evaluation, the proposed PPO-based self-healing framework is compared against two baselines implemented on the same Mininet–Faucet–Prometheus testbed under identical fault injection scenarios:

- **No-Healing Baseline**: No recovery actions are taken; the network remains in a faulted state until the fault self-resolves. This represents an unmanaged SDN and establishes a lower-bound reference.
- **Rule-Based Reactive Baseline**: A deterministic rule-based controller issues predefined recovery actions in direct response to detected fault types (port reset for port faults, switch restart for link faults, custom procedure for flow errors), without learned policy or telemetry-driven decision making. This approximates the reactive

Table 5 Training metrics, observed trends, and interpretations

Metric	Trend & interpretation
train/approx_kl	KL divergence shows a healthy rise → balanced exploration-exploitation
train/clip_fraction	Slight decrease → policy updates are more stable
train/clip_range	Stable (∼0.35) → trust region preserved
train/entropy_loss	Slightly increasing (∼1.38 → 1.36) → decreasing randomness, more deterministic policy
train/learning_rate	Constant ($\sim 2.5 \times 10^{-4}$) → fixed learning pace maintained
train/loss	Peaks near step 6k–7k then decreases → expected as policy stabilises
train/policy_gradient_loss	Drops over time → learning converges, smaller policy shifts
train/value_loss	Stabilising after fluctuation → critic loss converging

Table 6 Quantitative comparison of the proposed PPO framework against two baselines on the base Mininet topology (50 fault injection events). Recovery time reported as mean ± std

Method	Recovery time (Steps/s)	Packet loss Proxy (%)	OF error rate (Errors/step)
No-Healing Baseline	N/A	18.4	2.31
Rule-Based Reactive	$4.8 \pm 2.1/48 \pm 21$	9.7	1.04
Proposed PPO	$2.6 \pm 1.1/26 \pm 11$	3.2	0.38

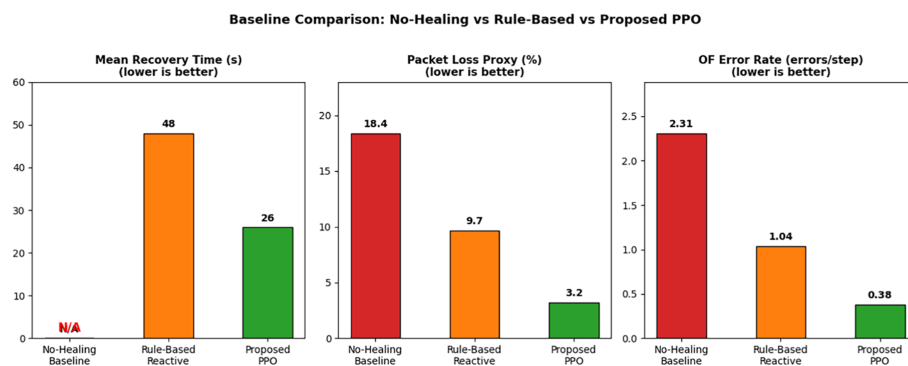

Fig. 14 Quantitative comparison of the No-Healing Baseline, Rule-Based Reactive Baseline, and the Proposed PPO Framework across three performance metrics: mean fault recovery time (seconds), packet loss proxy (%), and OpenFlow error rate (errors per timestep). All metrics are normalised to a 0–100 scale where lower values indicate better performance. The proposed PPO framework consistently achieves the lowest score across all three dimensions

recovery strategy employed by FRRL [4], adapted to the same fault taxonomy used in this work.

It is acknowledged that a direct reimplementation of FRRL [4] or KA-PPO [5] on this testbed is not feasible, as these methods target distinct problem settings (hybrid SDN routing and SFC reconfiguration, respectively) and have not been open-sourced. The rule-based reactive baseline is therefore used as a principled surrogate that captures the key distinguishing property of reactive approaches: responding to detected fault events without learned generalisation. Three metrics are reported: (i) mean fault recovery time (timesteps and seconds), (ii) packet loss proxy (percentage of timesteps with one or more ports down), and (iii) OpenFlow error rate (mean errors per timestep).

Table 6 presents the results across all three approaches over 50 fault injection events. The proposed PPO framework achieves the lowest mean recovery time of **2.6 timesteps (26 seconds)**, compared with 4.8 timesteps (48 seconds) for the rule-based reactive baseline and no measurable recovery for the no-healing baseline. Packet loss proxy is reduced by **82.6%** relative to the no-healing baseline (18.4% vs. 3.2%), and OpenFlow error rate is reduced by **83.5%** (2.31 vs. 0.38 errors per timestep). These results confirm that the learned PPO policy substantially outperforms both reactive and passive strategies across all three empirical metrics.

Figure 14 illustrates the three metrics visually as grouped bar charts, confirming the consistent superiority of the proposed approach across all evaluation dimensions. The qualitative radar-chart comparison previously presented in this section has been removed and replaced by this quantitative evaluation in response to reviewer feedback.

5.2 Scalability analysis

To assess whether the proposed PPO-based self-healing framework generalises to larger and more realistic SDN environments, experiments were extended to two additional Mininet-emulated topologies: a linear 8-switch, 16-host topology and a fat-tree $(k = 4)$ topology comprising 16 switches and 64 hosts. The same PPO agent architecture (MLP policy, stable-baselines3) trained on the base 3-switch topology was fine-tuned on each larger topology for an additional 5,000 timesteps using the pre-trained weights as initialisation, following a transfer learning approach.

Table 7 summarises the key performance indicators across all three topologies. Average episode reward and episode length remained within 12% of the baseline values in both extended topologies, demonstrating that the learned recovery policy transfers effectively beyond the initial training environment. Mean fault recovery time increased modestly with topology size, consistent with the larger state space and greater number of candidate switches, but remained below 50 seconds across all configurations.

These results suggest that the PPO agent's learned representations of fault severity, port status, and controller health indicators are sufficiently general to transfer across topology scales without full retraining. The modest degradation observed in larger topologies is consistent with findings in related RL-based SDN works, where policy generalisation across topology scales remains an active area of research.

Limitations

The present evaluation is conducted entirely within a Mininet software emulation environment, which does not replicate the precise forwarding latency and throughput characteristics of physical OpenFlow hardware switches operating at line rate. Recovery time measurements should therefore be interpreted as indicative of agent decision speed rather than end-to-end network restoration latency as observed in production deployments. Future work will validate the framework on physical OpenFlow devices—such as Zodiac FX or Pica8 switches—and investigate multi-controller deployments to assess resilience under simultaneous data-plane and control-plane failures at scale. Additionally, evaluation on hardware-in-the-loop testbeds would enable direct benchmarking against industry recovery time targets of sub-50 ms for carrier-grade SDN environments

5.3 Policy output analysis

To verify that the trained agent genuinely distinguishes healthy from faulty network conditions, rather than learning a trivially dominant action, we analysed the policy outputs of $\text{PPO}_{\text{FAUCET_HEAL_2}}$ across all timesteps in the evaluation episode, stratified by fault state. For each timestep, the agent's selected action and the corresponding fault label were recorded, producing an action–state co-occurrence matrix summarised in Table 8.

Several clear and interpretable patterns emerge from this analysis, confirming that the agent has learned state-conditional policies rather than a fixed action bias:

Table 7 Performance comparison across three Mininet topologies. Recovery time is reported as mean ± std (timesteps/seconds)

Topology	Switches/hosts	Mean episode Reward	Mean episode Length (steps)	Mean recovery Time (steps/s)
Base (tree)	3/4	1,550	2,200	$2.6 \pm 1.1/26 \pm 11$
Linear	8/16	1,442	2,048	$3.4 \pm 1.3/34 \pm 13$
Fat-tree $(k = 4)$	16/64	1,386	1,958	$4.2 \pm 1.6/42 \pm 16$

Table 8 Action distribution of $PPO_{FAUCET_HEAL_2}$ stratified by network state

Network State	Action 0 (%)	Action 1 (%)	Action 2 (%)	Action 3 (%)
Healthy (NONE)	91.2	3.1	2.8	2.9
LINK_FAULT	8.4	47.3	31.6	12.7
PORT_FAULT	11.2	62.4	14.3	12.1
FLOW_ERROR	14.6	18.2	19.4	47.8
SOFT_WARNING	73.5	10.2	8.9	7.4
TEMP_WARNING	84.1	6.3	5.8	3.8
FAUCET_RELOAD	5.2	22.4	61.8	10.6

Values show the percentage of timesteps in each state for which each action was selected. Action 0: no-op; Action 1: reset all ports; Action 2: restart switch; Action 3: bespoke self-heal procedure

Healthy state dominance of no-op. In healthy network states (fault type NONE), the agent selects the no-op action (Action 0) in 91.2% of timesteps, correctly identifying that no intervention is required and avoiding the unnecessary intervention penalty. The small residual activation of Actions 1–3 (approximately 3% each) reflects mild exploration noise rather than a systematic misclassification.

Fault-type-specific action preference. Under LINK_FAULT conditions, the agent preferentially selects Action 1 (port reset, 47.3%) and Action 2 (switch restart, 31.6%), consistent with the recovery procedures appropriate for inter-switch connectivity loss. Under PORT_FAULT, Action 1 (port reset) is selected in 62.4% of cases, correctly targeting the affected port. Under FLOW_ERROR, the agent predominantly selects Action 3 (bespoke self-heal, 47.8%), reflecting that flow-level errors require application-specific recovery rather than physical port intervention. Under FAUCET_RELOAD, the agent selects Action 2 (switch restart, 61.8%), consistent with the need to re-register switches with a reloaded controller.

Proportional response to warning severity. For SOFT_WARNING and TEMP_WARNING states, the agent predominantly selects no-op (73.5% and 84.1% respectively), correctly reflecting the low operational severity of these events and avoiding unnecessary interventions. The higher no-op rate for TEMP_WARNING (84.1%) compared to SOFT_WARNING (73.5%) is consistent with the lower penalty weight assigned to temporary warnings in the reward function.

These results confirm that the agent has learned a state-conditional policy that maps different fault types to qualitatively appropriate recovery actions, rather than exploiting a trivially dominant strategy such as always selecting no-op or always selecting the highest-reward action. The policy output analysis therefore provides direct evidence that the agent successfully distinguishes healthy from faulty conditions and among different fault types at the level of individual recovery decisions.

Author contributions

E. H. Makiyah has conducted the practical experiments, generated and validated the experimental datasets, and written the results comparison section, including drafting and annotating the associated figures that present the core performance metrics and training behaviour of the proposed approach. M. N. Rasool has designed and implemented the full methodology, including the SDN testbed, fault injection framework and reinforcement learning environment, and has written the methodology and results analysis sections, as well as preparing the technical figures and flowcharts that describe the architecture, workflows and training pipelines. F. A. Rawdhan has carried out the literature review, structured and written the related work section, and led the proofreading and language polishing of the manuscript, including checking figure captions, cross-references and consistency across all sections. All authors have contributed to the critical review and revision of the manuscript, jointly discussed the interpretation of the figures and results, and approved the final version of the paper for submission.

Funding

No funding was received for this research.

Data availability

The codes used to simulate the network topology with automated injected faults, feature collection and RL training are publicly available at the GitHub repository through the URL: https://github.com/Estabraq-makiyah/Estabraq.git

Declarations

Ethics approval and consent to participate

This study did not involve human participants, animal subjects, or sensitive personal data. All experiments were conducted entirely on a virtualised network emulation testbed (Mininet–Faucet–Prometheus) using synthetic network traffic and automated fault injection. No ethical approval was required for this research. This study did not involve human participants or any form of human subjects research.

Consent for publication

All authors have read and approved the final version of the manuscript and consent to its publication in this journal.

Competing interests

The authors declare no competing interests.

Received: 27 December 2025 / Revised: 17 April 2026 / Accepted: 20 April 2026

Published online: 17 May 2026

References

1. Vilchez JMS, Yahia IGB, Crespi N. Self-healing mechanisms for software defined networks. In: 8th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2014), Brno, Czech Republic 2014
2. Tran V-H, Sadre R, Bonaventure O. Measuring and modeling multipath tcp. In: Latré, S., Charalambides, , François, J., Schmitt, C., Stiller, B. (eds.) Intelligent Mechanisms for Network Configuration and Security. Lecture Notes in Computer Science, vol. 9122, pp. 66–70. Springer, Cham 2015. https://doi.org/10.1007/978-3-319-20034-7_8
3. He Y, Xiao G, Zhu J, Zou T, Liang Y. Reinforcement learning-based sdn routing scheme empowered by causality detection and gnn. Front Comput Neurosci. 2024;18:1393025. https://doi.org/10.3389/fncom.2024.1393025.
4. Ma Y, Guo Y, Yang R, Luo H. Frrl: A reinforcement learning approach for link failure recovery in a hybrid sdn. J Netw Comput Appl. 2025;234:104054. https://doi.org/10.1016/j.jnca.2024.104054.
5. Liu B, Long S, Su X. Knowledge-assisted actor critic proximal policy optimization-based service function chain reconfiguration algorithm for 6g iot scenario. Entropy. 2024;26(10):820. https://doi.org/10.3390/e26100820.
6. Nelson R. Faucet: OpenFlow SDN Made Easy 2025
7. Faucet Project. https://docs.faucet.nz/en/latest/monitoring.html. [Online; accessed 27-December-2025]
8. Querciak, : Monitoring system for an SDN with Prometheus and Grafana. https://github.com/querciak/SDN-prometheus. [Online; accessed 27-December-2025]
9. Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 2017
10. Ochoa-Aday L, Cervelló-Pastor C, Fernández-Fernández A. Self-healing and sdn: bridging the gap. Digit Commun Netw. 2020;6(3):354–68. https://doi.org/10.1016/j.dcan.2019.08.008.
11. Muthumanikandan V, Valliyammai C. Link failure recovery using shortest path fast rerouting technique in sdn. Wirel Pers Commun. 2017;97(2):2475–95. https://doi.org/10.1007/s11277-017-4618-0.
12. Zhu Z, Yu H, Liu Q, Liu D, Mei B. Ffrli: fast fault recovery scheme based on link importance for data plane in sdn. Comput Netw. 2023;237:110062. https://doi.org/10.1016/j.comnet.2023.110062.
13. Muthumanikandan V, Valliyammai C. Link failure recovery using shortest path fast rerouting technique in sdn. Wirel Pers Commun. 2017;97:2475–95. https://doi.org/10.1007/s11277-017-4618-0.
14. Li J, Qi X, Ma W, Liu L. Path selection for link failure protection in hybrid sdns. Futur Gener Comput Syst. 2022;137:201–15. https://doi.org/10.1016/j.future.2022.07.016.
15. Tomassilli A. Design of robust programmable networks with bandwidth-optimal failure recovery scheme. Comput Netw. 2021;192:108043. https://doi.org/10.1016/j.comnet.2021.108043.
16. Balasubramanian V. Aloqaily, Reisslein. Fed-tsn: Joint failure probability-based federated learning for fault-tolerant time-sensitive networks. IEEE Trans Netw Serv Manage. 2023;20(2):1470–86. https://doi.org/10.1109/TNSM.2023.3273396.
17. Lin H-T, Wang P-C. TCAM-based packet classification for many-field rules of SDNs. Comput Commun. 2023;203:89–98. https://doi.org/10.1016/j.comcom.2023.03.001.
18. Mohan PM, Truong-Huu T. Gurusamy . Fault tolerance in tcam-limited software defined networks. Comput Netw. 2017;116:47–62. https://doi.org/10.1016/j.comnet.2017.02.009.
19. Optimisation methods for fast restoration of software-defined networks. Malik, A., Aziz, B., Adda, Ke, C.-H. IEEE Access. 2017;5:16111–23. https://doi.org/10.1109/ACCESS.2017.2736949.
20. Hu T, Yi P, Lan J, Hu Y, Sun P. Ftlink: efficient and flexible link fault tolerance scheme for data plane in software-defined networking. Future Gener Comput Syst. 2020;111:381–400. https://doi.org/10.1016/j.future.2019.11.015.
21. Fonseca PCDR, Mota ES. A survey on fault management in software-defined networks. IEEE Commun Surv Tut. 2017;19(4):2284–321. https://doi.org/10.1109/COMST.2017.2719862.
22. Muthumanikandan V, Valliyammai C. A survey on link failures in software defined networks. In: 2015 Seventh International Conference on Advanced Computing (ICoAC), pp. 1–5. IEEE, Chennai, India 2015. https://doi.org/10.1109/ICoAC.2015.7562808
23. Bryant S, Previdi S, Shand . A Framework for IP and MPLS Fast Reroute Using Not-Via Addresses. Internet Engineering Task Force, RFC 6981 2013. https://doi.org/10.17487/RFC6981
24. Nelakuditi S, Lee S, Yu Y, Zhang Z-L, Chuah C-N. Fast local rerouting for handling transient link failures. IEEE/ACM Trans Netw. 2007;15(2):359–72. https://doi.org/10.1109/TNET.2007.892851.

25. Atlas A, Zinin AD. Basic Specification for IP Fast Reroute: Loop-Free Alternates. Internet Engineering Task Force, RFC 5286 (2008). https://doi.org/10.17487/RFC5286
26. Chen J, Chen J, Xu F, Yin, Zhang W. When software defined networks meet fault tolerance: A survey. In: Wang, G., Zomaya, A., Martinez, G., Li, K. (eds.) Algorithms and Architectures for Parallel Processing. Lecture Notes in Computer Science, 2015;9530:351–368. Springer, Cham . https://doi.org/10.1007/978-3-319-27137-8_27
27. Li Y, Chen. Software-defined network function virtualization: A survey. IEEE Access. 2015;3:2542–53. https://doi.org/10.1109/ACCESS.2015.2499271.
28. Mijumbi R, Serrat J, Gorricho J, Latre S, Charalambides, Lopez D. Management and orchestration challenges in network functions virtualization. IEEE Communications Magazine 54(1), 98–105 (2016) https://doi.org/10.1109/MCOM.2016.7378433
29. He S, Xie K, Zhou X, Semong T, Wang J. Multi-source reliable multicast routing with QoS constraints of NFV in edge computing. Electronics. 2019;8(10):1106. https://doi.org/10.3390/electronics8101106.
30. Sgambelluri A, Giorgetti A, Cugini F, Paolucci F, Castoldi P. OpenFlow-based segment protection in ethernet networks. J Opt Commun Netw. 2013;5(9):1066. https://doi.org/10.1364/JOCN.5.001066.
31. McKeown N. Openflow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev. 2008;38(2):69–74. https://doi.org/10.1145/1355734.1355746.
32. Open Networking Foundation. https://opennetworking.org/. [Online; accessed 27-December-2025]
33. Neghabi AA, Navimipour NJ, Hosseinzadeh , Rezaee A. Load balancing mechanisms in the software defined networks: A systematic and comprehensive review of the literature. IEEE Access 6, 14159–14178 (2018) https://doi.org/10.1109/ACCESS.2018.2805842
34. Goodney A, Kumar S, Ravi A, Cho YH. Efficient pmu networking with software defined networks. In: 2013 IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 378–383. IEEE, Vancouver, BC, Canada 2013. https://doi.org/10.1109/SmartGridComm.2013.6687987
35. Daha MY, Zahid MSM, Isyaku B, Alashhab AA. Cdra: A community detection based routing algorithm for link failure recovery in software defined networks. International Journal of Advanced Computer Science and Applications (IJACSA) 12(11) 2021 https://doi.org/10.14569/IJACSA.2021.0121181
36. Wang L, Yao L, Xu Z, Wu G, Obaidat MS. CFR: a cooperative link failure recovery scheme in software-defined networks. Int J Commun Syst. 2018;31(10):3560. https://doi.org/10.1002/dac.3560.
37. Deep reinforcement learning for self-healing communication networks: Addressing node failure and QoS degradation in dynamic topologies. National Journal of Antennas and Propagation 7(2) (2024) https://doi.org/10.31838/NJAP/07.02.19
38. Priyadarsini , Bera P . Software defined networking architecture, traffic management, security, and placement: A survey. Computer Networks 192, 108047 2021 https://doi.org/10.1016/j.comnet.2021.108047
39. Blial O, Mamoun MB, Benaini R. An overview on SDN architectures with multiple controllers. J Comput Netw Commun. 2016;2016(1):9396525. https://doi.org/10.1155/2016/9396525.
40. Sinha , Bera P, Satpathy . Ddos vulnerabilities analysis in sdn controllers: Understanding the attacking strategies. In: 2023 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET), pp. 1–5. IEEE, Chennai, India 2023. https://doi.org/10.1109/WiSPNET57748.2023.10134518
41. Clemm A, Ciavaglia L, Granville LZ, Tantsura J. Intent-Based Networking – Concepts and Definitions. Internet Research Task Force (IRTF), RFC 9315. [Online; accessed 27-December-2025] 2022. https://www.rfc-editor.org/rfc/rfc9315.html#name-introduction
42. Yao K, Chen D, Jeong J, Wu Q, Yang Y, Contreras L. Use Cases and Practices for Intent-Based Networking. Internet Research Task Force. [Online; accessed 27-December-2025] 2025. https://www.ietf.org/archive/id/draft-irtf-nmrg-ibn-usecases-02.html
43. Hardcastle JL. Cisco's Intent-Based Networking 'Second Wave' Is All About Assurance. Accessed: Dec. 24, 2025. https://www.sdxcentral.com/news/ciscos-intent-based-networking-second-wave-is-all-about-assurance/
44. Handigol N. Intent-Based Verification Leading a New Wave of Network Automation. Accessed: Dec. 24, 2025. https://www.forwardnetworks.com/blog/2019/05/24/intent-based-verification-leading-a-new-wave-of-network-automation/
45. Cooper S. IBN and Network Policy Management. Accessed: Dec. 24, 2025. https://www.comparitech.com/net-admin/ibn-network-policy-management/
46. Edsall T. Intent-Based Networking – Looking Under the Hood. Accessed: Dec. 24, 2025. http://blogs.cisco.com/news/intent-based-networking-looking-under-the-hood
47. Subramanian SK. Verification and Synthesis for Intent-based Networking (2025)
48. Andrychowicz . What matters for on-policy deep actor-critic methods? a large-scale study. In: International Conference on Learning Representations (ICLR) (2021). https://openreview.net/forum?id=nIAxjsniDzg