Network Management Research Group
Internet-Draft
Intended status: Informational
Expires: 3 July 2026

Y. Cui
C. Liu
X. Xie
Tsinghua University
C. Du
Zhongguancun Laboratory
30 December 2025

A Framework to Evaluate LLM Agents for Network Configuration
draft-cui-nmrg-llm-benchmark-01

Abstract

This document specifies an evaluation framework and related definitions for intent-driven network configuration using Large Language Model(LLM)-based agents. The framework combines an emulator-based interactive environment, a suite of representative tasks, and multi-dimensional metrics to assess reasoning quality, command accuracy, and functional correctness. The framework aims to enable reproducible, comprehensive, and fair comparisons among LLM-driven network configuration approaches.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://datatracker.ietf.org/doc/draft-cui-nmrg-llm-benchmark/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-cui-nmrg-llm-benchmark/.

Discussion of this document takes place on the Network Management Research Group mailing list (mailto:nmrg@irtf.org), which is archived at https://mailarchive.ietf.org/arch/browse/nmrg. Subscribe at https://www.ietf.org/mailman/listinfo/nmrg/.

Source for this draft and an issue tracker can be found at https://github.com/nobrowning/draft_llm_conf_benchmark.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Copyright Notice

Table of Contents

1. Introduction

Network configuration is fundamental to ensuring network stability, scalability, and conformance with intended design behavior. Effective configuration requires not only a comprehensive understanding of network technologies but also advanced capabilities for interpreting complex topologies, analyzing dependencies, and specifying parameters accurately. Traditional automation approaches such as Ansible playbooks[A2023], NETCONF[RFC6241]/YANG models[RFC7950], or program-synthesis methods-either demand extensive manual scripting or are limited to narrow problem domains[Kreutz2014]. In parallel, Large Language Models (LLMs) have demonstrated the ability to interpret natural-language instructions and generate device-specific commands, showing promise for intent-driven automation in networking. However, existing work remains fragmented and lacks a standardized way to measure whether an LLM can truly operate as an autonomous agent in realistic, multi-step configuration scenarios.

Despite encouraging results in individual subtasks, most evaluations[Wang2024NetConfEval] rely on static datasets and ad hoc metrics that do not reflect real-world complexity. As a result: - There is no common benchmark suite covering diverse configuration domains (routing, QoS, security) with clearly defined intents, topologies, and ground truth. - Existing tests seldom involve interactive environments that emulate vendor-specific device behavior or provide runtime feedback on command execution. - Evaluation metrics are often limited to simple syntactic checks or isolated command validation, failing to capture whether the intended network behavior is actually achieved.

Consequently, it is difficult to compare different LLM approaches or to identify gaps in reasoning, context-sensitivity, and error-correction capabilities[Long2025][Liu2024][Fuad2024][Lira2024]. To address these shortcomings, this document introduce *NetConfBench*, a holistic framework that provides: 1. An emulator-based environment (built on GNS3) to simulate realistic device interactions. 2. A benchmark suite of forty tasks spanning routing, QoS, and security, each defined by intent, topology, initial state, ground-truth configuration, annotated reasoning trace, and expert-crafted testcases. 3. Multidimensional metrics-_reasoning score_, _command score_, and _testcase score_-that evaluate an agent's internal reasoning coherence, semantic correctness of generated commands, and functional outcomes in the emulated network.

NetConfBench aims to enable reproducible, comprehensive comparisons among single-turn LLMs, ReAct-style multiturn agents, and knowledge-augmented variants, guiding future research toward truly autonomous, intent-driven network configuration.

2. Terminology

For clarity within this document, the following terms and abbreviations are defined:

* Agent: A software component powered by an LLM that consumes a task intent, interacts with a network environment, and issues configuration commands autonomously.

* Configuration Command: A device-specific instruction (e.g., a Cisco IOS CLI line or a Juniper Junos set statement) sent by the agent to a network device.

* Environment: An emulated or real network instance that exposes device status, topology information, and feedback on applied commands.

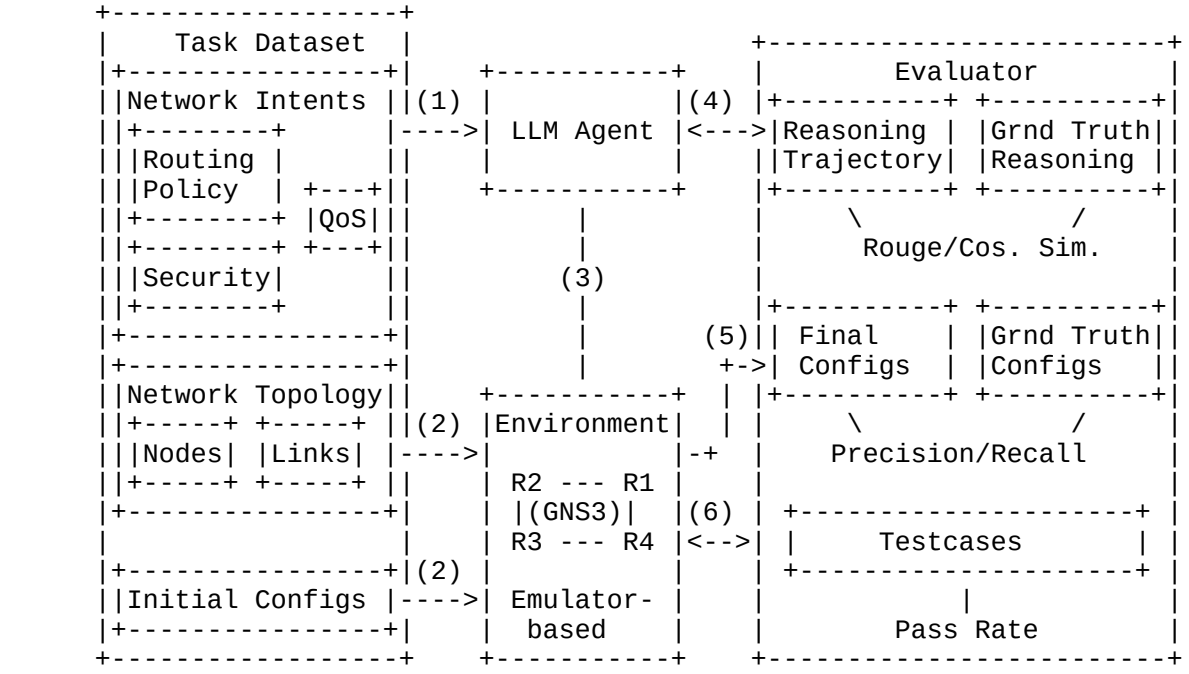
* Intent: A high-level specification of desired network behavior or objective, expressed in natural language or a structured format defined in this document.

* Task: A single evaluation unit defined by (1) a scenario category, (2) an environment topology, (3) initial device configurations, and (4) an intent. The agent is evaluated on its ability to fulfill the intent in the given environment.

* Testcase: A concrete, executable set of verification steps (e.g., ping tests, traffic-flow validation, policy checks) used to assert whether the agent’s final configuration satisfies the intent.

* MCP (Model Context Protocol): An open standard protocol designed to facilitate communication between LLMs and external data sources or tools, enabling standardized tool discovery, invocation, and result handling.

3. Framework Overview



Legend:
(1)Task Assignment (2)Environment Setup
(3)Interactive Task Execution (4)Reasoning Trajectory Export
(5)Final Configuration Export (6)Testcase Execution

Figure 1: The NetConfBench Framework

The proposed framework is shown in Figure 1. The flow begins with a *Task Dataset* defining network intents and topologies. The *LLM Agent* perceives the environment, reasons about required actions, and applies configuration commands. The *Environment* simulates or controls real devices, providing feedback for each action. Finally, the *Evaluator* compares the agent's outputs against ground-truth configurations and reasoning, computing scores for accuracy and completion.

The framework supports multiple communication protocols for agent-environment interaction, including direct API calls and standardized protocols such as MCP. When using MCP, network operations are encapsulated as tools that can be discovered and invoked by the LLM agent through the MCP client-server architecture.

3.1. Components

NetConfBench consists of four key components:

1. *Task Dataset*
   A repository of forty configuration tasks, each defined as a JSON object with:

   * *Intent*: One or more natural language instructions.

   * *Topology*: A list of node names and link definitions.

   * *Initial Configuration*: The initial configuration state of all nodes.

   * *Ground Truth Configuration*: Expert-validated CLI commands that achieve the intent.

   * *Ground Truth Reasoning*: A textual record of the agent's step-by-step reasoning that maps high-level intent to low-level configuration actions.

   * *Testcases*: A set of verification procedures (e.g., _show_, _ping_, _ACL_ checks) that confirm functional intent satisfaction.

2. *Emulator Environment*
   Built on GNS3, this component launches official vendor images for routers and switches, replicating realistic CLI behavior. Key interfaces include:

   * *Agent-Network Interface (ANI)*: Based on the key stages commonly involved in intent-driven network configuration, the framework provides an Agent-Network Interface to facilitate structured interactions between the LLM agent and the emulated network environment. This interface supports four core actions: get-topology, get-running-cfg, update-cfg, and execute_validation.

     - get-topology: provides this information in a format interpretable by the LLM.

     - get-running-cfg: enables the agent to obtain the active configurations of specified devices, providing essential context for planning subsequent updates.

- update-cfg: allows the agent to apply new configuration commands and provides detailed feedback on their execution, including whether each command was accepted or resulted in any errors.

- execute_validation: accepts a device name and a command string as parameters and returns the resulting output.

* *Task Evaluation Interface*: To enable reliable and objective assessment of the LLM agent's configuration behavior, the environment provides a Task Evaluation Interface that allows the evaluation module to access relevant execution results. Specifically, this interface supports:

  - *Exporting the final configurations of all devices*: This allows for direct comparison with ground truth configurations to evaluate the correctness and completeness of the agent's output.

  - *Executing a set of predefined testcases*: These testcases are designed to verify whether the resulting network behavior accurately reflects the intended configuration objectives, as defined by the network intent.

3. *LLM Agent*
   A modular component that can be implemented with any LLM (open-source or closed-source). It interacts with the emulator via the *Agent-Network Interface* (ANI), issuing queries such as get-topology, get-running-cfg, update-cfg, and execute_validation. Agents may use:

   * *Single-Turn Generation*: The entire reasoning and command generation in one pass.

   * *ReAct-Style Multi-Turn Interaction*: Interleaved reasoning and actions, with runtime feedback guiding subsequent steps.

   * *External Knowledge Retrieval*: (Optional) Queries to a command manual to resolve vendor-specific syntax.

4. *Evaluator*
   Computes three core metrics for each task:

   * *Reasoning Score (S_reasoning)*

     The reasoning score evaluates whether the agent can coherently map network intents to concrete configuration actions through semantically aligned reasoning. This score compares the

agent’s reasoning process with a predefined ground truth reasoning process, focusing on logical consistency and semantic similarity.

For one-shot prediction, prompts are designed to elicit the reasoning process prior to command generation, enabling direct comparison. For multi-turn interaction, an auxiliary LLM summarizes the interleaved steps into a unified reasoning process, which is then compared against the ground truth. The reasoning score is computed using cosine similarity:

S_reasoning = (r_agent * r_gt) / (||r_agent|| * ||r_gt||)

where r_agent is the embedding of the agent’s reasoning process, and r_gt is the embedding of the ground truth reasoning process.

* *Command Score (S_command)*

This evaluation comprehensively assesses the effectiveness of configuration commands generated by the agent. While syntactic correctness is a prerequisite, it does not ensure that configuration commands are correctly applied to the device, particularly when commands must be issued within specific configuration contexts.

After the agent completes its configuration task, the final configurations of all devices are exported and compared to their initial configurations to extract the set of commands that were actually applied. Hierarchical parsing using the Python library ciscoconfparse ensures structural completeness during comparison. Since certain configuration parameters (e.g., ACL numbers, route policy names) are manually defined and do not have fixed values, wildcard-based fuzzy matching is introduced to ignore non-essential differences and focus on semantic equivalence.

Based on the extracted command sets, standard precision and recall are computed: - Precision measures the proportion of correctly generated commands among all generated commands - Recall measures the proportion of correctly generated commands relative to the ground truth command set

The command score is reported as the harmonic mean of precision and recall:

S_command = (2 * Precision * Recall) / (Precision + Recall)

* *Testcase Score (S_testcase)*

  While command-level evaluation based on configuration differences can effectively measure the semantic correctness of generated commands, it does not fully reflect whether the configuration actually achieves the intended network behaviors. To address this limitation, a testcase-driven evaluation strategy is introduced that directly verifies the functional correctness of the agent’s configuration in the target environment.

  A set of validation testcases is defined for each task, where each testcase encodes a network intent in the form of executable verification commands. To support complex tasks involving multiple sub-goals, the overall intent is decomposed into sub-intents based on node-specific configuration objectives. Each sub-intent is then formulated as an individual testcase to enable fine-grained evaluation and enhance interpretability.

  Examples of testcases include: - *Routing intent*: Verifying the next hop selection on intermediate routers to confirm end-to-end path correctness - *ACL intent*: Simulating traffic flows and validating whether they are allowed or denied as expected - *QoS intent*: Inspecting interface statistics to check whether QoS policies are properly enforced

  The testcase score is defined as the proportion of passed testcases among all defined testcases:

  S_testcase = |Passed Testcases| / |Total Testcases|

  This score reflects the agent’s ability to produce configurations that meet functional requirements and demonstrates practical applicability in real-world deployment scenarios.

3.2. Workflow

The evaluation workflow for each task proceeds through six stages:

1. *Task Assignment*
   NetConfBench selects a task from the JSON dataset and provides only the high-level intent(s) to the LLM agent.

2. *Environment Setup*
   The framework instantiates a GNS3 topology based on the task's topology and applies the startup-config to each device. Once the emulated network reaches a stable state, control transfers to the agent.

3. *Interactive Execution*
   The LLM agent receives the partial prompt containing:

   * The API specification for get-topology, get-running-cfg, update-cfg, and execute_validation.

   * The natural language intent.

   * (Optionally) Device model/version hints.

   * The agent issues a sequence of API calls; for single-turn agents, it outputs reasoning followed by a batch of CLI commands. For multi-turn agents, it alternates reasoning traces and API calls. When using MCP, network operations are encapsulated as tools that can be discovered and invoked by the LLM agent through the MCP client-server architecture.

4. *Reasoning Trajectory Export*
   After execution completes (agent signals "task done" or after a predefined command budget), NetConfBench captures the entire reasoning log:

   * For single-turn: the reasoning paragraph embedded in the LLM's output.

   * For ReAct: an auxiliary summarization LLM condenses the interleaved reasoning and actions into a single coherent trace.

5. *Final Configuration Export*
   The framework uses the Task Evaluation Interface to extract the final running configs from each device.

6. *Testcase Execution and Scoring*

   * *Command Score:* Hierarchical diff against ground truth commands.

   * *Testcase Score:* Execute each testcase in sequence; record pass/fail.

* *Reasoning Score:* Compute embedding similarity between the agent’s reasoning trace and ground truth reasoning.

The final per-task score is typically reported as a tuple (S_reasoning, S_command, S_testcase). Aggregate results across the forty tasks enable comparisons among LLMs and interaction strategies.

4. Data Model

This section specifies the JSON schemas and interface conventions used to represent tasks and to enable structured interaction between the LLM agent and the emulated environment.

4.1. Task Definition Schema

Each configuration task is defined as a JSON object with the following structure:

```
{
  "task_name": "Static Routing",
  "intents": [
    "NewYork: create a static route pointing to the Loopback0 on
    Washington, traffic should pass the 192.168.1.0 network.",
    "NewYork: create a backup static route pointing to the Loopback0
    on Washington, administrative distance should be 100."
    ...
  ],
  "topology": {
    "nodes": ["NewYork", "Washington"],
    "links": [
      "NewYork S0/0 <-> Washington S0/0 ",
      "NewYork S0/1 <-> Washington S0/1"
    ]
  },
  "startup_configs": {
    "NewYork": "!\r\nversion 12.4\r\nservice timestamps
    debug datetime msec\r\n...",
    "Washington": "!\r\nversion 12.4\r\nservice timestamps
    debug datetime msec\r\n...",
  },
  "ground_truth_configs": {
    "NewYork": [
      "ip route 2.2.2.0 255.255.255.252 192.168.1.2",
      "ip route 2.2.2.0 255.255.255.252 192.168.2.2 100"
    ],
    ...
  },
  "ground_truth_reasoning": "NewYork to Washington Loopback
  (primary path): add a static route for Washington's
  Loopback0 network (2.2.2.0/30) pointing to the
  next-hop 192.168.1.2...",
  "testcases": [
    {
      "name": "Static Route from NewYork to Washington",
      "expected_result": {
        "protocol": "static",
        "next_hop": "192.168.1.2"
      }
    },
    ...
  ]
}
```

4.2. Agent-Network Interface (ANI)

The Agent-Network Interface defines the minimal API primitives necessary for intent-driven configuration. Each primitive uses JSON-RPC style request/response with the following methods:

1. *get-topology*

   * *Request*:

     ```
     {
       "method": "get-topology",
       "params": {
         "devices": ["R1", "R2", ...]
       }
     }
     ```

   * *Response*:

     ```
     {
       "topology": {
         "nodes": [...],
         "links": [...]
       }
     }
     ```

   * *Description*: Returns the full topology for the specified subset of devices. If "devices" is empty or omitted, returns the entire topology.

2. *get-running-cfg*

   * *Request*:

     ```
     {
       "method": "get-running-cfg",
       "params": {
         "device": "R1"
       }
     }
     ```

   * *Response*:

```
{
  "running_config": "
   interface Gig0/0
   ip address 192.168.1.1 255.255.255.255
   ...
  "
}
```

* *Description*: Retrieves the active (running) configuration of the specified device.

3. *update-cfg*

* *Request*:

```
{
  "method": "update-cfg",
  "params": {
    "device": "R1",
    "commands": [
      "configure terminal",
      "ip route 2.2.2.0 255.255.255.252 192.168.1.2"
    ]
  }
}
```

* *Response*:

```
{
  "results": [
    { "command": "configure terminal", "status": "success" },
    {
    "command": "ip route 2.2.2.0 255.255.255.252 192.168.1.2",
    "status": "success" }
  ]
}
```

* *Description*: Applies a sequence of CLI commands to the specified device. Returns per-command status and any error messages.

4. *execute_validation*

* *Request*:

```
{
  "method": "execute_validation",
  "params": {
    "device": "R1",
    "command": "show ip route 2.2.2.0 255.255.255.252"
  }
}
```

* *Response*:

```
{
  "output": "S 2.2.2.0/30 [1/0] via 192.168.1.2"
}
```

* *Description*: Executes a read-only command on the specified device and returns its output. Must not alter device state.

4.3. Task Evaluation Interface

After the agent signals completion, the framework uses the Task Evaluation Interface to retrieve results:

* *export-final-cfg*

  - *Request*:

```
{
  "method": "export-final-cfg"
}
```

  - *Response*:

```
{
  "configs": {
    "R1": "!\nversion 15.2\n...",
    "R2": "!\nversion 15.2\n..."
  }
}
```

  - *Description*: Returns the final running-configuration of each device.

* *run-testcases*

  - *Request*:

```
{
  "method": "run-testcases",
  "params": {
    "testcases": [
      {
        "device": "R1",
        "commands": ["show ip route 2.2.2.0 255.255.255.252"],
        "expected_output": "S 2.2.2.0/30 [1/0] via 192.168.1.2"
      },
      ...
    ]
  }
}
```

- *Response*:

```
{
  "results": [
    {
      "name": "Verify primary static route on R1",
      "status": "pass"
    },
    {
      "name": "Verify backup static route on R1",
      "status": "fail"
    }
  ]
}
```

- *Description*: Executes each verification command sequence on the appropriate device and compares actual output against expected_output (regular expression). Returns pass/fail for each testcase.

5. MCP-Based Implementation

The Model Context Protocol (MCP) provides a standardized approach for implementing the Agent-Network Interface (ANI). This section describes how MCP can be applied to NetConfBench for LLM-driven network configuration evaluation.

5.1. Benefits of MCP Integration

Integrating MCP into NetConfBench provides several advantages:

1. *Standardization*: MCP provides a uniform interface for tool invocation across different LLM implementations and network environments.

2. *Vendor Abstraction*: The MCP server can handle vendor-specific command translation, allowing the LLM to work with high-level operations without needing detailed knowledge of each vendor's CLI syntax.

3. *Tool Extensibility*: New network operations can be easily added as MCP tools without modifying the LLM agent implementation.

4. *Traceability*: The structured MCP communication protocol enables detailed logging of all tool invocations and results, facilitating debugging and analysis.

5. *Ecosystem Integration*: MCP-enabled network tools can potentially be reused across different AI applications beyond network configuration evaluation.

5.2. MCP Tool Definitions for ANI Operations

This subsection provides the complete MCP tool definitions for the four core Agent-Network Interface operations: get-topology, get-running-cfg, update-cfg, and execute_validation. These definitions use JSON Schema to specify tool parameters and enable LLMs to understand and invoke network operations through the MCP protocol.

5.2.1. 1. get_topology

This tool provides network topology information in a format interpretable by the LLM, returning topology for specified devices or the entire network if no devices are specified.

```
{

  "name": "get_topology",
  "description": "Retrieve network topology information including
   nodes and their interconnections. Returns topology for
   specified devices or entire network if no devices specified.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "devices": {
        "type": "array",
        "items": {
          "type": "string"
        },
        "description": "List of device names. Leave empty for
        entire network topology."
      }
    }
  }
}
```

Usage Example:

```
{
  "method": "tools/call",
  "params": {
    "name": "get_topology",
    "arguments": {
      "devices": ["R1", "R2", "R3"]
    }
  }
}
```

5.2.2. 2. get_running_config

This tool enables the agent to obtain the active configurations of specified devices, providing essential context for planning subsequent updates.

```
{
  "name": "get_running_config",
  "description": "Retrieve the active running configuration
  from a network device. Returns the complete configuration
  as a text string.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "device": {
        "type": "string",
        "description": "Device name or identifier to retrieve
         configuration from"
      }
    },
    "required": ["device"]
  }
}
```

Usage Example:

```
{
  "method": "tools/call",
  "params": {
    "name": "get_running_config",
    "arguments": {
      "device": "R1"
    }
  }
}
```

5.2.3. 3. update_config

This tool allows the agent to apply new configuration commands and provides detailed feedback on their execution, including whether each command was accepted or resulted in any errors.

```
{
  "name": "update_config",
  "description": "Apply configuration commands to a network
   device. Executes a sequence of CLI commands and returns
    detailed status for each command.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "device": {
        "type": "string",
        "description": "Device name or identifier to apply
         configuration to"
      },
      "commands": {
        "type": "array",
        "items": {
          "type": "string"
        },
        "description": "Ordered list of CLI commands to
         execute on the device"
      }
    },
    "required": ["device", "commands"]
  }

}
```

Usage Example:

```
{
  "method": "tools/call",
  "params": {
    "name": "update_config",
    "arguments": {
      "device": "R1",
      "commands": [
        "configure terminal",
        "interface GigabitEthernet0/0",
        "ip address 192.168.1.1 255.255.255.0",
        "no shutdown"
      ]
    }
  }
}
```

4. execute_cmd

This tool accepts a device name and a read-only command string as parameters and returns the resulting output. It must not alter the device state and is intended for validation and status inspection.

```
{
  "name": "execute_validation",
  "description": "Execute a read-only validation command
   on a network device to verify configuration or check
    device status. This command must not alter the
    device state.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "device": {
        "type": "string",
        "description": "Device name or identifier to
        execute command on"
      },
      "command": {
        "type": "string",
        "description": "Read-only command to execute
         (e.g., show commands)"
      }
    },
    "required": ["device", "command"]
  }
}
```

Usage Example:

```
{
  "method": "tools/call",
  "params": {
    "name": "execute_validation",
    "arguments": {
      "device": "R1",
      "command": "show ip route 2.2.2.0 255.255.255.252"
    }
  }
}
```

These four tools form the core MCP interface for NetConfBench. The MCP server must register these tools and handle the translation between MCP tool invocations and actual device communication protocols (CLI, NETCONF, RESTCONF, etc.). The JSON Schema definitions in inputSchema enable LLMs to automatically understand parameter requirements and generate valid tool calls.

5.3. Additional MCP Tools for Advanced Scenarios

Beyond the four core ANI operations, additional MCP tools can be defined for more complex scenarios. The following examples demonstrate extended tool definitions:

5.3.1. batch_configure_devices

For batch operations across multiple devices:

```
{
  "name": "batch_configure_devices",
  "description": "Apply configuration commands to
   multiple network devices simultaneously",
  "inputSchema": {
    "type": "object",
    "properties": {
      "device_ips": {
        "type": "array",
        "items": {"type": "string"},
        "description": "List of device IP addresses"
      },
      "commands": {
        "type": "array",
        "items": {"type": "string"},
        "description": "CLI command sequence to execute"
      },
      "credential_id": {
        "type": "string",
        "description": "Authentication credential
         identifier"
      }
    },
    "required": ["device_ips", "commands"]
  }
}
```

5.3.2. check_device_status

For comprehensive device health monitoring:

```
{
  "name": "check_device_status",
  "description": "Check operational status of network
   devices including CPU, memory, and interface metrics",
  "inputSchema": {
    "type": "object",
    "properties": {
      "device_ip": {
        "type": "string",
        "description": "Device IP address to check"
      },
      "metrics": {
        "type": "array",
        "items": {
          "enum": ["cpu", "memory", "interface"]
        },
        "description": "List of metrics to retrieve"
      }
    },
    "required": ["device_ip", "metrics"]
  }
}
```

These additional tools demonstrate the extensibility of the MCP approach, allowing the framework to support advanced scenarios such as batch operations and comprehensive device monitoring.

6. Security Considerations

LLM-driven network configuration introduces risks such as unintended or malicious commands, emulator vulnerabilities, and data exposure; to mitigate these, NetConfBench should enforce strict input validation (e.g., YANG/XML schema checks), run emulated devices in isolated sandboxes with limited privileges, encrypt and restrict access to task definitions and logs, employ human-in-the-loop approval for generated configurations, and use curated prompt templates and fine-tuning to reduce LLM hallucinations. Validation endpoints must enforce read-only execution (e.g., execute-validation) to prevent unintended state changes. Where appropriate, human-in-the-loop approval should gate privileged write operations (update-cfg/update-config) identified as high-impact.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/rfc/rfc6241>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/rfc/rfc7950>.

8.2. Informative References

[A2023] Hat, R., "Ansible", 2023.

[Fuad2024] Fuad, A., Ahmed, A. H., Riegler, M. A., and T. Cicic, "An intent-based networks framework based on large language models", 2024.

[Kreutz2014] Kreutz, D., Ramos, F. M. V., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and S. Uhlig, "Software-defined networking: A comprehensive survey", 2014.

[Lira2024] Lira, O. G., Caicedo, O. M., and N. L. S. da. Fonseca, "Large language models for zero touch network configuration management", 2024.

[Liu2024] Liu, C., Xie, X., Zhang, X., and Y. Cui, "Large language models for networking: Workflow, advances and challenges", 2024.

[Long2025] Long, S., Tan, J., Mao, B., Tang, F., Li, Y., Zhao, M., and N. Kato, "A Survey on Intelligent Network Operations and Performance Optimization Based on Large Language Models", 2025.

[Wang2024NetConfEval] Wang, C., Scazzariello, M., Farshin, A., Ferlin, S., Kostic, D., and M. Chiesa, "Netconfeval: Can llms facilitate network configuration?", 2024.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Yong Cui
Tsinghua University
Beijing, 100084
China
Email: cuiyong@tsinghua.edu.cn
URI: http://www.cuiyong.net/

Chang Liu
Tsinghua University
Beijing, 100084
China
Email: liuchang23@mails.tsinghua.edu.cn

Xiaohui Xie
Tsinghua University
Beijing, 100084
China
Email: xiexiaohui@tsinghua.edu.cn

Chenguang Du
Zhongguancun Laboratory
Beijing, 100094
China
Email: ducg@zgclab.edu.cn