dsm2cli: An Observable Pipeline for Translating Network Intents into Multivendor CLI with Independent Semantic Assessment

Jerônimo Menezes[1,2], Leonardo Bitzki[1,2], Diego Kreutz[2]
Gefte Almeida[2], Marcio Pohlmann[1,2], Rodrigo Mansilha[2]

[1]Universidade Federal do Rio Grande do Sul (UFRGS)

[2]AI Labs | PPGES, Universidade Federal do Pampa (UNIPAMPA)

{jeronimomenezes,leonardobitzki,geftealmeida,marciopohlmann}.aluno@unipampa.edu.br
{diegokreutz,rodrigomansilha}@unipampa.edu.br

Abstract. *Network configuration in multivendor environments remains challenging due to semantic inconsistencies and limited observability in intent-to-CLI workflows. We present dsm2cli, an observable pipeline that translates structured intents into multivendor CLI with independent LLM-based semantic assessment. The approach separates translation from evaluation and produces artifacts such as traceability data, votes, verdicts, and telemetry under a unified contract, without targeting formal verification. Across six scenarios, four configurations, and 72 executions, results show that dsm2cli enables comparison of strategies, exposes semantic failures missed by generation-only workflows, and highlights trade-offs between robustness, cost, and latency.*

1. Introduction

Configuring and operating heterogeneous networks remains challenging, particularly in *multivendor* environments, where syntactic and semantic differences across vendors hinder standardization, portability, and automation [Wallin et al. 2011, Hollósi et al. 2024, Wei et al. 2025]. In practice, network configuration still relies heavily on vendor-specific templates and ad hoc logic, resulting in brittle workflows that are difficult to reuse, inspect, and adapt to new scenarios [Leivadeas and Falkner 2023, Ansible Community 2026, Hollósi et al. 2024]. These limitations are exacerbated by subtle semantic divergences across devices, where configurations that are syntactically valid may still fail to enforce the intended operational state.

Recent advances have explored the use of large language models (LLMs) to translate high-level intents into device-specific configurations [Long et al. 2025, Hong et al. 2025, Tu et al. 2025, Wei et al. 2025, Lira et al. 2024, Tageldien et al. 2025]. While promising, these approaches largely treat translation as a generation problem, offering limited support for systematic comparison, inspection, and semantic assessment. In particular, current solutions rarely provide mechanisms to assess whether generated configurations adhere to the intended network state under a unified and reproducible evaluation workflow, or to compare alternative translation strategies under controlled conditions.

This gap is critical in realistic scenarios, where syntactic validity is insufficient and correctness depends on high-level intent. Generating CLI commands is not enough; their

semantic adherence must be assessed, and execution evidence captured for inspection and reproducibility. Evaluating translation strategies also requires controlled pipelines to analyze trade-offs in quality, cost, and latency. Intent-Based Networking (IBN) provides a conceptual foundation for addressing this problem by expressing network behavior through declarative, vendor-independent abstractions, such as *Desired State Models* (DSMs) [Zeydan and Turk 2020, Ansible Community 2026]. However, existing approaches do not provide a unified framework to translate DSMs into CLI configurations while also supporting independent semantic assessment and execution observability.

In this paper, we present *dsm2cli*, an observable pipeline that translates structured network intents into multivendor CLI configurations with independent semantic assessment by LLM evaluators. The approach separates translation from evaluation, using independent models to assess adherence between configurations and input intent, while producing structured artifacts such as traceability data, votes, verdicts, and telemetry. It does not target formal verification or production validation, but supports inspection, comparison, and complementary validation workflows. We evaluate *dsm2cli* across six multivendor scenarios, four pipeline configurations, and 72 executions, showing it enables comparison of translation strategies, exposes semantic failures missed by generation-only workflows, and highlights trade-offs between robustness, cost, and latency. The main contributions of this work are:

- an observable pipeline for translating DSMs into multivendor CLI configurations under a unified execution contract;
- an independent semantic assessment mechanism based on multiple evaluators and aggregated voting;
- a structured artifact model that provides traceability and telemetry for reproducible and comparable executions;
- an experimental evaluation demonstrating how the approach enables controlled analysis of translation strategies and their trade-offs.

The remainder of this paper is organized as follows: Section 2 discusses related work; Section 3 presents the system architecture; Section 4 reports experimental results; and Section 5 concludes the paper.

2. Related Work

Table 1 compares prior work across four dimensions: intent abstraction, output artifact, multivendor support, and verification or assessment capability. Next, we highlight key limitations and position our contribution accordingly.

Model-driven approaches. Systems such as NCS [Wallin et al. 2011] and YANG-based pipelines use structured models to reduce vendor dependencies and improve consistency. However, they rely on predefined schemas and tightly coupled workflows, limiting flexibility and preventing observable, reproducible comparison of alternative pipelines.

High-level abstractions. Approaches such as NetBuddy [Wang et al. 2023] raise abstraction through natural language or policy representations. While improving usability, they provide limited support for multivendor evaluation, controlled comparison, and structured semantic assessment of generated outputs.

LLM-based translation. Recent work leverages LLMs for intent-to-configuration translation (e.g., IBN+LLM [Tu et al. 2025], S-Witch [Jeong et al. 2024], INTA [Wei et al. 2025]). These approaches typically treat translation as a black-box generation task, with assessment either absent or tightly coupled to the generation process, and without a unified execution model for controlled comparison.

Table 1. Comparison of related approaches across input abstraction, target artifact, multivendor support, and verification or assessment capabilities.

Work	*Input*	*Target*	*Multivendor*	*Verification*
NCS [Wallin et al. 2011]	NETCONF/YANG	CFG/CLI	✓	✓
INTA [Wei et al. 2025]	CFG/Intent	CFG/CLI	✓	✓
IBN+LLM [Tu et al. 2025]	NL/Intent	CFG	✓	✗
S-Witch [Jeong et al. 2024]	NL	CLI	✗	✓
NetBuddy [Wang et al. 2023]	NL	CFG/API	✗	✓
YANG+LLM [Hollósi et al. 2024]	YANG	NETCONF	✗	✓
dsm2cli (this work)	**DSM**	**CLI**	✓	✓

Limitations. Across all categories, two limitations persist: the lack of separation between translation and assessment, and the absence of observable, reproducible execution frameworks. These limitations hinder independent inspection and systematic comparison of alternative strategies.

Our contribution. We address these gaps with *dsm2cli*, an observable pipeline for DSM-to-CLI translation. The system separates translation from semantic assessment by LLM evaluators and operates under a unified contract, producing structured artifacts such as votes, verdicts, traceability, and telemetry. This enables reproducible experiments and controlled comparison, exposing semantic failures and trade-offs not visible in generation-only approaches.

3. System Architecture

The *dsm2cli* [1] architecture addresses three limitations identified in existing approaches: the absence of separation between translation and semantic assessment, the lack of observable and reproducible execution pipelines, and the inability to systematically compare alternative translation strategies.

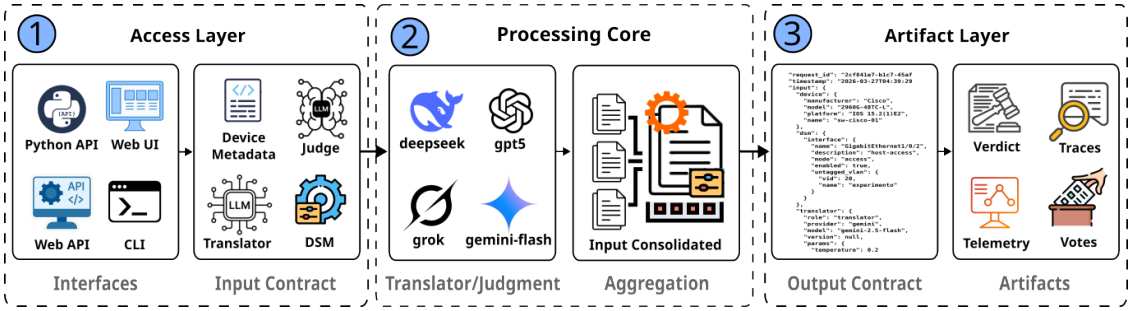

Figure 1. Overview of the *dsm2cli* architecture.

[1] https://github.com/net2d-community/dsm2cli

Figure 1 illustrates the overall architecture, organized into three layers: access, processing core, and artifacts, each with distinct responsibilities for interaction, execution, and observability.

3.1. Access Layer and Unified Contract

The access layer exposes the pipeline through four interfaces: Python module, CLI, Web API, and web UI, all sharing a unified input–output contract. This contract is the foundation of the observable and reproducible execution model: it specifies the target device, the DSM, and the execution components (translator and evaluators), enabling different pipeline configurations to be instantiated declaratively without modifying the external interface. The response assembles the generated CLI, individual votes, an aggregated verdict, and structured artifacts including traceability data, classifications, and telemetry, producing a complete and self-contained execution record. Request and response examples are provided in Appendices B and C.

3.2. Processing Core

Figure 2 details the pipeline execution flow. The processing core coordinates execution from the structured request: the input is validated and the DSM is forwarded to the translator, which produces the CLI configuration for the target device. This enforces a strict separation between translation and semantic assessment, a key design decision that prevents the semantic judgment stage from being conflated with or biased by the generation process itself.

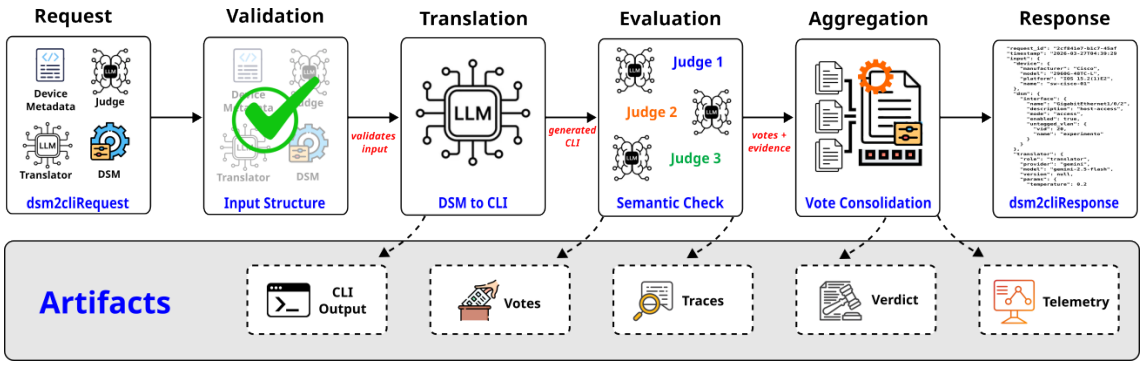

Figure 2. *dsm2cli* processing core (pipeline).

The generated CLI is evaluated by independent LLMs for semantic adherence to the input DSM, producing votes with structured evidence that are aggregated into a final verdict alongside pipeline artifacts. This execution exposes not only translation outcomes but also stage behavior, revealing semantic failures missed by generation-only workflows. The architecture treats semantic assessment as a native stage, enabling structured evaluation of intent adherence, while allowing flexible composition where translators, evaluators, and prompts can be modified without affecting the interface or artifact structure.

3.3. Artifact Layer

The artifact layer aggregates execution outputs, including generated CLI, votes, verdicts, traceability evidence, and telemetry. Structured evidence links DSM elements to CLI segments, enabling localization of semantic deviations, such as the missing `enabled` state

in Huawei scenarios, while metrics like latency and token consumption support trade-off analysis and reproducibility. These artifacts support validation workflows, including expert inspection, comparison across pipeline profiles, and reuse in studies with automated analyzers or execution-based validation. Together, these layers realize a unified execution contract as an observable pipeline where translation, semantic assessment, and telemetry are first-class concerns, positioning *dsm2cli* as a controlled environment for analyzing intent-to-CLI strategies in multivendor settings.

4. Experimental Evaluation

This section presents the experimental evaluation of *dsm2cli* across a controlled multivendor matrix. The evaluation is designed to demonstrate three capabilities introduced by the proposed approach: (i) the ability to execute and compare different translation pipelines under a unified input–output contract; (ii) the ability to detect semantic failures through independent semantic assessment, which would remain undetected in generation-only workflows; and (iii) the ability to expose trade-offs between robustness, cost, and latency across pipeline configurations.

4.1. Experimental Setup

The evaluation was conducted over six reproducible multivendor scenarios, obtained by combining three use cases with two target vendors: **UC1** (L2 port in access mode), **UC2** (L2 port in tagged mode with an untagged VLAN), and **UC3** (IPv4 addressing on an L3 interface), each instantiated for both **Cisco** and **Huawei** devices. Four pipeline profiles were defined by varying only the primary translator while keeping three independent evaluators distinct from the translator to prevent self-assessment: **P1** (OpenAI), **P2** (DeepSeek), **P3** (Gemini), and **P4** (XAI/Grok). Each profile was executed three times per scenario, yielding $4 \times 6 \times 3 = 72$ total executions.

For each execution, *dsm2cli* produced: (i) the generated CLI; (ii) individual votes from three independent evaluators; (iii) an aggregated verdict; and (iv) telemetry comprising per-stage latency and token consumption. In this study, correctness is defined as agreement under the semantic assessment policy: a verdict is correct when a majority of three evaluators approve it. The version used applies prompt refinements emphasizing semantic adherence, normative objects, and suppression of speculative inferences.

4.2. Functional Results

Table 2 summarizes the functional behavior of the four pipeline profiles across the six scenarios. Each cell reports the number of correct executions out of three repetitions and the mean number of favorable votes (μ) across the three evaluators (max. $\mu = 3.0$).

Profiles P1 and P3 achieved full correctness across all 18 executions under the adopted assessment policy. P3 (Gemini) reached perfect evaluator consensus (μ=3.00) in five of six scenarios, with a minor deviation only in UC3–Cisco (μ=2.67). P1 (OpenAI) exhibited equally robust correctness, with deviations in evaluator consensus restricted to UC3–Cisco (μ=2.67), indicating that the generated CLI was accepted but not unanimously considered optimal. Both profiles demonstrated stable behavior across all three repetitions per scenario, with no variance in the correctness outcome.

All profiles succeeded on Cisco scenarios. UC1, UC2, and UC3 on Cisco were accepted by all four profiles across all three repetitions, suggesting that Cisco CLI conventions for the evaluated use cases are sufficiently well represented in the translators' training data to produce semantically acceptable outputs regardless of the specific model used.

Table 2. Functional results per scenario and pipeline profile across three repetitions. Each cell reports correct executions out of three and mean favorable votes μ (max. 3.0).

Scenario	P1 (OpenAI)	P2 (DeepSeek)	P3 (Gemini)	P4 (XAI)
UC1 – Cisco	3/3; μ=3.00	3/3; μ=2.33	3/3; μ=3.00	3/3; μ=3.00
UC2 – Cisco	3/3; μ=3.00	3/3; μ=2.67	3/3; μ=3.00	3/3; μ=2.67
UC3 – Cisco	3/3; μ=2.67	3/3; μ=3.00	3/3; μ=3.00	3/3; μ=3.00
UC1 – Huawei	3/3; μ=3.00	0/3; μ=1.00	3/3; μ=3.00	0/3; μ=1.00
UC2 – Huawei	3/3; μ=3.00	3/3; μ=2.67	3/3; μ=3.00	3/3; μ=2.00
UC3 – Huawei	3/3; μ=3.00	0/3; μ=0.00	3/3; μ=3.00	0/3; μ=1.00
Total	**18/18**	**12/18**	**18/18**	**12/18**

Huawei scenarios exposed discriminating failures in P2 and P4. Profiles P2 (DeepSeek) and P4 (XAI) failed consistently in UC1–Huawei and UC3–Huawei: all three repetitions were rejected in both cases, yielding 0/3 correct executions with zero variance in outcome. This consistency rules out random generation noise and indicates systematic semantic deficiencies in the translated CLI for those scenarios. The most critical case is P2–UC3–Huawei, which obtained μ=0.00 — the only instance in the experiment where all three independent evaluators unanimously rejected every repetition. Under the adopted judging policy, this represents the strongest observed case of consistent semantic failure: the generated CLI was not merely imprecise, but judged structurally inconsistent with the intended network state as expressed in the DSM.

Independent semantic assessment reveals failures that syntactic inspection would not detect. In all failing cases, the generated CLI was syntactically plausible and would not be flagged by a parser or a generation-only pipeline. The evaluators identified semantic omissions, most notably the absence of an explicit `enabled` state on Huawei interfaces, where the CLI did not guarantee that the interface would be operationally active after configuration. This class of failure is invisible without assessment tied to the input intent, which directly addresses the limitation identified in generation-only approaches such as IBN+LLM [Tu et al. 2025]. UC2–Huawei exhibited partial robustness in P4 (μ=2.00), suggesting that the same deficiency manifests with varying severity depending on the structural complexity of the use case.

4.3. Telemetry and Cost Analysis

Beyond functional correctness, *dsm2cli* produced structured telemetry enabling systematic analysis of execution cost and latency across profiles. Tables 3 and 4 report per-profile performance metrics averaged across all 18 executions (six scenarios, three repetitions).

P3 dominates on both correctness and efficiency. Gemini achieved full correctness (18/18), perfect evaluator consensus (σ=0.000), the lowest mean latency (51.957 s), and the second-lowest total token consumption (8124.9 tokens). This combination makes P3

Table 3. Telemetry summary per pipeline profile: correctness, mean evaluator votes, and mean execution latency across three repetitions.

Profile	Correct executions	Mean votes ($\mu \pm \sigma$)	Mean latency (s)
P1 (OpenAI)	18/18	2.944 ± 0.236	68.770 ± 42.699
P2 (DeepSeek)	12/18	1.944 ± 1.056	55.740 ± 34.094
P3 (Gemini)	18/18	3.000 ± 0.000	51.957 ± 27.763
P4 (XAI)	12/18	2.111 ± 0.832	53.218 ± 23.830

the Pareto-optimal profile in the evaluated matrix: no other profile achieves the same correctness at lower cost or lower latency.

P1 incurs the highest cost for equivalent correctness. OpenAI achieved the same 18/18 correctness as Gemini but at substantially higher resource expenditure: mean latency 32% higher (68.770 vs. 51.957 s) and translator token consumption 147% higher (3628.1 vs. 1467.4 tokens). The high variance in both latency ($\pm$42.699 s, coefficient of variation 62%) and translator token count ($\pm$1092.3) suggests that the OpenAI model produces outputs of highly variable verbosity depending on the scenario, which propagates cost to the evaluator stage: a more verbose CLI requires more tokens to analyze, explaining why total token consumption also remains elevated even though evaluator prompts are shared across profiles.

Latency variance is high across all profiles and must be interpreted cautiously. Coefficients of variation range from 52% (P4) to 62% (P1 and P2). This level of variance is consistent with provider-side factors including request queuing, model cold starts, and network conditions, which are external to the pipeline and not controlled in this experiment. Latency comparisons should therefore be treated as indicative rather than definitive. Controlled benchmarking under fixed infrastructure conditions is left as future work.

Table 4. Resource consumption per pipeline profile: mean translator token usage, mean total token usage, and mean processing throughput across three repetitions.

Profile	Translator tokens (mean)	Total tokens (mean)	Throughput (tokens/s)
P1 (OpenAI)	3628.1 ± 1092.3	8675.2 ± 1081.2	145.407 ± 41.070
P2 (DeepSeek)	1395.9 ± 34.0	8249.1 ± 1549.8	178.746 ± 57.828
P3 (Gemini)	1467.4 ± 40.8	8124.9 ± 1325.1	170.075 ± 42.022
P4 (XAI)	1298.5 ± 47.7	8068.1 ± 1769.4	172.513 ± 54.232

Total token consumption is structurally dominated by evaluators, not the translator. For P2, P3, and P4, translator tokens represent 17–20% of total token consumption (e.g., P4: 1298.5 of 8068.1). For P1, the translator accounts for 42% (3628.1 of 8675.2). This indicates that in efficient profiles, the assessment stage (three independent evaluators) is responsible for approximately 80% of total token cost, a consequence of the independent evaluation design. This ratio is an inherent characteristic of the architecture and a deliberate trade-off: the cost of independent semantic assessment is also a practical cost of increased inspection capability.

4.4. Discussion

Results show that *dsm2cli* is a controlled experimentation platform.

Controlled comparison under a unified contract. All profiles were executed under identical inputs, evaluation policies, and output schemas. The observed differences, namely full correctness for P1 and P3 and systematic failures for P2 and P4 in Huawei scenarios, are attributable solely to the translator, with all other components held constant. This level of isolation is not achievable in generation-only workflows.

Semantic failure detection beyond syntactic validity. Failures in P2 and P4 were not syntactic but semantic: the generated CLIs were well formed yet incomplete. Independent evaluators detected omissions such as the missing explicit `enabled` state on Huawei interfaces by reasoning against the DSM. This validates the separation between translation and semantic assessment. The case P2–UC3–Huawei (μ=0.00) shows that failures are systematic and reproducible under the adopted judging policy, and would be harder to identify in the absence of independent assessment.

Trade-off visibility across robustness, cost, and latency. Telemetry shows that correctness and efficiency are not correlated. P3 achieves the best overall balance, while P1 matches correctness at higher cost, and P2 and P4 reduce cost at the expense of robustness. These trade-offs are observable only because *dsm2cli* enforces a unified contract and produces structured telemetry, a capability absent in systems such as INTA [Wei et al. 2025] and NetBuddy [Wang et al. 2023].

The assessment model is defined within its scope: semantic correctness is evaluated by independent LLMs under a unified policy, not constituting formal verification or replacing human or execution-based validation. Instead, *dsm2cli* provides a controlled architecture and structured artifacts to support inspection, comparison, and complementary validation.

5. Conclusions and Future Work

This paper presented *dsm2cli*, an observable pipeline for translating structured network intents (DSMs) into multivendor CLI configurations, separating translation from independent semantic assessment by LLM evaluators under a unified input–output contract. The architecture produces structured artifacts, including votes, verdicts, traceability data, and telemetry, enabling reproducible and comparable executions across pipeline configurations.

Evaluation across six scenarios, four pipeline profiles, and 72 executions shows that the unified contract enables controlled comparison, independent semantic assessment exposes failures difficult to detect in generation-only workflows, and telemetry reveals trade-offs between robustness, cost, and latency. In this setting, P3 achieved the best balance, while P1 incurred higher cost for equivalent correctness under the adopted policy.

The contribution of *dsm2cli* should be interpreted within its scope. The approach is not aimed at formal verification or production-grade validation, but provides a controlled architecture for translation and semantic assessment, with structured artifacts supporting inspection, comparison, and complementary validation workflows.

Availability and demonstration details are provided in Appendix A. Future work will expand scenario coverage, explore additional assessment strategies, and integrate complementary validation in real or emulated environments, positioning *dsm2cli* as a foundation for reproducible evaluation of intent-to-CLI pipelines.

Acknowledgments

This work was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq) under Grant 409743/2025-9; by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Finance Code 001; by the Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS) under Grants 24/2551-0001368-7, 24/2551-0000726-1, and 25/2551-0002572-9; and by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) under Grants #2020/05183-0 and #2023/00816-2.

References

Ansible Community (2026). Templating (jinja2) — ansible community documentation. Online documentation. Accessed: 2026-03-26.

Hollósi, G., Ficzere, D., and Varga, P. (2024). Generative AI for low-level NETCONF configuration in network management based on YANG models. In *2024 20th International Conference on Network and Service Management (CNSM)*, pages 1–7. IEEE.

Hong, J., Tu, N. V., and Hong, J. W.-K. (2025). A comprehensive survey on llm-based network management and operations. *International Journal of Network Management*, 35(6):e70029.

Jeong, E.-D., Kim, H.-G., Nam, S., Yoo, J.-H., and Hong, J. W.-K. (2024). S-witch: Switch configuration assistant with LLM and prompt engineering. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 1–7. IEEE.

Leivadeas, A. and Falkner, M. (2023). A survey on intent-based networking. *IEEE Communications Surveys & Tutorials*, 25(1):625–655.

Lira, O. G., Caicedo, O. M., and da Fonseca, N. L. (2024). Large language models for zero touch network configuration management. *IEEE Communications Magazine*, 63(7):146–153.

Long, S., Tan, J., Mao, B., Tang, F., Li, Y., Zhao, M., and Kato, N. (2025). A survey on intelligent network operations and performance optimization based on large language models. *IEEE Communications Surveys & Tutorials*, 27(6):3915–3949.

Tageldien, M., Selim, B., and Sboui, L. (2025). Large language models in intent-based networking: a comprehensive survey across the intent lifecycle. In *2025 International Telecommunications Conference (ITC-Egypt)*, pages 810–817. IEEE.

Tu, N., Nam, S., and Hong, J. W.-K. (2025). Intent-based network configuration using large language models. *International Journal of Network Management*, 35(1):e2313.

Wallin, S., Larsson, P., and Folkesson, P. (2011). Automating network and service configuration using netconf and yang. In *LISA'11: Large Installation System Administration Conference*. USENIX.

Wang, C., Scazzariello, M., Farshin, A., Kostic, D., and Chiesa, M. (2023). Making network configuration human friendly. arXiv preprint.

Wei, Y., Xie, X., Hu, T., Zuo, Y., Chen, X., Chi, K., and Cui, Y. (2025). INTA: Intent-based translation for network configuration with LLM agents. arXiv preprint.

Zeydan, E. and Turk, Y. (2020). Recent advances in intent-based networking: A survey. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5.

Appendices

A. Availability, Demonstration, and Reproducibility

The *dsm2cli* artifact associated with this work is publicly available at `https://github.com/net2d-community/dsm2cli`, including source code, documentation, usage examples, and scripts required for execution and evaluation. The environment can be instantiated in a standardized manner using Docker Compose, enabling rapid deployment of both the Web API and the Web interface with minimal configuration effort. Alternatively, local execution via Python is also supported.

The repository provides example requests, automated tests, and a dedicated directory for reproducing the experiments reported in this paper. All components operate under a unified input–output contract, ensuring consistency across executions and enabling inspection of the generated artifacts.

The planned demonstration is centered on the Web UI. Participants will be able to load or inspect a structured DSM, select a translator and a set of evaluators, execute the pipeline, inspect the generated CLI, review individual evaluator votes and the aggregated verdict, and observe telemetry and structured artifacts produced during execution. This demonstration focuses on the translation-and-assessment workflow itself, highlighting how the proposed architecture supports controlled comparison and artifact-based inspection.

The demonstration can be performed on modest hardware (e.g., a standard laptop running an Ubuntu 20.04 virtual machine with 4 vCPUs and 8 GB RAM), without requiring specialized infrastructure or dedicated network environments, as models are accessed remotely. In addition, an offline (stub) mode is available for inspecting the execution contract and the structure of generated artifacts without relying on external providers, further improving accessibility and reproducibility.

Although the current demonstration does not aim at execution-based validation in real or emulated devices, the artifacts produced by *dsm2cli* are designed to support complementary validation workflows in future studies, including expert inspection, additional automated analyzers, and integration with real or emulated environments.

B. Example of a *dsm2cli* request structure

Listing 1 illustrates the main elements of a *dsm2cli* request. The structure makes explicit four components: (i) target device metadata, which identifies the intended vendor and platform; (ii) the DSM, which encodes the desired network state in a structured form; (iii) the translator configuration, which specifies the model responsible for CLI generation; and (iv) the evaluator set, which defines the independent semantic assessment stage. This explicit decomposition is central to the unified contract adopted by the pipeline.

Lista 1. Example of a *dsm2cli* request structure

```
{
  "device": {
    "manufacturer": "cisco",
    "platform": "nxos"
  },
  "dsm": {
    "interface": {
      "name": "Ethernet1/1",
      "mode": "access",
```

```
      "enabled": true,
      "untagged_vlan": { "vid": 20, "name": "usuarios" }
    }
  },
  "translator": {
    "provider": "openai",
    "model": "gpt-4o"
  },
  "judges": [
    { "provider": "openai", "model": "gpt-4o-mini" },
    { "provider": "deepseek", "model": "deepseek-chat" }
  ]
}
```

C. Example of a structured output generated by the *dsm2cli* pipeline

Listing 2 illustrates the main result groups returned by the pipeline. The response is organized into four parts: (i) the translation output, including status and generated CLI lines; (ii) the evaluator votes, which record the judgments produced during semantic assessment; (iii) the aggregated summary, which reports the final verdict under the adopted policy; and (iv) telemetry, which captures execution metadata such as latency. Together, these elements form a self-contained execution record that can be inspected, compared across profiles, and reused in complementary validation workflows.

Lista 2. Example of a structured output generated by the *dsm2cli* pipeline

```
{
  "translation": {
    "status": "success",
    "cli_lines": [
      "interface Ethernet1/1",
      "switchport",
      "switchport mode access",
      "switchport access vlan 20",
      "no shutdown"
    ]
  },
  "votes": [
    { "judge": "gpt-4o-mini", "verdict": "correct" },
    { "judge": "deepseek-chat", "verdict": "correct" }
  ],
  "summary": {
    "final_verdict": "correct"
  },
  "telemetry": {
    "translator_latency_ms": 1820
  }
}
```