Network Modeling (NETMOD)
Internet-Draft
Intended status: Standards Track
Expires: 19 October 2026

C. Feng
Ruijie Networks

17 April 2026

NAIM: A Canonical Data Format for AI-Assisted YANG Modeling
draft-feng-netmod-naim-00

Abstract

This document defines the NAIM (Natural AI Interface Modeling) Document format, a canonical JSON representation and a derived human-readable Markdown view designed to serve as a structured semantic intermediate representation between natural language descriptions and YANG data models [RFC7950].

NAIM addresses a recognized gap in the YANG authoring workflow: direct conversion from natural language to YANG is error-prone because essential modeling semantics — including configuration versus state distinction, list key identification, constraint expressions, operational preconditions, and cross-module relationships — are routinely absent or ambiguous in natural language input. NAIM makes these semantics explicit, structurally consistent, and machine-verifiable before YANG generation occurs.

This document standardizes the canonical JSON structure, Markdown rendering rules, node template families, type representation, visibility and deviation declarations, and interoperability expectations for tools that produce and consume NAIM artifacts. It does not standardize AI model selection, prompt engineering strategies, dialogue policies, implementation-internal workflow logic, or runtime execution systems.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 October 2026.

Table of Contents

1. Introduction

The YANG data modeling language [RFC7950] is the foundation of model-driven network management and is widely deployed in NETCONF [RFC6241], RESTCONF [RFC8040], and related systems. Despite its importance, authoring high-quality YANG modules requires simultaneous fluency in network protocol behavior, data modeling methodology, and YANG syntax. This combination makes YANG authoring a specialized skill that limits the pace at which network management models can be

developed and maintained.

Recent progress in large language model (LLM) technology has created an opportunity to partially automate and assist the YANG authoring process. However, this opportunity cannot be fully realized through direct conversion from natural language to YANG. The gap between natural language and YANG is not primarily syntactic; it is semantic. Natural language descriptions routinely omit, conflate, or leave ambiguous precisely the distinctions that YANG modeling requires.

1.1. Problem Statement

Direct conversion from natural language to YANG exhibits three recurring failure modes.

Semantic ambiguity. Natural language descriptions commonly omit distinctions that are essential to correct YANG modeling. Whether a field represents configuration or operational state, whether an entity is a list keyed by an identifier or a singleton container, and whether two fields are siblings or parent and child are all questions that a natural language description frequently leaves unanswered. Without an explicit clarification step, an automated tool must guess — and guesses in structural modeling propagate into downstream protocol behavior.

Implicit constraint loss. Operational constraints, preconditions for node creation or deletion, cross-node dependencies, and side effects of configuration changes are often treated as common knowledge in human discourse and are therefore omitted from natural language descriptions. YANG models that omit these constraints are structurally valid but operationally incorrect.

Structural inference errors. Hierarchy relationships, augmentation targets, leafref paths, grouping reuse, and cross-module import requirements all require precise structural reasoning. Without a structured intermediate step, automated tools generate models that syntactically resemble correct YANG but contain incorrect structural relationships that are difficult to detect without domain expertise.

1.2. Motivation for an Intermediate Representation

A structured semantic intermediate representation addresses these failure modes by externalizing implicit knowledge into a machine-verifiable form before YANG generation is attempted. Rather than asking an automated system to traverse in one step from natural language to YANG, the workflow is decomposed into two controlled stages.

First, natural language input is converted into a structured, semantically explicit NAIM Document through a clarification and elicitation process. At this stage, missing or ambiguous information is identified and resolved. Second, the completed NAIM Document is transformed into a YANG module by deterministic tooling. Because the input to this stage is already structurally explicit and constraint-complete, the transformation can be made reliable.

The NAIM Document format defined in this document is the artifact that bridges these two stages. It is designed to be simultaneously suitable for automated processing, human review, and version control.

It is important to note that NAIM is not a JSON encoding of YANG. A NAIM Document can be transformed into YANG, but it carries semantics that YANG syntax does not represent, including operational descriptions, explanatory context, usage examples, cross-node relationship descriptions, and visibility conditions expressed in natural language. YANG is a target output of the NAIM workflow, not a semantic equivalent of NAIM.

1.3. Scope of This Document

This document defines the NAIM Document format. It is intended for implementors who build tools that produce, validate, transform, or consume NAIM Documents, and for protocol engineers who wish to understand the semantic representation choices made by the format.

This document does not specify a specific AI model or LLM backend, prompt engineering strategies, confidence scoring, YANG compilation internals, vendor-specific CLI generation, runtime network operation execution, or any protocol binding beyond the data format itself.

Informatively, the NAIM Document format may also serve as a candidate network-domain semantic artifact in Agentic Intent Network (AIN) style systems [AIN-ARCH]; such usage is outside the scope of this document.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

NAIM: Natural AI Interface Modeling.

NAIM Document: A structured artifact conforming to the canonical JSON schema defined in this document. A NAIM Document represents the semantics of one YANG module or one YANG submodule.

Canonical JSON Format: The normative JSON representation of a NAIM Document. All interoperable tools MUST treat the canonical JSON format as the authoritative source of NAIM Document content.

Markdown View: A human-readable rendering derived from the canonical JSON representation. The Markdown View is a presentation artifact; it is not normatively authoritative over the canonical JSON.

Node Template: A structured schema pattern describing one category of semantic model node within a NAIM Document.

Visibility Condition: A declaration associated with a node that specifies the conditions under which the node is present or applicable. Corresponds to YANG if-feature and when constructs.

Deviation Declaration: A structured statement describing the ways in which a specific device implementation departs from a base YANG module.

4. NAIM Document Format

4.1. Canonical JSON as Normative Source

A NAIM Document MUST be represented as a JSON object [RFC8259] that conforms to the schema expectations defined in this document.

The canonical JSON representation is the normative source of NAIM Document content. All interoperable tools that exchange NAIM Documents MUST treat the canonical JSON format as authoritative. The Markdown view defined in Section 5 is a derived presentation artifact; it does not override the canonical JSON. When any discrepancy exists between the canonical JSON and a corresponding Markdown view, the canonical JSON MUST take precedence.

4.2. Top-Level Object Structure

The top-level JSON object of a NAIM Document MUST contain:

* version: A string identifying the NAIM format version. For documents conforming to this specification, the value MUST be "1.0".

* Exactly one of module or submodule. A top-level object MUST NOT contain both.

The top-level object MAY additionally contain any of the following arrays: revisions, features, typedefs, groupings, identities, extensions, nodes.

A conforming NAIM Document representing usable model content SHOULD contain a nodes array with at least one node object.

4.3. The module Object

When the top-level object contains module, it represents a YANG main module. The module object MUST contain:

* name: A string. The YANG module name. It MUST conform to the identifier syntax defined in Section 6.2 of [RFC7950].

* description: A string. A human-readable description of the module’s purpose and scope.

The module object MAY contain: organization, contact, reference, and deviations (an array of deviation objects as defined in Section 10).

4.4. The submodule Object

When the top-level object contains submodule, it represents a YANG submodule. The submodule object MUST contain:

* name: A string. The submodule name.

* belongs-to: A string. The name of the YANG main module to which this submodule belongs.

The submodule object MAY contain a description field. A NAIM Document containing a submodule object MUST NOT also contain a module object at the top level.

4.5. Top-Level Arrays

4.5.1. revisions

The revisions array MAY be present. Each entry in the array SHOULD contain a revision field (a string in the format YYYY-MM-DD) and a description field. Each entry MAY also contain a reference field.

4.5.2. features

The features array MAY be present. Each entry represents a YANG feature and SHOULD contain name and description fields. Each entry MAY also contain status (which MUST be one of "current", "deprecated", or "obsolete" if present) and reference. The features array MAY be present in both module and submodule NAIM Documents, as YANG permits feature declarations in submodules.

4.5.3. typedefs

The typedefs array MAY be present. Each entry defines a reusable type. See Section 7.1.

4.5.4. groupings

The groupings array MAY be present. Each entry defines a reusable structural fragment. See Section 7.2.

4.5.5. identities

The identities array MAY be present. Each entry defines a YANG identity. See Section 7.3.

4.5.6. extensions

The extensions array MAY be present. Each entry defines an extension keyword. See Section 7.4. Extension use within nodes is described in Section 11.

4.5.7. nodes

The nodes array MAY be present. Each entry is a node object conforming to one of the node template families defined in Section 6. The nodes array is the primary structural content of a NAIM Document. When present, it MUST contain only objects that conform to a recognized node template family.

4.6. Path Conventions

All path fields in node objects MUST be expressed as absolute schema paths using the following conventions:

* The path MUST begin with "/".

* Path segments representing nodes in a specific YANG module MUST be qualified with the module name as a prefix, using the colon (":") separator. For example: "/ietf-interfaces:interfaces/interface".

* The prefix used MUST be the YANG module name, not the YANG prefix shorthand. This ensures path stability across implementations that may choose different YANG prefix values.

* For list nodes, key predicates SHOULD be omitted in schema paths but MAY be included when a specific instance is referenced in an operational context outside the scope of this data format.

Tools that generate YANG from a NAIM Document MAY derive YANG prefix shorthand and corresponding import statements from the module-name prefixes used in path fields, without requiring explicit import declarations in the NAIM Document.

5. Markdown View

5.1. Purpose and Normative Status

The Markdown view is a human-readable rendering of a NAIM Document, intended for human review, version-controlled document exchange, and editorial discussion. The Markdown view is a derived artifact; it is not independently normative.

Implementations that produce a Markdown view MUST derive it deterministically from the canonical JSON. Implementations that accept a Markdown view as input MAY use it to construct or update a canonical JSON document, subject to the constraints in Section 5.2.

5.2. Source of Truth Principle

The canonical JSON representation is the sole authoritative source of NAIM Document content.

When a Markdown view is used as input to reconstruct or update a canonical JSON document, implementations MUST NOT silently discard existing canonical JSON content. Specifically:

* Fields present in the canonical JSON but absent from the Markdown view MUST be preserved in the resulting canonical JSON unless the Markdown view contains an explicit deletion signal defined by the implementation.

* If the Markdown view and the canonical JSON express conflicting values for the same field, the implementation MUST resolve the conflict explicitly. The RECOMMENDED resolution strategy is to treat the Markdown view as expressing only the fields it explicitly represents, leaving all other canonical JSON fields unchanged.

5.3. Header Rules

A Markdown view MUST begin with a document header. The header rules are:

* For a main module document, the first line MUST be: "# module: {name}"

* For a submodule document, the first line MUST be "# submodule: {name}" and the second line MUST be "## belongs-to: {parent-module-name}"

* The module or submodule description SHOULD immediately follow as: "## description: {description}"

5.4. Section Blocks

For each top-level array that is present in the canonical JSON, the Markdown view SHOULD include a corresponding section.
Section headings SHOULD be: "## Revisions", "## Features", "## Typedefs", "## Groupings", "## Identities", "## Extensions". Each section SHOULD present its entries in list form, with enough detail to allow a reader to understand the content without reference to the canonical JSON.

5.5. Tree Diagram

The Markdown view SHOULD include a tree diagram following the section blocks and preceding the node definitions. The tree diagram is an informative visual aid only; it is not a normative representation of the schema.

The tree diagram SHOULD follow the YANG tree diagram conventions described in [RFC8340], using ASCII art to represent schema hierarchy. Because the tree diagram is informative and not used for round-trip reconstruction, implementations MAY omit it or vary its format.

5.6. Flat Node Definitions

Following the tree diagram, the Markdown view MUST include a flat definition block for each node in the nodes array. The flat definition format MUST follow these rules:

* Each node definition MUST begin with a heading of the form "### {local-name} ({node-type})" where {local-name} is the last path segment of the node's path field and {node-type} is the value of the node's node-type field.

* Node fields SHOULD be rendered in the following order when present: path, description, type, writable, key, operations, visibility, related-nodes, constraints, examples.

* Nodes MUST be presented in depth-first traversal order derived from their path field values.

5.7. Round-Trip Requirements

The combination of canonical JSON and the Markdown view MUST satisfy the following round-trip property: given a canonical JSON document J, a Markdown view M derived from J, and a reconstruction process R that produces JSON from M, the result R(M) MUST be semantically equivalent to J for all fields that the Markdown view represents.

Implementations MUST NOT produce Markdown views in which the rendering of any represented field loses information relative to the canonical JSON representation of that field.

6. Node Template Families

A node object in the nodes array MUST conform to one of the node template families defined in this section. Each node object MUST contain a node-type field whose value identifies the applicable template family.

6.1. Common Semantic Fields

The following fields MAY appear in any node object regardless of its template family:

* path (string, REQUIRED for all node templates): The absolute schema path of this node, following the conventions in Section 4.6.

* description (string, REQUIRED for all node templates): A human-readable description of the node’s purpose, semantics, and operational meaning.

* visibility (object, OPTIONAL): Conditions under which this node is present or applicable. See Section 9.

* related-nodes (array, OPTIONAL): References to other nodes that are semantically related to this node. Each entry SHOULD contain a path (absolute schema path) and a relationship (human-readable description).

* constraints (string or object, OPTIONAL): Operational constraints applying to this node beyond what the type system expresses. It MAY be a natural-language string or an object with xpath and description fields.

* examples (array, OPTIONAL): Usage examples. Each entry SHOULD contain a scenario and an operation field.

* extensions (array, OPTIONAL): Extension annotations applied to this node. Each entry MUST conform to the extension-use object structure defined in Section 11.

6.2. leaf and leaf-list

A node object with node-type value "leaf" or "leaf-list" represents a YANG leaf or leaf-list node.

REQUIRED fields: node-type, path, description, type (an object describing the data type; see Section 8), and writable (a boolean; true indicates a configuration node, false indicates a state node).

OPTIONAL fields: all common semantic fields defined in Section 6.1 apply.

6.3. container and list

A node object with node-type value "container" or "list" represents a YANG container or list node.

REQUIRED fields: node-type, path, description, and writable (a boolean). A list node MUST also contain key: a string containing the space-separated list key leaf names.

OPTIONAL fields: uses (a string reference to a grouping defined in the groupings array), and operations (an object that SHOULD contain preconditions and side-effects strings). All common semantic fields defined in Section 6.1 apply.

6.4. choice

A node object with node-type value "choice" represents a YANG choice node containing mutually exclusive alternatives. A case is not an independent node-type value; cases are represented inline within the cases array of the enclosing choice node object and do not appear as separate entries in the nodes array.

REQUIRED fields: node-type, path, description, and cases (an array of case objects). Each case object MUST contain name and nodes (an array of absolute schema paths identifying the nodes belonging to this case). Each case object SHOULD also contain description.

OPTIONAL fields: mandatory (a boolean; if true, exactly one case MUST be selected). All common semantic fields defined in Section 6.1 apply.

6.5. rpc and action

A node object with node-type value "rpc" or "action" represents a YANG RPC or action node.

REQUIRED fields: node-type, path, and description.

OPTIONAL fields: input (a string giving the absolute schema path of the input container), output (a string giving the absolute schema path of the output container), preconditions, side-effects, and error-conditions (all strings). All common semantic fields defined in Section 6.1 apply.

6.6. notification

A node object with node-type value "notification" represents a YANG notification node.

REQUIRED fields: node-type, path, and description.

OPTIONAL fields: trigger (a string describing the condition that causes this notification to be emitted), data (an array of objects each containing name, type, and description), and side-effects. All common semantic fields defined in Section 6.1 apply.

6.7. Node Ordering

In YANG, the order of schema nodes in a module definition has no semantic meaning [RFC7950]. Accordingly, this document does not define a required ordering of node objects within the nodes array. Implementations MUST NOT assign semantic meaning to the position of a node object within the array.

For deterministic output, implementations SHOULD produce canonical JSON and Markdown views using a consistent, documented traversal strategy. Depth-first traversal ordered by path value is RECOMMENDED.

7. Top-Level Array Entry Schemas

7.1. typedef

A typedef entry in the typedefs array defines a reusable named type. Each typedef object MUST contain name, description, and base-type (a string identifying the built-in or derived type on which this typedef is based).

Each typedef object MAY contain: range, length, pattern, units, and default (all expressed as strings using the same syntax as the corresponding YANG statement).

7.2. grouping

A grouping entry in the groupings array defines a reusable structural fragment. Each grouping object MUST contain name, description, and contains (an array of absolute schema paths identifying the nodes that constitute this grouping).

7.3. identity

An identity entry in the identities array defines a YANG identity. Each identity object MUST contain name and description.

Each identity object MAY contain: base (a string identifying the base identity, which MAY use a module-prefixed form such as "module-name:identity-name"), status (which MUST be one of "current", "deprecated", or "obsolete" if present), and reference.

7.4. extension Definition

An extension entry in the extensions array defines an extension keyword. Each extension object MUST contain name and description.

Each extension object MAY contain: argument (a string identifying the argument name if the extension takes an argument), and reference (a string citing the specification that defines the extension).

8. Type Representation

All leaf and leaf-list nodes MUST carry a type field. The type field MUST be a JSON object containing at minimum a base field that identifies the type.

8.1. Simple Types and Constraints

For YANG built-in scalar types (string, boolean, uint8, uint16, uint32, uint64, int8, int16, int32, int64, decimal64, empty), the type object MUST contain base (a string identifying the built-in type name).

The type object MAY contain the following constraint fields expressed as strings: range (applicable to numeric types), length (applicable to string), pattern (a regular expression, which MAY be expressed as an array of strings), default, and units. Constraint fields SHOULD only be included when semantically applicable to the declared base type; in particular, range, length, and pattern MUST NOT be included for boolean and empty types.

8.2. Enumeration

For enumeration types, the type object MUST contain base set to "enumeration" and an enum array. Each entry in the enum array MAY be either a string (short form, in which case the implementation MAY assign integer values sequentially starting from zero) or an object containing name (REQUIRED) and value (integer, OPTIONAL). Implementations that require stable enumeration value mappings SHOULD use the object form with explicit value fields.

8.3. Bits

For bits types, the type object MUST contain base set to "bits" and a bits array. Each entry in the bits array SHOULD contain name and position fields. If position is absent, the implementation MAY assign positions sequentially starting from zero.

8.4. Union

For union types, the type object MUST contain base set to "union" and a types array of nested type objects, each conforming to this section.

8.5. Leafref

For leafref types, the type object MUST contain base set to "leafref" and a path field giving the absolute schema path of the referenced leaf, following the path conventions in Section 4.6. The type object MAY contain require-instance (a boolean corresponding to the YANG require-instance statement; defaults to true if absent).

8.6. Identityref

For identityref types, the type object MUST contain base set to "identityref" and an identity field identifying the base identity (which MAY use a module-prefixed form).

8.7. Typedef Reference

When a node’s type is a previously defined typedef rather than a built-in type, the type object MUST contain base set to the typedef name. When resolving the base field, an implementation MUST first check whether the value matches an entry in the typedefs array of the same NAIM Document. If a match is found, the type is treated as a typedef reference. If no match is found, the value is interpreted as a YANG built-in type name. Typedef names SHOULD NOT duplicate YANG built-in type names to avoid ambiguity.

9. Visibility Fields

The visibility field of a node object expresses the conditions under which the node is present or applicable. This field corresponds to the YANG if-feature and when constructs.

9.1. if-feature

The if-feature field within a visibility object declares that the node is conditional on one or more features. The if-feature value MAY be: a simple string naming a single feature, or a logical expression object using and, or, or not keys. The and and or values are arrays of feature names or nested logical expression objects; the not value is a single feature name or nested logical expression object.

9.2. when

The when field within a visibility object declares a conditional expression that must evaluate to true for the node to be present. The when value MAY be: a natural-language string describing the condition, or an object containing xpath (REQUIRED in this form) and an optional description.

When when is expressed as a natural-language string, implementations SHOULD attempt to derive an equivalent XPath expression during YANG generation. If a reliable XPath expression cannot be derived, the implementation MUST NOT silently discard the condition. The implementation SHOULD either report that the condition requires manual completion or retain the natural-language form in the generated YANG description statement with a clear annotation that XPath completion is pending.

10. Deviation Declarations

Deviation declarations describe the ways in which a specific device implementation departs from a base YANG module. They correspond to the YANG deviation statement. Deviation objects MAY appear in the deviations field of the module object defined in Section 4.3.

Each deviation object MUST contain: target (a string giving the absolute schema path of the node being deviated), type (a string which MUST be one of "not-supported", "add", "replace", or "delete"), and reason (a string providing a human-readable explanation of why this deviation exists).

Deviation objects of type "add", "replace", or "delete" MAY contain additional fields describing the specific property being added, replaced, or deleted, using the same field names as the corresponding node template fields.

11. Extension Use

Extension use within a node object allows a node to carry annotations defined by extension declarations in the extensions array or by external modules. When present, the extensions field of a node object MUST be an array. Each entry MUST contain:

* name: A string. The qualified extension keyword, typically in the form "prefix:keyword".

* source-module: A string. The name of the YANG module that defines the extension.

* meaning: A string. A human-readable description of why this extension is applied to this node and what it semantically conveys.

Each entry MAY contain value (a string giving the literal argument value passed to the extension). If any of the required fields cannot be determined, the implementation MUST omit the extension-use entry entirely rather than emit an incomplete representation. The implementation SHOULD inform the user or operator that the extension entry was omitted and that manual completion may be required.

12. Submodule Handling

A NAIM Document representing a YANG submodule MUST contain a submodule object at the top level instead of a module object, as specified in Section 4.4.

A submodule NAIM Document MAY contain typedefs, groupings, identities, extensions, features, and nodes arrays. Feature declarations in submodules are valid per [RFC7950] and are incorporated into the parent module’s feature namespace.

When a conforming implementation transforms a submodule NAIM Document into YANG, the result MUST include a belongs-to statement referencing the parent module name given in the belongs-to field of the submodule object, MUST NOT include a namespace statement (as the namespace is inherited from the parent module), and MUST NOT include a module-level prefix statement for the same reason.

13. Interoperability Expectations

Interoperability in this specification is defined at the representation level. Independent implementations conforming to this document SHOULD be able to exchange canonical NAIM Document JSON objects and interpret them consistently.

Specifically, conforming implementations:

* MUST accept and produce canonical JSON objects that conform to the top-level structure defined in Section 4.

* MUST interpret node-type values consistently with the node template families defined in Section 6.

* MUST interpret type field contents consistently with the type representation rules defined in Section 8.

* MUST preserve the path, description, writable, and type fields of node objects when exchanging NAIM Documents.

* SHOULD preserve all optional semantic fields when exchanging NAIM Documents, even if the receiving implementation does not use those fields internally.

* MUST NOT reject NAIM Documents solely because they contain fields that the implementation does not recognize. Unknown fields SHOULD be preserved and forwarded without modification.

* SHOULD produce Markdown views that conform to the rendering rules in Section 5 when a Markdown view is requested.

Implementations are NOT required to share identical AI interaction logic, reconstruction algorithms, confidence handling strategies, or YANG generation internals. Interoperability is defined at the data format boundary, not at the workflow boundary.

14. Security Considerations

NAIM Documents are semantic model artifacts. They do not themselves constitute executable code or protocol messages, but they may drive downstream processes — including YANG module generation and, in other systems, runtime network configuration — that have direct operational consequences.

Integrity of NAIM Documents. Because a NAIM Document may be used as input to deterministic tool pipelines that generate YANG modules or drive network configuration, unauthorized modification of a NAIM Document can introduce incorrect or malicious semantics into downstream artifacts. Implementations SHOULD protect NAIM Documents in storage and transit using integrity mechanisms appropriate to the deployment environment.

Confidentiality of semantic model assets. NAIM Documents may contain detailed descriptions of network topology, device capabilities, and operational constraints. This information may be sensitive. Implementations SHOULD apply confidentiality protections appropriate to the sensitivity of the modeled information.

Validation before downstream use. Implementations that consume NAIM Documents as input to YANG generation or other downstream processes MUST validate the structural conformance of the document before processing. Accepting malformed or adversarially crafted NAIM Documents without validation may produce incorrect YANG modules or other erroneous outputs.

AI-generated content review. When a NAIM Document is produced in whole or in part by an AI-assisted authoring process, the resulting document SHOULD be subject to human review before it is used to

generate or deploy production YANG modules. AI-generated semantic content may contain inaccuracies that are not apparent from structural validation alone.

Use in larger AI systems. If NAIM Documents are used as semantic artifacts in larger AIN-style systems [AIN-ARCH], integrity, origin authenticity, and version consistency become even more important, because a modified semantic artifact may change downstream routing, selection, review, or execution decisions in ways that are difficult to observe after the fact.

Audit logging. Implementations that participate in workflows where NAIM Documents drive operational changes SHOULD maintain audit logs sufficient to reconstruct the chain of actions from a NAIM Document to a resulting downstream state change.

15. IANA Considerations

This document has no IANA actions. Future companion documents defining a media type for the NAIM canonical JSON format MAY request registration of an appropriate media type with IANA at that time.

16. References

16.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

16.2. Informative References

[AIN-ARCH] Feng, C., "Agentic Intent Network (AIN) Architecture", Work in Progress, Internet-Draft, draft-feng-nmrg-ain-architecture-00, April 2026, <https://www.ietf.org/archive/id/draft-feng-nmrg-ain-architecture-00.txt>.

[RFC6241] Enns, R., "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011, <https://www.rfc-editor.org/rfc/rfc6241>.

[RFC6901] Bryan, P., "JavaScript Object Notation (JSON) Pointer", RFC 6901, April 2013, <https://www.rfc-editor.org/rfc/rfc6901>.

[RFC8040] Bierman, A., "RESTCONF Protocol", RFC 8040, January 2017, <https://www.rfc-editor.org/rfc/rfc8040>.

[RFC8340] Bjorklund, M., "YANG Tree Diagrams", BCP 215, RFC 8340, March 2018, <https://www.rfc-editor.org/rfc/rfc8340>.

[RFC8343] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 8343, March 2018, <https://www.rfc-editor.org/rfc/rfc8343>.

Appendix A. Canonical JSON Schema Summary

The following table summarizes the top-level structure of a canonical NAIM Document (Section 4).

Field	Type	Required	Description
version	string	Yes	Format version, MUST be "1.0"
module	object	One of	Main module metadata
submodule	object (one)	module/ submodule	Submodule metadata
revisions	array	No	Revision history entries
features	array	No	Feature declarations
typedefs	array	No	Reusable type definitions
groupings	array	No	Reusable structural fragments
identities	array	No	Identity declarations
extensions	array	No	Extension keyword definitions
nodes	array	No	Schema node objects (primary content)

Node objects in the nodes array carry a node-type field whose recognized values are:

node-type	Section	Description
leaf	6.2	Scalar configuration or state value
leaf-list	6.2	Ordered or unordered sequence of scalar values
container	6.3	Structural grouping node
list	6.3	Sequence of instances keyed by one or more leaves
choice	6.4	Mutually exclusive alternative branches
rpc	6.5	Remote procedure call
action	6.5	Node-bound action
notification	6.6	Event notification

Appendix B. Example Canonical JSON Document

The following is an example of a conforming canonical NAIM Document representing a simplified Ethernet interface module.

```
{
  "version": "1.0",
  "module": {
    "name": "eth-interface",
    "description": "Ethernet interface configuration and state model",
    "organization": "Example Networks"
  },
  "revisions": [
    {
      "revision": "2026-04-17",
      "description": "Initial version"
    }
  ],
  "features": [
    {
      "name": "high-speed",
      "description": "Indicates support for interfaces operating at 10 Gbps or above",
      "status": "current"
    }
  ],
  "typedefs": [
    {
      "name": "bandwidth-mbps",
      "description": "Interface bandwidth expressed in megabits per second",
      "base-type": "uint32",
      "range": "1..1000000",
      "units": "Mbps"
    }
  ],
  "nodes": [
    {
      "node-type": "container",
```

```
      "path": "/eth-interface:interfaces",
      "description": "Top-level container for all Ethernet interface configuration",
      "writable": true,
      "operations": {
        "preconditions": "None",
        "side-effects": "None"
      }
    },
    {
      "node-type": "list",
      "path": "/eth-interface:interfaces/interface",
      "description": "An individual Ethernet interface entry",
      "key": "name",
      "writable": true,
      "operations": {
        "preconditions": "The interface name must be unique across the system.",
        "side-effects": "Creating an interface allocates system hardware resources. Delet
ing an interface releases all associated configuration and counters."
      }
    },
    {
      "node-type": "leaf",
      "path": "/eth-interface:interfaces/interface/name",
      "description": "The unique name of the Ethernet interface, such as 'eth0' or 'Gigab
itEthernet0/1'.",
      "type": { "base": "string", "length": "1..64" },
      "writable": false
    },
    {
      "node-type": "leaf",
      "path": "/eth-interface:interfaces/interface/bandwidth",
      "description": "The configured bandwidth of the interface in Mbps.",
      "type": { "base": "bandwidth-mbps" },
      "writable": true,
      "examples": [
        { "scenario": "Configure a 1 Gbps interface", "operation": "1000" },
        { "scenario": "Configure a 10 Gbps interface", "operation": "10000" }
      ]
    },
    {
      "node-type": "leaf",
      "path": "/eth-interface:interfaces/interface/admin-status",
      "description": "The administrative state of the interface as set by the operator.",
      "type": {
        "base": "enumeration",
        "enum": [
          { "name": "up",   "value": 0 },
          { "name": "down", "value": 1 }
        ]
      },
```

```
      "writable": true,
      "constraints": "The admin-status MUST be set to 'down' before the interface can be
deleted.",
      "visibility": {
        "if-feature": "high-speed"
      }
    },
    {
      "node-type": "leaf",
      "path": "/eth-interface:interfaces/interface/oper-status",
      "description": "The current operational state of the interface as observed by the s
ystem.",
      "type": {
        "base": "enumeration",
        "enum": [
          { "name": "up",      "value": 0 },
          { "name": "down",    "value": 1 },
          { "name": "testing", "value": 2 }
        ]
      },
      "writable": false
    }
  ]
}
```

Appendix C. Example Derived Markdown View

The following is the Markdown view derived from the canonical JSON document in Appendix B.

```
# module: eth-interface
## description: Ethernet interface configuration and state model

## Revisions
- 2026-04-17: Initial version

## Features
- high-speed: Indicates support for interfaces operating at 10 Gbps or above

## Typedefs
- bandwidth-mbps (uint32, range: 1..1000000, units: Mbps):
  Interface bandwidth expressed in megabits per second

## Tree
module: eth-interface
  +--rw interfaces
     +--rw interface* [name]
        +--rw name         string
        +--rw bandwidth    bandwidth-mbps
        +--rw admin-status enumeration {high-speed}?
```

```
        +--ro oper-status  enumeration
```

```
### interfaces (container)
- path: /eth-interface:interfaces
- description: Top-level container for all Ethernet interface configuration
- writable: yes
- operations:
  - preconditions: None
  - side-effects: None

### interface (list)
- path: /eth-interface:interfaces/interface
- description: An individual Ethernet interface entry
- key: name
- writable: yes
- operations:
  - preconditions: The interface name must be unique across the system.
  - side-effects: Creating an interface allocates system hardware resources.
    Deleting an interface releases all associated configuration and counters.

### name (leaf)
- path: /eth-interface:interfaces/interface/name
- description: The unique name of the Ethernet interface, such as 'eth0'
  or 'GigabitEthernet0/1'.
- type: string (length: 1..64)
- writable: no

### bandwidth (leaf)
- path: /eth-interface:interfaces/interface/bandwidth
- description: The configured bandwidth of the interface in Mbps.
- type: bandwidth-mbps
- writable: yes
- examples:
  - Configure a 1 Gbps interface: 1000
  - Configure a 10 Gbps interface: 10000

### admin-status (leaf)
- path: /eth-interface:interfaces/interface/admin-status
- description: The administrative state of the interface as set by the operator.
- type: enumeration [up(0), down(1)]
- writable: yes
- if-feature: high-speed
- constraints: The admin-status MUST be set to 'down' before the interface
  can be deleted.

### oper-status (leaf)
- path: /eth-interface:interfaces/interface/oper-status
- description: The current operational state of the interface as observed
```

by the system.
- type: enumeration [up(0), down(1), testing(2)]
- writable: no

Appendix D. End-to-End Example with YANG Output

This appendix presents the YANG module that a conforming implementation would produce from the canonical JSON document in Appendix B.

```
module eth-interface {
  yang-version 1.1;
  namespace "urn:example:eth-interface";
  prefix "eth";

  organization "Example Networks";

  revision 2026-04-17 {
    description "Initial version";
  }

  feature high-speed {
    description
      "Indicates support for interfaces operating at 10 Gbps or above";
  }

  typedef bandwidth-mbps {
    type uint32 {
      range "1..1000000";
    }
    units "Mbps";
    description "Interface bandwidth expressed in megabits per second";
  }

  container interfaces {
    description
      "Top-level container for all Ethernet interface configuration";

    list interface {
      key "name";
      description "An individual Ethernet interface entry";

      leaf name {
        type string {
          length "1..64";
        }
        config false;
        description
```

```
          "The unique name of the Ethernet interface,
           such as 'eth0' or 'GigabitEthernet0/1'.";
      }

      leaf bandwidth {
        type bandwidth-mbps;
        description "The configured bandwidth of the interface in Mbps.";
      }

      leaf admin-status {
        if-feature "high-speed";
        type enumeration {
          enum up   { value 0; }
          enum down { value 1; }
        }
        description
          "The administrative state of the interface as set by the operator.";
      }

      leaf oper-status {
        config false;
        type enumeration {
          enum up      { value 0; }
          enum down    { value 1; }
          enum testing { value 2; }
        }
        description
          "The current operational state of the interface
           as observed by the system.";
      }
    }
  }
}
```

Appendix E. Informative: Structured Validation Error Reporting

This appendix is informative. It describes a RECOMMENDED approach for implementations to report validation failures to higher layers in a form suitable for programmatic consumption. Implementations are not required to adopt this exact structure, but are RECOMMENDED to provide error information at a comparable level of detail.

When an implementation detects that a NAIM Document fails structural or semantic validation, it SHOULD produce a structured error object rather than a generic exception or unstructured error string. A suitable error object contains the following fields:

* error_type (string): A machine-readable error category. Examples include "missing_required_field", "type_constraint_violation", "invalid_path_format", and "duplicate_key".

* field_path (string): A JSON Pointer [RFC6901] expression identifying the location of the error within the NAIM Document. Example: "/nodes/2/type".

* message (string): A human-readable description of the error, expressed in terms that can be understood without reference to implementation internals.

* expected (string, OPTIONAL): A description of the expected value or format at the identified location.

Producing error information in this form enables higher layers — including AI-assisted workflow components — to identify and correct the specific location of errors without parsing unstructured text.


```
{
  "error_type": "missing_required_field",
  "field_path": "/nodes/2/type",
  "message": "The node at path '/eth-interface:interfaces/interface/bandwidth' has node-t
ype 'leaf' but is missing the required 'type' field.",
  "expected": "A type object with at least a 'base' field, for example: {\"base\": \"uint
32\"}"
}
```


Author's Address

Chong Feng
Ruijie Networks
Email: fengchonglly@gmail.com