Independent Submission
Internet-Draft
Intended status: Informational
Expires: 27 October 2025

D. Abaris
Individual Contributor

25 April 2025

JSON-Based Dynamic Configuration Management
draft-abaris-json-dcm-00

Abstract

JavaScript Object Notation (JSON) is a widely used data interchange format, suitable for storing configuration data due to its simple syntax, machine-readable structure, and ease of parsing and generation.

This document describes informational practices associated with JSON-Based Dynamic Configuration Management. It outlines a recommended naming convention for configuration files in the form of a structured filename pattern, such as "<project>@<version>.json", and specifies a configuration schema to support validation, traceability, and non-disruptive updates. The document also describes the rationale for standardization and presents real-world scenarios where these practices apply.

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 27 October 2025.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction

In this document, JSON-based information practices are described for managing dynamic configuration data in environments where versioning, portability, and machine validation are beneficial (e.g., distributed systems, Infrastructure as Code (IaC), DevOps). It is intended as a recommended approach for implementers seeking to adopt structured configuration formats suitable for dynamic or heterogeneous system contexts.

The document describes a JSON structure featuring top-level name, version, and config fields. It also provides a JSON Schema that implementers MAY adapt for domain-specific validation needs.

While a consistent identification pattern for configuration resources, such as <project>@<version>.json for files, is discussed as an aid to traceability and version management, its use is entirely OPTIONAL. The specific method for naming or identifying configuration resources MUST be determined based on operational constraints, deployment tooling capabilities, operating system limitations, and storage system characteristics.

The practices described in this document do not constitute an IETF standard. They may, however, serve as a useful reference or starting point for implementers developing configuration management solutions in research computing, DevOps, microservices or other automation-driven domains where dynamic updates and validation are primary concerns.

2. Scope Limitation

This document defines practices and a naming convention for JSON-based configuration files. It does not address how configuration files are distributed, secured, or integrated into specific technical domains.

Operational concerns such as secret management, transport mechanisms, or runtime application of configuration are out of scope.

3. Conventions and Definitions

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.

4. Recommended Practices

This section describes practices that implementers may find useful when managing dynamic configurations according to the approach presented in this document.

4.1. File Naming Convention

This section describes an optional filename format for versioned configuration files.

Implementers MAY use the following filename pattern:

<project>@<version>.json

For example, a configuration file for "webapp" at version 1.2.0 would be:

webapp@1.2.0.json

This pattern supports traceability, version comparison, and rollback tooling. Alternative formats MAY be chosen based on system requirements or environmental constraints, for instance, a more explicit variant like <project>@<version>.config.json MAY be used where clarity is paramount.

Implementers SHOULD validate chosen filenames against portable filename constraints (e.g., character repertoire, length limits) as defined in POSIX [POSIX] and consider file URI syntax rules in RFC 8089 [RFC8089].

In particular, filenames MUST NOT include control characters, MUST NOT include OS-reserved code points and sequences (for example < > : " / \ | ? *), MUST NOT use reserved device names (e.g., CON, PRN, AUX), and SHOULD respect platform path-length limitations.

Some filesystems or constrained environments (e.g., embedded FAT [MS-FAT] implementations or legacy systems enforcing 8.3 filename formats [MS-FSCC]) may not allow special characters like @ in filenames. In such cases, implementers SHOULD consider using -, _, or alphanumeric-only alternatives.

4.2. Key-Value Pair Descriptions

This section describes each top-level key defined in the JSON-based project information, explaining its purpose, expected data type, and usage context. For each key, a concise description is provided along with an example value to illustrate the intended format.

Key	Type	Description	Example
name	string	Identifier for the target system	"my-web-app"
version	string	Version of the configuration object	"1.0.0"
config	object	Domain-specific settings object	{"logLevel":"INFO",...}

Table 1

4.3. JSON Schema

This section describes a JSON Schema that implementers MAY use to validate configuration documents that conform to structure described in this document. The schema conforms to the JSON Schema 2020-12 specification [JSON-SCHEMA-2020-12] and structured mechanism for representing versioned configuration metadata across a variety of environments.

```
{
  "$id": "https://example.com/schemas/config-practice-v1.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Dynamic Configuration Structure Schema",
  "description": "Structure for versioned JSON config.",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "Project or app name. Unique within context.",
      "examples": [
        "my-web-app",
        "data-processing-pipeline",
        "network-device-base"
      ]
    },
    "version": {
      "type": "string",
      "description": "Version ID for the config content.",
      "examples": [
        "1.0.0",
        "2.1.0-beta.1",
        "v3.1.4"
      ]
    },
    "config": {
      "type": "object",
      "description": "Implementation-specific configuration.",
      "additionalProperties": true,
       "examples": [
        {
          "logLevel": "INFO",
          "port": 8080,
          "featureFlags": {
            "newDashboard": true,
            "enableApiRateLimit": false
           },
          "allowedOrigins": ["https://app.example.com"]
        }
      ]
    }
  },
 "required": ["name", "version", "config"],
 "additionalProperties": false
}
```

The schema defines the following required fields:

* name - identifies the system, component or application the configuration is intended for.

* version - specifies the version identifier using a string value, typically following Semantic Versioning [SEMVER].

* config - represents the domain-specific configuration object. Its internal structure is implementation-defined and MAY be constrained using additional schemas.

This JSON Schema MAY be dereferenced and reused across configuration tooling, systems or services. It is RECOMMENDED that implementers assign a canonical, dereferenceable $id to the schema.

This schema is interoperable with existing JSON Schema tooling and validation engines. When implementing similar schema-backed validation in credential-based or identity systems, implementers MAY refer to practices outlined in the [VC-JSON-SCHEMA].

4.4. Example Configuration

The following configuration example demonstrates the usage of this specification in a scientific context. It draws inspiration from the configuration patterns used to control High-Level Trigger (HLT) settings in the ATLAS experiment at CERN.

The file is named using the prescribed convention: atlas@3.1.0.json - where atlas is the project identifier and 3.1.0 denotes the configuration version.

```
{
  "name": "atlas-hlt-config",
  "version": "3.1.0",
  "config": {
    "project": "ATLAS",
    "runType": "collisions",
    "hltMenu": "Physics_pp_run3_v1",
    "chains": [
      "HLT_e26_lhtight_ivarloose",
      "HLT_mu20_iloose_L1MU15"
    ],
    "logging": "INFO"
  }
}
```

This configuration specifies the data-taking mode, selected trigger menu, and active HLT chains for a proton-proton collision run. It illustrates a concise, versioned configuration adhering to the schema described in this document.

5. Implementation Considerations

The implementation of this specification is informed by key principles recognized in modern configuration management practices. The naming convention - "<project>@<version>.json" - promotes version traceability and supports non-disruptive configuration updates. This aligns with Infrastructure as Code (IaC) practices, where configuration artifacts are declaratively defined, versioned, and integrated into automated provisioning workflows. This approach is in line with versioned configuration practices as described in the live configuration management systems deployed at Facebook, as detailed in [FACEBOOK-CM].

Various approaches to dynamic configuration management are employed in practice. The OpenCAPIF represents one such approach, where configuration data is maintained in a single top-level container (e.g., "capif_configuration") hosted within a MongoDB collection [CAPIF-DYNAMIC-CONFIG].

This document defines only a file naming convention, and a minimal JSON structure. It does not specify how configuration files are loaded, validated, or applied within systems.

Implementers are expected to name their configuration files according to the defined pattern and structure their contents using the prescribed schema.

Schema evolution SHOULD be handled in a backward-compatible manner whenever possible. Additive changes such as introducing new optional keys are preferred over breaking changes like renaming or removing fields, or altering value types.

When updating configuration schemas, the type of change should influence how you version it [SEMVER]:

Change Type	Description	Version Impact
Add optional fields	Doesn’t break existing usage	Minor (e.g. 1.0.0 to 1.1.0)
Remove or rename fields Change data types	Breaks compatibility with older configs	Major (e.g. 1.0.0 to 2.0.0)

Table 2

When non-compatible updates are introduced, a new version identifier MUST be assigned and reflected in both the version field and the file name.

Due to the line-oriented structure of JSON, implementers using version control systems (e.g., Git) MAY experience merge conflicts, especially when deeply nested structures or unordered keys are involved. To mitigate this, implementers MAY adopt canonical formatting, deterministic key ordering, or leverage structured merge strategies such as JSON Merge Patch [RFC7396] where applicable.

6. Use-Case Examples

The practices described in this document are most applicable to systems where configuration needs to be dynamically loaded, versioned, potentially exchanged between systems, and programmatically validated. The explicit versioning and schema support address challenges common in complex, automated, or distributed environments.

The following examples illustrate environments that benefit from such capabilities, categorized by the domain and corresponding technology stack.

6.1. Use Case of Scientific Computing

As detailed in the ATLAS example, large-scale experiments require reproducible, traceable, and dynamically adjustable configurations. Complex systems like the High-Level Trigger (HLT) rely on structured, versioned configurations (often using JSON) to manage components like trigger menus (L1Menu) or beam conditions (BunchGroupSet) [ATLAS-TRIGGER-2024]. The ability to validate these configurations against a schema before deployment and associate specific versions with data-taking periods is crucial for operational stability and analysis reproducibility. This specification provides a consistent structure for such versioned components.

During nightly integration tests, ATLAS produces full configuration sets in JSON format that are used to initialize and operate the trigger system [ATLAS-TRIGGER-2024]. These files are fixed for a given run, versioned, and stored in a way that enables test reproducibility, consistency across clusters, and auditability. They are used directly by HLT software during initialization. Some configurations may evolve during a run and are linked to specific run conditions in the ATLAS database.

A portion of this bunch group definition is shown below. Each group entry contains an identifier, a descriptive name, and a list of BCIDs (Bunch Crossing Identifiers) representing beam structure segments:

```
"bunchGroups": {
  "BGRP0": {
    "name": "BCRVeto",
    "id": 0,
    "info": "3543 bunches, 2 groups",
    "bcids": [
      { "first": 0, "length": 3540 },
      { "first": 3561, "length": 3 }
    ]
  },
  "BGRP1": {
    "name": "Paired",
    "id": 1,
    "info": "3452 bunches, 3 groups",
    "bcids": [
      { "first": 0, "length": 3445 },
      { "first": 3536, "length": 4 },
      { "first": 3561, "length": 3 }
    ]
  }
}
```

The operational practice in ATLAS often involves managing configuration components (like BunchGroupSet, L1Menu) as independently versioned JSON documents, linked via external systems (like databases) to specific run conditions or time intervals. This modular approach aligns well with the structured, schema-validatable format proposed here, allowing individual components to be managed consistently while supporting dynamic assembly based on runtime needs.

For example, during a multi-hour data-taking run, different configurations may be assigned based on real-time accelerator conditions. A run might begin with a "standby" BunchGroupSet and switch to a "physics" set as the beam fills, before reverting back.

This approach enables dynamic reconfiguration and fine-grained traceability. Similar principles apply to prescale and monitoring files.

While bundling all related configuration fragments into a single large JSON object per run is theoretically possible, operational practice in ATLAS favors keeping these files decoupled to support reusability, clarity and independent versioning. This design aligns well with the layered, schema-validatable configuration model proposed in this specification.

6.2. Use Case of Network Protocol Configuration

Secure group communication protocols, such as the IETF's Messaging Layer Security (MLS) [RFC9420], require careful configuration of cryptographic parameters and operational policies to ensure security and interoperability. These parameters include selections of cipher suites, credential validation rules, group size limits, and integration details with underlying delivery services.

Managing these configurations across potentially large, dynamic, or federated deployments benefits significantly from a versioned, structured, and schema-validatable format like the one described in this document. For instance, different client implementations or service providers might need to load specific, validated policy versions to ensure compatible behavior or adhere to security updates.

The following example illustrates how a configuration file adhering to the recommended structure described in this document could be used to manage settings for an MLS client or component. If using an illustrative naming pattern like mls-client-policy@1.0.0.json, the content might be:

```
{
  "name": "mls-client-policy",
  "version": "1.0.0",
  "config": {
    "protocolSuite": "MLS",
    "allowedCipherSuites": [
      "MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519",
      "MLS_128_DHKEMP256_AES128GCM_SHA256_P256"
    ],
    "credentialOptions": {
      "requireCertificate": true,
      "trustedRoots": [
        "cert-root-a.pem",
        "cert-root-b.pem"
      ]
    },
    "groupLimits": {
      "maxMembers": 1024,
      "maxLifetimeHours": 720
    },
    "deliveryServiceUrl": "https://mls.provider.com/v1/messages"
  }
}
```

In this example, the config object contains specific settings relevant to MLS operation: selecting allowed cryptographic suites, defining credential requirements, setting group size and lifetime policies, and specifying the endpoint for the message delivery service. This demonstrates how the flexible config object can encapsulate protocol-specific parameters within the versioned structure.

6.3. Use Case of Infrastructure Provisioning

Infrastructure as Code (IaC) practices, using tools like Terraform, Pulumi, or Ansible, rely on structured, declarative configuration files to describe and manage infrastructure components. Versioned and schema-validatable JSON configurations reduce the likelihood of misconfiguration, enable reproducibility, and support integration with automated provisioning workflows.

The following example illustrates how a configuration file adhering to the recommended structure described in this document could be used to configure network infrastructure components in a declarative and automation-friendly manner. In environments where routers, firewalls, or service edge devices expose RESTCONF APIs based on YANG models, vendors like Cisco support JSON-encoded configurations that can be applied using standardized interfaces aligned with [RFC8040]

and practical guidelines such as those provided by [CISCO-RESTCONF-API]. If using an illustrative naming pattern like dhcp-subnet@1.0.0.json, the content might be:

```
{
  "name": "dhcp-subnet",
  "version": "1.0.0",
  "config": {
    "dhcp:subnet": {
      "net": "10.254.239.0/27",
      "range": {
        "dynamic-bootp": {},
        "low": "10.254.239.10",
        "high": "10.254.239.20"
      },
      "dhcp-options": {
        "router": [
          "rtr-239-0-1.example.org",
          "rtr-239-0-2.example.org"
        ]
      },
      "max-lease-time": 1200
    }
  }
}
```

6.4. Use Case of Cloud Infrastructure

Microservice environments demand frequent, isolated service updates. A dynamic and portable configuration system allows each service to load the appropriate configuration version at runtime, enhancing flexibility and reducing downtime.

In Kubernetes deployments, services often require runtime configuration that adapts based on environment, load or feature flags. Using versioned JSON files served via config maps or mounted volumes, these services can boot with valid, schema-verified data and switch configuration at runtime with zero downtime.

6.5. Use Case of CI/CD and DevOps

Declarative, machine-validated configuration enables safe and auditable deployments across environments. Systems like GitHub Actions, Jenkins, or ArgoCD benefit from a consistent config format for managing build, test and deployment workflows.

A JSON schema defines permissible stages, secrets and artifacts within a pipeline. Users can validate workflow definitions before committing them to version control. Runtime systems load validated configurations to deploy applications across dev, staging and prod environments safely.

6.6. Use Case of Data Orchestration

Workflow engines like Nextflow and Apache Airflow use configuration files to define pipeline behavior. A typed, versioned JSON configuration ensures pipeline reproducibility and allows validation before runtime execution, reducing execution failures.

Scientists author pipelines once and store configurations separately. The same pipeline can be executed across cloud environments or local clusters by switching configuration inputs. This abstraction improves portability and simplifies reproducibility of data analyses.

7. Security Considerations

This document defines a specification naming convention and format for configuration files; there are no related security considerations.

8. IANA Considerations

This document has no IANA actions.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

9.2. Informative References

[ATLAS-TRIGGER-2024] "The ATLAS Trigger System for LHC Run 3 and Trigger Performance in 2022", n.d., <https://iopscience.iop.org/article/10.1088/1748-0221/19/06/P06029>.

[CAPIF-DYNAMIC-CONFIG]
"Dynamic Configuration - ETSI SDG OCF Documentation", n.d., <https://ocf.etsi.org/documentation/develop/configuration/configuration/>.

[CISCO-RESTCONF-API]
"The RESTCONF API - Network Services Orchestrator (NSO) v6.3 - Cisco DevNet", n.d., <https://developer.cisco.com/docs/nso-guides-6.3/the-restconf-api/#getting-started>.

[FACEBOOK-CM]
"Holistic Configuration Management at Facebook", n.d., <https://research.facebook.com/publications/holistic-configuration-management-at-facebook/>.

[JSON-SCHEMA-2020-12]
JSON Schema Community Group, "JSON Schema: A Media Type for Describing JSON Documents", 2020, <https://json-schema.org/draft/2020-12/schema>.

[MS-FAT] Microsoft, "Naming Files, Paths, and Namespaces", 28 August 2024, <https://learn.microsoft.com/en-us/windows/win32/fileio/naming-a-file>.

[MS-FSCC] Microsoft, "File System Control Codes (FSCC)", 7 April 2025, <https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/18e63b13-ba43-4f5f-a5b7-11e871b71f14>.

[POSIX] The Open Group, "Information Technology--Portable Operating System Interface (POSIX) Base Specifications, Issue 7", 2018, <https://pubs.opengroup.org/onlinepubs/9699919799/>.

[RFC7396] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7396, DOI 10.17487/RFC7396, October 2014, <https://www.rfc-editor.org/rfc/rfc7396>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <https://www.rfc-editor.org/rfc/rfc8040>.

[RFC8089] Kerwin, M., "The "file" URI Scheme", RFC 8089, DOI 10.17487/RFC8089, February 2017, <https://www.rfc-editor.org/rfc/rfc8089>.

[RFC9420] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", RFC 9420, DOI 10.17487/RFC9420, July 2023, <https://www.rfc-editor.org/rfc/rfc9420>.

[SEMVER] Preston-Werner, T., "Semantic Versioning 2.0.0", 2013, <https://semver.org/>.

[VC-JSON-SCHEMA] "Verifiable Credentials JSON Schema Specification", n.d., <https://www.w3.org/TR/vc-json-schema/>.

Acknowledgments

The author thanks Tim Martin and Zach Marshall from CERN for their detailed explanations and guidance on the ATLAS High-Level Trigger system’s configuration architecture, which informed the structure and practical alignment of this specification.

The author also thanks Ege Korkan from Siemens AG, and Daron Yondem from Microsoft for detailed review comments and critical feedback on early staging of this document.

Author’s Address

Dogu Abaris
Individual Contributor
Email: abaris@null.net