

OpenAV Technical Guidelines - Cloud API Documentation Requirements

Document Version: 1.0

Prepared by: OpenAVCloud Technical Working Group

Date: September 8th, 2026

Status: 1.1 · Draft Update (Working Group Proposal) **Applies to:** Any manufacturer offering a cloud API for integration with OpenAV. **Worked example:** A complete reference that satisfies every requirement below lives in [sample-documentation/](#).

 Scope: documentation only — not implementation

This document defines **HOW TO DOCUMENT** a cloud API. It does **NOT** define how to build, design, or operate one. Whether your cloud already exists or is newly built, every requirement here is about **describing** it clearly and completely — never about changing how it works. Wherever a standard is named (OAuth 2.0, RFC 9457, ...), it is a **documentation-format** requirement or a **recommendation**, never a mandate to re-implement your system.

Table of Contents

1. [Introduction & Scope](#)
2. [Foundational Requirement: A Machine-Readable Contract](#)
3. [General API Information](#)
4. [Authentication](#)
5. [Authorization](#)
6. [Endpoints & Operations](#)
7. [Data Models & Schemas](#)
8. [Error Handling](#)
9. [Rate Limiting & Other Limits](#)
10. [Versioning & Lifecycle](#)
11. [Pagination, Filtering & Sorting](#)
12. [Asynchronous & Event-Driven Interfaces](#)
13. [Security Documentation](#)
14. [Operational Behavior](#)
15. [Examples, SDKs & Tooling](#)
16. [Documentation Quality & Publication](#)
17. [Submission & Conformance to OpenAV](#)
18. [Appendices](#)

Document History and Change Log

This section captures the history of changes made to this document.

Version	Date	Reason for Change
1.0	2026-09-08	Draft

1. Introduction & Scope

These guidelines define the mandatory and recommended contents of a cloud API's documentation so that third-party developers — and OpenAV — can integrate against it without guesswork or vendor support tickets.

- **Purpose.** Establish a consistent, complete documentation baseline across all manufacturers.
- **Audience.** Manufacturers with an existing or new cloud API, and the developers who integrate with it.
- **In scope:** how the API is documented. **Out of scope:** how the API is designed, built, or operated.
- **Conformance.** An API is OpenAV-compatible when its documentation satisfies **every MUST** in this guideline and passes OpenAV validation (Section 17). The full list of MUST items is in [Appendix C](#).
- **Sections are mandatory; some answers may be "not yet."** Where a required detail genuinely does not exist yet or does not apply, the section is still kept and the gap stated explicitly — see [Appendix C](#), *Declaration of missing or pending information*. A documented gap is conformant; a missing section is not.

Normative language

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHOULD**, **SHOULD NOT**, **MAY**, and **OPTIONAL** are to be interpreted as described in **RFC 2119** and **RFC 8174**.

Keyword	Meaning
MUST / REQUIRED	Absolute requirement. Non-conformance fails validation.
SHOULD / RECOMMENDED	Strongly advised; omit only with good reason.
MAY / OPTIONAL	Genuinely optional.

2. Foundational Requirement: A Machine-Readable Contract

All documentation is anchored to a machine-readable contract. For RESTful APIs that contract is an [OpenAPI](#) document; other architectures use the equivalent standard for their protocol (§2.1).

2.1 Machine-Readable Specifications

Every compliant API **MUST** provide a machine-readable specification corresponding to its underlying architecture:

Architecture	Required specification format
RESTful APIs	OpenAPI 3.1.x document (<code>openapi.yaml</code> / <code>openapi.json</code>)
Event-driven & WebSocket APIs	AsyncAPI specification
GraphQL APIs	Complete, valid GraphQL Schema Definition Language (SDL) document
gRPC / Protobuf services	Valid <code>.proto</code> definitions for every exposed service and message

An API that exposes more than one architecture (e.g., a REST control plane plus a WebSocket telemetry stream) **MUST** provide the corresponding specification for **each** surface.

Requirements

- A machine-readable spec **MUST** be the single source of truth.
- Human-readable documentation **SHOULD** be generated from the spec (e.g., Redoc, Swagger UI, Stoplight, AsyncAPI Generator) so prose and contract never drift.
- The spec **MUST** validate cleanly, with zero errors, against the schema for its format (OpenAPI 3.1 schema, AsyncAPI schema, GraphQL SDL parse, `protoc` compile).
- The spec **MUST** describe the **entire public API surface**. Undocumented ("hidden") endpoints, channels, or operations are not permitted.
 - "Surface" is judged **per specification format**, not by HTTP path count. A GraphQL API that exposes a single transport endpoint (e.g., `GET/POST /graphql`) satisfies this requirement by documenting that endpoint plus a complete SDL schema — the schema, not the path list, is the surface. The same holds for a gRPC service endpoint or an MQTT broker: document the transport entry point, and the full operation set in the format's own specification.
- The spec **MUST** be version-controlled and each published revision **MUST** be retrievable.

Note on the examples in this document. Sections 3–16 illustrate requirements with OpenAPI 3.1 because REST is the most common case. Where a requirement names an OpenAPI construct, the equivalent construct in the API's own specification format satisfies it — e.g., AsyncAPI `channels/messages/operations`, GraphQL type and field definitions, or protobuf `service/message` declarations.

Minimum OpenAPI skeleton

openapi: 3.1.0

info:

title: <Product> Cloud API

version: 1.0.0

description: One-paragraph summary of what the API does.

servers:

- url: https://api.example.com/v1

description: Production

paths: {}

components: {}

2.2 Purpose-Driven Endpoint Documentation

Human-readable documentation **SHOULD** describe the **core business purpose and semantic context** of each endpoint or operation: what it is for, when a client should call it, and what it affects.

Field-level descriptions, data types, and constraint definitions **SHOULD** be embedded directly in the machine-readable specification rather than repeated in Markdown tables. Duplicating the contract in prose creates maintenance overhead and drift; the specification is the single source of truth (§2.1), and generated reference documentation renders it for human readers.

2.3 Supplemental Guides & Workflows

Vendors **MAY** provide supplemental, human-readable guides for complex operational sequences or multi-step integration workflows — for example OAuth authentication handshakes, device provisioning sequences, or streaming setup. These guides serve to

contextualize operations whose correct use is not immediately obvious from individual endpoint definitions, and are encouraged wherever a working integration requires calling several operations in a specific order.

A "getting started" walkthrough is a special case of this and is separately **RECOMMENDED** — see Section 15.

3. General API Information

The reader must immediately understand what the API is, where it lives, and how to reach it.

Requirements

- The `info` object **MUST** include `title`, a meaningful `description`, `version`, and `contact`. `license` and `termsOfService` **SHOULD** be present.
- `servers` **MUST** list every base URL, with environments clearly labeled (production, sandbox, etc.).
- Transport **MUST** be documented as **HTTPS only**; the minimum TLS version **SHOULD** be stated (Section 13).
- Default media type(s) (e.g., `application/json`), character encoding (UTF-8), and date/time format (RFC 3339) **MUST** be documented.
- Naming and casing conventions (e.g., `camelCase` JSON fields) **MUST** be documented and applied consistently.

Example

info:

title: OpenAV Display Cloud API

version: 1.0.0

description: Monitor and control OpenAV networked displays.

termsOfService: <https://developer.example.com/terms>

contact:

name: Developer Support

email: api@example.com

url: https://developer.example.com/support

license:

name: Proprietary

servers:

- url: https://api.example.com/v1

description: Production

- url: https://sandbox.api.example.com/v1

description: Sandbox (test data, no real devices)

4. Authentication

Explain exactly how a developer proves **who they are**. (*Document the scheme(s) already in use — do not change them.*)

Requirements

- Every authentication scheme the API uses **MUST** be declared under `components.securitySchemes` and applied via `security`.
- The credential-provisioning / onboarding flow (how a developer gets keys or a client) **MUST** be documented.
- Token acquisition, **lifetime**, **refresh**, and **rotation/revocation** behavior **MUST** be documented where applicable.
- Where credentials belong in a request (e.g., `Authorization: Bearer ...`) **MUST** be documented. Credentials **MUST NOT** be placed in URLs (Section 13).
- Sandbox/test credentials or flows **SHOULD** be documented.

Example — OAuth 2.0 client credentials

components:

securitySchemes:

oauth2:

type: oauth2

flows:

clientCredentials:

tokenUrl: https://auth.example.com/oauth2/token

scopes:

displays:read: Read display state and telemetry.

displays:control: Send commands and update displays.

security:

- oauth2: []

Document at minimum: how to obtain a client, the token endpoint, grant type(s), token TTL, how to refresh, and how to revoke.

5. Authorization

Explain what an **authenticated caller is allowed to do**.

Requirements

- The scopes / roles / permissions the API enforces **MUST** be documented and mapped to operations.
- Per-operation access requirements **MUST** be expressed via the OpenAPI **security** field.
- Multi-tenancy and resource-ownership rules (who can see/act on which resources) **MUST** be documented.
- The error returned on insufficient permission (typically **403**) **MUST** be documented (Section 8).

Example — scope table

Scope	Grants
displays:read	List/read displays and their telemetry.

Scope	Grants
<code>displays:control</code>	Send commands and modify displays.

Example — per-operation security

paths:

`/displays/{displayId}/commands:`

post:

security:

- oauth2: [displays:control]

6. Endpoints & Operations

Every operation must be fully described from the caller's perspective.

Requirements

- Each operation **MUST** define: `path`, HTTP method, a unique `operationId`, a `summary`, a `description`, and `tags` for grouping.
- All parameters (`path/query/header`) **MUST** document name, location, type, whether required, and constraints **in the specification itself**; restating them in external prose is not required (§2.2).
- Request bodies **MUST** reference a schema and declare required fields and content type.
- **All** responses — success and error — **MUST** be documented with status code and schema (Section 8).
- At least one **request example** and one **response example** per operation **MUST** be provided.
- Every example **MUST** validate against the schema it is declared under. Examples are part of the contract: a non-validating example is a documentation defect, not a cosmetic issue (§16).
- Idempotency of each operation **SHOULD** be documented; if idempotency keys are supported, they **MUST** be documented (Section 14).

Example — documented operation

paths:

/displays/{displayId}/commands:

post:

operationId: sendDisplayCommand

tags: [Commands]

summary: Send a control command to a display

description: Queues a command (e.g., power on/off, set input) for the display.

security:

- oauth2: [displays:control]

parameters:

- name: Idempotency-Key

in: header

required: false

description: Client-generated key to make retries safe (Section 14).

schema: { type: string, format: uuid }

requestBody:

required: true

content:

application/json:

schema: { \$ref: '#/components/schemas/Command' }

examples:

powerOn:

value: { type: power, value: "on" }

responses:

'202': { \$ref: '#/components/responses/CommandAccepted' }

'400': { \$ref: '#/components/responses/BadRequest' }

'401': { \$ref: '#/components/responses/Unauthorized' }

'403': { \$ref: '#/components/responses/Forbidden' }

'404': { \$ref: '#/components/responses/NotFound' }

'409': { \$ref: '#/components/responses/Conflict' }

'422': { \$ref: '#/components/responses/Unprocessable' }

'429': { \$ref: '#/components/responses/TooManyRequests' }

'500': { \$ref: '#/components/responses/InternalServerError' }

7. Data Models & Schemas

Shapes of data must be explicit, reusable, and unambiguous.

Requirements

- Reusable models **MUST** be defined under `components/schemas` and referenced (`$ref`), not duplicated.
- Each field **MUST** document `type`, `format` (where relevant), whether it is `required`, and nullability.
- Enumerations **MUST** list every allowed value.
- **Units of measure** for physical/telemetry values **MUST** be documented (e.g., temperature in °C).
- Field descriptions **SHOULD** be present for every non-obvious field, written in the specification rather than in a parallel prose table (§2.2).

Example

components:

schemas:

Display:

type: object

required: [id, name, status, createdAt]

properties:

id: { type: string, format: uuid, description: Unique display ID. }

name: { type: string, description: Human-friendly name. }

status: { type: string, enum: [online, offline], description: Connectivity. }

model: { type: [string, "null"], description: Hardware model, if known. }

createdAt: { type: string, format: date-time, description: RFC 3339 timestamp. }

DisplayStatus:

type: object

required: [power, temperatureC]

properties:

power: { type: string, enum: [on, off, standby] }

input: { type: [string, "null"], enum: [hdmi1, hdmi2, usb-c, null] }

temperatureC: { type: number, description: Panel temperature in degrees Celsius. }

8. Error Handling

Errors are part of the contract and **MUST** be documented as thoroughly as successes.

Requirements

- Every **application-level** status code the API can return **MUST** be documented per operation — specifically **all 4xx client-error conditions** and **all 2xx success conditions**.

- Generic server-side infrastructure errors (5xx) generated by the web server, gateway, proxy, or load balancer fronting the API **need not** be enumerated exhaustively. Any 5xx the application itself emits with a documented, application-specific meaning **MUST** be documented.
- The error response body **MUST** be documented. A **single consistent format MUST** be used across the API; **RFC 9457 (Problem Details for HTTP APIs)** is **RECOMMENDED**.
- A **catalog** of the API's application-level error codes/messages **MUST** be provided (Appendix-style table).
- Each operation **MUST** document its possible failure modes and their causes.

Standard status codes to cover (document each application-level code your API can emit):

Code	Meaning	Typical cause
200 / 201 / 202 / 204	Success	Request succeeded (sync / created / accepted / no content).
400 Bad Request	Malformed request	Invalid JSON, missing/'wrong-type' parameter.
401 Unauthorized	Not authenticated	Missing/expired/invalid token.
403 Forbidden	Not authorized	Authenticated but scope/ownership denies access.
404 Not Found	No such resource	Unknown ID or path.
409 Conflict	State conflict	Duplicate, version conflict, incompatible state.
422 Unprocessable	Semantic validation failed	Well-formed but invalid values.
429 Too Many Requests	Rate limited	Limit/quota exceeded (Section 9).
500 / 503	Server error	Internal failure / temporarily unavailable. Document application-emitted 5xx; generic

Code	Meaning	Typical cause
		gateway/load-balancer 5xx need not be enumerated.

Example — RFC 9457 Problem Details

components:

schemas:

Problem:

type: object

description: RFC 9457 problem detail.

required: [type, title, status]

properties:

type: { type: string, format: uri, description: URI identifying the problem type. }

title: { type: string, description: Short, human-readable summary. }

status: { type: integer, description: HTTP status code. }

detail: { type: string, description: Human-readable explanation for this occurrence. }

instance: { type: string, description: URI/identifier of this specific occurrence. }

code: { type: string, description: Stable application error code (see catalog). }

{

"type": "https://developer.example.com/errors/display-offline",

"title": "Display is offline",

"status": 409,

"detail": "Display 6f1c... cannot accept commands while offline.",

```
"code": "DISPLAY_OFFLINE",  
  
"instance": "/v1/displays/6f1c.../commands"  
  
}
```

Error catalog format (document every application code):

code	HTTP	Meaning	Resolution
DISPLAY_OFFLINE	409	Command sent to an offline display.	Wait for <code>display.online</code> event, retry.
INVALID_INPUT_SOURCE	422	Requested input not supported by model.	Use a value from <code>DisplayStatus.input</code> .

9. Rate Limiting & Other Limits

Developers must be able to design for the API's limits **before** hitting them.

Requirements

- All rate limits and quotas **MUST** be documented (values, window, and scope — per token, per tenant, etc.).
- Payload size limits, pagination limits, and concurrency caps **MUST** be documented.
- How the API communicates limit state **MUST** be documented — response headers, 429 status, `Retry-After`, and/or response body.
- Whether limits/quotas can be increased, and **how to request an increase**, **MUST** be documented.

Recommended headers (document whichever your API emits):

Header	Meaning
<code>RateLimit-Limit</code>	Requests allowed in the current window.
<code>RateLimit-Remaining</code>	Requests remaining in the window.

Header	Meaning
RateLimit-Reset	Seconds until the window resets.
Retry-After	Seconds to wait after a 429.

Example — 429 response

HTTP/1.1 429 Too Many Requests

Retry-After: 30

RateLimit-Limit: 600

RateLimit-Remaining: 0

RateLimit-Reset: 30

Content-Type: application/problem+json

```
{ "type": ".../errors/rate-limited", "title": "Rate limit exceeded", "status": 429, "code":
"RATE_LIMITED" }
```

10. Versioning & Lifecycle

Developers must know how the API evolves and how they will be warned of change.

Requirements

- The **versioning strategy** and how versions are expressed (URL path, header, media type) **MUST** be documented.
- **Backward-compatibility guarantees** — what constitutes a breaking vs. non-breaking change — **MUST** be documented.
- The **deprecation & sunset policy** **MUST** be documented, including notice period and any **Deprecation** / **Sunset** headers used.
- A **changelog** **MUST** be maintained and published.
- **How customers receive change notifications** (mailing list, RSS/Atom feed, changelog page) **MUST** be documented.

Guidance

- Semantic versioning (**MAJOR.MINOR.PATCH**) **SHOULD** be used; breaking changes **SHOULD** increment **MAJOR** and appear under a new version path (e.g., **/v2**).
- Non-breaking additions (new endpoints, new optional fields) **SHOULD NOT** break existing clients.

Example — deprecation headers

Deprecation: Sat, 01 Nov 2025 00:00:00 GMT

Sunset: Sun, 01 Feb 2026 00:00:00 GMT

Link: <https://developer.example.com/changelog#v2>; rel="successor-version"

11. Pagination, Filtering & Sorting

Collection endpoints must explain how to page through and narrow results.

Requirements

- The pagination pattern (cursor / offset / page) **MUST** be documented, including request parameters and response metadata.
- Supported **filtering** and **sorting** parameters **MUST** be documented per collection endpoint, including allowed fields and syntax.
- **Default and maximum page sizes MUST** be documented.

Example — cursor pagination

components:

parameters:

PageCursor:

name: cursor

in: query

description: Opaque cursor from the previous page's `meta.nextCursor`.

schema: { type: string }

PageLimit:

name: limit

in: query

description: Items per page. Default 25, maximum 100.

schema: { type: integer, minimum: 1, maximum: 100, default: 25 }

schemas:

PageMeta:

type: object

properties:

nextCursor: { type: [string, "null"], description: Cursor for the next page, or null if last. }

total: { type: [integer, "null"], description: Total items, if known. }

12. Asynchronous & Event-Driven Interfaces

For device-centric AV clouds the asynchronous surface — status changes, telemetry, presence, command results — is often as important as the synchronous request/response surface, and is documented to the same standard as Sections 6–8.

This section is mandatory for any API that pushes data to clients or to other clouds, by any mechanism, including:

- outbound **webhooks** (HTTP callbacks),
- **WebSocket** connections and subscriptions,
- **server-sent events (SSE)**,
- **MQTT** or similar topic subscriptions,
- **GraphQL subscriptions**,
- **cloud-to-cloud** event delivery (EventBridge-style buses, pub/sub topics).

If the API exposes none of these, the section **MUST** still be retained in the submitted documentation and declared **N/A** — see [Appendix C](#), *Declaration of missing or pending information*.

Requirements

- Every event **MUST** be documented with its **trigger**, **payload schema**, and an **example**, in the machine-readable specification appropriate to its transport — AsyncAPI **channels/messages** for WebSocket, MQTT, SSE and pub/sub; the OpenAPI **webhooks** (or **callbacks**) object for HTTP callbacks; the **Subscription** type in the SDL for GraphQL.
- The **transport and connection lifecycle** **MUST** be documented: connection/handshake, how the stream is authenticated, keep-alive/heartbeat, reconnection behavior, and any backfill or replay available after a gap.
- Subscription/registration and unsubscription **MUST** be documented (how endpoints or subscriptions are established, and which events each receives).
- **Delivery semantics** **MUST** be documented: retry policy, timeouts, ordering guarantees, deduplication, and the delivery guarantee (at-least-once / at-most-once / exactly-once).
- **Payload authenticity verification** **MUST** be documented for pushed payloads (e.g., a signature header and how to validate it).

Example — webhook definition (OpenAPI)

webhooks:

display.status_changed:

post:

operationId: onDisplayStatusChanged

summary: Sent when a display's status or telemetry changes.

requestBody:

content:

application/json:

schema: { \$ref: '#/components/schemas/DisplayStatusChangedEvent' }

responses:

'200': { description: Acknowledged. Return 2xx within 5s or delivery is retried. }

Signature verification (document the exact scheme): e.g., header **OpenAV-Signature:** **t=<unix>, v1=<hex-hmac-sha256>** over the raw body, using the endpoint's signing secret.

13. Security Documentation

Document the security behavior a developer needs to integrate safely. (*Documenting behavior — not prescribing how to implement it.*)

Requirements

- Transport security **MUST** be documented: HTTPS-only and the minimum TLS version accepted.
 - It **MUST** be documented that credentials/secrets are passed in headers and **never** in URLs or query strings.
 - CORS behavior (whether browser-based cross-origin calls are supported, and from where) **MUST** be documented if the API is callable from browsers.
 - Handling of sensitive/PII data in requests and responses **SHOULD** be documented.
-

14. Operational Behavior

Document how the API behaves in normal and degraded operation.

Requirements

- Health/status endpoint(s), request **timeouts**, and expected **retry** behavior **SHOULD** be documented.
- **Idempotency keys** — if supported — **MUST** be documented: header name, scope, and retention window.
- Any published **SLA**, **public status page**, or **incident-reporting** process **SHOULD** be documented.

Guidance

- Clients **SHOULD** be told to retry [429/503](#) with exponential backoff and to treat idempotency keys as safe for retries.
-

15. Examples, SDKs & Tooling

Reduce time-to-first-successful-call with concrete, runnable material.

Requirements

- Copy-pasteable request/response examples **MUST** be provided for every operation (Section 6).
 - A "getting started" / quickstart **SHOULD** walk from credentials → first authenticated call.
 - Official SDKs, a Postman/Insomnia collection, or a "try-it"/sandbox **SHOULD** be linked where available.
-

16. Documentation Quality & Publication

The docs must be reachable, current, and machine-checkable.

Requirements

- Documentation **MUST** be publicly reachable, or reachable via a clearly documented access process.
 - Documentation **MUST** be kept in sync with the live API (regenerated from the spec on each change — Section 2).
 - The spec **SHOULD** pass an agreed linter with the OpenAV ruleset — **Spectral** for OpenAPI and AsyncAPI, **graphql-schema-linter** (or equivalent) for SDL, **buf lint** for protobuf.
 - Examples **SHOULD** be validated against their declared schemas automatically in CI, so example drift is caught before publication (§6).
 - A **last-updated date** and the **spec version MUST** be visible in the published docs.
-

17. Submission & Conformance to OpenAV

How a manufacturer proves compatibility.

Process

1. The manufacturer **MUST** submit a valid, machine-readable specification for **every** architecture the API exposes (§2.1) — **OpenAPI 3.1.x**, **AsyncAPI**, **GraphQL SDL**, and/or **.proto** — as a URL to the hosted spec(s) or the file(s) themselves.
2. OpenAV validates the submission against: the schema for each specification format, the **OpenAV linter ruleset**, and the **MUST checklist** ([Appendix C](#)) — including the declaration rule for information that is unavailable or pending.
3. Any gaps are returned to the manufacturer with the specific failing items.
4. Once **all MUST items pass**, the API is recorded as **OpenAV-compatible**.

Manufacturer responsibilities

- Keep the submitted spec URL live and current.
- Re-submit (or keep the hosted spec updated) on every breaking change and version bump.

18. Appendices

Appendix A — Normative & Informative References

- **RFC 2119 / RFC 8174** — requirement keywords.
- **RFC 3339** — date/time format.
- **RFC 9457** — Problem Details for HTTP APIs (recommended error format).
- **RFC 6749** — OAuth 2.0 (if used for authentication).
- **OpenAPI Specification 3.1** — <https://spec.openapis.org/oas/v3.1.0> · overview: https://en.wikipedia.org/wiki/OpenAPI_Specification
- **AsyncAPI Specification** — <https://www.asyncapi.com/docs/reference/specification/latest> (event-driven, WebSocket, MQTT and pub/sub APIs).
- **GraphQL Specification** (incl. Schema Definition Language) — <https://spec.graphql.org/>
- **Protocol Buffers / gRPC** — <https://protobuf.dev/> · <https://grpc.io/docs/>
- **IETF RateLimit** header fields and **Deprecation/Sunset** headers — as referenced in Sections 9–10.

Appendix B — Glossary

Term	Definition
OpenAPI document / spec	The machine-readable contract describing the API.
Operation	A single method+path (e.g., <code>POST /displays/{id}/commands</code>).
Scope	A named permission grant tied to a token.
Idempotency key	A client-supplied key making a retried request safe.
Sunset	The date after which a deprecated version stops working.

Appendix C — Conformance Checklist

Every **MUST** in this document, as a checkable list. Every section of this guideline is a mandatory component of the documentation **structure**; an API is OpenAV-compatible when each MUST item is either satisfied or explicitly declared under the rule below.

Declaration of missing or pending information

If a specific operational parameter, policy, or technical detail — explicit rate-limit thresholds, sunset timelines, WebSocket delivery semantics, and so on — is not available or not applicable at the time of submission, the documentation **MUST** retain the section and state explicitly that the information or capability is **unmapped, omitted, or pending definition**. A best-effort description is expected wherever one can be given; where it cannot, an explicit "not available at this time" is itself a conformant answer.

Silently omitting a mandatory section is a **non-compliant submission**. The intent is that an integrator never has to hunt for this information in other sources, or discover its absence by trial and error.

Status summary

Submissions **SHOULD** include this table, with a status flag for each area:

Section / requirement	Compliance requirement	Status flag options
Machine-readable spec — OpenAPI / AsyncAPI / GraphQL SDL / <code>.proto</code> (§2.1)	Mandatory	Provided / Non-Compliant
Application error catalog & status codes (§8)	Mandatory section	Documented / Declared Pending
Rate limiting & quota semantics (§9)	Mandatory section	Defined / Declared Pending
Versioning, deprecation, sunset & change notifications (§10)	Mandatory section	Defined / Declared Pending
Async / WebSocket / event delivery semantics (§12)	Mandatory section	Defined / Declared Pending / N/A

Section / requirement	Compliance requirement	Status flag options
Security & transport constraints — minimum TLS, CORS (§13)	Mandatory section	Defined / Declared Pending
Operational behavior — request timeouts, retry policy, idempotency (§14)	Mandatory section	Defined / Declared Pending

Foundation

Machine-readable spec provided for **every** architecture exposed — OpenAPI 3.1.x (REST), AsyncAPI (WebSocket / event streams), GraphQL SDL, and/or **.proto** (§2.1).
Machine-readable spec is the single source of truth.
Spec validates against its format's schema with zero errors.
Entire public surface documented (no hidden endpoints, channels, or operations).
Endpoint documentation describes business purpose; field-level detail lives in the spec rather than duplicated prose (§2.2).

General info

info has title, description, version, contact.
servers lists every base URL with labeled environments.
HTTPS-only documented.
Media type, encoding, date format, naming conventions documented.

Authentication

All auth schemes declared and applied.
Credential provisioning/onboarding documented.
Token lifetime/refresh/rotation documented (where applicable).
Credential placement documented; none in URLs.

Authorization

Scopes/roles documented and mapped to operations.
Per-operation **security** present.
Multi-tenancy/ownership rules documented.
Insufficient-permission error documented.

Operations & models

Every operation has path, method, **operationId**, summary, description, tags.
All parameters and request bodies documented with schemas/constraints.
All responses (success + error) documented with schema.
Request + response example per operation.
Every example validates against its declared schema.
Reusable schemas under **components**; fields document type/format/required/nullability;
enums complete; units documented.

Errors

Every application-level status code documented per operation — all **4xx** and all **2xx** (generic gateway/load-balancer **5xx** not required).
Consistent error body documented (RFC 9457 recommended).
Application error-code catalog provided.
Failure modes documented per operation.

Limits

Rate limits/quotas documented (value, window, scope).
Payload/pagination/concurrency limits documented.
Limit-communication mechanism documented.
Limit-increase request process documented.

Versioning

Versioning strategy documented.
Backward-compatibility guarantees documented.
Deprecation & sunset policy documented.
Changelog maintained and published.
Change-notification mechanism documented.

Pagination

Pagination pattern documented with params + metadata.
Filtering/sorting documented per collection.
Default and max page sizes documented.

Asynchronous & event-driven interfaces *(declare N/A if the API pushes no data)*

Every event documented with trigger, schema and example, in the specification for its transport.
Transport & connection lifecycle documented (handshake, stream auth, keep-alive, reconnect, replay).
Subscription/unsubscription documented.

Delivery semantics (retries / ordering / deduplication / guarantee) documented.
Payload authenticity verification documented.

Security & operations

TLS/HTTPS and min TLS version documented.
No secrets in URLs (documented).
CORS documented (if browser-callable).
Idempotency keys documented (if supported).

Publication

Docs publicly reachable (or documented access).
Docs kept in sync with the API.
Last-updated date and spec version visible.

Submission

Valid specification file(s) for every exposed architecture submitted to OpenAV and kept current (§2.1).

Appendix D — Starter `openapi.yaml` Skeleton

A minimal, valid starting point. A complete worked example is in [sample-documentation/openapi.yaml](#).

openapi: 3.1.0

info:

title: <Product> Cloud API

version: 1.0.0

description: One-paragraph summary.

contact: { name: Developer Support, email: api@example.com }

servers:

- url: https://api.example.com/v1

description: Production

security:

- oauth2: []

tags:

- name: <Resource>

paths:

/<resource>:

get:

operationId: list<Resource>

tags: [<Resource>]

summary: List <resource>

responses:

'200':

description: OK

content:

application/json:

schema: { \$ref: '#/components/schemas/<Resource>List' }

'401': { \$ref: '#/components/responses/Unauthorized' }

'429': { \$ref: '#/components/responses/TooManyRequests' }

'500': { \$ref: '#/components/responses/InternalServerError' }

components:

securitySchemes:

oauth2:

type: oauth2

flows:

clientCredentials:

tokenUrl: https://auth.example.com/oauth2/token

scopes: { "read": Read access. }

schemas:

Problem:

type: object

required: [type, title, status]

properties:

type: { type: string, format: uri }

title: { type: string }

status: { type: integer }

detail: { type: string }

code: { type: string }

responses:

Unauthorized:

description: Not authenticated.

content: { application/problem+json: { schema: { \$ref: '#/components/schemas/Problem' } } }

TooManyRequests:

description: Rate limited.

content: { application/problem+json: { schema: { \$ref: '#/components/schemas/Problem' } } }

InternalError:

description: Server error.

content: { application/problem+json: { schema: { \$ref: '#/components/schemas/Problem' } } }