Lumen's API Strategy: A Framework for Digital Growth

Table of Contents

Lumen's API Strategy: A Framework for Digital Growth

This document outlines the comprehensive API strategy for Lumen. It details the core business objectives driving this initiative and the strategic framework that will guide our technology, governance, and cultural transformation. The goal is to evolve Lumen's APIs from a collection of inconsistent interfaces into a true, scalable platform that drives new revenue and delivers a world-class partner experience

Strategic Business Goals (The "Why")

This API strategy is a direct response to critical business needs. The significance of this program is its ability to fundamentally change how Lumen grows and competes.

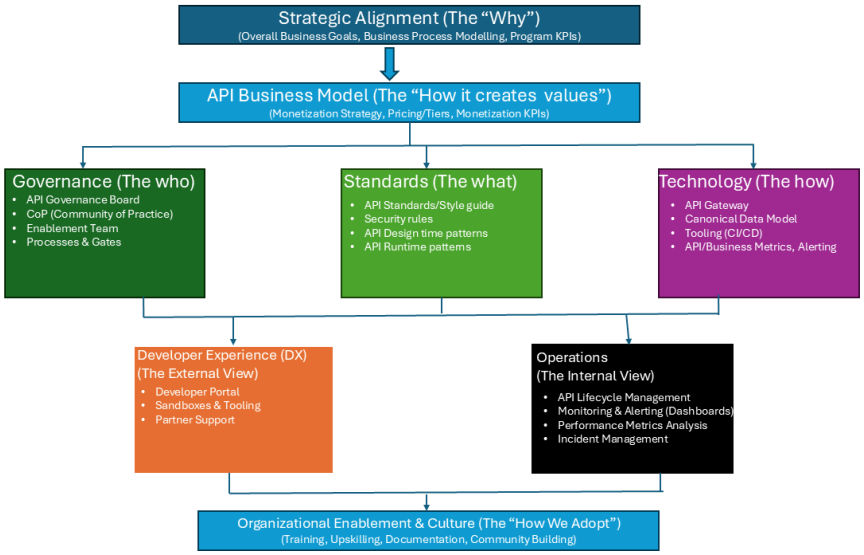
- **Unlock New Revenue and Growth** *This strategy opens new sales channels by letting us sell our services directly in cloud marketplaces and expand our reach globally through partners***.The primary goal is to create new, scalable sales channels. The strategy enables Lumen to sell its network services directly where customers are, such as in the** AWS and Azure marketplaces**. It also allows Lumen to dramatically expand its service footprint by using partners as** "offnet provider(s),"** enabling sales in markets where we don't own the physical network.
- **Increase Operational Agility and Speed to Market** *We're replacing slow, manual processes with full automation to deliver services in minutes, not months, giving us a critical speed advantage over our competitors*. To win against nimble competitors, speed is a critical advantage. This strategy is designed to replace slow, manual processes with a fully automated, **"digital quoting, ordering and activation"** experience. This allows for the **"rapid activation and management"** necessary to displace competitors like **PacketFabric and MP**.
- **Enhance the Partner Experience** *This initiative delivers a world-class API experience, establishing us as the partner of choice for industry leaders in the cloud and telecom ecosystems***. We are creating a**

professional, world-class API experience to make Lumen an easy and attractive partner for high-value companies like AWS and AT&T. For a business that relies on partners, the integration experience is the customer experience. This strategy directly addresses the current "Inconsistent API experience" **and** "Immature Developer Forward experience"**. By creating a consistent and professionally governed framework, Lumen becomes a more attractive and easier partner to work with for high-value companies like** AWS and AT&T** as well as our Data Center Operator partners.

- **Future-Proof the Technology Platform** *This initiative builds a single, reusable framework for all future integrations, allowing us to scale efficiently and stop building expensive, one-off solution**s*. **This initiative is a long-term investment in scalability. It creates a** "framework that can be consistently applied between the initiatives" **instead of building expensive, one-off solutions for each new partner. This solves the problem of** "Low code reusability"** and builds a platform that can adapt and grow as new partners and services are added.

Our Strategic Framework

To achieve these goals, we will implement a holistic strategy that aligns our business, governance, technology, and culture. The following diagram outlines the core components of our approach.



To achieve these goals, we will implement a holistic strategy that aligns our business, governance, technology, and culture. The following framework outlines the core components of our approach.

Explaining the Components

- **Strategic Alignment & Business Model:** This is the foundation. We start by ensuring every decision is aligned with Lumen's overall business goals and defining the specific business model for our APIs, including monetization strategies and the KPIs that measure financial success.

- **Execution (Governance, Standards, & Technology):** This is the core of our implementation, defining the people, processes, and tools required for success.

- **Governance:** This defines the "who" and "how" of our decision-making process. It's a phased model designed to evolve as our maturity grows:

    - **Phase 1 (Crawl/Walk):** Initially, a formal **API Governance Board** will provide centralized oversight. To accelerate progress, a hands-on **Core Working Team** will partner directly with our first product teams to build foundational assets like the canonical data models and URI standards.

During this phase, the **Community of Practice (CoP)** will act as a forum for learning and building our "cultural muscle" around these new standards.

- **Phase 2 (Run):** In the long term, the CoP will evolve into the primary **review 'board'** for new product APIs. This federated, peer-review model scales governance and ensures our standards remain practical and developer-friendly. The formal Governance Board will then transition to a strategic oversight role.
- **Standards:** These are the technical rulebooks for ensuring consistency and quality across all APIs. They will cover critical areas like API design style, security protocols, versioning, and error handling.
- **Technology:** This is the architectural blueprint. It is centered on a robust **API Gateway** that acts as an abstraction layer to hide internal complexity, and a **Canonical Data Model (CDM)** that provides a consistent, universal language for our business data.
- **The Result (Developer Experience & Operations):** The execution of our strategy produces two key outcomes: a world-class external **Developer Experience (DX)** with a portal and great support, and efficient internal **Operations** to ensure reliability through monitoring and lifecycle management.
- **Organizational Enablement & Culture:** This foundational layer supports the entire framework. It focuses on creating a true **"API-First Culture"** through a **"Robust training program"** to address the identified **"Skillset gap"** and proactive community building.

API Governance

Table of Contents

Mission and Vision

Mission: To establish, govern, and drive the adoption of a unified enterprise API strategy. This group will act as the engine for transforming Lumen's APIs from a collection of inconsistent interfaces into a cohesive, secure, and scalable platform that accelerates business growth.

Vision: To create a world-class API ecosystem that empowers our product teams, delights our partners and customers, and establishes Lumen as a leader in digital connectivity.

Scope and Key Responsibilities

This working group is the primary operational body responsible for defining and implementing the API strategy. Its scope includes:

- **Reference Architecture Ownership:**
- Define, document, and govern the official **"Sell" Side (SIL) Reference Architecture** for customer-facing APIs.
- Define, document, and govern the official **"Buy" Side (PIL) Reference Architecture** for partner integrations.
- **Standards and Canonical Models:**
- Finalize, approve, and manage the lifecycle of the enterprise **API Style Guide**.
- **Govern the Canonical Data Model (CDM):** Establish the enterprise-wide standards, technical format (e.g., JSON Schema), and versioning strategy for the CDM. This group provides the guardrails and the playbook.
- **Facilitate CDM Creation:** Act as expert consultants and a central review body to **assist Product Teams** in defining the canonical models for their specific business domains. While this group governs the overall standard, the **Product Teams own and are accountable for the specific CDM** for their respective domains (e.g., the Order team owns the Order model).

- **Governance Process & Tooling:**
- Define the end-to-end **API lifecycle**, including the governance "toll gates" and required artifacts for each stage.
- Create linter with rules from API Styleguide to enforce design time API governance during OAS creation
- Establish a process for automating governance checks within the CI/CD pipeline.
- Oversee the evaluation and implementation of all tooling required for API governance.
- **Tactical Execution and Enablement:**
- Act as the initial **API design review board** for new and in-flight projects.
- Perform **gap analysis** for critical projects (e.g., MCGW, LMCC) to create actionable alignment plans.
- Define and document **standard** integration**patterns** (e.g., Stateful vs. Stateless PILs, classes of partner integrations).

Membership

This is a cross-functional working group composed of key stakeholders from architecture, product, and engineering.

- **Driver:** Maninder Batth (API Architect)
- **Sponsor:** Bryan Dreyer (DJ Architecture and Solutions)
- **Core Members:**
- Anuj Tyagi (Platform API Product Management)
- Steve Edwards (Platform API Enablement)
- Santiago Cardin (API Enforcement/Standards)
- Syed Haider (API Integration Solution Architect - API Integration - Buy/OffNet SME)
- Boris Abramovich (Solution Architect Architect - LMCC/MCGW)
- William Muscato (Platform API Product Management)

- Jacob Johansen (IOD product manager)
- Heather P Muirbrook (Product Owner - Architecture and Solutions)
- **Consulting Members (as needed):**
- James Dwyer (Execution Team Architect)
- Prakash Agrawal (Enterprise Architect)

Ways of Working

To ensure a high tempo and full transparency, the group will adhere to the following operational model:

- **Work Tracking:** A dedicated **Kanban board** will be used as a master tracker for all initiatives, epics, and tasks. Members will link detailed work from their home team backlogs to this master board using a common Jira label (`API_Governance`).
- **Meeting Cadence:** The group will establish a rhythm that balances deep-dive strategic work with rapid issue resolution, likely consisting of:
- One 60-minute weekly working session.
- **Reporting:** The group will maintain a self-service status report (e.g., a Confluence page with a Jira dashboard) for the main API Governance Board to review, detailing progress, next steps, and risks.

Initial Priorities (First 30-60 Days)

The immediate focus of the working group will be to deliver tangible value by tackling the highest-priority foundational and tactical items.

1. **Finalize the Playbook:**
2. Formally approve the v1.0 **API Style Guide**.

3. Formally ratify the **"Sell" and "Buy" Reference Architectures**, including their deployment mandates.
4. Draft the v1.0 **API Lifecycle and Governance Process**.
5. **Facilitate the "Face of Lumen":**
6. Begin the process of working with the relevant product teams to define the **Canonical Data Model** for the most critical resources, using the MCGW/LMCC project as the primary input.
7. **Execute the First Tactical Engagements:**
8. Perform a formal **gap analysis** of the MCGW/LMCC APIs against the new standards and deliver a prioritized backlog of alignment tasks to the delivery team.
9. Initiate a **deep-dive analysis of the Netex APIs** to validate and document the "Stateless" PIL pattern.
10. **Success Metrics**

The success of this working group will be measured by its ability to drive tangible improvements in:

- **Consistency:** An increase in the percentage of new APIs that adhere to the approved standards and architectural patterns.
- **Agility:** A reduction in the time required to onboard new partners and launch new API products.
- **Reuse:** A measurable decrease in the number of bespoke, one-off solutions being

API Governance Process (Integrated with SDLC)

1. Overview

This document defines the standard lifecycle and governance process for designing, building, deploying, and deprecating APIs at Lumen. It ensures APIs are consistent, secure, aligned with strategic goals (per the Principles of Cloud-

First API Design), and adhere to the official Lumen API Style Guide. This process integrates governance checkpoints directly into the standard Software Development Lifecycle (SDLC).

2. Process Stages

Stage 1: Planning Phase (e.g., PI Planning)

- **Trigger:** Product Owner (PO) or Solution Architect identifies the need for a new API or significant changes (including deprecation) to an existing one.
- **Action:** Notify the API Governance Working Group (via designated intake form/meeting).
- **Required Info:**
- Clear **Business Purpose** & Use Case.
- High-level functional requirements.
- Anticipated consumers (internal/external/partner/wholesale).
- If deprecating: Reason, proposed timeline, and replacement strategy (if any).
- **Governance Review (Initial Checkpoint):**
- Strategic Alignment: Does this align with Cloud-First Principles?
- Domain Placement: Correct Business Function or Product Domain?
- Reuse Analysis: Does this capability exist? New API or extend existing?
- Scope & Pattern: High-level agreement on API Pattern (CRUD, Façade, Async).
- Deprecation Impact: Initial assessment of deprecation plan and user impact.
- **Benefits:** Prevents duplication, ensures strategic alignment early, guides teams to correct domain/pattern, anticipates governance workload.

Stage 2: Design Phase

- **Trigger:** Development team is ready to start API design.
- **Action (Repo Request):**
- **New API:** Request new Git repo from API Governance (provide domain, name, purpose, team). Governance provisions repo per standards.
- **Existing API:** Identify existing repo. Discuss versioning strategy (new major version `v(n+1)` for breaking changes, minor update to `v(n)` for non-breaking additions) with governance, based on the approved API Versioning Standard.
- **Action (OAS Development):**
- Development team designs the API in a feature branch, creating/updating `openapi.yaml`.
- **Mandatory:** Use the central Spectral linter ruleset locally during development. Only lint-error-free OAS files should be submitted.
- **Performance NFRs:** Define and document expected latency, TPS, and payload limits alongside the OAS.
- **Documentation:** Ensure all operations, parameters, schemas, and examples are clearly documented within the OAS (`summary`, `description`).
- **Governance Support:** Provide design consultation, clarification on standards, examples.

Stage 3: Review Phase (Pull Request)

- **Trigger:** Development team completes OAS design/update.
- **Action:** Submit a Pull Request (PR) against the main branch. Tag API Governance Working Group.
- **Automated Checks (CI/Jenkins):**
- PR **MUST** trigger automated checks.

- **Job Steps:** Run official Spectral linter ruleset; (Optional) Run breaking change detection.
- **Outcome:** Failed checks **MUST** block the PR. Dev team must fix and resubmit.

- **Security Review (Post-Automation, Pre-Governance):**

- **Trigger:** Automated checks pass.

- **Action:** The designated Security team reviews the OAS PR, focusing on authentication, authorization scopes, data sensitivity, potential vulnerabilities (input validation, etc.), and adherence to security standards.
- **Outcome:** Security provides approval or required remediation comments on the PR.

- **Manual Governance Review (Post-Security Approval):**

- **Trigger:** Security review is approved.

- **Focus:**

    - Semantics: Correct domain modeling? Clear resource names?
    - Design Patterns: Correct Cloud API Pattern? REST principles followed?
    - Consistency: Adherence to URI Standard, Error Handling (LPDP-Mini), Headers, etc.? Documentation quality within OAS?
    - Versioning: Correct application of the API Versioning Standard?
    - Performance NFRs: Documented and realistic?

    - **Outcome:**

    - **Approved:** Governance approves the PR.

    - **Rejected with Comments:** Governance rejects PR with actionable feedback.
    - **Exception Required:** Governance identifies a necessary deviation from standards.
    - **Exception Handling:**

- If an exception is required, the Governance Working Group documents the justification and submits it to the **API Governance Approval Board**.
- The Board reviews, tracks, and formally approves or denies the exception request. Approved exceptions are documented alongside the standard.
- **Final Approval & Merge:** Once all reviews (Automated, Security, Governance) pass and any exceptions are approved, the PR is merged. This merge signifies the OAS is **officially approved**.

Stage 4: Implementation & Testing Phase

- **Trigger:** OAS is approved and merged.
- **Action (Development):** Implement API backend based on the approved OAS contract.
- **Action (Performance Testing):**
- Performance tests are run against the implemented API.
- **Goal:** Validate against documented NFRs.
- **Feedback Loop:** Failures require remediation. Significant changes impacting the OAS contract **MUST** notify API Governance.

Stage 5: Deployment Phase

- **Trigger:** API implementation is complete and tested.
- **Action (Gateway Team):**
- Gateway team is notified (via automated process or ticket) that an approved API version is ready.
- Gateway team pulls the **approved OAS** (from the main branch or designated tag in the repo).
- Configure Apigee proxy based on OAS, applying policies.
- Expose API on the gateway.

Stage 6: Publication Phase

- **Trigger:** API is deployed and ready for consumers.
- **Action (Dev Portal Team / Automation):**
- The **approved OAS** from the repository is ingested into the Developer Portal.
- API reference documentation is automatically generated and published.

Stage 7: Deprecation Phase

- **Trigger:** Decision made during Planning Phase (Stage 1) to deprecate an API, operation, field, etc.
- **Phase 1: Announce** 📣
- Endpoint remains fully functional.
- OAS **MUST** be updated: mark relevant element(s) with `deprecated: true`.
- API responses **SHOULD** include `Deprecation` and `Sunset` headers (per RFC 8594) indicating announcement date and removal date.
- Developer Portal documentation **MUST** be updated with deprecation notice, timeline, and migration path (if any).
- Communicate proactively to known consumers.
- **Phase 2: Sunset** 🌇
- On the announced `Sunset` date:
- Remove the deprecated element(s) from the API implementation.
- The gateway **SHOULD** return appropriate errors (e.g., `404 Not Found` or `410 Gone` for removed endpoints; potentially `400 Bad Request` if a deprecated field is used incorrectly).
- Update OAS to remove the element(s).
- Update Developer Portal documentation.

---

3. RACI Matrix

Activity	Product Owner	Sol. Architect	Dev Team	API Governance WG	API Gov. Board	Security Team	Gateway Team	Dev Portal Team
1. **Planning: Identify Need & Notify Gov.**	**R**, A	**R**	I	C	I	I	I	I
1. **Planning: Initial Governance Review**	C	C	I	**A**, R	I	I	I	I
2. **Design: Request Repo / Versioning**	I	C	**R**, A	C	I	I	I	I
2. **Design: Develop OAS & NFRs**	C	C	**A**, R	C	I	C	I	I
2. **Design: Use Linter Locally**	I	I	**A**, R	I	I	I	I	I
3. **Review: Submit PR**	I	I	**R**, A	I	I	I	I	I

3. **Review: Run Automated Checks (CI)**	I	I	R	**A** (Ruleset)	I	I	I	I
3. **Review: Security Review**	I	C	R	I	I	**A**, R	I	I
3. **Review: Manual Governance Review**	C	C	R	**A**, R	C (Exceptions)	C	I	I
3. **Review: Request/Approve Exception**	I	I	I	R	**A**	I	I	I
3. **Review: Merge Approved PR**	I	I	**A**, R	I	I	I	I	I
4. **Implement: Build API**	I	C	**A**, R	I	I	C	I	I
4. **Implement: Performance Testing**	I	C	R	I	I	I	I	I
5. **Deploy:**	I	I	**R**	I	I	I	I	I

Notify Gateway Team								
5. **Deploy: Configure & Deploy Proxy**	I	I	I	C	I	C	**A**, R	I
6. **Publish: Update Developer Portal**	I	I	I	I	I	I	C	**A**, R
7. **Deprecate: Announce (OAS, Headers)**	C	C	**A**, R	C	I	I	C	R
7. **Deprecate: Sunset (Remove Code)**	I	C	**A**, R	I	I	I	I	I
7. **Deprecate: Update Gateway/Portal**	I	I	I	C	I	I	R	R

Legend: **R**=Responsible, **A**=Accountable, **C**=Consulted, **I**=Informed

Proposed Roadmap

Thematic Roadmap

Our strategy will progress through three distinct phases:

- **Q4 2025: Foundation & Piloting:** Formalize all core standards and validate them with a pilot project.
- **Q1 2026: Expansion & Enablement:** Operationalize our governance process and expand modeling to the next strategic domain.
- **Q2 2026: Operationalization & Automation:** Scale governance through automation and drive adoption with a world-class Developer Experience (DX).

Q4 2025 — Foundation & Piloting

Goal: Formalize our core API standards, processes, and reference architectures while designing the first canonical models for a selected pilot project.

Foundational Playbook & Governance

Objective: Draft, socialize, and ratify all v1.0 foundational governance documents.
Deliverables:

- API Style Guide v1.0
- Draft Domains/SubDomains and their corresponding URIs
- API Security Standard v1.0
- “Sell” Side Reference Architecture v1.0
- “Buy” Side Reference Architecture v1.0
- API Lifecycle and Governance Process v1.0

Activities: * Draft all initial governance and reference documents. * Socialize drafts with Architecture, Security, and Engineering. * Incorporate feedback and ratify through the API Governance.

Canonical Data Modeling (Pilot)

Objective: Validate the canonical modeling approach through a real-world pilot project.
Deliverables:

- Ratified Decision Document on Modeling Strategy (Build vs. MEF/TMF).
- Selected Pilot Project (e.g., LMCC / MCGW) with defined Bounded Contexts.
- Immutable Core and Optional Extensions for pilot domain.
- Ratified Canonical Models v1.0 published to the Schema Registry.
  Activities:
- Convene working group and perform rapid gap analysis.
- Conduct domain workshops with subject matter experts.
- Draft and review OpenAPI/JSON Schema specifications.

Governance Automation & Observability

Deliverables:

- Implement automated governance checks using Spectral / Redocly CLI.
- Integrate validation pipeline within pilot CI/CD workflow.
- Implement W3C trace headers in gateway

Q1 2026 — Expansion & Enablement

Goal: Operationalize the governance process, expand canonical modeling to the next domain, and initiate structured enablement across teams.

Foundational Playbook & Governance

Deliverables:

- Operationalized governance process with first formal API design reviews.
- Enablement and training materials delivered to product teams.

Partner Data-Sharing & Consent Architecture

Objective: Define and enforce the shared-customer model, including consent-based visibility and partner projections.
Cross-Dependencies: Security, Data Governance, API Governance.
Deliverables:

- Defined Partner Data-Sharing Model.
- Consent Artifact Schema and Enforcement Framework.
- Partner “Subset Visibility” API Design Pattern.

Canonical Data Modeling (Expansion)

Deliverables:

- Selected next business domain for modeling (e.g., Order Management).
- v1.0 Canonical Models ratified and published.

Tactical Engagement & Analysis

Deliverables:

- Completed API portfolio analysis for the target domain.
- Prioritized backlog of alignment and remediation tasks delivered to project teams.
- **Platform & Enablement (New)**

Deliverables:

- API Observability Standard v1.0 drafted.
- Defined Core Metrics (latency, error rate, throughput).
- Published SLO Template Library for API services.
- Assess if Swaggerhub can satisfy needs of internal API Portal

---

Q2 2026 — Operationalization & Automation

Goal: Scale governance through automation, strengthen reliability with observability standards, and improve adoption through a world-class Developer Experience (DX).

Focus Areas:

- Expand automated governance checks across enterprise CI/CD pipelines.
- Launch internal API Portal

- Establish SLO Governance Dashboards for operational visibility.
- Begin cross-domain canonical alignment and schema versioning.

API Governance Metadata Model

1. Introduction

This document defines the metadata model for API governance. Its purpose is to capture the essential "delta" information—such as ownership, business context, and governance status—that is not already present in the OpenAPI Specification (OAS) file.

This model is a two-part system designed to work with an API's source of truth (the OAS) to enable discovery, automated governance, and lifecycle management.

1. **API-Level Metadata (`api-metadata.yaml`):** A separate metadata file that lives alongside the OAS file in the API's repository. This file describes the API as a whole.
2. **Endpoint-Level Metadata (OAS `x-` tags):** Custom extension properties placed *inside* the OAS file on specific operations to enable fine-grained, automated policy enforcement.

This structure makes the metadata discoverable by a cataloging tool (like Backstage) while keeping the machine-readable, automated governance rules co-located with the endpoint contracts they apply to.

2. API-Level Metadata (The `api-metadata.yaml` File)

This metadata **must** be stored in a dedicated YAML file (`api-metadata.yaml`) in the API's root directory.

- `schemaVersion` (String, Required): The version of this metadata schema, which enables CI validation and controlled evolution.
- **Value:** `v1.0`

2.1. Ownership & Team

Identifies who is responsible for the API's development, maintenance, and business alignment. This is critical for accountability and for consumers to know who to contact.

- `apiName` (String, Required): The human-readable, canonical name of the API (e.g., "User Profile API", "Payment Orchestration Service").
- `ownershipModel` (String, Required): For the current governance scope, this value is always `Lumen-Owned`.This ensures a consistent ownership declaration and establishes a foundation for future expansion (e.g., `Externally-Owned`, `Joint-Owned`) when the internal API repository is introduced.
- `assetId`*(string, required)*: **Lumen Asset ID (SYSGEN)** that maps the API to its registered owner application in the Asset Management system (e.g., `SYSGEN-45123`). Used for authority, audit, and MAL PoC linkage.
- `businessOwner` (String, Required): The primary stakeholder or Product Manager responsible for the API's business function and roadmap (e.g., an email, user ID, or group alias).
- `technicalOwner` (String, Required): The Engineering Lead or Architect responsible for the API's technical design, implementation, and operational health.
- `developmentTeam` (String, Required): The name or alias of the team that builds and maintains the API (e.g., "identity-platform-team").
- `supportContact` (String, Required): The official support channel for consumers (e.g., a group email `api-support@example.com`, a Slack channel `#slack-team-identity`).

2.2. Business Context

Provides the "why" and "where" for the API within the larger enterprise architecture.

- `consumerAudience` (String, Required): The intended consumer channel, which dictates security and exposure.

- **Values:** `internal-ui` (for internal UIs/frontends), `internal-service` (for other internal systems/automation), `external-partner`, `external-public`

- `apiLayer` (String, Required): The architectural classification of the API, based on its primary role.
- **Values:** `Experience`, `Canonical`, `Process`, `System`
- `system` (String, Optional): A key/name that links this API to a larger "System" or "Domain" in the software catalog (e.g., `billing-platform`, `crm`, `customer-identity-domain`).

2.3. Governance & Lifecycle

Tracks the API's maturity and the level of governance applied.

- `lifecycleStage` (String, Required): The current stage of the API's life.
- **Values:** `Design`, `In-Development`, `Active`, `Deprecated`, `Retired`
- `governanceLevel` (String, Required): The level of review and adherence to standards applied to this API. This informs consumer trust and risk.
- **Values:** `Fully Governed`, `Partially Governed`, `Lift and Shift`, `Exception`
- `deprecationDate` (String, Optional): (If `lifecycleStage` is "Deprecated") The date (ISO 8601) when the API will no longer be supported.
- `sunsetDate` (String, Optional): (If `lifecycleStage` is "Deprecated") The date (ISO 8601) when the API will be shut down.

2.4. Technical & Design Details

Captures high-level technical attributes not defined in the OAS.

- `apiStyle` (String, Required): The architectural style of the API.
- **Values:** `REST`, `RPC-style`, `GraphQL`, `gRPC`, `Async`
- `protocol` (String, Required): The primary protocol used.
- **Values:** `HTTPS`, `WSS` (WebSockets), `AMQP` (Messaging), etc.

2.5. Security & Compliance

Defines the overall risk and compliance posture for the API.

- `dataClassification` (String, Required): The **highest** sensitivity level of data handled by *any* endpoint in the API. This acts as a high-water mark for compliance.
- **Examples:** `Public`, `Internal`, `Confidential`, `Restricted (PII, PCI, PHI)`
- `complianceRequirements` (List, Optional): A list of any compliance regulations that apply to the data handled by this API.
- **Examples:** `["GDPR", "HIPAA", "PCI-DSS"]`

2.6. Review Audit Trail

An immutable log of the *latest* governance review. This section is updated automatically by the governance process or tooling upon a successful review.

- `reviewAuditTrail` (Object, Required): A container for all review-related metadata.
- `reviewId` (String, Required): A unique identifier for the latest governance review.
- `reviewDate` (String, Required): The timestamp (ISO 8601) when the latest review was completed.
- `reviewStatus` (String, Required): The outcome of the review.
    - **Values:** `Pending`, `Approved`, `Approved with Conditions`, `Rejected`
    - `securityReviewId` (String, Optional): A unique identifier that links this governance review to an internal risk, compliance, or security review record.

2.7. Schema Validation

To enable automated CI validation, editor autocompletion, and ensure conformity, all `catalog.yaml` files **must** validate against the official JSON Schema.

- **Schema Location:** `catalog-schema.json`

3. Endpoint-Level Metadata (Inside the OAS File)

For fine-grained, automated governance (e.g., security policies, linting, gateway configuration), we must apply metadata at the **Operation (Endpoint) level**.

This metadata **must** be stored *inside* the OAS file using **OpenAPI Specification Extensions** (fields prefixed with `x-`).

3.1. How to Store Endpoint Metadata

This metadata lives directly on the operation object within the OAS file, making the spec a self-contained source of truth for automation.

Example within the OAS (`api.oas.yaml`):

```
paths:
  /users:
    post:
      summary: Create a new user
      operationId: createUser

      # --- Automated Governance Metadata Starts Here ---
      x-data-classification: "Restricted (PII)"
      x-operation-type: "lro" # (Long-Running Operation)
      x-financial-operation: false
      x-idempotent: false
      # --- Automated Governance Metadata Ends Here ---

      tags:
        - Users
      requestBody:
        description: User object to be created
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '202':
          description: Accepted. The user creation is in progress.
```

```
        '400':
          description: Invalid input
```

3.2. Key Endpoint-Level Metadata Fields

- `x-data-classification` (String, Required): The sensitivity of the data handled *by this specific endpoint*. This is the most critical field for automated security and can be more granular than the API-level high-water mark.
- **Examples:** `Restricted (PII)`, `Confidential`, `Internal`, `Public`
- `x-operation-type` (String, Optional): Defines the behavior of the endpoint, allowing the API gateway to apply different policies (e.g., timeouts, retry logic).
- **Examples:** `lro` (Long-Running Operation), `bulk-data`, `standard-sync`
- `x-financial-operation` (Boolean, Optional): Flags if this operation initiates a financial transaction, which can trigger stricter audit logging policies. Defaults to `false`.
- `x-idempotent` (Boolean, Optional): Explicitly declares if a `POST` or `PATCH` operation is safe to retry. This is crucial for resilient consumer design.

3.3. Addressing the "Messy Presentation" Concern

A common concern is that `x-` tags will clutter the rendered documentation (e.g., Swagger UI). This is a solvable presentation problem.

1. **Priority:** The primary consumer of `x-` tags is **automation** (linters, security scanners, API gateways). It is essential for these tags to be machine-readable.
2. **Solution:** The documentation tools should be configured to *interpret* these tags rather than just displaying them. Your developer portal can be customized to:
3. **Hide** governance tags from the default view.
4. **Render them as badges:** `x-data-classification: "Restricted (PII)"` could be rendered as a highly visible red "PII Data" badge.

5. **Render them as icons:** `x-operation-type: "lro"` could show a small "hourglass" icon.

4.0 API-Metadata JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "API Governance Catalog Metadata",
  "description": "Schema for the catalog.yaml file, which provides governanc
  "type": "object",
  "properties": {
    "schemaVersion": {
      "description": "The version of this metadata schema.",
      "type": "string",
      "enum": ["v1.0"]
    },
    "apiName": {
      "description": "The human-readable, canonical name of the API.",
      "type": "string",
      "minLength": 1
    },
    "ownershipModel": {
    "description": "Defines the ownership classification of the API. For gov
    "type": "string",
    "enum": ["Lumen-Owned"]
  },
    "assetId": {
      "type": "string",
      "minLength": 1,
      "description": "Lumen Asset ID (SYSGEN) linking to Asset Management",
      "examples": ["SYSGEN-45123"]
    },
    "businessOwner": {
      "description": "The primary stakeholder or Product Manager (email, ID,
      "type": "string",
      "minLength": 1
    },
    "technicalOwner": {
      "description": "The Engineering Lead or Architect (email, ID, or alias
      "type": "string",
      "minLength": 1
    },
    "developmentTeam": {
      "description": "The name or alias of the team that builds and maintain
      "type": "string",
```

```
      "minLength": 1
    },
    "supportContact": {
      "description": "The official support channel for consumers (email or S
      "type": "string",
      "minLength": 1
    },
    "consumerAudience": {
      "description": "The intended consumer channel, which dictates security
      "type": "string",
      "enum": [
        "internal-ui",
        "internal-service",
        "external-partner",
        "external-public"
      ]
    },
    "apiLayer": {
      "description": "The architectural classification of the API, based on
      "type": "string",
      "enum": [
        "Experience",
        "Canonical",
        "Process",
        "System"
      ]
    },
    "system": {
      "description": "A key/name that links this API to a larger System or D
      "type": "string"
    },
    "lifecycleStage": {
      "description": "The current stage of the API's life.",
      "type": "string",
      "enum": [
        "Design",
        "In-Development",
        "Active",
        "Deprecated",
        "Retired"
      ]
    },
    "governanceLevel": {
      "description": "The level of review and adherence to standards applied
      "type": "string",
      "enum": [
        "Fully Governed",
        "Partially Governed",
```

```
      "Lift and Shift",
      "Exception"
    ]
  },
  "deprecationDate": {
    "description": "The date (ISO 8601) when the API will no longer be sup
    "type": ["string", "null"],
    "format": "date-time"
  },
  "sunsetDate": {
    "description": "The date (ISO 8601) when the API will be shut down.",
    "type": ["string", "null"],
    "format": "date-time"
  },
  "apiStyle": {
    "description": "The architectural style of the API.",
    "type": "string",
    "enum": [
      "REST",
      "RPC-style",
      "GraphQL",
      "gRPC",
      "Async"
    ]
  },
  "protocol": {
    "description": "The primary protocol used.",
    "type": "string",
    "enum": [
      "HTTPS",
      "WSS",
      "AMQP",
      "MQTT"
    ]
  },
  "dataClassification": {
    "description": "The highest sensitivity level of data handled by any e
    "type": "string",
    "enum": [
      "Public",
      "Internal",
      "Confidential",
      "Restricted (PII, PCI, PHI)"
    ]
  },
  "complianceRequirements": {
    "description": "A list of any compliance regulations that apply to thi
    "type": "array",
```

```
      "items": {
        "type": "string"
      },
      "uniqueItems": true
    },
    "reviewAuditTrail": {
      "description": "An immutable log of the latest governance review.",
      "type": "object",
      "properties": {
        "reviewId": {
          "description": "A unique identifier for the latest governance revi
          "type": "string",
          "minLength": 1
        },
        "reviewDate": {
          "description": "The timestamp (ISO 8601) when the latest review wa
          "type": "string",
          "format": "date-time"
        },
        "reviewStatus": {
          "description": "The outcome of the review.",
          "type": "string",
          "enum": [
            "Pending",
            "Approved",
            "Approved with Conditions",
            "Rejected"
          ]
        },
        "securityReviewId": {
          "description": "A unique identifier that links to an internal risk
          "type": "string"
        }
      },
      "required": [
        "reviewId",
        "reviewDate",
        "reviewStatus"
      ],
      "additionalProperties": false
    }
  },
  "required": [
    "schemaVersion",
    "apiName",
    "ownershipModel",
    "assetId"
    "businessOwner",
```

```
    "technicalOwner",
    "developmentTeam",
    "supportContact",
    "consumerAudience",
    "apiLayer",
    "lifecycleStage",
    "governanceLevel",
    "apiStyle",
    "protocol",
    "dataClassification",
    "reviewAuditTrail"
  ],
  "additionalProperties": false
```

Evangelism & Adoption

API Evangelism Master Plan: A Repeatable Framework

Purpose

To create a realistic, repeatable, and effective framework for driving the adoption of the new API Strategy. This plan is designed to be piloted with a key department and then scaled to the wider organization.

Core Message (The "Message House"):

We are launching a new, unified API framework. Everything we do anchors to these three pillars:

1. **Predictable APIs:** (Via the new Style Guide)
2. **Automated Governance:** (Via the new Publishing Playbook)
3. **Faster Delivery:** (By automating the process from idea to production)

Phase 0: Prerequisite Readiness (The "Go/No-Go" Gate)

Goal: To finalize all assets and solve technical blockers *before* the first announcement. This phase is owned by the API Governance team.

Action Item	Owner	Status	Details
Define & Approve Developer Tooling (POC)	Evangelism Lead (Maninder), API Platform Engineer	**Blocked**	**CRITICAL:** Current tool (SwaggerHub) does not support Spectral linting at design-time. Need POC + Legal/Licensing review for a VS Code + Spectral + 42 Crunch-like environment.
Finalize Core Assets	API Governance Team	**Blocked**	The **API Style Guide** , Governance Process, Centralized Repo Design, Developer workflow for authoring and working with governance workflow and the **API** Publishing Playbook (Public **vs. Internal)** must be 100% final and published on Confluence.
Create Evangelism Toolkit	Evangelism Lead (Maninder), Ravi	To-Do	Create the "master" PowerPoint deck. This deck *must* include the "Why" (Strategy), the "What" (Style Guide), and the "How" (a visual of the new Governance Process).
Solve "Confluence Blocker"	Evangelism Lead (Maninder)	To-Do	1. Get the official "How to get a

Action Item	Owner	Status	Details
			Confluence license" link. 2. Create a SharePoint site/folder. 3. Upload PDF versions of the Style Guide and Playbook to SharePoint.
Create "Intake Form"	Evangelism Lead (Maninder)	To-Do	Create the official Jira Form for teams to request "White Glove" support, as mentioned by management.
Create FAQ / Objection Handling Doc	Evangelism Lead (Maninder)	To-Do	Preemptively answer objections ("Will this slow us down?", "Why is this required for internal APIs?") to reduce friction.

Phase 1: Pilot Launch (Target: Oscar's Department)

Goal: To test, refine, and prove our evangelism model with a friendly, high-context audience.

Step	**Channel**	**Target Audience**	**Key Message & "Ask"**
1. **The Announcement**	Mailing List	All (Engineers, PMs, Leaders)	"Introducing the new API Governance process, built on 3 pillars: Predictable APIs, Automated Governance, & Faster Delivery. We're piloting it with you." **Links:** Confluence (w/ license link) & SharePoint (PDFs).

2. **Developer Deep Dive**	Engineering Garage / Community of Practice (CoP)	Engineers / Developers	**"Live Demo: How to Publish Your API in 2026."** We will walk through the *entire* `feature-branch` -->`governance-approved` --> `main` workflow to show how the pipeline enforces the new standards.
3. **Product Manager Sync**	PM Forum / Scheduled Meeting	Product Managers	**"How the New Governance Process Accelerates Your Roadmap."** We will focus on how this process enables the Partner/Wholesale models and reduces manual review time.
4. **"White Glove" Pilot**	Direct Engagement	2-3 Selected Projects	We will offer hands-on support for 2-3 key projects in this department. This is our chance to be embedded, find gaps in our process, and get our first "wins."
5. **Pilot Retro & Success Story**	Internal Meeting	API Governance Team + Pilot Team Leads	1. "What worked? What was confusing? What's missing in the docs?" 2. **Create a 1-slide "Success Story"** with a "Before & After" and a quote from the dev lead. Nothing accelerates adoption like early wins.

Phase 2: General Rollout (Target: All PAX/STEPN)

Goal: To scale the successful, refined pilot program to the entire engineering and product organization.

Step	Channel	Target Audience	Details
1. **Public Announcement**	MyEvive / SharePoint Blog Post	All Engineering & Product	Take the successful "Mailing Letter" (from Phase 1), refine it with feedback, and post it as the official "Big Bang" announcement. **Feature the "Success Story"** from the pilot.
2. **Company-Wide Demos**	Brown Bags / Lunch & Learns (Scheduled)	All Engineers / Developers	Take the successful "Live Demo" (from Phase 1) and offer it as a recurring, scheduled workshop for all.
3. **Open "White Glove" Queue**	Jira Intake Form	All Teams	Formally open the "White Glove" support queue (created in Phase 0) to all teams, with clear SLAs based on what we learned in the pilot.

Phase 3: Sustain & Measure (The "Repeatable Framework")

Goal: To ensure this effort has a lasting impact and to prove its value to leadership.

Activity	Frequency	Owner	Metrics for Success (Leadership KPIs)
Community of Practice (CoP)	Monthly	API Governance Team	# of attendees; # of questions/topics submitted.
API Evangelism Blog	Monthly	Evangelism Lead (Maninder)	Post views; engagement; new topics from the "post ideas" list.
White Glove Support	Ongoing	API Architects	**% of new APIs conformant without human involvement**; queue backlog.
Leadership Report	Quarterly	Evangelism Lead (Maninder)	**Process Adoption:** Time from idea -> `governance-approved` OAS. **Efficiency:** Reduction in manual review time. **Discoverability:** % of APIs discoverable in the central catalog. **Consolidation:** Reduction in API duplication.

Why API Governance Matters: The Engine of Digital Velocity

Build the Future, Not the Firefights:

Why API Governance Is the Engine Behind Lumen's Growth, Speed, and Partner Ecosystem ?

APIs are no longer just integration tools — they are the **growth engine** of modern digital businesses.
At Lumen, APIs now determine:

- how fast we activate services,

- how easily partners can sell our products,
- how consistently customers experience our network,
- and how quickly we can expand into new markets we don't own physically.

But today, our biggest barrier isn't network capacity or cloud footprint.
It's **inconsistent, siloed, non-reusable APIs** that force us into manual work, create friction for partners, and slow our entry into new sales channels.

This is why Lumen's API Governance Framework and Style Guide exist.
Not to enforce rules — but to **unlock the strategic business goals that leadership has committed to**.

The Business Imperative (THE "WHY")

This API strategy is not a technical initiative.
It is a **business transformation program**.

Below are the four business drivers guiding everything we are doing.

1. Unlock New Revenue and Growth

APIs are the new sales channel — and this strategy opens them.

Lumen can now sell network services **directly in cloud marketplaces** (AWS, Azure), giving customers the ability to buy Lumen products where they already shop.

This is a fundamental shift in our business model.

But this only works if we offer:

- standardized APIs,
- predictable workflows,
- clean provisioning interfaces,
- and world-class partner onboarding.

Inconsistent APIs block our ability to scale in cloud marketplaces; standardized governed APIs **unlock global reach**.

Beyond marketplaces, this strategy lets us expand our footprint by using partners as **offnet providers**, enabling Lumen to sell services **anywhere**, even in markets where we don't own the physical network.

This is the fastest route to **geographic expansion** Lumen has ever had.

2. Increase Operational Agility & Speed to Market

Automation is the competitive weapon — APIs are the mechanism.

Our competitors (PacketFabric, MP) already differentiate through speed.
To win, we must deliver what they cannot:
network activation in minutes, not months.

That requires:

- digital quoting,
- digital ordering,
- digital provisioning,
- and digital activation —
  with **zero human steps**.

The only way to achieve this is with governed, consistent APIs that operate as a unified automation layer.

The Style Guide solves the problem of inconsistent, ad-hoc patterns that slow down automation.

Once consistent, our APIs become an assembly line — predictable, repeatable, and fast.

This is not an architecture decision.
It is a **speed-to-market strategy**.

3. Enhance the Partner Experience

In a partner-driven business, the integration is the product.

For partners like AWS, Azure, AT&T, and data center operators, the quality of our APIs **IS** the experience.

Today, we struggle with:

- inconsistent patterns
- confusing payloads
- unpredictable behavior
- "manually assisted" integrations
- slow onboarding cycles

These cause friction, which translates into lost deals and lost strategic partnerships.

A professionally governed API program changes the equation:

- predictable interfaces
- reusable canonical models
- clear contracts
- consistent pagination, error handling, naming
- repeatable onboarding playbooks
- automated linting and governance checks

This transforms Lumen into a **partner-ready, integration-friendly provider** — the one enterprise customers want to work with.

This is how we become the **partner of choice** in the cloud and telecom ecosystem.

4. Future-Proof the Technology Platform

A single API framework replaces decades of one-off integrations.

Historically, every new partner, system, or initiative required a **brand-new, bespoke integration**:

- different models
- different patterns
- different versioning
- different authentication
- different workflows

This created:

- enormous operational cost
- fragile dependencies
- impossible scalability
- low code reusability
- slow onboarding
- higher security risk

The new API Governance Framework and Style Guide solve this by creating **one reusable framework** for all integrations — internal, partner, marketplace, automation, wholesale, enterprise.

This lets us:

- scale with partners
- add new services faster
- modernize systems without breaking experiences
- grow into new cloud ecosystems
- avoid costly redesigns
- reduce architectural fragmentation

This is how Lumen stops building **expensive one-off solutions** and starts building a **platform**.

The Architecture: A Foundation Built for These Business Goals

To achieve these business outcomes, we use the **Strategic Four-Layer Architecture**:

1. **Experience APIs** — tailored for each channel

2. **Canonical APIs** — stable enterprise contracts

3. **Process APIs** — orchestrated business workflows

4. **System APIs** — clean abstraction from backend systems

This architecture ensures:

- reuse across channels, partners, and marketplaces
- stability when backend systems change
- clarity for developers
- reliability for partners
- standardization for governance
- scalability for future integrations

This is how we build once and scale infinitely.

The Style Guide: Business Value Delivered Through Technical Consistency

The Style Guide is the **operational mechanism** that makes these business drivers real.

It ensures:

- Consistent contracts
- Predictable behavior
- Zero guesswork
- Safe retries
- Enterprise-grade pagination
- Clean resource modeling
- Unified error handling
- Compression & content negotiation
- Bulk operations via Claim Check
- LRO for marketplace provisioning
- Mandatory idempotency
- ETag for concurrency control

Every standard exists for a **business reason**:

- Idempotency prevents billing errors
- ETags prevent data corruption
- Pagination ensures performance at scale
- Canonical models accelerate marketplace certification
- LROs enable automated provisioning
- Claim Check enables massive cloud-scale data flows

These are not technical preferences — they are **enablers of revenue, speed, partner trust, and scalability**.

Security: A Business-Grade Foundation

Audience-driven API security ensures:

- partners get only what they're entitled to
- multi-tenant access is secure
- compliance is built-in
- onboarding is frictionless
- auditability is consistent

This protects revenue AND accelerates onboarding.

Governance Is the Accelerator, Not the Bottleneck

Governance enforces consistency **automatically**, using:

- Spectral linting
- SDLC integration
- governance metadata
- white-glove assessments
- publishing workflows

This creates:

- low-friction design
- higher quality
- predictable integration cycles
- safer releases
- faster delivery

Governance is the **runway**, not the restraint.

The Outcome: A Revenue-Generating, Partner-Ready Digital Platform

When we apply:

- API Governance
- the Four-Layer Architecture
- the Style Guide
- canonical models
- partner-driven security
- automated CI/CD checks

we transform Lumen from a telecom company with APIs into a **platform company with revenue-generating capabilities.**

This strategy:

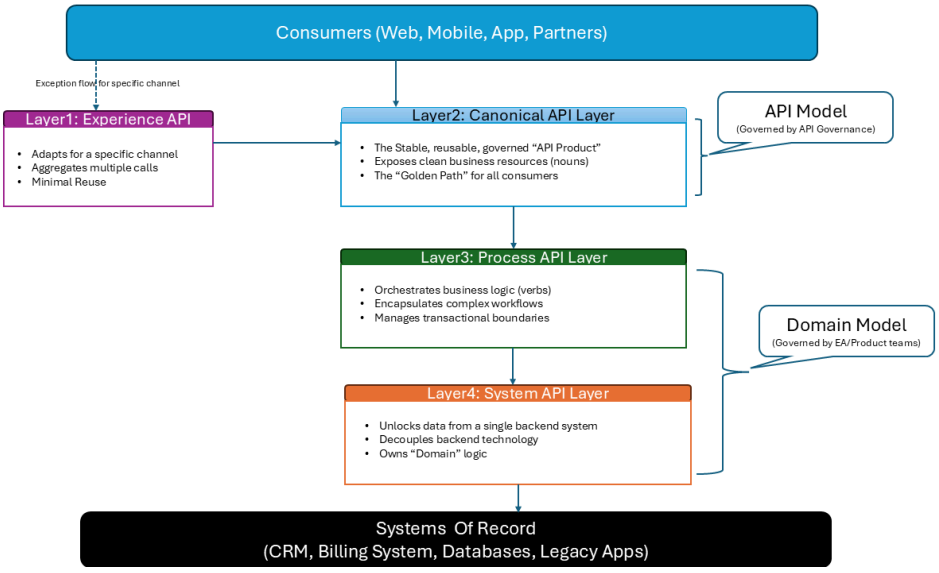
- unlocks new global markets
- accelerates automation
- increases partner adoption
- reduces operational cost
- improves developer experience
- eliminates one-off integrations
- prepares us for cloud-native growth
- strengthens our competitive position

This is not a technology initiative. This is a **business growth initiative powered by APIs.**

This is how Lumen wins.

Architectural Principles & Patterns

A Strategic Four-Layer Approach to API Architecture



A Layered Approach to API Design

In a large-scale enterprise like Lumen, a well-defined API architecture is the foundation for achieving digital agility, scalability, and reusability. While the traditional three-layer model (`Experience`, `Process`, `System`) provides a good starting point, a more mature **four-layer architecture** offers greater clarity by explicitly separating the stable, public-facing API contract from consumer-specific adaptations.

This model serves three distinct but related functions: as a classification system, a design pattern, and most critically, as an architectural pattern.

The Three Facets of the Four-Layer Model

It's important to understand that this model provides value in three distinct ways.

1. **As a Classification System:** At its simplest, the model is a taxonomy. It gives us a shared vocabulary to categorize our APIs based on their function (e.g., "Is this a Process API or a System API?"). This helps us inventory our assets and understand their intended purpose at a glance.
2. **As a Design Pattern:** The model provides guidance for separating concerns within a single application or service. It encourages developers to write cleaner, more maintainable code by isolating orchestration logic from backend integration logic.
3. **As an Architectural Pattern:** This is its most strategic role. The model defines the **non-negotiable rules of interaction *between* the layers**. It mandates, for example, that an Experience API must call the Canonical Layer and is forbidden from calling a System Layer directly. This is the rule that enforces enterprise-wide decoupling, security, and governance.

The Four Layers Explained

This refined model consists of four logical layers: Experience, Canonical, Process, and System.

1. The Experience API Layer (The Exception)

This is a thin, non-reusable adaptation layer whose sole purpose is to handle the unique needs of a specific consumer that cannot be efficiently served by the standard canonical API. It is an **exception, not the norm**.

- **The "Why":** To optimize for a specific channel's user experience (e.g., a mobile app, a partner portal) or to handle the technical vagaries of a particular consumer.
- **Advantages:**
- **Flexibility & Usability:** Tailored to specific user experiences, making them easier for front-end developers to consume.
- **Consumer Agility:** Allows teams to rapidly build or change APIs to meet new channel needs without impacting the core API products.
- **Performance Optimization:** Can aggregate multiple calls to the Canonical Layer into a single, efficient response for "chatty" clients.
- **What to Watch For (Architectural Concerns):**
- **Minimal Reuse:** These APIs are purpose-built and should not be reused, leading to potential maintenance overhead if not governed properly.
- **No Business Logic:** This layer must never contain core business rules. It only transforms, aggregates, and can obfuscate data for presentation.
- **User-Centric Design:** Low latency is crucial. Performance must be monitored and optimized for speed and responsiveness.
- **Scalability & Deployment:** Must be designed for elastic scalability to handle fluctuating user traffic. Regional deployment closer to end-users may be required to reduce latency.
- **Edge Security:** As the public-facing entry point, this layer requires robust security measures like DDoS protection, Web Application Firewalls (WAF), and client certificate management.
- **Governance:** The creation of an Experience API must be a deliberate, justified decision to prevent API sprawl.

2. The Canonical API Layer (The Norm)

This is the **default, reusable, and governed API "product"** for the enterprise. It is the stable, well-documented storefront that should be used by all internal and external channels whenever possible.

- **The "Why":** To provide a single, consistent, and stable contract for all consumers, abstracting away the complexity of the internal systems. This is the API that realizes our strategic business goals.
- **Advantages:**
- **High Reusability:** Designed to be the "golden path" used by 90% of consumers.
- **Stability & Decoupling:** Provides a long-term, stable contract. The internal implementation can change, but this public-facing contract remains the same.
- **Clear Ownership:** Managed as a formal product with a clear roadmap and governance.
- **What to Watch For:**
- **Purity:** This layer must be protected from being polluted with channel-specific logic or internal implementation details.
- **Data-Centric:** Focuses on exposing clean, consistent representations of business resources (the "nouns" like `Order`, `Service`, `Customer`).

3. The Process API Layer (The Engine)

This internal-facing layer contains the business logic that orchestrates multiple systems to complete a business process. It is the engine that powers the layers above it.

- **The "Why":** To encapsulate and reuse complex business processes and workflows, ensuring that business rules are applied consistently regardless of which channel initiated the request.
- **Advantages:**
- **Consistency:** Standardizes business processes across the organization.

- **Agility:** The business process can be changed or optimized in one place, and that improvement is instantly available to all consuming APIs.
- **Business Logic Encapsulation:** Centralizes complex orchestration, preventing business rules from being duplicated.

- **What to Watch For (Architectural Concerns):**

- **Owns Process Logic:** This is the correct place for logic that spans multiple systems or domains (e.g., "update billing, then update provisioning").

- **Transactional Boundaries:** This is where business transactions often begin and end, including logic for rollbacks, error handling, and concurrency management.
- **Dependency & Version Management:** Requires careful management of dependent APIs and their versions. A visual dependency map is critical for documentation.
- **Observability:** Requires robust monitoring for performance (response time, throughput), reliability (uptime), and business process analytics (workflow states, processing time). Alerts on dependent API performance are crucial.
- **Resilience:** The design must include strategies for graceful degradation, defining fallback behaviors or compensation steps if a dependent API fails.
- **Auditing:** Must ensure detailed logging of critical events per user for auditing and debugging purposes.

4. The System API Layer (The Foundation)

These are the foundational APIs that provide direct, unlocked access to core systems of record (e.g., a legacy CRM, a billing system, a database).

- **The "Why":** To expose the data and capabilities of a specific backend system in a clean, reusable way, while insulating the rest of the architecture from the specifics of that system's technology.

- **Advantages:**

- **Reusability & Stability:** Highly reusable across different processes and provides a stable foundation for other APIs to build upon.

- **Backend Decoupling:** If a legacy system is replaced, only its System API needs to be updated.
- **Access Control:** Provides a secure access point to core data.

- **What to Watch For (Architectural Concerns):**

- **Owns Domain Logic:** This layer **must** contain and enforce the business logic specific to the entity it manages (e.g., a CRM-System-API must enforce the rules for a valid "Customer").

- **No Process Logic:** This layer should **never** contain orchestration logic that calls other systems. A System API should not know about the existence of other systems.
- **Robustness:** Redundancy and high availability are critical for these foundational APIs.
- **Security:** Requires strong access controls (e.g., MFA for state-modifying APIs) and may reside in a highly restricted network segment.
- **Data Governance:** Must ensure data integrity, and enforce data retention and encryption policies.

Governance and Ownership: API as a Product

A successful API strategy requires a clear ownership model where APIs are treated as products. While governance is a shared responsibility, the roles for each group are distinct and complementary.

Role	Responsibility
Product Teams	Act as the **"API Product Owners."** They own the end-to-end lifecycle of the APIs within their business domain, across all four layers. This includes defining the roadmap for the Canonical API (Layer 2), prioritizing features, and ensuring the underlying Process and System APIs (Layers 3 & 4) meet the business needs.
API Governance	Act as the **"Portfolio Managers"** or "City Planners." They do not own individual APIs but own the overall API landscape. They define the enterprise-wide standards (the "building codes"), manage the API catalog,

Role	Responsibility
	and ensure consistency and quality across all API products.
Enterprise Architecture (EA)	Act as the **"Chief Architects"** or "Building Inspectors." They own the strategic patterns, like this four-layer model, and ensure that the technical implementations are sound, scalable, and align with the broader technology vision of the company.

Why These Should Be Separate APIs, Not a Single Monolith

The core reason for separating these layers is to prevent the creation of a **monolithic API** that becomes brittle, difficult to maintain, and a bottleneck for business innovation. A single API that attempts to do all three would suffer from several critical flaws:

a. Tight Coupling and Lack of Reusability

- **Single API Problem:** If a single API handles a request from a mobile app, calls a legacy system, and orchestrates a process, it is tightly coupled to all three. If the mobile app's data needs change, or the legacy system is replaced, or the business process is altered, you have to modify and re-deploy the entire monolithic API.
- **Layered Solution:** In the layered model, an **Experience API** for a new web portal can simply call the same **Process API** that the mobile app uses. This promotes code reuse and prevents redundant development. If a legacy billing system is replaced, only the corresponding **System API** needs to be updated. The higher-level Process and Experience APIs remain unaffected, as their contracts with the System API remain stable.

b. Inefficient Scalability and Performance

- **Single API Problem:** A monolithic API becomes a single point of failure and a performance bottleneck. A request for a simple data retrieval from the mobile app is forced through the same code path

as a complex, multi-system orchestration, leading to inefficient resource utilization.
- **Layered Solution:** Each layer can be scaled independently. If your mobile app experiences a surge in traffic, you can scale the **Experience API** without having to scale the expensive and resource-intensive **System APIs** that interact with core databases. Similarly, you can optimize each layer for its specific function (e.g., a high-volume caching layer for the Experience API).

c. Hindered Agility and Innovation

- **Single API Problem:** Changes become slow and risky. Any small change to the API requires a full regression test of the entire application, as a modification for one client could inadvertently break another. This slows down your ability to respond to market demands.
- **Layered Solution:** The layered model allows for parallel development and faster innovation.
- **Experience APIs** can be built quickly to experiment with new user experiences, as they are simply compositing existing functionality.
- **Process APIs** provide a stable foundation for business workflows.
- **System APIs** can be developed and maintained by specialized teams who are experts on the core systems of record, without worrying about how those systems are presented to the end user.

Api Classification Example

Consider the scenario of allowing a customer to upgrade their service plan.

The Right Way (Four-Layer Model):

1. **System APIs:**
2. `CRM-System-API`: Exposes raw customer data and enforces all business rules for a valid customer.
3. `Billing-System-API`: Exposes billing data and enforces all rules for valid billing plans.

4. `Provisioning-System-API`: Exposes low-level provisioning actions and enforces rules for valid service configurations.

5. **Process API:**

6. `Update-Service-Process-API`: Contains the cross-system orchestration workflow:

    1. Calls `Billing-System-API` to change the plan.
    2. Calls `Provisioning-System-API` to apply the new service configuration.
    3. Manages the transaction, with logic to roll back if a step fails.
    4. **Canonical API:**

7. `Customer-API`: Exposes a clean, governed `Customer` resource, including their `ServicePlan`. Provides a standard `PATCH /customers/{id}/servicePlan` endpoint. The mobile and web teams both build their UIs against this stable, reusable contract.

8. **Experience API (An Exception):**

9. A new, high-value partner requires a legacy SOAP XML interface for service upgrades. Instead of polluting the modern `Customer-API`, we create a short-lived `Partner-Upgrade-Experience-API`.

10. This API receives a SOAP request, transforms it into a standard JSON request, and calls the reusable `Customer-API`'s `PATCH` endpoint. This meets the partner's unique need without disrupting our core architecture.

By adopting this four-layer approach, Lumen can create a clear separation of concerns that drives standardization and reuse through the **Canonical API Layer**, while providing managed flexibility for specific consumer needs through the **Experience API Layer**.

The Canonical Data Model (CDM): The Foundation for API Standardization

In our pursuit of a robust and agile enterprise architecture, we face a fundamental challenge: managing the immense complexity of interrelated

services and configurations. The common pitfall is to chase a single, perfect, all-encompassing data model, which is unrealistic and ultimately stifles innovation.

This document outlines our strategy for the Canonical Data Model (CDM). It is not a proposal for a rigid, monolithic system, but rather a **principle-based framework for creating a system of models**. This approach allows us to build a stable foundation while embracing flexibility and isolating complexity. Our strategy is founded on three core architectural principles:

1. **Bounded Contexts:** We will manage complexity by breaking down our enterprise into logical domains, each with its own specific and consistent model.
2. **The Immutable Core + Optional Extensions:** We will define a stable, non-negotiable core for each data entity and provide flexibility through well-defined, optional extensions.
3. **Standardized & Extensible by Design:** All models will be defined using a common, machine-readable format that is inherently designed to be extended over time without breaking its foundation.

Adopting this approach is mandatory for achieving the goals of our Strategic Interface Layer (SIL) and Partner Interface Layer (PIL). It is the key to delivering standardization, agility, and true decoupling.

Example Bounded Contexts

Defining clear bounded contexts is crucial for managing complexity across Lumen's vast product and network portfolio. Here are some logical domains:

1. Product Catalog & Quoting Context

- **Responsibility:** Manages the commercial definitions of Lumen's products. This context is concerned with what can be sold, where it can be sold, and at what price.
- **Example Data:** Product SKUs, pricing rules, service level agreements (SLAs), and serviceability data (which addresses are on-net). It defines a "service" from a sales perspective.

---

2. Order Management & Orchestration Context

- **Responsibility:** Governs the end-to-end lifecycle of a customer order from submission to completion. It tracks the status of an order as it moves through various fulfillment systems.
- **Example Data:** Order ID, customer details, list of services on the order, and high-level order statuses (e.g., `In Progress`, `Completed`, `Cancelled`).

---

3. Customer & Site Management Context

- **Responsibility:** Acts as the master source for all customer account information and their physical locations (sites).
- **Example Data:** Customer legal name, account numbers, contact information, and site addresses (CLLI codes, latitude/longitude, site contacts).

---

4. Service Provisioning & Activation Context

- **Responsibility:** This is a large context often broken down further. It's concerned with the technical configuration of a service on the network.

- **Access/UNI Sub-Context:** Manages the customer-facing interface. Data includes port speed, encapsulation type and VLAN IDs. This is the physical or virtual handoff to the customer.

- **Metro & Transport Sub-Context:** Manages the underlying network paths within and between cities. Data includes MPLS LSPs, segment routing paths, and optical circuit details.
- **Edge & Peering Sub-Context:** Manages how customer traffic gets to the internet or other VPN sites. Data includes BGP sessions, route-maps, and VRF/GRT definitions.
- **Service Layer Sub-Context:** Defines the end-to-end service construct itself. For an L3VPN, this context would orchestrate elements from the Access and Edge contexts to build the complete virtual private network.

---

5. Network & Service Assurance Context

- **Responsibility:** Monitors the health of the network and services post-activation. It manages fault detection, performance monitoring, and trouble ticketing.
- **Example Data:** Circuit IDs, alarm statuses, performance metrics (latency, jitter), and trouble ticket numbers. A "service" in this context is something to be monitored.

---

6. Billing & Invoicing Context

- **Responsibility:** Manages the financial aspects of a customer's service. It's concerned with monthly recurring charges (MRCs), usage-based charges, and generating invoices.
- **Example Data:** Billing account numbers, charge codes, and invoice details.

What is the Canonical Data Model (CDM)?

The Canonical Data Model (CDM) is the single, unified, and consistent data structure for our key business entities (e.g., Customer, Order, Service, Port) across the enterprise. It acts as the universal language for all our APIs.

In the context of our new architecture—the Strategic Interface Layer (SIL) and the Partner Interface Layer (PIL)—the CDM is not just beneficial, it is **absolutely mandatory** for achieving our goals of standardization, agility, and decoupling.

Why the CDM is Critical in the PIL and SIL Context

The CDM's primary role is to enforce the principle of **decoupling**—shielding consuming applications from the technical complexity and volatility of backend systems.

Strategic Interface Layer (SIL)	Partner Interface Layer (PIL)
Mission: Provide clients (internal and external) with a stable, predictable, and consistent view of our business services.	**Mission:** Protect our internal services from the complexity and constant change of integrating with external third-party vendors.
How the CDM Helps: The SIL uses the CDM as its public contract. All data, regardless of its source system, is translated into the standard CDM format before being exposed through the SIL.	**How the CDM Helps:** The PIL uses the CDM as its internal standard. It translates vendor-specific data models into our internal CDM, ensuring our core services only ever have to understand one data language.

Benefits for the Strategic Interface Layer (SIL)

Benefit	Why it Matters
Consistent Client Contract	The CDM ensures that all clients (mobile, web, partners) receive the same field names, formats, and structures for an entity like 'Customer', regardless of which backend system is supplying the data. This eliminates ambiguity and reduces integration costs for consumers.
Simplified Orchestration	When the SIL orchestrates a complex business process, data retrieved from one system (e.g., Billing) is immediately recognizable and usable when passed to another system (e.g., Provisioning). The CDM removes the need for multiple, complex data translations within the SIL itself.
Accelerated Migration	We can replace a legacy backend system with a new strategic one without changing the client-facing API contract. The old system was translated to the CDM, the new system will natively speak the CDM. This gives us immense technical and business flexibility.

Benefits for the Partner Interface Layer (PIL)

Benefit	Why it Matters
Internal Decoupling	Internal services only ever see and use our Canonical Model for a concept like a 'Port'. They are entirely shielded from the specific,

Benefit	Why it Matters
	often messy, data formats (XML, JSON, SOAP) required by partners like AT&T or Verizon.
Plug-and-Play Partner Integration	Partner Adaptors become simple two-way translators: **Our CDM ⇄ Partner Format**. This means integrating a new partner (e.g., Starlink) is fast, requiring only a new adaptor, with **zero code change** to the PIL or our internal services.
Vendor Neutrality	Business rules for partner selection (managed by the PIL) can be based on standardized CDM attributes (e.g., `portType`, `locationID`), not vendor-specific identifiers. This makes switching partners a business decision, not a major technical project.

Governance and Ownership

This is a partnership between Product and Architecture.

Area	Product Team Responsibility	Architecture Team Responsibility
Definition	Define the **business fields, naming conventions, data types, and business rules** for their domain's model. They are the ultimate authority on what a "Customer" or "Order" means to the business.	Define the **technical format** (e.g., JSON Schema, OpenAPI), **security policies** (e.g., Field-Level Authorization), and overall **versioning strategy**. They provide the guardrails and tools.
Version Control	**Drive the roadmap** for the next version of the model, communicating changes clearly to all consumers and stakeholders. They manage the "product backlog" for the data model.	**Enforce Semantic Versioning** (e.g., v1.0.0), ensuring no breaking changes are introduced without proper notification and deprecation plans. They act as the release managers for the technical standard.
Enforcement	Ensure all **new strategic backends** created by the domain team natively expose	Mandate that all **Adaptors** (both SIL and PIL) perform the necessary translation to the CDM before data is

Area	Product Team Responsibility	Architecture Team Responsibility
	the CDM. Advocate for the CDM in all new development.	passed to core services. They ensure compliance at the architectural level.

The CDM is a Product, Not a Project

To succeed, we must treat our Canonical Data Models as long-lived, strategic assets.

A. The CDM as an Internal Product

A project has a start and an end date. A product has a lifecycle, a roadmap, and dedicated owners who seek to maximize its value for its customers.

- **Your Consumers are Customers:** The "customers" of your domain's CDM are all the other development teams (internal and external) who will use it. Your goal is to make their job easier by providing a clear, stable, and well-documented data contract.
- **Create a Roadmap:** What new attributes will be needed in the next 6-12 months? Are there upcoming business initiatives that will require changes to the `Order` or `Service` entity? Plan for this evolution and communicate it transparently.
- **Prioritize and Justify Changes:** Every change to the CDM has a ripple effect. Changes must be justified by clear business value, not just technical preference. Use a backlog to manage and prioritize requested changes from consuming teams.

B. The Strategic Framework: Domain-Driven Design (DDD)

The principle of "Context Boundaries" mentioned earlier comes directly from Domain-Driven Design (DDD), a crucial framework for avoiding common CDM pitfalls. DDD helps us manage complexity by acknowledging that a single model cannot effectively serve the entire business.

- **Bounded Contexts are Your Friend:** The business is composed of different "Bounded Contexts" (e.g., Sales, Support, Billing), each with its own specific language and model. A "Customer" in the Sales

context (with attributes like `leadScore`) is different from a "Customer" in the Billing context (with attributes like `paymentTerms`).
- **The CDM is the Shared Kernel:** Instead of a universal "God Model," the CDM should represent the **Shared Kernel**—the small, stable subset of the model that different contexts must agree on. The `customerID` and `legalName` are canonical and shared. The Sales team's `leadScore` is not; it belongs only within their Bounded Context.
- **Anti-Corruption Layers:** When contexts interact, DDD uses an "Anti-Corruption Layer" to translate between models. Our SIL and PIL adaptors are perfect examples of this. They protect our core services from the specific, and often messy, data models of other systems.

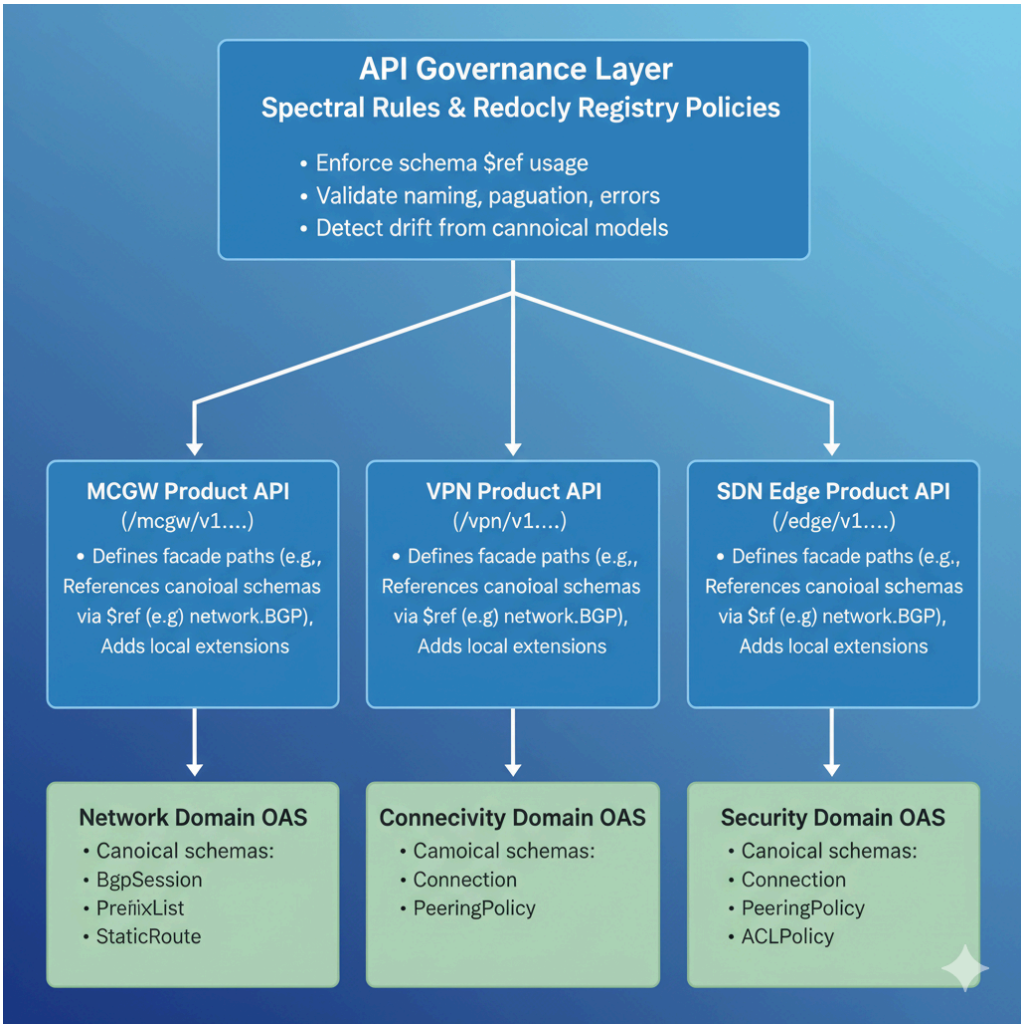
Canonical Model Creation Workflow

Introduction

This document provides the official blueprint and workflow for creating an API Canonical Model at Lumen. Its purpose is to guide architects and teams through a strategic, collaborative process that ensures every canonical model is a consistent, reusable, and valuable enterprise asset.

Following this workflow is essential for implementing our core architectural principles of decoupling, consistency, and agility. It transforms the creation of a model from a siloed technical task into a strategic design process that aligns with our business domains.

As illustrated in the diagram below, this workflow produces reusable **Domain Models** (the foundation), which are consumed by product-facing **Facade APIs** (the experience layer), and are all enforced by an automated **Governance Layer** (the contract).



Guiding Principles

Before starting the workflow, internalize these core principles that govern our approach:

- **Model the Domain, Not the System:** The model must represent the business reality (a "Customer," a "Service"), not the database tables or technical quirks of a legacy backend system.
- **Embrace Bounded Contexts:** Do not attempt to create a single model for the entire enterprise. Acknowledge that a "Port" in the Access domain has different attributes than a "Port" in the Assurance domain. Design models that are specific and unambiguous within their logical boundary.

- **Design for Stability and Extension:** Identify the stable, **Immutable Core** of a model and protect it. Allow for flexibility and future growth through well-defined **Optional Extensions**.
- **Treat the Model as a Product:** A canonical model is a long-lived asset with other development teams as its customers. It requires a product mindset, including dedicated ownership, a roadmap, and a clear versioning strategy.

The Workflow

Phase 1: Discovery and Scoping 🗺️

In this phase, you define the problem space and establish the boundaries for your model.

1. **Define the Business Capability:** Clearly articulate the business process the model will support. Start with the "why."
2. *Example: "Provision a Last Mile Cloud Connect service" or "Retrieve a customer's billing history."*
3. **Establish the Bounded Context:** Work with product and enterprise architects to draw the conceptual boundary around the domain. Determine what entities and attributes are inside this context and what belongs to others. This is the most critical step for managing complexity.
4. **Identify Domain Experts and Stakeholders:** Identify the key people who have authoritative knowledge of this domain. This includes Product Managers, business stakeholders, and senior engineers from the relevant teams. Modeling cannot happen in isolation.

Phase 2: Collaborative Model Design 🧱

This is the core workshop phase where the model takes shape through collaboration.

1. **Host Design Workshops:** Bring the identified domain experts together. Use a whiteboard or digital collaboration tool to visually

map out the entities and their relationships within the bounded context.

2. **Identify Core Entities:** Within the bounded context, identify the key "nouns."
3. *Example: For a Network domain, this might be* `Port`, `Circuit`, `LogicalDevice`, *and* `BgpSession`.
4. **Define the Immutable Core:** For each entity, define its essential, non-negotiable attributes that are stable and unlikely to change.
5. *Example: For a* `Port` *model, this would be its* `portID`, `siteID`, *and* `physicalInterfaceType`.
6. **Define Optional Extensions and Relationships:** Flesh out the model with attributes that provide flexibility or may not always be present. Define the relationships between entities.
7. *Example: A* `Circuit` *is composed of two* `Ports`.

Phase 3: Formalization and Review

This phase turns the conceptual model into a formal, ratified asset.

1. **Draft the Formal Specification:** Translate the workshop outputs into a formal, machine-readable contract using OpenAPI Specification (v3.x) and standalone JSON Schema files. These schemas will form the **Domain OAS** (e.g., Network, Connectivity) shown in Figure 1.
2. **Conduct Architectural Review:** Submit the draft specification for a formal review with the API Governance Guild. The model is checked for adherence to enterprise standards using automated tools like **Spectral** alongside peer review.
3. **Ratify and Version the Model:** Once approved, the model is formally ratified. It is assigned a version number (e.g., v1.0.0) following Semantic Versioning (SemVer) and becomes the official standard for that entity within its bounded context.

Phase 4: Publication, Implementation, and Governance 🚀

The final phase involves publishing the model and managing its lifecycle within our architecture.

1. **Publish to the Domain OAS Registry:** The ratified model specification is published to the official Lumen Schema Registry (a dedicated Git repository). This registry contains the versioned **Domain OAS** files (the green boxes in Figure 1) and is the single source of truth for all teams.
2. **Implement via Product API Facades:** The canonical models are consumed by the various **Product APIs** (e.g., MCGW, VPN), which act as facades. As shown in the middle layer of Figure 1, these APIs use `$ref` to reference the canonical schemas from the Domain OAS registry and can add their own local, product-specific extensions.
3. **Enforce via the Governance Layer:** The ratified model is a living asset. The **API Governance Layer** (the top box in Figure 1) uses tools like Spectral and the Redocly Registry to automatically validate the Product APIs. This ensures they are correctly implementing the canonical models, enforces consistent naming, and detects any "schema drift," protecting the integrity of our API ecosystem. Any proposed changes to a canonical model must follow this governance process before a new version is released.

Principles of Cloud-First API Design

These principles are the strategic solution to the very problems our customers and internal teams face. They provide the "North Star" that justifies our tactical API Style Guide.

How to Use These Principles: A Guide for Stakeholders

This section translates the high-level principles into actionable guidance for different roles.

For Product Managers

Your Role: Define the "What" and the "Why" You are the voice of the customer and the business. These principles empower you to define products, not just features.

- **Define Workflow Resources, Not Verbs:** When writing user stories, define the outcome the customer wants, not the steps they take. Your requirement should be "A customer can change bandwidth with a single API call" (**Principle 1 & 2**). This forces the creation of a simple orchestration façade.
- **Make Monetization a Requirement:** Every new API feature **MUST** have defined billing and entitlement logic. You must specify the `usageType` and `meteringUnit` as part of the feature definition (**Principle 5**).
- **Prioritize a "No-Ticket" Experience:** Your product vision **MUST** be geared toward 100% API-driven self-service. Any requirement for manual intervention or a support ticket is a design failure (**Principle 6**).
- **Own the API Metrics:** You are responsible for the API's success KPIs. You **MUST** define and track metrics for adoption, latency, and error rates to drive your roadmap and prove business value (**Principle 13**).

For Developers & Engineers

Your Role: Implement the "How" You build the reliable, consistent, and scalable platform. These principles are your blueprint for implementation.

- **Implement the Asynchronous Pattern:** For any operation that takes more than 2 seconds (e.g., provisioning), you **MUST** implement the async request-reply pattern. Return a `202 Accepted` and a `Location` header pointing to a job/operation resource (**Principle 3**). Do not build long-polling, synchronous endpoints.
- **Enforce Standards in Your Pipeline:** You **MUST** integrate the provided Spectral linting rules into your CI pipeline. A build that violates the API Style Guide (naming, structure, etc.) **MUST** fail (**Principle 11**).
- **Use the Standard Patterns:** Do not invent your own error models, pagination, or authentication flows. You **MUST** use the platform-

standard `problem+json` for errors, `Link` headers for pagination, and OAuth 2.0 for security (**Principle 12 & 8**).
- **Abstract Internal Complexity:** When building a façade, you are responsible for hiding the "identifier hell." Your implementation **MUST** handle the mapping between the single canonical API ID and all necessary internal system IDs.

For Architects & Platform Leadership

Your Role: Enforce the Platform Vision You are the guardians of platform cohesion, long-term strategy, and governance.

- **Own the Canonical Taxonomy:** You are responsible for defining and enforcing the uniform naming and versioning strategy (`/{product}/v{version}/{resource}`). You **MUST** reject designs that create URI conflicts or inconsistent paths (**Principle 4 & 7**).
- **Champion Automation:** You **MUST** provide and maintain the automated governance tools (e.g., the master Spectral ruleset) that enable developers to comply with the standards at scale (**Principle 11**).
- **Design for the Ecosystem:** Your architectural reviews **MUST** ensure that new APIs are designed for federation and extensibility. A design that only works for a single use case and cannot support partners is a strategic failure (**Principle 10**).
- **Maintain Platform Trust:** You are responsible for enforcing policies on backwards compatibility and graceful evolution. You **MUST** ensure that teams follow proper deprecation procedures to avoid breaking client integrations (**Principle 15**).

1. Resource-Oriented Interface (The "How")

This is our core design philosophy. It means we model **all** interactions—from simple data management to complex, multi-step workflows—as "resources" (nouns) that a client can interact with.

This principle **explicitly forbids** exposing internal verbs or process steps (RPC-style).

It's crucial to understand that "resource-oriented" is **not** just for simple CRUD (Create, Read, Update, Delete). It is the *interface style* we use for two distinct patterns:

1. **CRUD-Style Resources:** For simple, state-based entities (e.g., `/ports`, `/users`).
2. **Workflow-Style Resources:** For complex, outcome-based actions (e.g., `/orders`, `/connections`).

Why: A consistent, resource-based interface enables self-service, idempotency, and alignment with all modern cloud provider conventions (AWS/GCP/Azure).

---

2. Desired-State & Outcome-Based (The "What")

This is the *message* you send to a resource. It's the **opposite of an imperative command**.

Instead of a step-by-step *process*, the client sends a *declarative payload* describing the **desired end state**. Our orchestration façade is then responsible for making the "actual state" match that "desired state."

Example: The Core Distinction

This is the most important concept. We **abstract** our internal process into a single, simple, resource-based API call.

❌ Anti-Pattern: The RPC/Verb Process

The client must know our internal steps. **(FORBIDDEN)**

```
POST /api/checkPrice
POST /api/createQuote
POST /api/submitOrder
```

✅ **The "Workflow Resource" (Our Standard)**

The client interacts with **one resource** (`/v1/connections`) and provides a **desired-state** body. Our façade hides the *entire* price/quote/order process.

```
POST /v1/connections
{
  "bandwidth": "10G",
  "src_port": "port-abc",
  "dst_port": "port-xyz"
}
```

---

3. Asynchronous and Operation-Tracked

What it means: Long-running provisioning and teardown tasks (like the `POST /v1/connections` example above) **MUST** return an operation resource immediately.

Example:

1. `POST /fabric/v1/connections` → `202 Accepted`
2. **Response Header:** `Location: /fabric/v1/operations/12345`

Why: Mirrors GCP/Azure async operation patterns; avoids client timeouts and supports distributed job orchestration.

---

4. Product-Scoped Versioning

What it means: Versioning sits within the product namespace, never at the cross-domain layer.

Example:

- `/fabric/v1/telemetry/events`
- `/mcgw/v1/connections`

Why: Each product can evolve independently; prevents version collisions across domains.

5. Entitlement-Aware and Monetizable

What it means: Every resource request validates the caller's entitlements and records usage events for billing.

Example Metadata:

```
{
  "entitlementId": "ENT-123",
  "usageType": "bandwidth-hours",
  "meteringUnit": "GB"
}
```

Why: Supports SaaS-like pay-per-use and quota enforcement similar to cloud hyperscalers.

---

6. Self-Service and Declarative Lifecycle

What it means: Customers and partners can onboard, provision, and manage services fully via API or IaC templates—no ticketing.

Why: Enables elasticity, automation, and integration into DevOps pipelines.

---

7. Uniform Taxonomy and Namespacing

What it means: Product namespaces and resource naming follow a canonical structure: `/{product}/v{version}/{resource}/{subresource}`

Why: Drives predictability across APIs, allowing governance automation (Spectral rules, gateway routing) and consistent developer experience.

---

8. Strong Identity and Policy Integration

What it means: OAuth 2.0/OIDC as baseline; JWT claims carry entitlements, scopes, and tenant context.

Example Scopes: `fabric:read`, `mcgw:modify`, `wholesale:metrics`

Why: Enables fine-grained authorization and uniform policy enforcement across products.

---

9. Observable and Telemetry-Driven

What it means: APIs emit standardized metrics and events: latency, success rate, error codes, SLO violations, and usage.

Example:

```
POST /telemetry/events
{
  "api": "/fabric/v1/ports",
  "latencyMs": 245,
  "status": 200
}
```

Why: Provides the foundation for automated SLA tracking, billing accuracy, and health dashboards.

---

10. Extensible Through Federation and Partner APIs

What it means: Partner and wholesale APIs follow the same design principles, enabling multi-tenant federation.

Example:

- `/wholesale/fabric/v1/connections`
- `/partner/mcgw/v1/orders`

Why: Supports ecosystem integration (hyperscalers, value-added resellers) without duplicating backend logic.

---

11. Automation and Governance-Ready

What it means: Design patterns are machine-verifiable. URI structure, version placement, naming, and required metadata are enforced by Spectral linting and CI pipelines.

Why: Guarantees compliance at scale, reduces manual reviews, and allows self-serve publication to catalogs (e.g., Backstage).

---

12. Developer Experience Consistency

What it means: Consistent error models, pagination, and async patterns across all APIs; SDKs and documentation generated from OAS.

Example Error:

```
{
  "errorCode": "FABRIC-404",
  "message": "Port not found",
  "correlationId": "abc-123"
}
```

Why: Developers can reuse tooling and templates across products, just like AWS SDK parity across services.

---

13. Outcome Metrics and Continuous Improvement

What it means: APIs publish KPIs on adoption, latency, and SLA compliance to a governance dashboard.

Why: Drives data-driven evolution and productization—every API is treated as a measurable SaaS asset.

---

14. Compliance, Security, and Data Residency by Design

What it means: Security and compliance are embedded through consistent header policies, trace IDs, and encryption standards, not added after design.

Why: Supports multi-jurisdictional and regulated workloads (finance, healthcare).

---

15. Backwards Compatibility and Graceful Evolution

What it means: APIs evolve additively; breaking changes require a new major version. Deprecated endpoints remain discoverable with `Deprecation` headers and clear timelines.

Why: Ensures clients can safely migrate; matches cloud provider deprecation best practices.

Reference Architectures (The "Playbooks")

- Proposed Sell Reference Architecture: : A Strategic Interface Layer Architecture
- Proposed "Buy" Reference Architecture for Partner Integration at Lumen: The PIL Model
- API Publishing Playbook: Public vs. Internal Deployment Workflow

Proposed Sell Reference Architecture: : A Strategic Interface Layer Architecture

The Challenge

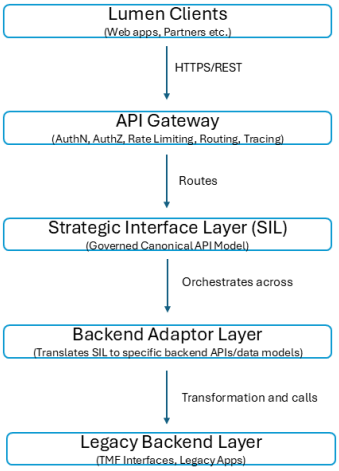
Lumen, a company with a history of frequent mergers and acquisitions, faces a common architectural dilemma: a fragmented backend landscape. Decades of integrating disparate systems have resulted in a lack of standardization, multiple APIs serving the same concepts with inconsistent data models, and a significant impediment to agility and unified customer experiences. The core challenge is to standardize Lumen's digital platform without requiring an immediate, disruptive "big bang" overhaul of all legacy systems

The Proposed Solution: A Strategic Interface Layer (SIL)

To address these challenges, we propose an architecture centered around a **Strategic Interface Layer (SIL)**. This layer acts as a crucial abstraction and orchestration point, designed to provide immediate standardization for clients while enabling a phased, long-term migration of legacy backends.

The architecture revolves around two key principles:

- **Client Abstraction:** Shielding client applications from the inherent complexity and inconsistencies of Lumen's diverse backend systems.
- **Phased Migration:** Allowing legacy backends to be refactored or replaced incrementally, without impacting the client-facing APIs.



Key Components:

1. **Lumen Clients:** Front-end applications (web, mobile, partner integrations) that consume Lumen's digital services. Their primary need is a stable, consistent, and well-documented API.
2. **API Gateway :** The external entry point for all client requests. After enforcing standard gateway functions (security, rate limiting, logging), this component will route all request to SIL Layer

3. **Strategic Interface Layer (SIL):** The is “the face of Lumen APIs”. It is key interface, based off a canonical, well governed API model exposed to clients. This is the orchestration and standardization layer. Its responsibilities include:

4. **Standardized Contract:** Exposing a unified, domain-driven API contract for core Lumen concepts (e.g., Customer, Order, Service).

5. **Lean Orchestration:** For requests that involve multiple legacy backends or complex workflows not yet encapsulated by a single strategic backend.
6. **Data Model Transformation:** Ensuring that data returned to clients conforms to the strategic data model, regardless of the underlying backend's format.

7. **Backward Compatibility:** Managing API versions to ensure existing clients are not broken during backend transitions.
8. **Backend Adapter Layer:** A sub-component of the SIL (or a set of distinct services called by the SIL) responsible for translating the standardized SIL requests into the specific API calls and data formats required by individual legacy backends. Each adapter is purpose-built for a particular legacy system.
9. **Legacy Backends:** The existing, unstandardized systems resulting from Lumen's M&A history. These systems will eventually be deprecated, refactored, or replaced.

Operational Flow

- A **Lumen Client** makes a request to the **API Gateway**.
- The **API Gateway / Smart Router** routes the request to the **Strategic Interface Layer (SIL)**.
- The **SIL** then performs any necessary orchestration, calls the appropriate **Backend Adapter** for legacy systems.
- Responses are transformed by the SIL (if originating from legacy systems) to adhere to the strategic data model before being sent back to the client via the Gateway.

Interaction Principles (The Rules of Engagement)

This defines how the components **must** interact to ensure decoupling and maintain clear architectural boundaries.

- **External Consumers:** All external clients **must** only ever call the APIs exposed at the API Gateway. They are forbidden from calling any internal service directly.
- **API Gateway:** The Gateway after performing its function, routes the requests to SIL
- **Decoupling:** The public-facing API contract **must** remain stable, even when a backend system is migrated from the "Legacy Path" to the "Modern Path." This transition must be completely transparent to the external consumer.

Governance & Non-Functional Requirements (The Standards)

This ensures that every implementation is secure, consistent, and observable.

- **Security:** All endpoints exposed at the API Gateway **must** be secured using the enterprise standard (e.g., OAuth 2.0).
- **Observability:** All transactions **must** include a correlation ID that is passed through every layer for end-to-end tracing.
- **API Standards:** All public-facing APIs **must** adhere to the official Lumen API Style Guide.

The Colocation Principle: Context

The "Sell" Side Reference Architecture is designed as a layered, distributed system. Effective performance of this architecture relies on low-latency communication between its internal components, particularly between the Strategic Interface Layer (SIL) and its Backend Adapters.

However, Lumen's current enterprise constraint of a single, centralized API Gateway in GCP creates a "network hairpin" problem, introducing prohibitive latency (~130ms+ per hop) for any service-to-service communication that crosses cloud boundaries. While long-term solutions like a Federated Gateway or a Service Mesh are the strategic goal, they are not immediately available.

Therefore, to ensure the performance and viability of all near-term SIL implementations, the **Colocation Principle is adopted as the mandatory, pragmatic interim deployment pattern**

The Colocation Principle: Mandated Rules

Given the immediate need for a workable solution, the following rules are mandated for all SIL implementations until a federated gateway or service mesh is available.

- **SIL and Adapters as a Single Deployment Unit:** The SIL and its corresponding Backend Adapters **must** be deployed in the same cloud, same VPC, and ideally the same cluster. They should be treated

as a single, cohesive deployment unit to guarantee low-latency communication.

- **Intra-Cloud Orchestration Only:** The SIL is strictly forbidden from performing real-time, synchronous orchestration that calls services or adapters across different cloud environments (e.g., a SIL in AWS cannot call an adapter in GCP). All real-time orchestration must be contained within a single cloud boundary.
- **Separate Logical Deployments Required:** While they must be colocated, the SIL and its Backend Adapters **must** remain logically separate deployments (e.g., distinct microservices or containers). Bundling adapters as a library (e.g., a JAR file) inside the SIL is considered an anti-pattern, as it creates a monolith and prevents independent deployment, scaling, and maintenance.
- **Multi-Cloud Aggregation via Asynchronous Data:** For any use case that requires a unified view of data from multiple clouds, the approved pattern is **asynchronous data aggregation**, not real-time orchestration. The SIL **must** perform a single, low-latency query against a pre-aggregated data store (e.g., a Data Hub or Lakehouse) rather than performing a live, cross-cloud "fan-out" to multiple services.

Proposed "Buy" Reference Architecture for Partner Integration at Lumen: The PIL Model

This document outlines the proposed **"Buy" Reference Architecture**—the strategy for integrating external partner capabilities (e.g., off-net ports, connectivity from AT&T, Verizon, Starlink) into Lumen's digital platform. This architecture is designed for **security, governance, and decoupling** to ensure that Lumen's internal services are protected from the complexity and potential volatility of third-party APIs.

The Need for Standardization

Just as the Strategic Interface Layer (SIL) standardizes Lumen's *selling* capabilities, a robust "Buy" architecture is needed to standardize *partner sourcing*. When buying capabilities, the goal is to abstract the partner's

technology and data model from your internal systems. Without it, internal Lumen services would be tightly coupled to specific partner technologies, leading to:

- **High Maintenance Cost:** Any change to a partner's API would require updating every Lumen service that calls it.
- **Lack of Agility:** Adding a new partner (e.g., Starlink) would require extensive code changes across the organization.
- **Security Risk:** Scattered management of external partner credentials and security policies.

The Core Principle: A Playbook, Not a Single Solution

This Reference Architecture is not a single, monolithic system. It is a strategic **pattern** that will be implemented for each distinct "Buy" side business domain (e.g., an "OffNet Connectivity PIL," a "Cloud Marketplace PIL").

The Reference Architecture is prescriptive about the **principles and components** (the "what" and "why") but flexible about the **implementation specifics** (the "how"). It provides strong guardrails while empowering delivery teams to make the right architectural choices for their specific business problem.

Structural Components (The "What")

This defines the non-negotiable building blocks that **must** be used in every implementation to ensure structural consistency across the enterprise.

Component	Primary Role	Rationale
Partner Interface Layer (PIL)	The internal-facing facade for a specific business domain.	Centralizes domain-specific business logic (e.g., partner selection) and exposes a single, clean API for internal consumers.
Partner Adaptors	A dedicated microservice for each individual partner.	Ensures technical isolation. If a partner's API changes, only its single adaptor is updated,

Component	Primary Role	Rationale
		protecting all other components.
External Partner Gateway	The single, governed exit point for all outbound traffic to partners.	Critical for governance. Enforces uniform security, rate limiting, and centralized logging, preventing unmanaged service-to-internet sprawl.
Webhook Ingestion Layer	The single, governed entry point for all inbound asynchronous events from partners.	Provides a secure and resilient "front door" for partner webhooks, authenticating and queuing events before they are processed.

Interaction Principles (The Rules of Engagement)

This defines how the components **must** interact to ensure decoupling and maintain clear architectural boundaries.

- **Internal Consumers:** Internal services **must** only ever call the PIL's interface. They are forbidden from calling a Partner Adaptor directly.
- **The PIL:** The PIL **must** expose a clean, domain-specific API that hides the underlying complexity of partner orchestration and selection.
- **Partner Adaptors:** An Adaptor's responsibility is limited to the technical concerns of a single partner (authentication, data transformation). It **must not** contain business logic that spans multiple partners.

Governance & Non-Functional Requirements (The Standards)

This ensures that every implementation is secure, consistent, and observable, regardless of the specific use case.

- **Security:** All endpoints exposed by the PIL and the Webhook Ingestion Layer **must** be secured using the enterprise standard (e.g., OAuth 2.0). All outbound traffic from the External Partner Gateway **must** use appropriate transport-layer security (e.g., mTLS).
- **Observability:** All transactions **must** include a correlation ID that is passed through every layer (PIL, Adaptor, Gateway) for end-to-end tracing.
- **Logging:** A standardized logging format **must** be used by all components to enable centralized analysis and alerting.

A Decision Framework for Extensions (The "How")

The Reference Architecture provides guided flexibility. Any team building an implementation **must** consciously make and document their decisions based on this framework.

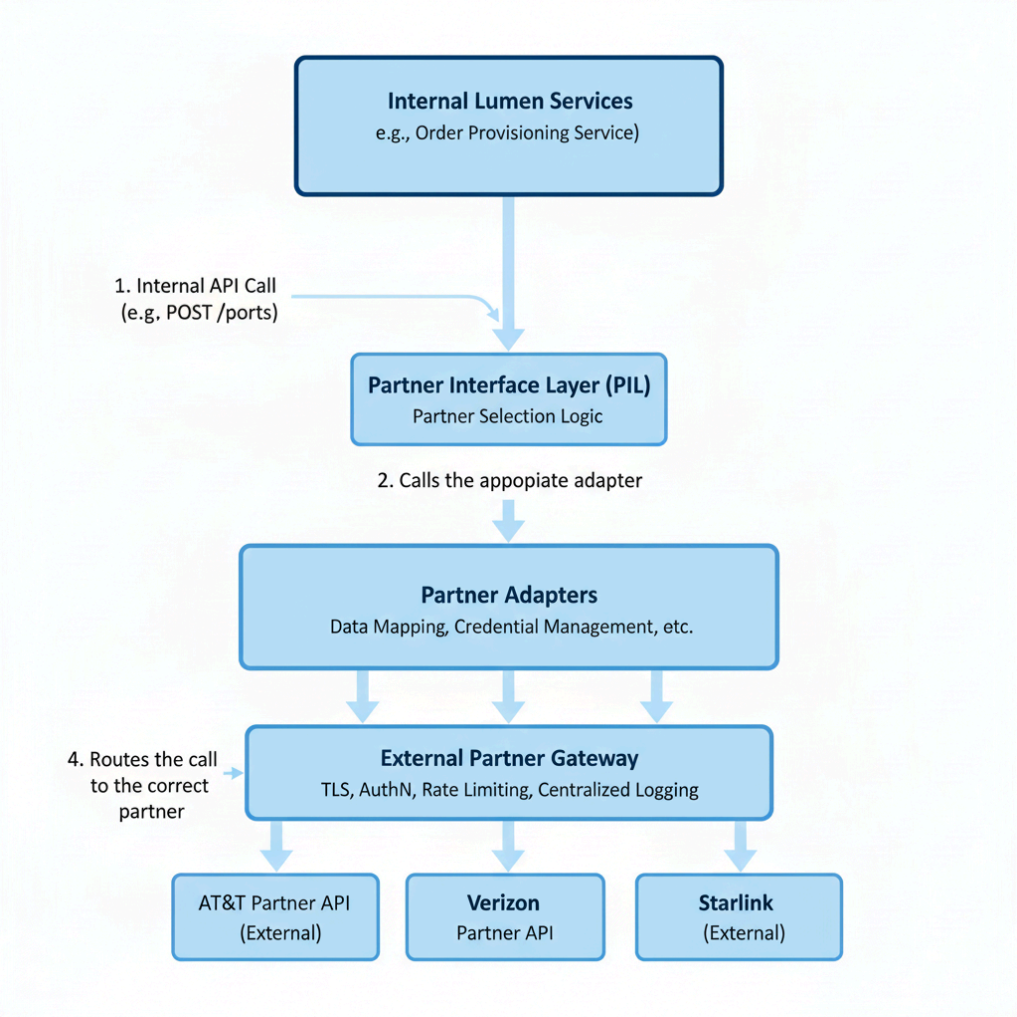
The most critical decision is the state management strategy. The design **must** explicitly document whether it is using a "Stateful" or "Stateless" pattern and provide a clear justification.

- **Question** 1: What **is the Source of Truth?**
- If the **partner** is the absolute source of truth for the resource (like a cloud resource), a **Stateless** pattern is required.
- If **Lumen** is the source of truth for the *transaction* itself (like an OffNet order), a **Stateful** pattern is required.
- **Question 2: What is the Lifecycle of the Transaction?**
- Is it a short-lived, real-time request/response? -> **Stateless**.
- Is it a long-running, asynchronous process that could take days or weeks? -> **Stateful**.
- **Question 3: How are Out-of-Band Changes Handled?**
- The design **must** address how it will handle the "tricky case" where a resource's state is changed on the partner's side without a webhook

(e.g., via their portal). A Stateful pattern, for example, must include a state reconciliation mechanism

The Secure Outbound Flow

All requests for partner services must follow a stringent, controlled path to ensure security and maintainability.



Internal Lumen Service → PIL → Adaptor → External Partner Gateway → 3rd Party Partner API

Step	Component	Action
Internal Call	**Lumen Service**	Makes a standardized request to the PIL (e.g., "Find

Step	Component	Action
		Port in Dallas").
Business Logic	**PIL**	Applies business rules to select the best partner (e.g., "Verizon is cheapest here") and forwards the request to the corresponding Adaptor.
Transformation/Vendor specific AuthN	**Adaptor**	Transforms the standardized PIL request into the exact format and data model required by the chosen partner's API. Also responsible for obtaining/caching the OAuth token for a specific external vendor.
Routing	**External Partner Gateway**	Manages mTLS/TLS, applies rate limits, and centrally logs the external transaction.
External Execution	**3rd Party Partner**	Receives and executes the request.

Key Benefits

This layered "Buy" architecture solves the core enterprise challenges of partner integration:

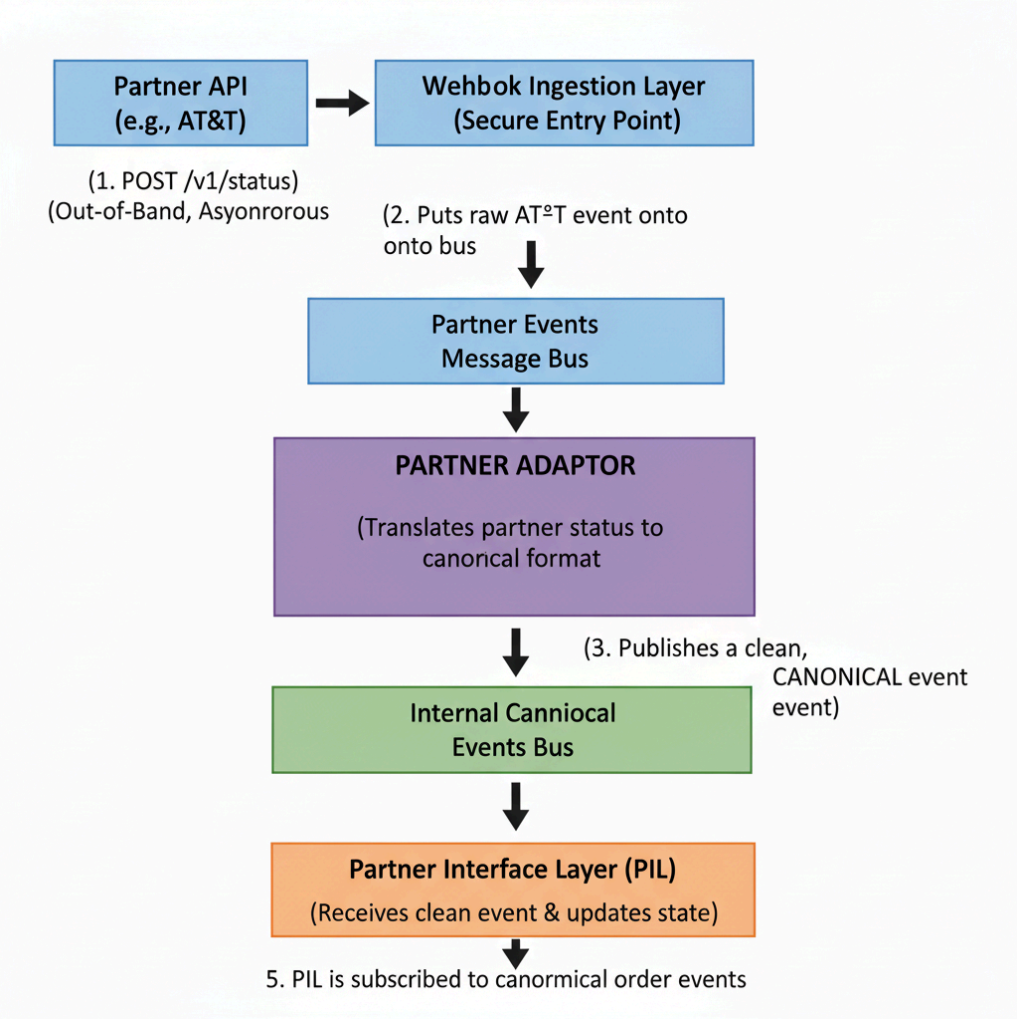
- **True Decoupling:** Lumen's internal systems only ever talk to the **PIL**'s standardized contract. They are completely ignorant of partner technology, meaning replacing 3rd party vendor with a new provider has zero impact on core Lumen code.
- **Centralized Security:** By forcing all outbound calls through the **External Partner Gateway**, Lumen ensures all traffic is uniformly secured and adheres to external partner agreements and rate limits.
- **Agile Onboarding:** Integrating a new partner is a focused project—it requires only building a new **Partner Adaptor**. The existing PIL, internal services, and Gateway remain untouched.
- **Resilience:** The **PIL** can implement failover logic. If the vendor Adaptor reports an API error, the PIL can automatically retry if the error is recoverable.

Inbound Flow - Handling Asynchronous Partner Events (Webhooks)

For long-running processes like service provisioning, partners provide asynchronous status updates via webhooks. The Reference Architecture mandates a secure, resilient, and decoupled pattern for ingesting and processing these events.

The flow is the mirror image of the outbound request, ensuring that internal systems are shielded from the complexity and proprietary formats of partner events.

1. **Ingestion:** A partner system sends a `POST` request to a single, secure endpoint exposed by the **Webhook Ingestion Layer**. This endpoint is specific to the partner (e.g., `https://webhooks.lumen.com/v1/att`). The payload is in the partner's native format.
2. **Authentication & Queuing:** The Ingestion Layer's only jobs are to authenticate the request (e.g., via an API key or HMAC signature) and immediately place the raw, untranslated event onto a dedicated **Partner Events Message Bus**. This ensures resilience and durability.
3. **Adaptor Translation:** The appropriate **Partner Adaptor** (e.g., the `AT&T Adaptor`) is subscribed to its topic on the message bus. It consumes the raw event and performs the critical translation, converting the partner's proprietary status codes and data fields into the official **Lumen Canonical Model**.
4. **Canonical Event Publication:** The Adaptor publishes a new, clean, **canonical event** (e.g., `lumen.order.status.updated`) onto a separate **Internal Canonical Events Bus**.
5. **Internal Consumption:** The appropriate **PIL** (or other internal service) is subscribed to the internal bus. It receives the clean, canonical event and can reliably take action, such as updating an order's state in its database.



Deployment Mandate: The Colocation Principle for the PIL

1. Context and Rationale

The "Buy" Side Reference Architecture is designed as a layered, distributed system that relies on high-chattiness, low-latency communication between its internal components, particularly between the **Partner Interface Layer (PIL)** and its **Partner Adaptors**. This is especially critical during the "fan-out" process for multi-partner quoting.

However, Lumen's current enterprise constraint of a single, centralized API Gateway in GCP creates a "network hairpin" problem, introducing prohibitive latency (~130ms+ per hop) for any service-to-service communication that crosses cloud boundaries. While long-term solutions like a Federated Gateway or a Service Mesh are the strategic goal, they are not immediately available.

Therefore, to ensure the performance and viability of all near-term PIL implementations, the **Colocation Principle is adopted as the mandatory, pragmatic interim deployment pattern.**

2. The Colocation Principle: Mandated Rules

Given the immediate need for a workable solution, the following rules are mandated for all PIL implementations until a federated gateway or service mesh is available.

- **PIL and Adapters as a Single Deployment Unit:** The PIL and the Partner Adapters it needs to orchestrate for a specific domain **must** be deployed in the same cloud, same VPC, and ideally the same cluster. They should be treated as a single, cohesive deployment unit to guarantee low-latency communication during quote fan-outs and other internal calls.
- **Intra-Cloud Orchestration Only:** The PIL is strictly forbidden from performing real-time, synchronous orchestration that calls Partner Adapters across different cloud environments (e.g., a PIL in AWS cannot call an adapter in GCP). All real-time orchestration **must** be contained within a single cloud boundary.
- **Separate Logical Deployments Required:** While they must be colocated, the PIL and its Partner Adapters **must** remain logically separate deployments (e.g., distinct microservices or containers). Bundling adapters as a library (e.g., a JAR file) inside the PIL is considered an anti-pattern, as it creates a monolith and prevents independent deployment, scaling, and maintenance.
- **Handling Multi-Cloud Data Requirements:** For any use case requiring a unified view of data from partners whose adaptors reside in different clouds, the primary principle is to **colocate all adaptors required for real-time orchestration**. Since Partner Adaptors are location-agnostic, they should be deployed in the same cloud environment as the PIL that consumes them.

API Publishing Playbook: Public vs. Internal Deployment Workflow

Purpose

This playbook defines the mandatory governance process for all APIs, using the `lifecycleStage` metadata field to track an API's authoritative status. The `main` **branch** serves as the definitive **"Approved for Publish"** source for both Public and Internal APIs.

Stage 1: Technical Governance Approval (The Baseline)

This stage establishes and validates the structural contract of the API and is required for **ALL** APIs. An automated process sets up the feature branch and domain folder structure (e.g., `/domains/<project>`).

1. **Developer Setup & Submission**
2. Developer populates the OAS and the `api-metadata.yaml` files within the pre-created feature branch.
3. Developer opens a PR from the feature branch, targeting the protected `governance-approved` branch.
4. **Lifecycle State Transition: Design** --> **Development**
5. **Pipeline Validation & Review**
6. The Pipeline is triggered to run technical validation against **Lumen Standards** on the OAS and metadata. Architectural review takes place.
7. **Lifecycle State:** Stays at **Development**
8. **Merge to** `governance-approved`
9. The PR is approved by the API Architect and merged.

10. **Lifecycle State Transition: Development** --> **Governance-Approved** (Contract is now locked as technically compliant.)

Publishing Path Fork

Once the contract is technically approved (state: **Governance-Approved**), the workflow splits based on the API's designated audience.

Path A: Public API Publishing (Two-Stage Review)

Public APIs require a second PR for human review of external-facing documentation quality.

1. **Documentation & Metadata Review (Stage 2)**
2. A new PR is created from `governance-approved`, targeting the `main` branch.
3. The Technical Writer reviews, edits the narrative, and modifies **only non-contract** OAS fields (e.g., summary, description, and examples). The Pipeline is triggered to run technical validation against **Lumen Standards** on the OAS and metadata.
4. **Lifecycle State:** Stays at **Governance-Approved**
5. **Merge to `main` & Publish**
6. The PR is merged into the `main` **branch**. The Public Publishing Pipeline is triggered.
7. **Tooling:** Pushes content to **Amazon S3/CloudFront** at `developer.lumen.com`.
8. **Lifecycle State Transition: Governance-Approved** -→ **Published**

Path B: Internal API Publishing (Single-Stage Direct Merge)

Internal APIs skip the human documentation review and use automation to achieve the final state immediately after technical approval.

1. **Automated Merge Trigger**

2. Automation detects the successful merge into the `governance-approved` branch.
3. **Lifecycle State:** Stays at **Governance-Approved**
4. **Merge to `main` & Publish**
5. Automation performs a fast-forward merge from `governance-approved` to the `main` **branch**. The Internal Publishing Pipeline is triggered immediately.
6. **Tooling:** Syncs OAS to the **SwaggerHub** internal workspace.
7. **Lifecycle State Transition: Governance-Approved** --> **Published**

Post-Publishing Lifecycle

Future Action	**Lifecycle State Transition**
Deprecation Decision	**Published** --> **Deprecated**
API Shutdown	**Deprecated** --> **Retired**

Lumen API Style Guide

- API Style Guide Governance Tracker
- OAS Specification Best Practices
- API URI Standard Proposal
- API Pagination Standards
- API Standard: Lumen Problem Details — Minimal Profile (LPDP-Mini v1.0)
- API Standard: Idempotency
- API Standard: Partial Resource Retrieval
- API Standard: Delete Method
- API Standard: HTTP Headers
- API Standard: Caching & Concurrency with ETag
- API Standard: Long-Runing Operations (LRO)
- API Standard: Bulk Data Transfer (Handling Massive Payloads)
- API Standard: Date & Time Naming

- API Standard: GET Method
- API Standard: POST Method
- API Standard: PUT Method
- API Standard: PATCH Method
- API Standard: Batch Operations
- API Standard: Standardized Data Types and Formats
- API Standard: Rest & Resource Design
- API Standard: JSON Payload
- API Versioning Strategy

API Style Guide Governance Tracker

1. Overview

This page tracks the progress of the API Working Group in reviewing, consolidating, and approving the official Lumen API Style Guide. The goal is to create a single, unified set of standards that will be used for all API development and governance.

Status Legend

- 🟩 **Approved:** The standard has been reviewed and approved by the working group.
- 🟧 **Pending:** The standard is under review, has open issues, or is awaiting a formal decision.
- ⬜ **Draft:** The standard exists but has not yet been formally reviewed by the group.

2. Master Governance Tracker

Subject Area	Source	Status	Open Issues / Questions	Notes / Rationale
Principles of Cloud-First API Design	Confluence	🟩 **Approved**		

OAS Specification Best Practices	Confluence	**Approved**	3.1.x (limited support). Current support 3.0.x	Updated support for 3.0.x
API Versioning Strategy	Confluence	**Approved**	`Deprecation`: date format to follow RFC. `Modify status code`	Updated
API Pagination Standards	Confluence	**Approved**	`next/prev` is not possible at this time	Removed
API Standardized Error Responses	Confluence	**Approved**	`traceparent`, `9457` (RFC 9457 Problem Details)	Created RFC 9457 Problem Detail profile for Lumen
API URI Standards and Design Patterns	Confluence	**Approved**		
API Standard: Idempotency	Confluence	**Approved**		
API Standard: Partial Resource Retrieval	Confluence	**Approved**		
API Standard: Delete Method	Confluence	**Approved**		
API Standard: Caching & Concurrency	Confluence	**Approved**		
API Standard: HTTP Headers	Confluence	**Approved**		
API Standard: Date & Time Naming	Confluence	**Approved**		
API Standard: Standardized Data Types	Confluence	**Approved**		
API Standard: Rest &	Confluence	**Approved**		

Resource Design				
API Standard: JSON Payload	Confluence	**Approved**		extracted date standards seperately from json payload
API Standard: GET Method	Confluence	**Approved**		
API Standard: POST Method	Confluence	**Approved**		
API Standard: PUT Method	Confluence	**Approved**		
API Standard: PATCH Method	Confluence	**Approved**		
webhook	(New Item)	**Draft**	`jacob(webhooks - undermetristic), casey (sre)`	Lower priority, not required for base API Styleguide
API Governance Process	Confluence	**Draft**		Lower priority, not required for base API Styleguide
API Standard: Long-Runing Operations (LRO)	Confluence	**Approved**		Lower priority, not required for base API Styleguide
API Standard: Bulk Data Transfer	Confluence	**Approved**		Lower priority, not required for base API Styleguide
API Standard: Batch Operations	Confluence	**Approved**		

Export to Sheets

OAS Specification Best Practices

This document provides a set of standardized guidelines for creating high-quality, maintainable, and developer-friendly OpenAPI specifications. Adhering to these practices ensures consistency and clarity across all our APIs.

OAS Version and Format

Adopt OAS 3.0.3 as the Standard

- Utilize OAS 3.0.3 for all new API specifications to ensure alignment with ApigeeX industry practices.
- Include the correct version declaration at the start of the document:

```
openapi: 3.0.3
```

Use YAML as the Documentation Format

- Employ YAML for writing OAS documents for better readability and maintainability.
- Use a consistent indentation of two spaces to structure the document clearly.

API Metadata (`info` object)

The `info` object is the first thing a developer sees. It must clearly articulate the API's purpose and business value.

- `info.description`: This field **MUST** describe the business capability the API unlocks. It should answer the question, "What problem does this solve for the consumer?" Avoid technical jargon.
- ***Bad***: "An API for managing MCGW connections."
- ***Good***: "Use the Fabric Connections API to programmatically create, manage, and delete high-speed, private connections between your on-premises infrastructure and cloud providers like AWS, Azure, and GCP."

API Documentation Structure (`tags` and `summary`)

These fields control how your API is organized and displayed in documentation tools, making them critical for usability and context.

- `tags`: All operations **MUST** be grouped using `tags`. The tags **MUST** be based on the resource name (e.g., a "Connections" tag for all operations related to connections) to organize the API around its core business objects.
- `summary`: Every operation **MUST** have a `summary`. It must be a short, action-oriented phrase from the user's perspective that clearly states what the endpoint does.
- *Bad*: "GET connections"
- *Good*: "List all active connections"

Operation IDs (`operationId`)

The `operationId` is used by code generation tools to create method names in SDKs. Inconsistent or missing IDs lead to messy, unpredictable code and a poor developer experience.

- **Requirement**: Every operation **MUST** have a unique `operationId`.
- **Convention**: The `operationId` **MUST** follow a consistent naming convention, such as `verbResource` (e.g., `listConnections`, `createConnection`). This ensures the generated code is predictable and intuitive.

Security

Apply security requirements at the individual path or operation level rather than globally. This approach provides precise control and accommodates varying security needs for different endpoints.

```
paths:
  /items:
    get:
      security:
        - OAuth2:
            - read
```

Reusability and Naming

- **Reuse Schemas**: Leverage the `components` section to define and reuse schemas, parameters, and headers. This ensures consistency and makes the specification easier to maintain.
- **Descriptive Naming Conventions**: Use meaningful names for paths, parameters, and other elements, and follow the company's established naming standard.

How to Document Reusable Components

The `components` section is the central repository for all reusable definitions.

Reusable Request Parameters (Query & Path)

Define reusable query and path parameters under

```
components.parameters.
```

Declare Reusable Parameters example

```
components:
  parameters:
    ResourceId:
      name: id
      in: path
      description: The unique identifier for the resource.
      required: true
      schema:
        type: string
        format: uuid
    LimitParam:
      name: limit
```

```
      in: query
      description: The maximum number of items to return per page.
      schema:
        type: integer
        default: 20
```

Reference them in your paths

```
paths:
  /v1/items/{id}:
    parameters:
      - $ref: '#/components/parameters/ResourceId'
```

Reusable Request Headers

Request headers are treated as parameters with “in: header” and are also defined under

```
components.parameters
```

Define the reusable header as a parameter

```
components:
  parameters:
    CorrelationIdHeader:
      name: X-Correlation-ID
      in: header
      description: Tracks a request across multiple services for debugging.
      required: true
      schema:
        type: string
        format: uuid
```

Reference it in your path

```
paths:
  /v1/items:
    post:
      parameters:
        - $ref: '#/components/parameters/CorrelationIdHeader'
```

Reusable Response Headers

Response headers have their own dedicated section for reusability:

```
components.headers
```

Define the reusable response header

```yaml
components:
  headers:
    RateLimitRemaining:
      description: The number of requests left for the current time window.
      schema:
        type: integer
```

Reference it in a response definition

```yaml
paths:
  /v1/items:
    get:
      responses:
        '200':
          description: A successful response.
          headers:
            RateLimit-Remaining:
              $ref: '#/components/headers/RateLimitRemaining'
```

Data Types and Examples

- **Define Reusable Schemas**: Define reusable schemas for complex data types in the `components` section. Clearly specify ***data types, formats, constraints, and examples***.
- **Provide Realistic Examples**:
- Include examples that cover various real-world scenarios (common, edge, and error cases).

- Use the `example` or `examples` keyword within schema definitions for both requests and responses.
- Ensure examples use realistic and relevant data.

OAS `examples` Example

```
paths:
  /v1/items:
    post:
      summary: Create a new item
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Item'
            examples:
              # First example: A simple item with only required fields
              simpleItem:
                summary: A basic item
                description: An example of creating an item with the minimum
                value:
                  name: "Standard Widget"
                  price: 19.99

              # Second example: An item with optional fields included
              itemWithOptionalFields:
                summary: An item with optional data
                description: An example of creating an item that includes op
                value:
                  name: "Premium Widget"
                  price: 29.99
                  description: "A high-quality, durable widget."
                  sku: "PREM-WID-001"
```

Casing Requirements

Schema objects defined in the `components` section **MUST** use **PascalCase** (also known as UpperCamelCase). This convention makes it easy to distinguish schema objects from `snake_case` field names within the specification.

```
# ✅ Good: Uses PascalCase
CustomerOrder:
  type: object
  properties:
    order_id:
      type: string
# ❌ Bad: Uses snake_case
customer_details:
  type: object
```

Best Practices for Naming

Use Singular Nouns

Schema object names **MUST** represent a single instance of that object. Use singular nouns for clarity. The concept of a collection is handled by defining an `array` that references the singular object.

- *Good*: `Customer`, `Order`, `Invoice`
- *Bad*: `Customers`, `OrdersList`

Be Descriptive and Unambiguous

Names should be clear and specific, avoiding generic or vague terms that could cause confusion.

- *Good*: `BillingAddress`, `ShippingAddress`, `ProductInventory`
- *Bad*: `Data`, `Items`, `Record`, `Object`

Avoid Jargon and Abbreviations

Names **SHOULD** use simple, common terms and avoid internal project codenames, jargon, or unnecessary abbreviations.

- *Good*: `MultiCloudGateway`, `Invoice`
- *Bad*: `McgwObject`, `BillingRecord`

Use Suffixes for Clarity

When a schema has different variations (e.g., for requests vs. responses), use consistent suffixes to distinguish them.

- **For Create/Update Operations**: Use suffixes like `Request` or `Update`.
- `CreateCustomerRequest`
- `UpdateCustomerRequest`
- **For Different Views**: Use suffixes that describe the context.
- `CustomerSummary`
- `CustomerDetails`

Use API Prefixes to Prevent Namespace Collisions

To prevent conflicts when developers generate code from multiple OAS files, all schema names **MUST** be prefixed with a short, unique **API Prefix**.

An **API Prefix** is a short identifier, unique to an API, that is added to the beginning of every schema object's name. Its purpose is to guarantee that all generated class names will be unique, even when a developer consumes multiple APIs that have schemas with similar names.

- **The Problem**: If the MCGW API has an `Order` schema and the IOD API also has an `Order` schema, code generation tools will create two different classes with the same name, causing a conflict.
- **The Solution**: By using prefixes, the schemas are named `McgwOrder` and `IodOrder`, which generate unique and conflict-free classes.

API URI Standard Proposal

Overview & Purpose

This document proposes a standardized URI structure for Lumen APIs. The goal is to create a consistent, predictable, and governable namespace that

clarifies the purpose and scope of each API, supports different customer segments (Enterprise vs. Wholesale), and facilitates platform evolution.

Core URI Structure

The proposed core structure follows a `{domain}/{version}/{resource}` pattern:

- `{domain}`: Identifies the primary functional area or customer segment (e.g., `product`, `business-function`, `wholesale`).
- `{version}`: The major API version (e.g., `v1`, `v2`).
- `{resource}`: The specific resource (noun) being acted upon (e.g., `users`, `keys`, `orders`).

Proposed API Domains

The following top-level domains are proposed to categorize Lumen APIs:

Product Function APIs

- **Description:** Enterprise APIs performing product-specific functions.
- **URI Template:** `/{product}/{version}/{resource}`
- **Example:** `/fabric/v1/ports`, `/mcgw/v1/connections`

Business Function APIs

- **Description:** Enterprise APIs performing cross-cutting business functions (e.g., identity, billing, ordering).
- **URI Template:** `/{business-function}/{version}/{resource}`
- **Example:** `/admin/v1/users`, `/ordering/v1/orders`
- **Note:** If a specific business function needs product-specific variations (e.g., inventory responses differing by product), a separate URI within the function domain might be necessary, rather than embedding product names directly.

Wholesale APIs

- **Description:** A dedicated domain for APIs serving wholesale customers, covering both product and business functions.

- **URI Templates:**
- `/wholesale/{product}/{version}/{resource}`
- `/wholesale/{business-function}/{version}/{resource}`
- **Rationale:** Provides a distinct namespace for wholesale use cases. This enables:
- **Impact Isolation:** Separates wholesale traffic from enterprise traffic.
- **Tailored SLOs:** Allows different performance characteristics (e.g., higher timeouts for bulk wholesale orders).
- **Targeted Auditing & Metrics:** Simplifies tracking of wholesale usage.
- **Implementation Note:** While the gateway URI is separate, a Wholesale API *can* potentially route to the same backend implementation as an Enterprise API to avoid code duplication where functionality aligns.
- **Cost Implication:** Each distinct gateway deployment (like `/wholesale/...`) may incur separate proxy deployment costs in Apigee X.

Partner APIs (Open Question)

- **Discussion Needed:** Do we need a distinct `/partner/...` domain? What specific use cases (Pricing, Quoting, Marketplace integration, shared customer experiences) fall under this category? What are the partner types (Indirect, Hyperscalers, VARs)? This requires further discussion.

Examples: Business Function APIs

This table illustrates how common business functions might map to the proposed URI structure. *Note: Many comments and questions here require discussion by the working group.*

Business Function	**API Name**	**Comment / Questions**	**Method**	**Proposed URI**

User Administration	*(N/A - UI Only?)*	New User Setup and role change not supported through API? Functions performed by admin in UI?	-	-
	`resetUserPassword`		`POST`	`/admin/v1/user/{user_id}/password`
	`deactivateUser`		`POST`	`/admin/v1/user/{user_id}/accounts` *(Note: URI seems mismatched? Should it be* `/admin/v1/users/{user_id}/status`*?)*
	`listEntitlements`	Enables customers to consume on-demand products. Should this be administered through UI only?	`GET`	`/admin/v1/user/entitlements` *(Note: Per user or all? Needs clarification -* `/admin/v1/users/{user_id}/entitlements`*?)*
	`addEntitlement`	Add entitlement to a user	`POST`	`/admin/v1/user/entitlements/{entitlement_id}` *(Note: Needs user context -* `/admin/v1/users/{user_id}/entitlements`*?)*
	`removeEntitlement`	Remove entitlement from a user	`DELETE`	`/admin/v1/user/entitlements/{entitlement_id}` *(Note: Needs user context -* `/admin/v1/use`

				`rs/{user_id}/entitlements/{entitlement_id}`*?)*
API Key Management	`createAPIKey`	Create API Key. Requires existing key for admin APIs? Need mapping of key to allowed endpoints/methods.	`POST`	`/api-key/v1/keys`
	`getAPIKeyList`	List all API Keys for a given user.	`GET`	`/api-key/v1/keys`
	`getAPIkeyDetails`	List APIs a key is allowed to access.	`GET`	`/api-key/v1/keys/{api_key}`
	`deleteAPIKey`	Delete an API key.	`DELETE`	`/api-key/v1/keys/{api_key}`
Authorization	`requestAccessToken`	Needs API Key. No user/pass auth. Key created via platform or API.	`POST`	`/oauth/v2/token`
Service Availability	`listServicesAvailableAtAddress`	Replaces current location API. Use address, not MasterSiteID.	`GET`	`/services/v1/availability`
Inventory Management	`getAvailableInventoryByLocation`	Get list of available product inventory at a location.	`GET`	`/inventory/v1/product` *(Note: Ambiguous URI? Needs location?* `/inventory/v1/locations/{loc_id}/products`*?)*
	`getAvailableInventoryByProduct`	Get list of available product inventory	`GET`	`/inventory/v1/product` *(Note: Ambiguous URI? Needs product*

		(globally?). Priority for MCGW. Dependency on Blue Planet migration?		*filter?* `/inventory/v1/products/{prod_id}/locations`*?)*
	`getAvailableInventoryByProductByLocation`	Get list of available product inventory at a given location.	`GET`	`/inventory/v1/product` *(Note: Ambiguous URI? Needs both filters?* `/inventory/v1/locations/{loc_id}/products/{prod_id}`*?)*
Pricing	`getPricing`	Unauthenticated standard pricing. Covers most on-demand products.	`GET`	`/pricing/v1/catalogue`
Quoting	`createQuote`	Custom pricing for authenticated customers. More for classic products? Wholesale only?	`POST`	`/quoting/v1/price-request`
	`saveQuote`	Save a quote.	`POST`	`/quoting/v1/price-request/{quote_id}`
	`retrieveQuotes`	Get list of saved quotes.	`GET`	`/quoting/v1/price-request`
Ordering	`createOrder`	Creates order from saved quote (classic products). On-demand service order created behind the scenes. Wholesale only?	`POST`	`/ordering/v1/order`

	getOrdersList		GET	/ordering/v1/order
	getOrderDetails		GET	/ordering/v1/order/{order_id}
	checkOrderStatus		GET	/order/v1/order-status/{order_id}
	cancelOrder		POST	/order/v1/resource *(Note: URI seems incorrect?* /ordering/v1/orders/{order_id}/cancel*?)*
Service Management	getServiceLists	High Priority for MCGW	GET	/services/v1/services
	getServiceDetails	Including usage, alerts. What does "service" mean? Align with Portal view.	GET	/services/v1/services/{service_id}
	runServiceDiagnostic		POST	/services/v1/services/{service_id}/diagnostics *(Proposed)*
	createServiceTicket		POST	/support/v1/tickets *(Proposed)*
	listServiceTickets		GET	/support/v1/tickets *(Proposed)*
	checkServiceTicketStatus		GET	/support/v1/tickets/{ticket_id}/status *(Proposed)*
	cancelServiceTicket		POST	/support/v1/tickets/{ticke

				`t_id}/cancel` *(Proposed)*
Scheduled Maintenance	*(TBD)*			
Billing	*(TBD)*			
Telemetry	*(TBD)*	APIs monitoring and usage		

Refined Business Function Categories Proposed

This is an atttempt to group the functions into clearer, more standard domains:

- `identity`: Covers users, authentication, authorization, API keys, and entitlements. (Replaces `User Administration`, `API Key Management`, `Authorization`, parts of `admin`).
- `catalog (or offerings)`: Focuses on discovering *what* services are available *where*. (Replaces `Service Availability`).
- `inventory`: Manages the status and details of specific network resources/assets.
- `pricing`: Handles retrieval of standard pricing information.
- `quoting`: Manages the creation and retrieval of customer-specific quotes.
- `ordering`: Handles the submission and tracking of orders.
- `services`: Manages *provisioned* or *active* customer services (post-order). (Replaces `Service Management`).
- `support`: Manages trouble tickets and diagnostics related to active services. (Extracted from `Service Management`).
- `billing`: (Kept from original TBD).

Improved URIs Based on Refined Categories

Business Function (Domain)	**API Name / Action**	**Method**	**Proposed URI**	**Original URI Notes Addressed**

Identity	Reset User Password	`POST`	`/identity/v1/users/{user_id}/actions/reset-password`	Uses action pattern, clearer than `/password`.
	Deactivate User	`PATCH`	`/identity/v1/users/{user_id}` (Body: `{"status": "inactive"}`)	Uses `PATCH` for state change, corrects mismatched `/accounts` URI.
	List User Entitlements	`GET`	`/identity/v1/users/{user_id}/entitlements`	Clarifies per-user scope.
	Add Entitlement to User	`POST`	`/identity/v1/users/{user_id}/entitlements` (Body: `{"entitlement_id": "...", ...}`)	Creates an entitlement *assignment*.
	Remove Entitlement from User	`DELETE`	`/identity/v1/users/{user_id}/entitlements/{entitlement_id}`	Deletes the *assignment*.
	Create API Key for User	`POST`	`/identity/v1/users/{user_id}/api-keys`	Assumes keys belong to users.
	List User's API Keys	`GET`	`/identity/v1/users/{user_id}/api-keys`	
	Get API Key Details	`GET`	`/identity/v1/users/{user_id}/api-keys/{key_id}`	
	Delete API Key	`DELETE`	`/identity/v1/users/{user_id}/api-keys/{key_id}`	

(Standard OAuth)	Request Access Token	`POST`	`/oauth/v2/token`	Kept standard URI.
Catalog/ Offerings	Check Service Offering Availability	`GET`	`/catalog/v1/availability?address=...` (or other filters)	
Inventory	Get Product Inventory	`GET`	`/inventory/v1/product-inventory?location_id=...&product_id=...`	Uses query parameters for flexible filtering by location, product, or both. Addresses ambiguity. Resource named `product-inventory`.
Pricing	Get Price Catalog	`GET`	`/pricing/v1/rates`	
Quote	Create Quote	`POST`	`/quote/v1/quotes`	
	Save/Update Quote	`PUT` / `PATCH`	`/quote/v1/quotes/{quote_id}`	
	Retrieve Quotes	`GET`	`/quote/v1/quotes`	
	Retrieve Specific Quote	`GET`	`/quote/v1/quotes/{quote_id}`	
Order	Create Order	`POST`	`/order/v1/orders` (Body includes `quote_id` if needed)	
	List Orders	`GET`	`/order/v1/orders`	
	Get Order Details	`GET`	`/order/v1/orders/{order_id}`	
	Cancel Order	`POST`	`/order/v1/orders/{order_id`	

			}/actions/cancel	
Services (Provisioned)	List Services	GET	/services/v1/services	
	Get Service Details	GET	/services/v1/services/{service_id}	(Includes usage, status, alerts etc.).
	Run Service Diagnostic	POST	/services/v1/services/{service_id}/actions/run-diagnostics	Uses action pattern.
Support	Create Support Ticket	POST	/support/v1/tickets	Moved ticketing to dedicated domain.
	List Support Tickets	GET	/support/v1/tickets	
	Get Ticket Details	GET	/support/v1/tickets/{ticket_id}	(Includes status).
	Cancel Support Ticket	POST	/support/v1/tickets/{ticket_id}/actions/cancel	Uses action pattern.
Billing	*(TBD)*	*(TBD)*	*(TBD)*	

Open Issues & Discussion Points for Working Group

- **Partner APIs:** Confirm need, scope, and URI structure (`/partner/...`?).
- **Wholesale Exclusivity:** Confirm if Quoting/Ordering APIs are truly Wholesale-only or need Enterprise equivalents.
- **Product-Specific Business Functions:** Finalize approach for handling variations within business functions (e.g., Inventory).
- **UI vs. API Scope:** Clarify which functions (like User Admin, Entitlements) are intentionally UI-only vs. candidates for APIs.

- **Inventory API Structure:** Proposed URIs seem ambiguous and need refinement to properly filter by location/product. Confirm dependency on Blue Planet migration.
- **Service Management Scope:** Define "service" clearly (align with Portal?). Refine proposed URIs for diagnostics/ticketing.
- **URI Mismatches:** Review URIs noted in the table above (e.g., `deactivateUser`, `cancelOrder`) for correctness.
- **Authentication/Authorization Details:** Clarify API Key management flow, especially creation and permissions mapping.

Next Steps

1. Review proposed domain structure (`product`, `business-function`, `wholesale`).
2. Discuss and decide on the "Partner API" question.
3. Review the Business Function examples, address embedded questions, and refine URIs for clarity and consistency.
4. Assign owners/actions for resolving open issues.
5. Formalize approved structure in the main API Style Guide.

API Pagination Standards

1. Introduction

Pagination is the process of dividing a large set of data into smaller, discrete pages. Proper pagination is essential for creating APIs that are performant, scalable, and easy for our customers and partners to consume. Failure to implement a consistent pagination strategy leads to slow response times, server strain, and a poor developer experience.

This document establishes the proposed official standards for implementing pagination within Lumen's API ecosystem. All development teams are required to adhere to these guidelines to ensure a cohesive and predictable platform.

---

2. The Default Standard: Offset-Based Pagination

For the vast majority of use cases, **Offset-based pagination** should be the default method. It is intuitive, flexible, and meets the needs of most applications where users need to navigate to specific pages.

Query Parameters

Paginated endpoints MUST accept the following query parameters:

- `limit`: An integer specifying the maximum number of items to return per page.
- If not provided, the API MUST default to `25`.
- The API MUST enforce a maximum value of `100`. Requests for more than 100 items should result in a `400 Bad Request` error.
- `offset`: An integer specifying the number of records to skip. This is how the client navigates to subsequent pages.
- If not provided, the API MUST default to `0` (the first page).

Example Request: `GET /v1/services?limit=50&offset=100` (Returns 50 services, starting after the 100th service).

Response Structure

The response body for a paginated request MUST be a JSON object containing two top-level keys: `data` and `pagination`.

- `data`: An array containing the resource objects for the current page.
- `pagination`: An object containing the following metadata fields:
- `total`: The total number of items available in the entire collection.
- `limit`: The `limit` value used for the current page.
- `offset`: The `offset` value used for the current page.

Example Response:

```
{
  "data": [
    { "id": "svc-abc-101", "name": "Service 101" },
    // ... 49 more items
    { "id": "svc-xyz-150", "name": "Service 150" }
  ],
  "pagination": {
    "total": 473,
    "limit": 50,
    "offset": 100
  }
}
```

---

3. High-Performance Standard: Keyset (Cursor-Based) Pagination

For endpoints that expose very large datasets (millions of records) or data that changes frequently (e.g., event logs, real-time data), **Keyset pagination** MUST be used. This method is more performant and provides a stable window into the data, avoiding issues where items are skipped or repeated as new data is added.

Query Parameters

- `limit`: Same definition as in Offset pagination (default `25`, max `100`).
- `after`: A string representing an opaque cursor that points to the last item of the previous page. The API will return items that come after this cursor.

Example Request: `GET /v1/events?limit=50&after=NGM5YTYxYjItM2RiYi00YjU4LTg5ZmMtMTdiYmI5ZjIzMjc5`

Response Structure

The response structure for Keyset pagination is simplified to facilitate "infinite scroll" style navigation.

- `data`: An array containing the resource objects for the current page.
- `pagination`: An object containing the following metadata:
- `next_cursor`: The cursor string for the next page of results. This value is passed into the `after` parameter in the subsequent request. It is `null` if there are no more pages.
- `has_next_page`: A boolean indicating if more pages are available.
- `next`: A full URL string pointing to the next page of results, including the `after` parameter.

Example Response:

```
JSON
```

```
{
  "data": [
    { "id": "evt-123", "timestamp": "2025-10-15T14:30:00Z" },
    // ... 49 more items
    { "id": "evt-456", "timestamp": "2025-10-15T14:28:00Z" }
  ],
  "pagination": {
    "next_cursor": "ZjkzMjc5YTYxYjItM2RiYi00YjU4LTg5ZmMtMTdiYmI5Zj",
    "has_next_page": true,
    "next": "https://api.lumen.com/v1/events?limit=50&after=ZjkzMjc5YTYxYjIt
  }
}
```

---

4. Summary of Guidance

1. **Default to Offset:** Use Offset pagination for all standard resource collections.
2. **Use Keyset for Scale:** Use Keyset (Cursor) pagination for event streams, logs, and any dataset exceeding one million records where performance and data consistency are critical.
3. **Provide Hypermedia Links:** Always include fully-qualified URLs for `next` and `previous` links. This reduces the burden on the client and makes the API more discoverable (HATEOAS).

4. **Enforce Limits:** Always enforce a default and maximum `limit` to protect the API from misuse.
5. **Document Clearly:** The chosen pagination method and its parameters must be clearly documented in the OpenAPI specification for every applicable endpoint.

API Standard: Lumen Problem Details — Minimal Profile (LPDP-Mini v1.0)

Lumen Problem Details — Minimal Profile (LPDP-Mini v1.0)

Purpose

Provide a **lightweight, RFC 9457-compliant** error envelope that's consistent across all Lumen APIs.
It focuses on three essentials — `code`, `message`, and optional `meta` — while letting teams extend `meta` per API as needed.

Profile Summary

Design Goal	Description
Simplicity	Practical & viable RFC 9457 extension that developers can implement correctly.
Stability	`code` is machine-readable and stable across languages and messages.
Extensibility	`meta` can be open, omitted, or refined per API.
Security	No PII, credentials, or stack traces in error payloads.
Compliance	Fully aligns with RFC 9457 – Problem Details for HTTP APIs.

Core Structure (Org-wide Standard)

Media Type

All error responses **MUST** use:

`Content-Type: application/problem+json`

JSON Schema (OpenAPI fragment)

```
components:
  schemas:
    LpdpProblem:
      type: object
      additionalProperties: false
      description: Minimal Problem-Details envelope (RFC 9457-compliant)
      properties:
        title:
          type: string
          description: Short, human-readable summary.
        detail:
          type: string
          description: Optional longer explanation.
        errors:
          type: array
          minItems: 1
          items: { $ref: '#/components/schemas/LpdpError' }
      required: [ title, errors ]

    LpdpError:
      type: object
      additionalProperties: false
      description: Individual error item.
      properties:
        code:
          type: string
          description: Stable, machine-readable identifier (string form pref
        message:
          type: string
          description: Human-readable explanation of the issue.
        meta:
          type: object
          description: >
            Optional unregulated extension for machine-readable context.
            Lumen does not prescribe its structure. Teams may define custom
```

```
            schemas if needed.
          additionalProperties: true
```

RFC 9457 Alignment

RFC 9457 Member	LPDP-Mini Treatment	Rationale
`type`	Optional / constant URI	RFC 9457 §3.1.1 allows omission.
`title`	Required	Short human summary.
`status`	Omitted	HTTP status line authoritative.
`detail`	Optional	Narrative text.
`instance`	Optional	Rarely used.
`errors`	Extension (§3.2)	Array of per-error details.
`code`,`message`,`meta`	Extensions	Fully compliant.

`code` — Role and Representation

What `code` is

A **stable, machine-readable key** identifying the error category; independent of message wording or localization.

Naming Convention

```
<domain>.<category>.<reason>
```

Examples: `auth.missing`, `resource.not_found`, `dependency.failed`.

Why string codes are standard

Aspect	String (`auth.missing`)	Numeric (`1043`)
Readability	Human-friendly in logs	Opaque

Aspect	String (`auth.missing`)	Numeric (`1043`)
Namespacing	Simple (`auth.*`)	None
Client branching	Safe (`if code==...`)	Ambiguous
Governance	Regex enforceable	Central registry needed
Size	Slightly larger	Smaller
Interop	Locale-agnostic	Requires catalog
HTTP overlap	Distinct from status codes	Confusing

> **Recommendation:**
> Lumen should use **string codes** exclusively.
> Numeric identifiers, if required for backward compatibility, appear only as `meta.legacy_code`.

`meta` — Optional and Team-Defined

Guidance

- Optional, uncontracted unless a team defines its own schema.
- **Never** include secrets, internal identifiers, hostnames, stack traces, PII, or full payloads.
- Keep small (< 4 KB) and limited to relevant machine context.
- Teams may narrow `meta` locally if they need a structured contract.

Example of Per-API Override

```
components:
  schemas:
    LpdpError_MCGW_V1:
      allOf:
        - $ref: '#/components/schemas/LpdpError'
        - type: object
          properties:
            meta:
              $ref: '#/components/schemas/McgwErrorMetaV1'

    McgwErrorMetaV1:
      type: object
```

```
    additionalProperties: false
    description: Meta structure specific to MCGW v1 APIs.
    properties:
      subsystem: { type: string }
      operation: { type: string }
      cause:
        type: string
        enum: [timeout, unavailable, error, authorization, unknown]
    required: [ subsystem, cause ]
```

Canonical Examples and HTTP Status Mapping

Use Case	Status	Example Summary
Malformed Request	**400 Bad Request**	Invalid JSON or schema syntax.
Unauthorized / Forbidden	**401 / 403**	Missing credentials / insufficient rights.
Validation Errors	**422 Unprocessable Entity**	Field-level or semantic violations.
Resource Not Found	**404 Not Found**	Composite key absent.
Ambiguous Match / Conflict	**409 Conflict**	Multiple resource matches or conflicting state.
Dependency Failure	**424 Failed Dependency** (alt 502/503/504)	Downstream timeout / failure.
Internal Failure / Unknown Cause	**500 Internal Server Error**	Unexpected system condition.

(a) Validation Error — 422

```
{
  "title": "Validation failed",
  "errors": [
    { "code": "string.min", "message": "[name] must be at least 3 characters
    { "code": "cidr.invalid", "message": "[ip_address] must be a valid IPv4
  ]
}
```

(b) Dependency Failure — 503

```
{
  "title": "Backend dependency failed",
  "errors": [
    {
      "code": "dependency.failed",
      "message": "VRF service did not respond.",
      "meta": { "subsystem": "VRFService", "operation": "lookupVrf", "resour
    }
  ]
}
```

(c) Composite Key Not Found — 404

```
{
  "title": "VRF not found",
  "errors": [
    { "code": "resource.not_found", "message": "No VRF matched the provided
  ]
}
```

(d) Ambiguous Resource — 409

```
{
  "title": "VRF selection is ambiguous",
  "errors": [
    { "code": "resource.ambiguous", "message": "More than one VRF matches th
  ]
}
```

(e) Unknown Failure — 500

```
{
  "title": "Unable to fetch VRF information",
  "errors": [
    { "code": "resource.lookup_failed", "message": "VRF information could no
```

```
  ]
```

(f) Auth Failures — 401 / 403

```
{
  "title": "Unauthorized",
  "errors": [ { "code": "auth.missing", "message": "Authorization header is
}
```

```
{
  "title": "Forbidden",
  "errors": [ { "code": "auth.forbidden", "message": "Caller not permitted t
}
```

(g) Malformed Request — 400

```
{
  "title": "Malformed request",
  "errors": [ { "code": "request.invalid_json", "message": "Request body is
}
```

Governance Rules

1. All 4xx / 5xx responses → `application/problem+json`.
2. Each error → `code` and `message`.
3. `code` → **string** form only; numeric legacy values in `meta.legacy_code`.
4. `meta` → optional; must exclude sensitive or internal data.
5. Per-API teams may narrow `meta` via local schemas without breaking the global profile.

Summary

Field	Required	Purpose
`title`	✅	Human-readable summary
`errors`	✅	Array of issues
`code`	✅	String identifier (machine key)
`message`	✅	Human explanation
`meta`	⚙️	Optional contextual data
`detail`, `instance`, `type`	⚙️	Optional per RFC
`status`, `trace_id`	❌	Omitted
Rate-limit info	—	Conveyed via headers only

API Standard: Idempotency

1. What is Idempotency?

An **idempotent** operation is an operation that can be performed multiple times without changing the result beyond the initial application.

In the context of a REST API, this means that making the same request multiple times (e.g., due to a network error and retry) will produce the same result and system state as making it just once.

2. Why is Idempotency a Standard?

Idempotency is not an optional feature; it is a **critical component of a robust and reliable API platform.** Clients (mobile apps, web frontends, other services) operate on unreliable networks. A client may send a request, the server may process it, but the response may be lost before it reaches the client.

Without idempotency, this client has no safe way to recover.

- **The Risk:** The client doesn't know if the request succeeded.If it retries a `POST /users` request, it may create two identical users. If it retries a

`POST /charges` request, it may charge the customer twice.

- **The Solution:** This standard ensures all endpoints are safe to retry, protecting our systems and our customers from data duplication and unintended side effects.

3. Idempotency by HTTP Method

This standard defines the required behavior for each HTTP method.

Method	Idempotent by Standard?	Notes
`GET`	**MUST**	`GET` requests are *safe* (they don't change state) and must be idempotent.
`PUT`	**MUST**	A `PUT` request replaces a resource. `PUT /users/123` twice is the same as once.
`DELETE`	**MUST**	A `DELETE` request deletes a resource. The first call deletes it (returning `204`). The second call (and all subsequent) should return `404 Not Found`. The system state remains the same ("deleted").
`PATCH`	**MUST**	A `PATCH` operation must be idempotent. Server logic must ensure that re-applying the same patch does not produce a different result. (e.g., `PATCH /item/123 {"name": "new"}` twice is fine).
`POST`	**MUST** support idempotency	`POST` is not naturally idempotent. This standard **requires** all `POST` endpoints to support an idempotency-key mechanism to *make* them idempotent.

4. `POST` Idempotency: The `idempotency-key` Header

Since `POST` requests create new resources, they are the highest risk for duplication. To mitigate this, all `POST` endpoints that create resources **MUST** support idempotency via a request header.

The Standard

`POST` endpoints **MUST** honor the `idempotency-key` header.

Client Responsibility

When a client sends a `POST` request, it **SHOULD** generate a unique key (e.g., a **UUID**) and send it in the `idempotency-key` header.

```
HTTP
```

```
POST /v1/customers
idempotency-key: a8b4-12a8-8f81-9b48-18e0-128a
Content-Type: application/json

{
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
}
```

If the client suffers a network error and needs to retry, it **MUST** send the *exact same* `idempotency-key` with the retry request.

Server Responsibility

The server **MUST** perform the following logic for any `POST` request containing an `Idempotency-Key`:

1. **Check for Key:** The server checks an internal cache (e.g., Redis) for this key.
2. **Key Not Found (New Request):**
3. This is the first time this request has been seen.

4. The server processes the request as normal (e.g., creates the customer in the database).
5. The server then **stores the HTTP response** (e.g., the `201 Created` status and the JSON body) in the cache, using the `Idempotency-Key` as the cache key.
6. The server returns the response to the client.

7. **Key is Found (Retry):**

8. This is a retry. The server **MUST NOT** re-process the request.

9. It immediately fetches the *original, saved response* from the cache.
10. It returns that exact same response to the client.

This flow guarantees that a client can safely retry a `POST` without fear of creating duplicate resources.

Key Details

- **Key Format:** `idempotency-key` values **SHOULD** be a **UUID** to ensure uniqueness.
- **Key Lifecycle:** The server **SHOULD** store idempotency keys for at least **24 hours** to allow clients ample time to retry failed requests.

API Standard: Partial Resource Retrieval

Purpose

This page extends the **Lumen API Style Guide (v1.0)** to define a standard for **partial resource retrieval** — enabling clients to request only the fields they need, reducing payload size and latency for high-volume NaaS APIs (e.g., telemetry, connections, interfaces).

Guiding Principle

> "Always return full canonical resources **by default**, but allow clients to **opt-in** to partial responses using explicit field projection."

Partial retrieval improves performance and bandwidth efficiency without changing the canonical model or breaking API contracts.

Standard Definition

Category	Rule	Requirement
Parameter Name	`fields`	**MUST**
Location	Query Parameter	**MUST**
Type	Comma-separated list of field names (no spaces)	**MUST**
Behavior	Return only specified fields; ignore unknown ones or return `400`	**MUST**
Default	Full resource if `fields` not provided	**MUST**
Case Sensitivity	Field names are case-sensitive and match the schema exactly	**SHOULD**
Nested Objects	Use dot notation for sub-fields (e.g., `location.city`)	**MAY**
Error Handling	Return `400 Bad Request` with `problem+json` if invalid field names supplied	**SHOULD**

Example: Fabric Connection Service

🔹 Full Resource

```
GET /fabric/v1/connections/12345
```

Response

```
{
  "id": "12345",
  "name": "AWS-Transit-Connect",
  "status": "active",
  "bandwidth": {
```

```
    "committed_mbps": 1000,
    "burst_mbps": 2000
  },
  "location": {
    "city": "Denver",
    "region": "US-CENTRAL-1"
  },
  "provider": "aws",
  "tags": ["gold", "naas"]
}
```

◆ Partial Resource (Using `fields`)

```
GET /fabric/v1/connections/12345?fields=id,name,status
```

Response

```
{
  "id": "12345",
  "name": "AWS-Transit-Connect",
  "status": "active"
}
```

Nested Field Example – MCGW Interface

```
GET /mcgw/v1/interfaces/7df9a?fields=id,device.name,device.state
```

Response

```
{
  "id": "7df9a",
  "device": {
    "name": "edge-router-01",
    "state": "up"
  }
}
```

Validation & Governance Automation

Tool	Rule	Example
Spectral Linter	Ensure `fields` is declared in GET operations where allowed	`oas-parameter-name: fields`
Governance Rule	Lint rule `naas-partial-response-allowed`	Enforced only for `GET` endpoints returning resources > 10 fields

Performance Considerations

- Reduces payload size by **30–80 %** for high-volume telemetry endpoints.
- Lowers serialization/deserialization overhead in API Gateway and MCGW controllers.
- Prevents over-fetching in portal UI and CLI integrations.

Security & Compliance

- Projection **cannot** bypass authorization. The API MUST validate that each requested field is permitted under the caller's security scope.

Summary

Benefit	Impact
Reduced payloads and latency	Better API responsiveness for UI and CLI clients
Standardized pattern across products	Consistent developer experience
Governance-ready	Easily enforced via OAS and Spectral rules

API Standard: Delete Method

DELETE Method – Request and Response Standards

Summary

DELETE represents a request to remove the entire resource identified by the URI.
It is *not* intended for partial removal, filtered deletion, or passing instruction payloads.

Request Rules

Rule	Requirement	Rationale
MUST NOT include a request body	DELETE has undefined semantics for a body per RFC 9110 §9.3.5	Ensures predictable behavior across proxies, SDKs, and gateways
MUST target a single resource URI	DELETE /{resource}/{id}	Full deletion semantics, not partial update
MUST NOT support conditional filters or lists in the body	Use PATCH or POST /{resource}:delete instead	Keeps REST semantics pure and idempotent
MUST be idempotent	Multiple identical DELETEs should have the same effect	Required for retry safety

---

📜 Example (Correct)

```
DELETE /fabric/v1/connections/12345
```

Response

```
HTTP/1.1 204 No Content
```

🚫 Example (Incorrect)

```
DELETE /mcgw/v1/routes
Content-Type: application/json

{
  "routeIds": ["r1","r2","r3"]
}
```

Why This Is Non-Compliant:
This DELETE request attempts partial modification via a request body, which has no defined semantics under RFC 9110.
Use PATCH or a POST action endpoint instead.

Response Rules

Status Code	Meaning	Use Case
204 No Content	✅ Successful deletion Or Resource does not exist	Resource deleted successfully Or Target resource already deleted or invalid
400 Bad Request	Invalid URI, malformed query parameter	Syntax or parameter validation failure
401 Unauthorized	Authentication required	Caller not authenticated
403 Forbidden	Insufficient permission	Caller lacks permission to delete
405 Method Not Allowed	DELETE not supported for this resource	Used when DELETE not implemented
409 Conflict	Discouraged. If 409 is possible during delete, model it as POST with cancel action.	e.g., “Resource locked” or “pending operation”
422 Unprocessable Content	Discouraged. If 422 is possible during delete, model it as POST with cancel action.	e.g., “Cannot delete active policy” — prefer PATCH or POST action
500 Internal Server Error	Platform/system failure	Server-side unexpected error

---

Response Body Rules

- **MUST NOT** include a response body for Delete.

Alternatives for Non-Standard Deletion Use Cases

Use Case	Recommended Design	Example
Delete subset of items	PATCH with remove operation	`PATCH /mcgw/v1/routes { "remove": ["r1","r2"] }`
Batch or filtered deletion	POST to action endpoint	`POST /fabric/v1/connections:delete { "ids": ["c1","c2"] }`
Conditional deletion	POST to /{resource}:terminate with conditions	`POST /mcgw/v1/sessions:terminate { "force": true }`

Governance Enforcement Examples

Rule 1 – No DELETE Request Body

```
rules:
  lumen-no-delete-body:
    description: "DELETE requests must not define requestBody"
    given: "$.paths[*].delete"
    then:
      field: requestBody
      function: falsy
```

Rule 2 – Valid DELETE Responses

```
rules:
  lumen-delete-status-codes:
    description: "Allowed HTTP status codes for DELETE"
    given: "$.paths[*].delete.responses"
    then:
      function: lumen-validate-status-code
      functionOptions:
        validCodes: [204, 400, 401, 403, 404, 405, 500]
```

Summary Table

Category	Guideline
Request	DELETE must target a single resource and have no body
Response	Use only predefined codes (204, 400–500)
Governance	Enforced via Spectral rules for DELETE behavior and status codes

---

Key Takeaways

- DELETE is for **entire resource removal only** — not partial, conditional, or batch operations.
- No request body allowed — maintain idempotency and HTTP compliance.
- Use **PATCH** for partial deletion or **POST :action** for complex or conditional deletions.
- Responses should be predictable and minimal (typically `204 No Content`).
- All error responses should follow **RFC 7807** for consistency and automation.

API Standard: HTTP Headers

1. Naming & Formatting

- **MUST** use lowercase kebab-case ASCII for all HTTP header names.
- ✅ `idempotency-key`, `traceparent`, `content-type`, `if-match`
- ❌ `X_Lumen_Entity_ID`, `X-Lumen-Entity-Id`, `RequestId`, non-ASCII
- **MUST NOT** use the legacy `x-` prefix for new headers (per RFC 6648).
- **MUST** document any legacy interoperability exceptions in the API's local specification.

---

2. Header Registry (Allow-list)

To ensure client simplicity and enforceable governance, APIs **MUST** restrict headers to the following allow-list. Additions require a formal exception via the API Governance process.

2.1 Standard Headers (Allowed)

- **Content & Negotiation:** `accept`, `content-type`, `content-length`, `content-encoding`
- **Authentication:** `authorization`
- **Caching:** `cache-control`, `etag`, `last-modified`, `expires`, `vary`
- **Conditional Requests:** `if-match`, `if-none-match`, `if-modified-since`, `if-unmodified-since`
- **Conformance & Links:** `link`, `location`, `retry-after`, `date`
- **Range (when applicable):** `range`, `content-range`
- **Tracing & Correlation:** `traceparent`, `tracestate` (W3C)

Any new custom headers **MUST** be defined with a purpose, schema, max length, examples, and security considerations before being added to this list.

---

3. Authentication & Security

- **MUST** use `authorization: Bearer <access_token>` for OAuth2/JWT authentication.
- **MUST NOT** introduce proprietary authentication headers without a formal review from the Security team.
- **MUST NOT** place secrets or PII in headers or URLs. Always use request bodies over TLS.
- **SHOULD** include the `traceparent` in all responses for correlation.

---

4. Content Negotiation & Errors

- **Clients SHOULD** send `accept: application/json`. Clients must only request media types the API supports.

- **Servers MUST** set `content-type: application/json` for all JSON response bodies.
- **Servers MUST** use `content-type: application/problem+json` and conform to **RFC 9457 (Problem Details)** for all 4xx and 5xx error responses.
- **Servers MUST** return a `406 Not Acceptable` if the client's `accept` header does not match a supported media type.

---

5. Caching & Conditional Requests

- `ETag`: Servers **MUST** include an `ETag` header on all `GET` responses for cacheable resources. This is preferred over `Last-Modified`.
- `If-None-Match`: Servers **MUST** honor the `If-None-Match` request header, returning a `304 Not Modified` if the `ETag` matches.
- `If-Match`: Servers **MUST** use the `If-Match` header to enable optimistic concurrency control for `PATCH`, `PUT`, or `DELETE` requests.
- `Cache-Control`: **MUST** be set according to the resource's caching policy (e.g., `no-store` for sensitive data, `max-age` for public metadata).
- `Location`: **MUST** be returned with a `201 Created` response, containing the canonical URI of the newly created resource.

---

6. Observability & Correlation

- **MUST** implement W3C Trace Context (`traceparent`, `tracestate`).
- **Clients SHOULD** generate and send the `traceparent` header.
- **Gateways MUST** ensure every request has a `traceparent`. If a client request is missing the `traceparent` header, the gateway **MUST** generate one before forwarding the request to internal services.
- **Servers MUST** propagate the `traceparent` header across all internal service boundaries.
- **Servers MUST** include the `traceparent` value in all error responses for end-to-end tracing.

---

7. Rate Limiting

Gateway should handle ratelimiting and use below headers.

- **SHOULD** use the IETF-standardized `RateLimit` response fields:
- `ratelimit-limit`: The quota for the time window.
- `ratelimit-remaining`: The number of requests remaining.
- `ratelimit-reset`: The remaining time in seconds until the quota resets.
- **MUST** include the `Retry-After` header (in seconds) when returning a `429 Too Many Requests` or `503 Service Unavailable` status due to throttling.

---

8. Size Constraints

- **Size Limits:** Header values **MUST** be ASCII. Total header size **SHOULD** remain within gateway limits (e.g., 8KB).

---

9. Governance & Linting (Spectral Snippets)

The following Spectral rules SHOULD be used to enforce these header standards.

Header Name Format

```
rules:
  lumen-header-name-format:
    given: "$..parameters[?(@.in=='header')].name"
    then:
      function: pattern
      functionOptions: { match: "^[a-z][a-z0-9-]*$" }
```

---

10. Change Control

- Additions to the allow-list or new custom headers **MUST** go through API Governance with Security review.

API Standard: Caching & Concurrency with ETag

1. What is an ETag?

An **ETag (Entity Tag)** is an HTTP header that provides an identifier for a specific version of a resource.

Think of it as a "**fingerprint**" or "**version hash**" for a resource's state. When a resource (like a JSON object for a user) is created or modified, the server **MUST** generate a new, unique ETag value for it.

Clients **MUST** treat the ETag as an opaque string. They should store it, but never try to parse or interpret its contents.

Example ETag Response Header:

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "abc-123-xyz-789"
{
  "id": "user-42",
  "name": "Jane Doe"
}
```

1.1. Standard Mandate: Strong vs. Weak ETags

While all ETags serve a purpose, our standard **MANDATES** the use of **Strong ETags** because they are required for reliable concurrency control.

Feature	**Strong ETag (Required for our API)**	**Weak ETag (Not allowed for Concurrency)**

Primary Use	**Concurrency Control** (Optimistic Locking) and reliable caching.	Caching only (when byte-for-byte fidelity is not critical).
Guarantee	The resource is **byte-for-byte identical**. Any change, no matter how small, **MUST** generate a new ETag.	The resource is **semantically equivalent**, but minor presentation changes (like a footer timestamp) may not generate a new ETag.
Format	Enclosed in standard quotes (no prefix). `"v1-hash-xyz"`	Preceded by `W/`. `W/"v1-hash-xyz"`

Crucial Note: Only the **Strong ETag** provides the guarantee necessary for the server to safely allow a destructive update (`PUT`, `PATCH`, `DELETE`). The server **MUST NOT** accept an `If-Match` header containing a Weak ETag.

---

2. Why This Standard is Required

ETags are not optional. They are our platform's standard for solving two critical, high-level API problems:

- **Efficient Caching:** Prevents clients from re-downloading data they already have, saving bandwidth and improving client-side performance.
- **Concurrency Control:** Prevents the "lost update" problem, ensuring that clients don't accidentally overwrite each other's changes.

This guide defines the two ways ETags **MUST** be used.

---

3. Use Case 1: Caching (Conditional GET)

This flow prevents a client from re-downloading a resource that has not changed.

The Flow

1. Server **MUST** Send ETag: All `GET` responses for a single, cacheable resource **MUST** include an `ETag` header.
2. Client **SHOULD** Store ETag: The client application (web app, mobile app) **SHOULD** cache the resource's body and its `ETag` value.
3. Client **MUST** Send If-None-Match: The next time the client wants to `GET` that same resource, it **MUST** include the stored ETag value in an `If-None-Match` request header.
4. Server **MUST** Validate: The server receives the request and compares the `If-None-Match` value with the resource's current ETag.

The Result

- If the ETags **MATCH**: The client's version is up-to-date. The server **MUST** stop and return an empty **304 Not Modified** response. This saves the server from computing the response and saves bandwidth by not sending the data again.
- If the ETags **DO NOT MATCH**: The resource has changed. The server **MUST** proceed as normal, returning a **200 OK** with the full, new resource and the new `ETag`.

---

4. Use Case 2: Concurrency Control (Optimistic Locking)

This is the most critical function of ETags. It prevents clients from overwriting each other's work.

The "Lost Update" Problem

1. Alice `GET` /users/123. She receives the user's data and the ETag: `"v1"`.
2. Bob also `GET` /users/123. He receives the same data and the same ETag: `"v1"`.
3. Alice sends a `PATCH` to change the user's name. Her request succeeds. The server updates the user and generates a new ETag: `"v2"`.
4. Bob now sends a `PATCH` to change the user's phone number. His request is based on the old `"v1"` data. Bob's update overwrites Alice's

name change. Alice's work is lost.

The ETag Solution (Mandatory)

All state-changing methods (`PUT`, `PATCH`, `DELETE`) **MUST** be protected against this.

1. Server **MUST** Enforce If-Match: Servers **MUST** require the `If-Match` header on all `PUT`, `PATCH`, and `DELETE` requests.
2. Client **MUST** Send If-Match: A client **MUST** include the ETag of the resource it thinks it's updating in the `If-Match` header.

The Correct Flow

1. Alice `GET` /users/123. She receives ETag: `"v1"`.
2. Bob `GET` /users/123. He also receives ETag: `"v1"`.
3. Alice sends `PATCH` /users/123 with the header `If-Match: "v1"`.
4. The server checks: `If-Match` value (`"v1"`) matches the current ETag (`"v1"`).
5. The server accepts the update, processes it, and generates a new ETag: `"v2"`.
6. Bob now sends `PATCH` /users/123 with his header `If-Match: "v1"`.
7. The server checks: `If-Match` value (`"v1"`) **does not match** the current ETag (`"v2"`).
8. The server **rejects** the request with **412 Precondition Faile**d .
9. Bob's update fails, which is the correct behavior. He is notified that the resource changed underneath him. He must now re-`GET` the resource to get the new data (and the `"v2"` ETag) and then retry his change.

5. Server Response Summary

This table summarizes the server's mandatory behavior when `ETag`-related headers are used.

Request Header Sent	**Use Case**	**Client ETag vs. Server ETag**	**Server Response**	**What It Means**

`If-None-Match`	Caching (`GET`)	**Match**	`304 Not Modified`	"The client's cached version is still good. Don't send the body."
`If-None-Match`	Caching (`GET`)	**No Match**	`200 OK`	"The resource has changed. Here is the new version and new `ETag`."
`If-Match`	Concurrency (`PUT`, `PATCH`, `DELETE`)	**Match**	`200 OK` (or `204`)	"The client was working on the correct version. The update is accepted."
`If-Match`	Concurrency (`PUT`, `PATCH`, `DELETE`)	**No Match**	`412 Precondition Failed`	"**Conflict!** The resource was changed by someone else. The update is rejected."

---

6. OpenAPI Specification (OAS 3.1) Examples

Example: Defining ETag on a GET Response

```
paths:
  /users/{userId}:
    get:
      summary: Get a single user
      responses:
        '200':
          description: The user resource.
          headers:
            etag:
              schema:
                type: string
              description: The ETag for this version of the resource.
          content:
            application/json:
```

```
              schema:
                $ref: '#/components/schemas/User'
```

Example: Requiring If-Match on a PATCH Request

```
paths:
  /users/{userId}:
    patch:
      summary: Update a user
      parameters:
        - in: header
          name: if-match
          description: The ETag of the resource to update, for concurrency c
          required: true
          schema:
            type: string
      requestBody:
        # ... request body definition
      responses:
        '200':
          description: Update successful.
          headers:
            etag:
              schema:
                type: string
              description: The new ETag for the updated resource.
        '412':
          description: Precondition Failed. The resource has been modified.
          content:
            application/problem+json:
              schema:
                $ref: '#/components/schemas/Problem'
```

API Standard: Long-Runing Operations (LRO)

1. Summary

This document defines the standard asynchronous Long-Running Operation (LRO) pattern for all Lumen APIs. This pattern **must** be used when an API operation cannot be completed within a standard, synchronous HTTP request-response cycle (e.g., > 2-3 seconds).

Holding an HTTP connection open is fragile and resource-intensive. The LRO pattern solves this by immediately acknowledging the client's request, returning a "job" or "operation" resource the client can poll for status, and decoupling the request's acceptance from its final completion.

This pattern provides:

- Deterministic asynchronous behavior
- Standard client polling and progress tracking
- Reliable retries via idempotency
- A clear separation of **resource state** vs. **operation state**
- Structured error reporting (per our **LPDP-Mini v1.0** profile) for failures

---

2. Why This Pattern Matters for Lumen

Many of our core business processes are complex and not instantaneous. Synchronous APIs are unsuitable and unreliable for these tasks. This pattern is the standard for:

- **Service Provisioning:** Activating a new network service involves orchestration across multiple systems and can take minutes or hours.
- **Wholesale & Bulk Operations:** A partner submitting a POST for a wholesale order containing 5,000 circuits cannot be handled in a single synchronous transaction.
- **Network Decommissioning:** Tearing down a complex service requires a series of ordered steps that must be tracked.
- **Large Data Exports:** Generating a large-scale report (e.g., "export all billing data for the last 12 months").

---

3. The LRO Pattern Explained

3.1 Workflow

Step	**Description**	**Example**

1. **Initiate**	Client sends request (POST, PUT, PATCH, DELETE) to start a long-running transaction.	`DELETE /ports/{id}`
2. **Accept**	Server validates, starts background job; responds 202 Accepted.	Headers: `Location: /operations/{opId}`
3. **Poll**	Client polls `GET /operations/{operationId}` until terminal.	Payload: status, percent, error?
4. **Result**	On success, link to result; on failure, return Problem Details.	`resource.href: "/ports/123"`
5. **Cleanup**	Operation record is purged after a defined TTL.	`GET /operations/{opId}` → 404

3.2 Mandatory Operation Resource Schema

The `GET /operations/{operationId}` endpoint is the "source of truth" for the job. It **must** return a response body adhering to the following fixed schema.

Here is an **example** of a `FAILED` operation response:

```
{
  "operationId": "a1b2-c3d4-e5f6",
  "status": "FAILED",
  "percent": 33,
  "startedAt": "2025-10-29T16:00:00Z",
  "completedAt": "2025-10-29T16:00:15Z",
  "metadata": {
    "message": "Failed during partner verification.",
    "step": "PARTNER_VERIFY"
  },
  "resource": null,
  "error": {
    "title": "Partner timeout",
    "detail": "The operation failed because a downstream partner did not res
    "errors": [
      {
        "code": "PARTNER_TIMEOUT",
        "message": "Partner did not confirm teardown within SLA",
        "meta": {
          "traceId": "traceId-abc123"
        }
```

```
    }
  ]
}
```

3.2.1 Formal OAS 3.1 Schema Definitions

```
# These schemas MUST be used for the GET /operations/{id} response
components:
  schemas:
    Operation:
      type: object
      required: [operationId, status, startedAt]
      properties:
        operationId:
          type: string
          description: Unique identifier for the operation.
          example: "a1b2-c3d4-e5f6"
        status:
          type: string
          description: The current state of the operation.
          enum: [PENDING, RUNNING, SUCCEEDED, FAILED, CANCELLED]
        percent:
          type: integer
          format: int32
          description: Estimated progress percentage (0-100).
          example: 33
        startedAt:
          type: string
          format: date-time
          description: UTC timestamp when the operation was accepted.
          example: "2025-10-29T16:00:00Z"
        completedAt:
          type: string
          format: date-time
          description: UTC timestamp when the operation entered a terminal s
          example: "2025-10-29T16:00:30Z"
        resource:
          type: object
          description: Present on SUCCEEDED for POST/PUT/PATCH.
          properties:
            id:
              type: string
              description: The ID of the created/modified resource.
              example: "port-1234"
            href:
```

```
          type: string
          format: uri
          description: The full URI to GET the resource.
          example: "/v1/ports/1234"
    error:
      description: Present on FAILED. Follows the LPDP Problem Details p
      $ref: '#/components/schemas/LpdpProblem'
    metadata:
      type: object
      description: >
        Service-specific metadata. This object contains rich, real-time
        status information.
      properties:
        message:
          type: string
          description: The current human-readable status message.
          example: 'Step 2 of 4: Verifying configuration...'
        step:
          type: string
          description: A machine-readable code for the current sub-state
          example: 'VERIFYING_CONFIG'
      additionalProperties: true

# ------ LPDP Problem Details Profile Schemas ------
LpdpProblem:
  type: object
  additionalProperties: false
  description: Minimal Problem-Details envelope (RFC 9457-compliant)
  properties:
    title:
      type: string
      description: Short, human-readable summary.
    detail:
      type: string
      description: Optional longer explanation.
    errors:
      type: array
      minItems: 1
      items: { $ref: '#/components/schemas/LpdpError' }
  required: [ title, errors ]

LpdpError:
  type: object
  additionalProperties: false
  description: Individual error item.
  properties:
    code:
      type: string
```

```
    description: Stable, machine-readable identifier (string form pref
  message:
    type: string
    description: Human-readable explanation of the issue.
  meta:
    type: object
    description: >
      Optional unregulated extension for machine-readable context.
      Lumen does not prescribe its structure. Teams may define custom
      schemas if needed.
    additionalProperties: true
```

3.3 Industry Standard Schemas (Google & Microsoft)

Major cloud providers use fixed schemas for operations response. This predictability is why the pattern is successful.

- **Google Cloud (AIP-151):** Uses a fixed schema for `google.longrunning.Operation`. It has standard fields: `name` (the op ID), `done` (boolean), a `result` field (for `error` or `response`), and a `metadata` object. This `metadata` object is used to hold the exact kind of qualitative, real-time status (e.g., 'Verifying configuration...') that our standard now requires.
- **Microsoft Azure:** Also uses a fixed schema. The polling URL (from `Location` or `Azure-AsyncOperation` headers) returns a standard object with fields like `status`, `id`, `startTime`, `endTime`, `percentComplete`, and `error`.

Our schema in 3.2 is an industry-aligned "best-of-both" approach, standardizing on the `operationId`, `status`, and our `LpdpProblem` error structure.

3.4 HTTP Semantics by Use Case

Use Case	**URI**	**Expected Codes**
Async Create	POST /resource	201 (sync) or 202 (async)
Async Replace/Update	PUT /resource/{id}	200/204 or 202
Async Partial Update	PATCH /resource/{id}	200 or 202
Async Delete	DELETE /resource/{id}	204 (sync) or 202 (async)

Poll for Status	**GET /operations/{id}**	**200 (status) → 404 after TTL**
List Operations	GET /operations	200
Request Cancellation	DELETE /operations/{id}	202

3.5 State Machine

3.5.1 State Definitions

- **RUNNING:** The operation is actively being processed.
- **SUCCEEDED:** The operation completed successfully.
- **FAILED:** The operation failed due to an error. The `error` field in the response **must** be populated.
- **CANCELLED:** The operation was successfully cancelled by a client request (`DELETE /operations/{id}`) and did not complete.

3.5.2 State Transitions

Operation states are terminal. Once an operation is SUCCEEDED, FAILED, or CANCELLED, its state **must not** change.

- RUNNING → SUCCEEDED, FAILED, CANCELLED
- SUCCEEDED → (No transitions)
- FAILED → (No transitions)
- CANCELLED → (No transitions)

3.6 Client Polling Strategy

Clients **must** implement a robust polling strategy.

- **Backoff:** Clients **should** use **exponential backoff with jitter** to avoid overwhelming the server (e.g., 1s, 2s, 4s, 8s... capped at 30-60s).
- **Retry-After Header:** If the `202 Accepted` or `200 OK` (on a poll) response includes a `Retry-After` header, the client **must** respect it. This is the server's primary backpressure mechanism.
- **Rate Limiting:** Polling endpoints (`GET /operations/{id}`) **are** subject to API rate limits. Excessive polling (e.g., ignoring backoff) **will** result in a `429 Too Many Requests` error, which should also include a `Retry-After` header.

- **Header Format:** Per RFC 9110, `Retry-After` can be seconds (`Retry-After: 30`) or an HTTP-date (`Retry-After: Wed, 29 Oct 2025 16:30:00 GMT`). Clients **must** support both formats.

---

4. Explaining the Pattern: Operation Polling vs. Resource Polling

This section details *why* this LRO pattern (polling `/operations`) is the standard and *why* the common anti-pattern (polling the resource itself) is disallowed.

4.1 The Correct Pattern: Polling the Operation Resource

The core of this pattern is the separation of the *job's* state from the *resource's* state. The `GET /operations/{id}` endpoint is the **only** source of truth for the *job*.

This section defines what a client should expect from the *business resource* (`GET /resource/{id}`) while an operation is in progress.

4.1.1 Behavior of `GET /resource` During an LRO

This is the standard for what a `GET` on the resource itself must return while a DELETE operation is in progress.

1. **While In-Progress (Job status: "RUNNING"):**
2. The `GET /resource/{id}` endpoint **must** return `200 OK` with the full, normal resource representation.
3. **Caching:** The response **should** include `Cache-Control: no-store` or `max-age=10`.
4. **(Optional) Discoverability:** The response **may** include a `Link: </operations/{operationId}>; rel="operation"`.
5. **(Optional) Lifecycle State:** You **may** add a *minimal, read-only* flag: `"lifecycleStatus": "DELETING"`. Clients **must not** poll this field for completion.
6. **After Completion (Job status: "SUCCEEDED"):**

7. `GET /resource/{id}` **must** return `404 Not Found` or `410 Gone`.

4.1.2 Policy: 410 Gone vs. 404 Not Found

- `410 Gone` is **preferred** for a short "tombstone" period (e.g., 1-24 hours) after a successful DELETE. It asserts the resource *did* exist. The response body **may** include an `LpdpProblem` body with the `deletedAt` time and `operationId` in the `meta` block.
- After the tombstone TTL, the server **should** revert to returning a standard `404 Not Found`.

4.1.3 Concurrency During Deletion

While a resource is being deleted (LRO status: "RUNNING"), all access to that resource must be handled explicitly.

Request	**Expected Response**	**Error Code (code)**
`GET /resource/{id}`	**200 OK**	N/A
`PUT` or `PATCH /resource/{id}`	**409 Conflict** or **423 Locked**	`RESOURCE_LOCKED`
`DELETE /resource/{id}` (duplicate)	**202 Accepted**	N/A (Returns original `Location` header)
`POST` (dependent sub-resources)	**409 Conflict**	`OPERATION_IN_PROGRESS`

4.2 The Anti-Pattern (Disallowed): Polling the Business Resource

A common but flawed alternative is to return `202 Accepted` and then have the client poll `GET /resource/{id}`, waiting for its state to change (e.g., for `status` to become `DELETED` or for it to return `404`). This anti-pattern is **disallowed** for the following reasons.

4.2.1 Flaw: Poor Observability

- **Problem:** The LRO pattern provides a "job log" (the operation resource). The anti-pattern only provides the final state of the resource. You cannot debug a failed process by only looking at the final object.

- **Example (Anti-Pattern):**
- A `DELETE /ports/123` fails. The team adds a `status: "DELETE_FAILED"` field to the `GET /ports/123` response.
- **The Issue:** An operations team tries to debug. They have no idea *why* it failed (the error message doesn't belong on the port model). They don't know *when* the delete was attempted, *how long* it ran before failing, or *which* trace ID to look up. They can't answer: "What is our p95 time-to-delete?" or "What is our error rate for port deletions?"
- **Example (LRO Pattern):**
- The `GET /operations/op-abc` response shows `status: "FAILED"`, `startedAt: "...", completedAt: "..."`, and a full `error` object with `code: "PARTNER_TIMEOUT"` and `meta: { "traceId": "..." }`.
- **The Benefit:** The operations team can now build a dashboard. They can query `GET /operations?status=FAILED` to find all failures. They can aggregate `errors[0].code` to see the top-most failure reason. They can compute exact durations (`completedAt - startedAt`) to get p95 latency SLOs.

4.2.2 Flaw: Security Risk & Information Leaks

- **Problem:** The resource (e.g., `/ports/{id}`) and the operation (e.g., `/operations/{op-abc}`) often have different security audiences. The resource is for *anyone* who can read it. The operation's *result* is often only for the *initiator*.
- **Example (Anti-Pattern):**
- To "fix" the observability flaw, a team adds an `errorInfo: "Downstream partner 'XYZ-Corp' timed out (trace-id: 99a-88b)"` field to the `GET /ports/123` response.
- **The Risk:** A low-privilege, read-only user from a *different tenant* (who just has read-access to the port) now polls the resource. They have just learned:
    1. A delete was attempted (which they shouldn't know).
    2. The internal name of our downstream partner (`XYZ-Corp`).

3. An internal `trace-id` they can use for other attacks.
4. This is a major information leak.
5. **Example (LRO Pattern):**

- The read-only user polls `GET /ports/123` and sees the normal `200 OK` response. They have no idea a delete is in progress.

- The initiator (with correct auth) polls `GET /operations/op-abc` and gets the `FAILED` status and error.
- The read-only user tries to access `GET /operations/op-abc` and gets a `404 Not Found` (or `403 Forbidden`). The sensitive error details are protected.

4.2.3 Flaw: High Server Load

- **Problem:** A `GET` on a business resource (e.g., `/ports/{id}`) is often a "heavy" query. A `GET` on an operation is a "light" query. Forcing clients to poll a heavy endpoint creates unnecessary load.

- **Example (Anti-Pattern):**

- A client polls `GET /ports/123` every 5 seconds. This endpoint is "heavy" because it's designed for UI/customer use. To build the response, the server must:

    1. Join 5 tables in the database (port details, customer info, location data, child circuits, config).
    2. Call 2-3 downstream microservices.
    3. Serialize a large 100KB JSON payload.
    4. A thousand clients polling this every 5 seconds creates massive read load on the main business database and downstream services.
    5. **Example (LRO Pattern):**

- A client polls `GET /operations/op-abc` every 5 seconds. This endpoint is "light" because it's *designed* for polling.

- The server does one thing: `SELECT * FROM operations_table WHERE id = 'op-abc'` (or even better, a Redis `GET`).
- It returns a tiny 1KB JSON payload (`{ "status": "RUNNING", "percent": 30 }`).

- This has almost zero impact on the core system. Furthermore, the server can use the `ETag` header. If the status hasn't changed, it returns `304 Not Modified` (an empty body), saving bandwidth and processing.

---

5. Applying the Pattern to Workflows

5.1 Individual Example (DELETE)

1. Client: `DELETE /connections/con-123`
2. Server: → `202 Accepted`, `Location: /operations/op-abc`
3. Client: `GET /operations/op-abc` (polls until terminal)
4. Server: → `200 OK`, `{"status": "FAILED", "error": {...}}`

5.2 Bulk / Wholesale Example

This LRO pattern is the *same* one used for bulk interfaces; no new pattern is needed. The operation resource simply contains a list of sub-operations. (Note: Detailed bulk API design is a separate topic).

- **Partial Failure Policy:** The overall operation `status` reflects the batch:
- `SUCCEEDED`: All sub-operations succeeded.
- `FAILED`: *Any* sub-operation failed.
- **Client Resubmission:** The client is responsible for inspecting the `subOperations` array in the response and re-submitting *only* the items that `FAILED`, using new `Idempotency-Key` headers for the new batch request.
- **Limits:** Services **must** define and enforce:
- A maximum number of items per bulk request (e.g., 200).
- A per-tenant concurrent operation limit (e.g., 10 active `RUNNING` operations).

---

6. Governance Requirements

All LRO-enabled APIs **must** adhere to these standards.

6.1 Idempotency

- **Requirement:** All LRO-initiating requests (POST, PUT, PATCH, DELETE) **must** require an `Idempotency-Key` header.
- **Scope:** The key's uniqueness is scoped *per tenant* and *per operation*.
- **Lifetime:** The server **must** cache the idempotency key and its result for a minimum of 24-72 hours.

- **Behavior:**

- If a request is retried *while the operation is active* (RUNNING), the server **must** return the `202 Accepted` with the *same* `Location` header.

- If a request is retried *after the operation is terminal* (SUCCEEDED/FAILED), the server **must** return the final stored result (e.g., `200 OK` with the final op body, or `410 Gone` if the resource was deleted) until the idempotency key expires.

6.2 Cancellation

- **Mechanism:** Clients **may** request cancellation by sending `DELETE /operations/{id}`.
- **Allowable States:** Cancellation can only be attempted on operations in `RUNNING` state.

- **Response:**

- `202Accepted`: The cancellation request was accepted (best-effort). The client must poll the operation, which will eventually transition to `CANCELLED` or `FAILED`.

- `409 Conflict`: The operation cannot be cancelled. The server must return an application/problem+json response body. The detail or message should state: "The operation is already in a terminal state (SUCCEEDED, FAILED, CANCELLED) and cannot be cancelled, or cancellation is not supported for this RUNNING operation."
- `404 Not Found`: The operation is unknown or has expired.

6.3 Security & Tenancy

- **Visibility:** An operation resource `GET /operations/{id}` **must** be protected by the same authorization rules as the resource it's acting upon. An operation **must only** be visible to the initiator, resource owners, or tenant admins.
- **Listing:** `GET /operations` **must** be filtered by tenancy. A user must **never** see operations from another tenant.
- **Data Redaction:** Error messages (`message` and `detail`) **must not** contain PII, secrets, or internal stack traces.

6.4 Rate Limiting & Quotas

- **Requirement:** All LRO endpoints **must** be protected by rate limits.
- **Limits:** Services **must** enforce limits on:
- Max concurrent `RUNNING` operations per tenant.
- Max items per bulk request.
- Polling frequency.
- **Response:** When a limit is exceeded, the server **must** return `429 Too Many Requests` with a `Retry-After` header.

6.5 Error Taxonomy (Problem Details)

All errors **must** use `application/problem+json` and our `LPDP-Mini v1.0` profile. The `code` is the canonical identifier.

Error Code (code)	**HTTP Status**	**Meaning**
`OPERATION_EXPIRED`	404	Polled an operation ID that is past its TTL.
`RESOURCE_LOCKED`	409 / 423	PUT/PATCH failed, delete in progress.
`OPERATION_IN_PROGRESS`	409	POST (sub-resource) failed, parent delete in progress.
`VALIDATION_ERROR`	400	The initial request body failed validation.
`PARTNER_TIMEOUT`	504	A downstream system failed to respond in time.

`RATE_LIMIT_EXCEEDED`	429	Too many requests or concurrent operations.

6.6 Caching & Headers

- `GET /operations/{id}`: **Must** return `Cache-Control: no-store`. **Should** support `ETag` and `If-None-Match` to allow clients to poll efficiently and receive a `304 Not Modified` if the status hasn't changed.
- `GET /resource/{id}` (during delete): **Must** return `Cache-Control: no-store` or a very small `max-age` (e.g., 10 seconds).

6.7 Operation Resource Lifecycle

Phase	**Description**	**Response Code**
Active	Operation created (PENDING, RUNNING).	200 OK
Terminal	Operation reaches SUCCEEDED, FAILED, or CANCELLED.	200 OK (for the duration of the TTL)
Expired (purged)	Operation record past retention TTL (7-30 days).	404 Not Found with `code: "OPERATION_EXPIRED"`
Manually deleted	Client performs `DELETE /operations/{id}`.	202 Accepted, then 404 Not Found

---

7. Industry References (for Architects)

- **Google Cloud:** AIP-151 – Long-Running Operations
- **Microsoft Azure:** Asynchronous Operations
- **RFC 9457:** Problem Details for HTTP APIs

---

8. OAS 3.1 Snippets (For Implementation)

8.1 LRO Initiator (e.g., Async DELETE

```
paths:
  /resources/{id}:
    delete:
      summary: Deletes a resource asynchronously
      parameters:
        - in: header
          name: Idempotency-Key
          required: true
          schema: { type: string, minLength: 8 }
      responses:
        '202':
          description: Accepted; poll the operation resource for status.
          headers:
            Location:
              schema: { type: string, format: uri }
              description: The URL of the /operations/{operationId} resource
            Retry-After:
              schema: { type: integer, example: 10 }
        '404': # ... standard errors
        '409': # ... concurrency/lock error
        '429':
          description: Rate limit exceeded (e.g., too many concurrent ops).
          headers:
            Retry-After: { schema: { type: integer, example: 60 } }
          content:
            application/problem+json:
              schema: { $ref: '#/components/schemas/LpdpProblem' }
        '503':
          description: Service Unavailable (e.g., job queue is full).
          headers:
            Retry-After: { schema: { type: integer, example: 120 } }
          content:
            application/problem+json:
              schema: { $ref: '#/components/schemas/LpdpProblem' }
```

8.2 Get Operation Status (Polling)

```
paths:
  /operations/{operationId}:
```

```
    get:
      summary: Gets the status of a long-running operation.
      parameters:
        - in: header
          name: If-None-Match
          schema: { type: string }
          description: ETag for a 304 Not Modified response.
      responses:
        '200':
          description: Operation status.
          headers:
            ETag: { schema: { type: string } }
            Retry-After: { schema: { type: integer, example: 10 } }
          content:
            application/json:
              schema: { $ref: '#/components/schemas/Operation' }
        '304':
          description: Not Modified. Operation status has not changed.
        '404':
          description: Operation expired or not found.
          content:
            application/problem+json:
              schema: { $ref: '#/components/schemas/LpdpProblem' }
```

8.3 Cancel Operation

```
paths:
  /operations/{operationId}:
    delete:
      summary: Requests cancellation of an operation.
      responses:
        '202':
          description: Cancellation request accepted (best-effort).
        '404':
          description: Operation not found or expired.
        '409':
          description: Conflict. Operation is already in a terminal state.
```

---

9. Implementation & Observability

9.1 Time & Consistency

- **Timestamps:** All timestamps (`startedAt`, `completedAt`) **must** be in UTC and formatted as **ISO-8601**.
- **Consistency:** A small delay (eventual consistency) is permitted between the LRO terminal state (`SUCCEEDED`) and the resource state (`GET /resource/{id}` returning `404`). The `GET /operations/{id}` endpoint is **always** the source of truth for the *job's* status.

9.2 Observability & SLOs

- **Required Metrics:** Services **must** emit metrics for:
- `operation_started_total` (Counter, tagged by type, tenant)
- `operation_completed_total` (Counter, tagged by type, tenant, status: [SUCCEEDED, FAILED, CANCELLED])
- `operation_duration_seconds` (Histogram, tagged by type, status)
- **Tracing:** The `operationId` **must** be correlated with the distributed trace context (e.g., in `error.errors[0].meta.traceId`).
- **Target SLOs:** Teams **must** define and publish SLOs for LROs (e.g., p95 completion for DELETE < 5 minutes).

---

10. Optional Extensions: Event-Based Notifications

Polling is the default, but services **may** offer event-based notifications for advanced clients.

- **Methods:** Server-Sent Events (SSE), Webhooks, or Kafka topics.
- **Payload:** The event payload **should** match the Operation schema.
- **Webhooks:** If webhooks are used, the service **must** provide a way for clients to register a callback URL and **must** use an HMAC signature (e.g., in `Webhook-Signature` header) for verification.

API Standard: Bulk Data Transfer (Handling Massive Payloads)

1. Purpose & Scope

This standard is an extension of the **API Standard: Long-Running Operations (Async API Pattern)**.

The "Long-Running" guide solves the **TIME** problem (jobs > 60 seconds). This guide solves the **SIZE** problem (payloads > 5MB).

Together, they form the complete, mandatory pattern for any asynchronous job that processes bulk data.

2. The Problem (The Anti-Pattern)

Synchronous REST APIs and API Gateways (like Apigee) are designed for small, fast, metadata-based transactions (typically `< 5MB`).

Teams **MUST NOT** attempt to use synchronous API calls for bulk data jobs. This anti-pattern includes:

- Sending multi-megabyte JSON/XML payloads in a `POST` or `PUT`.
- Expecting a multi-megabyte JSON/XML payload from a `GET`.
- Requesting gateway timeouts greater than the 60-second default.

This practice creates critical stability, security, and resource-exhaustion risks for the entire platform, affecting all other services. **Requesting longer timeouts or larger payload limits is not the correct solution.**

3. The Standard Pattern (Decoupled "Claim Check")

For any operation that involves a request or response payload larger than the 5MB gateway limit, the **Decoupled Data Transfer** pattern described below **MUST** be used.

This pattern separates the small, fast *control plane* (the API call) from the large, slow *data plane* (the file transfer). The API Gateway **MUST** only handle the control plane. The data plane is delegated to the cloud storage provider.

Use Case A: Bulk Request (Massive Upload)

This flow is used when a client needs to *send* a massive file to an API.

1. **Step 1: Client Requests Upload URL (Control Plane)**

The client makes a small, fast, and authenticated API call to the gateway to request a secure upload location.

`POST /v1/reports`

(See Metadata Schema below for required request/response formats) 2. **Step 2: API Responds with Pre-Signed URL (Control Plane)**

The API service generates a secure, one-time-use upload URL from GCS/S3 or any cloud storage bucket and returns it to the client.

`{ "fileId": "abc-123-xyz", "preSignedUploadUrl": "[https://storage.googleapis.com/bucket/upload?signature=..." }` 3. **Step 3: Client Uploads File (Data Plane)**

The client uploads the massive payload directly to the preSignedUploadUrl. This traffic bypasses the API Gateway. 4. **Step 4: Client Commits the File (Control Plane)**

Once the upload is complete, the client makes a second small API call to notify the service that the file is ready for processing.

`POST /v1/reports/abc-123-xyz/process` 5. **Step 5: API Responds Asynchronously (Control Plane)**

The API Gateway now follows the API Standard: Long-Running Operations, returning an HTTP 202 Accepted with a statusUrl.

```
{ "jobId": "job-999", "status": "Queued", "statusUrl":
"/v1/reports/status/job-999" }
```

Use Case B: Bulk Response (Massive Download)

This flow is used when a client makes a request that *produces* a massive result.

1. **Step 1: Client Starts the Job (Control Plane)**

The client makes a small, fast POST request to start the job, per the Long-Running Operations standard.

`POST /v1/reports/generate` 2. **Step 2: API Responds Asynchronously (Control Plane)**

The API Gateway instantly returns an HTTP 202 Accepted with a statusUrl. The 60-second timeout problem is solved.

`{ "jobId": "job-777", "status": "InProgress", "statusUrl": "/v1/reports/status/job-777" }` 3. **Step 3: Server Produces File (Background Work)**

The backend service runs the 5-minute query and saves the massive payload as a file in a temporary cloud storage bucket, adhering to the security requirements below. 4. **Step 4: Client Polls for Status (Control Plane)**

The client polls the statusUrl.

`GET /v1/reports/status/job-777` 5. **Step 5: API Responds with Result URL (Control Plane)**

Once the job is done, the API returns a small JSON payload containing a new, pre-signed download URL for the file in GCS/S3 or any cloud storage bucket. The massive payload problem is solved.

200 OK

`{ "jobId": "job-777", "status": "Complete", "resultUrl": "[https://storage.googleapis.com/bucket/download/report-777.csv?signature=..."](https://storage.googleapis.com/bucket/download/report-777.csv?signature=...") }` 6. **Step 6: Client Downloads File (Data Plane)**

The client follows the resultUrl to download the massive file directly from the cloud storage bucket. This traffic bypasses the API Gateway.

Use Case C: Failure Handling & Retries

The process is designed to be resilient.

Upload Failure Logic (Use Case A)

If the upload in **Step 3** fails (e.g., network error, URL expires), the `preSignedUploadUrl` is considered invalid.

- **DO NOT** retry the `PUT` to the same failed URL.
- **DO** restart the process from **Step 1** to request a **brand new** `preSignedUploadUrl`.

Download Failure Logic (Use Case B)

If the download in **Step 6** fails (e.g., network error, `resultUrl` expires), the job itself (`job-777`) is **still complete**.

- **DO NOT** queue a new job (do not call `POST /v1/reports/generate` again).
- **DO** restart the process from **Step 4** and call the `statusUrl` (`GET /v1/reports/status/job-777`) again.
- The server will see the job is "Complete" and generate a **brand new, fresh** `resultUrl` with a new expiration, allowing the client to retry the download.

4. Security Requirements

All services implementing this pattern **MUST** adhere to the following security controls:

- **Short Expiry (TTL):** Pre-signed URLs MUST be generated with the shortest possible Time-to-Live (TTL).
- **Upload URL (Use Case A):** Recommended TTL is **15 minutes**.
- **Download URL (Use Case B):** Recommended TTL is **60 minutes**.
- **Scoped Access:** The URL MUST be scoped to the *minimum* required operation. For `preSignedUploadUrl`, this MUST be `PUT`. For `resultUrl`, this MUST be `GET`. List/Delete/Write permissions are forbidden.

- **Encryption:** All data at rest in the storage bucket (even temporary data) **MUST** be encrypted using server-side encryption (AES-256 or equivalent).
- **Auditing:** The API service **MUST** log the generation of every pre-signed URL, including the associated `fileId`/`jobId` and the authenticated client context (ideally including the W3C `trace_id`).

5. Metadata Schema Requirements

The JSON payloads for the control plane (Steps 1, 2, 5) MUST be part of the API's formal OpenAPI 3.1 specification.

Example Schema for Upload Response (Step 2)

```
{
  "type": "object",
  "properties": {
    "fileId": {
      "type": "string",
      "description": "The unique identifier for the uploaded file.",
      "example": "abc-123-xyz"
    },
    "preSignedUploadUrl": {
      "type": "string",
      "format": "uri",
      "description": "The temporary URL to PUT the file to."
    }
  },
  "required": ["fileId", "preSignedUploadUrl"]
}
```

Example Schema for Download Response (Step 5)

This schema defines the successful response (`status: "Complete"`) and the failed job response (`status: "Failed"`).

```
{
  "type": "object",
  "properties": {
    "jobId": {
      "type": "string",
```

```
      "example": "job-777"
    },
    "status": {
      "type": "string",
      "enum": ["InProgress", "Complete", "Failed"],
      "example": "Complete"
    },
    "resultUrl": {
      "type": "string",
      "format": "uri",
      "description": "Present only when status is 'Complete'. This is the pr
    },
    "error": {
      "description": "Present only when status is 'Failed'. Follows the LPDP
      "$ref": "#/components/schemas/LpdpProblem"
    }
  },
  "required": ["jobId", "status"]
```

Dependent Schemas (Error Profile)

The following schemas are required to support the `error` property above, per the **API Standard: Lumen Problem Details (LPDP-Mini v1.0)**.

```
{
  "components": {
    "schemas": {
      "LpdpProblem": {
        "type": "object",
        "additionalProperties": false,
        "description": "Minimal Problem-Details envelope (RFC 9457-compliant
        "properties": {
          "title": {
            "type": "string",
            "description": "Short, human-readable summary."
          },
          "detail": {
            "type": "string",
            "description": "Optional longer explanation."
          },
          "errors": {
            "type": "array",
            "minItems": 1,
            "items": { "$ref": "#/components/schemas/LpdpError" }
          }
```

```
      },
      "required": [ "title", "errors" ]
    },
    "LpdpError": {
      "type": "object",
      "additionalProperties": false,
      "description": "Individual error item.",
      "properties": {
        "code": {
          "type": "string",
          "description": "Stable, machine-readable identifier (string form
        },
        "message": {
          "type": "string",
          "description": "Human-readable explanation of the issue."
        },
        "meta": {
          "type": "object",
          "description": "Optional unregulated extension for machine-reada
          "additionalProperties": true
        }
      },
      "required": [ "code", "message" ]
    }
  }
}
```

6. Related Standards

This standard MUST be implemented in conjunction with the following governance standards:

- **API Standard: Long-Running Operations**: This is a required component for the asynchronous flow.
- **API Standard: Lumen Problem Details (LPDP-Mini v1.0)**: This defines the mandatory error structure for a failed job.
- **API Standard: Pagination**: Teams should first attempt to use standard pagination to handle large *list-based* responses. This Bulk Data standard is for *single-object* massive payloads (like a generated file).

7. Automation Notes (for Governance Tools)

To support automated governance and enforce this standard, the following checks MUST be implemented in our tooling.

1. Design-Time Check (OAS & Metadata)

This check validates the API's *design* and *metadata* to ensure the correct pattern is being used.

- **Metadata Usage:** Developers **MUST** add a `tags` array containing `"bulk-data"` to any OpenAPI operation that follows this standard.
- **Automation Rule:** The Spectral linter in the CI pipeline MUST enforce the following:
- If `tags` contains `"bulk-data"`, the operation **MUST NOT** have a synchronous `200 OK` or `201 Created` response with a `content` body (it must be an async `202 Accepted`).
- If `tags` contains `"bulk-data"`, the operation **MUST** reference the standard schemas defined in this guide (e.g., for `preSignedUploadUrl` or `resultUrl`).

2. Build-Time Check (CI Pipeline & Gateway Config)

This check is the "hard guardrail" that validates the *as-built configuration* during deployment.

- **Automation Rule:** The CI/CD pipeline MUST scan the API Gateway (Apigee) proxy configuration files.
- **Action:** The build **MUST FAIL** if the configuration attempts to set the payload limit above the **5MB** enterprise standard.
- **Exception:** Any request for an exception (e.g., for a legacy system) MUST be flagged for manual review and approval by the API Governance Council.

8. Industry Precedent (The "Why")

This is the standard, non-negotiable pattern used by all major cloud providers to protect their platforms. They all separate the metadata API from the data plane.

- **Amazon (AWS):** Uses **S3 Pre-Signed URLs** to allow temporary, secure access to storage buckets.
- **Google (GCP):** Uses **Cloud Storage Signed URLs** for the same purpose.
- **Microsoft (Azure):** Uses **Shared Access Signatures (SAS)** to provide delegated access to storage.

By adopting this standard, we are aligning with industry best practices for building stable, secure, and scalable services.

API Standard: Date & Time Naming

1. Purpose & Scope

This standard defines how **date and date-time fields** must be named, structured, and represented in API payloads.
Consistent date naming and typing improves discoverability, reduces ambiguity, and enables automated validation through governance tools such as **Spectral**.

This convention applies to:

- All API resource representations (request and response payloads)
- Both system metadata (e.g., auditing fields) and business fields (e.g., expiration, activation)
- All date and date-time fields defined in OpenAPI specifications

---

2. Core Rules

2.1 Field Naming Suffixes

Suffix	Meaning	OpenAPI Format	Example
`_at`	Timestamp with time component (RFC 3339 `date-time`)	`format: date-time`	`created_at`, `updated_at`, `starts_at`
`_on`	Calendar date only (RFC 3339 `date`)	`format: date`	`effective_on`, `expires_on`
`*_from` / `*_until`	Time or date ranges	`format: date-time` or `date`	`valid_from`, `valid_until`

> ⚠️ The OpenAPI schema **MUST** align the suffix to its proper format:
> `_at → format: date-time`, `_on → format: date`.

2.2 State vs Schedule Semantics

Use timestamp names that clearly express **intent** — whether the value represents a **state change** (what has happened) or a **scheduled event** (what will happen).

Category	Description	Examples	Verb Tense	Mutability
State-based	Reflect past events that changed system state. Used for audit, versioning, or ordering.	`created_at`, `updated_at`, `deleted_at`, `completed_at`	Past	Immutable
Schedule-based	Define future or ongoing timeframes within a business process.	`starts_at`, `ends_at`, `expires_at`, `effective_on`, `renews_on`	Present / Future	Mutable
Range-based	Define valid windows for	`valid_from`, `valid_until`	Mixed	Usually Mutable

Category	Description	Examples	Verb Tense	Mutability
	temporal entities.			

💬 **Clarification:**

- *State fields* answer "**When did this happen?**"
- *Schedule fields* answer "**When will or should this occur?**"

2.3 Examples

✅ Recommended

```
{
  "created_at": "2025-10-12T15:45:33Z",
  "updated_at": "2025-10-15T19:02:11Z",
  "starts_at": "2025-11-01T00:00:00Z",
  "ends_at": "2025-11-30T23:59:59Z",
  "valid_from": "2025-11-01T00:00:00Z",
  "valid_until": "2026-01-01T00:00:00Z"
}
```

❌ Not Recommended

```
{
  "created": "2025-10-12",
  "modification_date": "2025-10-15T19:02:11Z",
  "start_date": "2025-11-01",
  "expire_at": "2025-11-30"
}
```

Issues:

- Mixed and inconsistent suffixes (`_date`, `_at`)
- Some fields missing time component where expected
- Ambiguous intent (is `expire_at` a schedule or state?)

---

3. Governance & Enforcement (Spectral Rule)

The following Spectral rule enforces this convention during API design-time validation.

Rule ID: `lumen-date-field-naming`

```
rules:
  lumen-date-field-naming:
    description: "Date/time fields must use '_at' for RFC 3339 date-time and
    message: >-
      "{{property}} field name does not follow Lumen Date/Time naming conven
       Use '_at' for RFC 3339 date-time, '_on' for RFC 3339 date.
       For ranges, use *_from / *_until."
    given: "$.components.schemas.*.properties.*"
    severity: warn
    then:
      - function: pattern
        field: "@key"
        functionOptions:
          match: ".*(_at|_on|_from|_until)$"
      - function: truthy
        field: "format"
      - function: enumeration
        field: "format"
        functionOptions:
          values:
            - date
            - date-time
```

✅ **What it checks:**

- Ensures all date-related fields end with `_at`, `_on`, `_from`, or `_until`.
- Validates that `_at` fields use `format: date-time`.
- Validates that `_on` fields use `format: date`.
- Flags inconsistent or ambiguous names (`created`, `start_date`, `expire_at`).

---

4. Automation Notes

- Integrate this rule into the **Spectral ruleset** used by the API linter in the CI pipeline.
- Apply it to all repositories that define OpenAPI 3.x specifications.
- Violations are classified as **warnings** (initial rollout) and will be escalated to **errors** once adoption reaches >80%.

---

5. Related Standards

- RFC 3339: Date and Time on the Internet

API Standard: GET Method

Purpose

The `GET` method retrieves a resource or a collection of resources.
It **MUST NOT** change server state and **MUST** be safe and idempotent as defined in RFC 9110.

This page defines how to design, document, and implement `GET` operations across Lumen APIs to ensure predictable, cache-friendly, and standards-compliant behavior.

Semantics

Property	Requirement
Safe	✅ YES, never modifies data or triggers side effects.
Idempotent	✅ YES, repeating the same GET yields the same result (unless data has changed).
Request Body	❌ NOT ALLOWED.
Response Body	✅ REQUIRED for `200 OK`, MUST represent the requested resource(s).
Caching	✅ Strongly RECOMMENDED, use `ETag` and `Cache-Control`.

URI Design

Examples

- Single resource: `/v1/customers/{customer_id}`
- Collection: `/v1/customers`
- Filtered collection: `/v1/customers?status=active&country=US`

MUST

- Use plural nouns for collections.
- Avoid action verbs (`/getCustomer`) — the HTTP verb already defines intent.
- Preserve canonical noun hierarchy (**no verbs, no RPC style**).

Singleton Resources

Definition

A *singleton resource* represents a unique conceptual instance of a type that exists once per context (e.g., current user profile, organization settings, service status).
Singletons do not require an identifier or belong to a collection.

Examples

```
GET /v1/profile               → current user profile
GET /v1/organization/settings  → organization configuration
GET /v1/configuration          → global configuration
GET /v1/status                 → system status or health
```

Design Guidance

Rule	Description
MUST	Treat singletons as first-class nouns (`/profile`, `/settings`, `/status`).
MUST NOT	Nest singleton resources under collections **if they represent contextual or global concepts** (e.g., avoid `/users/{id}/profile` for the authenticated user's profile). Such

Rule	Description
	resources have a single instance determined by the caller's identity or environment — not by path variables — and therefore **belong at the root level** (e.g., `/profile`).
MAY	Nest singleton resources **only when they are scoped to a parent resource** (e.g., `/organizations/{orgId}/settings` — one settings resource per organization).
MUST	Support `GET` for retrieval and `PATCH` for updates (if mutable). Never use `POST` or `DELETE`.
MUST	Return `200 OK` with a body when available; `404 Not Found` if temporarily unavailable.
SHOULD	Include `ETag` and support `If-None-Match` for cache validation.
MUST NOT	Return `204` for existing singletons — always return a representation unless explicitly empty.

Example

```
GET /v1/profile
If-None-Match: "v12"

HTTP/1.1 200 OK
ETag: "v13"
Cache-Control: private, max-age=60
{
  "id": "u123",
  "name": "Jane Doe",
  "email": "jane@example.com"
}
```

```
GET /v1/profile
If-None-Match: "v13"

HTTP/1.1 304 Not Modified
ETag: "v13"
```

Governance Note

Singletons must be documented as distinct resources (`/profile`, `/settings`,

`/status`) and enforced through Spectral rules to verify `ETag` and `If-None-Match` support.

Response Structure

Every successful `GET` MUST return a JSON object.

Single Resource

```
{
  "id": "c123",
  "name": "Jane Doe",
  "email": "jane@example.com",
  "created_at": "2025-10-20T12:34:56Z"
}
```

Collection

```
{
  "data": [
    { "id": "c123", "name": "Jane Doe" },
    { "id": "c124", "name": "John Smith" }
  ],
  "pagination": {
    "limit": 20,
    "offset": 0,
    "total": 245
  }
}
```

MUST

- Wrap collections in a `data` array to allow metadata (`pagination`, `links`) at the top level.
- Use consistent pagination keys (`limit`, `offset`, `total`).

Headers and Conditional Requests

ETag and Cache Validation

Header	Usage	Example
`ETag`	Server-generated version tag for the resource	`ETag: "v7a2"`
`If-None-Match`	Client cache validation header	`If-None-Match: "v7a2"`
Response on match	`304 Not Modified` (no body)	

MUST

- Return `ETag` on all cacheable responses.
- Support `If-None-Match` on all `GET` endpoints.
- Return `304 Not Modified` if ETag matches.

SHOULD

- Add `Cache-Control` (e.g., `private, max-age=60`) to define freshness.
- Include `Last-Modified` when available for weak validation.

Query Parameters and Filtering

Principle	Guidance
Predictability	Parameter names must be consistent across APIs (`status`, `sort`, `limit`, `offset`).
Multiple Filters	Use & separated query pairs.
Sorting	`?sort=+name` ascending, `?sort=-created_at` descending.
Pagination	Refer to pagination standards.
Empty Results	Return `200 OK` with an empty `data` array, not `404`.
Invalid Filters	Return `400 Bad Request` with Problem Details.

Status Codes

Code	Meaning	Applies To	Usage Notes
200 OK	Successful retrieval	✅ Single / ✅ Collection	Always return for successful GETs; include response body.
304 Not Modified	Resource unchanged since client cache	✅ Single / ✅ Collection	Used with `If-None-Match`, no body.
400 Bad Request	Invalid query or parameter	✅ Single / ✅ Collection	Syntactic or invalid filter errors.
401 Unauthorized	Authentication required or invalid	✅ Both	
403 Forbidden	Authorized but lacks permission	✅ Both	
404 Not Found	Resource not found	✅ Single	Never use for empty collections.
405 Method Not Allowed	HTTP verb not supported	✅ Both	
422 Unprocessable Content	Valid syntax but invalid semantic input	✅ Both	
500 Internal Server Error	Unexpected server failure	✅ Both	No stack traces exposed.

Example Request / Response Set

Request

```
GET /v1/customers/123
If-None-Match: "v5"
Accept: application/json
```

Response (Resource Updated)

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "v6"
Cache-Control: private, max-age=60
```

```
{
 "id": "123",
 "name": "Jane Doe",
 "email": "jane@example.com"
}
```

Response (Not Modified)

```
HTTP/1.1 304 Not Modified
ETag: "v6"
```

OAS Authoring Requirements

```
paths:
  /v1/customers/{id}:
    get:
      summary: Retrieve customer by ID
      parameters:
        - in: path
          name: id
          required: true
          schema: { type: string }
        - in: header
          name: If-None-Match
          schema: { type: string }
      responses:
        '200':
          description: Successful response
          headers:
            ETag:
              description: Entity tag for versioning
              schema: { type: string }
        '304':
          description: Not Modified
        '400': { description: Invalid request }
        '404': { description: Not found }
```

MUST

- Include `ETag` in `200` response headers.
- Include `If-None-Match` parameter.
- Restrict status codes to the approved matrix.
- Return `application/json`.

Governance Checklist

Check	Validation
✅ `GET` defined as safe and idempotent	No side effects
✅ `ETag` header returned	Required for cacheable responses
✅ `If-None-Match` supported	Enables `304 Not Modified`
✅ No request body allowed	Specification enforces it
✅ Proper status codes used	200/304/400/401/403/404/405/422/500 only
✅ OAS validated via Spectral rules	Governance automation ready

Developer Summary

- Use `GET` exclusively for read-only operations.
- Never require or process a body in `GET`.
- Always include `ETag` and handle `If-None-Match` for conditional caching.
- Return `200 OK` for success, `304` for cache hits, `404` only for missing single resources.
- Treat collection queries as successful even when empty.
- Keep response structure predictable (`data`, `pagination`).
- Follow the approved status-code matrix and Problem Details format.

API Standard: POST Method

Purpose

The `POST` method creates new resources or performs non-CRUD actions on existing resources.
It is **not inherently idempotent** and should be used when neither `PUT` nor `PATCH` semantics apply.

This page defines the required and recommended practices for all `POST` operations across Lumen APIs, ensuring consistency, predictability, and safe retries through **idempotency keys** and **standardized responses**.

Semantics

Property	Requirement
Safe	❌ No, may modify server state.
Idempotent	⚠ No by default; **MUST** use `Idempotency-Key` if retries are expected.
Typical Use	Resource creation or triggering an action.
Request Body	✅ Required, defines input parameters or resource representation.
Response Body	✅ Required, returns the created resource or operation result.

When to Use `POST`

Use Case	Example	Required Behavior
Create a new resource	`POST /v1/orders`	Returns `201 Created` + `Location` header for the new resource.
Create multiple resources (batch / composite operation)	POST /v1/customers/batch	Performs creation of multiple items in one request. Returns **207 Multi-Status** if results are mixed, or **202 Accepted** if processed asynchronously via a job. Batch requests SHOULD complete within the synchronous timeout budget (≤ 30 s p95 / 60 s absolute) or follow the **LRO pattern**.
Trigger an action on a resource	`POST /v1/invoices/{id}/void`	Returns `200 OK` or `202 Accepted` depending on sync vs. async.
Start an asynchronous or workflow operation	`POST /v1/network/connections/provision`	Returns `202 Accepted` + operation URL for status tracking.
Perform search or query with complex filters (non-CRUD)	`POST /v1/orders/search`	Allowed when query parameters are too complex for GET; must be safe and side-effect free.

Use Case	Example	Required Behavior
Submit sensitive criteria or large query payloads (PII, account numbers, tokens, advanced filters)	`POST /v1/customers/search` with body `{"email":"…","dob":"…"}`	Use `POST` with a **request body** instead of putting sensitive data in the URL. **MUST** be side-effect free. **MUST** set `Cache-Control: no-store`. **MUST NOT** log or echo sensitive fields. **MUST NOT** include PII or secrets in the path or query string. **SHOULD** return 413 `Payload Too Large` for oversized bodies and document limits. **SHOULD** include `Vary: Authorization` for authenticated results.

Non-Applicable Cases

Rule	Description
MUST NOT	Use `POST` for **singleton resources** (e.g., `/profile`, `/configuration`, `/status`). These resources are pre-defined by the system and not client-created. Use `GET` to retrieve and `PATCH` to update them.

Why the Sensitive-Data Rule Exists

- URLs appear in logs, browser history, and referrers; request bodies do not.
- Shared or transparent caches may store GET responses; using POST + `Cache-Control: no-store` minimizes risk.
- Request bodies allow schema-driven validation and redaction (`x-lumen-sensitive: true` in OAS).

Creation Semantics

Resource Creation

- **MUST** create new resource(s) under the collection URI.
  Example:
  `POST /v1/customers` → creates a customer resource.
- **MUST** return:
- `201 Created`
- A `Location` header with the canonical URI of the new resource.
- The newly created resource in the response body.
- **MUST NOT** use `POST` to **replace** an existing resource; use `PUT` instead.
- **MAY** return `202 Accepted` for **asynchronous creation**
- **MAY** return `207 Multi-Status` if a **batch request** partially succeeds.

Example:

```
POST /v1/customers
Content-Type: application/json
{
  "name": "Jane Doe",
  "email": "jane@example.com"
}

HTTP/1.1 201 Created
Location: /v1/customers/c123
Content-Type: application/json
{
  "id": "c123",
  "name": "Jane Doe",
  "email": "jane@example.com"
}
```

Action Semantics

When performing an action on an existing resource that does not map to a CRUD operation:

Rule	Description
Pattern	`POST /{resource}/{id}/{action}` (e.g., `/invoices/{id}/approve`).
Response	`200 OK` for synchronous result, `202 Accepted` for async.
Body	MAY contain parameters relevant to the action.
Idempotency	MUST support `Idempotency-Key` to handle retries safely.

Example:

```
POST /v1/invoices/123/void
Idempotency-Key: 4af7e7d9-91a6-4823-a1c0-8d112c884cb5
HTTP/1.1 200 OK
Content-Type: application/json
{
  "id": "123",
  "status": "voided"
}
```

Idempotency

`POST` is **not idempotent by default** — but Lumen APIs must provide an idempotent experience when requests are retried.

Header	Description
`Idempotency-Key`	A client-supplied opaque UUID identifying the request.
Scope	Applies to unsafe POSTs that could be retried.
Server Behavior	Deduplicate requests with the same key and identical payload; replay original result.
Retention	Store key and result for a minimum of 24 hours.
Conflict Handling	Return `409 Conflict` if the same key is reused with a different payload.

Example:

```
POST /v1/payments
Idempotency-Key: 827dcf3e-44fb-4f07-94b3-6b47cf3b813d

→ 201 Created

# Retry (same key)
→ 201 Created (identical response)

# Retry (same key, different payload)
→ 409 Conflict
```

Headers

Header	Usage
`Idempotency-Key`	Ensures retry-safe behavior.
`Location`	Points to newly created resource or operation resource.
`Retry-After`	Used when returning `202 Accepted` for async processing.
`Content-Type`	MUST be `application/json` or vendor subtype.
`Accept`	MUST be `application/json` (or compatible).

Status Codes

Code	Meaning	Applies To	Notes
201 Created	Resource successfully created	✅ Resource creation	MUST include `Location` header and representation.
202 Accepted	Request accepted for asynchronous processing	✅ LRO / async	MUST include operation URL in `Location`.

Code	Meaning	Applies To	Notes
207 Multi-Status	Batch/composite request produced mixed results	✅ Bulk / composite POSTs	Use when some items succeed and others fail, body MUST include per-item statuses and summary.
200 OK	Successful synchronous action	✅ Resource actions	Used only for non-creation actions.
400 Bad Request	Invalid request body or parameter	✅ Both	Syntax or schema violation.
401 Unauthorized	Authentication required	✅ Both	
403 Forbidden	Authenticated but lacks permission	✅ Both	
404 Not Found	Target resource or parent collection missing	✅ Both	
405 Method Not Allowed	Wrong HTTP method used	✅ Both	
409 Conflict	Resource already exists or idempotency key conflict	✅ Creation / Action	
422 Unprocessable Content	Semantic validation error	✅ Both	Schema valid but business rule failure.
500 Internal Server Error	Unexpected server failure	✅ Both	No internal details exposed.

OAS Authoring Requirements

When documenting POST operations in OpenAPI:

```
paths:
  /v1/orders:
    post:
      summary: Create an order
      parameters:
```

```
    - in: header
      name: Idempotency-Key
      schema: { type: string, maxLength: 255 }
  requestBody:
    required: true
    content:
      application/json:
        schema: { $ref: '#/components/schemas/OrderCreate' }
  responses:
    '201':
      description: Created
      headers:
        Location:
          schema: { type: string, format: uri }
    '202': { description: Accepted (async creation) }
    '207': { description: Partial success (batch results) }
    '400': { description: Invalid request }
    '409': { description: Conflict (duplicate or idempotency) }
    '422': { description: Validation failed }
```

MUST

- Include `Idempotency-Key` header definition.
- Include `Location` header for `201` or `202` responses.
- Constrain response codes to the approved matrix.
- Document LRO patterns (`202 + Location`) when applicable.
- Reference `207` for batch/composite POSTs if supported.

Governance Validation Checklist

Check	Expectation
✅ Correct verb usage	`POST` used only for create or action semantics
✅ Location header present	For `201` and `202` responses
✅ Idempotency supported	`Idempotency-Key` required for unsafe POSTs
✅ Proper status codes	Matches approved matrix
✅ Async workflows standardized	`202 + Location` pattern used
✅ OAS validated via Spectral	Governance enforcement ready

Developer Summary

- Use `POST` to **create** new resources or **trigger** actions — never for updates.
- Always return `201 Created` with a `Location` header for creations.
- Use `Idempotency-Key` to protect clients from duplicate processing.
- For asynchronous operations, return `202 Accepted` and an **operation URI**.
- For partial batch success, return `207 Multi-Status` with `results[]` and `summary`.
- Adhere strictly to the **approved status-code matrix**.
- Document every `POST` in OpenAPI with consistent headers and response structure.

API Standard: PUT Method

Purpose

The `PUT` method **replaces the full representation** of a resource at its canonical URI.
It is **idempotent** — identical requests produce the same final state.

This page defines Lumen's standards for designing, documenting, and implementing `PUT` operations, including creation semantics, conditional updates, and approved response codes.

Semantics

Property	Requirement
Safe	❌ No, modifies server state.
Idempotent	✅ Yes, repeating the same request yields the same result.
Typical Use	Full replacement of a resource.
Request Body	✅ Required, represents the complete, updated resource.

Property	Requirement
Response Body	✅ Required for `200 OK`, returns the latest representation.

When to Use `PUT`

Use Case	Example	Required Behavior
Replace an existing resource completely	`PUT /v1/customers/{id}`	Replaces the full customer object with the provided payload; returns `200 OK` and the updated resource.
Create a resource at a client-known URI (rare)	`PUT /v1/buckets/{id}`	MAY be used if the client controls ID generation. Return `201 Created` if the resource did not exist before.
Update large static configurations or documents	`PUT /v1/configurations/{id}`	Appropriate when sending an entire configuration file or template.
Atomic overwrite of immutable document	`PUT /v1/policies/{id}`	Old version replaced entirely, new ETag generated.

Singleton Resources

Definition

A **singleton resource** represents a unique instance that exists once per logical context (for example, `/configuration`, `/profile`, `/settings`, `/status`, `/organization/{id}/policy`).
It is not part of a collection. A singleton **may or may not pre-exist** , if it does not, the client can create it by `PUT`ting a full representation at its canonical URI.

Guidance

Rule	Description
MAY use PUT to create a singleton	When the resource has a known, fixed URI and does not yet exist. PUT is idempotent, repeating the same request yields the same result. Return **201 Created** on first creation, **200 OK** on subsequent replacements.
MUST	Require `If-Match` on updates when the singleton already exists to prevent concurrent overwrites.
MUST return 201 Created	When the singleton did not exist previously and has now been created.
MUST return 200 OK	When replacing an existing singleton with a new representation.
SHOULD include ETag	Every GET and PUT response for a singleton must include an ETag for version control.
MUST NOT	Use PUT for actions or only for full replacement. Use PATCH for partial updates.
MUST NOT	Allow DELETE on singleton endpoints.

Examples

Create (first-time PUT)

```
PUT /v1/organization/settings
Content-Type: application/json

{
  "auto_approve": true,
  "timezone": "America/Chicago"
}

HTTP/1.1 201 Created
ETag: "v1"
Location: /v1/organization/settings
```

Replace (subsequent update)

```
PUT /v1/organization/settings
If-Match: "v1"
Content-Type: application/json

{
  "auto_approve": false,
  "timezone": "America/Chicago"
}

HTTP/1.1 200 OK
ETag: "v2"
```

Conflict (stale ETag)

```
HTTP/1.1 412 Precondition Failed
Content-Type: application/problem+json
{
  "type": "https://api.lumen.com/errors/conflict",
  "title": "Precondition Failed",
  "status": 412,
  "detail": "The resource has been modified by another client."
}
```

Governance Note

- Singleton PUT operations follow the same idempotency, full-replacement, and ETag/If-Match rules as normal resources.
- Example Spectral validation:

```
rules:
  singleton-put-must-support-etag:
    given: "$.paths[?(@property.match(/^\\/v1\\/(configuration|profile|setti
    then:
      field: "responses['200'].headers.ETag"
      function: truthy
```

Summary

> Singleton resources can be created or replaced using PUT when they have a fixed URI and no existing representation.

Return 201 on first creation, 200 on subsequent updates.
Always include ETag and require If-Match for concurrency control.

General Rules

Rule	Description
MUST	Replace the **entire resource representation**; omitted fields are treated as deleted.
SHOULD NOT	Use PUT for partial updates — use PATCH instead.
MUST	Return the updated resource in the response body.
MUST	Preserve the resource identifier and canonical URI.
MUST NOT	Allow structural mutations (e.g., changing resource type or parent linkage).
MUST	Be **idempotent** — re-submitting the same payload must not create duplicates.

Conditional Requests & ETag

Header	Usage	Example
`ETag`	Returned with GET/PUT responses.	`ETag: "v12"`
`If-Match`	Required header for optimistic concurrency.	`If-Match: "v12"`

Rules

- **MUST** return a new ETag on every successful modification.
- **SHOULD** reject updates without If-Match when concurrent edits are possible.
- **MUST** return 412 Precondition Failed if If-Match does not match.
- **MUST NOT** accept stale updates.

Example

```
PUT /v1/customers/c123
If-Match: "v4"
Content-Type: application/json

{
  "id": "c123",
  "name": "Jane Doe",
  "email": "jane@example.com",
  "status": "active"
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "v5"
Cache-Control: private, max-age=60

{
  "id": "c123",
  "name": "Jane Doe",
  "email": "jane@example.com",
  "status": "active"
}
```

Status Codes

Code	Meaning	Applies To	Notes
200 OK	Resource successfully replaced	✅ Existing resource	MUST include updated representation.
201 Created	Resource newly created at client-specified URI or singleton first creation	✅ Creation	MUST include Location header.
204 No Content	Update succeeded, no body returned	⚠ Optional	Only if representation unchanged.

Code	Meaning	Applies To	Notes
400 Bad Request	Invalid or missing fields	✅ Both	Schema/validation errors.
401 Unauthorized	Authentication required	✅ Both	
403 Forbidden	Authenticated but not allowed	✅ Both	
404 Not Found	Resource does not exist	✅ Both	
405 Method Not Allowed	Wrong HTTP verb	✅ Both	
409 Conflict	Version/state conflict	✅ Both	For concurrent updates or integrity violations.
412 Precondition Failed	If-Match mismatch	✅ Both	Stale ETag.
422 Unprocessable Content	Semantically invalid	✅ Both	Valid JSON, violates business rule.
500 Internal Server Error	Unexpected server failure	✅ Both	No internal details exposed.

Headers

Header	Usage
`If-Match`	Required for conditional updates.
`ETag`	Returned in every successful PUT.
`Cache-Control`	MUST specify `private` or `no-store`.
`Content-Type`	MUST be `application/json`.
`Accept`	MUST be `application/json` or compatible.

OAS Authoring Requirements

```
paths:
  /v1/customers/{id}:
    put:
      summary: Replace a customer resource
      parameters:
        - in: path
          name: id
          required: true
          schema: { type: string }
        - in: header
          name: If-Match
          schema: { type: string }
      requestBody:
        required: true
        content:
          application/json:
            schema: { $ref: '#/components/schemas/Customer' }
      responses:
        '200':
          description: Successful replacement
          headers:
            ETag:
              description: Entity tag for versioning
              schema: { type: string }
        '201': { description: Created (first-time PUT) }
        '412': { description: Precondition failed (stale ETag) }
        '400': { description: Invalid request }
        '404': { description: Not found }
```

MUST

- Define `If-Match` and `ETag` headers.
- Restrict status codes to the approved matrix.
- Use `application/json`.

Governance Checklist

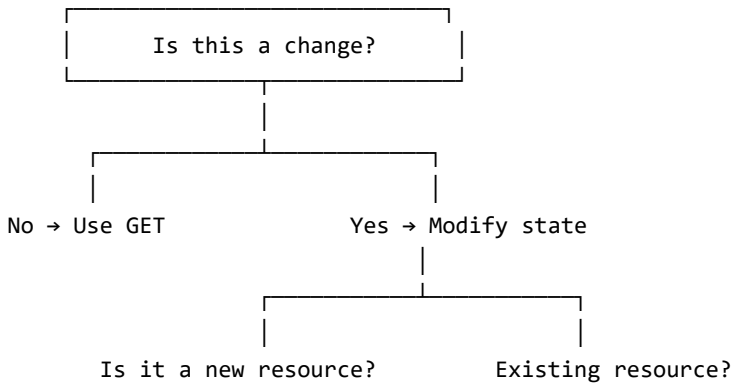
Check	Validation
✅ Idempotency guaranteed	PUT deterministically replaces the same resource

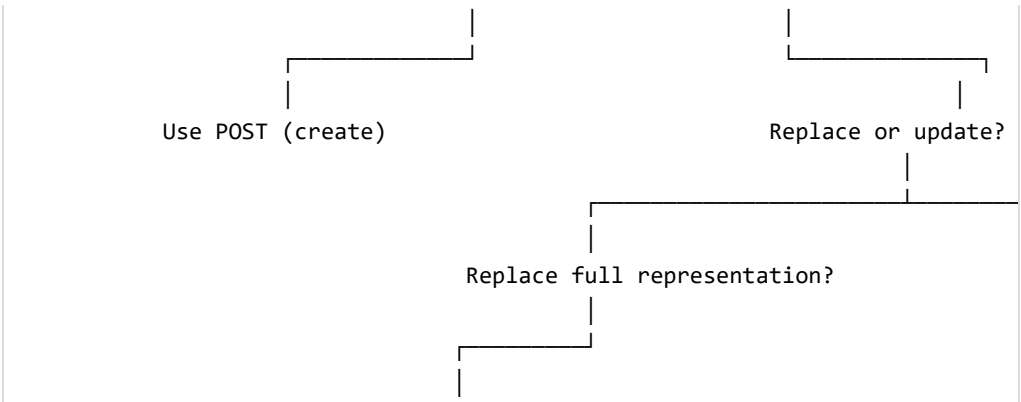
Check	Validation
✅ Full resource in body	Required
✅ Returns updated representation	Required
✅ ETag / If-Match implemented	Required for concurrency control
✅ Proper status codes used	200/201/204/400/401/403/404/405/409/412/422/500
✅ Singleton creation allowed via PUT	201 Created on first creation
✅ OAS validated via Spectral	Governance automation ready

Developer Summary

- Use **PUT** for full resource replacement or idempotent creation at a known URI.
- Include the entire object; missing fields imply deletion.
- Support **ETag** and **If-Match** for concurrency safety.
- Return `201 Created` on first creation, `200 OK` on updates.
- Ensure idempotency — repeated identical requests yield identical state.
- Use **PATCH** for partial updates.
- Follow the approved status-code matrix and RFC 7807 error model.
- For **singleton resources**, PUT can create or replace them at a fixed URI; always include ETag and If-Match.

Conditional Update Decision Tree





API Standard: PATCH Method

Purpose

The `PATCH` method partially updates an existing resource at its canonical URI. Unlike `PUT`, it does not require the full representation, only the fields being changed.

`PATCH` is **not inherently idempotent**, but can be made so when combined with `If-Match` and deterministic patch semantics.

This page defines Lumen's governance requirements and best practices for designing, documenting, and implementing `PATCH` operations across APIs.

Semantics

Property	Requirement
Safe	❌ No, modifies server state.
Idempotent	⚠ Not guaranteed, SHOULD use `If-Match` for concurrency control.
Typical Use	Partial modification of resource fields.
Request Body	✅ Required, describes changes to apply.
Response Body	✅ Required for `200 OK`, returns updated representation.

When to Use PATCH

Use Case	Example	Required Behavior
Modify selected fields	`PATCH /v1/customers/{id}`	Updates only given attributes; returns `200 OK` with updated object.
Change workflow state	`PATCH /v1/orders/{id}`	Updates only `status` or `stage`.
Partial configuration change	`PATCH /v1/configurations/{id}`	Applies incremental changes.
Partial update of singleton	`PATCH /v1/organization/settings`	Updates subset of fields in a singleton resource.

⚖️ Governance Note — When PATCH Is Business-Critical

> The PATCH method **MUST NOT** be used merely for developer convenience.
> Teams **SHOULD use** PATCH **only when partial updates are business-critical** — meaning that sending or overwriting the entire resource representation (via PUT) would introduce data-loss, concurrency, or operational issues.

PATCH is business-critical when:

- Multiple systems or users own different parts of a resource (e.g., shared ownership of a profile).
- Partial mutation prevents overwriting external or read-only fields.
- Resource size or update frequency makes PUT inefficient.
- Incremental updates are required for workflows, IoT configuration, or multi-step state transitions.

PATCH is *not* business-critical when:

- The resource is small and easily replaced with PUT.
- The update semantics are simple (single owner, no concurrency risk).
- PATCH is used only to avoid sending redundant fields.

Situation	Recommended Method	Rationale
Toggle a single user preference or flag	**PATCH**	Partial update is isolated and critical to UX.
Update multiple independent fields in a shared resource	**PATCH**	Prevents data overwrite from concurrent systems.
Replace a complete entity such as customer, invoice, or address	**PUT**	Simple and predictable full-replacement semantics.
Minor API optimization (no concurrency risk)	**PUT**	Avoids unnecessary PATCH complexity.

Singleton Resources

Definition

A singleton resource (e.g., `/configuration`, `/profile`, `/settings`, `/status`) represents a unique, addressable entity that may be mutable.

Guidance

Rule	Description
MAY use PATCH	For partial updates to existing or pre-created singletons.
MAY use PUT	When replacing the entire representation.
MUST include If-Match	To prevent overwriting concurrent changes.
MUST return 200 OK	With updated representation on success.
MUST return 412 Precondition Failed	If `If-Match` does not match current ETag.
SHOULD include ETag	Every GET/PATCH response must include new ETag.
MUST NOT	Use PATCH to create a singleton — creation uses PUT.

Example

```
PATCH /v1/organization/settings
If-Match: "v1"
```

```
Content-Type: application/json

{
  "auto_approve": false
}

HTTP/1.1 200 OK
ETag: "v2"
Content-Type: application/json
{
  "auto_approve": false,
  "timezone": "America/Chicago"
}
```

Patch Semantics

JSON Merge Patch (RFC 7396) — Preferred

Simplified document-based merge:
present keys → overwrite,
`null` → delete,
absent → no change.

```
PATCH /v1/customers/123
Content-Type: application/merge-patch+json

{ "status": "inactive" }
```

JSON Patch (RFC 6902) — Optional

Explicit, ordered operations for atomic control.

```
PATCH /v1/customers/123
Content-Type: application/json-patch+json

[
  { "op": "replace", "path": "/status", "value": "inactive" },
```

```
  { "op": "remove", "path": "/temporary_flag" }
]
```

⚖️ Governance Callout — PATCH Format Standards

> **Lumen Default Standard:**
> All Lumen APIs **MUST** implement `PATCH` using **JSON Merge Patch (RFC 7396)** with
> `Content-Type: application/merge-patch+json`.

Rationale

- Aligns with standard REST partial-update semantics.
- Easy for developers; payload mirrors resource shape.
- Fully idempotent when combined with `ETag + If-Match`.

Optional Alternative (Advanced Use Only)

`application/json-patch+json` (**RFC 6902**) **MAY** be supported for:

- Explicit `add` / `remove` / `replace` operations.
- Ordered multi-field or audited mutations.
- Complex structured documents.

Governance Enforcement

Rule	Description
MUST	Default `Content-Type = application/merge-patch+json`.
MAY	Support `application/json-patch+json` only after design review.
MUST NOT	Implicitly accept both without declaring them in OAS.
MUST	Validate PATCH media type via Spectral rule.

Spectral Example

```
rules:
  patch-must-use-merge-patch:
    description: "PATCH operations must default to application/merge-patch+j
    given: "$.paths[*].patch.requestBody.content"
```

```
    then:
      field: "application/merge-patch+json"
```

Summary

Aspect	Governance Standard
Default RFC	**RFC 7396 — JSON Merge Patch**
Optional RFC	RFC 6902 — JSON Patch
Default Content-Type	`application/merge-patch+json`
Optional Content-Type	`application/json-patch+json` (advanced)
Idempotency	`ETag` + `If-Match` required
Documentation	OAS MUST explicitly list supported types

Conditional Requests & Concurrency

Header	Usage	Example
`ETag`	Returned with GET/PATCH responses.	`ETag: "v10"`
`If-Match`	Required for optimistic concurrency.	`If-Match: "v10"`

Rules

- **MUST** return new ETag after successful PATCH.
- **MUST** reject updates without `If-Match` when concurrent writes possible.
- **MUST** return `412 Precondition Failed` on mismatch.
- **MUST NOT** overwrite concurrent updates.

Idempotency

`PATCH` is not guaranteed idempotent but SHOULD be made so where possible.

Technique	Description
ETag + If-Match	Ensures update applies only to current version.
Idempotency-Key	Optional for safe retry behavior.
Deterministic Patch	Same payload → same final state.

Example

```
PATCH /v1/customers/123
If-Match: "v4"
Content-Type: application/merge-patch+json

{
  "email": "jane.doe@example.com",
  "status": "inactive"
}

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "v5"

{
  "id": "123",
  "name": "Jane Doe",
  "email": "jane.doe@example.com",
  "status": "inactive"
}
```

Status Codes

Code	Meaning	Applies To	Notes
200 OK	Resource partially updated	✅ Existing	MUST return updated representation.
204 No Content	Update succeeded (no body)	⚠ Optional	Only if representation unchanged.

Code	Meaning	Applies To	Notes
400 Bad Request	Invalid JSON or schema	✅ Both	Syntax errors.
401 Unauthorized	Authentication required	✅ Both	
403 Forbidden	Authenticated but not permitted	✅ Both	
404 Not Found	Resource missing	✅ Both	
405 Method Not Allowed	Wrong verb	✅ Both	
409 Conflict	Version/state conflict	✅ Both	Patch cannot apply.
412 Precondition Failed	`If-Match` mismatch	✅ Both	Stale ETag.
422 Unprocessable Content	Semantic validation failure	✅ Both	Violates business rule.
500 Internal Server Error	Unexpected failure	✅ Both	No internal details.

OAS Authoring Requirements

```
paths:
  /v1/customers/{id}:
    patch:
      summary: Partially update customer
      parameters:
        - in: path
          name: id
          required: true
          schema: { type: string }
        - in: header
          name: If-Match
          schema: { type: string }
      requestBody:
        required: true
        content:
          application/merge-patch+json:
            schema: { $ref: '#/components/schemas/CustomerPatch' }
      responses:
        '200':
```

```
      description: Successful partial update
      headers:
        ETag:
          description: Entity tag for versioning
          schema: { type: string }
    '412': { description: Precondition failed (stale ETag) }
    '400': { description: Invalid request }
    '404': { description: Not found }
```

Governance Checklist

Check	Validation
✅ Partial update only	PATCH does not require full resource.
✅ Concurrency control	`ETag` + `If-Match` required.
✅ Media type governed	Default RFC 7396 only.
✅ Status codes approved	200/204/400/401/403/404/405/409/412/422/500.
✅ Error model compliant	RFC 7807 Problem Details.
✅ OAS validated via Spectral	Governance automation ready.

Developer Summary

- Use `PATCH` for **partial updates** only.
- Default to **JSON Merge Patch (RFC 7396)**.
- Use `If-Match` and `ETag` for safe concurrency.
- Return `200 OK` with updated representation or `204 No Content` if unchanged.
- For **singleton resources**, use PATCH for partial changes, PUT for full replacement.
- Follow approved status codes and RFC 7807 error structure.
- Declare supported patch type(s) explicitly in OAS.
- All `PATCH` endpoints must pass Spectral linting for governance compliance.

API Standard: Batch Operations

1. Purpose

Batch operations enable clients to perform **the same action on multiple resources** in a single request (e.g., create, update, or trigger actions across many items).
This guide defines **Lumen's governance standards** for batch APIs—latency thresholds, idempotency, status codes, response shape, and OpenAPI documentation.

Batch APIs must behave **predictably**, support **safe retries**, and meet **bounded synchronous latency**. If those guarantees can't be met, use the **Long-Running Operation (LRO)** pattern.

---

2. Semantics

Property	Requirement
Verb	`POST`
Safe	❌ No — modifies server state
Idempotent	⚠ Recommended for synchronous; **required** for asynchronous
Response Type	`application/json`
Behavior	Processes multiple items of the same resource type atomically **per item** (not per batch)

---

3. When to Use Batch Operations

Use Case	Example	Behavior
Batch Create or Update	`POST /v1/customers/batch`	**201** if all succeed, **207** if mixed results, **202** if deferred.

Use Case	Example	Behavior
Batch Action	`POST /v1/invoices/batch/void`	**200/207/202** depending on outcome.
Large Imports / Heavy Processing	`POST /v1/orders/import`	**202 Accepted** + `Location` to a job; follow the LRO guide.

---

4. Execution Policy

4.1 Synchronous Mode

A batch **may** execute synchronously when:

- **Predictable latency:** ≤ **30 s p95 / 60 s absolute**
- **Deterministic outcome:** same input → same observable result (conflicts included)
- **Retry safety (recommended):** supports `Idempotency-Key` for deduping retries

4.2 Asynchronous Mode

If synchronous boundaries can't be guaranteed:

- Return `202 Accepted`
- Include `Location` to a job resource
- Follow the **LRO Style Guide** for polling, job states, lifecycle

---

5. Status Codes

Code	Meaning	Applies To	Notes
201 Created	All items created	Batch create	May return representation or summary
200 OK	All items succeeded (non-create)	Batch action	Uniform success

Code	Meaning	Applies To	Notes
207 Multi-Status	Mixed outcomes	Create/action	Return per-item results + summary
202 Accepted	Deferred job	Any	`Location` → job URI
4xx/5xx	Uniform failure	Any	Use RFC 7807 Problem Details

Rules: Use **207** only when per-item outcomes differ. Use **201/200** for uniform success. Use **202** when work is deferred.

---

6. Response Structure

6.1 207 Multi-Status (Partial Success)

```
HTTP/1.1 207 Multi-Status
Content-Type: application/json
{
  "results": [
    { "id": "c101", "status": 201, "message": "Created" },
    { "id": "c102", "status": 409, "message": "Duplicate customer" }
  ],
  "summary": { "total": 2, "succeeded": 1, "failed": 1 }
}
```

Rules

- **MUST** include `summary` (total/succeeded/failed)
- **MUST** use numeric HTTP status codes (100–599) for each item's `status` field
- **MAY** include `message` or failure details
- **MAY** offer **compact** (failures-only) vs **full** modes
- **MUST NOT** truncate without pagination metadata
- **MAY** expose a **results** endpoint (optional, use-case dependent)

---

7. 202 Accepted (Asynchronous Jobs)

Submit

```
POST /v1/customers/batch
→ 202 Accepted
Location: /v1/jobs/9827
Retry-After: 10
```

Poll

```
GET /v1/jobs/9827
→ 200 OK
{
  "id": "9827",
  "state": "in_progress",
  "accepted": 1000,
  "processed": 300,
  "succeeded": 295,
  "failed": 5,
  "links": {
    "results": "/v1/jobs/9827/results"
  }
}
```

Rules

- **MUST** include `Location`
- **SHOULD** include `Retry-After`
- **MUST** dedupe via `Idempotency-Key` (same key + payload → same job)
- **MUST NOT** embed resource URIs or per-item data in Job
- See **LRO Style Guide** for states & polling

---

8. Headers

Header	Usage
Idempotency-Key	Required for all unsafe batch POSTs; dedupes retries
Location	Required for 201/202

Header	Usage
Retry-After	Recommended for 202/throttling
Cache-Control	`no-store` for sensitive/transient results
Vary	Include `Authorization` when results depend on user context

---

9. Idempotency

Requirement	Description
MUST	Deduplicate by `(method, canonical path, Idempotency-Key, request hash)`
MUST	Return identical outcome on same key + payload
MUST	Retain key + response metadata ≥ 24h (72h recommended for async)
MUST	`409 Conflict` if key reused with **different** payload
RECOMMENDED	`Retry-After` on transient failures

Async specifics:

- Same key+payload while job queued/in-progress → **return same job** (don't create a new one)
- After completion → **replay** stored outcome / job status

---

10. OpenAPI Authoring (Using Reusable Schemas)

10.1 Batch Endpoint (sync/async)

```
paths:
  /v1/customers/batch:
    post:
      summary: Create multiple customers
```

```
    parameters:
      - in: header
        name: Idempotency-Key
        required: true
        schema: { type: string, maxLength: 255 }
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: array
            items: { $ref: '#/components/schemas/CustomerCreate' }
    responses:
      '201': { description: All created successfully }
      '200': { description: All items succeeded (non-creation action)' }
      '207':
        description: Partial success (mixed outcomes)
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/BatchMultiStatusResponseCompact'
      '202':
        description: Accepted for asynchronous processing
        headers:
          Location:
            schema: { type: string, format: uri }
          Retry-After:
            schema: { type: integer, minimum: 1 }
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/JobStatus'
      '400': { description: Invalid input or schema error }
```

If you also support **full** results inline for small batches, offer an alternate 207 response using `BatchMultiStatusResponseFull`.

10.2 Job Polling

```
paths:
  /v1/jobs/{jobId}:
    get:
      summary: Get job status
      parameters:
        - in: path
```

```
          name: jobId
          required: true
          schema: { type: string }
      responses:
        '200':
          description: Job status
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/JobStatus'
```

10.3 (Optional) Exported Full Results

```
paths:
  /v1/exports/{exportId}:
    get:
      summary: Download exported batch results
      parameters:
        - in: path
          name: exportId
          required: true
          schema: { type: string }
      responses:
        '200':
          description: Export ready
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ExportManifest'
        '202': { description: Still generating; retry later }
        '410': { description: Export expired }
```

---

11. Reusable Schemas (Components)

```
components:
  schemas:

    BatchSummary:
      type: object
      description: Counts for a batch operation
      required: [total, succeeded, failed]
      properties:
```

```
    total: { type: integer, minimum: 0 }
    succeeded: { type: integer, minimum: 0 }
    failed: { type: integer, minimum: 0 }

BatchFailure:
  type: object
  description: Per-item failure record (used in compact and full)
  required: [index, status, code, message]
  properties:
    index:
      type: integer
      minimum: 0
      description: Zero-based position of the item in the submitted arra
    id:
      type: string
      nullable: true
      description: Server-assigned or client key if available
    status:
      type: integer
      description: HTTP status for this item (e.g., 409, 412, 422)
    code:
      type: string
      description: Stable application error code (string, not numeric)
    message:
      type: string
      description: Human-readable summary (safe for logs)
    details:
      $ref: '#/components/schemas/ProblemDetails'

BatchSuccess:
  type: object
  description: Per-item success record (used only in 'full' view)
  required: [index, status]
  properties:
    index:
      type: integer
      minimum: 0
    id:
      type: string
      nullable: true
      description: Created/affected resource identifier if available
    status:
      type: integer
      description: HTTP status for this item (e.g., 200, 201)
    resource:
      type: object
      additionalProperties: true
      description: Optional representation of the created/updated resour
```

```
BatchMultiStatusResponseCompact:
  type: object
  description: Default compact batch response (summary + failures only)
  required: [summary]
  properties:
    summary:
      $ref: '#/components/schemas/BatchSummary'
    failures:
      type: array
      description: Failed items only (paged if large)
      items: { $ref: '#/components/schemas/BatchFailure' }
    page:
      $ref: '#/components/schemas/PageMeta' # refer to pagination style

BatchMultiStatusResponseFull:
  type: object
  description: Full batch response (summary + successes + failures). Use
  required: [summary]
  properties:
    summary:
      $ref: '#/components/schemas/BatchSummary'
    successes:
      type: array
      description: Successful items (optional and typically limited)
      items: { $ref: '#/components/schemas/BatchSuccess' }
    failures:
      type: array
      description: Failed items (may be paged)
      items: { $ref: '#/components/schemas/BatchFailure' }
    page:
      $ref: '#/components/schemas/PageMeta' # refer to pagination style

JobStatus:
  type: object
  description: Operational status for an asynchronous batch job (no per-
  required: [id, state, submitted_at, progress, links]
  properties:
    id: { type: string }
    type: { type: string, enum: [batch] }
    state: { type: string, enum: [queued, in_progress, completed, failed
    submitted_at: { type: string, format: date-time }
    progress:
      type: object
      required: [total, processed, succeeded, failed]
      properties:
        total: { type: integer, minimum: 0 }
        processed: { type: integer, minimum: 0 }
```

```
          succeeded: { type: integer, minimum: 0 }
          failed: { type: integer, minimum: 0 }
      links:
        type: object
        required: [self]
        properties:
          self:
            type: string
            format: uri
            description: Canonical job URI for polling
          results:
            type: string
            format: uri
            nullable: true
            description: Optional link to compact or exported results (no
      idempotency_key:
        type: string
        description: Echo of the submission key to support replay semantic
      request_hash:
        type: string
        description: Hash/fingerprint of original request body for conflic
      error:
        $ref: '#/components/schemas/ProblemDetails' # refer to LPDP

  ExportManifest:
    type: object
    description: Descriptor for large exported batch results
    required: [id, state]
    properties:
      id: { type: string }
      state: { type: string, enum: [pending, generating, completed, failed
      file_type: { type: string, enum: [application/jsonl, application/jso
      size_bytes: { type: integer, minimum: 0 }
      expires_at: { type: string, format: date-time }
      download_url: { type: string, format: uri }
      error:
```

---

12. Governance Validation Checklist

- ✅ Latency boundaries: ≤ 30 s p95 / ≤ 60 s absolute (else LRO)
- ✅ Status codes: 200 / 201 / 207 / 202 only

- ✅ 207 response uses **BatchMultiStatusResponseCompact** (or **... Full** when allowed)
- ✅ Idempotency: `Idempotency-Key` required (async); recommended (sync)
- ✅ OAS coverage: 201 + 207 + 202 documented; job uses **JobStatus** schema
- ✅ Spectral rules: enforce headers, responses, and schemas

API Standard: Standardized Data Types and Formats

Purpose

To ensure consistency, readability, and interoperability across all APIs published by Lumen, this section defines the canonical data types, formats, and reusable schemas to be used in all OpenAPI 3.0.3 specifications.

Principles

- APIs **MUST** use only the data types and formats listed here.
- All fields that accept structured values (e.g., country, currency, language, etc.) **MUST** include:
- a `description` explaining the expected standard,
- a `pattern` (regex) to make the rule self-explanatory to external developers, and
- an `example` value.
- Custom `format` identifiers (e.g., `country-code`) are **advisory only**.
- They are intended for internal governance, linting, and future migration to OAS 3.1.
- External consumers should be able to use the schema **without** referencing any style guide.

Rationale: Custom `format` vs Regex

Aspect	Custom `format`	Regex + Example
Purpose	Governance keyword for Spectral, SDKs, portals	Human & machine validation rule
External Developer Value	Low – tooling hint only	High – immediately clear
Governance Value	High – centralized semantics	Medium – must duplicate everywhere
Recommended Use	Together – `format` + `pattern` + `example`	

Standard Data Types (OAS 3.0.3)

Type	OAS `type`	OAS `format`	Description	Example
Boolean	`boolean`	—	`true` or `false`	`true`
Integer (32 bit)	`integer`	`int32`	−2,147,483,648 … 2,147,483,647	`42`
Integer (64 bit)	`integer`	`int64`	−9,223,372,… 7, use ≤ `2^53 - 1` for I-JSON	`9007199254740991`
Number (float)	`number`	`float`	IEEE-754 single precision	`3.14`
Number (double)	`number`	`double`	IEEE-754 double precision	`3.1415926535`
String	`string`	—	UTF-8 text	`"Lumen rocks!"`
Date	`string`	`date`	RFC 3339 date	`"2025-11-11"`
Date-Time	`string`	`date-time`	RFC 3339 timestamp (UTC `Z`)	`"2025-11-11T14:05:00Z"`
Byte (Base64)	`string`	`byte`	Base64 RFC 4648 § 4	`"VGVzdA=="`
Binary	`string`	`binary`	Raw octet stream (upload/download)	*(file body)*

Custom (Governed) Formats

> ✅ These are optional keywords for internal tooling and do **not** alter wire format.

Logical Type	OAS type	OAS format	Description / Rule	Example
URI	`string`	`uri`	URI per RFC 3986	`"https://example.com/x"`
Email	`string`	`email`	RFC 5322 address	`"user@example.com"`
UUID	`string`	`uuid`	RFC 4122 UUID	`"279fc665-d04d-4dba-bcad-17c865489dfa"`
Time	`string`	`time`	RFC 3339 partial time	`"08:26:40Z"`
Decimal	`string`	`decimal-string`	Arbitrary precision string	`"1234.5678"`
Language	`string`	`language-tag`	ISO 639-1 / BCP 47 code	`"en-US"`
Country	`string`	`country-code`	ISO 3166-1 alpha-2 uppercase	`"US"`
Currency	`string`	`currency-code`	ISO 4217 alpha-3 uppercase	`"USD"`

Each field **must also define a pattern and example** so the meaning is clear even if clients ignore `format`.

Reusable Schema Definitions

CountryCode

```
CountryCode:
  type: string
  description: ISO 3166-1 alpha-2 country code (uppercase).
  format: country-code
```

```
  pattern: '^[A-Z]{2}$'
  example: "US"
```

CurrencyCode

```
CurrencyCode:
  type: string
  description: ISO 4217 currency code (uppercase).
  format: currency-code
  pattern: '^[A-Z]{3}$'
  example: "USD"
```

LanguageTag

```
LanguageTag:
  type: string
  description: Language tag per ISO 639-1 / BCP 47.
  format: language-tag
  pattern: '^[A-Za-z]{2,3}(-[A-Za-z0-9]{2,8})*$'
  example: "en-US"
```

Email

```
Email:
  type: string
  description: Email address per RFC 5322.
  format: email
  pattern: '^[^@\s]+@[^@\s]+\.[^@\s]+$'
  example: "user@example.com"
```

UUID

```
UUID:
  type: string
  description: Universally Unique Identifier per RFC 4122.
  format: uuid
  pattern: '^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-
  example: "279fc665-d04d-4dba-bcad-17c865489dfa"
```

URI

```
URI:
  type: string
  description: URI string per RFC 3986.
  format: uri
  pattern: '^(https?|ftp)://[^\s/$.?#].[^\s]*$'
  example: "https://api.lumen.com/v1/services"
```

Money Object (Integer Minor Units)

> Represents a monetary amount using ISO 4217 minor units (integer-based).

```
Money:
  type: object
  required: [amount, currency]
  properties:
    amount:
      type: integer
      minimum: 0
      maximum: 9007199254740991   # I-JSON safe (2^53 – 1)
      description: >
        Monetary amount expressed in ISO 4217 minor units.
        Example: USD 12.34 → amount=1234, currency="USD"
    currency:
      $ref: '#/components/schemas/CurrencyCode'
  example:
    amount: 1234
    currency: "USD"
```

Address Object

A common postal address representation used consistently across all domains.

```
Address:
  type: object
  description: Postal address (country-specific formatting may apply).
  additionalProperties: false
  properties:
    line1:
      type: string
```

```
      description: Primary street line (street, number, unit).
      maxLength: 256
      example: "100 Main St Apt 5B"
    line2:
      type: string
      description: Secondary line (building, suite, floor) if needed.
      maxLength: 256
      example: "Building A, Floor 3"
    city:
      type: string
      description: City, town, or locality.
      maxLength: 128
      example: "Denver"
    state:
      type: string
      description: State, region, or province (country-specific).
      maxLength: 128
      example: "CO"
    postal_code:
      type: string
      description: Postal/ZIP code (country-specific format).
      maxLength: 32
      example: "80202"
    country:
      $ref: '#/components/schemas/CountryCode'
  example:
    line1: "100 Main St Apt 5B"
    city: "Denver"
    state: "CO"
    postal_code: "80202"
    country: "US"
```

Context-specific variants:

```
UsShippingAddress:
  allOf:
    - $ref: '#/components/schemas/Address'
    - type: object
      required: [line1, city, state, postal_code, country]
```

```
InternationalBillingAddress:
  allOf:
    - $ref: '#/components/schemas/Address'
    - type: object
      required: [line1, city, postal_code, country]
```

Governance Validation Rules

Rule	Enforcement
Disallow unrecognized `format` values	Spectral lint
Require `pattern`, `example`, `description` when custom `format` used	Spectral lint
Validate Money.amount ≤ $2^{53} - 1$	Spectral + CI schema check
Enforce ISO regex patterns for country/currency/language	Spectral rule
Warn if `float`/`double` used for monetary values	Spectral rule

Summary

- **External developers** should understand schemas immediately through **description + regex + example**, without needing to read the style guide.
- **Governance tooling** (Spectral, SDK generators, portals) can rely on **custom formats** for consistency and automation.
- **Money, Address, Country, Currency, and Language** fields are standardized across all domains.

API Standard: Rest & Resource Design

Purpose

This section defines how APIs at **Lumen** are structured around resources — the fundamental building blocks of RESTful architecture — ensuring consistency, discoverability, and governance automation across all API domains.

REST Principle — Resource-Centric Design

Governance Objective

APIs must be modeled around *resources*, not actions.
A *resource* represents a logical entity (e.g., `customer`, `order`, `connection`) that can be acted upon via standard HTTP methods.

> "The key abstraction of information in REST is a resource."
> — *Roy Fielding, REST Dissertation §5.2*

Governance Rules

- Every REST API must identify its **primary resource(s)** before design begins.
- Resources are **nouns**, not verbs.
- CRUD behavior must be represented through **HTTP methods**, not in the URI.

✅ Example

Not Recommended	Recommended
`POST /createUser`	`POST /users`
`POST /updateUser`	`PATCH /users/{id}`
`POST /deleteUser`	`DELETE /users/{id}`

Resource Hierarchies and URI Scope

Governance Objective

Maintain intuitive, predictable, and domain-contained URI structures.

Governance Rules

- URIs must follow:

`/{domain}/v{n}/{resource}/{resource_id}/[subresource]` * Each resource belongs to **one business domain**. * **No cross-domain fan-out**:
A resource in one domain cannot directly reference or nest resources from another. * URI depth must not exceed **two resource levels** (max 4 path segments after version).

✅ Example

Allowed	Disallowed
`/customers/v1/customers/{id}/accounts`	`/projects/{id}/tasks/{taskId}/comments/{commentId}`
`/network/v1/connections/{id}`	`/billing/v1/customers/{id}/network/connections`

Collections vs Singleton Resources

Governance Objective

Differentiate between **collections** (plural) and **singletons** (singular) to ensure consistent structure and expectations.

Governance Rules

- **Collections:** Plural noun, contain multiple resources (e.g., `/customers`).
- **Singletons:** Singular noun, one-per-parent (e.g., `/projects/{id}/config`).
- **Singleton resources must not define:**
- `id` property
- `POST` or `DELETE` operations
- **Singletons must define:** `GET` and `PATCH`.

Implementation Hint

Mark singletons explicitly in OAS:

```
x-singleton: true
```

✅ Example

Type	Path	Allowed Methods
Collection	`/customers`	`GET`, `POST`
Singleton	`/projects/{id}/config`	`GET`, `PATCH`

Resource Naming

Governance Objective

Enable predictable API navigation and uniform discoverability.

Governance Rules

- Use **nouns** for resources; avoid verbs.
- Pluralize collection names (e.g., `/users`, `/invoices`).
- Singular for singletons (`/me`, `/company`).
- **MUST** use `lowercase kebab-case` (`/service-offers`).
- Resource URIs must always be **URL-safe** and **lowercase**.

Actions on Resources (Non-CRUD)

Governance Objective

Support non-CRUD operations that represent *state transitions* or *processes* without breaking REST uniformity.

Governance Rules

- Use verbs **only** when:

- The action cannot be represented by standard HTTP methods.

- It represents a *state transition* or *long-running operation (LRO)*.
- Must be scoped to a resource (not root-level).
- Use **process/job** pattern for asynchronous operations.
- Allowed form:

```
POST /{resource}/{id}/action
```

✅ Example

Not Recommended	Recommended
`POST /archiveUser`	`POST /users/{id}/archive`
`POST /reprovision`	`POST /connections/{id}/reprovision`

If the operation is long-running, annotate:

```
x-lro: true
```

Resource Identifiers (IDs)

Governance Objective

Ensure resource identity is secure, unique, stable, and automation-friendly.

Governance Rules

- **MUST be opaque** — consumers should not infer meaning from IDs.
- **MUST be generated by provider**, never client-supplied.
- **MUST be globally unique** within API space.
- **MUST be URL-safe**.
- **MUST use consistent field naming convention**: `<resource>_id`.
- **MUST have fixed length** — changing length constitutes a breaking change.
- **MUST NOT** use sequential IDs.
- **MAY** use UUID v4 or Snowflake-style identifiers.
- **MAY** use short type prefixes (e.g., `PAY_`, `ACC_`) for troubleshooting.

Reusable Schema

```
ResourceId:
  type: string
  description: >
    Opaque, globally unique identifier for a resource.
    Generated by the API provider. Never sequential or client-supplied.
  pattern: '^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-
  example: "279fc665-d04d-4dba-bcad-17c865489dfa"
```

Resource Metadata

Governance Objective

Provide uniform lifecycle visibility (creation, updates) across all resources.

Governance Rules

Every **collection resource** MUST include:

- `id`
- `created_at`
- `last_updated_at`

Singletons exclude these (redundant with parent).

Reusable Schema

```
ResourceMetadata:
  type: object
  required: [id, created_at, last_updated_at]
  properties:
    id:
      $ref: '#/components/schemas/ResourceId'
    created_at:
      type: string
      format: date-time
      description: Creation timestamp (UTC)
```

```
    last_updated_at:
      type: string
      format: date-time
      description: Last update timestamp (UTC)
```

Governance Enforcement Rules

Rule ID	Enforcement Type	Description
`path-domain-scope`	Error	Paths must begin with a domain (e.g., `/billing/`, `/network/`).
`path-versioned`	Error	Paths must include major version segment (e.g., `/v1/`).
`path-segment-depth`	Error	Maximum path depth = 4 segments after version.
`singleton-no-post-delete`	Error	Singleton resources cannot define POST/DELETE.
`id-schema-ref`	Error	All `*_id` fields must use `ResourceId` schema.
`resource-metadata-required`	Warn	Collection schemas must include `ResourceMetadata`.
`verb-endpoint-lro`	Error	Verbs in path allowed only if x-action: true or x-lro: true. This explicitly permits action endpoints for complex state transitions.

Example Resource Model

```
paths:
  /customers:
    get:
      summary: List all customers
      responses:
        '200':
          content:
            application/json:
```

```
            schema:
              type: object
              properties:
                data:
                  type: array
                  items:
                    allOf:
                      - $ref: '#/components/schemas/ResourceMetadata'
                      - type: object
                        properties:
                          name:
                            type: string
                            example: "Acme Corp"
  post:
    summary: Create a new customer
    responses:
      '201':
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Customer'

/customers/{customer_id}/accounts:
  get:
    summary: List accounts for a customer
    parameters:
      - name: customer_id
        in: path
        required: true
        schema:
          $ref: '#/components/schemas/ResourceId'
```

Governance Rationale

Principle	Outcome
Resource-centric modeling	Aligns APIs to REST uniform interface, improving predictability.
Strict URI domain boundaries	Enables per-domain governance, versioning, and automation.
Opaque, stable identifiers	Ensures consistency across distributed systems and prevents information leakage.

Principle	Outcome
Singleton control	Simplifies lifecycle management for dependent configurations.
Action resource governance	Preserves REST semantics while supporting modern async/LRO workflows.

Summary

This REST & Resource section moves Lumen APIs from loosely interpreted REST conventions to a **governable, automatable, and externally predictable API architecture**.

- Consistent structure = easier discoverability and SDK generation.
- Enforced ID and resource schemas = audit-ready data lineage.
- Governed URI scope = scalable domain modeling without overlap.

API Standard: JSON Payload

Overview

All Lumen APIs **must use JSON** as the canonical message format for request and response bodies.
From a governance standpoint, this ensures predictable interoperability across platforms, easier validation using OpenAPI 3.0.3, and forward-compatible payload design.

To achieve this, all JSON payloads must follow the structural, naming, and encoding rules defined in RFC 7159 (JSON) and RFC 7493 (I-JSON).
The goal is to enforce **clarity, safety, and uniformity** across all APIs regardless of team or domain.

Governance Principles

Governance Objective	Policy Requirement	Rationale
Standardized Format	All request and response bodies **MUST** be valid JSON	Guarantees structured, parsable data with clear

Governance Objective	Policy Requirement	Rationale
	objects.	schema evolution paths.
I-JSON Compliance	JSON payloads **MUST** conform to the I-JSON profile (RFC 7493).	Ensures consistent behavior across clients, SDKs, and backend systems.
Encoding	All JSON content **MUST** use UTF-8 encoding.	Avoids encoding ambiguity and corruption across gateways.
Uniqueness of Keys	Each JSON object **MUST** contain **unique property names**.	Prevents parser ambiguity and security issues (duplicate key override).
Content Type	Media type **MUST** be declared as `application/json`.	Enables correct content negotiation and API discovery.
Extensibility	Payloads **MUST** support additive evolution (new fields tolerated).	Allows forward-compatible releases without breaking clients.

Structural Rules

Nested Structures

SHOULD prefer nested, well-scoped objects over flattened key sets.
This improves logical grouping, maintainability, and future extensibility.

✅ Recommended

```
{
  "customer_id": "C123",
  "billing_address": {
    "line1": "100 Main St",
    "city": "Denver",
    "postal_code": "80202"
  }
}
```

❌ Not Recommended

```
{
  "customer_id": "C123",
```

```
  "billing_address_line1": "100 Main St",
  "billing_address_city": "Denver",
  "billing_address_postal_code": "80202"
}
```

Field Naming Convention

Field names **MUST** follow **lower snake_case**, matching regex:

```
^[a-z][a-z0-9_]*$
```

Rules:

- First character must be lowercase.
- Use underscores to separate words.
- No hyphens, camelCase, or PascalCase.
- ASCII-only.

✅ **Recommended** → `customer_name`, `invoice_number`, `created_at`
❌ **Not Recommended** → `salesOrderId`, `CustomerName`, `sales-order-id`

Array Naming Convention

Array properties **MUST** use plural names to signal multiplicity.
Empty arrays must be represented as `[]`, *not* `null`.

✅ **Recommended**

```
{
  "line_items": []
}
```

❌ **Not Recommended**

```
{
  "line_items": null
}
```

I-JSON Compliance Checklist

Category	Requirement	Description
Encoding	UTF-8	Required for all JSON payloads.
Duplicate Keys	Forbidden	Parsers must reject payloads where a key appears more than once in the same object.
Numbers	Finite, safe range only	No `NaN`, `Infinity`, or numbers > $2^{53} - 1$.
Strings	Valid Unicode scalars only	No control or invalid Unicode characters.
Top-Level Type	Object only	Arrays, primitives, or raw strings as top-level payloads are not permitted.
Date/Time	RFC 3339 UTC format	Example: `"2025-11-11T18:30:00Z"`

Example — Valid Payload

```
{
  "order_id": "ORD_9876",
  "customer": {
    "customer_id": "C12345",
    "name": "Acme Corp"
  },
  "line_items": [
    {
      "item_id": "I567",
      "quantity": 3
    }
  ],
  "total_amount": 1500,
  "currency": "USD",
  "created_at": "2025-11-11T18:30:00Z"
}
```

Example — Invalid Payloads

Duplicate Keys

```
{
  "name": "Acme",
  "name": "Acme Inc"
}
```

Null Array

```
{
  "line_items": null
}
```

Unsafe Number

```
{
  "amount": 99999999999999999
}
```

Governance Enforcement

- **Spectral Linting Rules:**
  Validate UTF-8 compliance, key uniqueness, and snake_case fields. Disallow null arrays and duplicate keys.
- **Gateway Validation:**
  Reject invalid or non-I-JSON payloads at request-time.
- **Schema Linting in CI:**
  Automated pipelines must validate sample payloads and OpenAPI schemas for I-JSON compliance.
- **Documentation Consistency:**
  All examples in the Developer Portal must use valid, lint-clean JSON payloads.

Summary

JSON payload hygiene directly determines API quality and interoperability. Lumen APIs **must** adhere to I-JSON constraints, snake_case naming, predictable array behavior, and consistent UTF-8 encoding.
This ensures that payloads are **machine-safe, forward-compatible, and consistent** across every domain and consumer.

API Versioning Strategy

This document outlines the proposed official strategy for versioning all public-facing REST APIs. A stable and predictable versioning strategy is critical for building trust with API consumers and ensuring a reliable developer experience.

Versioning Principles

Our strategy uses both **Major** and **Minor** versions, but they are applied in different places to serve distinct purposes.

Major Versions: The Public Contract 📜

The major version represents the public contract with the client and **MUST** only change when a **breaking change** is introduced.

- **Implementation**: The major version **MUST** be included in the URI path and prefixed with a "v" (e.g., `v1`, `v2`).

`https://api.lumen.com/inventory/v1/connections` * **Client Impact**: A client's integration is tied to this major version. We guarantee that no breaking changes will be made within a given major version.

Minor Versions: The Diagnostic Tool 🩺

The minor version is an internal reference used to track non-breaking, additive changes. It serves as a crucial tool for communication and diagnostics.

- **Implementation**: The server **SHOULD** include the semantic version (major.minor) in an HTTP response header.

`API-Version: 1.2` * **Client Impact**: This has **zero impact** on the client's code. They are not required to send or parse this header. Its purpose is to provide a precise reference point for debugging, especially in distributed systems where deployment lag could lead to different nodes running different code versions.

Defining Breaking vs. Non-Breaking Changes

To ensure clarity, the following examples define what constitutes a breaking change versus a non-breaking change. When in doubt, a change **SHOULD** be treated as breaking.

Breaking Changes (Requiring a New Major Version)

Breaking changes are modifications that can potentially break an existing client integration.

- **Request Changes:**
- Removing or renaming a parameter.
- Adding a new *required* parameter.
- Making a previously optional parameter *required*.
- Changing the data type of a parameter.
- Adding a new validation rule to an existing parameter (e.g., restricting the length of a string).
- Changing authentication or authorization requirements.
- **Response Changes:**
- Removing or renaming a field in the response body.
- Making a previously *required* response field optional.
- Changing the data type of a response field (e.g., from an integer to a money object).
- Changing the structure of the JSON payload (e.g., nesting an existing field).
- Removing a value from an `enum`.
- Any change to the primary HTTP Status Code returned by an existing API endpoint.
- Adding a new value to enum

Non-Breaking Changes (Minor Version Increments)

Non-breaking (additive) changes are modifications that should not break an existing integration. These changes will be available in all supported API versions.

- **Request Changes:**
- Adding a new *optional* parameter.
- Adding a new *optional* request header.
- Adding an optional field in the request body
- **Response Changes:**
- Adding a new field to the response body.
- Adding a new endpoint or operation.
- Adding a new response header.

API Lifecycle Management

Announcing Major Changes (Pre-Release Communication)

To provide our consumers with adequate time to plan and budget for upcoming migrations, breaking changes **SHOULD** be announced at least **6 months** *before* a new major version is released.

- This announcement **MUST** be made through the official API changelog, documentation, and, where possible, direct email communication to affected consumers.
- The announcement **MUST** detail the upcoming breaking changes, the reasons for them, and provide a preliminary migration guide.

Active Version Support

All API products **MUST** support at least two active major versions during a migration period.

- When a new REST API version with breaking changes is released, the previous API version **MUST** be supported for at least **24 more months**.
- This overlap period allows clients to migrate at their own pace without service disruption.

Deprecating Endpoints and Features

Deprecating an endpoint or feature requires a phased approach. The removal of any API element is a **breaking change** and can only happen in a new major version.

1. **Announce (Phase 1) 📣: The endpoint remains fully functional.**
2. In the OpenAPI specification, the operation **MUST** be marked with `deprecated: true`.
3. The server **SHOULD** return a `Sunset` header (RFC 8594) containing the exact date and time when the endpoint will be fully removed.
4. The server **SHOULD** also return a `Deprecation` header containing the date when the deprecation was announced. The format required by the RFCs is a standard **HTTP-date**, as defined in RFC 7231
5. The documentation **MUST** be updated to reflect the deprecation, the removal timeline, and any new preferred endpoints.

Example HTTP Headers

`HTTP/1.1 200 OK Deprecation: Wed, 22 Oct 2025 13:30:00 GMT Sunset: Fri, 22 Oct 2027 13:30:00 GMT` 2. **Guide (Phase 2) 🚀**: Provide a clear path forward for consumers.

- If a replacement exists, the server **SHOULD** return a `Link` header pointing to the new endpoint (e.g., `Link: <.../v1/new-endpoint>; rel="alternate"`).
- If the feature is being removed entirely, the documentation **MUST** state this clearly.

- **Remove (Phase 3) ❌**: After the support window, the endpoint is removed.

- The deprecated endpoint **MUST** be completely removed in the next major version release (e.g., `/v2`).

Sunsetting and Decommissioning a Major Version

After the mandated 24-month support window for a previous version ends, that version will be decommissioned.

- **Communication**: Consumers still using the old version will be notified via multiple channels about the upcoming shutdown.
- **Brownouts**: A "brownout" period may be implemented, where the old version is temporarily disabled for short periods to alert remaining users.
- **Final Response**: Once decommissioned, the old version's endpoints **SHOULD** return an HTTP `410 Gone` status code to indicate that the resource is intentionally and permanently unavailable.

Handling Experimental (Beta) Features

To gather feedback on new features, a beta version may be released. Beta features are not bound by the standard 24-month support policy and can be changed or removed without the standard deprecation process.

- **Implementation**: Beta features **SHOULD** be identified clearly, either through a beta-specific URI (e.g., `/v1-beta/new-feature`) or a version header (`API-Version: 2.1.0-beta`).

Updating Shared Data Models

Making a breaking change to a common data model (e.g., the `Address` object) across multiple APIs **MUST** be handled with the **"Expand and Contract"** pattern to avoid a disruptive, simultaneous release of new major versions for all dependent APIs.

1. **Expand (Non-Breaking)**: In the current major version, add new fields as *optional* and mark the old fields as *deprecated*. The server logic is updated to handle both old and new fields.

2. **Transition (Monitor)**: Allow consumers to migrate to the new fields over the 24-month support window. Monitor the usage of the deprecated fields.
3. **Contract (Breaking)**: In the next major version, remove the deprecated fields entirely. The new fields can now be made *required*

Tactical Engagements & Analysis

IoD API Analysis

IOD APIs and Customer Interactions

Information contained in these pages is accurate to the best of our ability. If something needs to be clarified or corrected, please reach out.

Table of Contents

Overview

This document captures current IOD APIs and how they are used to order services and gaps. It also talks short term fix for pressing customer issues with API like customers are having

- Getting status of order via webhook
- Issue setting up sandbox without functional webhook for sandbox environment.

Details

Current API and Purpose

- Provide information what the current APIs are available and their purpose and gaps.

Option# 1 Current APIs with API to get order request status reporting orderId and serviceId

- use id (b2b) which returned today in orderRequest POST as reference to collect status/state and orderId and serviceId.
  {
  "id": "57ab4320-9aaf-11ee-8cbc-bf8b3daa320c",
  "externalId": 1702578835,
  ....
  }

- Api will return following information to client. it can follow current data format to be compatible with POST payload which is extension TMF payload design.

- **state/status** RUNNING, SUCCEEDED , FAILED , for status follow API Standard: Long-Runing Operations (LRO)

    - Status of the order Request

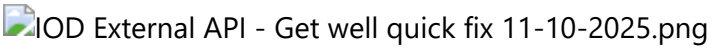
    - **orderId** (88 number, once we have it available in backend or when orderRequest goes to SUCCEEDED)

    - The order id which is need to submit service-configuration request using serviceProvisioning API

    - **serviceId** (77 number, once we have it available in backend or when orderRequest goes to SUCCEEDED)

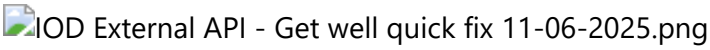
    - The service id which is needed to get the information about service using serviceProvisioning API

    - Use Retry After header attributes to guide client on configurable delay, follow API Standard: Long-Runing Operations (LRO)



Option#2 Current APIs with enhancement to how customer submit order and API they use

- Provide information what can be done in short term/ minimal effort to improve Aysnc experience of order submission API
- Remove dependency on webhook to get status.
- New URI for order submission (compatible with current orderRequest URI for input and response) and get order status , collect serviceId etc and start moving in direction suggested by API style guide for future enhancement.



Future Sate of IoD APIs

- It is expected that IoD will be ordered as Fabric Connection using North Star Design
- API URIs would look different and payload will be different as data dependency will be changed.
- Future IOD API state is not expected to be define here. TODO: add reference to documentation when ready.

Decision

Option#1 is selected as proposal with understanding that it will require minimal development effort without compromising much on the current API experience. UI and naasOpsDashboard already provide this kinda visibility so it is expected that it won't be difficult to create the relationship as API data will be there or collected from backend systems. This will solve the immediate issues customer are facing. With understanding that IoD APIs needs overhaul to match target resource model inline with API Strategy initiative.

Appendix

https://developer.lumen.com/apis/internet-on-demand

MCGW / LMCC Gap Analysis

White Glove API Review Template

The White Glove Approach: Definition and Expectations

The **White Glove Approach** signifies the **high-touch, proactive support and partnership** provided by the API Governance team to guide product teams through the adoption of foundational API standards, principles, and canonical models.

Definition

The White Glove API Review is a **mandatory, deep-dive architectural evaluation** that ensures alignment with the platform's core standards. It is understood that due to real-world constraints, **technical debt** may exist. The primary objective is to make this debt **transparent, quantifiable, and formally managed** rather than blocking the API's release.

Expectations

The expectation for any API undergoing a White Glove review is to achieve the **highest possible degree of conformance** while ensuring any non-conforming elements are explicitly accepted, prioritized, and scheduled for resolution.

Expectation Area	**White Glove Standard (Focus)**
Model Alignment	Full usage of canonical resources and fragments; minimal exposure of non-canonical IDs.
Style Conformance	Adherence to all **mandatory** style guide rules; **Optional** deviations are documented and approved.

Architectural Debt	All identified technical debt **must be clearly outlined** in the **Summary of Gaps** .
Remediation Plan	A clear, committed **Remediation Roadmap** must be established and owned by the Product Team, detailing the plan, timeline, and resources to address all debt.

> **Key Principle:** The White Glove Review serves as the **formal contract** for managing technical debt. An API may proceed with identified gaps, but only if the **Remediation Roadmap is agreed upon** and the **Final Architecture Rating** is acceptable to the architectural stakeholders.

Project Overview

Field	Description
Project / Product Name	
Domain / BU	
Architect Reviewer(s)	
Date of Review	
Primary Stakeholders	
API(s) Under Review	
High-Level Business Capability	
Deployment Model	(*internal / external / partner / wholesale*)

Canonical Model Alignment

This section validates whether the API's data model aligns with Lumen's **canonical domain model**, including **resources, subresources, and reusable schema fragments**.

Resource Inventory

Resource	Type	Description	Canonical?	Notes

Resource Type Options:

- CRUD Resource
- Workflow Resource
- Configuration Resource
- Operational Resource
- Composite Resource (aggregates other resources)

Reusable Schema Fragments Inventory

List all reusable fragments defined or referenced by the API.

Examples of fragments:

- address
- money
- port-reference
- customer-reference
- name
- error-detail
- version-info
- metadata fragments
- pagination fragments
- resource_links

Fragment Name	Description	Canonical Source?	Reused Across APIs?	Notes

This ensures consistency AND reveals fragmentation (“address1/address2 vs line1/line2”).

Canonical Alignment Evaluation

Evaluate the model holistically, not just the top-level resources.

Resource-Level Checks

- Does each resource map to a canonical domain concept?
- Do resource names follow the uniform taxonomy?
- Are relationships represented as resources (nouns) vs verbs (RPC)?
- Are resource boundaries clear (no overlapping responsibilities)?

Reusable Fragment Checks (NEW)

- Are reusable canonical fragments used instead of redefining schemas?
- Does the API introduce duplicate fragments that already exist in the platform?
- Are fragment names consistent with canonical naming conventions?
- Are fragments correctly versioned or tied to product namespace?

Identifier Checks

- Are IDs opaque, stable, non-sequential, URL-safe?
- Does the resource use standardized `<resource>_id` naming?
- Are internal system IDs leaked to clients?

Composition & Structure Checks

- Does the API follow canonical composition rules (nested vs reference)?
- Are subresources modeled consistently (/{resource}/{id}/{subresource})?
- Are reusable components appropriately referenced via `$ref`?
- Are optional and required fields consistent across services?

Canonical Model Gaps & Recommendations

Document discovered issues:

Gap	Impact	Recommendation	Priority

Examples:

- Duplicate address schema detected
- Non-canonical ID format used
- Workflow resource incorrectly modeled as CRUD
- Missing canonical fragments for links, pagination, metadata
- Internal BSS/OSS IDs exposed

API Style Guide Conformance

Evaluate the API against the official Lumen API Style Guide across the following areas:

Style Guide Areas

- URI Design & Structure
- Resource Modeling & Taxonomy
- Naming Conventions
- Versioning Strategy
- Request & Response Schemas
- Identifiers & Resource IDs
- Error Model & Problem Details
- Pagination & Filtering
- Asynchronous Operations Pattern
- Header Standards (Required & Optional)
- Authentication & Authorization Model
- Idempotency
- HTTP Methods & Status Codes
- Content Type & Encoding Standards

- Security, PII, and Data Handling Requirements
- Rate Limiting & Throttling Metadata
- Observability (Logging, Tracing, Correlation IDs)
- Deprecation & Backwards Compatibility Rules
- Governance & Automation Integration
- API Version Lifecycle Management
- Bulk Operations & File Transfer Patterns (if applicable)
- Partner, Wholesale, and Federation-Specific Guidelines

Alignment Summary

- Fully Aligned:
- Partially Aligned:
- Not Aligned:
- Notes & Gaps:

Alignment with Cloud-First API Principles

For each principle, mark:
✓ Fully Aligned ~ Partially ✗ Not Aligned

Resource-Oriented Interface

Desired-State & Outcome-Based Design

Asynchronous & Operation-Tracked

Product-Scoped Versioning

Entitlement-Aware & Monetizable

Self-Service & Declarative Lifecycle

Uniform Taxonomy & Namespacing

Strong Identity & Policy Integration

Observable & Telemetry-Driven

Extensible via Federation & Partner APIs

Automation & Governance-Ready

Developer Experience Consistency

Outcome Metrics & Continuous Improvement

Backwards Compatibility & Graceful Evolution

API Architecture Pattern Evaluation

On-Demand Architecture

(desired-state, async workflow resources)

Traditional Quote → Order Pattern

(check for RPC leakage)

Resource-Driven CRUD

Experience-Driven Façade APIs

(check for ID mapping, complexity masking)

Workflow & Async Operations Review

- 202 Accepted usage
- Operation resource presence
- Location header correctness
- Status transitions
- Error handling
- Timeouts + retry model

Developer Experience (DX) Assessment

- Schema clarity
- Example payloads
- Error examples
- Documentation quality
- OAS completeness
- Try-it-out readiness
- Consistency across APIs

Summary of Gaps

Category	Gap	Severity	Recommendation	Owner

Recommended Remediation Roadmap

- Taxonomy updates
- Converting RPC to workflow resources
- Async conversion
- Canonical modeling changes
- Identity/claims alignment
- Observability improvements
- Security fixes
- Governance automation in CI

Final Architecture Rating

Category	Score (1–5)	Notes
Style Guide Conformance		
Cloud-First Principle Alignment		
API Consistency		
Identity & Entitlement Model		
Security Posture		
Developer Experience		
Operational Readiness		

White-Glove API Security Assessment Template

White-Glove API Security Assessment for

As an: API Security Architect
I want to: Perform a comprehensive API security assessment of the platform and its customer-facing APIs
So that: We can identify gaps across authentication, authorization, token handling, runtime protections, entitlement enforcement, and auditability to align with Lumen's API Security Standards and prepare for future enhancements.

---

1. Platform Security (Internal Posture)

Security of the platform hosting and exposing the API.

Acceptance Criteria

- **[Confirm]** Sensitive platform secrets (API keys, certificates, DSNs) are stored in enterprise-approved vaults and rotated per policy.
- **[Confirm]** All service-to-service communication uses TLS 1.2+ with certificate validation.
- **[Verify]** Administrative and operational actions are audit-logged and logs are tamper-evident.

---

2. Customer API Authentication (AuthN)

Assessment of OAuth 2.0 implementation, token lifecycle, onboarding, and identity flows.

Acceptance Criteria

- **[Review]** End-to-end customer onboarding:
- Application registration
- Customer/account association
- Client ID/Secret issuance
- Partner behavior when acting on behalf of customers
- **[Verify]** OAuth 2.0 Client Credentials is implemented correctly (or document if another flow is used).
- **[Confirm]** No refresh tokens are issued for client_credentials flows (required by OAuth spec).
- **[Review]** Token lifecycle:

- Access token TTL (short-lived)
- Client secret rotation
- JWKS rotation
- **[Verify]** Token validation behavior:
- issuer (iss)
- audience (aud)
- algorithm (RS256/ES256)
- exp, nbf, iat handling
- replay protection (e.g., JTI, nonce, or opaque tokens)
- **[Identify]** Gaps caused by **not using OAuth scopes today**, including:
- lack of standardized permission representation
- entitlements not represented in token
- authorization logic scattered across backend services
- reduced auditability of enforcement decisions

---

3. Customer API Authorization (AuthZ)

Understanding how access decisions are made and enforced today.

Acceptance Criteria

- **[Identify]** The current entitlement model used by :
- read vs write vs config permissions
- customer/account-level authorization
- per-resource vs per-subresource permissions
- partner delegation rules
- implicit entitlements implemented in service code

- **[Document]** Where each entitlement is enforced today:
- API Gateway (coarse authorization)
- Resource Server (business authorization)
- Hybrid or duplicated enforcement
- Places with **no enforcement**
- **[Identify Gaps]** in the existing authorization model:
- no OAuth scopes
- inconsistent entitlement enforcement
- token does not carry required authorization context
- duplicated or siloed authorization logic
- missing data-boundary protections
- **[Review]** Data boundary enforcement:
- Does gateway ensure requested ID/account == token claims?
- Do backend services re-validate?
- Are bypass paths possible?
- **[Identify]** Missing claims that would be required for secure future authorization
  (capture only—do NOT propose new claims).

---

4. Gateway vs Resource Server Enforcement Responsibilities

Clear mapping of runtime security responsibilities.

Acceptance Criteria

API Gateway (Current-State Assessment)

Catalog what the gateway enforces:

- OAuth 2.0 token validation
- Signature, issuer, audience checks
- Basic boundary checks (IDs, account numbers, customer identifiers)
- Request size limits
- Content-type validation
- Schema validation (if enabled)
- Rejection of malformed JSON/XML

Identify missing responsibilities, such as:

- lack of entitlement checks
- no operation-level authorization
- no scope-driven authorization
- absence of a centralized entitlement model

Resource Server (Current-State Assessment)

Catalog business-level authorization enforced in services:

- Account/ownership validation
- Write/configuration permissions
- Delegation logic for partners
- Operation-specific entitlements
- Privileged action audit trails

Identify risks:

- inconsistent implementation across services
- missing secondary validation checks
- unclear trust boundary between gateway and services
- authorization bypass opportunities

---

5. API Gateway Runtime Security Controls

Threat detection, prevention, and real-time hardening.

Acceptance Criteria

- **[Verify]** Per-application and per-customer rate limiting & quota enforcement.
- **[Confirm]** WAF protections (SQLi, SSRF, XSS, injection attacks).
- **[Verify]** Enforcement of:
- request/response size limits
- allowed methods
- schema validation
- content-type restrictions
- rejection of malformed payloads
- **[Review]** mTLS posture (internal or external).
- **[Confirm]** Sensitive data handling:
- PII redaction
- masking rules
- no secrets in logs
- **[Verify]** Protections against:
- ID/account/customer enumeration
- brute-force attempts
- credential stuffing
- replay attacks

---

6. Observability, Logging & Security Monitoring

Acceptance Criteria

- **[Verify]** API logs include:
- trace-id
- client-id
- account/customer ID
- principal-id
- request path
- auth outcome
- **[Confirm]** Logs flow into SIEM (Splunk/Sentinel).
- **[Verify]** Alerts exist for anomalies:
- auth failure spikes
- elevated 403/429
- abnormal traffic per customer
- **[Review]** Log retention policies for compliance.
- **[Confirm]** PII redaction and masking rules are applied uniformly.

---

7. Threat Modeling & Vulnerability Assessment

Acceptance Criteria

- **[Deliverable]** Conduct a threat model (STRIDE/LINDDUN) for the API.
- **[Identify]** Abuse cases such as:
- resource/account enumeration

- unauthorized config/state changes
- partner impersonation
- misuse of presigned URLs (if applicable)
- **[Review]** Latest penetration test results and confirm remediation.
- **[Recommend]** A targeted API pentest if coverage is insufficient.

---

8. White-Glove Assessment Deliverables

Acceptance Criteria

- **[Deliverable] API Security Assessment Report** including:
- Authentication findings
- Authorization & entitlement findings
- Token lifecycle/correctness issues
- Gateway vs resource server enforcement mapping
- Runtime security gaps
- Observability/logging gaps
- Identified risks
- Prioritized remediation plan
- **[Deliverable] Entitlement Discovery Document**
  A catalog of all fine-grained entitlements that exist today across services and where they are enforced.
- **[Action]** Conduct alignment and review sessions with Product Owner & engineering team.
- **[Action]** Update customer-facing or internal documentation (AuthN/AuthZ behavior, claims, onboarding) to reflect validated security posture.

Lumen's Audience-Driven API Strategy

Executive Summary

To win in the digital marketplace, Lumen cannot offer a single, one-size-fits-all API. Our success depends on providing a tailored, world-class experience for each of our key customer segments: **Carriers**, large **Enterprises**, and **Direct/Digital-Native** customers.

This document outlines our proposed strategy for designing and exposing APIs for each audience. Our approach is to leverage industry standards like MEF and TM Forum for interoperability and enterprise completeness, while offering a simplified, proprietary API for customers who prioritize speed and developer experience.

This strategy will be implemented through our architectural cornerstones: the **Strategic Interface Layer (SIL)** and the **Partner Interface Layer (PIL)**, which will present the correct, purpose-built "API face" to each consumer.

The Core Principle: A Multi-Faceted API Platform

Our architecture is designed to present a specific, purpose-built API "face" to each audience from a common set of underlying capabilities. This allows us to meet the unique needs of each segment without creating duplicative backend logic.

API Strategy by Customer Segment

Carrier-to-Carrier: The Interoperability Standard

- **Recommendation:** Strictly adhere to **MEF LSO** `Sonata` **APIs**.

- **Rationale:** This is the non-negotiable global standard for inter-carrier automation. When another carrier wants to connect with Lumen to buy or sell services, they will expect a MEF-compliant API. This reduces friction, eliminates the need for custom integrations, and positions Lumen as a modern, easy-to-integrate partner.
- **Architectural Implementation:** The `Partner Interface Layer (PIL)` is responsible for exposing and consuming MEF LSO `Sonata` APIs.

Enterprise Customers: The Complete Digital Experience

- **Recommendation:** A combination of **MEF LSO** `Cantata` **APIs** and **TM Forum Open APIs**.
- **Rationale:** Large enterprise customers require a complete, end-to-end digital experience that covers both technical and business operations.
- **MEF LSO** `Cantata` provides the standardized, on-demand interface for their **technical teams** to programmatically order, configure, and manage network services.
- **TM Forum Open APIs** provide the standard interface for their **business operations** to handle commercial processes like service qualification, quoting, trouble ticketing, and billing.
- **Architectural Implementation:** The `Strategic Interface Layer (SIL)` will expose a combination of MEF and TMF APIs to enterprise customers.

Direct Customers & Developers: The Simplicity Standard

- **Recommendation:** A simplified, proprietary **"Lumen Native API"**.
- **Rationale:** For direct customers, developers, and DevOps teams who value speed and simplicity above all, a more opinionated and streamlined proprietary API can provide a superior developer experience (similar to the native APIs of PacketFabric or Megaport). This API should be designed for maximum ease of use and the fastest possible time-to-first-call.
- **Architectural Implementation:** This "Lumen Native API" will be a simplified facade built on the `Strategic Interface Layer (SIL)`. A

single, easy-to-use call to our native API will trigger a series of orchestrated calls to the underlying standard MEF and TM Forum APIs, hiding the complexity from the end-user.

Summary & Implementation

Audience	Recommended API Standard	Architectural Layer	Key Purpose
Carrier Partners	**MEF LSO** `Sonata` + **TM Forum Open APIs**	Partner Interface Layer (PIL)	Interoperability & Wholesale
Large Enterprise Customers	**MEF LSO** `Cantata` + **TM Forum Open APIs**	Strategic Interface Layer (SIL)	End-to-End Digital Experience
Direct / Digital-Native Customers	**Simplified Proprietary API**	Strategic Interface Layer (SIL)	Ease of Use & Developer Experience

The **API Governance** will oversee the creation, versioning, and documentation of these APIs. All public-facing API specifications will be published to the Lumen Developer Portal, with clear guides indicating which API is best suited for each use case.

API Security

1. The Problem: Why API Security is a Critical Business Risk for NaaS

As Lumen's strategy shifts toward API-driven products like Network-as-a-Service (NaaS), our approach to API security must evolve. For NaaS, our APIs are not just an interface—they are the control plane for our customers' critical network infrastructure.

A security failure would have catastrophic consequences, including:

- **Unauthorized Reconfiguration:** An attacker could reconfigure a customer's network, change BGP routing policies to hijack traffic, or

shut down critical ports, causing immediate and significant outages.
- **Data Exposure:** A breach could expose sensitive customer network topology and security details, which could be exploited in further attacks.
- **Denial of Service (DoS):** An attacker could use the APIs to launch DoS attacks by rapidly creating and deleting services, overwhelming the control plane and disrupting service for multiple customers.

The core question we must address is, "Can we afford security to destroy our reputation now that we are going with NaaS and APIs?". The answer is unequivocally no. This document outlines the current gaps and a proposed path forward to establish a robust API security governance model.

---

2. Current State: A Fragmented and Bypassed Process

Our current API security process is not undefined; rather, it is spread across multiple teams and stages, leading to inconsistency and a lack of enforcement. As observed by stakeholders, security currently touches five to six loosely connected control points:

1. **API Documentation & Guidelines**: Largely ignored as teams prioritize immediate delivery.
2. **Formal Security Review**: Often viewed as a bottleneck and is frequently bypassed.
3. **Automated Code Scans**: Tools are in place, but their results are often ignored or skipped.
4. **Enterprise Architecture (EA) Review**: A general design checkpoint, not security-specific.
5. **API Gateway Authentication Review**: A final check before publishing.

The primary challenge is not a lack of rules, but that **enforcement has been the main challenge always, not the lack of a guide**.

---

3. Proposed Path Forward: A Phased Approach

To address this, we must move from our current state to a target state with clear ownership and a standardized, enforceable process.

- **Phase 1: Assess and Document**

Inventory all systems, processes, and stakeholders involved in API security today to build a comprehensive view of the current landscape. This includes documenting complex scenarios like the Lumen partner and customer identity models. * **Phase 2: Rationalize and Standardize**

Once the current state is understood, rationalize our various security schemes (OAuth2, API Keys, mTLS, etc.) and establish a single, unified standard for different API archetypes. * **Phase 3: Govern and Enforce**

Establish clear ownership for API security governance and leverage automation to enforce our standards. This ensures that quality and security checks are integrated seamlessly into the development lifecycle, not treated as an optional gate.

---

4. Immediate Actions and Leadership Ask

To begin closing this critical gap, we need to take the following immediate actions:

1. **Create a Backlog Item:** Formally add "Define API Security Guidelines and Standards" as a strategic item in the API Governance program backlog.
2. **Signal the Gap:** Reserve a placeholder section in our foundational API documentation to make the need for security standards visible.
3. **Initiate Discovery:** Begin a lightweight discovery effort to baseline the current state, as outlined in Phase 1.

To succeed, this initiative requires executive sponsorship. We are asking the Governance Working Group to endorse this plan and help secure the engagement from senior leadership needed to create a dedicated API Security Standards Workstream.

Current State Assessment (AuthN, AuthZ, Terminology)

Story Summary

Document the *current state* of Lumen's API security model — including authentication, authorization, customer onboarding, and terminology — across all domains and customer types.
This baseline will identify current enforcement mechanisms, systems of record, and gaps before the introduction of standardized OAuth 2.1 and scoped access models.

Problem Statement

Lumen's API security landscape has evolved organically across multiple platforms and customer types (Partner, Enterprise, Federal, Internal).
Today, teams use inconsistent terminology (e.g., "API Key" = `client_id + secret`), and authentication/authorization responsibilities are distributed across multiple systems (Gateway, Resource Server, Onboarding, etc).
There is no unified documentation of *who authenticates*, *who authorizes*, or *what each term means*. This story establishes a comprehensive **current-state inventory** to support the design of future standardized security patterns.

Scope

In Scope:

- All externally (high prioirity) and internally (lower priority) exposed API domains (e.g., MCGW, Fabric, Billing, Provisioning, Customer360).
- Customer types: Partner, Enterprise, Internal, Federal.
- Systems involved in authentication, authorization, and onboarding.
- Terminology as used in current gateway policies, onboarding documentation, and partner contracts.

Out of Scope:

- Designing new OAuth 2.1 or OpenID Connect flows (Phase 2).
- Remediation or enforcement changes.

Deliverables

1. **Confluence Documentation:**
2. A dedicated page per *Domain* × *Customer Type* using the "API Security Current-State Template".
3. Completed sections for:
    - Authentication (AuthN)
    - Authorization (AuthZ)
    - Acting-on-Behalf-of / Partner–Customer Entitlement
    - Network & Transport Controls
    - Customer Onboarding
    - Attached **flow diagram** illustrating AuthN/AuthZ interactions and system responsibilities.
    - Evidence links to existing documentation.
    - **Terminology Glossary:**
4. Current-to-Standard mapping table (API Key ↔ client_id/client_secret, etc.).
5. Defined list of Lumen-specific terms and their formal equivalents (RFC 6749, 8705, 9068 alignment).
6. **System Inventory Table:**
7. Each system's role (Authentication, Authorization, Onboarding, Audit).
8. Integration points between gateway, onboarding, resource servers, and PKI.
9. **Summary Analysis:**
10. Key observations and known gaps.
11. Diagram or table summarizing enforcement responsibility (Gateway vs Resource vs Onboarding).

Acceptance Criteria

#	Acceptance Criterion	Verification Method
1	A Confluence page exists per active domain capturing all AuthN/AuthZ attributes using the standard template.	Review completed Confluence pages.
2	Each page includes a flow diagram showing systems and enforcement responsibilities.	Verify diagram presence and accuracy.
3	Terminology glossary page created and reviewed by API Governance & Security teams.	Cross-review against gateway configs and onboarding documentation.
4	System responsibility matrix completed for Gateway, Resource, Onboarding, IdP, and PKI.	Governance review.
5	Key gaps and inconsistencies documented and approved in governance log.	Review meeting minutes.

Story Outcome

When complete, Lumen will have a **governance-grade baseline** describing:

- Which systems perform authentication and authorization
- How customer onboarding issues credentials
- What terminology each team uses
- Where inconsistencies or gaps exist

This becomes the authoritative input to **Phase 2 – Security Standardization & Scopes Adoption**.

Details to Capture

Overview

Field	Description
Domain / Product	e.g. MCGW / Fabric / Billing
Customer Type(s)	Partner / Enterprise / Internal / Federal
Exposure Channel	Public-Internet / Partner-VPN / Lumen-Intranet / GovNet
Business Owner	Team or contact responsible

Authentication (AuthN)

Attribute	Description	Allowed Values / Examples
Entry Point	Where authentication occurs	API-Gateway / Direct-Service
AuthN Scheme	Mechanism used	mTLS / API-Key / JWT-Bearer / Basic / None
IdP / Issuer	Token or cert issuer	Internal-IdP / Ping / Keycloak / Partner-Provided
Credential Type / Store	Key / Cert / JWT and where stored	Vault / GW-KV / DB
Token Format	If JWT	Opaque / JWT (JWS) / JWT (JWE)
Token Audience (aud)	Expected audience	Free text
Token TTL (min)	Duration	Integer
TLS Version Min	Transport security level	TLS 1.2 / TLS 1.3
mTLS Verification Mode	Identity check method	SAN-DNS / SAN-URI / CN
Secrets Rotation Policy	Rotation cadence	≤90d / ≤180d / Ad-hoc
AuthN Owner System	Which system enforces	API Gateway / IdP / Hybrid

Authorization (AuthZ)

Attribute	Description	Allowed Values / Examples
AuthZ Model	Access-control type	None / RBAC (GW) / RBAC (Resource) / ABAC-Claims /

Attribute	Description	Allowed Values / Examples
		Endpoint-Allowlist / Tenant-Partition / Backend-ACL
Entitlement Source	Where rules reside	GW-Policy / Client-Profile / Partner-Contract / Backend-ACL
AuthZ Decision Point (PDP)	Who decides	Gateway / Resource / External-PDP / Hybrid
AuthZ Enforcement (PEP)	Who enforces	Gateway / Resource / Both
Gateway Enforcement	Coarse-grained policies	JWT-Signature / Client-Cert / Rate-Limit / Header-Policy / Size-Limit
Resource Enforcement	Fine-grained policies	Object-Level-ACL / Tenant-Filter / Partner-Entitlement
Entitlement Registry Source	Data source validating Partner↔Customer	Partner_Customer_Link / CRM / Custom DB

Acting-On-Behalf-Of / Delegated Access

Attribute	Description	Allowed Values / Examples
Acting-On-Behalf Model	Delegation type	None / Partner-On-Behalf-Of-Customer / Internal-On-Behalf-Of-Partner
Customer Context Assertion	How customer ID conveyed	Header (X-Customer-Id) / JWT Claim (customer_id) / Path Param
Partner-Customer Validation Source	Where entitlement checked	Entitlement DB / Gateway / Consent Service
Consent Mechanism	Customer consent model	None / Static Contract / Dynamic Consent / Open-Banking-Style
Audit Trail Completeness	Whether both IDs logged	Full / Partial / None

Network & Transport Controls

Attribute	Description	Allowed Values / Examples
Network Path / Zone	Boundary type	Internet / VPN / GovNet / Private-Link / Intranet

Attribute	Description	Allowed Values / Examples
IP / Geo Restrictions	Network restrictions	None / IP-Allowlist / Geo-Block / VPN-Only / GovNet-Only
FIPS Requirement	Crypto compliance	None / FIPS-140-2 / FIPS-140-3
Forward Secrecy Required	FS enabled	Yes / No / Unknown

Data Classification & Handling

Attribute	Description	Allowed Values / Examples
Data Classification	Sensitivity	None / PII / PCI / PHI / Mixed

Customer Onboarding Process

Attribute	Description	Allowed Values / Examples
Onboarding System	Source system	Portal / CRM / Manual Workflow
Onboarding Mode	How partner/customer is onboarded	Manual-Registration / Self-Service-Portal / Ticketed / Contract-Only
Credential Issuance	How credentials delivered	Portal-Download / Email / Automated API
Entitlement Recording	Where relationship stored	Partner_Customer_Link / CRM / Database
Rotation / Revocation Process	How keys or certs rotated	Manual / Automated / N/A
Governance Owner	Team managing onboarding	Security Ops / API Governance / Partner Mgmt

Current Auth Flows

There are 4 possible authentication mechanisms currently supported for External API integration.

There appear to be additional legacy flows in place

1. App Key and Digest authentication. (Note: This method is currently listed as deprecated and is in the process of being retired by the API

application teams).
2. Portal Framework (PFW) token based auth. This is an oauth2 based authentication flow used by Control Center for human interactions with the system.
3. Lumen Identity Access Management (LIAM) token based auth. This is an oauth2 based authentication flow used by Developer Center APIs for machine to machine interactions. (This is the go forward mechanism for API integrations).
4. APIGEE token based auth. This is an oauth2 based authentication flow hosted by APIGEE itself and is only supported for legacy app integrations.

It is critical to note that the 4 JWT flows are distinct and generate tokens independently, signed by different authorities.

Detailed sequence diagrams for the assorted flow can be found here: Integrating with LIAM APIs

For the purposes of this document I will maintain focus on the go forward LIAM auth flow as highlighted in the current developer center documentation.

There is a secondary auth flow that is similar to the one outlined below but instead adds the following additional steps:

- Initial client login occurs directly with Azure B2C using standard client_credential grant type with client credential and secret
- The B2C auth token is then presented to LIAM as client_assertion with client_id where it is validated by:
- Ensuring “azp” claim equals registered client’s clientId
- Uses clientId to fetch LIAM user details
- Validates that environment in returned user matches current environement

API credential creation

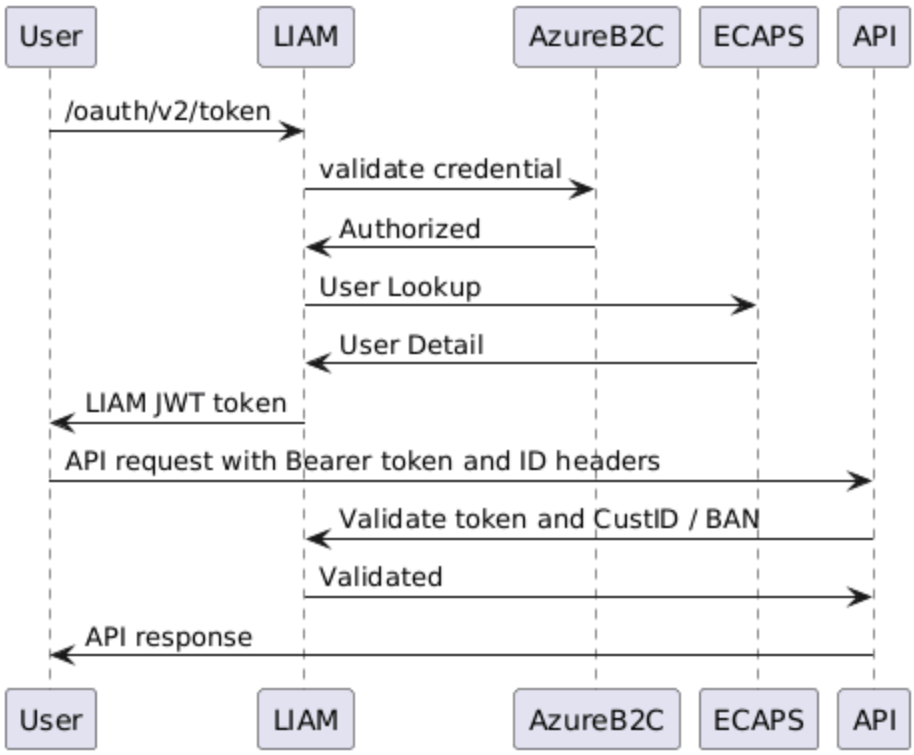
1. A control center user with developer permissions in their account can request to create and API application. At creation time the developer will:

2. Associate a set of BANs to the application.

3. Select token lifetime.
4. App creation occurs in AzureB2C and client credentials are returned to the user.
5. Once crential has been created they are sent to the `https://api.lumen.com/oauth/v2/token` using standard oauth client credentials grant_type.
6. Client credentials are sent to AzureB2C where they are authorized.
7. After authentication is validated a LIAM token is generated populating claims from it's own user info. This token is returned to the login requester.
8. Token is then sent in the Authorization header for subsequent calls to the API where the API may enforce additional Authz.

APIs may require additional headers to be sent as part of their Authentication requirements such as x-customer-number and x-billing-account-number

At a high level the flow appears as follows:



Secure API Access Model for Partner & Wholesaler Personas

User Story:

GNTSARCH-516

Getting issue details... STATUS

As an API Product Manager,

I want a secure, scalable, and standardized API security model for our Partner and Wholesaler personas,

So that they can securely access their end-customers' data to place orders and manage resources without forcing customers to share their API credentials.

Problem Statement / Business Context

The current "SELL" strategy for digital services has identified three key personas, but our security implementation has critical gaps:

- **Lumen Customer (Enterprise):** [SOLVED] This is the standard Enterprise Customer model.
- **Wholesaler:** [PARTIAL GAP] The model for accessing their *own* resources vs. their *customers*' resources is unclear, especially regarding billing identification.
- **Partner:** [CRITICAL GAP] The desired model is "access to lumen customer resources. Lumen Customer will be billed directly." The current "solution" is an insecure workaround that requires the end-customer to share their API credentials or use a complex portal login. This is a major security risk and a significant point of business friction.

Goal

This story's goal is to **design the technical solution** to close these gaps, focusing on providing a formal, delegated access model for the Partner persona and clarifying the Wholesaler model.

Acceptance Criteria (AC)

1. **[Design]** A formal **Capability Solution Document** is created that details the proposed security architecture.
2. **[Partner Flow]** The design MUST define a secure, 3-legged OAuth flow (e.g., Authorization Code Grant or a similar delegated authority pattern) for the Partner persona.
3. **[No Credential Sharing]** The solution MUST explicitly eliminate the need for end-customers to share their API credentials with Partners.
4. **[Wholesaler Flow]** The design MUST clarify the access and billing model for the Wholesaler persona, defining how they manage their own resources vs. their end-customer's resources.
5. **[Token Claims]** The resulting LIAM token MUST contain distinct, verifiable claims for the authenticated party (the Partner/Wholesaler) **and** the end-customer they are acting on behalf of (e.g., `partner_id`, `customer_ban`, `scope`).
6. **[Gateway Enforcement]** The API Gateway reference architecture is updated to demonstrate how to enforce this new, delegated authority model (i.e., validating claims for both the Partner and the Customer).
7. **[Onboarding Flow]** The solution must document the new "Partner Onboarding" and "Customer Consent" user experience (e.g., how a customer grants a partner permission).