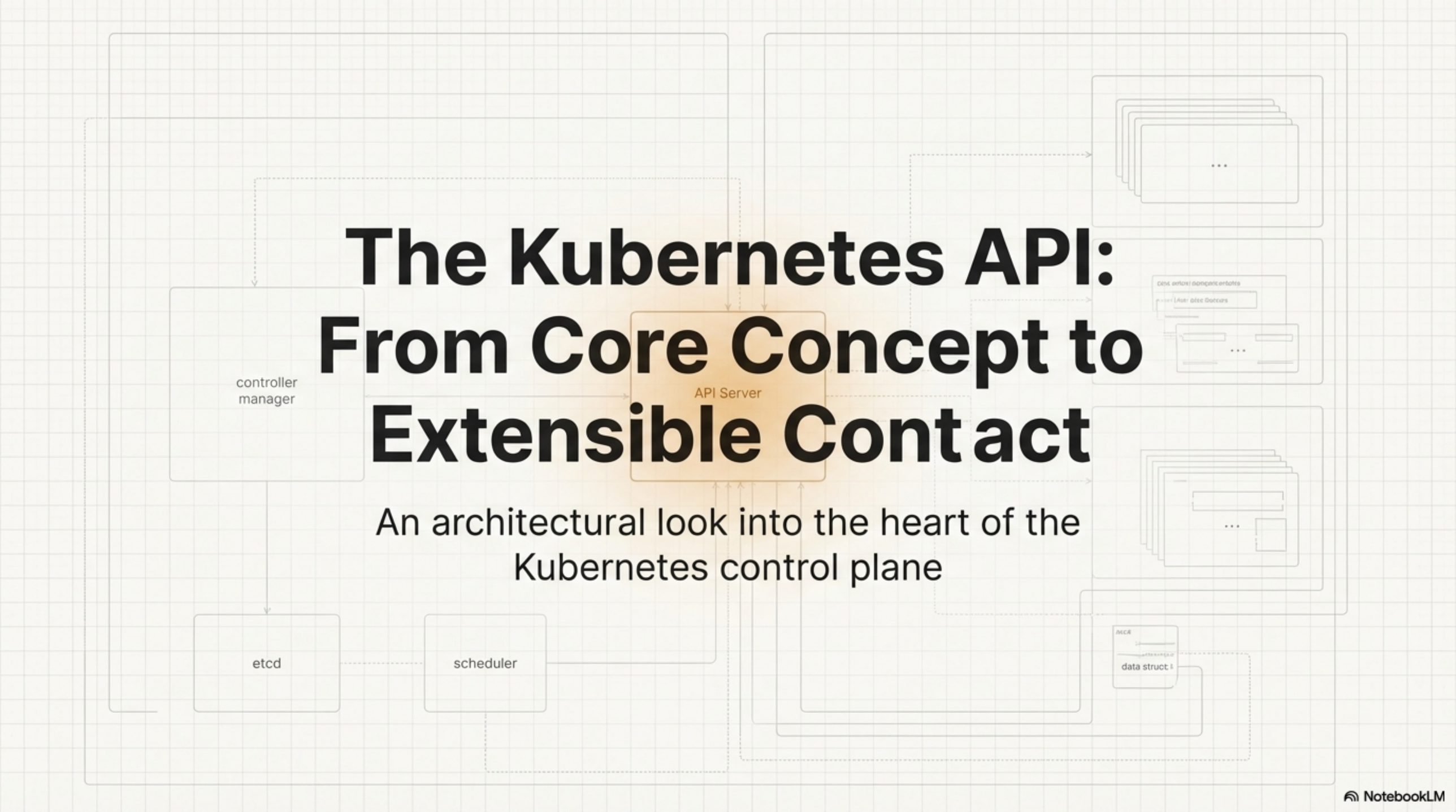
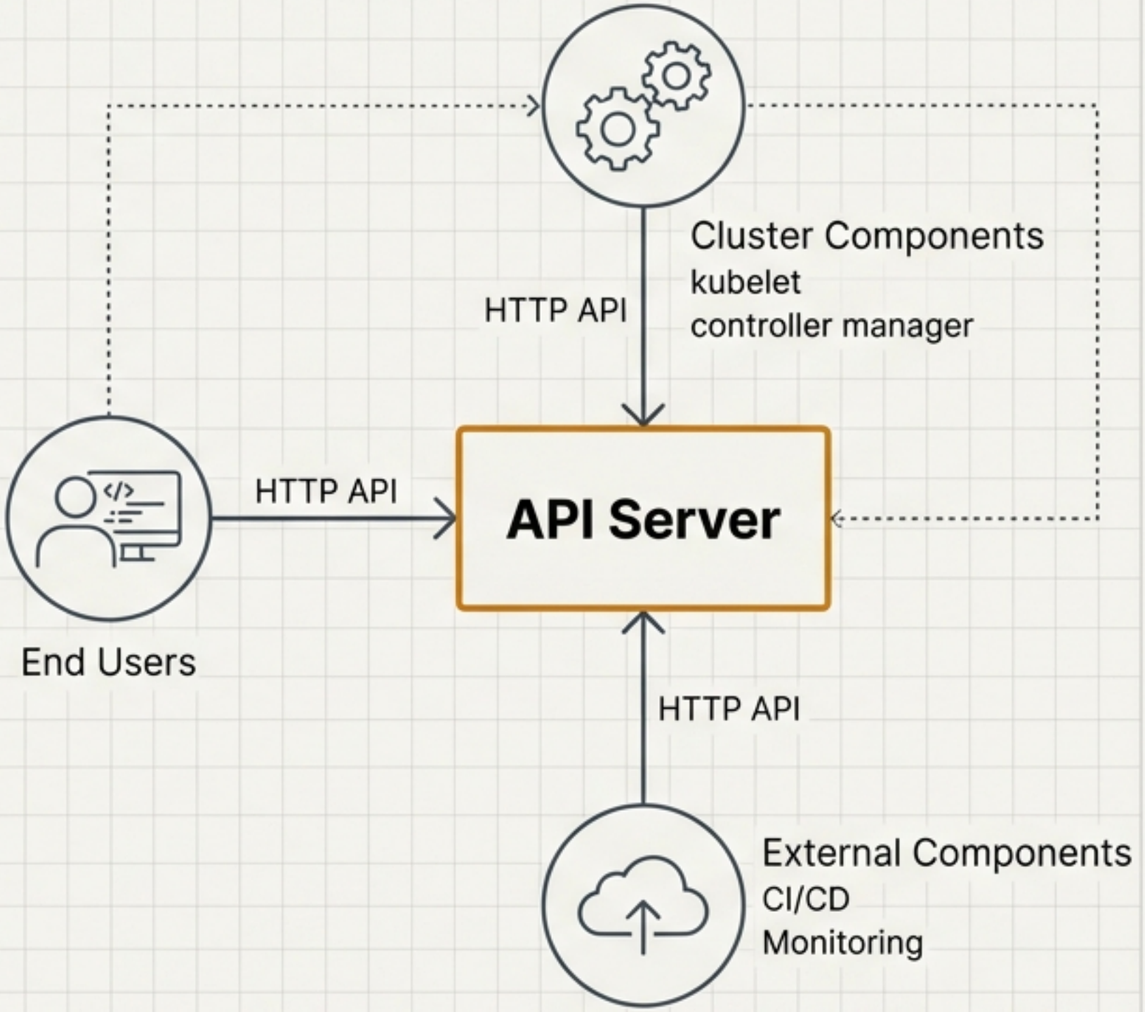


The Kubernetes API: From Core Concept to Extensible Contact

An architectural look into the heart of the Kubernetes control plane

The API Server is the Brain of the Kubernetes Control Plane

The core of Kubernetes' control plane is the API server. It exposes an HTTP API that serves as the central communication hub. All interactions—from users, cluster components, and external tools—flow through this single point. It is the mechanism through which you query and manipulate the state of all objects in Kubernetes.

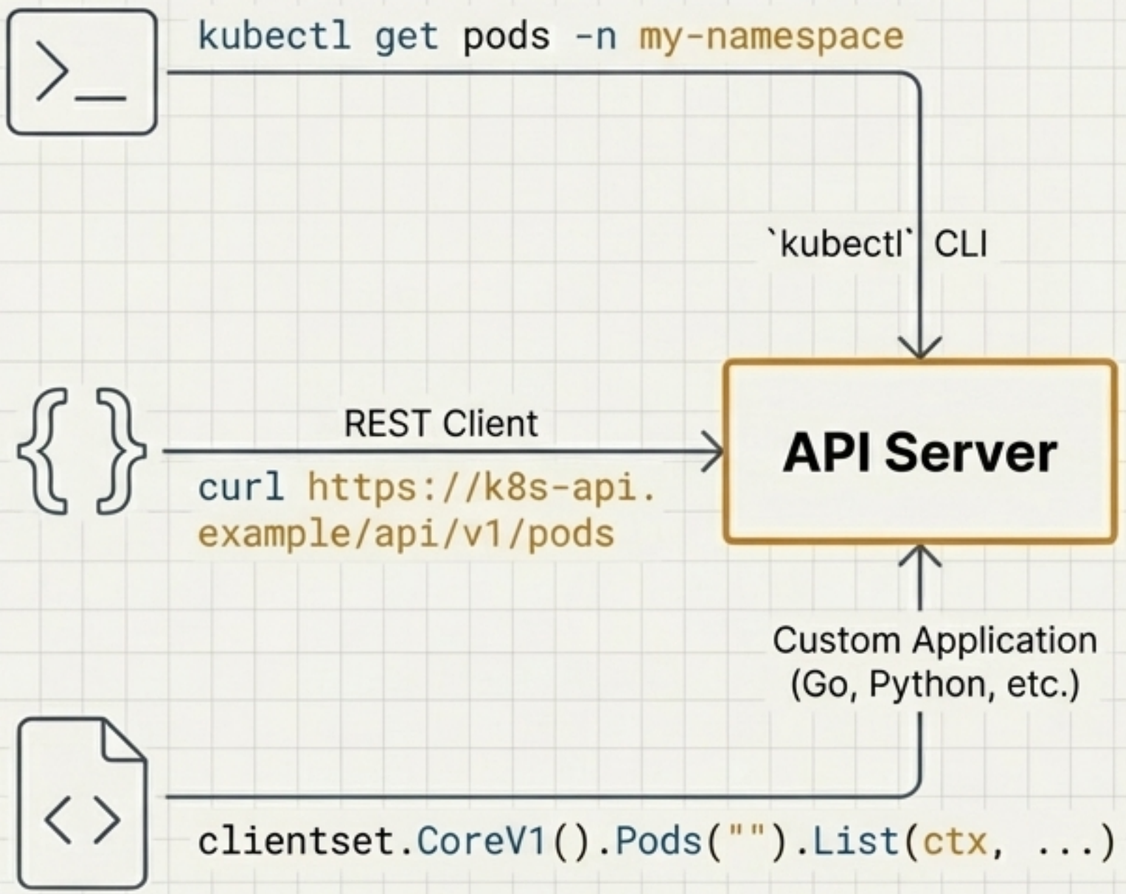


A Declarative Interface for Managing Object State

The Kubernetes API lets you query and manipulate the state of API objects like Pods, Namespaces, ConfigMaps, and Events.

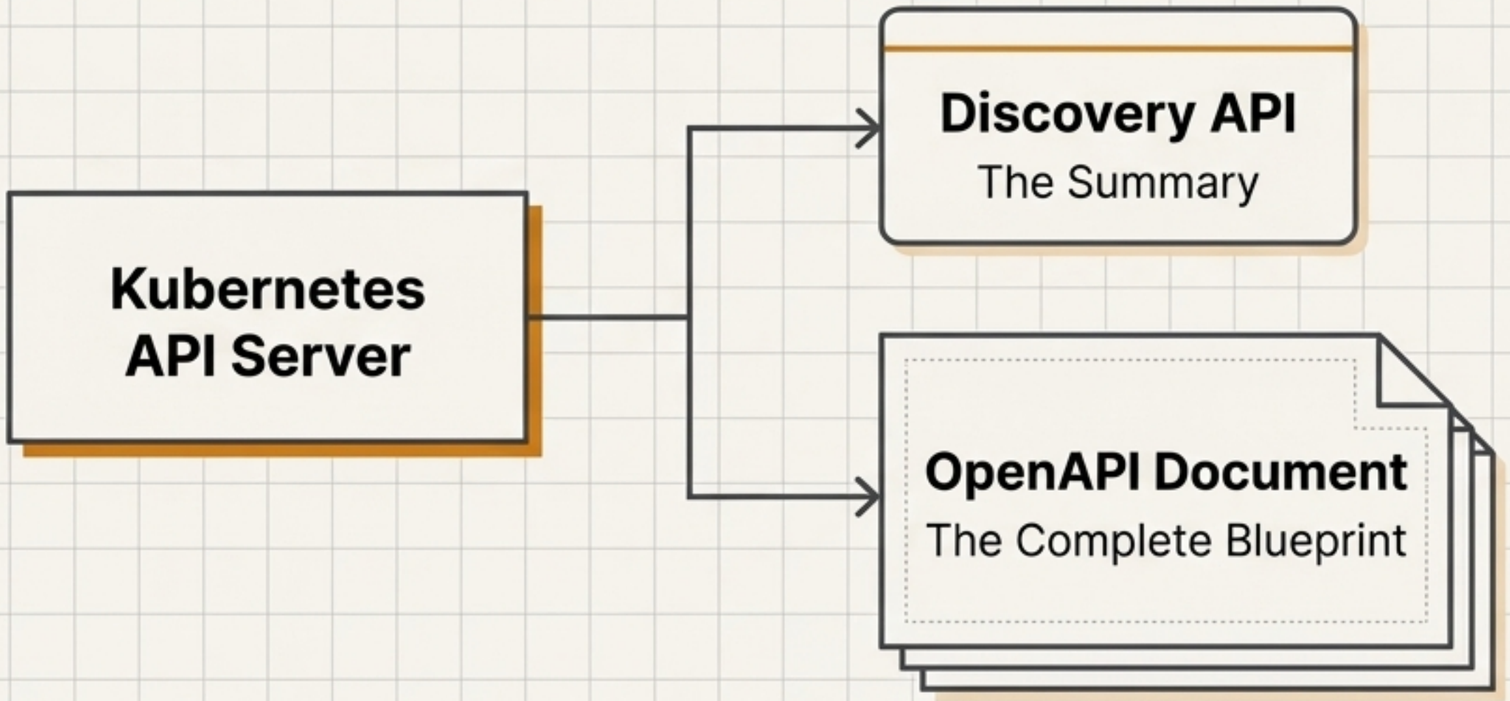
Most operations are performed through command-line tools that use the API.

However, the API can also be accessed directly via REST calls or client libraries for programmatic control.



How Kubernetes Publishes Its Own API Specifications

To enable automation and tooling (like `kubectl`'s command-line completion), every Kubernetes cluster publishes the specification of the APIs it serves. This self-description is achieved through two distinct mechanisms.



The Discovery API: A High-Level Summary of Resources

Purpose

Provides a brief summary of available API resources. It answers the question: “What groups, versions, and resources does this cluster support?”

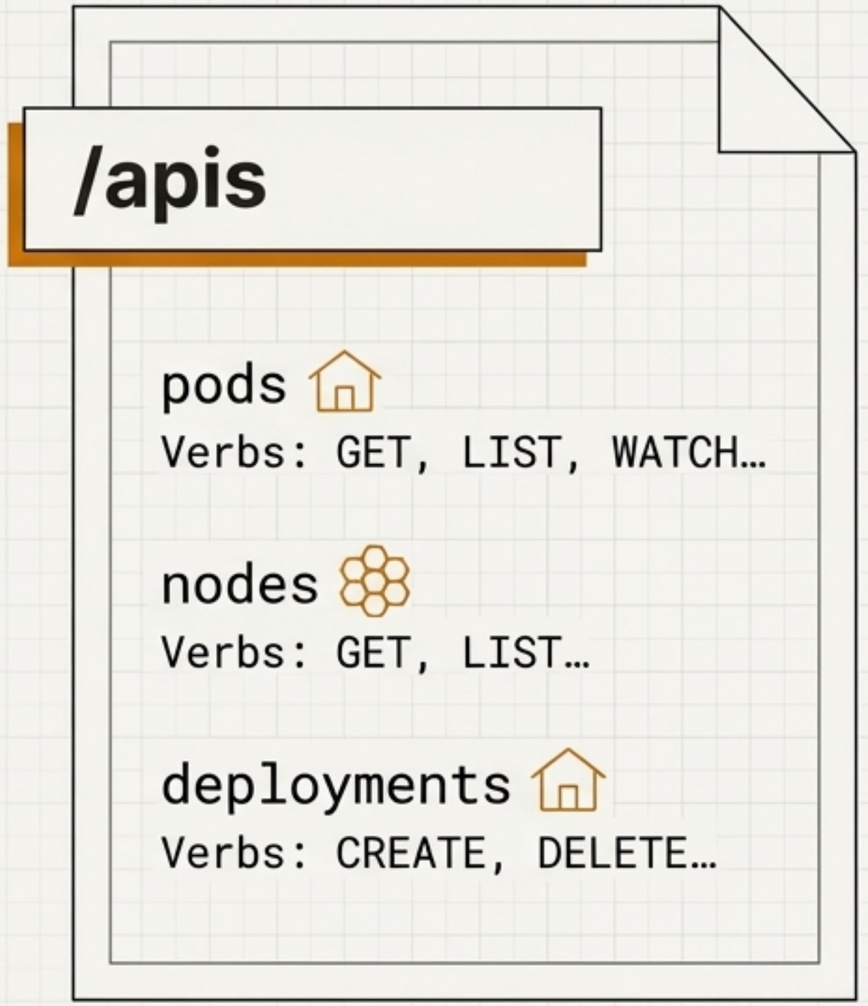
Information Provided

For each resource, it lists key metadata:

- Name (e.g., `pods`)
- Scope (Cluster or Namespaced)
- Endpoint URL and supported verbs (e.g., GET, LIST, WATCH, CREATE)
- Alternative names (e.g., `po`)
- Group, Version, and Kind (GVK)

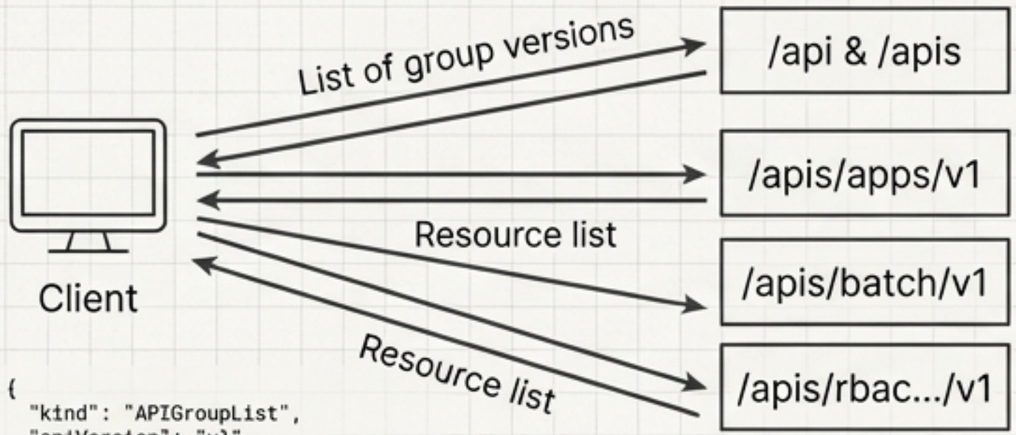
Key Takeaway

The Discovery API is a Kubernetes-specific API, separate from OpenAPI. It intentionally omits detailed resource schemas.



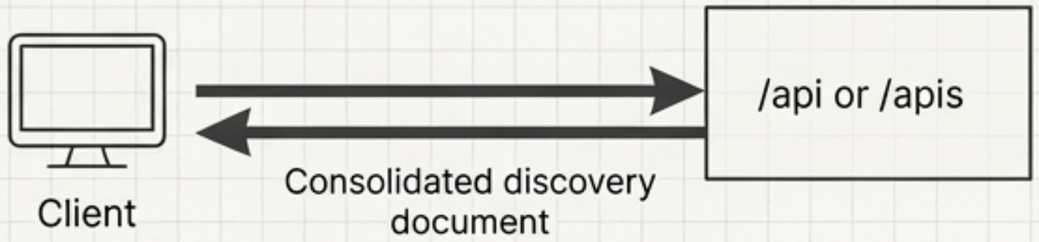
Two Modes of Discovery: Aggregated and Unaggregated

Unaggregated Discovery (The Legacy Method)




```
{
  "kind": "APIGroupList",
  "apiVersion": "v1",
  "groups": [
    {
      "name": "apps",
      "versions": [
        {
          "groupVersion": "apps/v1",
          "version": "v1"
        },
        // ...
      ]
    },
    // ...
  ]
}
```

Aggregated Discovery (The Efficient Method - Stable in v1.30+)



Drastically reduces requests. Enabled by sending a specific `Accept` header:

```
Accept: application/json;v=v2;g=apidiscovery.k8s.io;as=APIGroupDiscoveryList
```

Without this header, the endpoint returns the unaggregated document for backward compatibility.

The OpenAPI Document: The Complete API Blueprint

Purpose

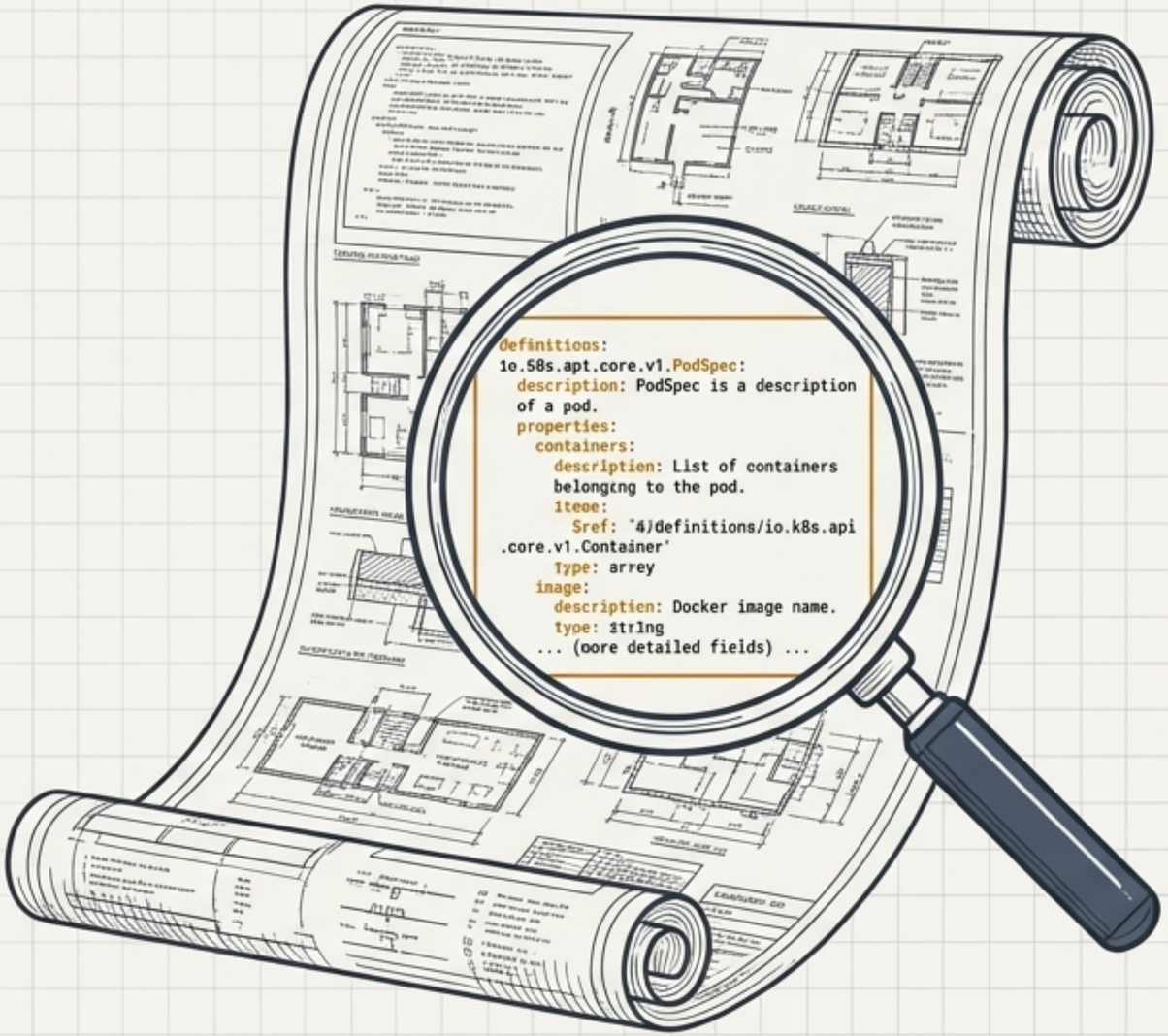
Provides full OpenAPI v2.0 and v3.0 schemas for all Kubernetes API endpoints. This is the exhaustive specification.

Information Provided

- All available API paths.
- Detailed schemas for all resources consumed and produced for every operation.
- Includes any extensibility components (like CRDs) that a cluster supports.

Key Takeaway

The data is a complete specification and is significantly larger and more detailed than the Discovery API.



OpenAPI V3 is the Preferred, Lossless Specification

OpenAPI V2

Endpoint: Served via `/openapi/v2'

Limitations:
Due to OpenAPI v2 constraints, certain fields are dropped from the schema, including `default', `nullable', and `oneOf'.

The validation rules in V2 schemas may not be complete. For precise verification, use `kubectl apply --dry-run=server'.

OpenAPI V3 (Stable in v1.27+)

Endpoint: Discovery at `/openapi/v3' lists all group/version URLs. Specs are at `/openapi/v3/apis/<group>/<version>?hash=<hash>'.

Provides a more comprehensive and lossless representation of Kubernetes resources.

Caching: Uses immutable URLs with hashes to improve client-side caching.

Kubernetes 1.34 publishes OpenAPI v2.0 and v3.0. There are no plans to support 3.1 in the near future.

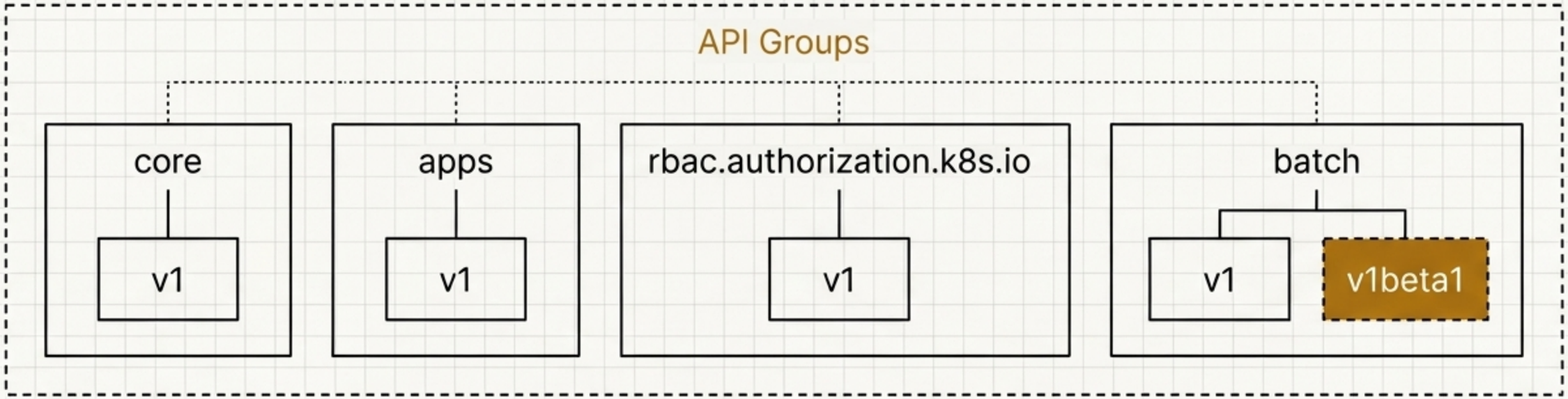
Discovery API vs. OpenAPI: Choosing the Right Tool

Criterion	Discovery API	OpenAPI Document
Purpose	Summarize available resources, groups, and versions.	Provide a complete, detailed schema for all API endpoints and objects.
Detail Level	Low. Lists names, scopes, and verbs. No resource schemas.	High. Full schemas, including fields, types, and validation rules.
Primary Endpoint	`/api` and `/apis`	`/openapi/v2` and `/openapi/v3`
Primary Use Case	Client-side discovery of supported resources (e.g., `kubectl` resource discovery).	Generating client code, building detailed dashboards, API validation tooling.

The Foundation of Evolution: API Groups and Versioning

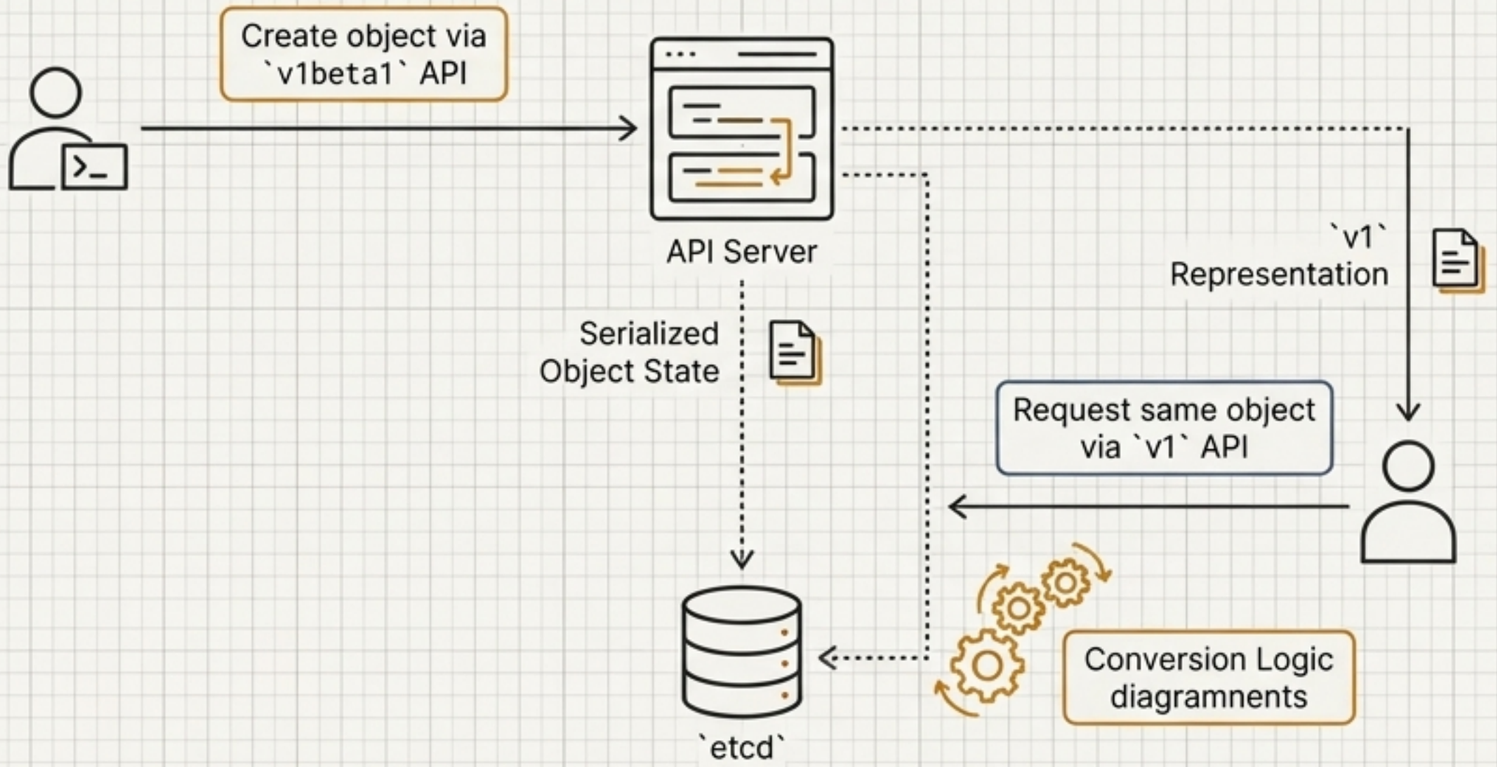
To evolve and extend its API safely, Kubernetes implements two key concepts:

1. **API Groups:** A collection of related APIs that can be enabled or disabled (e.g., `apps`, `rbac.authorization.k8s.io`). This allows for modularity.
2. **API Versions:** Each group can have multiple versions, each at a different path (e.g., `/api/v1`, `/apis/apps/v1`). Versioning is done at the API level, not the field level, to ensure a clear and consistent view of system resources.



The API Server Transparently Converts Between Versions

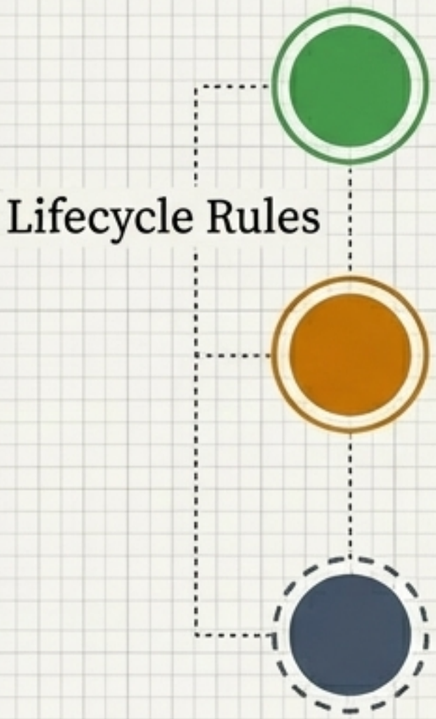
All different API versions are representations of the same persisted data stored in `etcd. The API server handles the conversion between versions transparently.



This allows you to create an object with a beta API and continue to access and modify it with the stable `v1` API after the feature graduates.

A Deliberate Approach to API Changes and Deprecation

Kubernetes makes a strong commitment to maintain compatibility for official APIs once they reach general availability (GA), typically `v1`.



GA (`v1`)
Highly stable.

Beta (`v1beta1`)
Compatibility is maintained for persisted data. You will need to transition to a subsequent beta or stable version, but objects remain accessible via both versions during the deprecation period.

Alpha (`v1alpha1`)
Experimental. No compatibility guarantees; may require deleting objects before a cluster upgrade.

Guiding Principle: New API resources and fields can be added frequently. Removing resources or fields requires following a strict API deprecation policy.

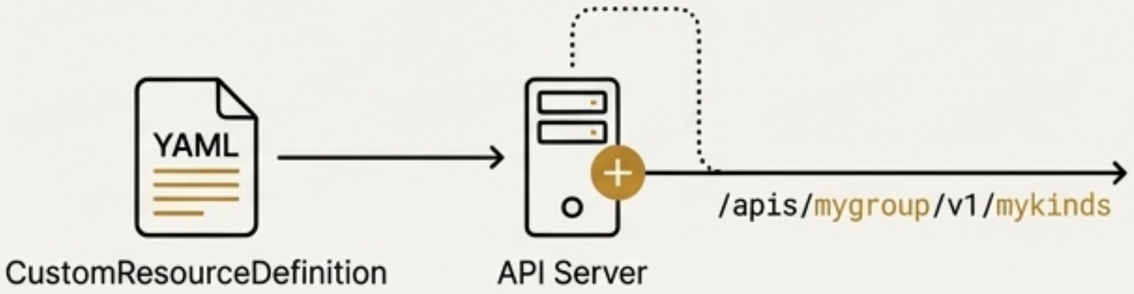
Extending the Core: Two Paths for API Customization

The Kubernetes API is not fixed; it is designed to be extended to meet custom needs. There are two primary extension methods:

1. Custom Resources (CRDs)

A declarative way to extend the Kubernetes API. You define a `CustomResourceDefinition` (CRD), and the API server provides a new resource API for your chosen Kind.

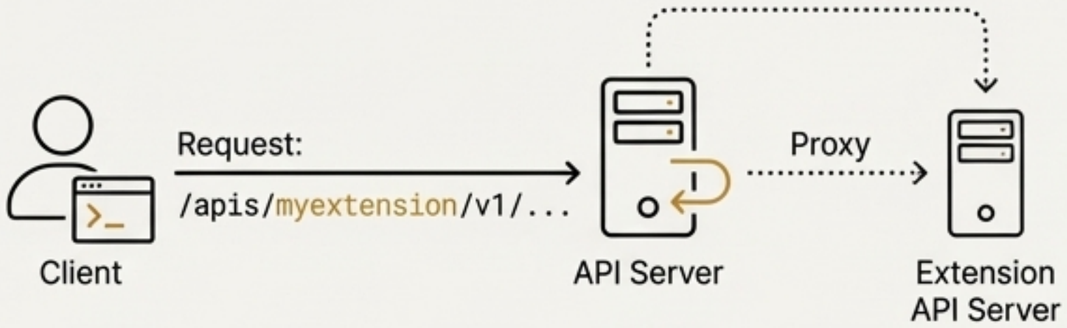
Best for: Adding new object types that behave like native Kubernetes resources.



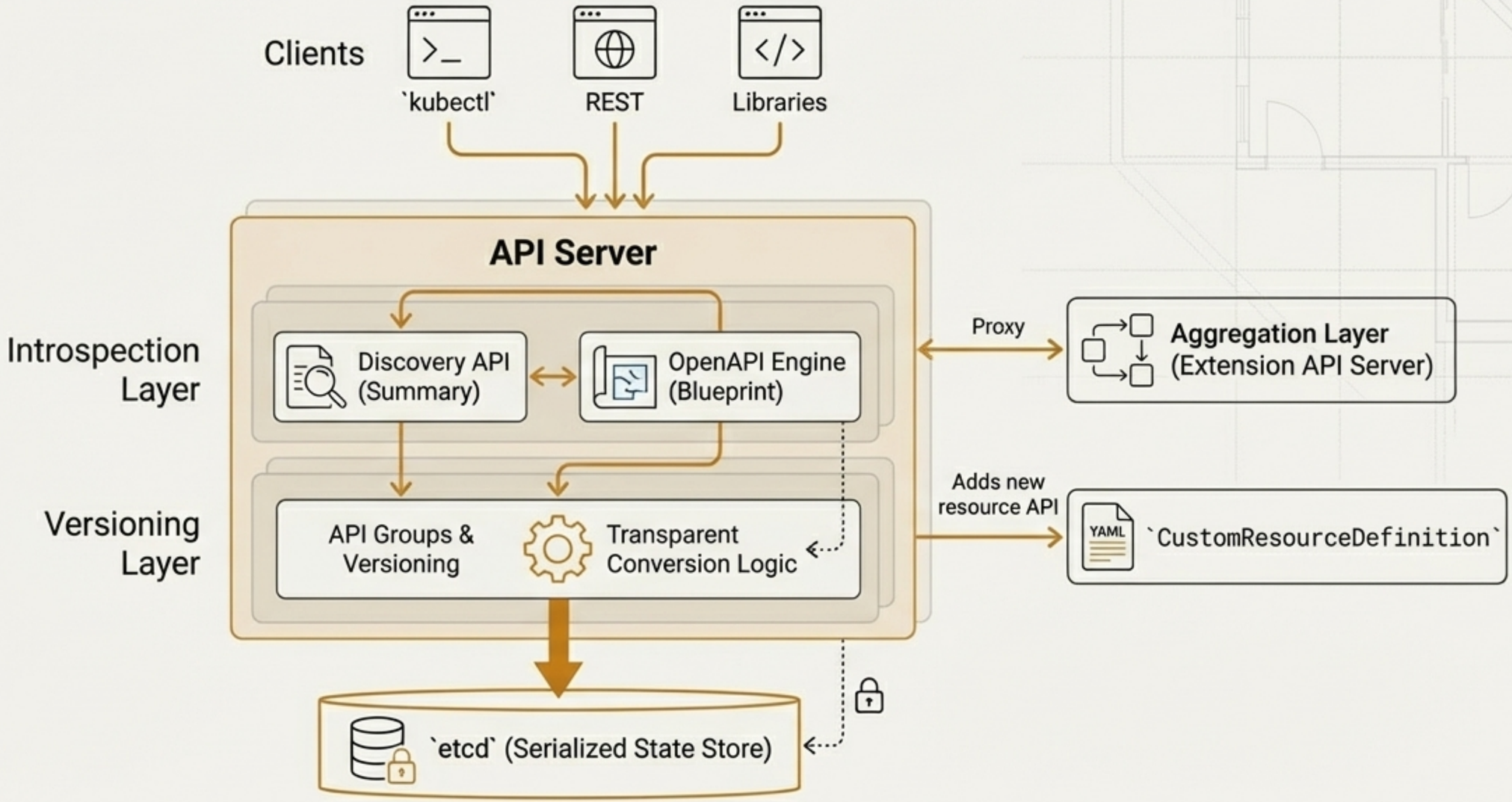
2. Aggregation Layer

A more powerful method where you run your own 'extension API server' that the main Kubernetes API server proxies requests to.

Best for: When you need complete control over the API's behavior or need to integrate an existing API.



The Complete Picture: A Cohesive Architectural System



Further Exploration

To deepen your understanding of the Kubernetes API, explore these key areas in the official documentation:

Extend the API

Learn how to add your own `CustomResourceDefinition`.

Secure the API

Read ‘Controlling Access To The Kubernetes API’ to understand authentication and authorization.

Browse the API

Use the ‘API Reference’ to explore endpoints, resource types, and samples.

Change the API

Understand what constitutes a compatible change by reading the ‘API Changes’ guide.