DRAFT - IT Project Guidance

Rationale for Queryable APIs

Version: 0.1

Purpose

This document presents the strategic rationale for adopting queryable APIs in government and enterprise information systems. It is intended for decision-makers responsible for systems investment, long-term service evolution, and integration strategy. The document outlines the limitations of current API practices, evaluates the merits of standards-based queryable interfaces, and recommends the adoption of mature technologies such as OData and GraphQL. Technical implementation details are deferred to appendices.

Synopsis

APIs have become foundational to modern system design, yet the majority of APIs deployed across government remain rigid and limited in scope. Typically designed around narrow, predefined use cases, these APIs are unable to support new analytical, operational, or integration needs without costly redevelopment. The result is an ongoing cycle of issue tracking, reprioritisation, and deferred delivery.

Queryable APIs address this systemic limitation. By exposing data through secure, standards-aligned interfaces capable of dynamic filtering, sorting, projection, and pagination, they allow authorised consumers to access the data they need in forms relevant to evolving business contexts. This approach enables interoperability, reduces cost of change, and supports data reuse across domains.

OData and GraphQL are the two dominant frameworks for delivering queryable APIs. Each has distinct technical strengths and deployment profiles. Their adoption mitigates key operational risks associated with hardcoded REST APIs and aligns with public sector obligations to prioritise international and industry standards.

Contents

Introduction

Application Programming Interfaces (APIs) have become a standard component of digital service delivery. They allow systems to expose data and operations to other systems in a controlled, documented manner. In recent years, their use has expanded significantly across central and local government, enabling integration between line-of-business systems, external partners, and analytical platforms.

However, most APIs in current use expose only narrow pathways to data. They are typically constructed around a limited set of anticipated use cases, implemented as fixed endpoints that accept a constrained set of parameters. These are often sufficient for transactional system integration or tightly scoped applications but become inadequate as business needs evolve.

APIs should not merely provide data access; they should provide data utility. The inability to dynamically query datasets constrains downstream users, impedes data-driven decision-making, and creates avoidable pressure on development teams. This challenge is especially acute in the public sector, where system funding is commonly limited to discrete project phases, and ongoing change requests are relegated to operational support backlogs.

As demand for integration, transparency, and responsiveness increases, APIs must support not only known interactions but also future questions. Systems must be designed for adaptability. This cannot be achieved by continually expanding the surface area of traditional REST endpoints. It requires a deliberate shift towards queryable interfaces grounded in well-established standards.

The sections that follow examine the current limitations of conventional API design, articulate the value proposition of queryable APIs, introduce OData and GraphQL as enabling standards, and provide a foundation for strategic adoption across enterprise system landscapes.

Current State: The Structural Constraints of REST APIs

The dominant API model across public sector systems remains RESTful APIs served over HTTP/S. REST, originally proposed by Roy Fielding in his 2000 doctoral dissertation, offered an elegant alternative to the complexity of SOAP and other earlier remote procedure call (RPC) paradigms. Its appeal stemmed from its simplicity, scalability, and alignment with the foundational web architecture. However, REST is an architectural pattern, not a formal standard. Consequently, it lacks defined conventions for essential API capabilities such as querying, versioning, filtering, and field selection.

In practice, this absence of specification has led to significant fragmentation. Individual developers, teams, and vendors implement REST APIs according to their own

conventions, with no consistent guidance on query formulation, result shaping, or change tracking. Even within a single organisation, it is not uncommon to find multiple REST APIs—each with different query semantics and inconsistent design philosophies.

As a result, data consumers are constrained by the specific endpoints that developers have anticipated. Common use cases such as retrieving a filtered subset of records, requesting a sorted list by attribute, projecting only selected fields, or accessing paged datasets often require custom endpoints. For example, separate endpoints may be required to:

- Retrieve a student record by national identifier
- Return all students in a given year level, sorted by surname
- Retrieve enrolments modified since a specified date

Each of these endpoints must be designed, implemented, secured, documented, tested, and maintained. This pattern scales poorly. As system usage grows and new analytical, operational, or policy-driven requirements emerge, the burden on development teams increases exponentially. In the absence of queryable interfaces, every new data access requirement becomes a development request.

This results in a familiar and costly delivery pattern:

1. A new requirement emerges—often from policy, oversight, or analytical stakeholders.
2. The API cannot fulfil the requirement without extension.
3. A ticket is raised, prioritised, and queued.
4. Development effort is scoped, resourced, and scheduled.
5. Delivery is delayed, often due to BAU constraints or reallocation of technical staff.
6. When eventually delivered, the solution may be narrowly targeted and of limited future value.

This reactive approach creates systemic inefficiencies. It also undermines confidence in digital platforms. Consumers—whether internal staff, partner agencies, or external reporting functions—become accustomed to delays, inflexibility, and partial workarounds. In parallel, developers experience backlog fatigue and architectural drift as APIs grow unwieldy with hundreds of narrowly-scoped endpoints.

More significantly, the assumption that business analysts can anticipate and document all required queries at project inception has proven unsustainable. Business requirements evolve continuously. In a typical system, meaningful new data access needs may arise months or years after initial deployment. Without queryable interfaces, systems must be re-extended each time. This imposes a structural cost and undermines digital service agility.

The following section explores the strategic importance of queryability in supporting long-term data access, reducing operational debt, and sustaining system responsiveness across diverse and changing business contexts.

The Case for Queryability

[todo intro]

Supporting Agility, Reuse, and System Responsiveness

The adoption of queryable APIs enables a fundamental improvement in how systems accommodate change. Rather than requiring new endpoints for every new question, a queryable API provides a flexible interface capable of supporting a broad range of authorised data access scenarios. This reduces technical overhead, accelerates responsiveness, and reinforces architectural integrity.

At the core of this capability is the delegation of query logic to the consumer, within the bounds of defined data contracts and security controls. When authorised consumers are empowered to retrieve the data they need—filtered, sorted, shaped, and paged to their requirements—the burden of fulfilling every future query shifts from the developer to the interface itself. This results in:

- Greater responsiveness to evolving analytical and integration needs
- Reduced cost and friction of change
- Improved reusability of data services across use cases
- Increased confidence in the long-term value of system investments

In the public sector, where information systems must operate within constrained funding models and deliver enduring value, this pattern is especially compelling. Most government systems transition from capital-funded projects to support-funded operations shortly after deployment. Once in support, the capacity to implement new endpoints is significantly reduced. Without a queryable interface, this transition often marks the end of the system’s ability to adapt.

Queryable APIs mitigate this risk by deferring the need for specific future development. They allow systems to remain operationally responsive even when project funding has ceased. Stakeholders can derive new insights, build new integrations, or construct bespoke views using the same foundational interface. This enables digital systems to serve a broader range of users, over a longer lifecycle, with significantly less maintenance burden.

From an enterprise architecture perspective, queryability also supports system composability. Modular services that expose queryable interfaces are more easily integrated into federated platforms, orchestrated workflows, or shared analytical

environments. This strengthens the system's strategic value and reinforces alignment with open data and service interoperability goals.

The following sections introduce the two dominant standards for delivering queryable APIs—OData and GraphQL—and explore their respective capabilities and deployment considerations.

Cost and Benefit: Lifting the Long-Term Burden

The introduction of queryable APIs represents a pivotal opportunity to break the cycle of rework, bottlenecks, and stakeholder frustration that characterises many government information systems. For too long, new data needs—whether driven by operational insight, integration demands, or analytical refinement—have been deferred, deprioritised, or abandoned altogether. Development teams are caught in a loop of designing and delivering narrowly-scoped endpoints, each tailored to a single query. Business units are left waiting, or worse, develop workarounds in spreadsheets or third-party platforms.

This pattern is not a matter of poor performance or bad faith. It is the inevitable result of designing systems that cannot respond to questions not already imagined. Queryable APIs interrupt that cycle. They allow consumers—within secure, defined boundaries—to access the data they need without waiting on development pipelines. This enables business agility, supports policy responsiveness, and significantly reduces the volume of minor changes that otherwise accumulate technical debt.

The return on investment is long-term and structural:

- Systems remain useful and responsive years after the initial build.
- Developers are freed from a backlog of trivial variations on existing queries.
- Business units gain confidence that data services will meet their evolving needs.
- Integration partners can build once, not wait for change requests.

The upfront investment in queryability is modest. In the .NET ecosystem, for example, OData support is mature and largely aligned with existing development practices. The additional effort lies in design and modelling—work that reinforces good architectural discipline and system governance.

The alternative—perpetuating the current model—means continued delays, growing disconnect between systems and business need, and the proliferation of shadow solutions outside the system of record.

Strategic Alignment and Analytical Readiness

Beyond internal productivity gains, queryable APIs support measurable improvements in organisational maturity. They promote the adoption of international standards (OData: OASIS, ISO/IEC; GraphQL: Open Specification) and satisfy government obligations to use open, non-proprietary protocols wherever available.

They also enable direct data access for authorised consumers, reducing the need to extract, duplicate, and warehouse data solely for the purpose of analysis. This has significant implications:

- Operational and policy decisions can be made against real-time or near-real-time data.
- Analysts and partners can explore hypotheses directly, within their permission scopes.
- Warehousing efforts can focus on longitudinal aggregation, not tactical reporting gaps.

This strengthens both data quality and decision-making agility. More importantly, it ensures that systems are not bypassed by the very users they are meant to serve. Instead of delivering data after the fact, systems become tools for real-time operational insight.

These benefits are not speculative. They are observable across jurisdictions where queryable APIs have been implemented with clear design guardrails and mature governance. For government systems expected to serve complex, evolving needs over decades—not years—this capability is no longer optional. It is foundational.

Strategic Recommendation and Next Steps

Government information systems are expected to support a wide array of stakeholders across long lifecycles. The current generation of narrowly-scoped REST APIs fails to meet this expectation. A shift to queryable APIs is both strategically aligned and operationally overdue.

It is recommended that all new and re-platformed services adopt a queryable interface model by default. Specifically:

- **OData should be adopted first**, particularly for internal domains where relational data, access governance, and type safety are well defined. It aligns with .NET Core development patterns and satisfies international standards obligations.
- **GraphQL should be considered** in contexts where data aggregation across multiple services, highly dynamic response shaping, or extensibility by third parties is required. GraphQL complements OData rather than replaces it.

To support this transition, organisations should:

- Establish architectural principles endorsing queryable interfaces as standard
- Invest in developer capability uplift around schema modelling and DDD-based design
- Review existing services and identify opportunities to expose queryable endpoints

- Include operational guardrails—query limits, observability, role-based projection—in all implementations

Queryable APIs must become the default approach for modern service design. Their adoption directly addresses the recurring delivery bottlenecks and governance limitations that impair many public systems today. Most importantly, they empower the organisation to deliver data utility—not just access—and enable service responsiveness long after initial delivery funding has ceased.

Standards-Based Solutions: OData and GraphQL

Two frameworks have emerged as the most mature and widely adopted solutions for delivering queryable APIs: OData and GraphQL. Each offers a structured approach to exposing flexible query interfaces, but they differ significantly in philosophy, technical model, and implementation footprint.

Standards-First Environments

Government agencies are mandated to prioritise international and sector-specific standards before selecting proprietary or internally-developed alternatives. This obligation supports consistency, interoperability, and long-term maintainability across government digital systems. In the context of queryable APIs, both OData and GraphQL meet this standard-first requirement, and crucially, they are not mutually exclusive in implementation. It is therefore recommended that agencies adopt a staged strategy: begin by exposing services using OData to meet standards-aligned obligations, and, where required by external integration or market demand, complement these with GraphQL interfaces.

OData (Open Data Protocol)

OData is an open standard developed initially by Microsoft and now stewarded by the OASIS standards body. It provides a REST-compatible query model layered over HTTP and supports features such as filtering, ordering, selection, expansion of related entities, and pagination. OData defines both a URL-based query syntax and a metadata format (typically via EDMX or JSON schema) to allow client applications to dynamically discover and consume services.

OData is particularly well integrated with the .NET ecosystem, where it can be automatically layered over existing API controllers and projected onto Data Transfer Objects (DTOs). Combined with LINQ, it enables highly expressive queries while maintaining strict control over what data is exposed and how it is shaped.

Key advantages of OData include:

- Alignment with REST principles and compatibility with existing HTTP infrastructure
- Mature support for filtering, sorting, projection, and paging using simple URL conventions

- Strong integration with typed languages and frameworks (notably .NET Core)
- Support for query validation, permission scopes, and automatic documentation generation

OData is well-suited to monolithic systems or services based on a single data context. It is most effective when the underlying data is well-defined, relational, and served by a consistent query engine.

GraphQL

GraphQL, developed by Facebook and now maintained by the GraphQL Foundation, presents a strongly typed query language that allows clients to specify the exact shape and structure of the response they require. Unlike OData, GraphQL is not bound to REST or URL patterns; it operates via a single HTTP endpoint that processes structured queries defined in a JSON-like syntax.

GraphQL's strengths lie in composability and federation. It is especially effective when integrating across multiple systems, aggregating disparate data sources, or constructing dynamic views tailored to individual client needs. This makes it highly suitable for microservice environments, extensible platforms, or public-facing APIs where consumer requirements vary.

Hence GraphQL is often promoted in the context of microservice architectures due to its ability to aggregate responses across loosely coupled services. However, it is important to approach this architectural model with caution in government and enterprise environments. Microservices offer real advantages where independent deployment, team autonomy, or diverse scaling requirements are genuinely needed. Yet in most public sector systems, these conditions do not apply.

In such cases, introducing microservices increases complexity without proportional benefit—leading to fragmented governance, duplicated authentication, distributed state, and inconsistent integration practices. It is therefore strongly advised that GraphQL be used to extend or complement stable monolithic services, rather than used as justification to fragment an otherwise cohesive architecture.

Further guidance on this topic is provided in *Appendix D: Evaluating Microservice Architecture in Government Systems.*

Key advantages of GraphQL include:

- Explicit control over response shape, reducing over-fetching and under-fetching
- Schema introspection and client-side tooling for dynamic application development
- Suitability for federated systems and distributed service architectures
- Strong community support across multiple languages and platforms

However, GraphQL may introduce additional complexity for access control, caching, and performance optimisation, particularly in environments with high variability in query patterns or limited control over consumer behaviour.

The choice between OData and GraphQL should be guided by system architecture, intended consumer base, and maturity of the development environment. In many government and enterprise settings, OData provides a highly effective solution for internal and strongly governed domains, while GraphQL offers flexibility for distributed, composite, or external-facing services.

Development Platform Implications

The ability to support queryable APIs is not just a matter of technical capability—it is intrinsically linked to the development platform chosen at the outset of a system's design. Platforms vary in how well they support open standards, modular data exposure, and long-term maintainability. This section contrasts two common approaches—custom development and no-code/low-code platforms—highlighting their implications for flexibility, governance, and cost. It emphasises that while custom development once implied high overhead, today it offers a sustainable, standards-aligned alternative that increasingly outperforms constrained, proprietary environments.

Custom Development as a Strategic Enabler

Queryable APIs, by their nature, require a development platform that supports fine-grained modelling, projection, and control over interface exposure—capabilities that are best achieved through custom development frameworks. While no-code and low-code platforms offer rapid delivery for simple workflows or internal forms, they are not well-suited to implement durable, queryable, domain-aligned service interfaces.

Custom development no longer carries the burden it once did. Modern frameworks—such as .NET Core and others—have matured significantly, offering stable libraries, first-class tooling, and decades of operational refinement. When paired with sound architectural governance and domain-driven design, custom-built solutions can be developed efficiently and maintained sustainably.

Moreover, unlike no-code platforms, custom stacks avoid licensing lock-in, offer full control over domain models, and enable developers to implement open standards like OData or GraphQL natively. They also allow API versioning, schema evolution, and secure data projection without reliance on proprietary tooling or external gateway solutions.

What makes custom development increasingly viable is not just maturity, but control. It empowers teams to shape solutions around well-understood domains, not platform constraints. It supports the implementation of rich integration surfaces from day one, and permits incremental extension as future needs arise—without retrofitting brittle workarounds or absorbing additional licensing costs.

No matter how visually compelling or “pre-built” a no-code solution appears, it cannot substitute for structured domain analysis, stable interface governance, and sustainable API design. Queryable APIs require precision, maturity, and adaptability—all of which are increasingly achievable through well-executed custom development.

Contrast to No-Code and Low-Code API Platforms

The case for queryable APIs becomes especially relevant when contrasted with the capabilities of many no-code and low-code development platforms—such as Salesforce, BizTalk, and other workflow or form-driven platforms commonly promoted for rapid delivery.

While these platforms often advertise API capabilities, they typically lack full support for open query standards such as OData or GraphQL. Even when APIs are available, they are usually constrained by platform-specific logic, tightly coupled data models, or abstracted middleware layers that impede projection, filtering, or pagination.

Additionally, these platforms generally do not support API versioning in a first-class, standards-based way. Implementing version management often requires costly workarounds using API gateways or additional services, introducing complexity and increasing licensing overhead.

The result is a familiar pattern: APIs are developed quickly to meet initial project requirements, but they do not scale to support evolving analytical or integration needs. When new queries emerge, they must be designed, prioritised, defined, developed, tested, and deployed—all within constrained post-project support environments, typically by BAU teams unfamiliar with the original system design.

Over time, the gap between system capability and business demand widens. Without queryability, the promise of agility becomes a mirage. Systems stagnate, users develop workarounds, and shadow systems emerge to fill the gaps.

Queryable APIs offer an alternative path. They provide the expressive flexibility typically absent in no-code platforms, without locking agencies into proprietary toolchains or escalating licensing commitments. More importantly, they do so using open standards and reusable investment in sustainable service design.

However Queryability requires the solution to be developed on custom stacks.

Implementation Considerations and Operational Risks

While the benefits of queryable APIs are significant, their implementation must be carefully managed to avoid introducing new operational risks. These risks are not reasons to avoid adoption; rather, they simply represent areas where thoughtful design and appropriate safeguards are required.

Query Scope and Resource Exhaustion

Poorly formed or unbounded queries can strain databases or application resources, particularly in GraphQL where clients define deeply nested queries. However, this is easily mitigated through enforced limits on query depth, item counts, and server-side execution policies.

Data Exposure and Domain Leakage

The flexibility of queryable APIs requires a disciplined approach to domain modelling and schema design. Without careful partitioning of internal and external representations, sensitive data may be unintentionally exposed. Queryability reinforces the need for domain-driven design, where bounded contexts and projections are used to expose only what is necessary and permitted. The inclusion or exclusion of nested objects can be controlled.

Variability in Consumer Behaviour and Usage Patterns

Dynamic query interfaces can lead to variable performance characteristics and unpredictable access patterns. This requires enhanced instrumentation, query profiling, and caching strategies to ensure stable service under load.

Complexity in Observability and Diagnostics

In RESTful APIs, endpoints are predictable and easily logged. Queryable APIs embed logic within query parameters or payloads, making traditional logging insufficient. Implementers must invest in tooling that supports structured diagnostics and usage analytics.

Increased Support and Developer Enablement Requirements

The flexibility granted to API consumers can increase the need for support, particularly among integration partners less familiar with query syntax. This necessitates strong onboarding materials, schema documentation, and diagnostic examples.

These risks are as real as any other form of development, however are quite manageable. Each has established mitigation strategies, and their presence should not deter adoption. Rather, they underline the importance of clear governance, robust schema design, and modern tooling. Queryable APIs are not inherently riskier than REST—they simply shift complexity from endpoint sprawl to interface flexibility. This shift, when properly managed, yields substantial gains in adaptability and long-term system value.

Conclusion

In the context of enterprise and government systems—where longevity, interoperability, maintainability, and cost-effectiveness are critical—OData emerges as the preferred

technical option for implementing queryable APIs. It offers mature tooling, robust query support, and direct alignment with international standards, fulfilling both operational needs and mandated obligations. For most service scenarios, the cost-to-benefit ratio of OData is superior, providing powerful flexibility with minimal complexity.

If OData proves difficult to implement on a selected technical stack, then the suitability of that stack should be questioned. Standards-based, queryable APIs are not niche capabilities—they are foundational. A platform that cannot support them with reasonable effort may not be appropriate for enterprise or government use.

Once OData has been adopted as the default queryable interface—particularly in systems with stable domains or type-safe environments—GraphQL may be considered as a complementary interface. Its inclusion is most appropriate where highly dynamic, consumer-driven queries or cross-service composition are required. Importantly, GraphQL should enhance—not replace—a well-established OData foundation.

for implementing queryable APIs. It offers mature tooling, robust query support, and direct alignment with international standards, fulfilling both operational needs and mandated obligations. For most service scenarios, the cost-to-benefit ratio of OData is superior, providing powerful flexibility with minimal complexity.

Once OData has been adopted as the default queryable interface—particularly in systems with stable domains or type-safe environments—GraphQL may be considered as a complementary interface. Its inclusion is most appropriate where highly dynamic, consumer-driven queries or cross-service composition are required. Importantly, GraphQL should enhance—not replace—a well-established OData foundation.

The case for queryable APIs is not simply a matter of technical preference—it is a strategic imperative. As systems age, funding cycles tighten, and service demands evolve, the ability to adapt without recurring redevelopment becomes central to digital resilience. REST APIs, implemented without queryability, limit that adaptability and contribute to a culture of deferred delivery and unmet need.

Queryable APIs provide a path forward. They balance structure with flexibility, governance with agility, and cost control with long-term utility. Their adoption aligns with both technical best practice and public sector obligations to use open, standards-based, interoperable technologies. More importantly, they enable systems to remain relevant to the users they serve—long after initial funding has expired.

The shift to queryability is not disruptive. It is additive. It builds upon existing investments, strengthens existing platforms, and clears the path for sustainable, modular growth. Agencies that make this transition now will position their systems—and their stakeholders—for greater responsiveness, better decision-making, and lower long-term cost.

Appendices

Appendix A - Document Information

Authors & Collaborators

- Sky Sigal, Solution Architect

Versions

0.1 Initial Draft

Images

Tables

No table of figures entries found.

References

There are no sources in the current document.

Review Distribution

The document was distributed for review as below:

Identity	Notes
Russell Campbell, Project Manager	
Gareth Philpott, Solution Architect	
Carl Klitscher, Solution Architect	
Vincent Wierdsma, Lead Developer	
Chanaka Jayarathne Senior Developer	

Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

Structure

Where possible, the document structure is guided by either ISO-* standards or best practice.

Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

Acronyms

API : Application Programming Interface.

GUI: Graphical User Interface. A form of UI.

ICT: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

IT : acronym for Information, using Technology to automate and facilitate its management.

UI : User Interface. Contrast with API.

Terms

Refer to the project’s Glossary.

Application Programming Interface : an Interface provided for other systems to invoke (as opposed to User Interfaces).

User : a human user of a system via its UIs.

User Interface : a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces (GUI) or Text/Console User Interfaces (TUI).

Appendix B: Implementing Queryable APIs Using OData

This appendix provides a high-level demonstration of how queryable APIs can be implemented using OData within .NET Core. The technical steps outlined below are intended for a developer audience, while the overall conclusions are relevant to architects and decision-makers. This is not an exhaustive implementation guide; detailed walkthroughs are available in supporting documentation.

This example demonstrates the setup of a queryable API using OData in .NET Core. The objective is to expose Student, Enrolment, and School entities using a single queryable endpoint, allowing filtering, sorting, paging, and projection over a defined DTO surface.

Steps

1. Define Entity Models: Student, Enrolment, School.
2. Define DTOs: Flattened views for external exposure (e.g., StudentSummaryDTO).
3. Create a DbContext and register DbSet<> for the entities.
4. Use AutoMapper or manual projection to map entities to DTOs.
5. Register OData routing and query options in Startup.cs.
6. Apply query guards (e.g., MaxTop, SelectExpand limitations).
7. Expose an endpoint like /odata/students and support $filter, $orderby, $select, $expand, $top, and $skip.

Observations

- Initial setup is straightforward within .NET environments.
- Projection and security are enforced using attributes and global query configuration.
- Once implemented, the endpoint supports a wide range of use cases without further code changes.

Appendix C: Implementing Queryable APIs Using GraphQL

This appendix offers a comparable example using GraphQL. As with the previous section, implementation steps are summarised for technically experienced readers, but the focus remains on comparative effort, governance needs, and sustainability. For production-grade implementations, reference documents and framework-specific best practices should be consulted.

This example demonstrates a similar implementation using GraphQL, exposing the same Student, Enrolment, and School concepts using a GraphQL schema and resolver pattern.

Steps

1. Define schema types for Student, Enrolment, and School using a GraphQL library (e.g., HotChocolate).
2. Define queries, resolvers, and data loaders.
3. Explicitly declare what fields are exposed and how they are resolved (e.g., nested queries to fetch enrolments by student).

4. Configure a root Query type and bind to a single endpoint (e.g., /graphql).
5. Add introspection support and GraphQL playground for development/testing.
6. Apply field-level guards and limits (e.g., max query depth, complexity scoring).

Observations

- Initial schema definition is more verbose and requires explicit resolver logic.
- Query shaping is highly flexible but needs more granular governance.
- New use cases often require schema extension or resolver updates.

Comparison

OData offers the optimal balance of functionality, simplicity, and standards alignment, making it the preferred default for queryable APIs. GraphQL remains valuable in scenarios requiring advanced composability and client-driven response shaping.

- OData integrates quickly and securely for internal, typed systems and requires less incremental effort over time.
- GraphQL excels in flexibility but incurs more upfront and ongoing effort to manage schema evolution, complexity control, and nested resolution performance.

Both approaches support queryability, but they differ in their emphasis and implementation effort. OData typically offers faster implementation and lower maintenance in systems with stable domain models, and is equally well-suited for both internal and external consumers. It supports structured filtering, sorting, and projection over well-defined endpoints, and can project fields through DTOs. While OData does not support arbitrary field renaming within query expressions, it enables ad hoc shape control through DTO design, view models, and selective exposure, which are sufficient in most enterprise use cases.

GraphQL is highly flexible, particularly in environments where clients need to construct arbitrarily shaped queries, rename fields dynamically, or combine data across multiple services in a single request. This flexibility, however, requires a more detailed schema and resolver setup, and typically results in more ongoing work to manage schema extensions, naming consistency, and nested resolution performance over time., particularly in environments where clients need to construct arbitrarily shaped queries or combine data across multiple sources. This flexibility, however, requires a more detailed schema and resolver setup, and typically results in more ongoing work to manage schema extensions and nested resolution behaviour over time.

The distinction is therefore not about internal versus external use, but about the level of query shaping freedom required and the complexity of data composition across boundaries. GraphQL is more suitable where external consumers require highly dynamic or federated data access.

Appendix D – Evaluating the use of Microservice architecture in Government Services

This appendix is a summary of a forthcoming document titled *IT Project Guidance – Considerations for Microservice Architecture in Public Sector Systems*. That document will provide a deeper analysis of when and how microservices may be appropriate, and what trade-offs must be acknowledged before their adoption.

While microservices are often associated with modern software architecture, their benefits arise in very specific operational contexts that are not commonly present in government settings.

Purpose and Scope of Microservices

Microservices were primarily developed to allow independently operating teams to release different parts of a larger system on separate cadences. This makes sense in large, globally scaled organisations with dozens of engineering teams working in parallel. However, this is rarely the structure of government service delivery.

Traditionally, such separation of deployment was handled through plug-in architectures—isolated modules running within a common framework. These allowed systems to grow or be extended without re-engineering the core. When this was not planned up front, modern teams often reach for microservices as a retrofitted workaround. In doing so, they externalise core application logic and orchestration into the cloud infrastructure layer.

Infrastructure-Led, Not Architecture-Led

The microservices movement has been significantly shaped by infrastructure vendors—particularly hyperscale providers like AWS—whose business model benefits from the adoption of highly distributed and deeply integrated services. The architecture promoted often involves:

- Externalising logging, orchestration, message brokering, and authentication
- Fragmenting data handling across queues, APIs, and ephemeral functions
- Replacing cohesive logic with low-friction modularity

While some of these patterns have merit in high-scale, high-change environments, they introduce considerable architectural and operational complexity. Without a strong central integration layer, services can become brittle, opaque, and highly dependent on proprietary infrastructure features.

Strategic Risks of Over-Adoption The long-term implications are often underestimated:

- Increased difficulty in integration and observability
- Steep learning curves for teams maintaining distributed interactions
- Higher risk of service coupling through shared configuration or service boundaries
- Greater exposure to vendor-specific failure modes and pricing volatility

Critically, the complexity introduced by microservices often exceeds the complexity of the problems they are meant to solve. In many public sector environments, microservices are not required to meet delivery, security, or team autonomy needs. Where plug-in extensibility or domain modularity is needed, these can be achieved within a well-structured monolithic service using modern modular design practices.

Conclusion

Microservices have legitimate use cases—particularly in federated environments where agencies or departments operate under separate delivery constraints, funding structures, or security models. In such scenarios, the ability to deploy, scale, or evolve services independently may justify the architectural overhead. Similarly, systems that are platform-based, offering extensibility to external developers or vendors, may benefit from a microservice model that supports plugin-style growth.

However, these are exceptions, not defaults. The broader public sector context typically involves centrally governed, synchronously deployed systems with limited team autonomy and aligned policy and data models. In these cases, a monolithic application with strong modular design offers simpler maintenance, faster onboarding, and greater architectural transparency.

It is important to distinguish hype from practice. Many microservice initiatives are driven by resume-driven development or vendor advocacy rather than an evidence-based need. When misapplied, microservices degrade system reliability, inflate support costs, and obscure core business logic behind layers of asynchronous tooling.

Architects should take care to evaluate not whether microservices are fashionable, but whether they are necessary. A clear test is this: can the business problem be solved using domain modularity, plugin extensibility, and well-structured APIs within a unified deployment model? If so, the additional complexity of microservices should be viewed as a liability, not an innovation.

Microservices should not be rejected outright, but their use must be carefully justified. They are not inherently superior—they are a solution to a deployment problem, not a requirement of sound architecture. In government systems, a well-structured monolith with clean boundaries, modular plug-in capacity, and queryable APIs will typically outperform a fragmented, service-heavy design over the system’s lifetime.