Computer Fraud and Security
ISSN (online): 1873-7056

Contract-Driven Modernization Framework: Migrating SOAP Services to GraphQL Interfaces in High-Traffic Platforms

Hima Bindu Yanala

Independent Researcher, USA

Abstract

Legacy service systems using the Simple Object Access Protocol are facing more challenges due to different consumer needs and a focus on mobile delivery. This is causing issues with how these systems work together, leading to problems like unnecessary data retrieval and difficulties in managing different versions, which slows down performance and hinders updates in busy environments. Proposed solutions GraphQL offers a different approach by using a schema-first, client-driven contract that allows for specific field-level queries and easy schema updates. Proposed trade-offs are discussed. To make high-traffic systems work well with GraphQL, it's important to plan carefully for contract testing, schema changes, fast resolver execution, and gradual management of changes to avoid problems like failures and downtime. This article explains a way to modernize systems by using a GraphQL gateway and a custom resolver layer in front of the current SOAP services, which helps to smoothly transition and work alongside existing systems. The controls include keeping track of contracts, designing schemas that fit specific contexts, managing how and when to phase out old features, overseeing query costs, testing contracts based on consumer needs, and monitoring The controls are designed to allow high-traffic platforms to gradually update their key interfaces with behavior equivalence, production stability, and sustainable maintenance.

Keywords: Service Modernization, Graphql Schema Evolution, Contract-Driven Development, Query Cost Governance, Backward Compatibility

1. Introduction to Legacy Service Interface Constraints and Modernization Imperatives

For many years, enterprise platforms have frequently adopted a SOAP-based service layer to support their integration architecture. Such interfaces may have been fit for purpose at the time of implementation, but as the diversity of consumers, first-class mobile delivery, and the adoption of agile development practices increase, the architectural challenge for service teams is how to evolve such interfaces rapidly while keeping the production systems stable [1].

SOAP contracts are fundamentally oriented around operations. A service exposes a fixed set of strongly typed messages that are exchanged between a client and a server using WSDL (Web Services Description Language) documents. But it's this rigidity that becomes a disadvantage as teams attempt to serve heterogeneous clients, with each having different data needs (e.g., overfetching, which is the process of retrieving more fields than any one client actually uses). The compound effect of inefficiencies within cycles of processing has been documented in the domains of motor learning and distributed systems. In the literature concerning motor sequence training, Bönstrup et al. (2019) found that 95% of the entire learning benefit occurred within the first eleven practice trials. Like practice trials, coupled with an unflagged efficiency drop during the construction of the payload serialization in SOAP-centric frameworks, downstream latencies can be very pronounced in a high-throughput context when the payload size has a direct effect on network buffer consumption and garbage collection in Java Virtual Machine-based runtimes. .

If the product continues to grow, the developer is left with the option of adding optional fields to existing operations, adding bloat to the payload and ambiguity to the parser, or maintaining multiple versions, increasing the testing and support effort. The specification for SOAP 1.2 itself provides VersionMismatch fault codes and Upgrade header blocks for this situation, as version mismatch is a common failure case in production deployments [1]. In highly scaled systems serving tens of thousands of requests per second and hundreds of downstream consumers, deploying even a small catalog of active services would require important continuing investment in regression testing, documentation, and consumer coordination. .

These three needs are now the main reasons for updating application interfaces from SOAP to better support mobile devices with slow internet, allow faster changes on the server without needing to sync with users, and create a unified interface that explains itself. GraphQL solves these three needs by employing a radically different interface contract. A

phased, contract-driven approach introduces the GraphQL gateway and resolver layer in front of existing SOAP services to support both interfaces and migrate service consumers over time.

2. GraphQL as a Client-Driven Interface Model: Architecture and Operational Considerations

GraphQL was originally developed as a data query language for use within Facebook before being released as an open specification. GraphQL is a schema-first, contract-based query language, meaning that clients can precisely specify the shape and structure of the data returned by the server. Unlike REST or SOAP, where servers define the response structure, in GraphQL the schema defines a typed graph of all possible fields. Clients define a selection set to request the fields that should be resolved. Hartig and Pérez (2018) proved that GraphQL evaluation is NL-complete but also that query responses can be enumerated with constant delay—meaning a server can answer a GraphQL query and transmit the response byte-by-byte while spending only a constant amount of time between every byte sent. This suggests that we can solve the evaluation and enumeration problem efficiently in production deployments. [3]

The model has a strong architectural impact. At the transport layer, all GraphQL operations, including queries, mutations, and subscriptions, are typically sent over a single HTTP endpoint. This makes load balancing and edge caching easier than it would be using REST, where the proliferation of resources can make cache key design difficult. This does mean the application is responsible for query complexity. In tests of GitHub's public GraphQL endpoint, Hartig and Pérez found that Walk's response is exponential in the nesting level (i.e., for levels 1 through 7 for traversal queries). They discovered that timeout errors are returned after approximately 10 seconds of a request for nesting levels 6 and 7 if the branching factor is 5. GitHub also prevents queries nested more than level 25 and returns more than 500000 nodes. Such restrictions are inadequate and often overly conservative.

From a security and operational perspective, the flexible structure of GraphQL exposes an additional attack surface, which must be reduced at the resolver level rather than the endpoint level. For example, a study by Ferry et al. (2015) of the 50 most frequently visited sites in Ireland found that 21 of these sites implemented OAuth-based access delegation; 2 of the 21 (10.5%) were vulnerable to high-impact vulnerabilities, such as account hijacking (as a result of OAuth implementation) [4]. These principles are also relevant for GraphQL field-level authorization, such as starting by disabling schema introspection in production and enforcing access control on all publicly available fields.

The schema can act as a rich, live contract that can drive automated client code, interactive documentation to aid exploratory use, and contract-test automation without the documentation bloat of legacy SOAP interfaces.

Nesting Level	Response Size Trend	Timeout at Branching Factor 5	Restriction Enforced
1	Baseline	No	No
2	Low growth	No	No
3	Moderate growth	No	No
4	High growth	No	No
5	Exponential growth	No	No
6	Exponential growth	Yes	No
7	Exponential growth	Yes	No

Table 1: GraphQL Query Nesting Depth and Observed Server Response Behavior [3, 4]

3. Contract Inventory, Domain Boundary Definition, and Schema Planning

The first step in any contract-driven modernization is to produce a strong inventory of all existing SOAP operations, their message schemas, the traffic they produce, and their dependents. This provides the empirical baseline that the new interface implementation must offer behavioral equivalence to. The process is conceptually similar to refactoring code. Fowler defined refactoring as improving internal code structure without changing its external behavior. In the case of a SOAP service catalog, this means that the architect needs to gain a deep understanding of how the current internal structure maps the external behavior before changing the interface definition to achieve the same observable behavior.

The contract inventory extracts all the WSDL and XSD artifacts of the currently available service endpoints. It then analyzes the cardinality of the input and output fields and compares them with the discovered usage of the service consumers to identify ignored fields. Based on Fowler's simplification principle and pre-existing design debt due to compatibility requirements with legacy systems, the fields that appear in the WSDL contracts but are not read (or used operationally) become candidates for systematic refactoring [5]. The output of the analysis is a prioritized catalog differentiating good candidates for modernization vs. operations that would incur excessive short-term costs. Error rates and latency percentiles are recorded as regressions in future steps.

Domain boundaries define the mapping between the operational catalog and the logical schema structure based on stable business entities and their relationships rather than SOAP operations mapped to GraphQL schema types. By Evans's definition of domain-driven design, a domain model should represent the conceptual structure of the problem domain as it is understood by domain experts, expressed using a shared ubiquitous language [6]. Therefore, GraphQL types should represent domain concepts rather than operational abstractions in SOAP services.

Evans's bounded context is a relevant concept for schema design [6]. A bounded context is a logical boundary where a domain model has internal validity and authority. In this version of the pattern, every domain team owns the schema part of their bounded context, and the aggregate boundaries that Evans describes are also, as a result, the boundaries inside the schema. This means that the query can easily determine which team owns the resolver and also means that the budget for performance is owned by the team with the most knowledge of the underlying data access pattern.

Inventory Criterion	High-Value Modernization Candidate	Deferred Candidate
Field cardinality vs. consumer usage	Fields present in WSDL but unused by consumers	Fields actively consumed across clients
Design debt type	Compatibility-driven legacy fields	Functionally required retained fields
Refactoring eligibility	Candidate for systematic removal	Requires dependency resolution first
Error rate profile	Low and stable	Elevated or inconsistent
Latency percentile trend	High tail latency observed	Within acceptable service thresholds
Migration priority	High	Low to medium

Table 2: SOAP Operation Modernization Prioritization Based on Contract Inventory Criteria [5, 6]

4. Schema Evolution, Resolver Efficiency, and Backward Compatibility Strategies

In a GraphQL environment, the additive compatibility principle maintains backward compatibility because the changes to the schema are additive, i.e., the behavior of existing query results is unchanged. Hartig et al. specify a formal semantics of GraphQL, wherein the result of evaluating a selection set against a typed schema is defined entirely by the fields of the query document that is sent by the client [3]. The formalism also justifies the operational soundness of additive evolution: if additive evolution is applied to add a new field to a type, then the evaluation semantics of any query that does not select this field remain unchanged. Because no request that does not include the new type in the select set is affected, GraphQL schemas are more easily extended in a non-breaking manner than the fixed-operation contracts of SOAP, where modifying a shared message type requires all downstream consumers to modify their implementation.

For breaking changes that involve removing a field, or changing the type, arguments or semantics of a field, the recommended approach is to instead add a new field with the desired behavior and mark the original field as being @deprecated by using the built-in @deprecated directive with a reason for migration, and then finally remove the old field after an appropriate deprecation window, during which consumer usage telemetry can be used to confirm that consumers have migrated. Hartig et al. showed the effect of the structural depth and shape of selection sets on the formal complexity of evaluating GraphQL queries, justifying the practice of tracking field-level usage during a deprecation window, as persisted usage of deprecated fields often indicates consumers issuing structurally complex queries that are slower to refactor [3].

Resolver efficiency is one of the most important engineering issues for GraphQL in large-scale production environments. According to Newman, on distributed service architectures, the bottlenecks are from chattiness between services and the lack of coordinated data fetching between service boundaries [7]. These problems manifest in GraphQL as the N+1 query problem, where querying for a list of parent entities results in a single downstream call for each parent entity to resolve fields on child entities. The DataLoader pattern is intended to help aggregate these individual entity lookups within a single request execution context into a single batched call per data source. Other than batching, Newman's microservice architecture favors multi-tier caching via an in-process cache, a distributed cache, and a response cache at the gateway. The in-process cache can be used for relatively static reference data. The distributed cache can be used to cache state shared among multiple services. The response cache can be used for deterministic queries when a result can be served in a time-to-live window [7].

In addition to schema changes, backward compatibility rules extend to resolver behavior. For example, if a GraphQL resolver forwards the call to a legacy SOAP service, it should provide the same field mappings, null handling, default value behavior, and error translation semantics as the legacy service. Even when the schema contract is satisfied, violations of this behavior are a regression from the perspective of the consumer, so resolver-level contract testing is a key complement to schema compatibility.

Schema Change Type	Compatibility Impact	Consumer Effect	Required Action
Add new field to existing type	Non-breaking	No existing query affected	Deploy directly
Add new object type	Non-breaking	No existing query affected	Deploy directly
Remove existing field	Breaking	Queries selecting field will fail	Add replacement → deprecate → monitor → remove
Change field return type	Breaking	Response shape altered for consumers	Add replacement field → deprecate original
Modify argument definition	Breaking	Existing query arguments rejected	Add replacement field → deprecate original
Alter field behavioral semantics	Breaking	Consumer output silently changes	Mark @deprecated with migration reason

Table 3: GraphQL Schema Change Types Classified by Compatibility Impact and Deprecation Workflow Action [3, 7]

5. Governance Controls, Consumer-Driven Testing, and Query Cost Management

Enterprise GraphQL interfaces require a multi-layered governance solution to consumers and implementations for preventing resource-heavy queries from harming shared enterprise health, enabling safe schema evolution without unintended breaking changes, and providing consumers with the information they need to prevent their queries from violating enterprise platform policies [8].

Query cost governance begins with a static analysis performed on the gateway. Before executing a query, the gateway computes a complexity score by traversing the selection tree and summing the weight of the fields whose resolvers will cause downstream calls with the estimated size of the list returned from the resolver. Queries exceeding a configured complexity limit receive a structured error message, indicating the offending fields, and tips for reducing the complexity. Limits on structural depth prevent queries with pathologically nested fields from exhausting call stack space or exponential result sets. In high-traffic environments, persisted queries are an even stronger operational guarantee, where clients register their queries with the gateway at build time and reference them by hash at runtime, meaning only a pre-approved and pre-analyzed set of query shapes can be executed.

Another form of rate limiting is used at the field level. In this context, it is also used for complexity scoring. The intent is similar to the Consumer-Driven Contracts pattern, where consumers need to validate "just enough" and only the data necessary to implement their business functions. Rate limit policies should be scoped to the bounded set of fields that a

consumer will use rather than trafficking all traffic. Fields backed by expensive resolvers and backend services with capacity constraints can have rate limits independently from the overall query rate limit. Rate limit policies should be documented in the schema registry and communicated to consumers in advance.

Consumer-driven contract testing ensures the correct implementation of schema changes and the expected behavior of resolvers. Each consumer has a set of hand-picked queries that cover the most important information they need from a service (similar to the Schematron assertions from the Consumer-Driven Contract pattern) and assertions about the responses [8]. Just as a client of ProductSearch would only care about certain fields such as CatalogueID, Name, and Price, a consumer contract in GraphQL would specify certain fields it requires the schema to provide. Once an agreement on the consumer contract is made, a schema change that breaks the contract cannot be made unless the consumer team accepts it or a schema analysis confirms that the change is non-breaking [8].

In addition to the functional assertions, the performance budgets in the consumer contract tests also provide a signal of quality for resolver efficiency, specifically the maximal fan-out of backend calls required to resolve a query. This helps isolate fan-out regressions in the schemas or resolvers. The performance assertions combined with the end-to-end latency tests based on the service level objective-based thresholds provide a multi-dimensional quality gate to prevent functional and non-functional regressions from entering production traffic.

Governance Control	Enforcement Point	Mechanism	Protection Scope
Complexity scoring	Pre-execution static analysis	Weighted field traversal of selection tree	Prevents expensive resolver fan-out
Structural depth limit	Pre-execution static analysis	Maximum nesting level cap	Prevents stack exhaustion and exponential result sets
Persisted queries	Build-time registration, runtime hash reference	Pre-approved query shape list	Restricts execution to reviewed query shapes only
Field-level rate limiting	Runtime per-consumer enforcement	Independent rate cap per expensive field	Protects capacity-constrained backend services
Schema registry documentation	Pre-enforcement publication	Policy communication to consumers	Enables consumer compliance before enforcement

Table 4: Gateway-Level Query Cost Governance Controls and Their Operational Protection Scope [8]

6. Phased Rollout Strategy, Observability Gates, and Decommissioning of Legacy Entry Points

In this paper, a phased rollout approach is a strategy for moving to a new system that aims to keep the impact on current operations low during each step, reducing the risk of switching from SOAP-heavy consumers to a GraphQL-based system, while also collecting proof of how well the new system works and how stable it is. The framework has five phases. Transition criteria between phases are based on types of architecture properties from network-based systems work [9].

The first stage maintains the contract inventory and baseline metrics suite: a set of metric values, such as payload sizes, response time distributions, errors, and downstream call volumes. Before traffic is switched in production, the GraphQL path must satisfy each of these values. Architectural research has shown that latency and completion time are primary factors in user-perceived performance. Latency may be divided into event detection, interaction configuration, transmission, component processing, and result rendering, each with potential architectural optimizations [10].

The second stage deploys the initial schema and core resolver implementations against the staging infrastructure, with parity tests ensuring that SOAP-resolved and GraphQL-resolved responses are identical when run against the same underlying data[11].

In a third option, some of the incoming SOAP requests to the service, usually between a few percent and twenty percent, are sent at the same time to the GraphQL gateway as a way to check for accuracy. The GraphQL gateway's response is then compared to the response from the backend using field-level diffing logic. This is similar to the client-stateless-

server pattern, but more visible to monitoring tools since the tool does not need to look beyond one request datum to identify the nature of the request [10]. These errors are logged with request context and routed to resolver teams without exposing consumers to inconsistencies in the GraphQL response[12].

In the fourth step, the consumers are switched over via feature flagging from internal tooling users to production users once there is enough confidence. Rollbacks are triggered by thresholds against service level objectives. If P95 latency levels or error rates deviate from margins for a prolonged period, the consumers are routed to the old SOAP interface. This implements the reliability principle of requiring architectures to narrow the scope of failures to recoverable actions [10].

These are surfaced in the observability infrastructure by means of end-to-end distributed traces, resolver instrumentation that records e.g., fan-out counts and cache hit rates, and schema usage (field-level query frequency per consumer [13].

And finally, we can now safely deprecate our legacy SOAP entry points once we've ensured that all critical consumers have migrated, that telemetry shows little remaining usage, and that we have demonstrated stability on the GraphQL path through several cycles of peak load [14]. Playbooks and runtime knobs (for example, decreasing the cost limits we'd normally apply to queries, or disabling expensive fields on responses) can still be used without another deployment [15].

Conclusion

This technique describes how to incrementally migrate high-volume service interfaces from Simple Object Access Protocol into GraphQL, ensuring behavioral parity, safety, and governance at every step along the way. Keeping a refactoring inventory indexed by service contract, identifying domain boundaries based on bounded contexts, and enforcing additive schema evolution with formal query semantics help platforms deliver GraphQL capabilities without breaking existing consumers. In distributed architectures, resolver batching, layered caches, and time budgets help avoid excessive conversations across service boundaries during a single request. Consumer contract testing, query complexity barriers, and field-level rate limiting protect shared infrastructure from degradation as it becomes increasingly used and widely installed. Using observability to control gradual rollouts, along with running two versions at once and having automatic rollback rules, reduces the chances of problems in production while creating a safety record. Once used consistently, this enables the removal of any legacy SOAP entry points, and the GraphQL interface itself becomes a baseline which can be sustainably built on.

References

[1] Martin Gudgin et al., “SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),” W3 Recommendations, 2007. [Online]. Available: https://www.w3.org/TR/soap12-part1/
[2] Amy L. Simmons et al., "The Practice of Rest," 2025. [Online]. Available: https://repositories.lib.utexas.edu/server/api/core/bitstreams/2a6328e0-82a0-4730-8132-fc1b19b0f024/content

[3] Olaf Hartig et al., "Semantics and Complexity of GraphQL," ACM Digital Library, 2026. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3178876.3186014

[4] Eugene Ferry et al., “Security evaluation of the OAuth 2.0 framework,” Information and Computer Security, Volume 23, Issue 1, 2015. [Online]. Available: https://www.emerald.com/ics/article-abstract/23/1/73/111054/Security-evaluation-of-the-OAuth-2-0-framework?redirectedFrom=PDF

[5] Jill Seaman, “Refactoring: Improving the Design of Existing Code,” 2015. [Online]. Available: https://userweb.cs.txstate.edu/~js236/201508/cs4354/refactoring.pdf

[6] Eric Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software,” 2003. [Online]. Available: https://fabiofumarola.github.io/nosql/readingMaterial/Evans03.pdf

[7] Sam Newman, “Building Microservices: Designing Fine-Grained Systems,” O'Reilly Media, 2015. [Online]. Available: https://books.google.co.in/books?id=jjl4BgAAQBAJ&printsec=frontcover&redir_esc=y#v=onepage&q&f=false

[8] Martinfowler.com, "Consumer-Driven Contracts: A Service Evolution Pattern," 2006. [Online]. Available: https://martinfowler.com/articles/consumerDrivenContracts.html

[9] Roy T. Fielding, "Software Architectural Styles for Network-based Applications," 1999. [Online]. Available: https://roy.gbiv.com/pubs/arch_survey.pdf

[10] Martin Kleppmann, "Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems," O'Reilly Media, 2017. [Online]. Available: https://unidel.edu.ng/focelibrary/books/Designing%20Data-Intensive%20Applications%20The%20Big%20Ideas%20Behind%20Reliable,%20Scalable,%20and%20Maintainable%20Systems%20by%20Martin%20Kleppmann%20(z-lib.org).pdf

[11] Quintero, F. A., "Optimized effects design in high-end simulation workflows: Impacts on production time and visual fidelity". *Sarcouncil Journal of Applied Sciences, 1*(1), 21–28,2021

[12] Belhassen, A., "Design, additive manufacturing, and mechanical characterization of continuous fiber-reinforced thermoplastic composites for lightweight robotic arm components" *Journal of Computational Analysis and Applications, 28*(6), 82–97,2020.

[13] Surana, S., "The evolving role of the financial controller in the Indian manufacturing sector: From accounting steward to strategic business partner" *Journal of International Crisis and Risk Communication Research, 4*(2), 439–449,2021.

[14] Darteh, F. K., "Integrating payment systems with revenue and expenditure reporting for improved financial governance," *Journal of International Crisis and Risk Communication Research, 5*(S12), 82–91,2022.

[15] Ascanio, G. A., "Wellness-driven design development in luxury residential architecture: Spatial, social, and environmental dimensions," *Journal of Information Systems Engineering and Management, 6*(1),1-10,2021