British Journal of Multidisciplinary Studies
ISSN: 2978-8110
DOI: 10.32996/bjms
Journal Homepage: www.al-kindipublisher.com/index.php/fcsai

BJMS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

GraphQL in Wealth Management Platforms: Optimizing Data Access and Performance

Ashmitha Nagraj
Senior Full Stack Engineer
Corresponding Author: Ashmitha Nagraj, **E-mail**: nagrajashmitha@gmail.com

| ABSTRACT

In regulated financial systems, such as wealth management platforms, it is necessary to have both a secure, compliant environment, as well as to provide users with real time portfolio analytics, and customizable dashboards. However, many traditional RESTful based API's require multiple round trips and pull down all the information requested from the server even if only some of it is needed; thus, resulting in significant delays when the system is at high volume usage levels. GraphQL allows for a single endpoint query model that is declarative in nature, where clients can ask for only what is needed. Previous studies have demonstrated how this model can significantly decrease the amount of data being transferred across the network and the number of network requests made when retrieving nested data [1]. The study reviews existing literature on empirical and theoretical studies that evaluate whether GraphQL is suitable for use in regulated financial environments. Included within this study's evaluation are benchmarks, cost analysis research, and best practices for performance optimization, and security. Additionally, this study discusses how Federated GraphQL architecture can be used to create modular microservice architecture, as well as hybrid REST/GraphQL architecture. This study suggests that, for complex aggregation queries, GraphQL can reduce latency and backend requests by approximately one-half or more, improve developer productivity when governed properly, but must be weighed against the increased CPU usage seen in GraphQL resolvers when not optimized properly, and the need to implement proper security and compliance measures [2].

| KEYWORDS

GraphQL, REST, wealth management, API performance, microservices, query complexity, federation, security, compliance

| ARTICLE INFORMATION

ACCEPTED: 01 September 2024 **PUBLISHED:** 20 September 2024 **DOI:** 10.32996/bjmss.2024.3.1.3

I. Introduction

Wealth Management Platforms bring together all the components of a Brokerage Account, Retirement Portfolio, Market Data Feed, and Analytics, in a Client Dashboard. Clients whether individual investors, registered investment advisors, or Compliance Officers expect rapid sub second responses when accessing their account information; fine grained control of the data displayed; and audit trails documenting all transactions and changes made to their account(s). Historically, RESTful APIs have dominated the Financial Services industry due to their proven track record and built-in support for HTTP level caching. However, many Dashboards are designed to provide users with aggregated views of their portfolio data. For example, "Client Holdings" that includes detailed security information, such as stock symbol, share quantity, etc., as well as Environmental Social and Governance (ESG) ratings. Each component of the portfolio view would likely be served via a separate endpoint call using a standard REST design pattern. Calling multiple endpoints creates additional latency: each endpoint introduces both network delay and processing delay; and, frequently, returns fixed structure data.

In contrast, GraphQL, an open-source technology developed by Facebook in 2015, allows for a single endpoint call that allows the client to request only the exact fields that are needed. Since the client only requests the data that is needed, there is significantly less data being returned over the wire. One benchmark found that total data transfer in GraphQL was only a few hundred megabytes, whereas the same type of REST calls resulted in approximately 4 GB of data being transferred.[3] Additionally, since the client specifies exactly what data is requested, GraphQL reduces the number of round trips needed to get the desired data. A complex portfolio query that requires 5–10 different REST calls can be accomplished with a single GraphQL query. When combined with client-side rendering, this results in significant latency savings of 50–80%, especially in cases where the data has many layers of nesting.[4]

While the potential benefits of using GraphQL in regulated finance appear to be substantial, several concerns exist related to its adoption. The primary concern is the ability of GraphQL to allow clients to create very costly queries if the server does not enforce some constraints. For example, A study on GraphQL query cost analysis, demonstrated how a poorly constructed GraphQL query could result in exponential growth in the amount of time required to execute the query based solely upon the query size [5]. If not properly constrained, GraphQL servers can also be susceptible to Denial-of-Service (DoS) attacks, based upon the depth or breadth of queries created by malicious clients. Furthermore, if GraphQL Introspection is enabled for public clients, it could potentially expose sensitive data model elements of the service.

Therefore, effective Governance Controls, such as limiting the depth of a query, charging clients for specific fields of data, or enforcing persisted queries are essential to prevent abuse of the service.

II. REST vs. GraphQL: API Characteristics in Finance

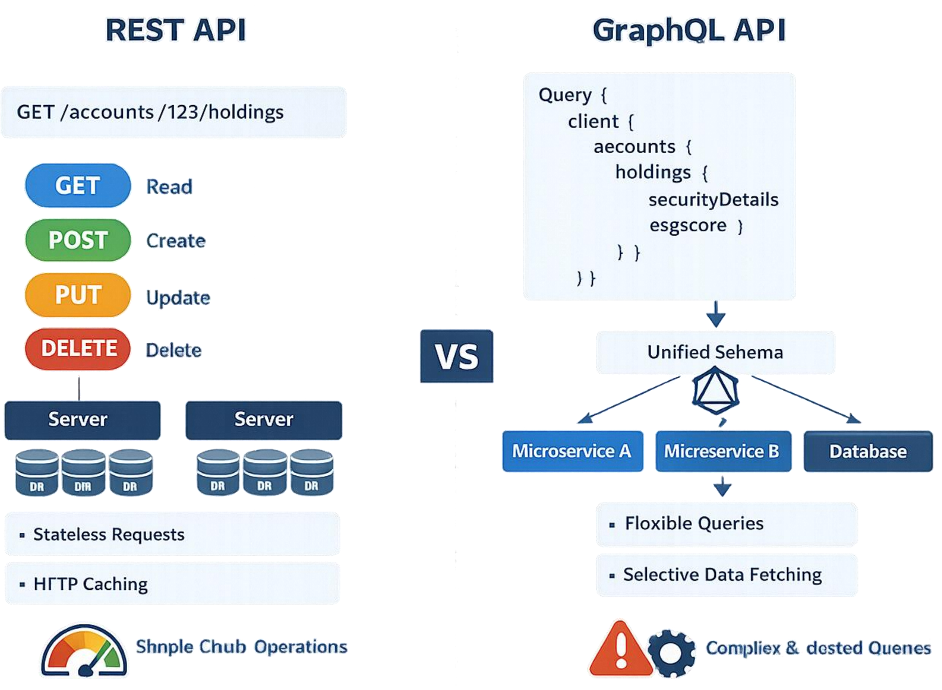

Figure 1: REST API vs GraphQL API comparison

The fundamentals of REST have served as the core of web-based API technology for nearly two decades. In REST, a resource (i.e., account, order, price, etc.) is represented via a unique URL, and the client interacts with the resource via HTTP Verbs (i.e., GET, POST, etc.). The simplicity of REST lends itself well to CRUD-type (create, read, update, delete) operations (i.e., GET /accounts/123/holdings). REST also provides a basis for implementing standardized HTTP caching on each individual endpoint. Additionally, in the realm of financial services, the stateless nature of REST and its uniform interface enable greater ease of compliance (i.e., logging every request) and fault isolation. Studies indicate that while REST performs better than GraphQL in high-load, simple scenarios; when considering high-complexity queries involving large amounts of data, the results vary greatly depending upon how well both technologies are implemented. For example, once a study showed that for single-table queries under heavy load, the average response time and throughput for REST was significantly higher than that of GraphQL[6].

GraphQL Semantics.

Unlike REST, which exposes a resource at a specific URL and relies on the HTTP Verb (GET, POST, PUT, DELETE, etc.) used to determine how to interact with the resource, GraphQL uses a global schema - a set of types representing a Client, Account, Holding, etc. and a single /GraphQL endpoint. The client sends a query to the /GraphQL endpoint and specifies the exact fields of the objects that it wants to retrieve. An example of such a query would be one requesting all Accounts of a given Client, along with the Holdings of those Accounts, Security Details of the Holdings, and the ESG Scores associated with the Securities, in a single request. Behind-the-scenes, the GraphQL Server breaks down the query into pieces and directs those pieces to the appropriate Micro-Service(s) or Database(s) where the data resides. By design, a GraphQL Response will include only the information requested by the client in their Query, thus eliminating the "over-fetch" problem inherent in REST [1].

However, GraphQL does introduce several new issues. The query language used by GraphQL enables complex and nested queries. Once research showed that the evaluation process of a GraphQL Query can exhibit exponential time complexity based on the number of layers in a nested List [4]. As such, if a Malicious or Buggy Query exists with numerous levels of List Relations, for example: portfolios → holdings → securities → sub-components, etc., the Server may be forced into excessive computational processing. As per study on formalized Query Cost Analysis: currently, the vast majority of GraphQL APIs do not implement Static or Dynamic Enforcement on Query Complexity, thus allowing Servers to be abused. According to study on Survey of 30 Public GraphQL APIs, 83% of the Surveys had No Static Complexity Checks in place, and 73% of the Surveys had No Rate Limiting in place [5]. These findings highlight the necessity for strict policies in Finance-Grade Deployments of GraphQL.

Limitations of REST in Complex Queries:

In a Wealth Dashboard application, a client's requirements for Aggregated Data are very common. Examples of such data include account balances, holdings by Asset Class, Real-Time Prices, Risk Metrics, etc. However, due to the Nature of REST, a client must typically perform Multiple Calls to obtain the desired Aggregated Data.

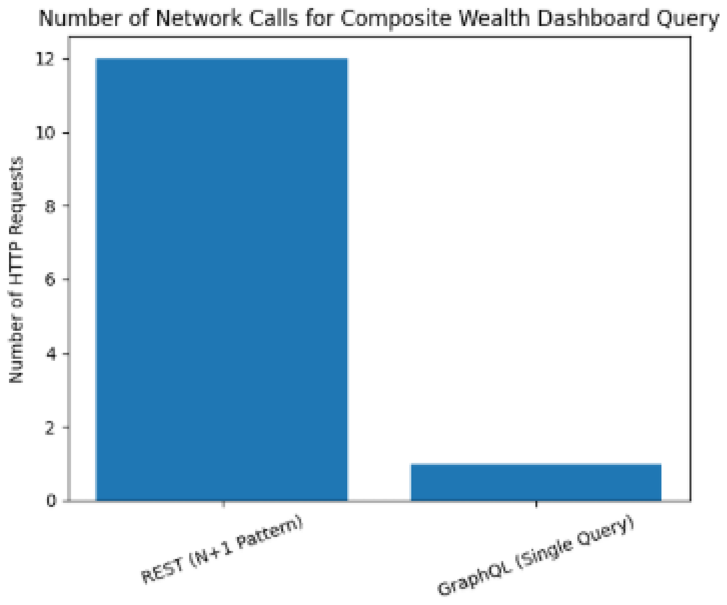

Figure 2: Total Data transferred in Composite Wealth Dashboard Query

For example, a client wishing to obtain an Account's Securities, followed by obtaining each Security's ESG Rating and Price from other Endpoints, can quickly create an N+1 Pattern. Each additional HTTP Call to the Server creates additional Latency and Overhead. Several Benchmarks have shown that for Composite Views of Data, REST typically generates 50% more network requests than the Single-Call Model of GraphQL [7]. As such, this has Practical Effects: during periods of High Market Volatility, the dozens of REST Calls made per user can result in significant Latency Spikes and Server Load Surges.

GraphQL solves this problem by moving the design of the query to the Client Side. A Single GraphQL Query can join multiple sources of data in one-go. As such, GraphQL can reduce roundtrips and payloads. GraphQL responses can be Orders of

Magnitude smaller than those generated by REST in complex scenarios, since unused fields are omitted. From a bandwidth perspective, one Benchmark Study showed that the REST API tested generated ~4GB total during the test, while the GraphQL API only transferred a few hundred MB for Equivalent Tasks [8]. In summary, GraphQL's Fine-Grained Data selection improves efficiency in Data-Intensive queries, however, at the expense of greater Server-Side computation per request.[8]

III. Empirical Performance Evaluations

The following studies have compared REST and GraphQL through experiments. Experimental comparison of developers' efforts during REST and GraphQL focused on developer-based tasks but noted performance differences. GraphQL developer effort was lower (implementation time median 6 minutes vs 9 minutes for REST) as well as easier to implement based on query composition [3]; however, this study was primarily concerned with system performance, specifically latency and throughput.

Extremely high-load testing environment on a complex management information system (MIS), measured response time, throughput, CPU and memory, in terms of latency (up to ~50%) and throughput, the REST method was significantly faster than GraphQL, which reflected the additional processing required by GraphQL per request [9]. Although GraphQL utilized approximately 37% less CPU and approximately 40% less memory than REST for the same workload [9], GraphQL would be best suited for those applications requiring query flexibility and/or optimal resource utilization.

Another study compared REST application demonstrated better performance for basic queries involving a single table (faster response time), although the GraphQL application was superior to the REST application when querying across four related tables. Moreover, the GraphQL application produced dramatically smaller payloads (up to 94% smaller payload than REST) in cases where multiple tables were queried [10]. GraphQL has a significant advantage in retrieving large amounts of complexly structured data, whereas REST is generally "better for simple and high-load scenarios".

Systematic reviews have also been conducted regarding the performance characteristics of REST and GraphQL, in the case of simple queries, GraphQL was as slow as 2.5x slower than REST; nevertheless, note that GraphQL reduces over-fetching: in cases where partial-field queries were made to retrieve only certain fields from a larger object or objects, GraphQL returned far fewer bytes than REST [11]. Overall, a balanced viewpoint: GraphQL provides clients with the ability to reduce the number of requests which is needed to make to the server, improve the overall user experience, and provide more functionality with respect to data retrieval, but can introduce server-side CPU overhead unless the requests are properly batched.

In addition, since throughput peaks and concurrency matter in real-time trading environments, some models suggest that GraphQL's parallel fetch potential can increase the amount of throughput generated by the system during periods of volatility. For example, if each portfolio update utilizes one GraphQL query instead of five or more REST calls, the peak system throughput can be increased by a factor of several.

IV. Developer Productivity and Payload Efficiency

According to various studies, GraphQL has a potential to accelerate the process of developing software applications. The larger difference in the time of completion occurred when dealing with more complicated tasks that included several parameters to be processed in REST. Even the most experienced developers were able to create queries in a significantly shorter amount of time when using GraphQL, which led to a 33% increase in efficiency among developers when completing such tasks [12]. The design of a well-organized GraphQL schema will allow developers to write less boilerplate code and make it easier for the frontend team to obtain data through fewer lines of code.

Another benefit of GraphQL from a developer standpoint is the ability to minimize payload sizes. REST technology provides a fixed structure of responses defined by each endpoint, and clients must remove any un-needed data from the response. GraphQL does away with this unnecessary overhead, allowing clients to only include in the query what is needed. When clients requested partial fields in a GraphQL query, the resulting response was 51% slower than of a comparable response received via REST technology [13]. As a result, when applied to real-world systems, this enables front-end applications to receive only the data that is necessary, and therefore, mobile applications or financial dashboard applications are both faster and more responsive. Additionally, GraphQL's built-in type system and introspection capabilities allow to automatically generate and validate queries at build-time, thereby reducing errors.

On the downside of GraphQL is the fact that each request made to a GraphQL server typically results in additional processing on the server-side. Specifically, each field queried within a given query will invoke a resolver function. While this added complexity is easily mitigated by implementing batching to combine multiple resolver invocations into a single database call, and caching to eliminate duplicate requests, the increased CPU utilization can become problematic if not properly addressed [14]. GraphQL servers can exhibit very high CPU usage due to the lack of optimization in the resolver functions, including a benchmark that showed an average of 915 ms for GraphQL, whereas REST had an average of 247 ms [15].

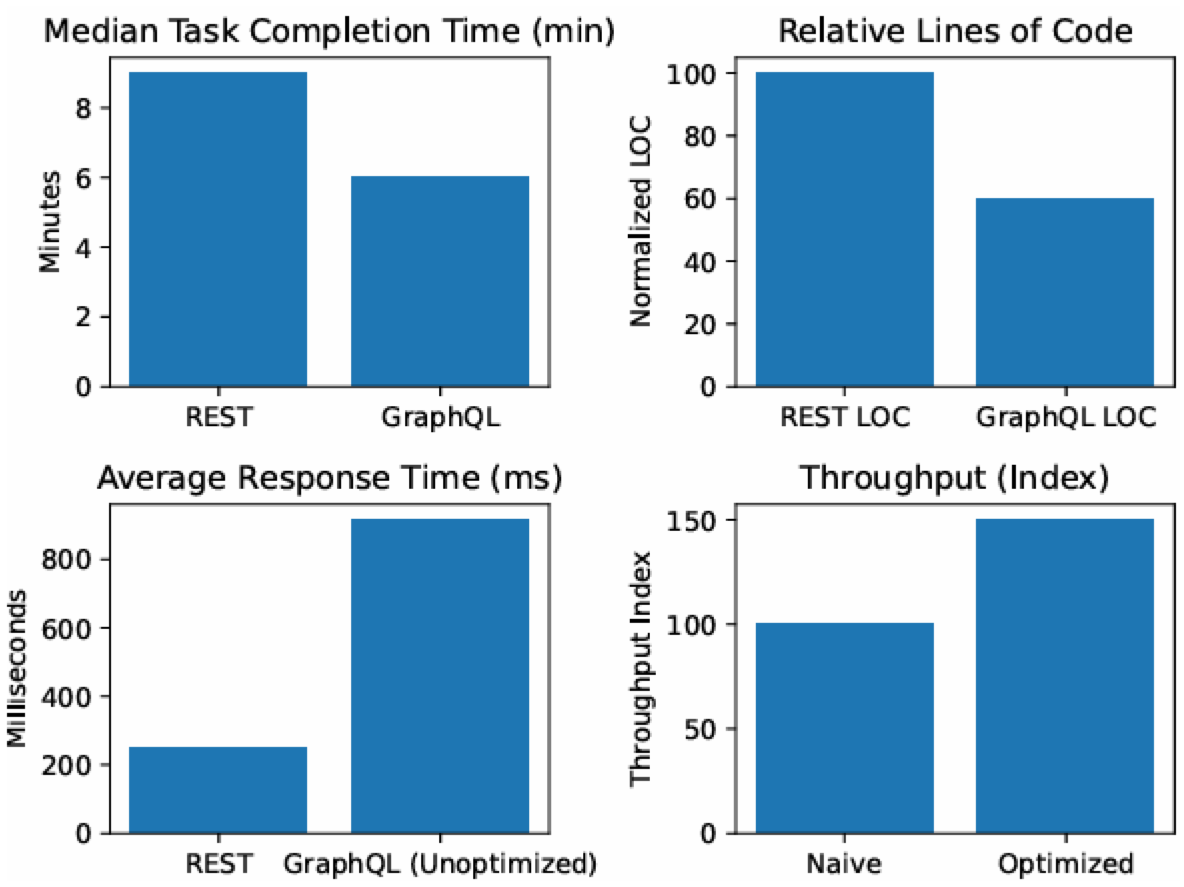

Figure 3: Developer productivity and performance trade-offs between REST and GraphQL

However, once the above-mentioned optimizations are implemented, GraphQL can provide performance advantages over naive resolver implementations, with reported throughput increases ranging from 30% to 70% depending upon the specific use case and implementation [16]. To summarize, while there are obvious benefits to developer productivity provided by GraphQL, teams must invest resources in testing and optimizing the resolver implementations to prevent CPU bottlenecks and realize optimal performance.

V. Federated Architecture for Financial Microservices

Microservices-based large wealth platforms are made up of multiple microservices, and a federation platform allows different data models to be composed into a single model. Each microservice has its own sub-graph. The gateway then combines the sub-graphs into one graph allowing for all the benefits of a modular microservice architecture while providing a single endpoint for clients to call. Graphql provides a federation model that natively supports this “federated microservices” pattern [17].

For Wealth Management, federation is a natural mapping of the various domains; i.e. the portfolio domain is managed by one team and has its own set of types and resolvers that expose its data, the market-data domain is managed by another team and has its own set of types and resolvers that expose its data, analytics is managed by yet another team and has its own set of types and resolvers that expose its data, and so on. The gateway simply stitches together the data from each of the sub-domains to provide a unified view of the data. An order service may define an Order type with @key (fields = "id") and the portfolio service would use the id to retrieve an order using a compose operation. The graphql documentation describes how to combine independent services into a unified schema, like a federal system where many small, independent states work together under a unified government [18].

Federation increases the scalability of the application by decoupling the development cycle of each service from the other services. It also increases the autonomy of the development teams since development of the services can happen independently without affecting other services. The gateway manages the query planning, i.e. if a complex query comes into the gateway, it will break down the query into smaller queries and send those to the respective services in parallel and then merge the results back into a single result. The ability of the gateway to handle complex queries in parallel is very important for performance under heavy loads. An enterprise environment with a federated GraphQL architecture can see approximately 40% faster end-to-end response times compared to a monolithic GraphQL server [19]. The speed improvement is due to the concurrent processing of services by the gateway, especially during high volume periods such as peak trading hours. This translates into real-world terms as the time taken for each service to respond to a request will add some delay but performing them concurrently will yield nearly the maximum possible delay rather than adding delays linearly.

Although federation has many benefits, it does increase the complexity of the overall system. Services need to have agreement on common types and the identity of objects being queried. When used in a regulated domain, the gateway also becomes a point of control for security-related issues. Despite these increased complexities, federation does align with the compliance requirements of many wealth-management organizations. Each microservice can enforce specific policies relevant to its domain. Since the GraphQL layer simply passes through user identity and performs per-field authorization at the appropriate resolver, federation appears to be a viable architecture for wealth firms that need to both rapidly develop new business capabilities while meeting the independent compliance requirements of the organization.

VI. Performance Optimization Strategies

To achieve the performance and scalability of GraphQL in production, there are specific engineering practices that platform teams should implement:

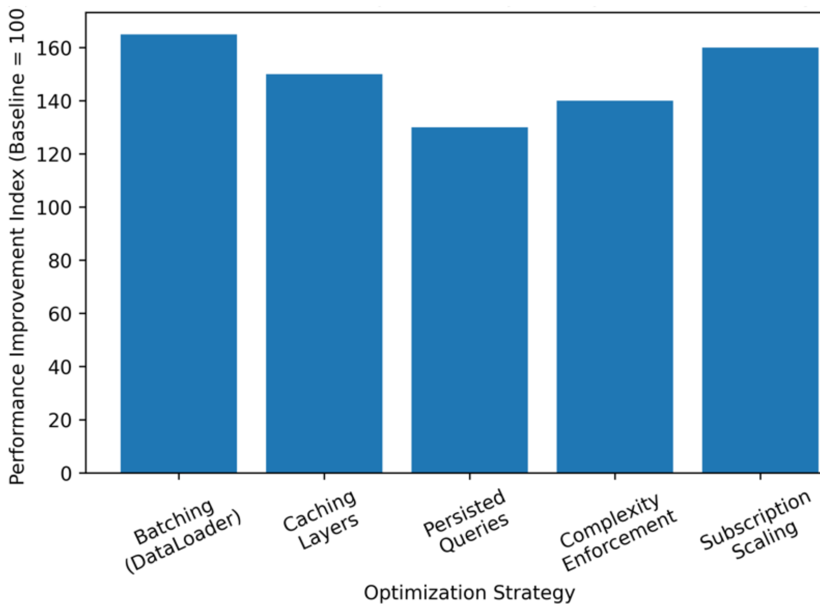

Figure 4: Relative Performance Impact of GraphQL Optimization Strategies

1. Query Batching (DataLoader):

Use a batching layer to prevent the N + 1 query problem. A batching layer will collect all the requested IDs during one GraphQL request cycle and make one batched call to the database [20]. For example, if a query retrieves 100 holdings, each holding referencing a Security, the DataLoader will retrieve all 100 Security IDs in one join versus 100 separate queries. Teams should measure resolver call counts before and after implementing DataLoader to confirm improvements.

2. Caching Layers:

Wealth platforms require different caching durations for the respective data types. GraphQL queries can be persisted/whitelisted and cached via an HTTP CDN or API gateway, essentially converting them to pseudo-REST endpoints behind the scenes. Utilizing an in-memory cache such as Redis to store recent query results, can significantly reduce the number of hits against the databases. Within a microservices environment, each subgraph can have its own cache, while the gateway caches the complete query results for composite queries.

3. Persisted Queries:

Require that all external clients utilize persisted queries identified by a hash. Not only does this minimize the payload size but also guarantees that only vetted queries will run. Allowing arbitrary query strings greatly increases the attack vector [21]. In a wealth context, a catalog of accepted queries for each client application release can be published. The server will reject any non-vetted query. This ensures that hidden fields or new query patterns that are introduced with an updated client application are reviewed. When utilized with persisted queries, API gateways can provide CDN caching at the HTTP level since the URL can now be considered cacheable.

4. Query Complexity Enforcement:

Implement static complexity rules. Each field or list type can have a “cost”. Prior to execution, the server calculates the total cost of a query. If it is above a defined limit, the server will reject it. Cost analysis effectively captures heavy queries with minimal false positives [5]. For example, a query may recursively traverse five levels of nested lists and would therefore be rejected prior to execution. These constraints protect the stability of systems during peak usage periods. Both static cost limits and rate limiting are used in industry practice: a user can only consume N “cost units” per minute. Firms must implement these mitigation strategies to ensure to not inadvertently allow a client to request all historical trade data in one request.

5. Subscription Scaling:

Many wealth applications rely upon real-time updates. GraphQL subscriptions over WebSockets can provide real-time updates. To scale, the subscriptions engine typically utilizes a pub/sub system where back-end services publish events, and the GraphQL gateway publishes the event to the subscribing clients. During a heavy load scenario, additional brokers and shards are required. In practice, companies isolate subscriptions from queries so bursty query traffic does not starve real-time feeds and vice versa [22].

VII. Implementation Trade-offs and Hybrid Models

GraphQL provides superior support for data aggregation while REST still maintains its own niche's. At critical points of transactional flow, REST is generally easier to cache and secure due to the simple nature of the requests. A Trade Submission API would require very little beyond a couple defined parameters; in such cases REST is sufficient and can utilize HTTP status codes to provide feedback on request completion. Richer data views can be provided via GraphQL; therefore, a common implementation pattern is to use both technologies within the same organization to provide a hybrid model, where REST based endpoints provide command/action functionality and GraphQL is used for Read-Heavy query and dashboard applications. The mapping study indicates hybrid architecture will be a major area of future research [11].

Another significant point of differentiation is caching behavior. REST takes advantage of standardized HTTP caching headers per endpoint, whereas, in GraphQL, caching typically occurs through custom caching mechanisms that cache the results of entire queries or utilize client-side caches. Organizations will have to determine which areas of the technology stack are willing to accept additional complexity. The mapping study states that caching GraphQL at the HTTP layer requires persisted queries but when cached, it does present unique invalidation issues [5].

Similarly, governance complexity is higher with GraphQL. A single GraphQL schema is a shared contract amongst all clients, if the schema is changed, many of the clients could potentially be broken. On the other hand, each REST endpoint can be versioned individually. To govern this type of scenario, organizations will need to implement formalized schema evolution practices and may choose to implement an API Gateway that can service different versions of the same GraphQL schema concurrently.

Key Trade-off Summary:

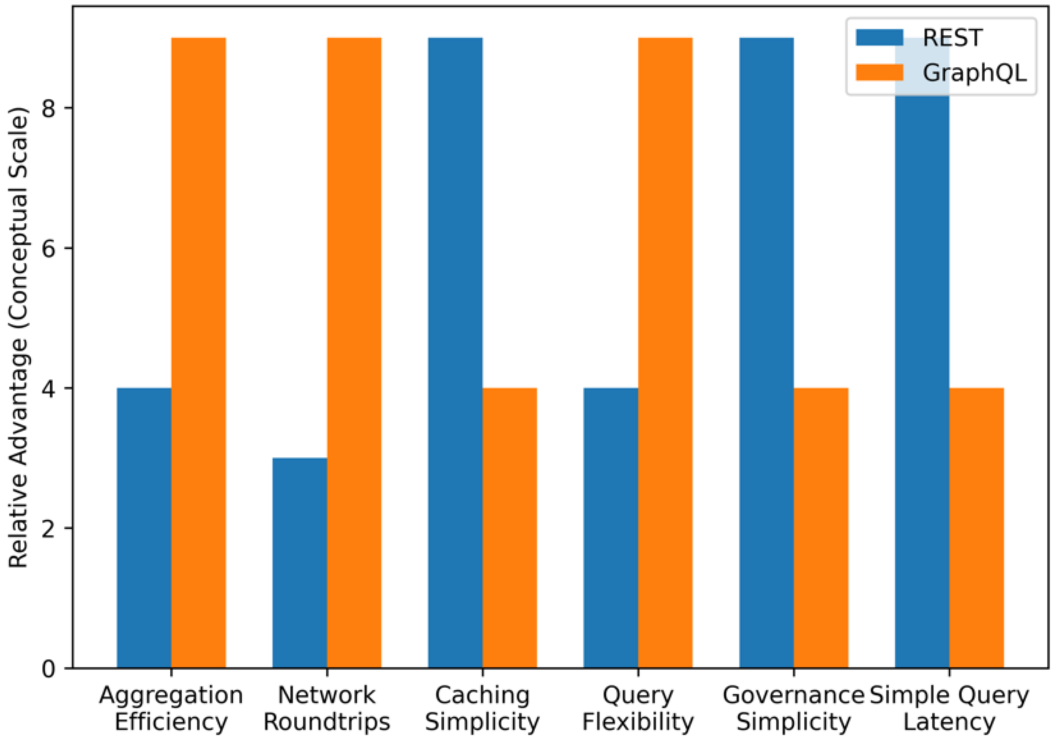

Figure 5: REST vs GraphQL Trade-off Profile in Hybrid Wealth Platforms

- **Aggregation Efficiency:** REST often requires multiple calls; GraphQL can fetch nested data in one go
- **Network Roundtrips:** REST: many; GraphQL: one.
- **Data Transfer:** GraphQL returns only requested fields.
- **Caching Simplicity:** REST: native HTTP cache; GraphQL: needs custom persisted queries/Cache-Control.
- **Query Flexibility:** GraphQL is client-driven; REST is static.
- **Governance Complexity:** REST APIs are independent; GraphQL has one evolving schema requiring rigorous review.
- **Developer Productivity:** GraphQL speeds up frontend development [4].
- **Latency & Throughput:** In simple queries, REST is typically faster [3]. In multi-domain aggregation, GraphQL can reduce end-to-end latency by eliminating sequential calls [6].

VIII. Limitations and Future Work

Most existing studies of the performance of fintech applications have either been artificial or at a relatively small scale. Therefore, there is a need for further evaluation of how well fintech applications perform in "real world" environments with larger numbers of users. For example, there has been very little published about how well GraphQL subscriptions scale when used in flash markets where thousands of clients are being fed price updates rapidly; this is another area of study that needs to be addressed. Additionally, there is a significant gap in the use of formal methods to support schema evolution during the execution of trading systems; the application of automated tools to evaluate compliance of GraphQL APIs with regulatory requirements is another area of study that is just beginning to emerge. Future possibilities for GraphQL include machine learning-based query optimizers and/or auto-scaling federation components.

IX. Conclusion

Data access efficiency in Wealth Management applications can significantly improve by using GraphQL's single endpoint and client driven query model as well as rich, nested portfolio view. GraphQL typically reduces the number of network calls and the size of payloads relative to multi-step REST aggregation [3][2], thus reducing perceived latency for clients and load on the backend infrastructure. In addition, developers experience increased productivity when building queries because the developer can create the exact GraphQL query required without having to write much glue code [4].

However, like all solutions there are trade-offs. Using GraphQL for a data access application will require additional work from the server side as well as more work to implement governance for GraphQL including query complexity limits, batching, and caching to prevent performance issues and security risks. However, if an organization implements governance correctly, a high level of regulatory compliance while providing a superior aggregation layer can be provided.

Typically, the best approach is to use both technologies together; use GraphQL for read heavy, complex queries, and use REST for simple commands and transactions. This allows organizations to take advantage of the strengths of GraphQL, while at the same time taking advantage of the caching capabilities of REST for everyday operations [22].

References:

[1] Quiña-Mera, Antonio & Fernandez, Pablo & García, José María & Ruiz-Cortés, Antonio. (2022). GraphQL: A Systematic Mapping Study. ACM Computing Surveys. 55. 10.1145/3561818.
[2] D. A. Hartina, A. Lawi and B. L. E. Panggabean, "Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University," 2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT), Makassar, Indonesia, 2018, pp. 237-240, doi: 10.1109/EIConCIT.2018.8878524.
[3] Brito, Gleison & Valente, Marco. (2020). REST vs GraphQL: A Controlled Experiment. 10.1109/ICSA47634.2020.00016.
[4] Goodnews, Ogboada & Anireh, Vincent & Matthias, Daniel. (2022). A Model for Optimizing the Runtime of GraphQL Queries. 9. 11-39.
[5] Mavroudeas, Georgios & Baudart, Guillaume & Cha, Alan & Hirzel, Martin & Laredo, Jim & Magdon-Ismail, Malik & Mandel, Louis & Wittern, Erik. (2021). Learning GraphQL Query Cost. 1146-1150. 10.1109/ASE51524.2021.9678513.
[6] Vadlamani, Sri Lakshmi & Emdon, Benjamin & Arts, Joshua & Baysal, Olga. (2021). Can GraphQL Replace REST? A Study of Their Efficiency and Viability. 10-17. 10.1109/SER-IP52554.2021.00009.
[7] Karlsson, Stefan & Causevic, Adnan & Sundmark, Daniel. (2020). Automatic Property-based Testing of GraphQL APIs. 10.48550/arXiv.2012.07380.
[8] Hartig, Olaf & Pérez, Jorge. (2018). Semantics and Complexity of GraphQL. WWW '18: Proceedings of the 2018 World Wide Web Conference. 1155-1164. 10.1145/3178876.3186014.
[9] Niklasson, A., & Werèlius, V. (2023). RESTful API vs. GraphQL a CRUD performance comparison (Dissertation). Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-122281
[10] Kanthed, Surbhi. (2023). Rest vs. GraphQL: Comparative Analysis of API Design Approaches. International Journal of Multidisciplinary Research and Growth Evaluation.. 4. 984-991. 10.54660/.IJMRGE.2023.4.1.984-991.
[11] Vogel, Maximilian & Weber, Sebastian & Zirpins, Christian. (2018). Experiences on Migrating RESTful Web Services to GraphQL. 10.1007/978-3-319-91764-1_23.
[12] Wittern, Erik & Cha, Alan & Davis, James & Baudart, Guillaume & Mandel, Louis. (2019). An Empirical Study of GraphQL Schemas. 10.1007/978-3-030-33702-5_1.
[13] Lawi, Armin & Panggabean, Benny & Yoshida, Takaichi. (2021). Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. Computers. 10. 138. 10.3390/computers10110138.
[14] Quiña-Mera, Antonio & Fernandez, Pablo & García, José María & Ruiz-Cortés, Antonio. (2022). GraphQL: A Systematic Mapping Study. ACM Computing Surveys. 55. 10.1145/3561818.
[15] Margański, Piotr & Pańczyk, Beata. (2021). REST and GraphQL comparative analysis. Journal of Computer Sciences Institute. 19. 89-94. 10.35784/jcsi.2473.
[16] Vohra, N., & Kerthyayana Manuaba, I.B. (2022). Implementation of REST API vs GraphQL in Microservice Architecture. *2022 International Conference on Information Management and Technology (ICIMTech)*, 45-50.
[17] Stünkel, Patrick & Bargen, Ole & Rutle, Adrian & Lamo, Yngve. (2020). GraphQL Federation: A Model-Based Approach.. The Journal of Object Technology. 19. 18:1. 10.5381/jot.2020.19.2.a18.
[18] Nogatz, Falco & Seipel, Dietmar. (2016). Implementing GraphQL as a Query Language for Deductive Databases in SWI-Prolog Using DCGs, Quasi Quotations, and Dicts. Workshop on (Constraint) Logic Programming (WLP 2016). 234. 42-56. 10.4204/EPTCS.234.4.
[19] Adrio, Kendricko & Tanzil, Clementius & Lianto, Michael & Rasjid, Zulfany. (2023). Comparative Analysis of Monolith, Microservice API Gateway and Microservice Federated Gateway on Web-based application using GraphQL API. 654-660. 10.1109/EECSI59885.2023.10295809.
[20] Erigha, Eseoghene & Obuse, Ehimah & Okare, Babawale & Uzoka, Abel & Owoade, Samuel & Ayanbode, Noah. (2021). Optimizing GraphQL Server Performance with Intelligent Request Batching, Query Deduplication, and Caching Mechanisms. International Journal of Multidisciplinary Futuristic Development. 2. 75-86. 10.54660/IJMFD.2021.2.1.75-86.
[21] Angele, Kevin & Meitinger, Manuel & Bußjäger, Marc & Föhl, Stephan & Fensel, Anna. (2022). GraphSPARQL: a GraphQL interface for linked data. 778-785. 10.1145/3477314.3507655.
[22] Ulrich, Hannes & Kern, Jori & Tas, D. & Kock, Ann-Kristin & Ückert, Frank & Ingenerf, Josef & Lablans, Martin. (2019). QL4MDR: a GraphQL query language for ISO 11179-based metadata repositories. BMC Medical Informatics and Decision Making. 19. 10.1186/s12911-019-0794-z.