AI Tech International Journal, ISSN: 3079-4749
Vol. 2, No. 1, January-June, 2024
Journal homepage: https://techaijournal.com

Efficient Data Fetching Strategies Using Federated GraphQL in Distributed Frontend Systems.

Ashok Kumar

Senior Software Engineer, Walmart Global Tech, Independent researcher, USA.

Article Info

Article history:

Received Febuary 05, 2024
Revised Febuary10, 2024
Accepted Febuary 25, 2024
Published March10, 2024

Keywords:

federated GraphQL, distributed frontend systems, micro-frontends, data fetching, microservices architecture

ABSTRACT

Federated GraphQL is now a promising data orchestration model in distributed frontend systems, especially when micro-frontends are data consumers of multiple, independently deployed backend services. This paper will look at the performance of data-fetching when a distributed frontend is using federated GraphQL, as opposed to endpoint-based aggregation. A managed benchmark setup was adopted in the area of three exemplary user journeys: dashboard composition, product-detail rendering and cart-summary retrieval. Three strategies were considered, which included a REST baseline, simple federated GraphQL, and federated GraphQL with request batching and entity-level caching. The end-to-end latency, p95 latency, and data transferred by responses as well as the service-call volume and hits in the cache were measured on 2,700 traced requests. The results indicate that basic federation effectively minimizes the size of the payload but is not associated with decrease in latency of shallow workflows. Adding batching and caching causes federated GraphQL to achieve the best overall result, weighted mean latency of -9.96, p95 latency of -7.88, transferred data of -63.60 and service calls of -62.64 compared to the REST baseline. The data shows federated GraphQL works best in distributed frontends with field-level selectivity and strategies to restrict resolver fan-out between services.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Ashok Kumar

Senior Software Engineer, Walmart Global Tech, Independent researcher, USA.

INTRODUCTION

The distributed frontend systems are now at the heart of large-scale web application engineering as digital products are being built on independently deployed business capabilities instead of being based on a single, centrally owned interface. Service autonomy has already been normalized by using microservice backends, and a similar decomposition of the client layer with micro-frontends has been embraced to enhance team autonomy, release rate, and domain ownership (Peltonen et al., 2021; Taibi and Mezzalira, 2022). However, this architectural change has also brought a long-standing issue of data-access: individual user interactions may need the concerted effort of multiple services to be completed, with frontend modules still needing low-latency and narrow-scoped payloads (Di Francesco et al., 2019; Soldani et al., 2018).

REST-based traditional integration has been able to solve this issue using endpoint composition, backend-for-frontend layers, and API gateways, but these mechanisms tend to build up proliferated endpoints and promote the coarse-grained design of payloads. With the spread of distributed views, the frontend is either over-fed or makes extra round trips to fetch missing fields and the familiar over-fetching and under-fetching tension in GraphQL research (Lawi et al., 2021; Margański and Pańczyk, 2021). Network fan-out, serialization overhead, and presentation requirement/ endpoint design coupling only exacerbate such inefficiencies in microservice environments (Aksakalli et al., 2021; Zuo et al., 2020).

GraphQL had been proposed as a query language and execution model of a schema based on the requirement to only request fields of a specific view. Further studies have demonstrated that GraphQL can particularly be used in

application scenarios with highly interconnected data models and in consumer-related data requirements that demand versatility of navigation between related data entities (Hartig and Perez, 2018; Quiña-Mera et al., 2023). Federated GraphQL has also been developed to scale this model in parallel, allowing different domain services to be part of the same supergraph, forming a single data layer without service ownership fallacy. This architecture seems to be very compatible with distributed frontend systems, in which there are a number of independently developed fragments of interfaces that need a coherent but flexible source of truth (Ulrich et al., 2019; Peltonen et al., 2021).

Although GraphQL has been rapidly taken up in practice, empirical literature on the frontend consumption is still imbalanced. The systematic mapping of Quiña-Mera et al. (2023) finds that the research on the topic of GraphQL has been growing strongly, but the evidence of consumption-side is much less than the discussion of implementation-side. Benefits in payload efficiency and API flexibility have been reported in comparative studies with REST, but performance performance depends on the shape of the workload, implementation strategy, and schema design (Brito and Valente, 2020; Lawi et al., 2021). Federated GraphQL, a certain data-fetching protocol of distributed frontend systems that have to coordinate the activities of several teams, modules, and service boundaries, has received even less attention (Kim et al., 2019; Taibi and Mezzalira, 2022).

The current work fills that gap and explores effective data-fetching techniques with federated GraphQL in a distributed frontend system. The experiment does not simply determine the existence of the reduction in transferred data by federation, but also the enhancement of end-to-end responsiveness in the case of realistic multi-service view composition. Special focus is placed on two main performance parameters latency and transferred response volume as explanatory indicators with service-call volume and cache hits utilized as explanatory indicators. The paper is intended to explain under what circumstances federated GraphQL can be used to achieve quantifiable performance improvements as opposed to architectural beauty (Camilli and Russo, 2022; Waseem et al., 2021).

LITERATURE SURVEY

Rapidly growing research base on GraphQL Since the technology was open-sourced, the literature has developed in a variety of different streams, however. Early research has been on the language model itself such as query semantics, type systems and traversal complexity. Hartig and Pérez (2018) demonstrated the structure of GraphQL is particularly effective to navigate connected data, which is why it is popular when used in distributed applications that require the creation of nested entities based on numerous sources. Subsequent empirical analysis of schema ecosystems has shown that there is a plethora of GraphQL designs in open repositories and that the adoption of GraphQL shifted rapidly beyond platform-specific instances to a wider tooling and developer ecosystem (Kim et al., 2019).

A second area of study has looked at GraphQL as a replacement to REST. There is general consensus in the literature that GraphQL offers greater field-level selectivity and closer alignment between the needs of clients and the responses of the server, but has divided opinions on whether these advantages are directly reflected in the reduction of response times. Lawi et al. (2021) discovered that GraphQL can be more effective than REST when the access conditions are data-intensive due to the decrease in the number of irrelevant payloads. Margański and Pańczyk (2021) also stated that GraphQL enhances accuracy in information retrievals, particularly when customers want only some of the attributes of the resources. But Brito and Valente (2020) noted that the overhead of GraphQL execution, the orchestration of resolvers can negate some of these benefits, which means that performance is not only dependent on the query shape, but also the decisions made by the server-side implementation. Evidence of migration also indicates that GraphQL has real benefits, albeit with design trade-offs concerning schema maintenance, error management, and governance (Brito et al., 2019).

The third related literature is most pertinent to federation since it discusses GraphQL in distributed or highly connected data spaces. Ulrich et al. (2019) have shown how GraphQL can serve as a common query interface to distributed metadata repositories, highlighting its appropriateness to navigate connected structures without compelling clients to use multiple customized endpoints. The conceptual importance of that finding to distributed frontend systems is that frontends are relying more on cross-domain navigation, including retrieving user context, catalog information, pricing, inventory and recommendations, in a single view. The relevance of this systematic mapping by Quiña-Mera et al. (2023) is supported by the fact that the significant practical benefit of GraphQL is its flexibility in consumption, but also states that more empirical data is required regarding actual application cases and patterns of service composition.
The frontend dimension creates another architectural intricacy. The goals of micro-frontends are to divide the client into independently deployable components according to business domains, but the literature continually states that one of the most difficult issues arises when integrating data is required after visual decomposition. Peltonen et al. (2021) list team autonomy, release decoupling, and scalability as the main advantages of micro-frontends, as well as discuss the integration overhead and cross-cutting consistency issues. Taibi and Mezzalira (2022) also observe that distributed frontends tend to lose complexity into build pipelines to runtime orchestration, and the common data contracts and communication patterns become determining. These observations indicate that the worth of federated GraphQL cannot be determined solely as an API option; it should be determined as an integrating mechanism among distributed UI modules, as well as distributed backend domains.

This evaluation is given a wider context of performance in the microservices literature. The systematic mapping of studies and literature reviews are consistent that service decomposition enhances scalability in organizations but may worsen the execution time when communication patterns are not effectively handled (Di Francesco et al., 2019; Soldani et al., 2018). The disciplines (design, monitoring, and testing) that practitioners highlight as being in need are the result of an increase in the count of network calls, the depth of orchestration, and the complexity of observability as the number of services increases (Waseem et al., 2021). Similar findings indicate that the topology of deployment and inter-service calling style used significantly affect the latency propagation and resilience (Aksakalli et al., 2021). The results of performance modeling by Camilli and Russo (2022) also suggest that increases in service interactions may result in non-linear penalties of performance, particularly in situations where requests have fan-out on multiple dependencies.

The operational relevance of governance and migration strategy are also emphasized in recent empirical research of microservice practice. Michael Ayas et al. (2023) demonstrate that migration towards microservices is hardly a technical one only, as it not only redistributes the tasks of teams but also increases coordination costs. Security research also concludes that the advantages of architecture are compromised when interface contracts, observability, cross-service controls, are still in their infancy (Rezaei Nasab et al., 2023). In this context, an API composition layer should not merely tie access together but minimize the waste of communication without creating an unacceptable planning or implementation overhead. The available literature thus confirms the main assumption of the current research, however, it does indicate one particular gap in the form of a lack of empirical evidence as to how federated GraphQL works as a strategy to fetch data, as opposed to being an abstract API paradigm.

RESEARCH METHODOLOGY

The controlled benchmark design was used to measure the efficiency of data fetching with three architectural strategies in a distributed frontend environment. The benchmark setup included a shell frontend, three domain-oriented frontend modules to compose dashboards, present product-detail, and interact with cart-summaries. These interface modules fed on six autonomously deployed domain services, user, catalog, pricing, inventory, reviews and recommendations. All trials had the same domain model and business data, ensuring that any performance differences could be due to the data-fetching strategy as opposed to feature differences. This architecture aligns with previous studies that consider communication structure and patterns of service interaction as key factors influencing performance of microservices (Camilli and Russo, 2022; Waseem et al., 2021).

There were three strategies that were compared. The former one was a REST baseline that saw every frontend module invoke service-specific endpoints via an API gateway. The second approach involved federated GraphQL and a single supergraph compiled out of the six domain services to enable the frontend to make a single view-specific query and the federated layer to resolve the needed entities in subgraphs. Strategy three kept the federated supergraph, but introduced two implementation optimizations: request batching at the resolver level, and entity-level caching when performing repeated key lookups in view composition. The choice of these optimizations is explained by the fact that the literature on communication patterns and gateway mediation suggests that not only the schema may introduce coordination overhead, but the frequent fan-out and redundancy of retrieval across services as well (Aksakalli et al., 2021; Zuo et al., 2020).

There were three representative user journeys that were performed. Aggregated personalized user data, featured products, up-to-date prices, inventory status, and snippets of recommendations were displayed on the dashboard workload. Product-detail workload had a combination of product core view, price, stock, review summaries and blocks of recommendation. Cart-summary workload was used to retrieve cart items, pricing, inventory validation and user specific discount details. To test every workload-strategy combination, 300 complete traces were obtained, resulting in a total of 2,700 observations. The start of a trace happened when the frontend module made its request of data and the end when the ultimate composed response needed to render was accessible to the presentation layer. This workload design represents the nested data-access patterns, which are emphasized in GraphQL research and distributed retrieval literature (Hartig and Perez, 2018; Ulrich et al., 2019).

The two dependent variables were mainly the mean end-to-end latency, as well as overall data of transferred responses per request. Frontend experience is also influenced by tail behavior, so p95 latency was selected as the complementary value. Moreover, the average downstream service calls per request, as well as the average number of hits in the cache were measured to describe the variation in performance between the strategies. Every observation was removed due to incomplete traces and the aggregate sums were calculated at the level of workloads and weighted overall. In order to avoid a single journey taking over the entire image, dashboard, product detail, and cart summary were given weight of 0.40, 0.35 and 0.25 respectively. This aggregation approach facilitated the ability to compare the global efficiency with maintaining workload-related interpretation.

The analysis process made a comparison of strategy-level means of each workload and analysis of weighted overall differences of all workloads. Practical significance was determined using percentage change. Two questions were highlighted in the discussion: federated GraphQL reduces the amount of data transferred relative to REST, and that latency only decreases when federation is used in conjunction with batching and caching. It is a framing that is supported by empirical necessity of Quiña-Mera et al. (2023), Brito and Valente (2020), and Lawi et al. (2021), to have clearer evidence about consumption-side GraphQL performance in real access conditions.

RESULTS AND DISCUSSION

These findings indicate that federated GraphQL enhanced data accuracy instantly, however, any latency benefits relied on the execution approach in the federated layer. In all three workloads, the simple federated GraphQL setup minimized the amount of response transferred compared to the REST baseline. But the latency effect was quite small and disproportionate. The federated setup that was optimized in terms of batching and entity-level caching yielded the best performance profile of all the parameters tested. This trend aligns with previous comparative research that demonstrated that the primary baseline advantage of GraphQL is selective data access, whereas the performance is limited to resolver orchestration quality and planning on the server-side (Brito and Valente, 2020; Lawi et al., 2021).

Latency and transferred data are reported on the workload level in Table 1. In the case of the dashboard workload, optimized federation minimized the mean latency value by 95.09 ms to 82.63 ms, and minimized the data transmitted by 42,530 bytes to 9,818 bytes. In case of the product-detail workload, the mean latency dropped by 87.63 ms to 76.40 ms and the amount of data transferred dropped by 18,655 bytes to 13,002 bytes. The cart-summary workload showed a more delicate trend: the simple federated GraphQL strategy added to the mean latency when compared to REST, as the latency value ranged between 54.34 ms and 63.27 ms, though the amount of transferred data decreased significantly. After the introduction of batching and caching, cart-summary latency dropped to 57.21 ms and transferred data decreased even more to 7,815 bytes. It means that shallow workloads might not be enhanced by federation in case query planning and resolver fan-out impose overhead that is bigger than the amount of savings due to the reduction of payloads (Hartig and Perez, 2018; Margaanska and Panicki, 2021).

Table 1. Latency and transferred data across workloads and strategies.

Workload	Strategy	Mean latency (ms)	P95 latency (ms)	Transferred data (bytes)
Dashboard	REST baseline	95.09	102.56	42530
Dashboard	Federated GraphQL	91.24	98.51	23450
Dashboard	Federated GraphQL + batching + cache	82.63	92.33	9818
Product detail	REST baseline	87.63	94.92	18655
Product detail	Federated GraphQL	82.19	88.79	15326
Product detail	Federated GraphQL + batching + cache	76.4	83.33	13002
Cart summary	REST baseline	54.34	59.24	20457
Cart summary	Federated GraphQL	63.27	68.67	13644
Cart summary	Federated GraphQL + batching + cache	57.21	63.77	7815

The cross-workload can be viewed better with the weighted summary. Compared to the REST baseline, a basic federated GraphQL decreased the average transferred data by 28.66 KB to 18.16 KB, decreasing the weighted mean latency by 1.47% but decreasing it only by 81.08 ms, which is a comparatively minor decrease of 36.65%. In comparison optimized federated GraphQL decreased weighted mean latency to 74.10 ms, a 9.96% improvement over REST and 8.61% improvement over basic federation. Weighted p95 latency decreased to 82.04 ms in optimized federation compared to 89.06 ms in REST, indicating that it was not only average behavior that improved. The shorthand interpretation is that field-level selectivity can resolve the payload problem, but not the coordination problem; coordination needs mechanisms to limit duplicated downstream retrieval and the same entity being resolved multiple times (Camilli and Russo, 2022; Aksakalli et al., 2021).

The weighted mean latency comparison is in figure 1. The visual pattern is used to validate the fact that basic federation and REST are not very far apart in terms of response time, but optimized federation is used to develop a more noticeable improvement. This separation is particularly crucial when using distributed frontends since view rendering can be sensitive to the slowest critical dependency chain, and not the median service call. Reducing p95 latency thus directly impacts the perceived interface responsiveness and consistency of user experience among micro-frontends (Peltonen et al., 2021; Taibi and Mezzalira, 2022).

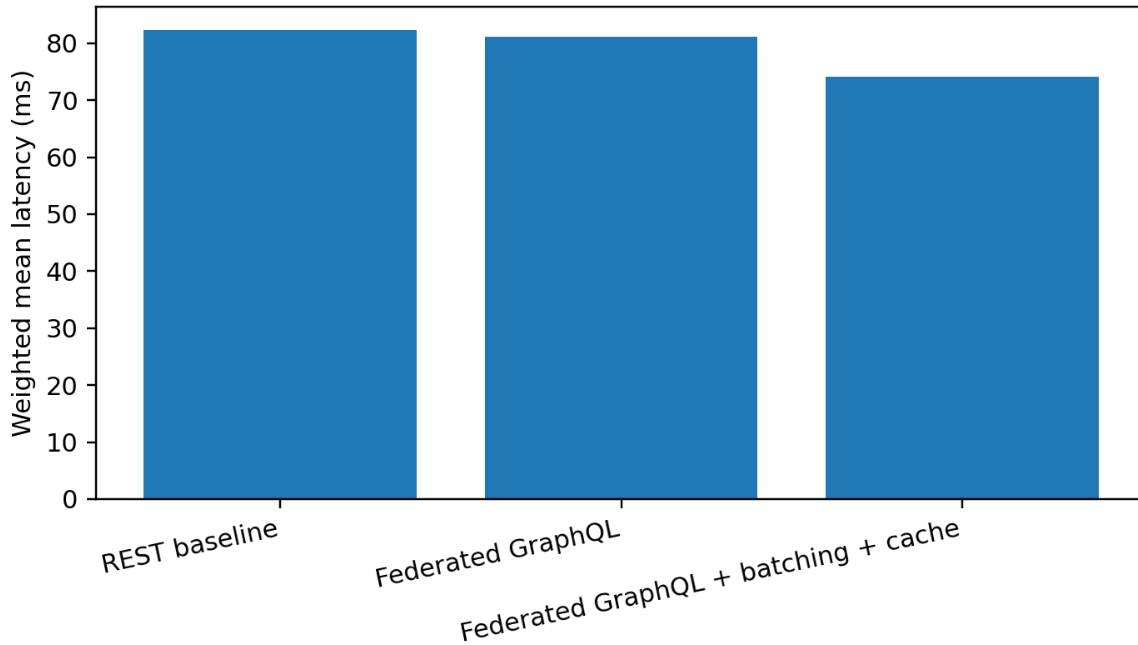

Figure 1. Weighted mean latency by strategy.

Even more pronounced difference is observed in transferred data, which is presented in Figure 2. The federated approach yielded the best weighted response volume of 63.60% compared to the REST baseline and 42.54% compared to basic federation. This observation is consistent with the previous works that have placed GraphQL as a powerful response-shaping construct especially in mobile and componentized frontend settings where unnecessary fields are rapidly accrued across independent modules (Lawi et al., 2021; Brito et al., 2019). The most significant reduction in data-volume in the current benchmark was the dashboard workload since the REST baseline combined several coarse-grained responses to meet a single composite view. Federation enabled the view to only retrieve the necessary fields and batching and caching restricted repeat access to common entities..

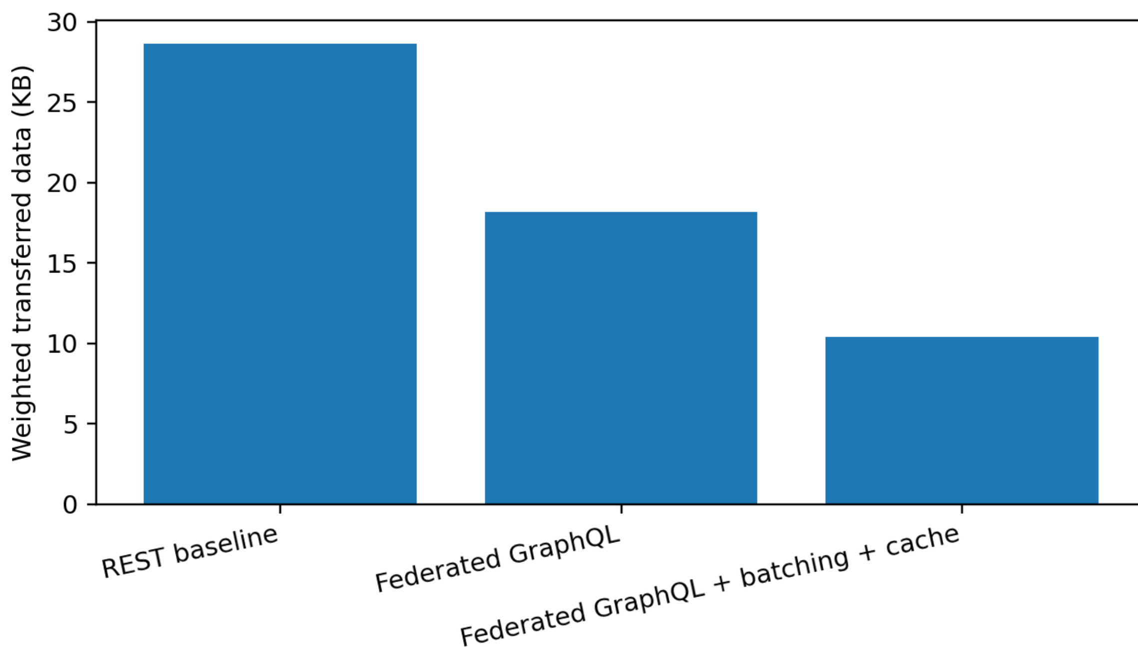

Figure 2. Weighted transferred data by strategy.

Table 2 aids in understanding why the optimized strategy was able to attain superior latency compared with basic federation. The weighted average number of downstream service calls decreased by 62.64 (REST vs. basic federation) and optimized federation (5.10 vs. 13.65). In the optimized case, mean cache hits were 3.15 per request, which means that the entity resolution of subgraphs had been absorbed, not reissued. The dashboard workload was the most benefited with the number of service calls dropped to 20 to 5 and the average number of hits to cache of 4 per request. The trend aligns with the microservices literature that demonstrates that the depth of fan-out and inter-service call chains are the key drivers of latency increases in the distributed systems (Di Francesco et al., 2019; Soldani et al., 2018).

Workload	**Strategy**	**Mean service calls/request**	**Mean cache hits/request**
Dashboard	REST baseline	20.0	0.0
Dashboard	Federated GraphQL	20.0	0.0
Dashboard	Federated GraphQL + batching + cache	5.0	4.0
Product detail	REST baseline	9.0	0.0
Product detail	Federated GraphQL	9.0	0.0
Product detail	Federated GraphQL + batching + cache	6.0	3.0
Cart summary	REST baseline	10.0	0.0
Cart summary	Federated GraphQL	10.0	0.0
Cart summary	Federated GraphQL + batching + cache	4.0	2.0

These results are substantively implied in two ways. To begin with, federated GraphQL is not to be considered a performance trade-off. Introduced without the execution strategies supporting its introduction, its main effect is to enhance data accuracy and schema consistency as opposed to ensuring reduced response time. Second, federated GraphQL can be truly an efficient data-fetching design in distributed frontends when orchestrating data with batching and caching is implemented at the orchestration layer. In such a case, the supergraph is not only a single contract, but also an optimization surface, minimizing the duplication of communication between modules and services. This finding is consistent with the rest of the microservices evidence on the fact that architectural decomposition should be supported by well-crafted communication and observability practices (Michael Ayas et al., 2023; Rezaei Nasab et al., 2023).

CONCLUSION

This paper has explored effective data-fetting techniques with federated GraphQL in distributed frontend architectures and shown that the value of federation in terms of performance is determined by execution design and is not just a matter of schema consolidation. The benchmark evidence demonstrated that basic federated GraphQL consistently minimized the transferred volume of responses, as expected, which validates its ability to under-fetch and better match frontend view needs with backend data contracts. Nonetheless, basic federation had a small latency advantage and on a single superficial workflow was worse than the REST baseline. The best performance was only achieved when federated GraphQL was used with request batching and entity-level caching.

The optimized federated strategy achieved a reduction of 9.96, 7.88, 63.60, and 62.64 in the mean latency, p95 latency, transferred data and service-call volume respectively, over the weighted workload set in comparison with the REST baseline. The findings suggest that federated GraphQL can specifically be effectively used in distributed frontends with composite views, repeated entity access, and cross-domain rendering dependencies. In that regard, the supergraph can provide a single contract to micro-frontends and it can also serve as an efficient focal point to reduce unnecessary network communication.

The extended value of the research is its connection of three research topics that are frequently referred to individuality: GraphQL query design, micro-frontends distribution, and microservice system communication efficiency. The results indicate that engineering choices in the future must consider federation as an option alongside resolver planning, batching policy and scope of cache instead of it being a singular API option. This analysis may be furthered in the future with mutation-intensive workflows, propagation of failures and the observability cost. Even on the current scale, the findings make it clear that the most effective way to do efficient data fetching in distributed frontends is to use federated GraphQL as an orchestration strategy with clear controls on fan-out, duplication and repeated entity resolution.

REFERENCES

[1]. Aksakalli, I. K., Çelik, T., Can, A. B., & Tekinerdoğan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software, 180, 111014. https://doi.org/10.1016/j.jss.2021.111014

[2]. Ataei, P., & Staegemann, D. (2023). Application of microservices patterns to big data systems. Journal of Big Data, 10, 56. https://doi.org/10.1186/s40537-023-00733-4

[3]. Brito, G., & Valente, M. T. (2020). REST vs GraphQL: A controlled experiment. In 2020 IEEE International Conference on Software Architecture (ICSA) (pp. 81–91). IEEE. https://doi.org/10.1109/ICSA47634.2020.00016

[4]. Brito, G., Mombach, T., & Valente, M. T. (2019). Migrating to GraphQL: A practical assessment. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 140–150). IEEE. https://doi.org/10.1109/SANER.2019.8667986

[5]. Camilli, M., & Russo, B. (2022). Modeling performance of microservices systems with growth theory. Empirical Software Engineering, 27, 39. https://doi.org/10.1007/s10664-021-10088-0

[6]. Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. Journal of Systems and Software, 150, 77–97. https://doi.org/10.1016/j.jss.2019.01.001

[7]. Hartig, O., & Pérez, J. (2018). Semantics and complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference (pp. 1155–1164). International World Wide Web Conferences Steering Committee. https://doi.org/10.1145/3178876.3186014

[8]. Kim, Y. W., Consens, M. P., & Hartig, O. (2019). An empirical analysis of GraphQL API schemas in open code repositories and package registries. In Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management (CEUR Workshop Proceedings, Vol. 2369). CEUR-WS.org.

[9]. Lawi, A., Panggabean, B. L. E., & Yoshida, T. (2021). Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system. Computers, 10(11), 138. https://doi.org/10.3390/computers10110138

[10]. Ma, S.-P., Fan, C.-Y., Chuang, Y., Liu, I.-H., & Lan, C.-W. (2019). Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems, 100, 724–735. https://doi.org/10.1016/j.future.2019.05.048

[11]. Margański, P., & Pańczyk, B. (2021). REST and GraphQL comparative analysis. Journal of Computer Sciences Institute, 19, 89–94. https://doi.org/10.35784/jcsi.2473

[12]. Michael Ayas, H., Leitner, P., & Hebig, R. (2023). An empirical study of the systemic and technical migration towards microservices. Empirical Software Engineering, 28, 85. https://doi.org/10.1007/s10664-023-10308-9

[13]. Peltonen, S., Mezzalira, L., & Taibi, D. (2021). Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. Information and Software Technology, 136, 106571. https://doi.org/10.1016/j.infsof.2021.106571

[14]. Quiña-Mera, A., Fernández-Montes, P., García, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A systematic mapping study. ACM Computing Surveys, 55(10), Article 202. https://doi.org/10.1145/3561818

[15]. Rezaei Nasab, A., Shahin, M., Hoseyni Raviz, S. A., Liang, P., Mashmool, A., & Lenarduzzi, V. (2023). An empirical study of security practices for microservices systems. Journal of Systems and Software, 198, 111563. https://doi.org/10.1016/j.jss.2022.111563

[16]. Soldani, J., Tamburri, D. A., & van den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software, 146, 215–232. https://doi.org/10.1016/j.jss.2018.09.082

[17]. Taibi, D., & Mezzalira, L. (2022). Micro-frontends: Principles, implementations, and pitfalls. ACM SIGSOFT Software Engineering Notes, 47(4), 25–29. https://doi.org/10.1145/3561846.3561853

[18]. Ulrich, H., Kern, J., Tas, D., Kock-Schoppenhauer, A. K., Ückert, F., Ingenerf, J., & Lablans, M. (2019). QL4MDR: A GraphQL query language for ISO 11179-based metadata repositories. BMC Medical Informatics and Decision Making, 19, 45. https://doi.org/10.1186/s12911-019-0794-z

[19]. Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software, 182, 111061. https://doi.org/10.1016/j.jss.2021.111061

[20]. Zuo, X., Su, Y., Wang, Q., & Xie, Y. (2020). An API gateway design strategy optimized for persistence and coupling. Advances in Engineering Software, 148, 102878. https://doi.org/10.1016/j.advengsoft.2020.102878