

THE EVOLUTION OF WEB SERVICES: HISTORY AND SYSTEMATIZATION

Maksymilian IWANOW

Department of Information Technology,

Wrocław University of Economics and Business: Komandorska 118/120 Wrocław 53-345 Poland,

E-mail: maksymilian.iwanow@ue.wroc.pl

ABSTRACT

The evolution of web services has fundamentally transformed the way software components interact over the Internet. They enable distributed computing, interoperability, and the development of scalable web applications. This article aims to systematically analyse the historical milestones, technical paradigms, and conceptual frameworks that have shaped web service development. The motivation for this study is to provide a comprehensive, structured overview of web service evolution, which is crucial for understanding both legacy integration and contemporary architectural choices. Findings highlight the trajectory from basic data exchange to highly efficient, scalable, and flexible services – providing insights into integration strategies, interoperability models, and service-oriented architectures. The study contributes to the scientific understanding of web service history and offers a conceptual foundation for future implementation-focused research in distributed systems and enterprise application development.

Keywords: Web service, Distributed Systems, SOA, CORBA, REST

1. INTRODUCTION

A web service can be defined as a self-describing, self-contained software module that is accessible over a network, such as the Internet [1]. It enables the execution of diverse computational tasks, problem-solving operations, and business transactions [2]. Over time, this concept has transformed the Web from a static collection of hyperlinked information into a dynamic, distributed computational platform capable of supporting complex, interoperable applications [3].

At its core, a web service is a software component that functions independently of the underlying hardware, operating system, or implementation technologies while providing specific, well-defined functionality. Conceptually, it can be viewed as a “black box” [4]; it receives inputs, processes them according to internal logic, and produces outputs. This abstraction enables the construction of distributed systems and Internet-based applications, allowing software components to interact seamlessly across distinct environments without requiring knowledge of each component’s internal implementation.

Interaction with web services is facilitated through standardized protocols and interfaces, with selection guided by functional and non-functional requirements. For service consumers, choosing an appropriate web service that satisfies specific needs is a critical challenge [5].

This article aims to systematize the historical milestones in the evolution of web services by examining both: their conceptual foundations and practical implementations. Through a focused review of the relevant literature, it traces key technological transitions and changing ways in which services have been designed, integrated, and managed over time. The study combines technical and business perspectives to build a framework for understanding how web services have been applied in different technological contexts. By organizing this knowledge, the review provides a structured reference that can be used to analyze contemporary service-oriented architectures and interoperability strategies. In this way, it supports researchers and practitioners in evaluating

existing solutions and designing more effective service-based systems.

The remainder of this article is structured as follows. The next section explores the technical and business significance of the service concept, laying the groundwork for understanding subsequent milestones. Following chapters chronologically examine the evolution of web services – from early networked services like E-Mail and FTP, through CORBA, SOAP, REST, and finally GraphQL – highlighting both conceptual innovations and practical impact. The article concludes with key findings and potential directions for future research.

2. FROM ISOLATION TO INTEGRATION: THE TECHNICAL AND BUSINESS RATIONALE FOR WEB SERVICES

Before the emergence of service-oriented systems, most software solutions functioned as isolated, monolithic entities, designed to serve specific organizational purposes. Such systems operated independently and rarely communicated with one another, even within the same company. As a result, organizations relying on legacy infrastructures faced substantial barriers in data exchange and inter-system communication, both internally and externally [6] [7].

This technological isolation led to several critical business and operational consequences [8]:

- Fragmented business processes, heavily dependent on human mediation to bridge disconnected system components.
- High maintenance and operational costs, particularly during software updates, due to the scarcity of qualified specialists for legacy platforms.
- Limited scalability, making it difficult for systems to handle growing workloads or adapt to new business demands.
- Restricted real-time processing, available only for a narrow subset of functionalities.

- Increased vulnerability and data security risks, resulting from outdated architectures and poor support for modern protection mechanisms.

The evolution of web services directly addressed these deficiencies. By introducing the concept of Service-Oriented Architecture (SOA), organizations gained a framework capable of integrating distributed and heterogeneous systems while maintaining modularity and flexibility. This architectural paradigm provided a foundation for building loosely coupled, interoperable components that could communicate seamlessly across networks, reducing the dependency on platform-specific technologies [9]. From a business standpoint, the advantages of adopting web services were both practical and strategic.

First, they made it possible to integrate legacy systems without rewriting them entirely. Rebuilding such systems is typically costly and time-consuming; however, by wrapping existing software components with service interfaces, companies could extend their lifespan and enable efficient data exchange [10] [9]. This approach offered a pragmatic solution – improving process efficiency without requiring full redevelopment.

Another key benefit was the increase in standardization across technologies. Although web services can be implemented in multiple programming languages, the underlying protocols, design principles, and hosting methods have become increasingly unified [9] [11]. Such standardization simplified system updates, extensions, and maintenance, substantially reducing long-term operational costs compared to traditional, tightly coupled architectures.

Equally significant is the scalability of web services [12] [13]. They are commonly hosted in server farms or data centers, where the number of service instances can be dynamically adjusted in response to traffic demands. Replicating or load-balancing instances of a service [14] ensures stable performance, minimizes response times, and enables real-time processing even under heavy workloads.

Security has also improved substantially. By leveraging standardized, secure transfer protocols and message formats, web services offer reliable data transmission even for highly sensitive business operations [10].

Finally, web services are inherently modular and composable [11] [10]. Each service encapsulates a specific business functionality, exposing only the necessary interfaces while concealing its internal logic and technical details.

From a technical perspective, web services consist of a set of well-defined components that ensure interoperability and communication across distributed environments [15] [16] [17]:

- Interface – Defines the set of operations a service offers, the input and output parameters, and the data types used. Interface specifications are expressed through formal description languages to guarantee clarity and consistency.
- Protocol – Determines how data is exchanged between applications. The evolution of communication protocols has played a crucial role in shaping the architectural styles of modern service frameworks.

- Data Format – Establishes a standardized structure for information exchange. Messages must conform to predefined schemas to ensure that data can be correctly interpreted by all communicating parties.
- Endpoint – Represents the network location where a service can be accessed. Each endpoint defines how and where clients can interact with the resources provided by the service.

3. PRE-WEB FOUNDATIONS: COMMUNICATION THROUGH E-MAIL AND FTP

In the earliest phase of Internet development, long before the concept of web services emerged, digital communication and data exchange were realized through a handful of simple yet revolutionary protocols. Among them, E-Mail and the File Transfer Protocol (FTP) became the two most influential technologies that shaped the early landscape of interconnected systems. Although they cannot be classified as web services in the strict, modern sense, they laid the conceptual and infrastructural groundwork for the development of distributed service-oriented solutions.

Electronic mail (E-Mail) represented one of the earliest mechanisms for asynchronous digital communication. It operates on the Simple Mail Transfer Protocol (SMTP) [18], a standardized communication method for routing and delivering messages over the Internet. SMTP is based on a store-and-forward model – each message is relayed between intermediate mail servers until it reaches its target destination. Upon arrival, the message is retrieved by the recipient's client using either the Post Office Protocol (POP) [19], or the Internet Message Access Protocol (IMAP) [20].

Although E-Mail originally supported only plain text, it soon evolved to transmit rich multimedia content, including images, documents, and other attachments. This was made possible through the Multipurpose Internet Mail Extensions (MIME) standard [21] which introduced a consistent mechanism for encoding and decoding diverse data types.

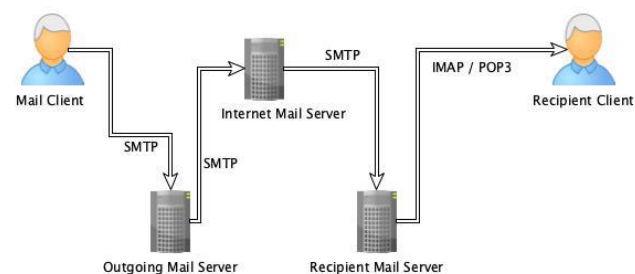


Fig. 1 E-Mail communication architecture

Parallel to E-Mail's evolution as a communication channel, the File Transfer Protocol (FTP) emerged as the primary tool for transferring data across networked systems [22]. FTP employs a client-server model [23], in which the client initiates a connection and transmits commands to the server to perform operations such as uploading, downloading, or listing files.

FTP communication is established through two distinct channels: a control channel responsible for sending commands, and a data channel that handles the actual file transmission. This separation improved efficiency and

enabled simultaneous command and data processing. Over time, the protocol evolved to include more secure variants, such as the Secure File Transfer Protocol (SFTP) [24] [25] [26] which added encryption to protect data in transit.

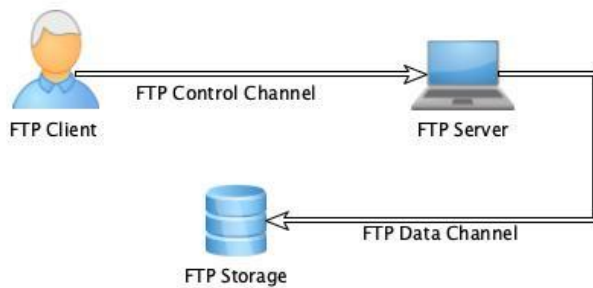


Fig. 2 FTP client-server architecture

Despite their pioneering roles, both E-Mail and FTP were inherently limited compared to today's dynamic web service architectures. Their data formats lacked strong standardization, and protocols were not designed for real-time, machine-to-machine communication. E-Mail was primarily intended for human interaction – with automation supported only in marginal, highly customized cases. Similarly, FTP's utility was restricted to transferring static files; it lacked the flexibility required for interactive, programmatic data exchange or integration across diverse platforms.

However, these early tools played a critical transitional role in shaping the understanding of what the Internet could achieve. They established key principles such as interoperability across heterogeneous systems, protocol-based communication, and asynchronous message exchange.

4. CORBA: THE FIRST STEP TOWARD STANDARDIZED DISTRIBUTED SERVICES

The Common Object Request Broker Architecture (CORBA) marked one of the earliest and most ambitious attempts to establish a standardized framework for distributed computing. Developed under the auspices of the Object Management Group (OMG), CORBA sought to provide a universal architecture enabling applications to interact seamlessly across different platforms, operating systems, and programming languages [27]. Although it predates what we today refer to as web services, CORBA can be viewed as their conceptual predecessor – an early middleware technology that introduced many of the principles later refined in service-oriented architectures.

At its core, CORBA operates as middleware, mediating communication between distributed objects in a client-server model [23]. A client application can invoke methods on remote objects residing on another host as if they were local, while the middleware layer – the Object Request Broker (ORB) – handles the complexity of locating, activating, and coordinating those objects. This approach abstracted away many low-level networking details and provided a high level of interoperability between heterogeneous systems.

CORBA defined a clear set of technical components to standardize how distributed applications communicate:

- **Interface:** CORBA specifies interfaces using the Interface Definition Language (IDL) [28]. The IDL formally describes the operations, data types, and attributes that a service exposes. This formalization allows both clients and servers – even if implemented in different languages – to dynamically discover and invoke remote methods without knowing internal implementation details.
- **Protocol:** Communication between distributed objects relies on the General Inter-ORB Protocol (GIOP) and its Internet mapping, the Internet Inter-ORB Protocol (IIOP) [29]. The latter enables CORBA objects to exchange requests and responses over standard TCP/IP networks, ensuring interoperability across diverse programming environments.
- **Data Format:** CORBA's IDL defines a consistent set of basic data types – such as short, long, float, and string – to describe message structures [28]. For example, a banking service might include a `getBalance()` method returning a float, while a bidding service could define a `wannaBid()` operation returning a boolean.
- **Endpoint:** Every CORBA object is identified by an Interoperable Object Reference (IOR), which encodes the location and access information of the object's interface. These references are opaque entities – their contents are hidden from developers – which introduced both flexibility and complexity when integrating with external APIs [30].

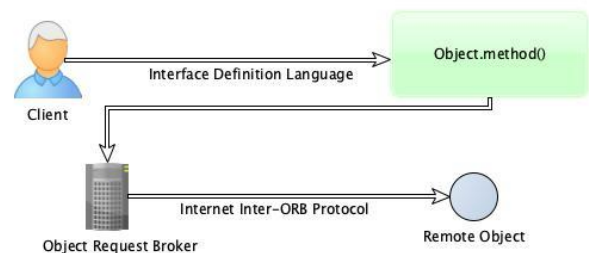


Fig. 3 CORBA distributed object architecture

CORBA's key contribution lies in its standardization of distributed object communication. By abstracting network, platform, and language differences, CORBA allowed developers to build complex, interoperable systems without manually handling socket connections or data serialization. Unlike earlier solutions such as E-Mail or FTP, CORBA enabled real-time, automated interactions between applications, effectively removing the human factor from routine communication processes. This positioned it as one of the first genuine precursors of web services – a bridge connecting the world of standalone applications with the emerging vision of interconnected, service-based computing.

Furthermore, CORBA offered significant benefits in terms of integration of legacy systems. Many existing applications could be wrapped as CORBA objects, exposing their functionality to other distributed components without the need for rewriting entire systems. This modularity and interoperability made it a powerful middleware option during the 1990s for enterprises seeking to modernize their infrastructures.

Despite its strong conceptual foundation, CORBA struggled to achieve widespread industry adoption. The framework was technically complex, requiring considerable expertise and programming effort to implement effectively. Developers often faced difficulties in debugging distributed object interactions, and the learning curve for IDL and ORB configuration was steep. Additionally, performance limitations and the lack of built-in encryption or transport security further hindered its long-term success [30]. In spite of that, CORBA still works in extremely legacy business and government environments (e.g. healthcare, medicine) [31] [32].

5. SOAP: STANDARDIZING STRUCTURED COMMUNICATION IN THE WEB ERA

Following the emergence of CORBA, the next major step toward the standardization of web services came with the introduction of the Simple Object Access Protocol (SOAP). Unlike its object-oriented predecessor, SOAP shifted the focus from distributed objects to message-oriented communication, offering a lightweight, interoperable mechanism for exchanging structured information between applications across the Internet. Designed in the late 1990s and maintained by the World Wide Web Consortium (W3C), SOAP became the foundation upon which modern web service architectures were built [33].

SOAP is essentially a protocol for structured message exchange that defines how data should be packaged, transmitted, and interpreted between distributed systems. It uses the Extensible Markup Language (XML) as its message format, which ensures both human readability and machine interpretability. Through XML-based encoding, SOAP provides a universal standard for representing data types, method calls, and message responses in a consistent way – independent of platform, operating system, or programming language.

Functionally, SOAP operates in a client–server architecture [23]. The client sends a structured request specifying the desired operation, while the server processes it and responds with the requested data. In this way, SOAP serves as both a messaging protocol and a framework for defining and invoking remote operations, enabling interoperability across heterogeneous environments.

SOAP's architecture is composed of several interdependent technical components that define how web services are exposed and consumed:

- **Interface:** SOAP employs the Web Service Description Language (WSDL) to describe the functionalities a service offers [34]. WSDL file, written in XML, provides a machine-readable specification detailing the service's available methods, resource identifiers, arguments, and associated data types [35]. It serves as a formal contract between service providers and consumers, allowing systems to interact without prior human agreement on implementation details.
- **Protocol:** SOAP messages are typically transmitted over the Hypertext Transfer Protocol (HTTP), most often via the POST method [36] although alternative transport protocols such as SMTP or TCP can also be used. This transport flexibility contributed to SOAP's

widespread adoption across both web-based and enterprise networks.

- **Data Format:** The SOAP message structure follows a hierarchical XML schema built around the root `<Envelope>` element, which encapsulates the `<Header>` and `<Body>` components. The Envelope defines the overall structure of the message; the Body carries the primary payload, including method calls and returned data; the Header (optional) holds metadata such as authentication credentials or routing information. Additionally, the Fault element is used to report errors or exceptions that occur during message processing [35]. This design provides both clarity and extensibility, allowing additional application-specific information to be seamlessly integrated into the message flow.
- **Endpoint:** WSDL also defines endpoints – the network addresses through which clients access specific service resources. These endpoints specify how SOAP operations are bound to transport protocols and provide the exact URLs or network locations for invocation. Importantly, endpoints can be customized without constraining how receiving systems implement their own configurations [37] [38].

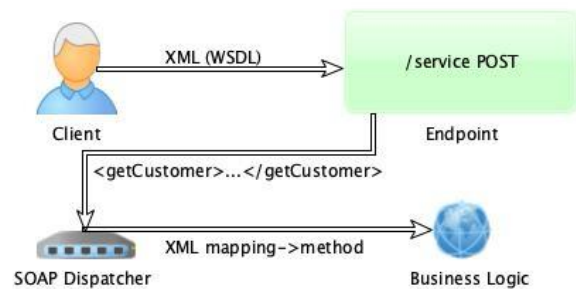


Fig. 4 SOAP web services architecture

One of SOAP's most significant achievements was its universality. By using XML as the common language for data representation, SOAP enabled heterogeneous systems – written in different programming languages and deployed on different platforms – to communicate seamlessly. It quickly became the standard for enterprise-level web services, particularly in sectors requiring strict structure, interoperability, and reliability.

In addition, SOAP's design allowed for integration with existing Internet infrastructure, leveraging HTTP to pass through firewalls and operate within conventional web architectures. This adaptability contributed greatly to its popularity during the early 2000s, when it became the core protocol for service-oriented architectures (SOA).

Despite its success, SOAP also introduced several challenges. The reliance on XML – while advantageous for standardization – imposed performance overheads due to verbose message structures and the computational cost of parsing. Moreover, the protocol's complexity often required specialized tools and frameworks for implementation, increasing development effort. Over time, these limitations led developers to seek simpler, more lightweight alternatives.

6. REST: THE LIGHTWEIGHT REVOLUTION IN WEB SERVICES

At the beginning of the 21st century, the landscape of web services underwent a fundamental transformation with the emergence of the Representational State Transfer (REST) architectural style. Introduced by Roy Fielding in his doctoral dissertation [39], REST proposed a simpler, more intuitive alternative to the heavy, protocol-driven designs of technologies like SOAP. Instead of relying on complex XML-based messaging systems, REST drew inspiration directly from the principles that underpin the World Wide Web, using its native mechanisms – particularly the HTTP protocol – as the foundation for distributed communication.

REST defines a set of architectural constraints for designing networked applications, emphasizing simplicity, scalability, and performance. It operates according to the client-server model [23], where clients send requests to servers to access or manipulate resources, and servers respond with the appropriate representations of those resources. Unlike previous paradigms, REST does not require specialized middleware or service brokers; instead, it treats every piece of data as a uniquely addressable resource that can be retrieved, modified, or deleted through standard web operations.

This design promotes a stateless communication model, meaning each client request must contain all the information necessary for the server to process it. The server does not maintain session data between interactions, which greatly simplifies scalability and load distribution across multiple servers.

Although REST is an architectural style rather than a formal protocol, it relies on several foundational components that make it consistent and easy to implement across different environments:

- **Interface:** RESTful web services expose uniform interfaces based on standard HTTP methods such as GET, POST, PUT, and DELETE [40] [41]. These phrases correspond directly to the basic operations of resource manipulation – retrieval, creation, updating, and deletion. To describe and document available endpoints, REST commonly uses tools such as Swagger [42] and OpenAPI [43]. Both frameworks define formal contracts written in JavaScript Object Notation (JSON) [44] or YAML (Ain't Markup Language) [45] ensuring clarity and interoperability across development teams.
- **Protocol:** REST uses the HTTP protocol as its primary communication layer, leveraging its native status codes to indicate operation results [46] [23]. The four principal HTTP methods are standardized across implementations: GET – retrieve a resource, POST – create a new resource, PUT – update an existing resource, DELETE – remove a resource. This uniformity allows developers to build and consume APIs using well-known web semantics, without the need for proprietary or domain-specific message formats.
- **Data Format:** REST supports flexible message representations, typically using JSON or XML for data serialization. However, JSON has become the

dominant choice due to its lightweight structure, readability, and seamless integration with modern web technologies such as JavaScript. Its efficiency in serialization and deserialization makes it particularly suitable for web and mobile applications, where performance and low latency are key concerns.

- **Endpoint:** Each resource in a RESTful system is identified by a Uniform Resource Locator (URL) [47]. These endpoints represent specific entities or collections of data that can be directly accessed through HTTP operations. REST's stateless nature requires that each request be self-contained; the server does not maintain session information between consecutive interactions, which enhances scalability and reliability.

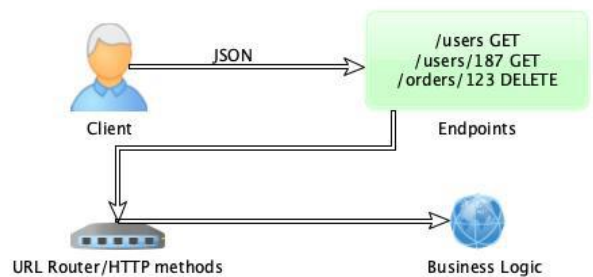


Fig. 5 RESTful service architecture

The primary strength of REST lies in its simplicity and scalability. By aligning with the existing web infrastructure, RESTful services are easy to develop, maintain, and extend. The architecture naturally supports horizontal scaling, as each request is independent and can be handled by different servers without additional synchronization overhead.

Furthermore, REST has been widely embraced by modern frameworks and programming languages, making it a default choice for building APIs in web and mobile environments. Its resource-oriented design, use of standard HTTP methods, and compatibility with browser technologies have made REST the most prevalent architecture for public web services. According to industry surveys (e.g., *SmartBear*), approximately 70% of public APIs rely on RESTful principles, underscoring its ubiquity across digital ecosystems.

REST also provided a conceptual shift: rather than modeling services as procedural operations (like in SOAP), it treats the Web as a network of resources, promoting uniformity, cacheability, and scalability – core principles that reflect the very essence of the Internet [48].

Despite its strengths, REST is not without limitations. It offers limited support for complex transactions or long-running operations, which may require multiple, stateful interactions. In data-intensive applications, repeated requests and statelessness can also introduce performance inefficiencies.

Nonetheless, REST's balance between simplicity, flexibility, and power has solidified its position as the dominant paradigm for modern web services. It represents a crucial milestone in the ongoing evolution of distributed systems.

7. GRAPHQL: PRECISION AND FLEXIBILITY IN MODERN WEB SERVICES

In recent years, the evolution of web services has been further advanced by GraphQL, a technology designed to provide more flexible, efficient, and precise methods for application interaction over the Internet. Originally developed by Facebook in 2012, GraphQL has since matured into a widely adopted alternative to REST, particularly for complex, data-centric applications [49].

GraphQL introduces several key components that differentiate it from RESTful and SOAP-based services:

- **Interface:** GraphQL employs an Interface Description Language (IDL) to define its schema, outlining the types, fields, and operations exposed by a service [50]. Each object type explicitly specifies which fields are retrievable, providing clients with a clear understanding of available data. This strongly-typed schema promotes consistency, maintainability, and predictability across complex applications [51].
- **Protocol:** Communication is typically handled over HTTP, leveraging its ubiquity and existing infrastructure. While POST requests are the default method for sending queries, GET requests may also be accepted for specific use cases [52]. This flexibility allows GraphQL to integrate seamlessly into modern web architectures.
- **Data Format:** Both clients and servers employ JSON serialization for request and response messages. Responses are usually labeled as *application/graphql-response+json*, although *application/json* can be used to maintain compatibility with legacy clients [52].
- **Endpoint:** Unlike REST, which requires separate URLs for each resource, GraphQL exposes a single endpoint (e.g., `/graphql`) through which all queries are executed [52]. Clients construct queries specifying exactly what data is required, minimizing unnecessary payloads and improving efficiency.

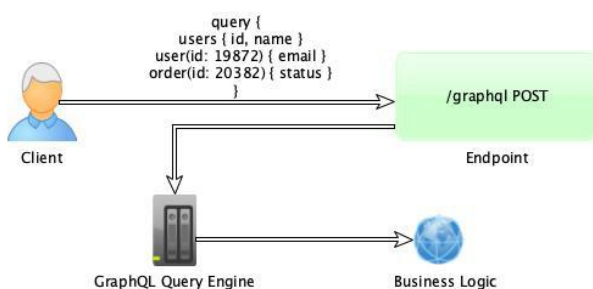


Fig. 6 GraphQL API architecture

The primary strength of GraphQL lies in its adaptability and efficiency. By allowing clients to retrieve precisely the data they need, it reduces network overhead and accelerates application performance. Its strongly-typed schema provides a clear contract between client and server, simplifying the development, testing, and maintenance of complex applications. This precision is particularly valuable in modern web and mobile ecosystems, where bandwidth constraints and diverse client requirements are common challenges.

Moreover, GraphQL supports evolving and modular APIs, enabling services to add new fields and types without breaking existing queries. This extensibility, combined with precise data selection, makes it ideal for large-scale, data-intensive applications that require both flexibility and consistency.

Despite its advantages, GraphQL also presents challenges. The learning curve can be steep for developers accustomed to REST, and query complexity may lead to performance issues when handling large volumes of nested data. Consequently, while GraphQL excels in data-rich environments, it may be overkill for simpler applications or services with limited data exchange needs.

8. CONCLUSIONS

The article has traced the evolution of web services from early communication tools such as E-Mail and FTP, through the development of CORBA, SOAP, REST, and GraphQL.

Early technologies such as E-Mail and FTP were designed primarily for human interaction, offering limited support for automated system-to-system communication. CORBA, and later SOAP, REST, and GraphQL, progressively shifted the focus toward fully automated, programmatic interfaces capable of real-time data exchange.

Middleware-oriented and strictly typed systems like CORBA and SOAP introduced powerful but complex mechanisms for defining and enforcing communication contracts. REST relaxed these constraints by embracing the simplicity of HTTP and lightweight data formats, while GraphQL reintroduced strong typing in a more flexible form.

Technologies such as CORBA relied on dedicated brokers and specialized middleware layers to mediate distributed operations, adding both power and complexity. SOAP maintained this reliance through extensive tooling and XML processing. REST eliminated intermediary infrastructure by aligning directly with native web standards. GraphQL builds upon this simplicity, demonstrating that modern service architectures benefit most from minimal transport overhead and broad compatibility.

Early approaches focused largely on ensuring that heterogeneous systems could communicate reliably across networks, emphasizing compatibility above all else. Modern paradigms such as REST and GraphQL place increasing emphasis on efficiency, responsiveness, and tailoring data delivery to the needs of diverse clients, especially web and mobile applications.

The analysis shows that different web service technologies are suited to different organizational and technical scenarios. Legacy environments that require strict contracts and strong interoperability may still benefit from SOAP or CORBA (despite their complexity). In contrast, REST is well suited for scalable, web-based and cloud-native systems where simplicity and compatibility are priorities. GraphQL is particularly advantageous in data-intensive applications with diverse clients where flexible queries and efficient data transfer are essential. Consequently, technology selection should be driven not by historical popularity, but by the specific requirements for

integration, performance, and client diversity in a system context.

Future research should move beyond the historical and conceptual analysis presented here to the implementation studies across the discussed architectures. Practical experiments could focus on interoperability of legacy systems, performance and resource usage of REST and GraphQL, security mechanisms, schema optimization, and the development of tools for testing and monitoring web services.

Such implementation-focused research would complement the historical and conceptual perspective provided in this paper, offering practical guidance for engineers and researchers seeking to design, deploy, and optimize modern distributed services in a variety of enterprise and academic contexts.

REFERENCES

- [1] KREGER, H.: "Web Services Conceptual Architecture (WSCA 1.0)," IBM, 2001.
- [2] PAPAZOGLU, M.: Web services: principles and technology, Pearson Prentice Hall, 2007.
- [3] FENSEL, D. – BUSSLER, C.: "The Web Service Modeling Framework WSMF," *Electronic Commerce Research and Applications*, vol. 1, no. 2, pp. 113-137, 2002.
- [4] ZHU, J. – HE, P. – XIE, Q. – ZHENG, Z. – LYU, M. R.: "CARP: Context-Aware Reliability Prediction of Black-Box Web Services," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017.
- [5] HAO, Y. – ZHANG, Y. – CAO, J.: "A novel QoS model and computation framework in web service selection," *World Wide Web*, vol. 15, pp. 663-684, 2012.
- [6] SNEED, H.: "Integrating legacy software into a service oriented architecture," in *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006.
- [7] ZAHRA, S. W. – NADEEM, M. – ABBASI, M. N. – ARSHAD, A. – RIAZ, S. – AHMED, W.: "Systematic Literature Review on Web Services in E-commerce and Business Communication," *E-Commerce for Future & Trends*, vol. 10, no. 3, 2023.
- [8] ROCHIMAH, S. – SHEKU, A.: "Migration of Existing or Legacy Software Systems into Web Service-based Architectures (Reengineering Process): A Systematic Literature Review," *International Journal of Computer Applications*, vol. 133, no. 3, pp. 43-54, 2016.
- [9] CONDE-GONZÁLEZ, M. – GARCÍA-PEÑALVO, F. J. – CASANY, M. – FORMENT, M. A.: "Adapting LMS architecture to the SOA: an Architectural Approach," in *Fourth International Conference on Internet and Web Applications and Services, ICIW 2009*, 2009.
- [10] VENKATRAMAN, S. S.: "Web-Services – The Next Evolutionary Stage of E-Business," *Journal of International Technology and Information Management*, 2004.
- [11] SHELTON, W. T.: "Web Services: A Strategic Analysis," Massachusetts Institute of Technology, 2003.
- [12] ALLAMARAJU, S.: RESTful Web Services Cookbook. Solutions for Improving Scalability and Simplicity, Yahoo Press, 2010.
- [13] GRICE, L. – BOTHA, R.: "E-Business Integration: Are Web Services the answer?," *Information Technology On The Move*, 2002.
- [14] HASHEMIAN, R. – KRISHNAMURTHY, D. – ARLITT, M. – CARLSSON, N.: "Characterizing the scalability of a Web application on a multi-core server," in *Special issue on International Conference on Performance Engineering 2013 (ICPE 2013)*, 2014.
- [15] BEAN, J.: "Web Services and Other Service Types and Styles," in *SOA and Web Services Interface Design*, 2010, pp. 43-54.
- [16] BEYER, D. – CHAKRABARTI, A. – HENZINGER, T. A.: "Web service interfaces," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*, 2005.
- [17] BENATALLAH, B. – CASATI, F. – TOUMANI, F.: "Representing, analysing and managing Web service protocols," *Data & Knowledge Engineering*, vol. 58, no. 3, pp. 327-357, 2006.
- [18] KLENSIN, J.: "Simple Mail Transfer Protocol," *RFC 2821*, 2001.
- [19] ROSE, M.: "Post Office Protocol: Version 3," 1988.
- [20] CRISPIN, M.: "Internet Message Access Protocol 4(1)," 2003.
- [21] FREED, N. – BORENSTEIN, N. S.: "Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples," *RFC 2049*, 1996.
- [22] POSTEL, J.: "File Transfer Protocol specification," *RFC 765*, 1980.
- [23] HAROON-SULYMAN, S. O.: "Client-server model," *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 57-71, 2014.
- [24] COHEN, G.: "Securing FTP," *login Usenix*, vol. 27, no. 4, 2002.
- [25] SEKHAR, U. I. – RAMADEVI, M. – SAI, B. V. – MUNDURAVINASH: "A Network Protocol That Uses Secure Shell for Secure, Encrypted File Transfers, Offering an Alternative to the Traditional File Transfer Protocol," *Journal of Nonlinear Analysis and Optimization*, vol. 16, no. 1, pp. 322-328, 2025.
- [26] BORIWAN, P. – NUNTA, A. – PHOKA, T. – KANZAKI, A.: "Semi-Closed File Transfer System With Transfer Time Optimization," *ICIC Express Letters*, vol. 19, no. 1, pp. 29-37, 2025.
- [27] Object Management Group, "Common Object Request Broker Architecture (CORBA) Specification, Version 3.4," 2021.

- [28] CANAL, C. – FUENTES, L. – PIMENTEL, E. – TROYA, J. M. – VALLECILLO, A.: "Extending CORBA interfaces with protocols," *The Computer Journal*, vol. 44, no. 5, pp. 448-462, 2001.
- [29] SCHMIDT, D. C. – KUHN, F.: "An overview of the Real-Time CORBA specification," *Computer*, vol. 33, no. 6, pp. 56-63, 2000.
- [30] HENNING, M.: "The Rise and Fall of CORBA: There's a lot we can learn from CORBA's mistakes.," *Queue*, vol. 4, no. 5, pp. 28-34, 2006.
- [31] LAKHAN, A. – MORTEN GROENLI, T. – MAJUMDAR, A. – KHUWUTHYAKORN, P. – HUSSAIN KHOSO, F. – THINNUKOL, O.: "Potent Blockchain-Enabled Socket RPC Internet of Healthcare Things (IoHT) Framework for Medical Enterprises," *Sensors*, vol. 22, no. 12, 2022.
- [32] KOVALEV, I. – KOVALEV, D. – KOVALEV, R. – PODOPLELOVA, V. – LOSEV, V. – VOROSHILOVA, A. – BOROVINSKY, D. – GADOEVA, M.: "The concept of architectural reliability of software for ensuring the functioning of request-free measuring stations," in *E3S Web of Conferences GEOTECH-2024*, 2024.
- [33] The World Wide Web Consortium, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," 2007.
- [34] CHINNICI, R. – MOREAU, J.-J. – WEERAWARANA, S. – RYMAN, A.: "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," *W3C working draft*, 2004.
- [35] SUDA, B.: "SOAP Web Services," University of Edinburgh, 2003.
- [36] POPA, A. GHODSI – STOICA, I.: "HTTP as the narrow waist of the Future Internet," in *Proceedings the 9th ACM SIGCOMM Workshop on Hot Topics Networks*, 2010.
- [37] GOVINDARAJU, M. – SLOMINSKI, A. – CHIU, K. – LIU, P. – VAN ENGELEN, R. – LEWIS, M. J.: "Toward characterizing the performance of SOAP toolkits," in *Fifth IEEE/ACM International Workshop on Grid Computing*, 2004.
- [38] FORNO, F. – SAINT-ANDRE, P.: "XEP-0072: SOAP Over XMPP," 2005.
- [39] FIELDING, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*, 2000.
- [40] RAPOLU, H. K.: "Comparative Analysis of Soap and Rest-For Enterprise-Level Applications," *Journal of Advances in Developmental Research*, vol. 12, no. 2, 2021.
- [41] RUBY, S. – RICHARDSON, L.: *RESTful Web Services*, O'Reilly Media, 2007.
- [42] VARANASI, B. – BARTKOV, M.: "Documenting REST Services," in *Spring REST*, 2012, pp. 129-145.
- [43] ESPINOZA-ARIAS, P. – GARIJO, D. – CORCHO, O.: "Mapping the Web Ontology Language to the OpenAPI Specification," in *Advances in Conceptual Modeling*, 2020, pp. 117-127.
- [44] PEZOA, F. – REUTTER, J. L. – SUAREZ, F. – UGARTE, M. – VRGOC, D.: "Foundations of JSON Schema," in *Proceedings of the 25th international conference on World Wide Web*, 2016.
- [45] DOS SANTOS, J. S. – AZEVEDO, L. G. – SOARES, E. F. – THIAGO, R. M. – DA SILVA, V. T.: "Analysis of Tools for REST Contract Specification in Swagger/OpenAPI," in *22nd International Conference on Enterprise Information Systems*, 2020.
- [46] MISHRA, A.: "Understanding Foundational Web Services Architectures: A Comprehensive Review," *International Scientific Journal of Engineering and Management*, vol. 3, pp. 1-7, 2023.
- [47] CHOU, W. – LI, L.: "Communication over IP based on Web services," in *2008 IEEE International Conference on Web Services*, 2008.
- [48] FINCH, L.: "Comparative Study: Soap Vs Rest For Infra-Centric Data Transfer," 2023.
- [49] The GraphQL Foundation, "Introduction to GraphQL," 2024. [Online]. Available: <https://graphql.org/learn/>.
- [50] QUINA-MERA, A. – FERNANDEZ, P. – GARCIA, J. M. – RUIZ-CORTES, A.: "GraphQL: A Systematic Mapping Study," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1-35, 2022.
- [51] The GraphQL Foundation, "Schemas and Types," 2024. [Online]. Available: <https://graphql.org/learn/schema/>.
- [52] The GraphQL Foundation, "Serving over HTTP," 2024. [Online]. Available: <https://graphql.org/learn/serving-over-http/>.

Received October 21, 2025, Accepted February 18, 2026

BIOGRAPHY

Maksymilian Iwanow received engineer's degree in the Faculty of Electronics at Wrocław University of Science and Technology in 2018. Two years later, in 2020, he obtained a master's degree from the same university in the Faculty of Computer Science. In 2025, he completed his PhD entitled *Optimization of Web Services Composition in the Context of Quality Requirements Defined by Business Entities* at the Wrocław University of Economics and Business. In addition, he has been working as a Java web application developer for five years. He specializes in backend development & analysis within microservice architectures. His research interests include the quality of web services, prediction and optimization of web resources, and the microservice architecture pattern.