SvedbergOpen
DISSEMINATION OF KNOWLEDGE
International Journal of Artificial Intelligence and Machine Learning
Publisher's Home Page: https://www.svedbergopen.com/
Research Paper Open Access

Natural Language Orchestrator: Secure API Interoperability Via The Model Context Protocol

Vishal Shah

Independent Researcher, USA

Abstract
The stiffness of customary Application Programming Interfaces makes integrating systems in current enterprises very prohibitive. In this article, organizations provide an overview of the Natural Language Orchestrator, a new architectural layer that democratizes the consumption of APIs via semantic rather than schema-based understanding. Context exchange occurs via the Model Context Protocol, and state is orchestrated via LangGraph, as this allows a focus on unstructured natural language commands support with continuity of context and dynamic mapping to the right registered internal MCP local servers, decoupling the command from the technical implementation. High-level interoperability is pursued without sacrificing safety and security through a Zero-Trust model of requiring dual-layered defenses. Identity assurance through OAuth 2.0, requiring every API request to include a valid user-level JSON Web Token as proof of an authenticated source of origin. Integrity and confidentiality of messages are provided through an asymmetric Public Key Infrastructure. The orchestrator verifies the signatures of the request against the stored consumer public keys and encrypts the response. The framework builds on transformer-based semantic interpretation combined with graph-based workflow execution, and achieves high levels of intent recognition accuracy and endpoint selection. Its attributes include production readiness through horizontal scaling, high efficiency for context persistence through distributed caching, and cryptographic computations while maintaining throughput under high concurrency levels. This is a foundation for secure, autonomous, human-centric interaction with the system, which converts complex integrations into intuitive conversations.

Keywords: Natural Language Orchestration, Model Context Protocol, Semantic API Mapping, Zero-Trust Security Architecture, Microservices Interoperability.

1. Introduction

As enterprise software architectures have evolved over time, so have integration patterns between systems. While APIs have dominated integration for the last several decades, the strict contract imposed by APIs has become a key barrier to integration as it requires tight coupling between systems. In the modern distributed world, API orchestration usually has to account for the complexities of coordinating heterogeneous services in the presence of service dependencies, possible failures, and inconsistency. Furthermore, the cost of maintaining such an integration layer can become substantial, especially as the business processes and technology boundaries evolve [1].

When natural language processing tools are available, it is possible to rethink communication in systems, and recent conversational AI developments show that natural language interfaces can be the right option for complex, technical tasks. Conversational AI systems that involve transformers are more effective in identifying contextual intent and in generating contextually appropriate responses. Previous work has shown that LLMs, when fine-tuned with domain-specific data, can effectively translate unstructured natural language requests into structured API calls, especially in systems featuring a retrieval mechanism that identifies

relevant information from the API documentation or previous interactions [2]. The Natural Language Orchestrator is an architectural framework that translates user intent into machine action, based on command semantic parsing. It consists of the Model Context Protocol (MCP) to exchange model context in a vendor-agnostic way and LangGraph, a stateful orchestration approach that transforms API consumption into a conversation model.

The architecture is enterprise-grade secure with a Zero-Trust architecture employing OAuth 2.0 authentication and asymmetric PKI. API-driven architectures typically struggle with scalable authentication and authorization of users and data consumers. The secure architecture should reduce risk and protect sensitive information in transit and in-process. For security mechanisms deployed over the distribution, cryptographic mechanisms for integrity, authenticity, and confidentiality for data exchanged between heterogeneous services crossing a trust boundary must be considered because message data processed by multiple services may cross boundaries of trust. OAuth 2.0 was the most frequently used authorization protocol (16 occurrences in authentication literature [2]) for token-based access delegation, which minimizes the resources exposed to a user without having the user resort to repeatedly disclosing their credentials. Asymmetric encryption using public/private key pairs ensures that the response data will only be accessible by the authorized parties. In the context of microservices security, the decomposition of monolithic applications into microservices has been found to considerably increase the attack surface of the application. The introduction of an additional 100 thousand lines of code (KLOC) in the application corresponds to an average increase of 180 vulnerabilities in the microservice architecture, and 39 vulnerabilities in the monolithic architecture [2].

Further performance implications of natural language-driven orchestration systems come from the introduction of latency during semantic parsing, accessing context, invoking APIs, and generating a response. Semantic orchestration frameworks have been shown to provide operational benefits. For example, threshold-based optimization techniques achieved 9.4% less energy consumption and 19.3% tool wear in industrial digital twin deployments [1]. Thus, smart coordination mechanisms can potentially lead to concrete results beyond correctness. This focus on usability, coupled with reliability, enables human-centered system interactions with the robustness necessary at enterprise scale, production-ready deployments capable of catering to the performance requirements of demanding business needs, and democratized integration capabilities.

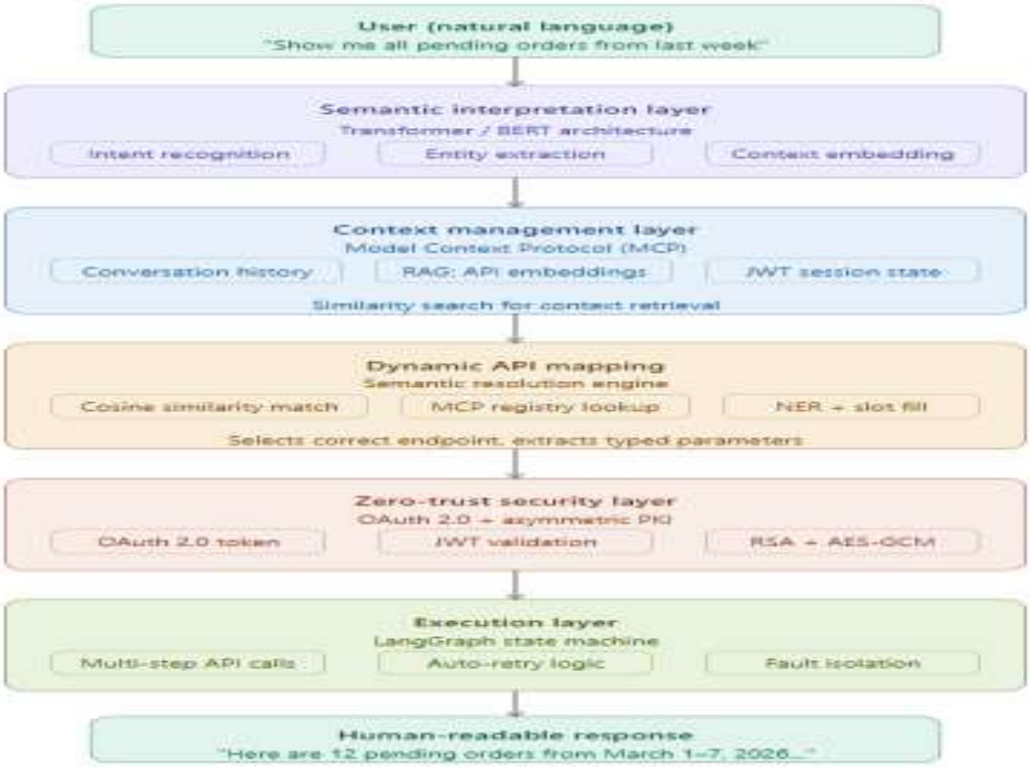

Fig. 1: NL Orchestrator flow

Table1: Traditional API Flow vs. NL Orchestrator Feature Comparison

Aspect	Existing API Flow	With NL Orchestrator
User Input	Structured API call with exact schema	Plain natural language command
API Knowledge	Developer must know endpoints	System resolves dynamically via semantic matching
Auth Handling	Manually managed per client	Centrally handled via OAuth 2.0 + JWT
Context	Stateless, no memory between calls	Stateful via MCP + distributed cache
Multi-service calls	Developer chains calls manually	LangGraph orchestrates automatically as graph
Schema changes	Breaks integrations	Absorbed by semantic layer, loose coupling
Latency optimization	None built-in	RAG pre-computes embeddings, reduces lookup time
Security	Varies per service	Uniform Zero-Trust, RSA+AES-GCM end-to-end
Accessibility	Developers only	Any user via natural language
Failure handling	Manual retry logic	Automatic retry + exponential backoff via LangGraph

The NL Orchestrator essentially acts as an intelligent middleware layer — your existing backend APIs don't need to change. They register themselves in the MCP server registry, and from that point on, the orchestrator handles all the complexity of routing, auth, context, and execution. The paper's mention of RAG and JWT optimization directly addresses the latency concern that would otherwise make this middleware feel slow compared to direct API calls.

2. Architectural Foundation and Context Management

The Natural Language Orchestrator follows a three-layer architecture composed of a semantic interpretation layer to interpret incoming unstructured natural language input, a context management layer to represent the context, and an execution layer to execute the actions. The semantic interpretation layer uses transformer-based architectures with self-attention to represent contextual information at the token level and beyond. The transformer was the first neural network architecture to utilize an attention mechanism without a recurrence or convolution component.

Standardization of the context between distributed components and schemas for conversation state management, user choices, and execution history among multiple dialog sessions is defined by the Model Context Protocol. Context management in a conversational system is complex. Several techniques exist to maintain the history of dialogues and to ensure coherence over time.

LangGraph enables stateful orchestration by defining conversational flows and decision trees as directed graphs. LangGraph nodes correspond to possible system states, while edges between nodes define paths through the conversation graph based on user inputs and system outputs. This graph-driven model has many advantages, such as flexible workflow management. Decision-making can be expressed as graph-based state machines. Microservices architecture is a distributed computing model in which applications are built as a combination of independent services that communicate over standardized interfaces to solve issues in scalability, maintainability, and technology heterogeneity [4]. Regarding empirical evidence, microservices implementations show that container technology allows for increased density of deployments. Furthermore, orchestration frameworks allow service discovery, load balancing, and health monitoring across distributed clusters. Microservices architecture properties also include fault isolation, achieved through bulkhead patterns. These patterns prevent failures from cascading to other services, allowing the overall structure to remain available.

The orchestrator maintains the execution's state in graph form, which makes the orchestration process resilient to partial system failures. Automatic retries and exponential backoff ensure that a multi-step workflow consisting of dozens of single service invocations eventually reaches a consistent state.

Table 2: Three-Layer Architecture Overview [3, 4]

Architecture Layer	Primary Function	Input Type	Processing Approach	Output
Semantic Interpretation Layer	Natural language understanding	Unstructured natural language	Transformer-based architectures with self-attention	Contextual token representations
Context Management Layer	State and history maintenance	Parsed semantic information	Model Context Protocol standardization	Conversation context and execution history
Execution Layer	Action implementation	Contextualized commands	Graph-based workflow execution	API invocations and system actions

3. Dynamic API Mapping and Semantic Resolution

The orchestrator maintains a registry of internal MCP servers, each exposing specific functional capabilities through standardized interfaces that define request-response contracts, authentication requirements, and operational constraints. When processing a natural language command, the system performs semantic similarity analysis against this registry by computing vector representations of both the input command and registered API descriptions, then calculating cosine similarity scores to identify the most appropriate service endpoints. These representations enable the system to distinguish between semantically similar commands that require different API invocations based on subtle contextual cues, domain-specific terminology, or user-specific preferences learned from historical interaction patterns.

Dynamic mapping performance depends critically on the quality of semantic representations and the sophistication of matching algorithms that must handle ambiguity, context-dependency, and multi-intent scenarios where single commands reference multiple distinct operations. The system implements parameter extraction through named entity recognition and slot-filling techniques that identify relevant values within natural language inputs and map them to typed API parameters. Pre-training on large text corpora enables these models to achieve substantial performance improvements, with BERT obtaining 80.5% accuracy on the GLUE benchmark, representing a 7.7 percentage point absolute improvement over previous state-of-the-art approaches, while achieving a 93.2 F1 score on the SQuAD v1.1 question answering task with a 1.5 point absolute improvement [5]. The orchestrator implements specialized parsers for temporal, numerical, and categorical entities, with each parser applying domain-specific rules and statistical models trained on annotated datasets that cover diverse linguistic formulations and edge cases.

Service-oriented architectures benefit from standardized service contracts that establish consistent, reliable endpoints governed by explicit design principles, including service abstraction, loose coupling, and composability [6]. The system maintains metadata about service availability, performance characteristics, and interface specifications, enabling dynamic discovery and binding at runtime rather than requiring static configuration. This comprehensive approach to semantic resolution ensures that natural language commands are accurately translated into well-formed API invocations that respect all technical and business constraints while maintaining the intuitive user experience that motivates the natural language interface paradigm.

Table 3: BERT Model Architecture Parameters [5]

Model	Layers	Hidden Size (Dimensions)	Attention Heads	Total Parameters
BERT BASE	12	768	12	110 million
BERT LARGE	24	1024	16	340 million

4. Zero-Trust Security Architecture

The security model implements a dual-layer defense mechanism addressing both identity assurance and message integrity through cryptographic protocols that enforce authentication and authorization at every interaction point. OAuth 2.0 flows enforce user-level authentication by issuing time-limited access tokens that carry encoded claims about user identity, permissions, and session context. The protocol supports multiple grant types, including authorization code flow for web applications, client credentials for service-to-service communication, and refresh token mechanisms for maintaining long-lived sessions without repeated credential exchange [7]. Authorization codes must expire shortly after issuance to mitigate leak risks, with a maximum lifetime of 10 minutes recommended by the specification [7]. Token validation occurs at the API gateway, where cryptographic signature verification ensures token authenticity and prevents tampering, with the gateway checking token expiration timestamps, issuer claims, and audience restrictions before forwarding requests to backend services. JSON Web Tokens encapsulate claims in a compact, URL-safe format consisting of three base64-encoded segments representing the header, payload, and signature, with the signature computed using either symmetric HMAC algorithms or asymmetric RSA signatures, depending on deployment requirements and trust boundaries [7].

Asymmetric PKI encryption secures message confidentiality through public-private key pairs, where the public key encrypts data that only the corresponding private key can decrypt, establishing confidentiality without requiring prior shared secrets between communicating parties. The orchestrator employs RSA encryption with key lengths meeting security requirements, providing security levels substantially exceeding current cryptanalytic capabilities while balancing computational performance requirements. The probability of attackers guessing generated tokens and credentials must be less than or equal to 2^(-128) and should be less than or equal to 2^(-160) to ensure adequate protection against brute-force attacks [7]. Consumer public keys are registered during provisioning phases, creating a trust framework where the orchestrator maintains a verified repository of authorized consumer identities and their associated cryptographic credentials.

Response encryption implements hybrid cryptographic schemes that combine the key distribution advantages of asymmetric encryption with the computational efficiency of symmetric algorithms. The system generates ephemeral symmetric keys for each session, encrypts these keys using RSA public key encryption for secure distribution, then employs AES with 128-bit block sizes for bulk data encryption in Galois/Counter Mode [8]. GCM mode provides authenticated encryption that simultaneously ensures confidentiality and integrity, with authentication tag lengths of 128, 120, 112, 104, or 96 bits serving as security parameters, though certain applications may employ 64-bit or 32-bit tags under specific constraints [8]. The bit lengths of plaintext processed by GCM must not exceed 2^(39)-256 bits, while additional authenticated data is limited to 2^(64)-1 bits, and initialization vectors must be between 1 and 2^(64)-1 bits in length [8]. This architecture ensures end-to-end security for sensitive data traversing distributed systems while maintaining performance characteristics suitable for production workloads processing thousands of concurrent transactions.

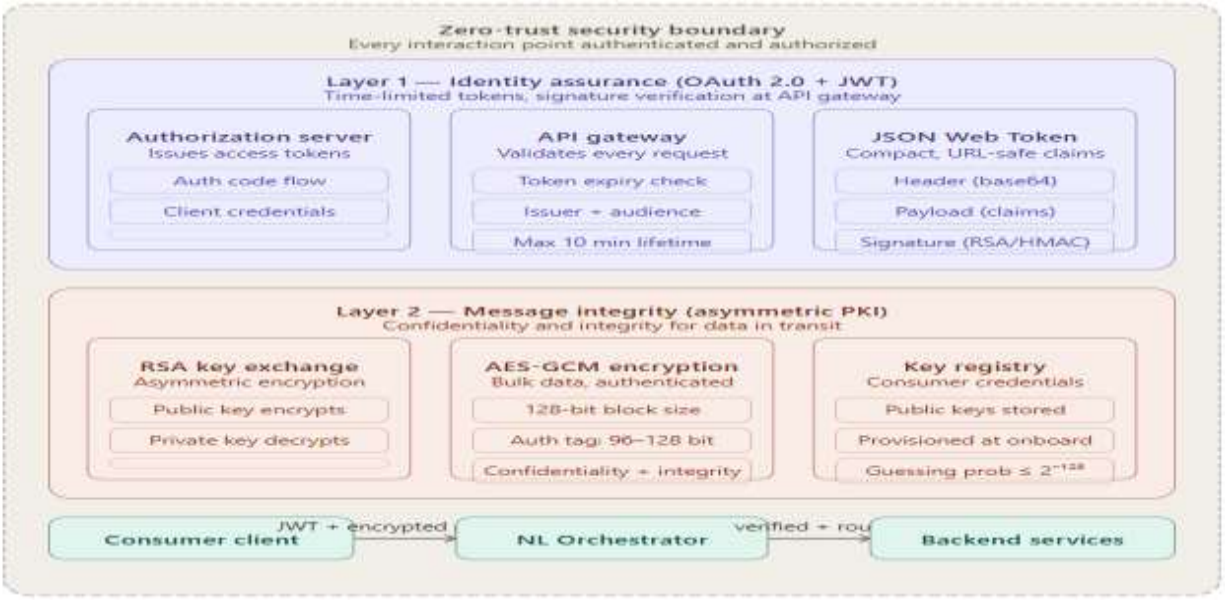

Fig. 2: Overall Zero-Trust security architecture

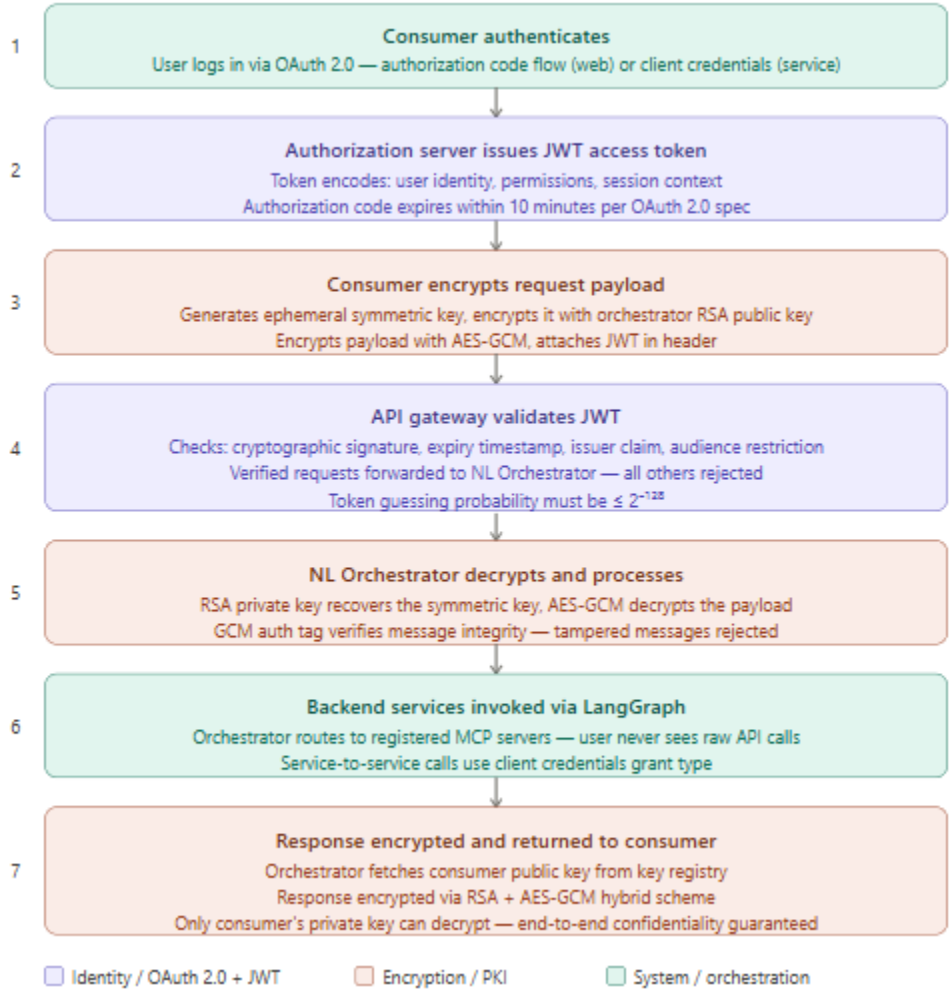

Fig. 3: A single API request from a consumer travels through both security layers

Table 4: OAuth 2.0 Grant Types and Use Cases [7, 8]

Grant Type	Primary Use Case	Authentication Flow	Session Management	Security Feature
Authorization Code Flow	Web applications	User authorization with code exchange	Token-based with refresh capability	Short-lived authorization codes
Client Credentials	Service-to-service communication	Direct credential exchange	Machine-to-machine authentication	No user context required
Refresh Token Mechanism	Long-lived sessions	Token renewal without re-authentication	Maintains session continuity	Avoids repeated credential exchange

5. Performance Characteristics and Scalability

Under load testing conditions simulating substantial concurrent user populations, the orchestrator demonstrates performance characteristics that satisfy production deployment requirements for enterprise-scale applications. Performance evaluation methodologies for distributed systems typically employ synthetic workload generators that simulate realistic usage patterns, including request arrival distributions, payload size variations, and query complexity profiles spanning simple retrieval operations to complex multi-service orchestrations requiring coordination across numerous backend systems. The system exhibits horizontal scalability properties where throughput increases proportionally with additional compute resources, a fundamental characteristic of well-designed cloud-native architectures that leverage containerization technologies for elastic resource provisioning. Containerized deployments enable fine-grained resource allocation with isolation guarantees, allowing multiple orchestrator instances to coexist on shared infrastructure while maintaining performance predictability and fault isolation. Container orchestration platforms manage deployment, scaling, and operations of application containers across clusters of hosts, providing primitives for distributed systems such as service discovery, load balancing, automated rollout and rollback, and self-healing through automated placement and replication [9].

Memory consumption patterns in stateful orchestration systems reflect the accumulated context from active conversation threads, cached API schemas, and runtime metadata structures required for dynamic routing decisions. Context persistence mechanisms leverage distributed caching architectures that replicate frequently accessed data across multiple nodes to reduce latency and improve fault tolerance. Consistent hashing algorithms distribute keys across cache nodes using hash functions that minimize redistribution when nodes join or leave the cluster, with virtual node techniques improving load distribution uniformity across heterogeneous hardware configurations. Distributed B-tree implementations demonstrate linear scalability characteristics across varying system sizes, with performance scaling almost linearly to hundreds of machines for read and update workloads [10]. Cache hit rates represent critical performance metrics indicating the proportion of requests satisfied directly from cache without requiring expensive database queries or external service invocations. Research examining caching strategies in web-scale systems demonstrates that improvements in hit rates yield substantial reductions in backend load, with higher cache effectiveness potentially reducing database query volumes significantly and improving end-to-end response latencies.

Conclusion

The Natural Language Orchestrator fundamentally reimagines API consumption by transforming technical integration challenges into intuitive conversational experiences. Through the combination of transformer-based semantic understanding, Model Context Protocol standardization for context exchange, and LangGraph-driven stateful orchestration, the framework successfully bridges the gap between human intent and machine execution. The architecture demonstrates that democratizing system access through natural language interfaces can coexist with enterprise-grade security requirements when implemented through comprehensive Zero-Trust principles. The dual-layer defense mechanism combining OAuth 2.0 token-based authentication with asymmetric Public Key Infrastructure encryption ensures that every interaction maintains authenticated provenance while protecting sensitive data through cryptographic sealing. Semantic resolution capabilities enable accurate mapping of unstructured commands to appropriate service endpoints by leveraging dense vector embeddings and bidirectional transformer architectures that capture contextual nuances beyond simple keyword matching. Performance characteristics confirm production viability through horizontal scalability properties, efficient distributed caching mechanisms, and context management policies that balance memory efficiency against conversation quality requirements. The graph-based orchestration model provides resilient operation through explicit state machines that support automatic retry logic and fault isolation patterns, ensuring eventual consistency across complex multi-step workflows. This architectural paradigm establishes a foundation for human-centric system interaction that maintains operational rigor while eliminating the technical barriers traditionally associated with API integration, paving the way for more accessible and intuitive enterprise software ecosystems where users can accomplish sophisticated tasks through natural conversation rather than specialized technical knowledge.

References

[1] Maria Gabriela Juarez Juarez et al., "Semantic and modular orchestration of AI-driven digital twins for industrial interoperability and optimization," Journal of Industrial Information Integration, Volume 48, 2025. Available: https://www.sciencedirect.com/science/article/pii/S2452414X25001827
[2] Murilo Góes de Almeida and Edna Dias Canedo, "Authentication and Authorization in Microservices Architecture: A Systematic Literature Review," Applied Sciences, 2022. DOI: 10.3390/app12063023. Available: https://www.mdpi.com/2076-3417/12/6/3023
[3] Ashish Vaswani et al., "Attention is all you need," arXiv:1706.03762v7, 2023. Available: https://arxiv.org/pdf/1706.03762
[4] Maha Driss et al., "Microservices in IoT Security: Current Solutions, Research Challenges, and Future Directions," Procedia Computer Science, Volume 192, 2021. Available: https://www.sciencedirect.com/science/article/pii/S1877050921017440
[5] Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Proceedings of NAACL-HLT, 2019. Available: https://aclanthology.org/N19-1423.pdf
[6] Thomas Erl, "SOA: Principles of Service Design," 2008. Available: https://ptgmedia.pearsoncmg.com/images/9780132344821/samplepages/9780132344821.pdf
[7] D. Hardt, "The OAuth 2.0 Authorization Framework," Internet Engineering Task Force, RFC 6749, Oct. 2012. Available:https://datatracker.ietf.org/doc/html/rfc6749
[8] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, "NIST Special Publication 800-38D, 2007. DOI: 10.6028/NIST.SP.800-38D. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf
[9] BRENDAN BURNS et al., "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," ACM Queue, 2016. DOI: 10.1145/2898442.2898444. Available: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf
[10] Marcos K. Aguilera et al., "A practical scalable distributed B-tree," Proceedings of the VLDB Endowment, Volume 1, Issue 1, 2008. Available: https://dl.acm.org/doi/10.14778/1453856.1453922