International Journal of Leading Research Publication (IJLRP)

E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

Reducing Cognitive Load In Complex Hybrid Systems: The Role of Microservices Architecture In Simplifying Developer Experience

Arun K Gangula

arunkgangula@gmail.com

Abstract:
The current software systems operate across on-premises infrastructure and private and public cloud environments while utilizing artificial intelligence and massive data analytics capabilities. The complex nature of modern systems creates an unbearable mental workload for developers, which obstructs their ability to innovate and work efficiently. The analysis of traditional monolithic architectures through Cognitive Load Theory reveals how their large, tightly coupled codebases create excessive mental work for developers. Strategic implementation of microservices functions as a cognitive management system. Microservices become more effective when designed as small, independent services that align with business domains, following Domain-Driven Design principles. This approach decreases unnecessary mental workload and aligns system boundaries with team mental capacity, as the Mirroring Hypothesis explains. The research investigates how platform engineering simplifies hybrid environment complexities and how the Strangler Fig Pattern enables cognitive-friendly system migration processes. Building sustainable, evolvable systems requires teams to prioritize cognitive load as their primary architectural concern.

Index Terms: Cognitive Load, Microservices, Monolithic Architecture, Developer Experience, Complex Hybrid Systems, Domain-Driven Design, Software Architecture, Cognitive Complexity.

I. INTRODUCTION

Modern software systems now operate on hybrid infrastructure models, which combine on-premises data centers with private clouds and multiple public cloud services, including AWS and Azure. The diverse architectural approach provides flexibility, together with resilience and compliance capabilities, but creates substantial complexity. The growing complexity of infrastructure meets an increasing demand for functional capabilities, such as AI/ML pipelines, IoT device real-time data streams, and large-scale analytics are now integral to enterprise applications. Developers need to handle both code and an elaborate system of APIs, together with services and deployment environments.

The increasing system complexity creates a significant mental challenge for software teams. The typical developer needs to understand distributed behaviors, performance bottlenecks, security implications, and business logic simultaneously. System growth towards interdependence and opacity makes human errors more likely while technical debt builds up and innovation slows. The main bottleneck today extends beyond tooling and infrastructure because human workers face cognitive capacity restrictions.

Cognitive Load Theory (CLT) represents a model that originates from educational psychology to describe this challenge. Human working memory functions to handle only a small number of distinct elements at any given time [1]. System complexity exceeding human working memory limits results in cognitive overload, which produces slower development, higher defect rates, and long-term maintainability issues [2]. Architectural decisions fail to recognize human cognitive limitations as a fundamental design constraint. The design of software continues to prioritize technical optimization above human understanding.

The strategic implementation of microservices architectural style functions as a cognitive load management system according to this research. The combination of Domain-Driven Design decomposition with platform engineering practices enables microservices to minimize unnecessary cognitive work while creating software boundaries that match team mental models. The approach creates sustainable systems that evolve sustainably while maintaining respect for human cognitive boundaries.

II. THEORETICAL FOUNDATION: COGNITIVE LOAD IN SOFTWARE ENGINEERING

To understand how software architecture affects developer experience, we ground the discussion in Cognitive Load Theory (CLT), which models human cognitive constraints in learning and problem-solving.

A. Human Cognitive Architecture: Memory Systems and Schema

The CLT model consists of two memory systems, which include working memory and long-term memory. Working memory functions as a limited system that maintains and transforms consciously focused information through a capacity of four to seven new "chunks" for a short duration of several seconds unless active processing occurs. The system performs poorly when overloaded, which explains why people struggle with mental calculations of big numbers and handling numerous unfamiliar concepts simultaneously. The vast capacity of long-term memory allows information retrieval to bypass the limited working memory constraints. Experts use this method by recognizing patterns as integrated units because their extensive knowledge in long-term memory transforms board configurations into familiar structures instead of individual pieces.

The cognitive construct of schemas combines multiple elements into single usable units through practice to enable automation, which frees working memory for new challenges. Expert drivers perform driving tasks automatically because their skills have transformed into a single driving schema that operates without conscious mental processing. The process of becoming an expert software developer requires developers to create and automate schemas, which enable them to reason about systems with minimal mental effort.

B. A Taxonomy of Cognitive Load

The CLT model identifies three separate load types that add up to form total cognitive demand yet exceed capacity to create overload. [1]

1. **Intrinsic Cognitive Load (ICL):** ICL refers to the built-in complexity of tasks that require learners to maintain multiple interacting elements at once. The basic control flow of an "if" statement creates low ICL, yet recursion demands high ICL because learners need to manage base case and recursive step and state evolution. The complexity level of ICL depends on learner expertise because experts can transform complex material into streamlined content using appropriate schemas. The core of ICL remains linked to the problem itself because presentation methods cannot eliminate its fundamental nature. [3]
2. **Extraneous Cognitive Load (ECL):** The unnecessary load caused by poor representation or structure of information, “bad” load that does not aid learning or problem-solving. The examples of ECL include confusing diagrams, split attention from scattered sources, and redundant explanations. The main objective of both effective instructional design and software architecture is to reduce ECL, thereby freeing up working memory space for schema construction. [3] [4]
3. **Germane Cognitive Load (GCL):** The productive load devoted to learning and schema acquisition— “good” effort such as abstracting principles, mentally rehearsing solutions, or organizing knowledge. The design process should minimize ECL to free up mental capacity, which can then be redirected toward GCL for developing deeper knowledge and expertise. [1]

C. Application for Software Development

The three cognitive load types directly relate to developers' daily work activities because the human mind functions as their main engineering limitation.

- Intrinsic Load in Software: The intrinsic load in software represents unavoidable domain or algorithm complexity, which includes business rules in financial calculations and state management in workflows and distributed consensus logic. The developer needs to understand these fundamental interacting elements to solve the problem.
- Extraneous Load in Software: The software generates excessive cognitive load due to inadequate architectural and organizational decisions, resulting in ECL. Developers waste cognitive resources when they try to understand complex "spaghetti" code and search through large monolithic systems for relevant functions and handle complicated deployment pipelines and recurring merge conflicts that result from simultaneous work. The accidental complexity of the system requires developers to use their working memory to understand its structure instead of focusing on the core problem. [2] Poor architecture is a primary driver of ECL.
- Germane Load in Software: The process of developing long-term, reusable understanding generates GCL, which is achieved through activities such as designing clean APIs, refactoring for clarity, implementing Domain-Driven Design to align code with business models, and learning and applying patterns for generalized solutions. The activities develop automated schemas that create more efficient and robust development processes for the future. [2]

Traditional monolithic architectures expand in size, leading to increased ECL as developers dedicate excessive working memory to understanding accidental complexity and navigating the complex system structure. The reduction of cognitive capacity prevents developers from handling intrinsic problems and performing deep, Germane work, which results in high-quality software.

The relationships between cognitive load types and software development manifestations are made explicit through Table I, which shows how architectural decisions either increase unnecessary load or support schema-building work.

TABLE I- TYPES OF COGNITIVE LOAD AND THEIR MANIFESTATIONS IN SOFTWARE DEVELOPMENT

Cognitive Load Type	Definition in Software Context	Concrete Examples
Intrinsic	The inherent complexity of the specific problem is being solved.	- Understanding the logic of a complex financial calculation. - Implementing a novel sorting algorithm. - Grasping the state transitions into a finite state machine.
Extraneous	Mental effort wasted on understanding the system's structure, tooling, or irrelevant information.	- Deciphering tangled dependencies in "spaghetti code". - Navigating a massive, poorly organized monolithic codebase to find a single function. - Struggling with a convoluted manual deployment process. - Resolving merger conflicts caused by many teams working on the same codebase.
Germane	Effort dedicated to building robust mental	- Designing a clean, extensible API. - Refactoring a complex method into smaller,

	models (schemas) of the problem and solution space.	understandable functions. - Engaging in Domain-Driven Design to model the business domain accurately. - Learning and applying a new design pattern.

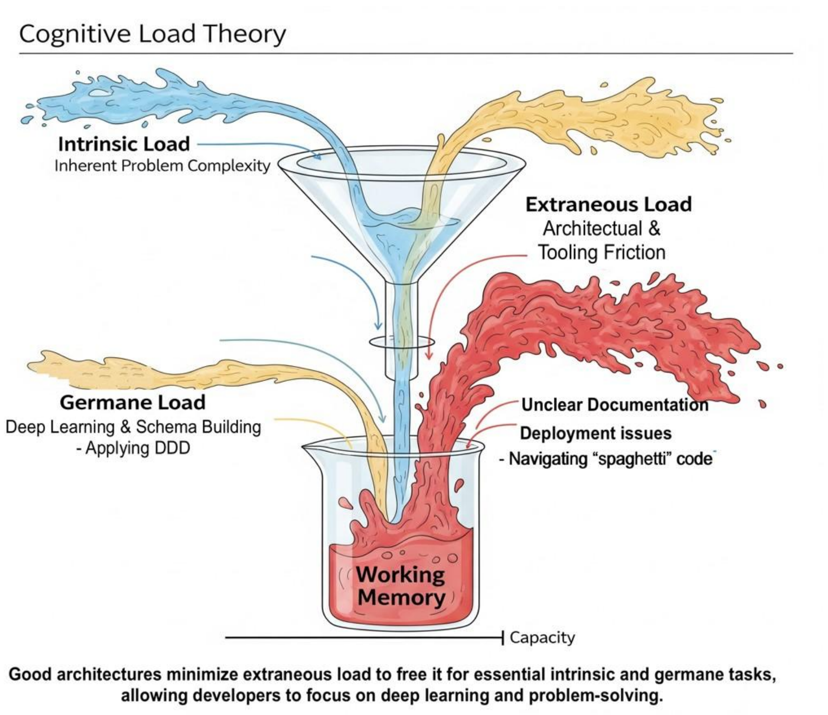
Cognitive Load Theory
Intrinsic Load
Inherent Problem Complexity
Extraneous Load
Architectual & Tooling Friction
Germane Load
Deep Learning & Schema Building
- Applying DDD
Unclear Documentation
Deployment issues
- Navigating "spaghetti" code
Working Memory
Capacity
Good architectures minimize extraneous load to free it for essential intrinsic and germane tasks, allowing developers to focus on deep learning and problem-solving.

Fig. 1. A conceptual model illustrates the impact of different cognitive loads on a developer's limited working memory.

III. THE COGNITIVE BURDEN OF MONOLITHIC ARCHITECTURES IN HYBRID SYSTEMS

Monolithic architecture served as the standard software design method until distributed systems gained popularity. Monolithic systems work well for initial applications but create increasing mental complexity when systems grow in hybrid environments.

A. The Nature of the Monolith: A Tightly Coupled System

The monolithic architecture integrates user interface elements, business logic, and data access functionality into a single, unified system. [5] The initial development benefits from this simplicity because it allows developers to work with one codebase and artifact, which streamlines building, testing, and deployment processes. The monolith transforms into a complex and difficult-to-modify system when applications grow, and teams increase in size. The architectural style fails to establish distinct modular divisions. The process of adding new features or implementing fixes creates additional dependencies that build up over time. The initial straightforward structure transforms into an unclear, tightly coupled codebase that needs a complete understanding of the entire system to modify any part safely.

B. Scaling Pains: Complexity and Dependency Hell

The cognitive load that monoliths create grows exponentially with their size. The absence of enforced boundaries leads to interdependent code structures, which developers commonly call "spaghetti code." [6] Developers must mentally analyze extensive parts of the system to predict any unintended effects

when making changes. The system's intrinsic cognitive load increases because developers need to process more elements, and they must spend additional effort to understand the structure instead of solving problems. [5]
The dependency hell problem arises because all modules operate from the same environment, so updating one dependency for a feature can cause unintended effects on system components that are not related. The system's fragility necessitates that developers remain vigilant while evaluating each modification for the potential to cause widespread system failures, thereby adding to their mental burden.
Cognitive Complexity serves as a measurable factor that indicates the mental work required to comprehend code structures and flows. [7] The dense network of implicit connections between monolithic components creates complexity that extends beyond individual components.
The growth of complexity leads developers to choose quick fixes rather than implementing clean and well-structured solutions. Under high pressure, developers tend to select simpler solutions that create technical debt because these options require less mental effort. The complexity of code creates a vicious cycle where disorder in the codebase makes improvement more difficult, thus leading developers to select additional short-term solutions. The codebase becomes unmanageable over time because developers make successive logical choices within an uncontrollable system rather than through a single bad decision.

C. Impact on Developer Experience in Hybrid Contexts

The cognitive burden of monolithic systems significantly degrades developer experience—especially in the context of
hybrid deployments, where operational complexity is magnified.
Onboarding has become a significant hurdle. New developers must navigate an extensive and poorly organized codebase to make substantial contributions. The high learning curve creates obstacles for team growth and prevents employees from moving between roles. The combination of inadequate documentation and hidden dependencies necessitates that experienced developers repeatedly learn about different parts of the system.
Monoliths also discourage experimentation and refactoring. The high risk of introducing regressions exists because of unpredictable interconnections, which creates a culture of fear when making code changes. [8] The fear of breaking fragile components leads developers to avoid them, allowing technical debt to accumulate and innovation to stall.
The deployment of monolithic systems in hybrid environments that span on-premises, private, and public clouds introduces additional deployment complexity. The infrastructure optimization of monoliths requires uniform systems. The deployment of a single artifact across different environments in hybrid contexts results in resource allocation inefficiencies, complicated configuration management, and higher operational costs. [9] These challenges create unnecessary mental strain, which diverts developer' focus from their core business responsibilities.
The initial convenience of monolithic architectures results in overwhelming cognitive demands that negatively affect maintainability, adaptability, and employee morale, especially when organizations shift toward hybrid and cloud-native ecosystems.

IV. MICROSERVICES ARCHITECTURE AS COGNITIVE MANAGEMENT STRATEGY

Monolithic systems face two main challenges: cognitive and technical scaling, which have led to the adoption of microservices as a leading alternative architectural style. The correct implementation of this socio-technical strategy reduces the development team's cognitive load directly.

A. Decomposition as a Strategy for Reducing Cognitive Load

Microservice architecture divides an application into multiple small, independent services that can

operate independently for deployment. The individual services operate independently as separate processes while using lightweight communication methods through network APIs to exchange data.
The process of decomposition actively fights against cognitive load. The architecture reduces the developer's working memory information requirements through its decomposition of monolithic problems into smaller, independent microservices. [2] Developers working on the "customer profile" service do not require knowledge of the "order processing" or "inventory management" system operations. The developer needs to understand only the customer profile service together with its public APIs.
The method significantly decreases both internal and external mental workload. The problem scope becomes smaller, which reduces intrinsic load because it contains fewer interacting elements. The developer experiences reduced extraneous load because the system complexity remains hidden from view while working with explicit API calls instead of internal dependencies. [10] The development lifecycle becomes simpler because each service operates independently for understanding, development, testing, and deployment purposes.

B. Aligning Architecture with Teams: The Mirroring Hypothesis

The advantages of microservices extend beyond programming code, as they reshape how organizations are structured. According to Conway's Law, organizations must create system designs that duplicate their existing communication structures. The Mirroring Hypothesis builds upon this concept by showing that technical product dependencies mirror the organizational connections, which include communication patterns and team structures of the development team. The natural approach to problem-solving through mirroring technical and organizational structures helps preserve limited cognitive resources. Monolithic architectures typically mirror the organizational structure of large organizations, which operate through functional silos (e.g., UI team, backend team, database team). The process of communication and coordination between these silos generates significant overhead and places excessive cognitive demands on teams. Microservices architecture enables organizations to structure themselves into small, independent teams that work across different functions. [11] Each team maintains complete responsibility for one or multiple services throughout their development and deployment, and operational
phases ("you build it, you run it"). [2]
The connection between team structure and service boundaries functions as a strong cognitive tool. Developers now need to focus their thinking on both smaller codebases and smaller social groups. The communication pathways become easier to navigate while large-scale coordination needs decrease substantially. [12] The team maintains complete technical freedom to select appropriate tools for their needs and to make service development choices as long as they keep their public API contracts. The independence granted to teams increases productivity while decreasing mental strain caused by organizational obstacles. [11]

C. Domain-Driven Design: Creating Cognitively Coherent Boundaries

The process of decomposition requires careful planning. The method of dividing a monolith into random small pieces results in a "distributed monolith," which maintains the original system's tight coupling and introduces distributed network complexity. The design of successful microservices requires establishing boundaries that maintain logical consistency and stability while being meaningful to the system.
Domain-Driven Design (DDD) delivers the necessary strategic tools to achieve this goal. The software development method of Domain-Driven Design focuses on creating software models that represent the business domain they support. Strategic DDD includes Bounded Context as its core concept, which establishes a domain model boundary where language, concepts, and rules maintain consistency and clarity. [13] The definition of "Customer" differs between "Marketing" and "Support" contexts because each context requires unique attributes and behaviors. The Bounded Context establishes clear distinctions between different concepts.

The most effective microservice architectures establish their service boundaries based on Bounded Contexts. [13] Each microservice contains a unified set of business capabilities that maintain coherence. Service boundaries derive from logical business domain divisions instead of technical layers such as "UI service." The method produces services that developers can understand better because their defined scope and purpose stem from real-world problem solutions. The alignment between services and Bounded Contexts reduces the mental effort needed to comprehend service functions and their position in the system structure. [14]
The cognitive benefits of this architectural change can be summarized through Table II, which compares monolithic and microservice architectures based on developer cognitive load factors.

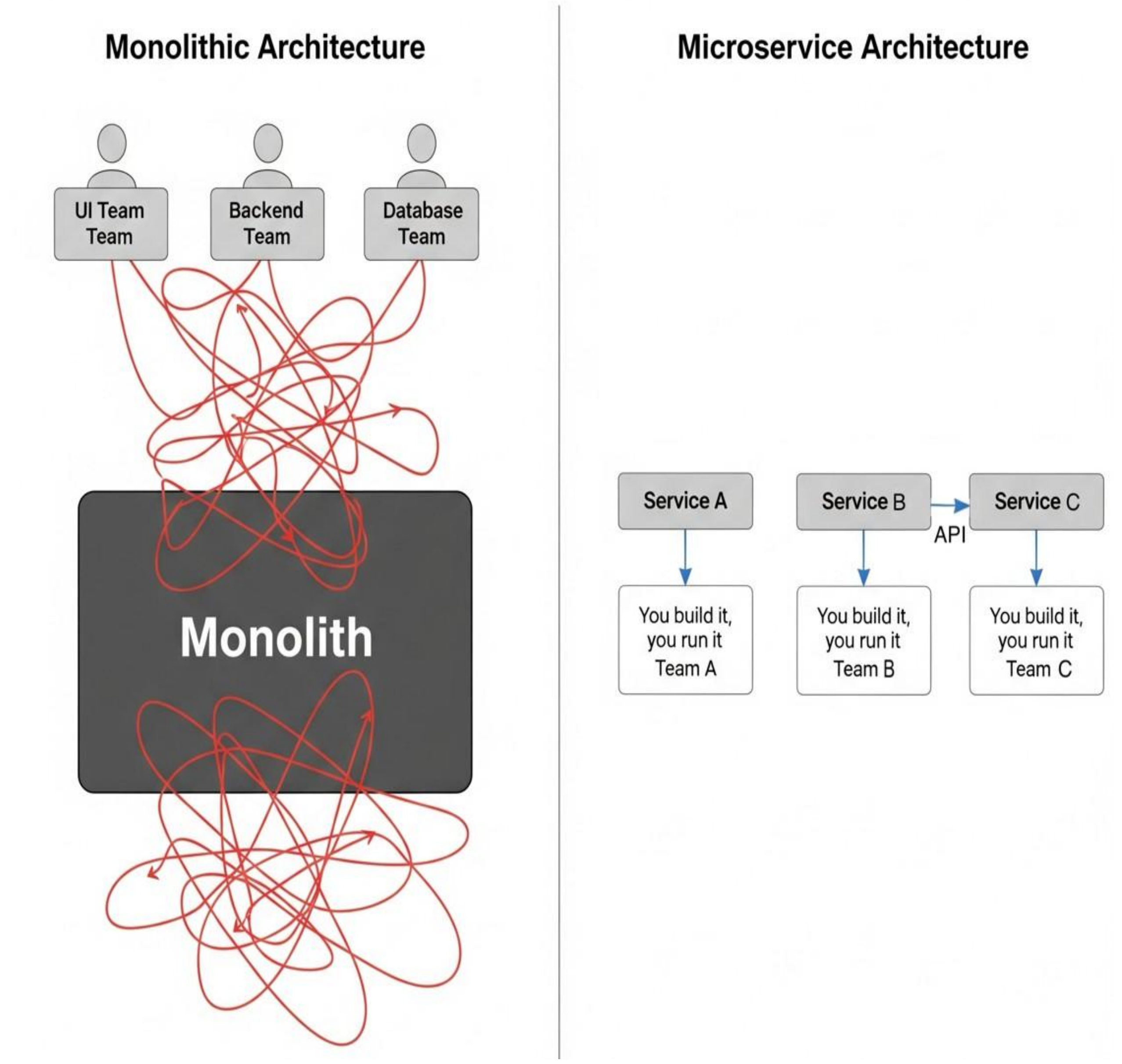
Monolithic Architecture
UI Team Team
Backend Team
Database Team
Monolith
Microservice Architecture
Service A
Service B
Service C
API
You build it, you run it Team A
You build it, you run it Team B
You build it, you run it Team C

Fig. 2. A comparison of team communication structures in monolithic versus microservice architectures.

TABLE II- COMPARATIVE ANALYSIS OF MONOLITHIC VS. MICROSERVICE ARCHITECTURES ON KEY COGNITIVE FACTORS

Cognitive Factor	Monolithic Architecture	Microservice Architecture
Scope of Context	The developer must understand or navigate the entire system. High intrinsic & extraneous load.	The developer focuses on a single service/bounded context. Low intrinsic & extraneous load.
Dependency Management	Tangled, implicit dependencies across the codebase ("spaghetti code"). High extraneous load.	Explicit dependencies via well-defined APIs. Dependencies are managed per service. Lower extraneous load.
Team Ownership & Autonomy	Often owned by large, siloed teams, leading to coordination overhead and diffuse responsibility.	Owned by small, autonomous teams. Clear ownership reduces the cognitive load of coordination.
Onboarding Time	Long. New developers face an incomprehensible codebase.	Short. New developers only need to learn one service to become productive.
Testing Complexity	A small change can require testing the entire application. High cognitive load to assess impact.	Services can be tested in isolation. The scope of testing is limited to the service and its contracts.
Technology Evolution	The technology stack is locked in. Changes require massive effort ("big bang" rewrite).	Polyglot persistence/programming is possible. New technology can be adopted one service at a time.

V. NAVIGATING THE TRANSITION AND HYBRID REALITIES

The transition to microservices brings cognitive advantages over monolithic systems yet this process remains complicated and generates additional difficulties. A successful migration needs organizations to identify new cognitive loads and implement strategic design and organizational support for their effective management.

A. Microservices: A New Set of Trade-offs

The transition from monolithic systems to microservice systems replaces monolithic system complexity with distributed system complexity. The transition without proper management leads to the creation of new cognitive burdens that replace previous ones. [15]

Key challenges include:

- **Network Latency and Reliability:** Service communication transitions from in-process calls to unreliable networks, thus requiring fault-tolerant designs for network latency and reliability.

- **Fault Isolation:** A single service failure should not trigger additional outages throughout the system. The implementation of circuit breakers together with bulkheads serves this purpose.
- **Service Discovery:** Dynamic environments demand mechanisms to locate services at runtime.
- **Observability:** The combination of distributed tracing, centralized logging, and metrics tools such as Prometheus becomes necessary for debugging multiple services. [16]
- **Data Consistency:** The use of independent databases in microservices creates data consistency challenges that require sagas or event-driven design patterns.

Every team faces overwhelming challenges when it must handle these issues independently.

B. Platform Engineering: Abstraction for Cognitive Relief

Organizations with high performance levels handle distributed complexity through platform engineering. A Platform Team dedicated to infrastructure management provides self-service reusable infrastructure instead of forcing developers to handle infrastructure concerns.

The platform delivers standardized solutions through its features, which include:

- **Deployment:** Container orchestration (e.g., Kubernetes) to control service health and scaling operations.
- **Observability:** pre-integrated logging, metrics, and tracing systems for observability.
- **Network & Security:** The service mesh provides discovery capabilities along with encryption features, retry mechanisms, and fault tolerance functions for networking and security needs.
- **CI/CD Pipelines:** The platform includes automated CI/CD pipelines, which handle build, test, and release operations.

The use of infrastructure as a service allows developers to escape operational tasks so they can concentrate on developing business logic. [2] The abstraction layer enables microservices scaling without causing the team to be overwhelmed.

C. The Strangler Fig Pattern: Managing Migration Load

Organizations typically begin their journey with existing monolithic systems instead of building new microservices from scratch. A complete rewrite poses significant risks while creating substantial mental and operational challenges. [17]

The Strangler Fig Pattern represents an effective incremental migration strategy for organizations to adopt. [18] [19] The pattern functions through a tree-like process that replaces its host by growing around it. The pattern enables step-by-step transformation:

1) A Façade (e.g., API gateway) serves as an entry point to redirect requests toward the monolith.
2) Extract a function (mainly edge functionality) from the monolith as the first step.
3) The new microservice needs to be developed and deployed as a separate entity.
4) The façade should redirect traffic for that functionality to the new service.
5) The process should continue for other components until microservice coverage reaches a certain level.
6) The Monolith becomes obsolete after all essential functions are migrated to new services.

The strategy enables organizations to manage cognitive strain by delivering business value through continuous delivery and step-by-step adoption. [20] Teams can learn new tools and practices at a gradual pace, which decreases both risk levels and employee burnout.

D. Hybrid Architecture: The Practical End State

Most enterprises adopt hybrid architecture as a result of modernization instead of implementing pure microservice ecosystems. The monolith maintains stable legacy components, which have low modernization value, while new functions that evolve often get refactored into microservices. [21] The hybrid approach represents the most cost-effective and cognitively manageable solution. Organizations can focus their efforts on modernization areas that yield the most significant impact, while preserving

well-functioning legacy code from
unnecessary rewriting.

VI. CONCLUSION

The increasing complexity of software systems, particularly in hybrid cloud environments, has made cognitive capacity the main limiting factor for developers in engineering. This paper presents software architecture as a mental tool that should handle developer cognitive load according to Cognitive Load Theory (CLT).

A. Architecture as Cognitive Prosthesis

The combination of microservices with Domain-Driven Design (DDD) and platform engineering creates a cognitive prosthesis system. The system decreases unnecessary mental workload through its design, which separates responsibilities and defines boundaries, and reduces superfluous dependencies between components. Monolithic architectures create complex logic connections that increase the cognitive workload for basic modifications, thus using up valuable time that should be focused on business solution development. [2]

The smaller cognitive scopes and domain-aligned services of microservice systems allow teams to work independently while maintaining high productivity, which leads to better developer sustainability.

B. Beyond Monoliths vs Microservices

This is not a binary choice. The main objective is to find the best approach for managing cognitive complexity instead of enforcing strict microservices adoption. Architecture needs to be designed to accommodate the mental abilities of its maintenance teams through modular monoliths, services, and hybrid patterns. [2]

C. Future Directions

Key areas for future research include:

- The development of empirical metrics for engineering cognitive load measurement remains a research priority.
- Research should examine how different team structures, including platform teams, affect cognitive function over time.
- The investigation should determine how AI tools affect software development cognitive load by either reducing or redistributing it.

The focus on human cognition during software design enables developers to build systems that scale while being resilient, maintainable, and humane.

REFERENCES:

[1] D. Wiliam *et al.*, “Cognitive load theory: Research that teachers really need to understand,” Sep. 2017. [Online]. Available: https://education.nsw.gov.au/content/dam/main-education/about-us/educational-data/cese/2017-cognitive-load-theory.pdf

[2] M. Skelton and M. Pais, “Monoliths vs Microservices is Missing the Point—Start with Team Cognitive Load - IT Revolution,” *IT Revolution*, Sep. 17, 2019. https://itrevolution.com/articles/team-cognitive-load-team-topologies/

[3] J. J. G. Van Merriënboer and J. Sweller, “Cognitive Load Theory and complex Learning: recent developments and future directions,” *Educational Psychology Review*, vol. 17, no. 2, pp. 147–177, May 2005, doi: 10.1007/s10648-005-3951-0.

[4] T. De Jong, “Cognitive load theory, educational research, and instructional design: some food for thought,” *Instructional Science*, vol. 38, no. 2, pp. 105–134, Aug. 2009, doi: 10.1007/s11251-009-9110-0.

[5] R. Chen, S. Li, and Z. Li, “From Monolith to Microservices: A Dataflow-Driven approach,” in *IEEE Xplore*, Nanjing, China. [Online]. Available: https://doi.org/10.1109/apsec.2017.53

[6] P. Królik, “MasterBorn | How much will Spaghetti Code really cost you?,” Jul. 08, 2022. https://www.masterborn.com/blog/how-much-will-spaghetti-code-really-cost-you

[7] J. Shao and Y. Wang, *A new measure of software complexity based on cognitive weights*, vol. 28. 2003. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=197731f69911b58dafafe2c01285a7fbd50c3cd4

[8] P. Golofit, “Technical Debt - the silent villain of web development,” *Accesto Blog*, Oct. 06, 2020. https://accesto.com/blog/technical-debt-the-silent-villain-of-web-development/

[9] IBM Cloud Education Team, “Design hybrid cloud architecture,” *IBM*, Oct. 22, 2022. https://www.ibm.com/think/topics/design-hybrid-cloud-architecture#:~:text=A%20hybrid%20cloud%20architecture%20brings,a%20single%20managed%20IT%20infrastructure.

[10] M. Skelton and M. Pais, “Minimize team cognitive load to increase flow - IT Revolution,” *IT Revolution*, Jan. 21, 2021. https://itrevolution.com/articles/minimize-cognitive-load-of-teams/

[11] O. Ezzheva, “7 Tangible benefits of microservices for your next web application project,” *TechSpective*, Sep. 18, 2018. https://techspective.net/2018/09/18/7-tangible-benefits-of-microservices-for-your-next-web-application-project/

[12] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges When Moving from Monolith to Microservice Architecture,” in *Lecture notes in computer science*, 2018, pp. 32–47. doi: 10.1007/978-3-319-74433-9_3.

[13] D. Baltor, “How to choose Microservice’s boundaries? | Bits and pieces,” *Medium*, Feb. 13, 2023. [Online]. Available: https://blog.bitsrc.io/how-to-choose-microservices-boundaries-5c68b0b1af24

[14] “Domain-Driven design starter modelling process,” *Ddd-starter-modelling-process*. https://ddd-crew.github.io/ddd-starter-modelling-process/

[15] M. Freshwaters, “Why Companies are Moving their Applications to Modular Architecture,” *ippon*, Jan. 23, 2023. https://blog.ippon.tech/why-companies-are-moving-their-applications-to-modular-architecture

[16] C. Harris, “Microservices vs. monolithic architecture | Atlassian,” *Atlassian*. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[17] Z. Dehghani, “How to break a Monolith into Microservices,” *martinfowler.com*, Apr. 24, 2018. https://martinfowler.com/articles/break-monolith-into-microservices.html

[18] B. Quillin, “The Strangler Architecture Pattern for Modernization,” *vFunction*, Oct. 20, 2022. https://vfunction.com/blog/strangler-architecture-pattern-for-modernization/

[19] Witkowska, “The Strangler Fig pattern,” *Tyk API Management*, Apr. 26, 2022. https://tyk.io/blog/res-strangler-fig-pattern/

[20] Quillin, “Strangler Fig Pattern to Move from Mono to Microservices,” *vFunction*, Jan. 03, 2023. https://vfunction.com/blog/fig-pattern-the-solution-to-your-mono-to-microservices-modernization/

[21] M. Milić and D. Makajić-Nikolić, “Development of a Quality-Based Model for Software Architecture Optimization: A case study of monolith and microservice architectures,” *Symmetry*, vol. 14, no. 9, p. 1824, Sep. 2022, doi: 10.3390/sym14091824.