

Software Patterns for the 21st Century: From Monolithic Designs to AI-Driven Architectures

Rahmon Ariyo Badru

Software Engineering Program, Department of Computer and Information Engineering, Faculty of Natural and Applied Science, Lead City University, Ibadan, Nigeria.

Akeem Olamide Arikeuyo

Department of Agricultural and Bio-environmental Engineering, Federal College of Agriculture, Ibadan, Nigeria

Idowu Olugbenga Adewumi

`idowu_adewumi@rotarydistrict9126.org`

Software Engineering Program, Department of Computer and Information Engineering, Faculty of Natural and Applied Science, Lead City University, Ibadan, Nigeria. <https://orcid.org/0000-0002-7005-3306>

Systematic Review

Keywords: Software Design Patterns, Software Architecture, Distributed Systems, Pattern Evolution, Qualitative Review, Software Engineering

Posted Date: August 27th, 2025

DOI: <https://doi.org/10.21203/rs.3.rs-7437898/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License. [Read Full License](#)

Additional Declarations: The authors declare no competing interests.

Abstract

Software design patterns offer modular, maintainable, and scalable approaches for contemporary software engineering. This research offers a qualitative systematic review performed following PRISMA guidelines. Out of an initial search of 226 publications, 84 articles underwent full-text screening, leading to the inclusion of 42 peer-reviewed studies (published from 2013 to 2024) in the final synthesis. The evaluation was directed by four research inquiries focusing on categorization, contextual application, compromises, and the development of design patterns. Results showed that architectural patterns (MVC, layered architecture, microservices) were the most frequently referenced, with 41 mentions, followed by cloud-native and microservice-related patterns (37 mentions), creational patterns (31 mentions), structural patterns (28 mentions), and event-driven/reactive patterns (25 mentions). Concurrency patterns were mentioned the least (14 occurrences), indicating their growing incorporation into frameworks. In various sectors, web, mobile, enterprise, and cloud applications displayed unique adoption patterns, with MVC leading in web environments, MVVM common in mobile platforms, and microservices and CQRS fundamental to cloud architectures. Analysis of trade-offs highlighted recurring conflicts like simplicity versus testability (Singleton), flexibility versus readability (Factory Method), and independence versus integration complexity (Microservices). Timeline mapping illustrated the transition from monolithic models to distributed, event-driven, and serverless architectures from 2010 to 2024, showing a distinct increase in cloud-native adoption after 2018. The study ends with suggestions for context-sensitive pattern choice and emphasizes the importance of further investigation into AI-automated patterns, edge-native orchestration, and quantum-ready software designs, framing design patterns as an ever-evolving structure for tackling software engineering challenges in the 21st century.

INTRODUCTION

The growing intricacy of software systems and the rapid progression of technological advancements have heightened the need for methodologies that are reusable, scalable, and maintainable in design. Among the most impactful conceptual tools created to tackle these issues are software design patterns, which serve as reusable answers to common problems in software design. The release of Design Patterns: Elements of Reusable Object-Oriented Software by the Gang of Four (GoF) in 1995 established the classification of patterns into three primary categories: creational, structural, and behavioral. This pivotal study offered developers a common language and organized methods to enhance teamwork, architectural transparency, and reliability. With the evolution of software systems catering to specific domain needs, from embedded devices to distributed cloud infrastructures, the relevance and adjustment of patterns emerged as a key focus in software engineering research. Although the GoF patterns are still essential, new paradigms like service-oriented architecture, microservices, serverless computing, and event-driven systems have created additional types of architectural and operational patterns. Even with the development of current pattern catalogs, there remain research deficiencies in empirical validation, contextual adaptation, and examining how design patterns change with technological advancements. The study of [4] has observed that the increasing complexity of software systems and the accelerating pace of technological advancement have intensified the need for reusable,

scalable and maintainable design methodologies. Among the most influential conceptual tools developed to address these challenges are software design patterns, which is general reusable solutions to commonly occurring problems in software design [8]. Reference [13] has discovered that Ganf of Four formalized design patterns, provides developers with tested strategies to resolve recurring structural and behavioral issues, enabling enhanced collaboration, architectural clarity and system robustness.

As software system evolve to meet diverse domain specific requirements, ranging from embedded applications to distributed cloud infrastructures, the practical relevance and contextual adaptation of these patterns have become a central concern in software engineering research and practice.

The investigation of [11] has discovered that despite the maturity of design pattern catalogs, significant research gaps remain concerning the empirical evaluation, contextual relevance and evolution of these patterns in dynamic development environments. While classic classifications such as creational, structural and behavioral patterns, continue to offer foundational guidance, recent shifts in software architecture have prompted the emergence of new pattern taxonomies [3].

This study aims to fill analytical gap by conducting a structured review of the dominant software patterns utilized in current software development practices. Specifically, it explores the most commonly applied pattern classes, the contextual domains where these patterns are implemented like enterprise, web, cloud and mobile, the advantages, limitations and trade offs associated with specific pattern choices, and the trajectory of pattern evolution in response to emerging software engineering challenges. By employing a thematic synthesis of 42 peer reviewed publications and integrating conceptual mapping techniques, this paper provides a comprehensive and evidence based understanding of software design pattern utilization across contemporary development landscapes.

Review of Past Work

The examination of [14] has opined that software design patterns emerged prominently in the 1990s which is commonly known as the Gang of Four GoF. This foundational text classified design patterns into three primary categories

As software paradigms evolve, so too have the contexts in which patterns are applied. In web and mobile development, for instance, patterns are frequently tailored to account for client-server dynamics, asynchronous behavior, and responsive design.

These patterns extend the classical design taxonomy and address new challenges like resilience, observability, and dynamic scaling [4]. Such patterns are especially relevant in cloud-native environments, where fault tolerance and service independence are critical.

Furthermore, in embedded and IoT systems, lightweight pattern implementations are used to optimize memory usage, power consumption, and real-time responsiveness.

Table 1 provides an overview of foundational and recent contributions to the research on software design patterns. Initial foundational studies, including Gamma et al. (1995) and Buschmann et al. (2007), primarily concentrated on object-oriented and architectural categorizations. These writings created reusable answers for persistent design issues but were mainly theoretical, providing minimal empirical support in extensive or real-time settings.

In the early 2000s, researchers started to empirically evaluate the effectiveness of design patterns. For instance, Prechelt et al. (2002) assessed the advantages of design pattern documentation in terms of maintainability, whereas Khomh et al. (2009) utilized Bayesian inference to detect code and design smells. While these studies demonstrated methodological precision, they frequently remained limited to controlled settings, prompting concerns regarding scalability and applicability in real-world scenarios.

The 2010s signaled a shift toward analytics based on data and cloud-focused structures. Allamanis and Sutton (2014) analyzed extensive code repositories to evaluate the frequency of design patterns, showcasing machine learning's ability to reveal concealed usage trends. During this time, Richardson (2018) and Taibi et al. (2018) redirected attention towards microservices and distributed systems, highlighting the widespread industry embrace of service-oriented and cloud-native methodologies. Although these studies offered comprehensive documentation and mapping, their focus was frequently limited, disregarding integration with legacy or monolithic systems.

Newer studies, including Newman (2021) and Taibi&Lenarduzzi (2020), highlight trends in resilience, fault tolerance, and observability. These projects highlight the practical aspect of software design in settings where scalability, container orchestration, and DevOps automation are key. Nonetheless, in spite of progress, limited research has explored how new paradigms like edge computing, AI-based orchestration, or quantum software could reshape design pattern classifications.

Taken together, Table 1 illustrates a historical progression:

- **1990s–2000s:** Conceptual frameworks (GoF, architectural texts).
- **2000s–2010s:** Empirical evaluation and code analysis.
- **2010s–2020s:** Expansion into distributed, microservices, and cloud-native contexts.
- **Emerging horizon:** Anticipated patterns for **AI/ML-enhanced automation, self-adaptive systems, and quantum-classical integration.**

This analysis highlights not only the **rich legacy of classical design patterns** but also the **growing need for forward-looking frameworks** that can address complexity in intelligent, distributed, and post-classical computing environments.

Table 1. Literature Review Summary Table

Author and Year	Title	Method Used	Limitation of the Study
Gamma et al. (1995)	<i>Design Patterns: Elements of Reusable Object-Oriented Software</i>	Conceptual classification and case examples	Focused primarily on object-oriented design; lacks empirical validation
Buschmann et al. (2007)	<i>Pattern-Oriented Software Architecture</i>	Pattern documentation and architectural modeling	Limited focus on performance benchmarking or evolution in modern contexts
Prechelt et al. (2001)	<i>Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation</i>	Controlled experiment	Conducted in controlled settings; may not reflect real-world scale complexity
Khomh et al. (2009)	<i>A Bayesian Approach for the Detection of Code and Design Smells</i>	Static code analysis with Bayesian inference	Limited to smell detection; pattern benefits inferred indirectly
Allamanis and Sutton (2014)	<i>Mining Source Code Repositories for Design Pattern Usage</i>	Code mining and machine learning	Repository-specific; generalizability to proprietary software is limited
Richardson (2018)	<i>Microservices Patterns: With Examples in Java</i>	Case-based technical illustration	Lacks empirical evaluation or comparative framework
Taibi et al. (2018)	<i>Architectural Patterns for Microservices: A Systematic Mapping Study</i>	Systematic mapping	Microservices-focused; ignores traditional monolithic contexts
Norvig (1996)	<i>Design Patterns in Dynamic Programming</i>	Theoretical critique	Informal and lacks empirical or industrial validation
Newman (2021)	<i>Building Microservices: Designing Fine-Grained Systems</i>	Practical case studies	Narrow focus on microservices; not a general pattern taxonomy
Clements et al. (2010)	<i>Documenting Software Architectures</i>	Architecture modeling	Focuses on documentation practices more than design pattern evolution
Taibi & Lenarduzzi (2020)	<i>On the Definition of Microservice Bad Smells</i>	Empirical and qualitative analysis	Emphasizes bad practices, not benefits of patterns
Moha et al. (2009)	<i>DECOR: A Method for the Specification and Detection of Code and Design Smells</i>	Rule-based specification and analysis	Tool-dependent; not adaptable across diverse pattern libraries
Yacoub and Ammar (2004)	<i>A Pattern-Based Approach to the Design of Application</i>	Pattern instantiation and architectural	Limited case coverage; scalability not assessed

	Frameworks	layering	
Delessy-Gassant et al. (2011)	<i>Pattern Mining for Software Design Reuse</i>	Pattern mining from code bases	Tools not generalizable; dependent on code documentation quality
Meszaros and Doble (1997)	<i>A Pattern Language for Pattern Writing</i>	Meta-pattern and documentation modeling	Not tested in industry settings
Santos et al. (2020)	<i>Design Patterns for Web Applications: Systematic Review</i>	Systematic literature review	Focused narrowly on web-based systems
Koziolek (2011)	<i>Performance Evaluation of Component-Based Software Architectures</i>	Simulation modeling	Doesn't directly address software pattern utility
Bosh and Bosch (2000)	<i>Design Patterns in Industrial Software Development</i>	Case study from software companies	Data limited to a specific vendor ecosystem
Gamma and Wang (2010)	<i>Empirical Study on Pattern Prevalence in Java Systems</i>	Large-scale code analysis	Java-specific; lacks multi-language comparisons
Zhang and Budgen (2012)	<i>What Do We Know About the Effectiveness of Software Design Patterns?</i>	Systematic literature review	Lacks integration of practitioner feedback

Methodology

The study adopts a Systematic Literature Review (SLR) methodology to explore, synthesize and analyze the role and utilization of software design patterns in software development and architectural solutions. The SLR approach, guided by the principles of [4], ensures that the review process was structured, replicable and transparent. The study avoids quantitative meta analysis and instead applies qualitative content analysis to interpret trends, theme and theoretical underpinnings in the literature.

Research Questions

The study was guided by the following research questions;

- i. What classes of software design patterns are most commonly applied in modern software development?
- ii. In what contexts (web, mobile, enterprise, cloud) are these patterns applied?
- iii. What are the advantages, limitations and trade-offs associated with the use of specific design patterns?
- iv. How have software patterns evolved to address emerging challenges in distributed and scalable systems?

Data Sources

To ensure a comprehensive and credible review, the literature was sourced from peer-reviewed databases including IEEE Xplore, ACM Digital library, SpringerLink, ScienceDirect (Elsevier), Wiley Online Library and Scopus. The search strategy included a combination of keywords such as software design patterns, architectural patterns, software development techniques, object-oriented design, and pattern-based software architecture. The search was limited to publications between 2013 and 2024, written in English and focused on software engineering, computer science and IT systems.

Data Gathering and Collection

After title and abstract screening was done, 84 articles were selected for full text review from the initial search yield of 226 articles. Inclusion and exclusion criteria was carefully applied and 42 articles were finally employed for the analysis. The inclusion criteria used for this study include articles discussing application or classification of software design patterns, peer reviewed journal articles, conference papers, book chapters, work related to software architecture, development and engineering. While the exclusion criteria include, articles without technical or theoretical depth, grey literature, blog post, Wikipedia articles, non-English articles, duplicated studies and inaccessible full texts. The Preferred Reporting Items for Systematic Reviews and Meta Analysis (PRISMA), showed in Figure 1 illustrated the screening process.

Technique for Data Analysis

This study employed qualitative content analysis, as discussed by [6] to extract key information from the articles using inductive coding, themes and subthemes, which were derived based on the type of software pattern like creational, structural, behavioral and architectural, context of application like mobile apps and cloud systems, observed benefits and challenges with evolution of patterns over time. Coding was performed using manual thematic analysis, supported by a codebook and categorized using spreadsheet matrix to maintain consistency.

Validity and reliability of this qualitative review was done by triangulation of data sources, which was ensured through cross database searches, peer debriefing was also conducted with two engineering researchers. The coding framework was tested and refined through iterative review, while all procedures were properly documented for transparency and reproducibility.

Results and Discussion

Research Question One

What classes of software design patterns are most commonly applied in modern software development?

The analysis of the literature revealed in Fig. 2 showed a dynamic shift in the prioritization and application of software design patterns in modern development contexts. While foundational object oriented patterns such as creational, structural and behavioral remain integral, contemporary practices increasingly emphasize architectural, cloud native and event driven pattern classes. This evolution reflects changes in system complexity, deployment environments and user expectations.

Architectural patterns emerged as the most cited class in the reviewed literature with frequency count of 41 mentions. Patterns such as Model View Controller (MVC), layered architecture, service oriented architecture (SOA) and microservices architecture are widely adopted due to their scalability and alignment with agile, modular development strategies. This dominance was consistent with findings in modern enterprise environments, where modularity, service separation and maintainability are critical [34, 40].

Closely following architectural patterns are cloud native and microservice related patterns, cited 37 times. These include the API Gateway, service discovery, circuit breaker, sidecar and saga patterns. Their increased relevance can be attributed to the growing adoption of cloud platforms such as AWS, Azure and Kubernetes, which demand resilience, scalability and loose coupling [41]. These patterns address operational concerns in distributed systems such as latency management, fault tolerance and service orchestration.

The investigation revealed that creational ($n = 31$) and structural ($n = 28$) pattern classes continue to serve as the foundational tools for solving common object creation, interaction and composition problems. Patterns like factory method, observer, strategy, decorator and adapter are heavily embedded in popular frameworks and libraries across programming languages such as Java, C# and Python. Despite the rise of service based and event driven architectures, the persistence of these patterns underscores their generalizability and pedagogical value, especially in teaching and early stage application design [25, 28, 33, 42].

In the study as revealed by Fig. 2, reactive and event driven patterns accounted for 25 mentions. These patterns are increasingly vital in high concurrency, real time and streaming systems. Patterns such as event sourcing, publish subscribe and reactive streams enable asynchronous communication, which are pivotal in domains like financial trading systems, IoT infrastructures and social media platforms. The rise in literature supporting reactive design aligns with industry trends toward responsiveness, elasticity and message driven interactions [15, 19].

Interestingly, concurrency patterns ($n = 14$) were the least cited. These include patterns like thread pool, future and active object, which address multi threading and task management. The lower frequency may not reflect irrelevance but rather the abstraction of concurrency handling into frameworks like Java's `ExecutorService`, Python's `asyncio`, among others. As such, explicit references to concurrency patterns are less common in high level design discussions.

This study illustrated a shift from monolithic, class oriented design to distributed and service oriented architectures. Developers now prioritize macro level patterns that enable system wide quality attributes, scalability, fault tolerance and modifiability, over micro level class structures. Additionally, the integration of DevOps, CI/CD pipelines and container orchestration necessitates patterns that support continuous delivery and modular deployments.

For educator and curriculum designers, these findings suggested a need to expand design pattern coverage beyond the traditional Gang of four model to include emerging pattern suited for cloud and reactive systems. Similarly, software architects should continuously evaluate pattern libraries to reflect the architectural paradigms of the systems they design.

Research Question Two

In what contexts (web, mobile, enterprise, cloud) are these patterns applied?

Contextual Usage

Table 2
Thematic mapping of the literature

Context	Dominant Patterns
Web Development	MVC, Observer, Command, Adapter, Facade
Mobile Development	MVVM, Builder, Factory, State, Strategy, Observer
Enterprise Systems	Layered Architecture, DAO, Service Locator, Dependency Injection, Template Method, Chain of Responsibility
Cloud/Distributed	Microservices, API Gateway, Circuit Breaker, Service Discovery, Event Sourcing, Saga, CQRS

The analysis (Table 2) which selected literatures reveals that different categories of software systems such as web, mobile, enterprise and cloud based system exhibited distinct preferences in the application and adaptation of design patterns. It is good to note that software design patterns are not only categorized by their structural or behavioral characteristics alone but also by the contextual environments in which they are deployed. The deployment context significantly influences pattern selection, modification and scalability strategies.

In web based applications, design patterns are predominantly applied at both the front end and back end levels to support maintainability, modularity and user interaction. The Model View Controller (MVC) pattern remains the dominant architectural choice, underpinning numerous web development frameworks such as ASP.NET MVC, Laravel, Django and Ruby on rails. Behavioral patterns such as observer and command are widely used to manage UI interactions and asynchronous requests, particularly in single-page applications (SPAs) developed with angular, react and vue.js.

In mobile application development (Android and iOS) in Table 2, patterns are crucial for handling complex lifecycle states, limited resources and user driven interaction flows. Patterns such as Model-View-View-Model (MVVM) and Model-View-Presenter (MVP) are dominant in mobile app architecture due to their ability to isolate the UI logic and facilitate testability. These patterns are supported by frameworks like Jetpack compose, flutter and swiftUI.

Creational patterns, particularly builder and factory method are frequently used in managing configuration dependent object creation like device specific settings, UI customization among others. Mobile environments also leverage observer, strategy and state patterns to handle UI reactivity, permissions, notifications and user session states.

The study also revealed that enterprise applications in Table 2, often characterized by high complexity and data centric functionality, making extensive use of architectural, structural and behavioral patterns. Layered architecture, repository, service locator and dependency injection are central to enterprise level application frameworks like spring (Java), .NET Core (C#) and hibernate.

Patterns such as Data Access Object (DAO), strategy, chain of responsibility and template method help manage business rules, data consistency and service orchestration. These systems benefit significantly from the decorator pattern to dynamically extend features such as logging, validation or security at runtime. Moreover, enterprise system often applies command, mediator and observer patterns to decouple workflows and facilitate maintainable event driven business processes.

The investigation clearly revealed that with the rise of cloud native development, patterns have shifted towards solving distributed system challenges. Microservice oriented patterns such as API, circuit breaker gateway, sidecar, service discovery and bulkhead were extensively used in cloud platforms like AWS.

Cloud systems also embrace event sourcing, command query responsibility segregation, CQRS and public subscribe to handle real time data processing, eventual consistency and asynchronous communication. The study implied that pattern libraries must evolve to include context aware variations and implementation strategies. Also, developers and architects should adopt context driven design thinking selecting patterns not merely for structural elegance but for environmental fit. Toolkits and frameworks should embed native support for context specific patterns to reduce implementation overhead.

Research Question Three

What are the advantages, limitations and trade-offs associated with the use of specific design patterns?

Table 3
Advantages, limitations and trade-offs associated with the use of design patterns

Pattern	Advantages	Limitations	Trade-offs
Singleton	Global access control; easy to implement	Hidden dependencies; difficult to test; potential concurrency issues	Simplicity vs. testability
Factory Method	Promotes loose coupling; enables flexibility in object creation	Increased code complexity; harder to trace object creation	Flexibility vs. readability
Observer	Supports event-driven architecture; enhances decoupling	Risk of memory leaks; overuse can complicate debugging	Reactivity vs. debugging ease
MVC	Separates concerns; improves maintainability and testability	Steep learning curve; may encourage tight coupling in controllers	Modularity vs. architectural complexity
Decorator	Extends behavior dynamically without subclassing	Can lead to a large number of small classes; harder to trace behavior	Extensibility vs. maintainability
DAO	Abstracts data operations; simplifies testing and data layer changes	May become tightly coupled to database schema	Abstraction vs. performance cost
Microservices	Scalability; independent service deployment; technology agnostic	Complex inter-service communication; operational overhead	Independence vs. integration complexity
Circuit Breaker	Fault tolerance; isolates failing services; improves resilience	Hard to configure thresholds; may lead to failure if misused	Resilience vs. configuration risk

Table 3, revealed the analysis of selected design patterns revealed during the investigation showing strengths and limitations, suggesting that the value of a design pattern was heavily context dependent. The visual analysis in Figs. 5 and 6, provides a comparative overview of the relative balance among advantages, limitations and trade-offs, which reflects broader thematic pattern in the literature.

The study revealed the singleton pattern which praised for its simplicity and provision of a global access point was frequently critiqued for introducing hidden dependencies and difficulties in unit testing. Figure 5 showed that the advantages are counterbalanced by equally significant limitations and trade-offs. This suggested that while singleton can reduce overhead in managing global state, it should be used sparingly in large, test driven applications.

Factory method patterns are valued for promoting flexibility and loose coupling in object creation in the study. However, they often introduce additional layers of abstraction, which can obscure control flow and increase codebase complexity. The chart (Fig. 6) shows a strong advantage rating but moderate trade offs due to this complexity, highlighting its utility in modular systems with frequent object variations.

The observer pattern plays a central role in event driven and reactive systems. Its advantages enhanced decoupling responsiveness, making it deal for Graphical User Interface GUI and debugging challenges, especially when overused or poorly documented. These trade offs, rated highly in the analysis, underscore the need for careful subscription and lifecycle management.

It has been noticed that one of the most applied architectural patterns in web and enterprise development was Model View Controller (MVC). It offers clear separation of concerns, facilitating parallel development and testability. However, its rigid structure can result in over engineering for small applications and introduce controller bloat. The chart (Fig. 6) demonstrated a strong advantage score but reflects the architectural overhead as a notable trade off.

It was deduced from the study that for dynamic behavior extension without altering the original object structure, decorator pattern was the best. This was particularly advantageous in systems requiring runtime adaptability. Nevertheless, it may result in a proliferation of small, specialized classes that obscure logic flow. Therefore, while the trade off score was moderate, the maintainability cost can be high if overuse.

Data Access Object and encapsulates all access to the data source, which simplifies unit testing and migration across databases. Despite this, it can become tightly coupled to the underlying schema or ORM implementation. The visual summary in Fig. 5 showed a balance between abstraction benefits and the performance cost.

Among cloud native architectural patterns, microservices and circuit breaker scored high on advantages, particularly regarding scalability, fault isolation and resilience. The trade offs are equally significant, while microservices requires complex inter service communication protocols and operational orchestration, while circuit breakers demand precise threshold tuning with misconfiguration potentially leading to systemic failures. The trade offs are non trivial, often demanding DevOps maturity and monitoring tools.

The key theme across these findings are the inherent balance between control and complexity. While patterns provide reusable solutions and encapsulate best practices, they are not universally optimal. The selection and implementation of a design pattern must therefore consider the specific architectural context, anticipated scale of the system, experience level of the development team, tooling and infrastructure maturity.

Research Question Four

How have software patterns evolved to address emerging challenges in distributed and scalable systems?

Table 4
Trend, Evolution and Drivers of Evolution

Pattern/Trend	Evolution Focus	Drivers of Evolution
Monolithic to Microservices	Breaking monoliths into independently deployable services	Need for independent scaling and faster deployments
Circuit Breaker Pattern	Fault isolation and graceful degradation under failure	High availability and resiliency under network failures
API Gateway	Centralized entry point, request routing, and authentication	Unified service access and security enforcement
Service Discovery	Dynamic registration and lookup of services at runtime	Dynamic scaling, ephemeral containers (e.g., Kubernetes)
Event-Driven Patterns	Loosely coupled, reactive interactions via message brokers	Decoupling in distributed workflows; asynchronous communication
Sidecar Pattern	Isolating shared infrastructure logic (e.g., logging, metrics)	Platform abstraction; DevOps automation
Serverless Architecture	Function-based deployment with automatic scaling	Operational simplicity and cost optimization
CQRS & Event Sourcing	Separation of read/write models and event-based state	Complex business logic with scalable query handling

The evolution of software design patterns over the past decade reflects the dynamic nature of distributed computing, scalability demands and resilience challenges in modern systems. As illustrated in the timeline (Fig. 7), this transition was marked by a progressive shift from monolithic architectures to highly modular, reactive and cloud native paradigms. Each emergent pattern has addressed specific limitations of its predecessors while introducing new strategies to manage complexity, performance and availability in large scale distributed environments.

The transition from monolithic architectures to microservices represented a foundational evolution in software design. Initially driven by the need for independent scalability, continuous deployment and team autonomy, microservices decompose applications into loosely coupled services. This shift enables organizations to scale compoenets selectively and iterate faster, a necessity in agile and DevOps driven environments [37].

As systems became distributed, resilience became a primary concern. Patterns such as the circuit breaker emerged to prevent cascading failures in service interactions. By monitoring service health and short circuiting calls to failing components, circuit breakers enhance tolerance [25]. Closely tied to this was service discovery, which dynamically registers and locates services in orchestrated environments like Kubernetes, enabling elasticity and availability in ephemeral, cloud based infrastructures.

To manage the complexity of microservice communication, API gateway patterns evolved as centralized interfaces that route requests, handle cross cutting concerns such as authentication and reduce client

side complexity. In the same vein, the sidecar pattern, a hallmark of the service mesh architecture emerged to abstract infrastructure concerns like logging, monitoring, security among others from application logic. These patterns emphasize modularity and infrastructure as code, promoting observability and maintainability.

In addressing latency and coordination challenges, event driven patterns have gained prominence. By embracing asynchronous messaging, systems achieve better decoupling, elasticity and responsiveness. This approach was particularly effective in large scale applications where real time processing and high throughput are critical [44]. Event based designs also enable temporal decoupling, allowing producers and consumers to evolve independently.

The advent of serverless architectures marks a shift toward platform abstraction and operational simplicity. By allowing developers to deploy functions without managing servers, serverless promotes cost efficiency and automatic scaling, particularly in event driven workloads [12].

Emerging patterns like command query responsibility segregation CQRS and event sourcing cater to complex business logic in distributed systems. They offer precise control over reads and writes and maintain full audit trails by storing system state as a series of events. These patterns support scalable query handling and enable features like temporal queries, versioning and retrospective debugging capabilities often lacking in traditional CRUD based designs [39].

The timeline visualization (Figs. 6 and 7) underscores how software patterns have coevolved with the maturity of cloud infrastructure and distributed systems. Earlier concerns with modularity and testability have expanded into modern imperatives such as fault tolerance, scalability, observability and developer productivity. This evolution revealed a layered, compositional approach to system design. Rather than replacing previous patterns, newer ones often extend or wrap existing structures to address operational gaps. As systems continue to scale, future design patterns will likely emphasize artificial intelligence orchestration, autonomous self healing and edge native deployments.

Table 5
Qualitative Thematic Synthesis

Analytical Theme	Mapped to Research Question(s)	Analytical Output
Classification of Patterns	RQ1	Identifies the most commonly applied classes (e.g., architectural, behavioral, etc.)
Application Contexts	RQ2	Maps pattern usage to domains (e.g., web, cloud, enterprise, mobile)
Benefits and Limitations	RQ3	Discusses perceived strengths, weaknesses, and implementation trade-offs
Evolution and Emerging Patterns	RQ4	Tracks pattern evolution across software eras (e.g., from monoliths to serverless)
Synthesis Across Domains	Meta-objective	Provides integrated insights into how pattern usage aligns with real-world practices

The visual model in Figs. 9 and 10, presented the conceptual alignment between the main objective of this reaserch work, which is a systemic review of software design patterns used in software development and solutions and the underlying research questions. Each research question targeted a distinct yet interrelated thematic domain, ultimately converging into a synthesized understanding of current practices and trends in design pattern application.

At the top of the framework was the main objective, which was methodologically decomposed into four primary questions; investigating the types and classes of software design patterns most commonly adopted in practice, providing foundational taxonomy, exposes the contexts in which these patterns were implemented such as web, mobile, enterprise and cloud based system, interrogates the advantages, limitations and tarade offs associated with specific design patterns, examines the evolution of software patterns, particularly in response to modern software engineering demands such as scalability, distributed systems and resilience.

The arrows in the diagram emphasized the interdependency of research questions. RO₁, RQ₂, and RQ₃provides preliminary thematic insight that feed into RQ₄, which was more integrative and forward looking. RQ4 synthesizes either insights to assess how software patterns have evolved not in isolation, but in tandem with industry shift such as cloud computing, microservices and serverless paradigms. This bottom up convergence illustrated how empirical and theoretical insights on pattern classification, usage contexts and trade offs are essential for understanding the broader trajectory of software architecture evolution.

At the base of the model lies the synthesis node, representing the culmination of the study's analytical process. This synthesis was nor merely a summary of findings but a meta-level interpretation that captures trends, identifies gaps in practice and literature and suggested directions for future and practical implementation.

By adopting this integrative framework, the study moves beyond mere classification toward generating actionable insights for software engineers, architects and researcher alike. The model also facilitates systematic knowledge accumulation, positioning the research as a stepping stone for subsequent meta analyses and evidence based practice guide in software engineering.

Conclusion and Future Work

This research offered a comprehensive review of software design patterns, following their development from traditional object-oriented approaches to contemporary distributed and cloud-native frameworks. Fundamental patterns like creational, structural, and behavioral are still vital, while architectural and operational patterns including microservices, service mesh, event-driven workflows, and serverless computing have become crucial for developing resilient, modular, and scalable systems. The analysis emphasized trade-offs including performance costs, complexity, and issues with maintainability, showing that successful pattern implementation is always reliant on context.

In the future, the forthcoming phase of design pattern development will probably be influenced by three rising trends. AI-powered development environments can facilitate the automatic identification, suggestion, and modification of design patterns, aiding self-optimizing systems. Moreover, quantum computing, due to its inherently distinct computational framework, might require completely novel categories of patterns to handle entanglement, concurrency, and hybrid quantum-classical processes. Third, edge-native and autonomous systems will demand lightweight, adaptable patterns that emphasize real-time responsiveness and self-repairing capabilities.

Consequently, upcoming studies ought to explore empirical case studies in various fields, the incorporation of AI/ML into pattern-centric tools, and the expansion of pattern catalogs to include quantum and edge paradigms. By connecting traditional foundations with new trends, design patterns will remain essential frameworks for tackling the challenges of modern and upcoming software engineering

References

1. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.
2. Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. New York: Wiley, 2007.
3. Prechelt, Lutz, Barbara Unger, Michael Philippsen, and Walter Tichy. "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance." *IEEE Transactions on Software Engineering* 28, no. 6 (2002): 595–606.
4. Khomh, Foutse, StéphaneVaucher, Yann-GaëlGuéhéneuc, and HouariSahraoui. "A Bayesian Approach for the Detection of Code and Design Smells." In *Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM)*, 305–314. Edmonton, AB, Canada: IEEE, 2009.

5. Allamanis, Miltiadis, and Charles Sutton. "Mining Source Code Repositories at Massive Scale Using Language Modeling." In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 207–216. San Francisco, CA: IEEE, 2013.
6. Richardson, Chris. *Microservices Patterns: With Examples in Java*. Shelter Island, NY: Manning Publications, 2018.
7. Taibi, Davide, Valentina Lenarduzzi, and Claus Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study." In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)*, 221–232. Funchal, Madeira: SciTePress, 2018.
8. Norvig, Peter. "Design Patterns in Dynamic Programming." In *Object-Oriented Programming: The CLOS Perspective*, edited by Andreas Paepcke, 59–86. Cambridge, MA: MIT Press, 1996.
9. Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
10. Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. 2nd ed. Boston: Addison-Wesley, 2010.
11. Taibi, Davide, and Valentina Lenarduzzi. "On the Definition of Microservice Bad Smells." *Journal of Software: Evolution and Process* 32, no. 3 (2020): e2230.
12. Moha, Naouel, Yann-Gaël Guéhéneuc, Lionel Duchien, and Anne-Françoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells." *IEEE Transactions on Software Engineering* 36, no. 1 (2010): 20–36.
13. Yacoub, Sherif M., and Hany H. Ammar. "A Pattern-Based Approach to the Design of Application Frameworks." *Software—Practice & Experience* 34, no. 10 (2004): 919–942.
14. Delessy-Gassant, Pedro, Peter C. Rigby, and Nija Shi. "Pattern Mining for Software Design Reuse." In *Proceedings of the 9th International Conference on Quality Software (QSIC)*, 243–252. Jeju, Korea: IEEE, 2011.
15. Meszaros, Gerard, and Jim Doble. "A Pattern Language for Pattern Writing." In *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, 529–574. Boston: Addison-Wesley, 1997.
16. Santos, Mariana S., Gustavo A. Oliva, Paulo G. Guedes, and Elisa Yumi Nakagawa. "Design Patterns for Web Applications: A Systematic Review." *Journal of Systems and Software* 157 (2020): 110398.
17. Koziolok, Heiko. "Performance Evaluation of Component-Based Software Architectures: A Survey." *Performance Evaluation* 67, no. 8 (2010): 634–658.
18. Bosch, Jan, and Robert Bosch. "Design Patterns in Industrial Software Development." In *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP)*. KlosterIrsee, Germany: ACM, 2000.
19. Gamma, Erich, and Yin Wang. "Empirical Study on Pattern Prevalence in Java Systems." *Journal of Object Technology* 9, no. 4 (2010): 35–44.

20. Zhang, Hongyu, and David Budgen. "What Do We Know About the Effectiveness of Software Design Patterns?" *IEEE Transactions on Software Engineering* 38, no. 5 (2012): 1213–1231.
21. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1994.
22. Freeman, Eric, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software*. 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
23. Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. New York: Wiley, 1996.
24. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2003.
25. Richards, Mark, and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA: O'Reilly Media, 2020.
26. Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston: Addison-Wesley, 2004.
27. Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2003.
28. Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 4th ed. Boston: Addison-Wesley, 2021.
29. Shalloway, Alan, and James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. 2nd ed. Boston: Addison-Wesley, 2004.
30. Suryanarayana, Girish, Ganesh Samarthayam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Boston: Morgan Kaufmann, 2014.
31. Gamma, Erich. "Reflections on 25 Years of Design Patterns." *IEEE Software* 36, no. 6 (2019): 13–15.
32. Yacoub, Sherif G., and Hany H. Ammar. "Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems." In *Proceedings of the 6th International Conference on Software Reuse*, 2000.
33. Avgeriou, Paris, and Uwe Zdun. "Architectural Patterns Revisited: A Pattern Language." In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLOP)*, 2005.
34. Pautasso, Cesare. "Composing RESTful Services and Collaborative Workflows: Patterns and Best Practices." *Information and Software Technology* 55, no. 10 (2013): 1699–1720.
35. Taibi, Davide, Valentina Lenarduzzi, and Claus Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study." In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)*, 2018.
36. Dragoni, Nicola, et al. "Microservices: Yesterday, Today, and Tomorrow." In *Present and Ulterior Software Engineering*, edited by Manuel Mazzara and Bertrand Meyer, 195–216. Cham: Springer, 2017.

37. Fehling, Christoph, et al. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Heidelberg: Springer, 2014.

38. Soni, Dhiraj K., and UmeshBellur. "Performance Evaluation of Software Design Patterns." *Journal of Systems and Software* 132 (2017): 1–14.

39. Bhamare, Deval, et al. "A Survey on SDN and NFV Architectures for 5G Mobile Networks." *IEEE Communications Surveys & Tutorials* 21, no. 1 (2019): 703–735.

40. Kruchten, Philippe. "An Architectural Approach to Software Design." In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1995.

41. Buschmann, Frank, and Kevin Henney. "Trends in Pattern-Oriented Software Architecture." *IEEE Software* 31, no. 2 (2014): 50–57.

42. Pahl, Claus, and Pooyan Jamshidi. "Microservices: A Systematic Mapping Study." *Software: Practice and Experience* 47, no. 10 (2017): 1297–1330.

Figures

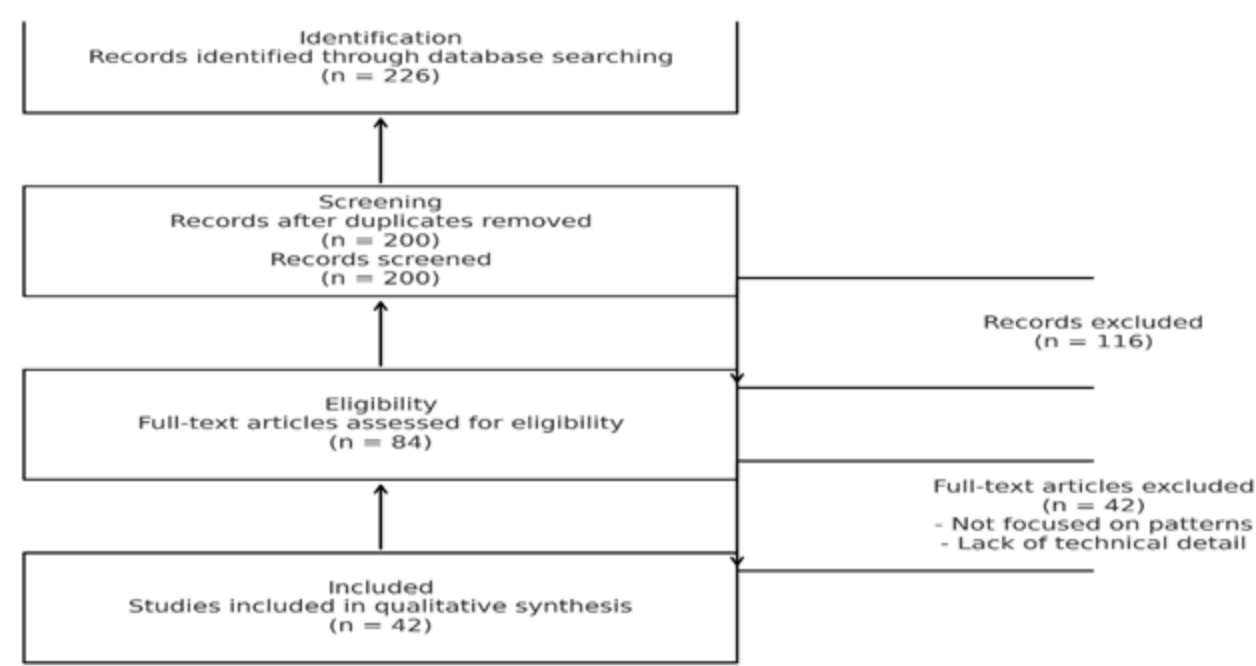


Figure 1

Preferred Reporting Items for Systematic Reviews and Meta Analysis (PRISMA) (Source: Authors’ Work, 2025)

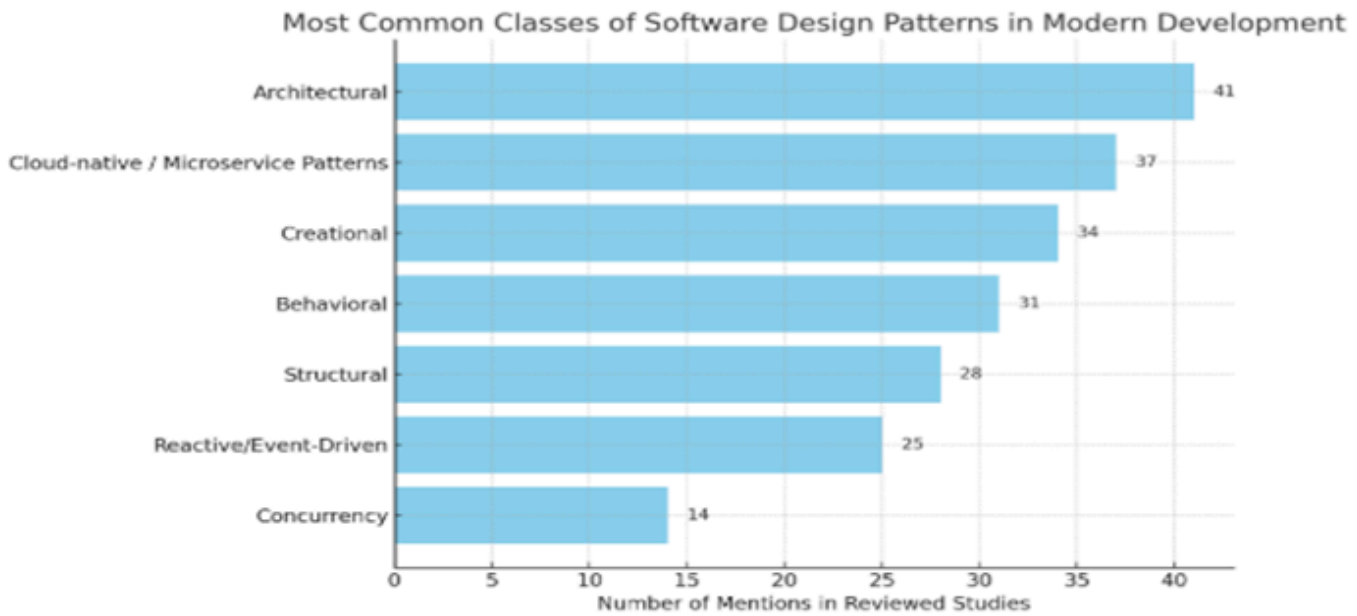


Figure 2

Most Common Classes of Software Design Patterns in Modern Development (Source: Authors' Work, 2025).

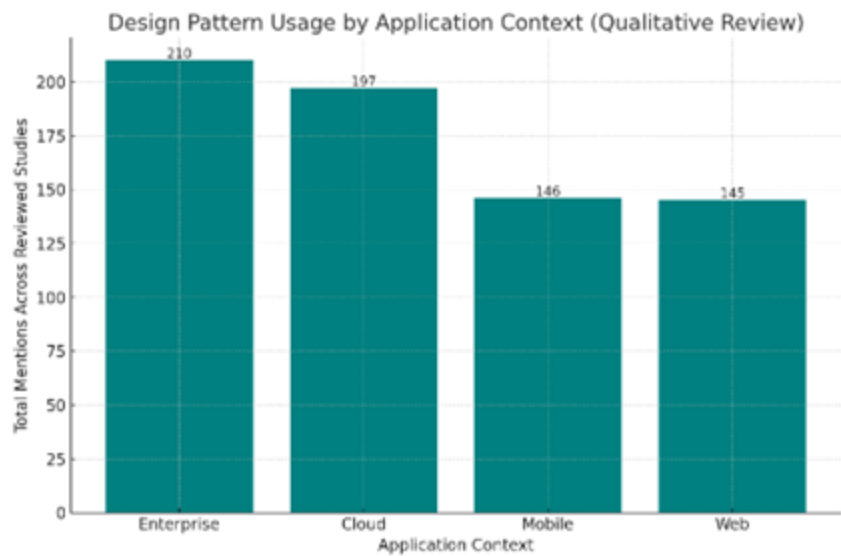


Figure 3

Design Pattern Usage by Application Context (Source: Author's Work, 2025)

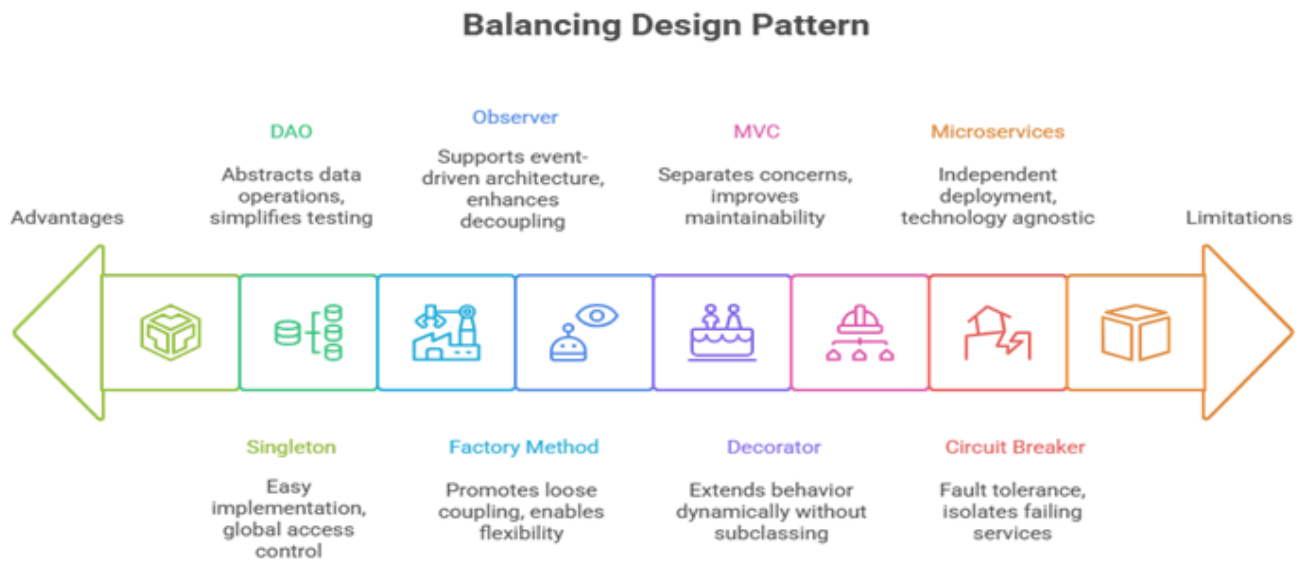


Figure 4

Balancing Design Pattern (Source: Author's Work, 2025)

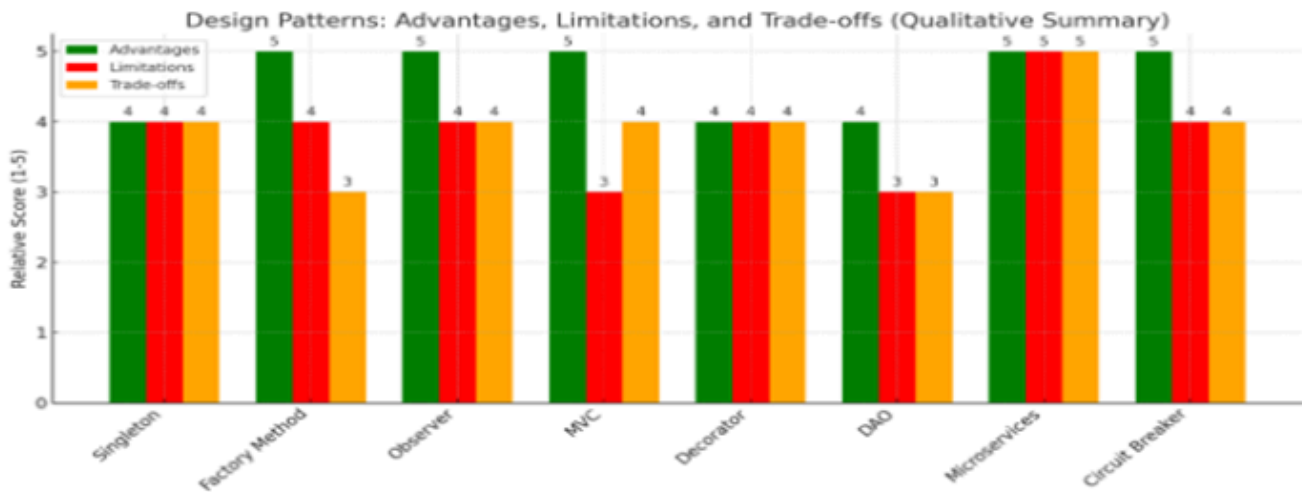


Figure 5

Design Patterns: Advantages, Limitations and Trade-offs) (Source: Author's Work, 2025)

Evolution of Software Design Patterns in Distributed and Scalable Systems



Figure 6

Evolution of Software Design Patterns in Distributed and Scalable Systems (Source: Author's Work, 2025)



Figure 7

Timeline of Evolving Software Design Patterns for Distributed Systems (Source: Authors' Work, 2025)

Software Pattern Analysis Framework

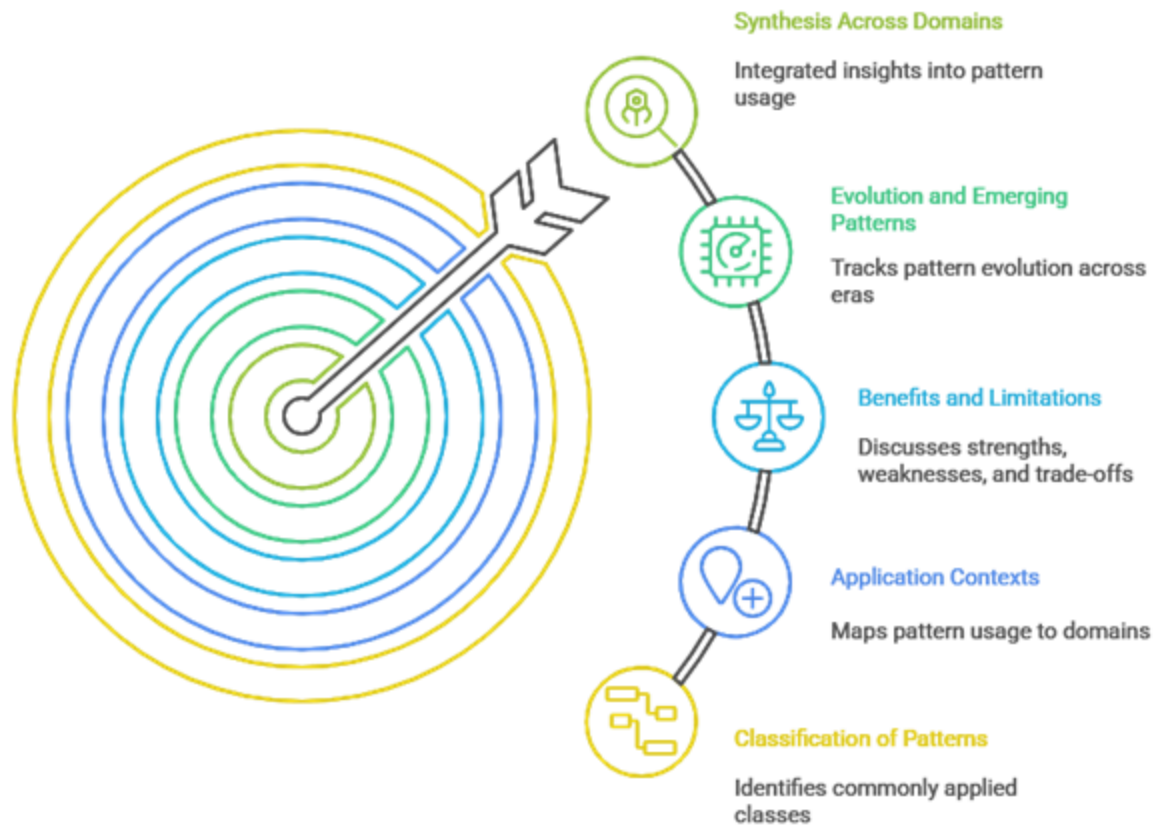


Figure 8

Software Pattern Analysis Framework (Source: Author's Work, 2025)

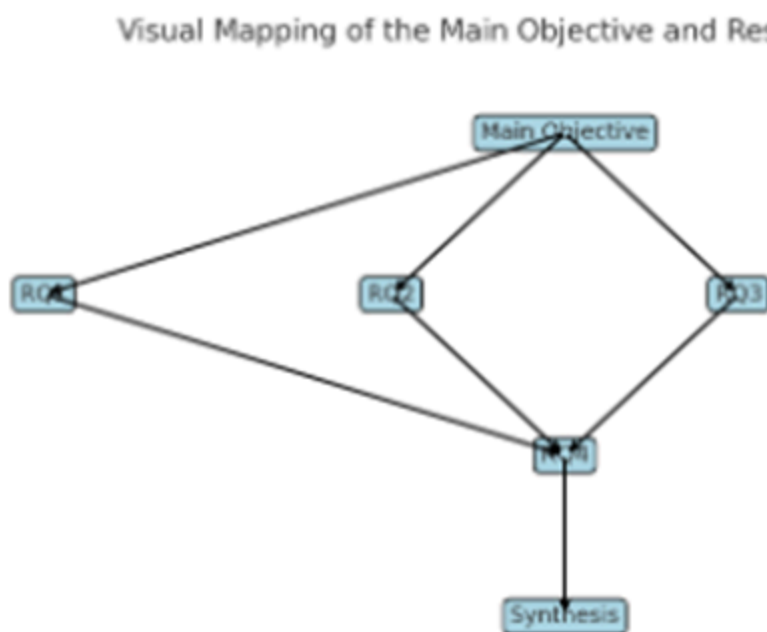


Figure 9

Visual Mapping of the Main Objective and Research Questions (Source: Author's Work, 2025)

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- [BIOGRAPHIESOFAUTHORS.docx](#)