Towards a Progressive Scalability for Modular Monolith Applications

Maurício Carvalho, Juliana de Melo Bezerra[a] and Karla Donato Fook[b]
Department of Computing Science, Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos, Brazil

Keywords: Software Engineering, Software Architecture, Cloud Computing, Modular Monolith, Microservices.

Abstract: Cloud-native software startups face intense pressure from limited resources, high uncertainty, and the need for rapid validation. In this context, early architectural decisions have lasting effects on scalability, maintainability, and adaptability. Although microservices are often favored for their modularity, they introduce significant operational overhead and require organizational maturity that many startups lack. Traditional monoliths offer simplicity but tend to evolve into rigid, tightly coupled systems. When designed with disciplined modularity, modular monoliths can offer internal boundaries that support sustainable growth while avoiding the fragmentation and complexity of premature microservices adoption. The existing literature emphasizes microservices, leaving gaps in guidance for modular monoliths on topics like modularization, scalability, onboarding, and deployment. This paper proposes guidelines for designing scalable modular monoliths, maintaining architectural flexibility, and reducing complexity, thereby supporting long-term evolution under typical startup constraints. The initial category of guidelines is presented, and their intended structure is thoroughly outlined.

1 INTRODUCTION

Startup entrepreneurs often face significant challenges when transforming ideas into products; only one in ten startups succeeds and self-inflicted issues are the reasons why most fail within two years (Crowne, 2002). Yet funding shortfalls are not the primary culprit because among venture-backed startups, only 25% generate any return on capital, and 30% to 40% of failures wipe out investors' entire initial investment. This suggests that internal missteps, rather than lack of money, drive the vast majority of startup failures.

A startup is a human institution designed to create new products or services under uncertainty (Ries, 2011), and it can be understood as a temporary organization searching for a repeatable and scalable business model (Blank and Dorf, 2012). Unlike small businesses, startups aim to scale rapidly once they reach product–market fit by identifying a target audience, understanding customer needs, and delivering an effective product (Ries, 2011).

Technological advances such as cloud computing, artificial intelligence and modern web and mobile frameworks have dramatically lowered the barrier to launching startups. These improvements make software a uniquely scalable form of leverage, since code can be deployed repeatedly at near-zero marginal cost. Code allows a single engineer to reach millions of users with minimal marginal cost. Yet even ten talented engineers can waste effort if they pursue the wrong model, build the wrong product, or make poor engineering decisions (Ravikant, 2020).

Startups face critical early choices. Some adopt microservices from the outset to seek modularity and scalability but incur complexity in deployment, orchestration and team coordination. Others begin with a monolithic approach to prioritize speed and simplicity yet encounter significant obstacles when evolving their systems to meet growing demands.

A recurring challenge in modular monolith and microservice architectures is achieving and maintaining *scalability*. This raises a critical question for software teams, namely *to avoid premature microservices adoption, as monoliths evolve, what architectural aspects must be evaluated to ensure modularity is preserved and the system remains distribution-ready?*

Despite recognition of these tensions and challenges in the cloud era, there are no widely adopted *guidelines* exist for progressively evolving software architecture, especially monoliths. Startups lack concrete criteria to assess when and how to introduce modular structures, split components, or transition to distributed systems including microservices.

[a] https://orcid.org/0000-0003-4456-8565
[b] https://orcid.org/0000-0002-3631-2554

Carvalho, M., Bezerra, J. M. and Fook, K. D.
Towards a Progressive Scalability for Modular Monolith Applications.
DOI: 10.5220/0013786800003985

In *Proceedings of the 21st International Conference on Web Information Systems and Technologies (WEBIST 2025)*, pages 228-235
ISBN: 978-989-758-772-6; ISSN: 2184-3252

This paper explores how modular monoliths address the need for flexible and scalable design and what is required to build modular, cloud-native applications through this architecture. It focuses on a key trade-off in cloud-native systems, that is, whether startups should adopt a modular monolithic architecture or a microservices-based design during their early stages of software development. The primary objective of this research is to propose a set of architectural guidelines that position modular monolithic architectures as a pragmatic starting point for building scalable and maintainable cloud-native applications in startups.

The paper is structured as follows. In the next section, we provide the background of our work. Section 3 outlines the related work. Section 4 provides a comprehensive overview of the proposed guidelines for designing and maintaining modular monolith applications, focusing on progressive scalability. The final section addresses our initial conclusions and explains the subsequent steps.

2 BACKGROUND

This section presents the technical foundations relevant to this paper, with a focus on the definitions and characteristics of cloud-native applications, microservices, monolithic architecture, system modularity and modular monoliths. Together, these paradigms shape how systems are built, maintained, and scaled in software startups.

Cloud native applications are software systems explicitly designed to operate in cloud environments leveraging elasticity, horizontal scalability, fault tolerance and automation. These systems adopt disposability, resilience and automation from inception, enabling faster delivery, effective failure recovery and seamless scaling using containerization (Docker), orchestration (Kubernetes) and CI/CD pipelines (Fowler, 2020). They emphasize service oriented or event driven architectures with decoupled, stateless components that degrade gracefully, recover autonomously and scale independently. For early stage startups, cloud native principles offer faster time to market and alignment with DevOps practices yet introduce complexity requiring clear modular boundaries and operational maturity.

Monolithic systems combine user interface, business logic and data access layers into a single deployable unit, simplifying early development and deployment. Fowler (2020) recommends starting with a well structured monolith and migrating to microservices only when demands justify added complexity. The challenge lies in engineering a monolith with clear internal modules from the outset. Without defined boundaries, many monoliths become tightly coupled and resistant to change, making future decomposition costly and error prone (Alshuqayran et al., 2016). Legacy monolith modernization remains one of the most complex and risky undertakings in software evolution.

Microservices architecture structures applications as collections of small, autonomous services, each encapsulating a distinct business capability. Services communicate via RESTful APIs or asynchronous messaging (Kafka, RabbitMQ), enforcing boundaries, promoting team autonomy, and supporting independent scaling. Fowler and Lewis (2014), Dragoni et al. (2017), and Taibi et al. (2018) emphasize how decomposition fosters organizational agility and technological flexibility.

At the same time, microservices introduce operational overhead, requiring distributed tracing, resilience strategies, and advanced automation that may exceed the capacity of early-stage teams. Premature adoption can lead to overengineering, performance bottlenecks, and cognitive overload (Fritzsch et al., 2019; Gysel et al., 2016). Moreover, translating ideals such as bounded context into clear service boundaries often proves difficult, resulting in architectural inconsistencies and accidental complexity (Taibi et al., 2018; Lauretis, 2019). These limitations motivate exploration of approaches that preserve modularity and scalability while minimizing overhead.

Modularity describes the degree to which a system's internal components can be isolated, composed and recombined without excessive coupling (Tilkov, 2015). It underpins sustainable software evolution across architectural styles but often remains assumed rather than engineered, leading to structural erosion over time. For startups, modularity enables rapid iteration, change isolation and adaptation of boundaries as products evolve. Enforcing internal boundaries early lets teams defer distribution decisions until real needs emerge. Neglecting modularity leads to two traps, either premature microservices adoption with unnecessary complexity or monoliths so tightly coupled that scaling requires expensive rewrites or risky migrations.

A *modular monolith* is an architectural style where a system is deployed as a single process but internally structured into independently evolvable and testable modules. Each module encapsulates a domain or functional responsibility, enforcing boundaries through internal APIs, strict dependency management, and clear ownership. Unlike traditional monoliths, modular monoliths aim for scalability and

maintainability from inception, combining deployment simplicity with design flexibility.

Literature shows that cloud-native monoliths can achieve coherence and scalability by applying evolutionary software architecture principles (Ford et al., 2017) and adhering to YAGNI *"You Ain't Gonna Need It"* (Beck, 1999). Proper boundary enforcement prevents the *big ball of mud* syndrome (Foote and Yoder, 1999), enables faster local testing, and provides intra-process performance advantages over microservices.

Recent industry case studies illustrate a return to modular monoliths in cloud-native applications. Segment adopted microservices for rapid parallel development but suffered operational overhead and latency; consolidating into a modular monolith reduced complexity and improved productivity (Segment, 2023). Shopify embraced a modular monolith from the start to preserve velocity and enforce boundaries (Shopify, 2022). Amazon Prime Video saw up to 90% performance and cost gains merging few microservices back to a modular monolith (Amazon, 2023). These cases show that modularity monoliths simplify CI/CD, reduce inter-service overhead and allow service extraction or merge without massive rewrites and complexity that increase development costs.

3 RELATED WORK

This section synthesizes prior research on cloud-native software architecture, with particular emphasis on modular monoliths. The analysis follows established Systematic Literature Review (SLR) guidelines (Kitchenham and Charters, 2007). The author screened peer reviewed journal and conference papers from 2019 to 2024 retrieved from IEEE Xplore and the ACM Digital Library. A Boolean search combined terms for modular monoliths, microservices, scalability and cloud native design. Inclusion required each study to address at least one core theme, modularity, maintainability, deployment strategy, scalability or system evolution, and was considered only English language publications. After full text screening and duplicate removal, 18 articles remained.

These studies were evaluated against twelve criteria organized into four dimensions. *Architectural Design* considers clear module boundaries, long-term maintainability, and scalability potential. *Operational Fit* examines migration readiness, deployment and automation strategies, and the adequacy of observability. *Organizational Alignment* evaluates the correspondence between architecture and team structure, the maturity of DevOps practices, and the ease of onboarding. Finally, *Guideline Orientation* emphasizes the use of practical patterns, adaptation to business context, and recognition of trade-offs. Together, these criteria establish a systematic basis for identifying strengths and weaknesses between studies.

Modularity is foundational because it enables systems to evolve without excessive coupling, supports clearer ownership boundaries, and lays the foundation for scalability and maintainability. Prakash and Arora (2024), Johnson et al. (2024), and Su et al. (2024) address the modularity of software directly. Migration-focused works such as Lauretis (2019) and Berry et al. (2024) examine it through refactoring or service extraction without detailed boundary analysis.

Maintainability emerges consistently across studies because it is central to the long-term viability of software systems. Berry et al. (2024) offer empirical metrics, while Su et al. (2024) provide qualitative insights. In microservices literature, maintainability is often linked to bounded contexts and team autonomy, whereas in modular monoliths it is associated with cohesion and local consistency. Yet few studies propose reusable patterns or tools that support long-term maintainability in real systems.

Scalability is emphasized across studies as it determines how systems respond to growth in users, data, and workload. Arya et al. (2024) and Berry et al. (2024) demonstrate microservices' horizontal scaling via containers and orchestration platforms. Conversely, Montesi et al. (2021) and Lauretis (2019) argue that modular monoliths satisfy moderate scalability requirements with lower operational complexity. However, formal models and guidance for hybrid or staged scalability remain scarce.

Migration readiness refers to assessing how prepared a monolithic system is to transition into a cloud-native, modular, and scalable application. Berry et al. (2024) and Ng et al. (2024) document legacy modernization efforts, while Montesi et al. (2021) position modular monoliths as a transitional stage between legacy architectures and modern systems.

Deployment strategy is well covered in microservices research because it underpins reliability, release velocity, and operational control. Fowler (2020), Arya et al. (2024), and Johnson et al. (2024) explore Kubernetes orchestration, CI/CD pipelines, and canary releases. Modular monoliths benefit from simplified, unified deployment but lack in-depth tooling analysis (Montesi et al., 2021).

Complexity management is observed across all architectures because it influences how teams handle accidental complexity and toolchain fragmentation. Su et al. (2024) and Montesi et al. (2021) propose

strategies to mitigate these challenges, while Prakash and Arora (2024) link complexity tolerance to organizational maturity, suggesting that teams with fewer resources often prefer monolithic coherence. However, there remains an opportunity to propose quantifiable strategies for managing complexity and cognitive load across architectural styles.

Computing Performance evaluations vary across studies. Berry et al. (2024) and Blinowski et al. (2022) compare response times and resource usage, showing microservices scale at the cost of inter service latency. Modular monoliths are less frequently benchmarked in production scenarios, leaving an evaluation gap.

Team fit is underexplored, particularly in terms of quantitatively assessing how architectural decisions affect collaboration and team structure. Prakash and Arora (2024) and Su et al. (2024) suggest that modular monoliths reduce cognitive load and improve collaboration in small cross-functional teams. However, most studies overlook broader social and organizational dynamics.

DevOps maturity is highlighted in Arya et al. (2024) and Johnson et al. (2024), contrasting the high operational demands of microservices with the accessibility of modular monoliths (Montesi et al., 2021). A clearer comparison of DevOps tooling and practices across architectures, particularly for early-stage teams, is still missing, and formal analyses of such requirements remain rare.

Onboarding of new engineers is seldom discussed, highlighting the need to evaluate how different architectural styles influence the onboarding experience and developer ramp-up. Montesi et al. (2021) and Tsechelidis et al. (2023) note that unified codebases can ease this process, but systematic evaluation remains absent.

Practical adoption guidance varies, particularly regarding how to bridge academic findings with concrete steps and decision support frameworks. Su et al. (2024) link architectural decisions to implementation workflows, while Tsechelidis et al. (2023) propose classification models without empirical validation. This persistent gap between theory and practice highlights promising avenues for future research.

Target context is inconsistently specified, underscoring the need to clarify the applicability of architectural recommendations to specific organizational scenarios. Prakash and Arora (2024) focus on small to medium teams, while Su et al. (2024) distinguish startup from enterprise settings. Greater clarity on organizational environments would improve the practical relevance of architectural recommendations.

Table 1 synthesizes literature coverage across the 12 criteria, highlighting key findings and gaps for future research.

4 GUIDELINES FOR MODULAR MONOLITH ARCHITECTURES

Rather than offering a static framework, the proposed guidelines aim to function as decision-making heuristics, designed to be actionable principles that inform architectural evolution without prescribing a single and rigid path. The guidelines are organized according to the four analytical dimensions, as shown in Figure 1: *Architectural Design*, *Operational Fit*, *Organizational Alignment*, and *Guideline Orientation*.

The *Architectural Design* dimension covers criteria related to the software's internal structure. Guideline *G1 (Enforce Clear Modular Boundaries)* emphasizes the need for explicit separation between modules, which reduces coupling, improves code coherence, and supports independent evolution of components. Guideline *G2 (Assure Long-Term Maintainability)* highlights practices that control complexity and preserve software quality, enabling that the system remains adaptable to future changes with minimal technical debt. Guideline *G3 (Ensure Scalability by Design)* requires structuring the architecture from the outset to accommodate increases in workload and user demand, thereby sustaining performance and reliability without structural degradation.

The *Operational Fit* dimension addresses the operational requirements needed to support production environments. Guideline *G4 (Ensure Migration Readiness)* emphasizes the importance of architectural flexibility, allowing systems to adapt to significant transitions such as decomposing into microservices or adopting new technological platforms. Guideline *G5 (Define a Robust Deployment Strategy)* establishes the need for reliable, automated, and repeatable release processes that minimize human error and accelerate delivery cycles. Guideline *G6 (Enable Comprehensive Observability)* highlights the role of logging, metrics, and monitoring in providing transparency, which supports fault detection, performance tuning, and continuous operational awareness.

The *Organizational Alignment* dimension focuses on aligning the software architecture with the organization's structure and development practices. Guideline *G7 (Balance Architecture with Organizational Structure)* emphasizes that architectural design should neither passively mirror organizational communication patterns nor entirely ignore them. While Conway's Law explains the natural tendency for systems to reflect team boundaries, this reflec-

Table 1: Literature Gap analysis by evaluation criterion and opportunities for research contributions.

Evaluation Criterion	**Literature Coverage**	**Opportunity for Contribution**
Modularity	High across studies, but often implied in migration-focused works.	To formalize modularity strategies beyond implicit mentions by proposing enforceable and testable design structures.
Maintainability	Frequently addressed with both qualitative and quantitative methods.	To develop reusable patterns and automated tools that enable sustainable long-term maintainability in practice.
Scalability	Widely benchmarked, especially for microservices; hybrid strategies are rare.	To design hybrid or staged scaling models and to evaluate their effectiveness in real-world scenarios.
Migration Readiness	Strong presence in legacy transformation works.	To model phased migration paths and to validate modular monoliths as a viable intermediate architecture.
Deployment Strategy	Well-covered in microservices; modular monoliths receive limited attention.	To analyze streamlined deployment approaches and to propose tooling practices optimized for modular monoliths.
Complexity Management	Acknowledged in most papers but rarely quantified or modeled.	To propose quantifiable strategies for managing complexity and cognitive load across architectural styles.
Performance Implications	Covered in about half the studies; modular monoliths are under-tested.	To conduct empirical benchmarks that measure performance trade-offs between modular monoliths and microservices.
Team Fit	Rarely analyzed directly; mostly inferred from architectural discussions.	To quantify how architectural decisions influence collaboration, communication, and team structure.
DevOps Maturity	Common in cloud-native and microservices literature.	To compare DevOps tooling and practices across architectures, with particular attention to early-stage teams.
Onboarding	Largely absent; only anecdotal mentions in modular monolith studies.	To evaluate onboarding processes and developer ramp-up experiences under different architectural styles.
Practicality of Adoption	Mentioned conceptually, rarely operationalized.	To bridge academic insights with actionable guidance and decision-support frameworks for practitioners.
Target Context	Often missing or broadly defined.	To clarify the applicability of architectural recommendations across distinct organizational scenarios.

tion can create silos and incoherent products if left unmanaged. Balancing requires intentionally shaping the relationship between organizational and technical structures to preserve modular coherence, encourage cross-functional collaboration, and sustain adaptability. Guideline *G8 (Foster SRE and DevOps Maturity)* highlights the integration of cultural and technical practices that connect software engineering with site reliability engineering, ensuring that system reliability, performance, and deployment pipelines are treated as shared responsibilities. This maturity enhances delivery speed while safeguarding stability through monitoring, automation, and continuous feedback loops. Guideline *G9 (Support Frictionless Onboarding)* underscores the importance of designing systems and workflows that reduce barriers for new contributors, enabling them to quickly gain autonomy and contribute productively with minimal technical friction.

The *Guideline Orientation* dimension refers to high-level recommendations that drive design decisions without imposing rigid rules. Guideline *G10 (Apply Actionable Design Patterns)* emphasizes the use of pragmatic and well-established architectural patterns that can be directly implemented in practice, ensuring that theoretical concepts are translated into concrete solutions. Guideline *G11 (Promote Contextual Adaptation)* highlights the need to tailor architectural guidelines to the specific organizational and business environment, recognizing that no single ap-

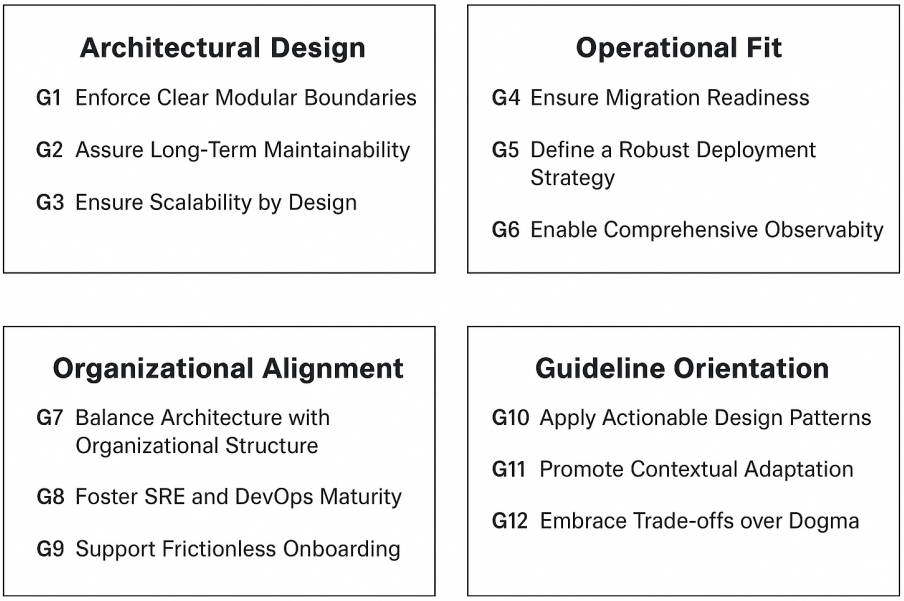

Figure 1: Four guideline dimensions for evaluating modular monolith architectures.

proach universally fits all scenarios. Guideline *G12 (Embrace Trade-offs over Dogma)* stresses that architectural design inherently involves balancing competing forces, encouraging engineers to evaluate dilemmas critically rather than adhere to rigid prescriptions.

Each dimension group provides a set of guidelines that, together, form the foundation for the proposal of this research. It illustrates how technical, operational, organizational, and orientation-based considerations should be integrated when selecting or evolving a modular monolith architecture in software startups. What follows is a descriptive presentation of the guidelines within each dimension, starting with those that address the architectural design of modular monoliths.

We aim to structure each guideline according to a template with the following topics: *Conceptual Overview*, *Objectives*, *Key Principles*, *Metrics and Verification*, *Tool Capabilities*, and *Literature Support Commentary*. Here we outline what is expected in the template, taking the G1 guideline (Enforce Clear Modular Boundaries) as a representative example.

The *Conceptual Overview* topic aims to explain the purpose of the guideline. Considering G1, it should define clear boundaries around each module so that each module encapsulates its internal implementation, dependencies between modules occur only through explicitly declared interfaces, and unintended coupling is prevented (allowing independent evolution of modules).

The *Objectives* topic outlines what a guideline should ensure. Regarding G1, it is expected that this guideline assures, for instance, encapsulation (to keep internal classes, data, and resources hidden within a module) and early verification (to detect boundary violations at build or test time rather than at runtime).

The *Key Principles* topic defines the directives to implement a guideline. For G1 these include inter-module communication via synchronous public-interface calls or automated asynchronous event-driven channels and traceability by recording boundary changes in an Architectural Decision Record.

The *Metrics and Verification* topic addresses what to measure in the application to verify the accomplishment of a guideline. For example, an interesting metric for G1 is the boundary violation count, which is the number of occurrences where a module references a class or resource in another module without a declared dependency. Another metric related to G1 can be the verification failures, in a way to count the build/test failures triggered by automated boundary checks.

The *Tool Capabilities* topic refers to the auxiliary tools that can aid in the implementation of the guideline. Regarding G1, it is suggested that the software tool (open-source or proprietary) used to enforce modular boundaries should support, for example, module discovery (to automatically identify modules via conventions or explicit configuration), and isolated testing (to allow tests to load only a given module and its declared dependencies, failing early if the module tries to wire code from undeclared modules).

The *Literature Support Commentary* topic aims to

explain the support of the literature behind the guideline. For example, this topic for G1 should explain that although modularity is acknowledged as essential, most research treats it as an outcome of refactoring rather than a design criterion (Prakash and Arora, 2024; Su et al., 2024). Few works propose proactive structural mechanisms for enforcing modularity in monoliths (Montesi et al., 2021; Tsechelidis et al., 2023), and they lack empirical validation. This gap underscores the necessity of G1 to provide clear and actionable steps to define and verify module boundaries to fill a critical void in both theory and practice.

By presenting Guideline *G1 (Enforce Clear Modular Boundaries)* in detail, the proposed template has been illustrated as a structured path to capture both conceptual and practical aspects of each recommendation. G1 serves as the first worked example, demonstrating how objectives, principles, verification mechanisms, tool support, and literature insights can be articulated in a consistent manner. The remaining guidelines (*G2–G12*) are introduced at a conceptual level and will be progressively developed and refined throughout the course of this research.

5 INITIAL CONCLUSIONS

Over the past decade, advances in cloud computing, open-source ecosystems, and software development tooling have significantly reduced the cost and complexity of launching digital products. These enablers have contributed to a surge in software startups capable of reaching global scale and onboarding thousands or even millions of customers within days or months, rather than the years or decades typically required for traditional businesses to mature. This acceleration, however, also introduces new forms of complexity that challenge sustainability and long-term adaptability.

Startups in particular operate under critical resource constraints and must validate their business models quickly, facing conditions that amplify the long-term consequences of early technical decisions. Among these, the choice of software architecture plays a pivotal role in determining a product's capacity to evolve, scale, and remain maintainable over time.

Our proposal introduces a set of initial guidelines for modular monolith architectures in cloud-native ecosystems that enforce clear module boundaries, incremental scalability, operational fit, and organizational alignment, all aimed at reducing complexity while preserving the option to extract services when real demand emerges. At this stage, these guidelines serve as a conceptual foundation; their detailing and implementation guidance will be progressively developed and refined throughout the course of this research. To advance from this conceptual stage toward practical applicability, several important steps remain.

First, validation of our proposal with industry practitioners is essential. We plan to engage specialists from companies that have successfully adopted modular monoliths, such as GitHub and Shopify, as well as engineering teams that continue to struggle with highly coupled legacy monoliths applications. Through expert interviews, case studies, and hands-on experiments, we will gather feedback to refine the guidelines, confirm their effectiveness, and ensure they address real-world pain points.

Second, we must test the applicability of these guidelines in real startups to measure how much they contribute to early architectural decisions, how they support initial product definitions, and how they facilitate the extraction of modules into independent services. In addition, we should evaluate alternative research paths: whether to focus on a smaller set of core guidelines with greater depth of analysis, or to present the full catalog of recommendations and assess each. Determining which approach delivers the greatest practical value for engineering teams operating under time and resource constraints is a key objective.

These research activities and experiments are necessary to move from theoretical heuristics to a validated set of guidelines that startup engineers can adopt with confidence. By combining expert validation with controlled pilot projects, we aim to produce both actionable patterns and empirical evidence that modular monoliths can serve as a pragmatic architectural foundation for cloud-native software startups.

REFERENCES

Alshuqayran, N., Ali, N., and Evans, R. (2016). A systematic mapping study in microservice architecture. *Proceedings of the 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–9.

Amazon (2023). Scaling up the prime video audio/video monitoring service and reducing costs by 90%. Accessed: 2025-05-12.

Arya, S., Chauhan, D., Tanishq, Anand, S., and Sharma, O. (2024). Beyond monoliths: An in-depth analysis of microservices adoption in the era of kubernetes.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, MA.

Berry, V., Castelltort, A., Lange, B., Teriihoania, J., Tibermacine, C., and Trubiani, C. (2024). Is it worth mi-

grating a monolith to microservices? an experience report on performance, availability and energy usage.

Blank, S. and Dorf, B. (2012). *The Startup Owner's Manual: The Step-by-Step Guide for Building a Great Company*. K&S Ranch Press.

Blinowski, G., Ojdowska, A., and Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation.

Crowne, M. (2002). Why software product startups fail and what to do about it. evolution of software product development in startup companies. In *Proceedings of the 2002 IEEE International Engineering Management Conference (IEMC '02)*, volume 331, pages 338–343, Cambridge, UK. IEEE.

Dragoni, N., Lanese, I., Larsen, S., Mazzara, M., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pages 195–216.

Foote, B. and Yoder, J. (1999). Big ball of mud. In *Pattern Languages of Program Design*, volume 4. Addison-Wesley, Boston, MA.

Ford, N., Parsons, R., and Kua, P. (2017). *Building Evolutionary Architectures: Support Constant Change*. O'Reilly Media, Sebastopol, CA.

Fowler, M. (2020). What does cloud-native mean? Accessed: Accessed: 2024-11-26.

Fowler, M. and Lewis, J. (2014). Microservices: A definition of this new architectural term. Accessed: Accessed: 2024-11-26.

Fritzsch, J., Bogner, J., Wagner, S., and Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490. IEEE.

Gysel, M., Kölbener, L., Giersche, W., and Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing (ESOCC 2016)*, volume 9846 of *Lecture Notes in Computer Science*, pages 185–200. Springer.

Johnson, J., Kharel, S., Mannamplackal, A., Abdelfattah, A. S., and Cerny, T. (2024). Service weaver: A promising direction for cloud-native systems? *arXiv preprint*, arXiv:2404.09357. Preprint.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, EBSE Technical Report. Available online.

Lauretis, L. D. (2019). From monolithic architecture to microservices architecture.

Montesi, F., Peressotti, M., and Picotti, V. (2021). Sliceable monolith: Monolith first, microservices later.

Ng, T., Rawi, A. A. B., Sum, C. S., Tso, E., Yau, P. C., and Wong, D. (2024). Migrating from monolithic to microservices with hybrid database design architecture.

Prakash, C. and Arora, S. (2024). Systematic analysis of factors influencing modulith architecture adoption over microservices.

Ravikant, N. (2020). *The Almanack of Naval Ravikant: A Guide to Wealth and Happiness*.

Ries, E. (2011). *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business.

Segment (2023). Goodbye microservices. https://segment.com/blog/goodbye-microservices/. Accessed: 2024-11-26.

Shopify (2022). Deconstructing the monolith. https://shopify.engineering/deconstructing-the-monolith. Accessed: 2024-11-26.

Su, R., Li, X., and Taibi, D. (2024). From microservice to monolith: A multivocal literature review. *Electronics*, 13(8):1452.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2018). Defining microservices architectural style. *Proceedings of the IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 74–78.

Tilkov, S. (2015). Don't start with a monolith. https://martinfowler.com/articles/dont-start-monolith.html. Accessed: 2024-11-26.

Tsechelidis, M., Nikolaidis, N., Maikantis, T., and Ampatzoglou, A. (2023). Modular monoliths the way to standardization.