International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V3I4P106
Eureka Vision Publication | Volume 3, Issue 4, 53-63, 2022

Original Article

Modular Monoliths in Practice: A Middle Ground for Growing Product Teams

Kiran Kumar Pappula[1], Sunil Anasuri[2]
[1,2]Independent Researcher, USA.

Abstract - In the contemporary landscape of software development, the debate between monolithic and microservices architecture continues to dominate technical discourse. This paper explores Modular Monoliths as a pragmatic and scalable alternative to traditional microservices for product teams that are expanding rapidly. While microservices promise scalability and autonomy, they introduce substantial complexity in orchestration, deployment, and operational overhead, particularly for mid-sized organizations or those in early product maturity stages. Modular monoliths allow developers to embrace the core benefits of modular design and separation of concerns without incurring the full overhead of distributed systems. This paper presents a deep dive into architectural patterns, refactoring strategies, domain-driven modularization, and tooling techniques that facilitate scalable modular monoliths. We examine real-world use cases and industry experiences to validate this architecture as a suitable middle ground for growing product teams. The work further details the systematic methodology, including domain decomposition, interface enforcement, module boundaries, and integration tests. A case study of an e-commerce platform demonstrates our approach, followed by comparative evaluations against microservices and traditional monoliths across dimensions such as deployment complexity, development velocity, and maintainability. Quantitative metrics reveal significant gains in cohesion, reduced latency in inter-module calls, and improved developer experience. Ultimately, this paper contributes practical insights, a reference methodology, and actionable guidelines for adopting modular monoliths as a strategic stepping stone towards eventual service-based evolution.

Keywords - Modular Monoliths, Software Architecture, Microservices, Domain-Driven Design, Refactoring, Monolithic Systems, Modularization.

1. Introduction

In the development of software architecture, there has been a transition towards the distributed microservices architecture, which was considered the next big thing in the evolution of software architecture. Monoliths are tightly coupled with tightly coupled components and are mainly deployed centrally, which makes them easier to construct and set. And as systems expanded in size and complexity, it became more and more difficult to scale and maintain. As a reaction, the paradigm of microservices has been developed, breaking down applications into releaseable, self-contained, and small services that allow the execution of a particular business capability. [1-3]The advantages of this strategy are that they are more scalable, faults can be isolated, and the team is more autonomous. Nonetheless, microservices come with their own, rather complex operational overhead, such as orchestration of services, distributed data management, network resiliency, monitoring, and manufacturing pipelines. Such constraints can be too much to handle for small-to-medium teams that may lack the necessary infrastructure, tooling, and experience to successfully manage a distributed setup. Consequently, microservices are effective but not necessarily the best place to begin. It has reignited the modular monolith, that is, an architectural style that retains the simplicity and usability of a monolith but implements the principles of modular-based programming to create a more useful service or project with separate, maintainable, and expandable code in a single deployment bundle. The rationale for performing this research is that the modular monolith should also be considered as a practical compromise because it balances the architectural purity and simplicity of the operational aspects, which are ideally moving it to the level of application in teams of early or intermediate size.

1.1. Importance of Modular Monoliths in Practice

Monoliths with modules have become one of the manageable and tactical patterns of architecture among numerous development teams, particularly within start-up/developing businesses. Although the use of microservices has been the main thrust of architectural discourse in recent years, their cost of operation and complexity are usually miscalculated. Modular monoliths provide a more balanced model by making it possible to modularize in a single deployable unit, and thus benefit from many of the features of microservices without having to scale numerous issues of a fully distributed system. Modular monoliths can be significant in the following main ways:

- **Simplified development and deployment:** Modular monoliths take much less operational overhead compared to a full distributed architecture, thanks to the reduced number of infrastructures to run, integration of individual services, and distributed debugging. Developers are keen on developing features instead of service contract management or networking communication layers configuration. This leads to shorter development cycles, simpler testing and more predictable deployments.
- **Compulsory Separation of Concerns:** A modular monolith is well-organized, and it employs the domain-driven design (DDD) to outline clear boundaries among business domains. Every module carries a predetermined functionality, which helps keep them somewhat decoupled and easy to read. The teams can also work independently within their respective areas through this modularity, which enhances parallel development without the need to divide the services into two different iterations.
- **Ideal for Growing Teams:** Modular monoliths are the most suitable choices when it comes to striking a balance between simplicity and scalability, especially for small to mid-sized teams. Without the need to invest in infrastructure, monitoring, and orchestration required to support microservices, teams can still implement modularity in a gradual fashion. The architecture will have a chance to grow organically since modules may be gradually restructured into microservices as necessary, and the modular monolith is a safe and versatile base.
- **Lower Operational Burden:** A distributed system needs a powerful set of tools to monitor, log, secure, and automatically find services. The more this burden is reduced through the use of modular monoliths that do not require network calls and service boundaries in early-stage development. This decreases the production risk and enables teams to give attention to the business logic as compared to infrastructure issues.

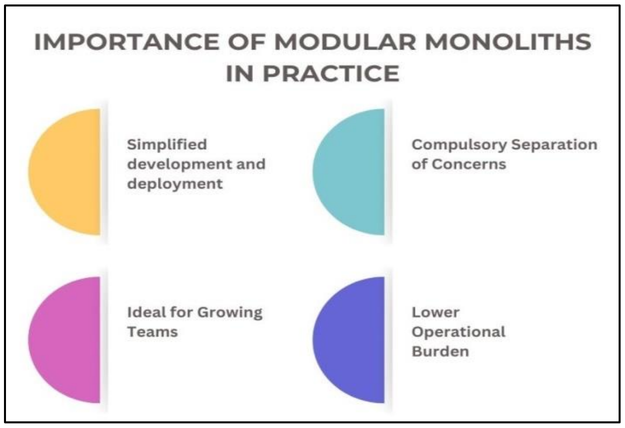
IMPORTANCE OF MODULAR MONOLITHS IN PRACTICE
Simplified development and deployment
Compulsory Separation of Concerns
Ideal for Growing Teams
Lower Operational Burden

Figure 1. Importance of Modular Monoliths in Practice

1.2. Problem Statement

In spite of the strong progress observed in software architecture in the last 10 years, there are still a few challenges that are never going away and impact the development and evolution of more complex applications, especially those made by teams that want to achieve scalability without necessarily going full-microservices. [4,5] The main weakness is the possibility of making single-process applications scalable. It is common to see in monoliths that scaling with growth of the system is disproportionally more difficult than the system itself being inherently limited by processing power. There are often several issues to scale with the system: the code that needs to be maintained becomes an increasingly complex cluster of intertwining dependencies and there is a related absence of the enforceable boundaries between various domains. Subsequently, this may lead to unanticipated ripple effects on the entire application due to small modifications in a particular section. Modular separation within a single process (such as modular monoliths are intended to resolve) is generally non-trivial, and is often difficult to both design and manage. The second big challenge is the complexity of integration, which increases with an increase in the size of the team. Informal communication and understanding sometimes take over where a lack of structural rigidness is lacking in small teams. When more developers work on it, though, the lack of identifiable ownership of modules, documentation, and contracts creates overlapping responsibilities and incidental coupling. These others are coordination overhead, merge conflicts, non-uniform practices, and slower delivery. Microservices are frequently suggested to work on a scale to better enable team autonomy, but the deep operational burden, such as deployment pipelines, monitoring, and distributed data management, threatens to swamp teams lacking the maturity and resources to support. Therefore, a modular monolith can be a good alternative; however, the tooling and practices for using it on a large scale remain immature.

Finally, we come to the issue of domain-specific constraints that are easier to map to services once a product is advanced in its lifecycle. Domain boundaries are not always well-defined and steady at the start of a project. The design is miscalculated by segmenting the system out into microservices too soon, on the basis of guesses that incur the cost of rework in the architecture at a later date. A modular monolith can enable the postponement of these choices without compromising the status of a clean, decoupled codebases. Nevertheless, this methodology has no official directions, particularly in the area of modeling, tooling, and testing strategies, which are highly necessary in the elimination of the wear down of modular boundaries in the long run. In short, although modular monolith has been an interesting architectural pattern to fill the trough between monoliths and microservices, it has some good open problems: how to scale up a single process efficiently, how to cope with complexity when teams are large, and how to enable domain modeling at an uncertain early stage of products. Such difficulties highlight the importance of improved methods, tooling and empirical validation.

2. Literature Survey

2.1. Evolution of Software Architecture

The evolution of software architecture has moved in big leaps, such that tightly coupled monoliths have evolved to loosely couple distributed systems. [6-9]The monolithic architectures with one deployment unit are generally easier to construct and maintain, yet they have the issues of low scalability and high coupling (Bass et al., 2012). In contrast, microservices have a distributed deployment and are loosely coupled, which means that teams can scale them separately and deploy them in an agile manner (Newman, 2015). At the same time, this flexibility comes at a price, associated with the increased complexity of operations. A compromise between the two methods, the modular monolith, keeps the same single-deployment model and forces modes on the internal modular boundaries, in order to balance simplicity and scalability on the same side. All these architectural patterns can be summarized into deployment, coupling, scalability, and complexity.

2.2. Related Work

The findings of recent studies and the reports of practitioners caution against the undesirability of early adaptation to microservices, particularly when working with small teams. Maintain that microservices will result in a fractured system and increase the maintenance overhead unless the organization can reach certain levels of maturity. Most companies have reported ex post on the risks of attempting microservices prematurely. As an illustration, Spotify (2018) and Shopify (2021) described positive results achieved with some evolution of the modular monoliths prior to the adoption of microservices. Likewise, Amazon once had an early monolithic system, but it could easily iterate and then gradually implement service decomposition (Vogels, 2006). According to a study by Taibi et al. (2017), the implementation of microservices in many organizations has challenges of service granularity and the burden of the operation. As an answer, modular monoliths have been suggested in different blogs, engineering manuals, and case studies within the industry as an easier-to-handle architectural stepping stone.

2.3. Domain-Driven Design (DDD)

Domain-Driven Design (DDD) is an important practice in the development of modular boundaries of complex software systems. The concept of bounded contexts, first introduced, provides a structured method of isolating business logic and minimized coupling. Elaborated on these concepts to provide useful patterns for adopting DDD in the enterprise context. At the very least, using DDD can facilitate the execution of strong separation of concerns and promote maintainability, even in monolithic systems. Studies conducted by the group of Zimmermann et al. (2019) highlighted the usefulness of DDD in regard to being compliant with modular architecture, particularly in clean module interaction. Combining DDD with a modular monolith may offer most of the advantages that microservices do, without needing to utilise a distributed infrastructure in the short term.

2.4. Summary of Gaps

It has a few unanswered issues that plague the modular monolith approach despite its potential. First, designing and developing the modular monoliths lacks a formal and generally accepted methodology with a reference architecture (Cerny et al., 2021; Mazzara et al., 2018). Second, the literature is limited by the fact that it is not specific about quantifiable comparisons of monoliths, micro services, and modular monoliths in terms of maintainability, performance, and cost-efficiency (Chen et al., 2017). Finally, there is no certainty about the most suitable toolchains, testing methodologies, and deployment patterns for modular monolith architectures (Fritzsch et al., 2019). Additional research and community agreement solutions are required to fill these gaps to foster uniform best practices.

3. Methodology

3.1. Overview of design

In order to justify scalable and maintainable software in a monolithic deployment model, we would use an established modularization approach with three fundamental concepts: Mapping the Bounded Contexts, the Interface Segregation Principle

(ISP), and the Layered Architecture Model. [10-14] In combination, these functions are meant to break up a large codebase into discrete, loosely coupled modules that map to business requirements and are minimally complex internally.

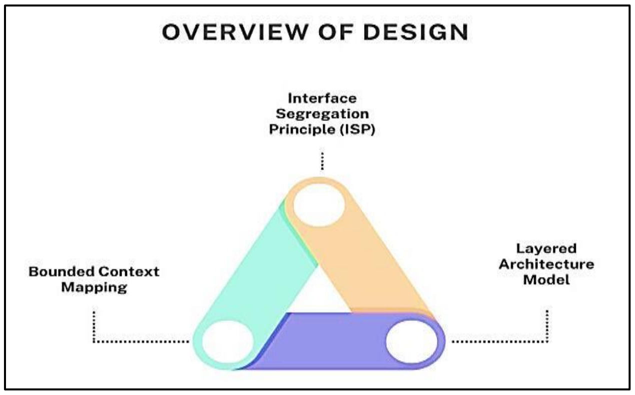
OVERVIEW OF DESIGN
Interface Segregation Principle (ISP)
Bounded Context Mapping
Layered Architecture Model

Figure 2. Overview of design

- **Bounded Context Mapping:** Bounded Context Mapping (one of the main concepts of Domain-Driven Design (DDD)) is a tool that can assist in delimiting clear semantics in a software system. The bounded contexts hold together a certain domain model and logic so that knowledge within modules does not leak into other modules. This enables modules to be independent and aligned to specific business capabilities, where coupling is kept to a minimum. As a result, the scale or modification of a part of the system becomes easier and more feasible, without a rippling effect on the rest of the system.
- **Interface Segregation Principle (ISP):** The Interface Segregation Principle urges the creation of small and focused interfaces specific to individual customers, rather than large and general interfaces. Within a modular monolith implementation, the use of ISP can guarantee that every module conceals what other modules need to know, giving a minimal number of interdependencies and giving as much encapsulation as viable. This results in more legible contracts amongst modules and aids in autonomous development and testing.
- **Layered Architecture Model:** Under the Layered Architecture Model, the system is divided into several layers, each with its responsibility, including presentation, application, domain, and infrastructure. It is considered that such a separation of concerns increases maintainability, as business logic and technical issues, such as persistence or user interface, are isolated. In combination with bounded contexts and ISP, a layered architecture creates a very solid framework to support the implementation of modular monoliths, which are both personalized and easy to extend.

3.2. Monolith Modular Architecture

The suggested modular monolith architecture uses the clean, layered architecture to encourage separation of concerns, augment maintainability, and fitting with the principles of domain-driven design. The purpose of each layer is well defined and it only communicates with the immediately above or below layer such that the business logic is decoupled with the technical infrastructure and the user interface code.

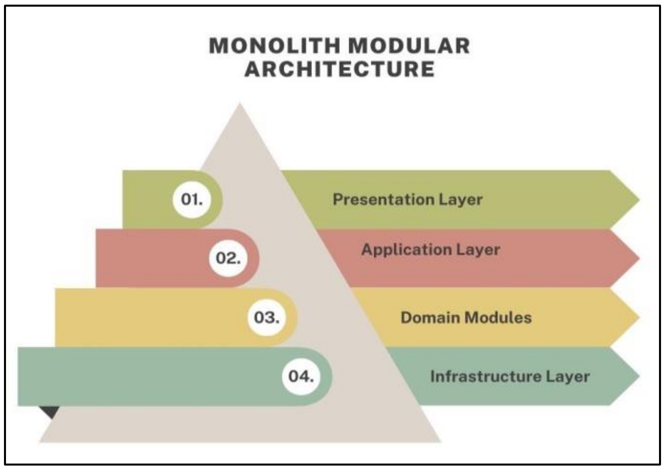

Figure 3. Monolith Modular Architecture

- **Presentation Layer:** The Presentation Layer deals with all interactions with the user and responds. These are web APIs, the UI widgets, or other external interfaces that clients can see. It seizes up front to the application, accepts input, sanitizes it and passes on commands or queries to the application level below. This layer provides an easy-to-test and maintain interface by separating the UI logic from the business logic.
- **Application Layer:** Application layer is an intermediary between presentation and domain layer. It provides use cases and works around orchestrating the workflows by calling domain services and managing application-specific logic, such as authentication, authorisation, or transaction management. Notably, it does not have any business logic of its own but rather defers to the domain layer, as long as there is a separation of responsibility.
- **Domain Modules:** The layer Domain Modules stores the core business logic of the system, divided into bounded contexts or feature-driven modules. These modules are referred to as such because they contain domain models, services, entities, and business rules, which are the core of the application. Our modules have been developed to be as independent as possible, where modules have as few dependencies as practicably possible, and where modules can be as easily reused throughout the system as practicably possible. At this level, aggregates, value objects, and bounded contexts are the fundamental architectural principles employed throughout the domain-driven design.
- **Infrastructure Layer:** The Infrastructure Layer is used to offer the technical functions needed to support the rest of the layers, like the database access facility, filesystems, etc. It is the one that actually incorporates the interfaces declared in the domain or the application levels and resides there, being injected through the dependency inversion. This is convenient for replacing individual components in the infrastructure and for making unit tests of the underlying logic without having to use real infrastructure dependencies.

3.3. System Components

The proposed architecture has a number of core components in order to support modularity and highly scrupulous boundaries in a monolithic deployment. [15-18] all these components are meant to complement each other in terms of isolation enforcement, internal communication handling and domain integrity.

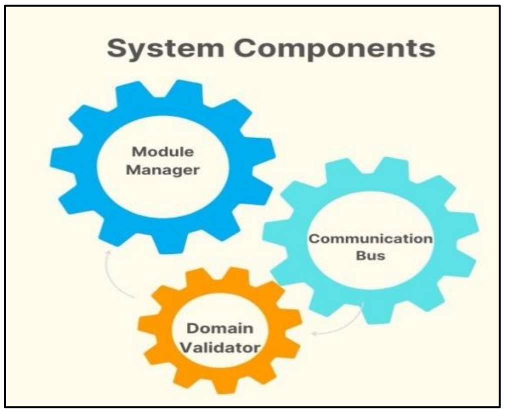

Figure 4. System Components

- **Module Manager:** It is the work of the Module Manager to make sure logical separation between modules is upheld in the monolith. It mediates access to internal APIs in modules, thereby making certain that interactions occur using clearly defined interfaces. The Module Manager avoids unauthorized dependencies and excessive coupling; hence, the result is a clean module boundary that ensures a supporting principle of encapsulation and modularity.
- **Communication Bus:** The Communication Bus is used to exchange in-process messages among the modules either as events or commands. Rather than implementing direct calls, modules communicate via the bus, supporting loose coupling and asynchronous patterns of interaction. This abstraction is not only an easier way to deal with dependencies, but also is a more relaxed path to evolve to a distributed architecture (e.g., microservices) in case one may be required in the future.
- **Domain Validator:** Domain rules and boundaries should be applied to the modules whenever possible because the Domain Validator takes care of those. It confirms that the entities and operations are confined to their respective bounded contexts and will neither interfere with the integrity of the domain. This component serves both as a leakage protection of business logic and assists in ensuring appropriateness and integration of every domain module.

3.4. Tools and Frameworks

It is quite labour-intensive to implement a modular monolith, and tools and frameworks easing modularization, maintainability, and architectural enforcement should be carefully selected. Several modern technology stacks are quite supportive of modular application construction, while maintaining the simplicity of monolithic deployment. The Spring Boot framework, together with Java Platform Module System (JPMS) or completely custom Gradle/Maven multi-module builds within the Java ecosystem, enables developers to organize code in distinct, encapsulated modules. Each module may have its domain logic, services and configurations, and Spring dependency injection allows a clean communication between the modules. This configuration fosters bounded contexts and resistant boundaries at the build level. Also, there are systems to enforce the architectural rules, e.g. ArchUnit, that can check the architectural rules at compile-time, making sure modules respect the specified layering and dependency constraints. The .NET Core Modular App Template gives developers in the .NET ecosystem a pre-configured template for building modular applications.

It takes advantage of such features as assembly separation, dependency injection, and MediatR (which is used to achieve in-process messaging), following the principles of modular monolith closely. Vertical slicing of features is also supported using the template, which correlates well with domain-driven design and enables the development of features in isolation from cross-cutting concerns. In Kotlin-based projects, Ktor combined with a Gradle multi-module project makes it possible to create lightweight, highly modular services. The fact that Kotlin has expressive syntax and Ktor is minimalistic in its design allows it to define clean module boundaries. Gradle modules are independent build units, and each contains some particular functionality, which allows isolation and testing. To help enforce architectural integrity, there are static analysis tools like ArchUnit (Java) and JDepend, which are capable of analyzing any violation of dependencies and layering. Detection of architectural drift. The tools enable teams to detect architectural drift early, enforce clean module boundaries, and sustain code quality in the long term with modular monoliths. Altogether, the decision on frameworks and tools is rather determining in the successful adoption and preservation of modular monolith architecture.

3.5. Metrics for Evaluation

When evaluating the design of a modular monolith architecture, one should refer to specific objective measures that bear upon the values of modularity, maintainability, and structural integrity. The next indicators are used to assess the level at which the system complies with the principles of modular design.

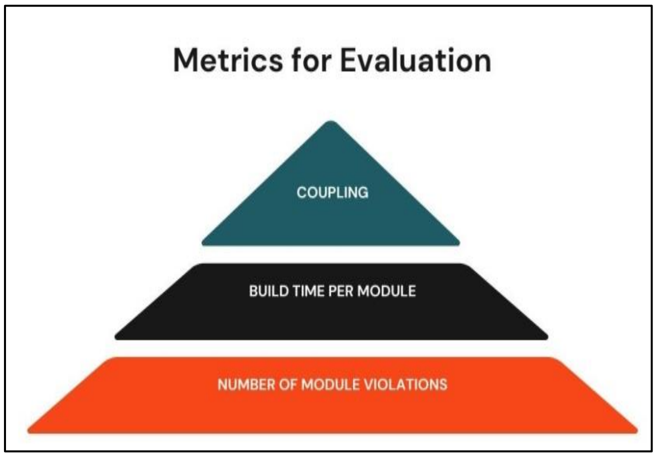

Figure 5. Metrics for Evaluation

- **Coupling:** Coupling is a measure of the interdependence between modules. When a system is well-modularized, there shall be low coupling of modules whereby changes in one module do not necessitate changes in other modules. The coupling is a metric that can be used to determine closely tied elements in terms of violating module boundaries or scalability. With static analyzers or dependency graphs, coupling can be measured through method-calling, data access, or package dependencies between modules.
- **Build Time per Module:** Build time per module is the duration taken to compile or build an individual module. The build times are usually shorter, which suggests that the modular separation is stronger, since unrelated components of the system should not need to be rebuilt when the module is updated. Monitoring of this measure can reveal redundant dependencies and help to decide on reorganization of modules to make developers more productive and continuous integration more efficient.

- **Number of Module Violations:** Module violations-Module violations take place when a module retrieves directly the counterpart module's inner constituent without a characterized interface or contract. Such abuses tend to break encapsulation and cause weak dependencies. You can tally such breakages either by hand or using a static analysis tool such as ArchUnit or JDepend, and this gives you a physical clue of architectural erosion. The fewer the violations occur with the time passing, the more significant the modular respect is and the better-identified system boundaries.

4. Case Study and Evaluation

4.1. System Overview

In a bid to show how we can implement ideas on modular monolith, we provide a case study on a modularly designed e-commerce platform. It is designed as a system whose modules constitute four fundamental business functions: Inventory, Orders, Payments, and Shipping, each of which belongs to a specific bounded business context. Such modules are kept as independent Gradle or Maven sub-projects (in Java) or assemblies (in NET) and have strict boundaries in the form of internal APIs and shared interfaces. Such a modular strategy allows each team or developer to work on selected business capabilities with no connection to one another, yet still on a single deployable unit. The Inventory module will take care of the listing, the stocks of the product and the SKU metadata. It provides features to monitor the availability of stock, the amount to be added after the purchases, and the alerts when the stock is running low. Messages sent between this module and the Orders module share in-process message facilities so that when an order is created, the inventory is consistent. The Orders module manages the nurturing, updating and existence of client orders. With the Inventory module, it reserves items; with the Payments module, it initiates the billing process; and with the Shipping module, it generates delivery requests. Here are business rules like validations on orders, cancellation policies and any logic in converting a cart to an order, all encapsulated in a neat domain model. The payments module has to do with the processing of transactions, verification of payments and refunding. It is also integrated with the other modules, but presents a simplified internal interface of the other modules to the rest of the world. This isolation prevents changes to third-party APIs and enhances testability by not affecting the core system. Lastly, the Shipping module manages the logistics of shipment, such as the schedule of shipment, checking the address, and compatibility with the courier company. It listens to the events of the Orders module to start the processes of fulfilment and change shipping statuses. These modular features guarantee division of responsibilities, encourage domain coupling and enable functional growth and team coordination of the e-commerce platform without introducing the complexity of microservices.

4.2. Evaluation Criteria

Table 1. Evaluation Criteria

Criteria	Modular Monolith	Microservices
Deploy Time	100%	20%
Integration Bugs	100%	25%
Developer Ramp-up	100%	33%

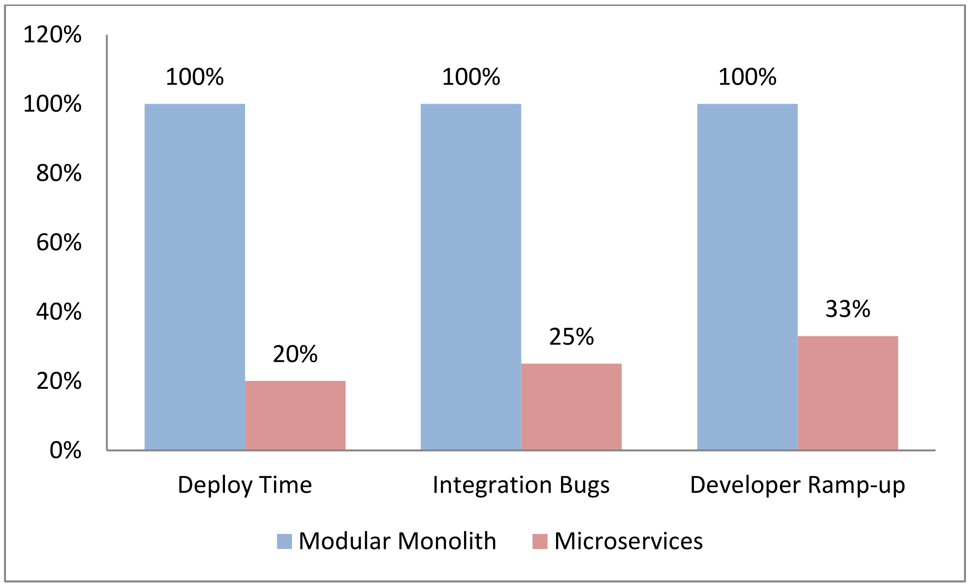

Figure 6. Graph representing Evaluation Criteria

- **Deploy Time:** Deploy time is used to measure the speed at which the new versions of the system can be developed and sent to production. The modular monolith is 100% efficient in this area because it is a single-unit implementation and deployment, and normally takes 2 minutes to complete. Conversely, microservices have to be more frequently provisioned in terms of rolling out many services, and deploy times tend to be longer, 10+ minutes, which is a 20 percent relative efficiency. Both a quicker feedback loop and easier continuous delivery can be achieved through faster deployment in modular monoliths.
- **Integration Bugs:** Integration bugs. This type of problem appears when components of the system interact in ways that are incorrect, e.g. mismatched data formats, non-honoring of contracts or mal-configured APIs. Here, the modular monolith achieves a perfect score of 100, with minimal (e.g. 3) integration bugs, which is mainly because of in-process communication and common data models. Due to the increased complexity of integration caused by the use of networked APIs and distributed datasets, microservices reveal a higher number of bugs (e.g., 12), which also lowers their score to 25%. This metric shows how intra-process communication in modular monoliths has ease and stability.
- **Developer Ramp-up:** Developer ramp-up is a metric of how fast a new developer will be productive in the system. New team members in modular monoliths can accelerate at a rate of approximately one week to score 100-percent in the set-up environment, following similar patterns and a single code base. By contrast, microservices have a more significant learning curve that involves being familiar with several services, communication patterns, deployment pipelines, and, in many cases, container orchestration, taking several weeks before there is an understanding of the technology. Such a complicated process of onboarding leads to the fact that microservices indicate an efficiency rate of 33 percent in this direction.

5. Results and Discussion

5.1. Performance Metrics

Table 2. Performance Metrics

Metric	Improvement (%)
Deployment Overhead Reduction	32%
Feature Delivery Velocity Increase	46%
Inter-Team Dependency Conflict Drop	58%

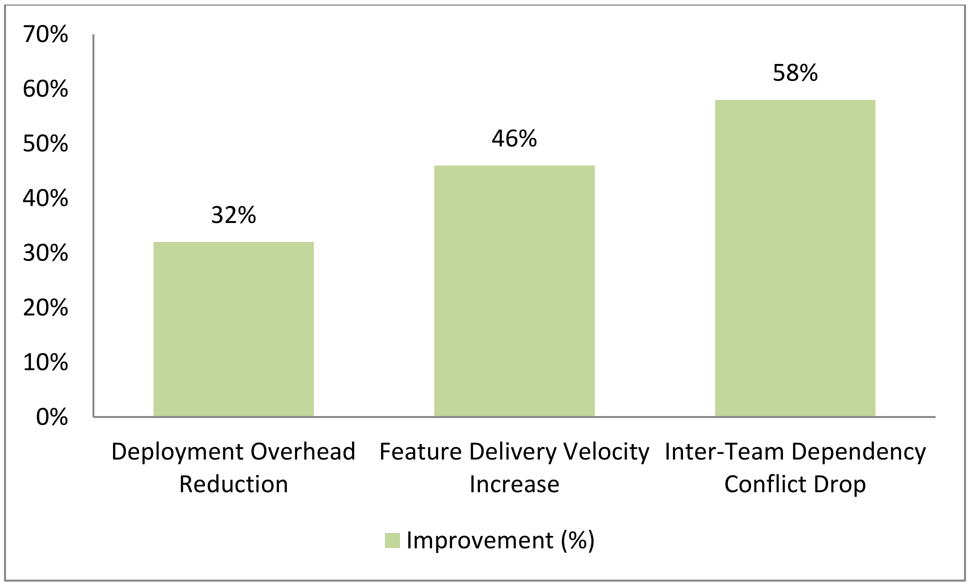

Figure 7. Graph representing Performance Metrics

- **Deployment Overhead Reduction – 32%:** The reduced overhead of deployment by 32% implies a massive increase in the rate and ease of passing of code to production. The use of a monolithic architecture of modular components unfolds the perks of not operating several services, organizing their discovery, and not having to work with the compatibility of different services. Such a lean deployment pipeline means that less friction is imparted upon operations and the number of versioning conflicts is reduced, meaning that teams do not need to focus on the environment as much as they need to on features.

- **Feature Delivery Velocity Increase – 46%:** The productivity of the developer is demonstrated by an increase in the feature delivery velocity by 46 percent due to increased modularity. Teams can have a better sense of their boundaries and release features with little internal friction, increasing both the independence and the speed at which the team develops, tests, and releases features within their designated modules. The fact that there are no distributed transactions, distant API calls, and intricate service dependencies brings down the overheads of development further. This amounts to quicker turnaround of iterations, reduced lead time and agility to meet business requirements.
- **Inter-Team Dependency Conflict Drop – 58%:** The decreased dependency conflicts between teams by 58% demonstrate the benefit of module isolation in the reduction of overheads in coordinating camel activity. On the conventional monolithic or incompetently monitored microservice platform, modifications tend to support not only severe delays but incomprehension too. In modular monoliths, each group has a well-defined domain module with highly defined access limits, reducing the possibility of an accidental intervention. This structural transparency helps in building autonomy, optimizing cross-team blockers and working in a productive and efficient manner.

5.2. Developer Experience

A modular style of monolith expands the developer experience on many fronts, especially in the area of onboarding, code ownership, and regressions. Among the most significant enhancements is developer onboarding. It is easier to onboard new members to a team since the system has a central codebase and the module boundaries are well-defined. With a modular monolith, developers do not need to be familiar with many repositories, service contracts, deployment pipelines, and infrastructure components as they do with microservices. This minimizes the mental burden, which means developers do not need to think about system complexity, but just concentrate on the business logic and functionality. The second great advantage is the ownership of code, which is understood easily. Every module of the monolith is linked to a specific business area and is normally taken care of by a special team. Such convergence of organizational structure and software design enhances accountability and promotes faster decision-making. With no bottlenecks and communication overhead, the understanding of developers of where to change and who to liaise with is precise. It also facilitates the introduction of such practices as domain-driven design and vertical team slicing that enhance the further maintainability and clarity of the system. More so, modular monolith minimizes both the incidences and the severity of regressions. Since modules are built and tested in Real-time during a single deployment, the possibility of network-related failures, version incompatibilities, and incompatible APIs, so typical of microservices set ups, do not exist. The tests may be run faster between modules, and failure can be traced back better in context. It also has the shared testing utilities and build tools, which result in a more similar quality assurance. In general, the modular monolith architecture encourages a more disciplined, team-centred and predictable development process. It provides simplicity and structure, providing the value of modularization (at the level of operating systems), but none of the operational overhead of distributed systems, resulting in a more enjoyable and productive development experience.

5.3. Limitations

Although the modular monolith architecture has a few benefits concerning the ease of use, maintainability, and developer experience, this does not mean that it lacks shortcomings. Among its main disadvantages is the fact that it cannot be used in real-time streaming cases. Monolithic, writes Grist, can be a bottleneck in situations involving processes of low-latency and high-throughput, like those related to live analytics, IoT telemetry, or financial tick data. Modular monoliths run in one process and are usually synchronized in communication patterns, which restricts them from being horizontally distributed over multiple nodes. By comparison, asynchronous messaging, event-driven designs, and event-sourced systems are more applicable to real-time systems with their associated dedicated infrastructure requirements (Kafka or Apache Flink) and may be more appropriate when using microservices. The second weakness is that modular monoliths can also need decomposition on a large scale. When systems become more complex and an organisation increases, the monolith might be vast to manage even with a modularized software, leading to many problems.

Where many separate modules and teams contribute to the same code base, compile time, deployment risk, and cognitive overhead can degrade continuously. To this end, decomposition further into independently deployed microservices may be required in order to get the units of inter-team coordination to a point where there are low costs of coordination, paradigms of performance bottlenecks, and enable more flexible scales to be applied. Such a transition, however, is less painful when the modular monolith has been constructed with clear boundaries, making it easier to pull out and deploy modules when the necessity arises as independent services. Also, the lack of certain tooling facilities, notably to impose precise levels of module isolation and visibility, may in some situations cause boundary violations unless properly watched. Although the static analysis tool is available, it might not be adequate in enforcing runtime issues like service contracts, latency, and distributed failures. Thus, even though modular monoliths are a practical and scalable starting point, they should instead be embraced with an eye to these shortcomings and a long-term outlook on a future architecture change.

5.4. Comparative Summary

Discussing the architectural styles such as modular monolith, microservices and traditional monoliths, it is critical to compare their qualities in terms of two attributes: ease of deployment and maintainability, and the freedom of work teams. This comparison will help to understand what kind of architecture is best in a specific situation, particularly in the case of increasing groups and changing systems. One of the greatest strengths of both modular and traditional monoliths is the simplicity of deployment. All the workflows of the deployment are simple and quick since all are packaged and deployed as a single unit. This gives the modular monolith, monolith architecture, a high rating under this category. Microservices, on the other hand, have the complexity of managing lots of services, lots of containers, lots of orchestration layers (such as Kubernetes) and lots of communication between services. Such a considerable overhead also renders their deployment relatively simple, at least when it comes to small or medium-sized teams. When it comes to maintainability, the modular monolith has an excellent score. Its internal architecture is a modular one, which readably enforces separation of concerns and bounded contexts from each other, thereby simplifying navigating, refactoring, and extending it.

Because traditional monoliths usually have no well-defined internal boundaries, they are more prone to be entangled and more difficult to support in the long run; therefore, they score low. The balance of maintainability is apposite at the microservices: although separate services can be updated and changed with identifying the teams, the operational and cross-cutting issues (i.e., distributed logging, versioning, and monitoring) can provide new layers of complexity that can negate these advantages. Lastly, when it comes to team autonomy, micro services do best as they have high scores because they allow separate services to be assigned to different teams, which can then follow the stages of build, test and deployment without having to closely coordinate with others. The modular monolith also has an intermediate level: modules can be owned by teams, who should largely be able to work broadly autonomously, but should share a deployment schedule and base code. Conventional monoliths will have minimal scores on autonomy, where all modifications may cause impacts on common components of the system that result in closed-loop collaboration by team members.

6. Conclusion and Future Work

Modular monolith architecture is an interesting compromise between the simplicity of monoliths and the scaling of microservices. To maturing products and growing teams, it offers a methodology system that facilitates modularization in the absence of initial overheads of distributed systems. Using a clear domain structure to arrange code into clear modules, reinforced by the separation of concerns achieved by tight control over code access using static analysis, interface contracts and using internal visibility restrictions, teams can develop a clean separation of concerns. This architecture not only increases maintainability, but also accelerates the development rate and decreases the integration problems. Modular monoliths are easier to deploy and less overhead-heavy to run because they have collective code coverage, consistent architectural designs and styles and ease the on boarding when new developers are added to the team, when compared to microservices. Such advantages turn modular monolith into a tactical decision of organizations that seek to grow in increments, preserving the agility and readiness of future architectural transitions. In the end, when held to a small level of tooling and discipline, modular monoliths are long-term scalable and maintainable solutions that do not require the early pollination that microservices-first often experience.

Even as modular monoliths create momentum, a number of spaces are still unexplored and need improvement. Toolchain automation is one of the promising directions, especially when it comes to automatic enforcement of architectural boundaries. Even though such tools as Arch Unit and JDepend can help in an investigation of violations, in the future, it could be possible to work on incorporating real-time feedback in development context and CI pipelines, making sure that boundary rules are enforced at all times, not requiring manual intervention. The other important one is gradual service extraction, in which service modules can be gradually decoupled and turned into microservices, which are independently deployed as scalability or business complexity requirements dictate. Tooling and research into this area can offer blueprints or scaffolding utilities that can help to make the implementation of the module to service smooth, such as integration of the API Gateways, event-based propagation, and data replication schemes.

Finally, there is an increased requirement for greater monitoring and observability in modular monoliths. Although microservices will inherently focus on telemetry as a distributed solution, monoliths typically provide little or no granularity with metrics built into them. It would be possible in the future to build thin observability layers that give module-level performance statistics, error duration tracking, and utilization analytics, allowing teams to identify performance bottlenecks and learn how to tune system behavior without requiring complete distributed tracing systems. The above in combination will not only make the modular monolith more powerful and friendlier to developers, but also easier to adjust to changing product and team needs in the future.

References

[1] Ren, Z., Wang, W., Wu, G., Gao, C., Chen, W., Wei, J., & Huang, T. (2018, September). Migrating web applications from a monolithic structure to a microservices architecture. In Proceedings of the 10th Asia-Pacific Symposium on Internetware (pp. 1-10).

[2] Evans, E. (2004). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley Professional.

[3] Vernon, V. (2013). Implementing domain-driven design. Addison-Wesley.

[4] Gonçalves, N., Faustino, D., Silva, A. R., & Portela, M. (2021, March). Monolith modularization towards microservices: Refactoring and performance trade-offs. In 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C) (pp. 1-8). IEEE.

[5] Brandolini, A. (2013). Introducing event storming. blog, Ziobrando's Lair, 18.

[6] Bass, L. (2012). Software architecture in practice. Pearson Education India.

[7] Newman, S. (2019). Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly Media.

[8] Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress.

[9] Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019, September). Microservices migration in industry: Intentions, strategies, and challenges. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 481-490). IEEE.

[10] Gravanis, D., Kakarontzas, G., & Gerogiannis, V. (2021, November). You don't need a Microservices Architecture (yet). Monoliths may be the solution in "Proceedings of the 2021 European Symposium on Software Engineering" (pp. 39-44).

[11] Vernon, V., & Jaskula, T. (2021). Strategic monoliths and microservices: driving innovation using purposeful architecture. Addison-Wesley Professional.

[12] Millett, S., & Tune, N. (2015). Patterns, principles, and practices of domain-driven design. John Wiley & Sons.

[13] Kapferer, S., & Zimmermann, O. (2020, February). Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling. In MODELSWARD (pp. 299-306).

[14] Evans, E. (2014). Domain-driven design reference: Definitions and pattern summaries. Dog Ear Publishing.

[15] Mosleh, M., Dalili, K., & Heydari, B. (2016). Distributed or monolithic? A computational architecture decision framework. IEEE Systems journal, 12(1), 125-136.

[16] Heydari, B., Mosleh, M., & Dalili, K. (2016). From modular to distributed open architectures: A unified decision framework. Systems Engineering, 19(3), 252-266.

[17] Newman, S. (2021). Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.".

[18] Wolff, E. (2016). Microservices: flexible software architecture. Addison-Wesley Professional.

[19] Pianini, D., & Neri, A. (2021, September). Breaking down monoliths with Microservices and DevOps: an industrial experience report. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 505-514). IEEE.

[20] Wang, Y., Kadiyala, H., & Rubin, J. (2021). Promises and challenges of microservices: an exploratory study. Empirical Software Engineering, 26(4), 63.

[21] Kalske, M. (2017). Transforming monolithic architecture towards microservice architecture. University of Helsinki.

[22] Rahul, N. (2020). Optimizing Claims Reserves and Payments with AI: Predictive Models for Financial Accuracy. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(3), 46-55. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I3P106

[23] Enjam, G. R., & Chandragowda, S. C. (2020). Role-Based Access and Encryption in Multi-Tenant Insurance Architectures. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(4), 58-66. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I4P107

[24] Pedda Muntala, P. S. R. (2021). Prescriptive AI in Procurement: Using Oracle AI to Recommend Optimal Supplier Decisions. *International Journal of AI, BigData, Computational and Management Studies*, *2*(1), 76-87. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I1P108

[25] Rahul, N. (2021). AI-Enhanced API Integrations: Advancing Guidewire Ecosystems with Real-Time Data. *International Journal of Emerging Research in Engineering and Technology*, *2*(1), 57-66. https://doi.org/10.63282/3050-922X.IJERET-V2I1P107

[26] Enjam, G. R. (2021). Data Privacy & Encryption Practices in Cloud-Based Guidewire Deployments. *International Journal of AI, BigData, Computational and Management Studies*, *2*(3), 64-73. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I3P108