International Journal of Computer Applications (0975 - 8887)
Volume 186 - No.68, February 2025

Architectural Trade-Offs in Modulith Architecture: A Case Study on Dependency and Data Management in Rewards Systems

Chandra Prakash
Senior Member, IEEE

ABSTRACT

As organizations increasingly shift towards microservices for scaling complex systems, challenges such as managing inter-service dependencies, efficient communication, and ensuring faster data access have emerged. This paper explores an alternative architectural approach, the Modular Monolith or Modulith architecture, for building a highly interdependent online rewards system. Through a case study of the system's development, this research examines the advantages of encapsulating business domains within modular components deployed in a unified, monolithic code base to optimize inter-service communication, performance, and data access. The study highlights the architectural tradeoffs, focusing on the ease of managing dependencies and improving data flow across tightly coupled services. Preliminary results from the implementation indicate significant improvements over a traditional monolith or microservices-based architecture, reduced complexity, and simplified deployment processes. This research contributes insights into the suitability of modular monolith architecture for high-demand, data-intensive applications and offers guidance for organizations considering alternatives to microservices for similar use cases.

General Terms

Modular Architecture, Architecture Pattern

Keywords

Modulith, Modular Monoliths, Microservices, Event-driven architecture, Software Architecture

1. INTRODUCTION

Online rewards systems are increasingly complex, requiring the seamless integration of multiple services such as user management, rewards tracking, and third-party integrations. These systems experience high transaction volumes, significant data exchange, and dependency between components, making the choice of software architecture a critical decision. On the one hand, traditional monolith architectures offer simplicity when designing an online rewards system. On the other hand, microservices architecture provides scalability. However, both approaches have limitations for highly dependent, data-intensive applications. In this context, modulith or modular monolith architecture presents a compelling alternative. Modulith architectural pattern combines the simplicity and cohesion of monolithic structures with the flexibility and maintainability of modular design, offering a balanced solution for complex applications[1]. As organizations strive to create robust and scalable online rewards systems, the modular monolith architecture presents an opportunity to address critical challenges in inter-service dependencies and data access management.

The concept of a modulith architecture is rooted in the principles of Domain-Driven Design (DDD), which emphasizes the importance of aligning software architecture with business domains[2]. Figure 1 provides a visual comparison of monolith and modulith application structure. By adopting ubiquitous language and consistent terminology across the entire domain, as advocated by Evans[3], organizations can minimize misunderstandings and foster clear communication among stakeholders[1]. This approach is particularly relevant in the context of online rewards systems, where the complexity of business rules and the need for high performance intersect.

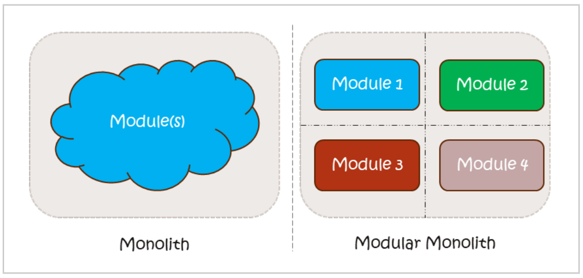


Fig. 1. A visual comparison of a monolith and modulith Architecture

This research paper explores the implementation of a modulith architecture for a high-utilized online rewards system, with a specific focus on managing inter-service dependencies and optimizing data access patterns. By examining this architectural approach, the study aims to provide insights into how organizations can leverage the benefits of modular design within a monolithic framework to achieve improved system performance, maintainability, and scalability[4]. The importance of effective dependency management in highly utilized systems cannot be overstated. As online rewards systems grow in complexity, the inter-dependencies between various components can lead to system evolution and maintenance challenges. The decision to opt for this architecture was driven by the high degree of dependency between modules and the need for efficient and frequent data exchange across services. The modular monolith architecture offers a promising solution by encapsulating related functionalities within well-defined boundaries, thereby reducing coupling and enhancing cohesion across the system[5].

2. LITERATURE REVIEW

Rewards systems, especially those operating in competitive online environments, demand real-time data processing and low-latency access to shared data across multiple services. Traditional monolith architectures can become cumbersome as systems grow in size, making deployment and testing more difficult. Meanwhile, microservices architecture introduces challenges such as increased complexity in managing distributed transactions, communication overhead, and potential bottlenecks due to inter-service calls[6, 4]. One can address these limitations by adopting and combining both approaches through modulith architecture. This hybrid model aims to provide a balance between the simplicity of deployment and the flexibility of modular development, making it particularly suitable for online rewards systems. It organizes the code base into well-defined modules that align with specific business domains, encapsulating services with clear boundaries while maintaining a unified deployment model. This approach minimizes the overhead associated with inter-service communication by keeping interactions within a single process and database context, thereby reducing the risk of latency, maintaining performance, and simplifying data consistency.

2.1 Evolution of Software Architectures

The progression from monolithic to microservices architectures represents a significant shift in software design. Monolithic architectures, characterized by their single-tiered software application structure, were the predominant approach for many years. In a monolithic system, all components of an application are tightly integrated into a single code base, sharing the same memory space and resources[7]. This architecture offered simplicity in development, deployment, and testing, as all functionalities were contained within a single unit. However, as applications grew in complexity and scale, the limitations of monolithic architectures became increasingly apparent. These limitations included difficulties in maintaining and updating large code bases, challenges in scaling specific components independently, and the potential for a single point of failure affecting the entire system[4, 7]. The need for more flexible and scalable solutions led to the rise of microservices architecture.

Microservices architecture emerged as a response to the challenges posed by monolithic systems. This approach decomposes an application into a collection of loosely coupled, independently deployable services, each responsible for a specific business capability[6, 8]. Microservices offer several advantages, including improved scalability, easier maintenance, and the ability to use different technologies for different services. However, this architecture also introduced new complexities, such as managing inter-service communication, data consistency across services, and increased operational overhead[6].

2.2 Modular Monolith or Modulith Architecture

As the software development community grappled with the trade-offs between monolithic and microservices architectures, a hybrid approach known as modulith emerged. This architecture combines the benefits of both monolithic and microservices architectures, offering a middle ground that addresses many of the challenges faced by pure implementations of either approach[9, 10].

A modulith is characterized by its single deployment unit, similar to a traditional monolith, but with a crucial difference: the internal structure is organized into distinct modules with clear boundaries and interfaces[4, 10]. Each module encapsulates a specific domain or functionality of the application, maintaining a high degree of independence in terms of code organization and business logic. A few key characteristics of modulith architecture include:

(1) Modular Structure: The application is divided into loosely coupled modules representing a distinct business capability or domain.

(2) Clear Boundaries: Modules have well-defined interfaces and boundaries, promoting encapsulation and reducing interdependencies.

(3) Single Deployment Unit: Despite its modular structure, the entire application is deployed as a single unit, simplifying deployment and operational processes.

(4) Shared Resources: Modules can share certain resources, such as databases or external services while maintaining logical separation.

(5) Gradual Migration Path: Modular monoliths offer a stepping stone for organizations transitioning from monolithic to microservices architectures, allowing for incremental modernization.

The modulith approach addresses several challenges faced by both monolithic and microservices architectures. It mitigates the complexity of distributed systems inherent in microservices while providing better organization and maintainability than traditional monoliths. This architecture is particularly beneficial for managing inter-service dependencies and data access in complex systems like high-performance online rewards platforms[4, 10].

By adopting a modulith architecture, organizations can achieve a balance between the simplicity of monoliths and the flexibility of microservices. This approach allows for easier refactoring, more straightforward testing, and improved team autonomy without the operational complexities associated with fully distributed microservices systems. As the software development landscape continues to evolve, the modulith architecture represents a pragmatic solution for organizations seeking to modernize their applications while managing complexity and maintaining performance. Its emergence underscores the importance of considering architectural trade-offs and choosing the most appropriate design based on specific project requirements and organizational constraints[11].

3. METHODOLOGY

This study employs a single-case study approach[12] to explore the design, implementation, and evaluation of a modulith architecture for the Online Rewards System. This case study offers an alternative to the microservices architecture approach, where the various modules require frequent data access and inter-module communication. All code developed during the case study is available here - `https://github.com/c-prakash/ezloyalty`

4. RESULTS

4.1 Modulith Architecture for Online Rewards Systems

Adopting modulith architecture for online rewards systems represents a significant shift in the approach to designing high-performance, scalable, and maintainable software solutions. This architectural paradigm offers a compelling alternative to traditional monolithic structures and microservices, particularly in the context of managing inter-service dependencies and data access within complex rewards systems.

4.1.1 Rationale for Modulith Architecture Adoption. The implementation of modulith architecture in online rewards systems presents several key advantages that address the unique challenges faced by these platforms. Primarily, this approach facilitates a more efficient management of inter-service dependencies, which is crucial in the intricate ecosystem of rewards programs where multiple components must interact seamlessly. Rewards systems' need for frequent and consistent access to shared data across modules made microservices impractical due to the potential overhead of distributed communication. In a modulith, these modules can directly communicate through function calls, reducing the latency associated with network-bound microservices.

One of the primary benefits is the enhanced data consistency and integrity that modular monoliths offer. In a rewards system, where transactional accuracy is paramount, the ability to maintain a single, coherent data model across all modules significantly reduces the risk of data inconsistencies that can arise in distributed systems. Rewards systems often require immediate access to consistent data across the system (e.g., user rewards balances must be updated immediately across all modules). The shared database model in a modular monolith simplifies this requirement without introducing the challenges of eventual consistency in distributed systems.

Also, the modulith architecture allows for a more streamlined approach to performance optimization. By keeping all modules within a single deployment unit, developers can more easily identify and address bottlenecks that will occur in the rewards calculation or redemption processes. This centralized structure enables more efficient resource utilization by eliminating the need for complex inter-service communication protocols that are often required in microservices architectures and minimizes the complexity of distributed state management.

Another significant advantage is the simplification of the development and deployment processes. In the context of online rewards systems, where frequent updates and feature additions are common, the modulith approach allows for easier code management and version control. Developers can work on individual modules without the complexity of managing multiple repositories and deployment pipelines, which is often the case with microservices. The modular nature of this architecture also promotes code reusability and maintainability. For instance, common functionalities such as authentication, logging, or partner API integrations can be encapsulated in shared modules, reducing duplication and ensuring consistency across the system. This is particularly beneficial in rewards systems where standardized processes, such as point calculation algorithms or redemption workflows, must be applied uniformly across different parts of the application. Lastly, the modular monolith approach provides a balanced solution for scalability. While it may not offer the same level of fine-grained scalability as microservices, it allows for selective scaling of modules that experience higher load, such as during promotional periods or peak redemption times. This targeted scalability can be achieved without the operational complexity associated with managing a fully distributed microservices ecosystem.

4.1.2 Challenges and Considerations. While the modulith architecture offers numerous benefits for online rewards systems, it also presents certain challenges that must be carefully considered and addressed during implementation. One of the primary challenges lies in managing the complexity of module boundaries and interactions. As the system grows, there is a risk of modules becoming tightly coupled, which can negate many of the benefits of modularity. To mitigate this, developers must invest significant effort in designing clear and well-defined interfaces between modules, ensuring that each component remains as independent as possible while allowing necessary interactions.

Performance optimization in a modulith can also be challenging, particularly as the system scales. While the unified nature of the architecture can simplify some aspects of performance tuning, it also means that inefficiencies in one module can potentially impact the entire system. This necessitates a comprehensive performance monitoring and optimization approach, focusing on identifying and addressing bottlenecks at the module and system levels. Data management presents another significant consideration. While the centralized data model of a modular monolith can enhance consistency, it also requires careful design to prevent data access patterns that could lead to performance issues or tight coupling between modules. Implementing effective data partitioning strategies and establishing clear data ownership boundaries between modules is crucial to maintaining the benefits of the modular architecture.

Scalability, while improved compared to traditional monoliths, still requires thoughtful planning in a modulith. As the rewards system grows, there may be scenarios where certain modules need to scale independently, which can be more challenging than in a microservices architecture. Developers must design the system with future scalability in mind, potentially incorporating strategies such as vertical slicing of functionality to allow for more flexible scaling options. The transition to a modulith architecture, especially for existing rewards systems, can be a complex process. It requires a significant upfront investment in redesigning the system architecture, refactoring existing code, and potentially retraining development teams. This transition period can be challenging and may temporarily impact development velocity.

Lastly, while the modulith approach can simplify many aspects of system management, it still requires disciplined development practices to maintain modularity over time. There is a risk of the system gradually devolving into a traditional monolith if module boundaries are not strictly enforced and developers are not vigilant about maintaining the separation of concerns.

4.2 Inter-Service Dependency Management

In the context of adopting a modulith architecture for an online rewards system, effective management of inter-service dependencies is crucial. This section explores strategies to optimize dependency relationships within the modular structure, ensuring system robustness, scalability, and maintainability.

4.2.1 Dependency Injection and Inversion of Control. Dependency Injection (DI) and Inversion of Control (IoC) are fundamental techniques in managing inter-service dependencies within a modular monolith architecture. These approaches significantly reduce tight coupling between services and enhance overall system modularity[13]. In a modulith, services are organized as distinct modules within a single code base. While this structure offers advantages in terms of deployment simplicity and transactional integrity, it also presents challenges in managing dependencies between modules. Dependency Injection addresses these challenges by externalizing the creation and management of dependencies.

The principle of Dependency Injection involves providing a service with its dependencies rather than having the service create or manage them internally. Dependency Injection is typically achieved through constructor, property, or method injection. For instance, consider a user action tracking controller to record user activity that depends on an action query service:

```
public class ActionsController :
    ControllerBase
{
    private readonly IMediator _mediator;
    private readonly IActionsQueries
        _actionsQueries;
    private readonly ILogger<
        ActionsController> _logger;

    public ActionsController(
        IMediator mediator,
        IActionsQueries actionsQueries,
        ILogger<ActionsController> logger)
    {
        _mediator = mediator ?? throw new
            System.ArgumentNullException(
            nameof(mediator));
        _actionsQueries = actionsQueries ??
             throw new System.
            ArgumentNullException(nameof(
            actionsQueries));
        _logger = logger ?? throw new
            System.ArgumentNullException(
            nameof(logger));
    }
```

In this example, the ActionsController receives its dependency (IActionsQueries) through constructor injection. This approach decouples the controller from the concrete implementation of the action query service, allowing for easier testing, maintenance, and potential future modifications. Inversion of Control complements Dependency Injection by shifting the responsibility of managing dependencies to an external container or framework. In the context of a modular monolith, an IoC container can manage the lifecycle and injection of dependencies across different modules. This centralized management of dependencies facilitates: a) Loose coupling: Services depend on abstractions rather than concrete implementations. b) Improved testability: Dependencies can be easily mocked or stubbed for unit testing. c) Flexibility: Implementations can be swapped without modifying the dependent services. d) Centralized configuration: Dependency relationships can be defined and managed in a single location. By leveraging DI and IoC, the modulith architecture can maintain clear boundaries between services while allowing for flexible and manageable inter-service communication. This approach aligns with the principles of microservices in terms of modularity and independence while retaining the benefits of a monolithic deployment model.

4.2.2 Event-Driven Architecture in Modular Monoliths. Event-driven architecture (EDA) presents a powerful paradigm for managing dependencies and enhancing system scalability within a modular structure. By adopting event-driven patterns, the online rewards system can achieve looser coupling between services, improved responsiveness, and better scalability[14]. In an event-driven modulith, services communicate primarily through events rather than direct method calls. This approach offers several advantages in managing inter-service dependencies: a) Decoupling: Services emit events without the knowledge of their consumers, reducing direct dependencies. b) Scalability: Event-driven systems can more easily scale to handle increased load, as events can be processed asynchronously. c) Extensibility: New functionality can be added by introducing new event consumers without modifying existing services. d) Resilience: Temporary failures in one service are less likely to cascade through the entire system.

Implementing an event-driven architecture within a modular monolith involves several key components:

—Event Bus: A centralized publishing and subscribing events engine. In a modulith architecture, this can be an in-memory event bus, avoiding the complexity of distributed message queues while providing event-driven communication benefits.
—Event Publishers: These services generate events based on specific actions or state changes.
—Event Subscribers: Services that listen for and react to specific events.
—Event Store: A persistent storage mechanism for events, enabling event sourcing and replay capabilities.

```
public class ActionStatusChangedToAwaiting-
    AccountValidationDomainEvent
            : INotification
{
    public int AccountNo { get; private set
        ; }
    public int ActionRecordId { get;
        private set; }
    public ActionStatusChangedToAwaiting-
        AccountValidationDomainEvent(int
        customerNo, int recordId)
    {
        AccountNo = customerNo;
        ActionRecordId = recordId;
    }
}

public class ActionStatusChangedToAwaiting-
    AccountValidationDomainEventHandler
          : INotificationHandler<
            ActionStatusChangedToAwaiting
            -
            AccountValidationDomainEvent
            >
{
    private readonly IActionsRepository
        _actionRepository;
    private readonly ILoggerFactory _logger
        ;
    private readonly
        IActionIntegrationEventService
        _actionIntegrationEventService;

    public ActionStatusChangedToAwaitingAccountValidationDomainEventHandler
        (
        IActionsRepository
            actionsRepository,
            ILoggerFactory logger,
            IActionIntegrationEventService
            actionIntegrationEventService)
    {
        _actionRepository =
            actionsRepository ?? throw new
            ArgumentNullException(nameof(
            actionsRepository));
        _logger = logger ?? throw new
            ArgumentNullException(nameof(
            logger));

        _actionIntegrationEventService =
            actionIntegrationEventService;
    }

    public async Task Handle(
        ActionStatusChangedToAwaitingAccountValidationDomainEvent
        actionStatusChangedToAwaitingValidationDomainEvent,
        CancellationToken cancellationToken
        )
    {
        // Change the action status to
            pending validation
        // Publish new integration event to
             a new service
        //  await
            _actionIntegrationEventService.
            AddAndSaveEventAsync(event);
    }
}
```

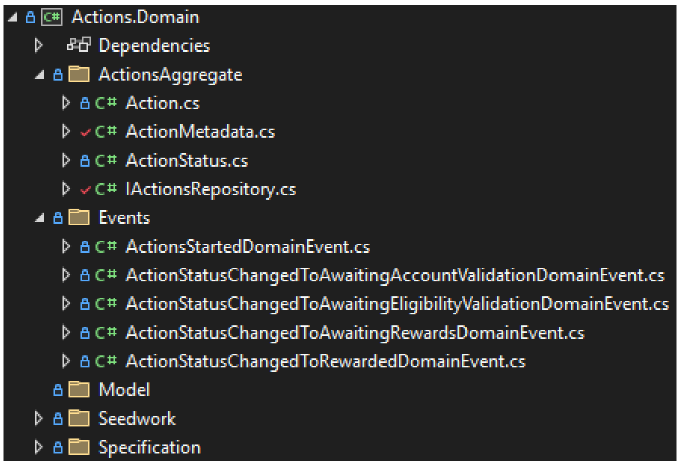
In the above scenarios, the Action service publishes the ActionStatusChangedToAwaitingAccountValidationIntegrationEvent event. When an activity, e.g., the customer completes a purchase, the Action service publishes the event to the event bus to validate the account status. The Account service then reacts to this event, providing the customer's account information, which is further communicated to the Incentive service to calculate and apply the rewards without directly coupling them to the purchase processing logic. Implementing event-driven patterns in a modulith requires careful consideration of event design, consistency, and performance. While events provide loose coupling, they also introduce challenges in maintaining data consistency and tracing complex workflows. To address these challenges, techniques such as event sourcing, CQRS (Command Query Responsibility Segregation), and saga patterns can be employed within the modular structure. By integrating event-driven architecture principles, the online rewards system can achieve a balance between the modularity of microservices and the simplicity of monolithic deployment. This approach facilitates easier management of inter-service dependencies, improves system scalability, and provides a pathway for future evolution towards a fully distributed architecture if required.

4.3 Data Access Strategies in Modular Monolith

In the context of developing a highly efficient online rewards system using a modulith architecture, optimizing data access strategies is crucial for ensuring system efficiency and scalability. This section explores key approaches to managing data access within the modulith framework, focusing on domain-driven design principles and performance optimization techniques.

4.3.1 Domain-Driven Design in Data Management. Domain-Driven Design (DDD) plays a pivotal role in structuring data access within a modular architecture. Through DDD principles, the system's data model can align with the business domain, thereby minimizing redundancy and enhancing overall system coherence. In the context of an online rewards system, DDD facilitates the creation of well-defined boundaries between different modules, such as user account management, activity tracking, reward calculation, and program management. Each module maintains its own domain model, encapsulating the data and business logic specific to its functionality. This approach allows for a unified database schema, a characteristic feature of modular architecture, while maintaining a clear separation of concerns[4, 10].

The implementation of DDD in data management for a modular monolith involves several key strategies. Firstly, the use of aggregates helps in defining clear boundaries around related entities and value objects. For instance, an "Action" aggregate might encompass account profile data, activity data, and reward points. This aggregation ensures that data integrity is maintained within the module and reduces the need for cross-module data access[4, 10]. Figure 2 provides an example of domain aggregate. Secondly, the concept of bounded contexts in DDD aids in managing the complexity of large-scale systems. Each module in the modulith can be treated as a bounded context with its own ubiquitous language and data model. This approach helps in avoiding conflicts in terminology and data representation across different parts of the system[10]. Lastly, the use of domain events facilitates communication between modules without tight coupling. When significant changes occur within a module, such as a user performing an activity to earn a reward, a domain event can be published. Other modules can subscribe to these events, allowing for loose coupling and asynchronous communication patterns, which are preferred in modular architectures[10].



```
Actions.Domain
  Dependencies
  ActionsAggregate
    Action.cs
    ActionMetadata.cs
    ActionStatus.cs
    IActionsRepository.cs
  Events
    ActionsStartedDomainEvent.cs
    ActionStatusChangedToAwaitingAccountValidationDomainEvent.cs
    ActionStatusChangedToAwaitingEligibilityValidationDomainEvent.cs
    ActionStatusChangedToAwaitingRewardsDomainEvent.cs
    ActionStatusChangedToRewardedDomainEvent.cs
  Model
  Seedwork
  Specification
```

Fig. 2. Action aggregate and domain events

4.4 Caching and Performance Optimization

Optimizing data access performance is critical in high-load scenarios typical of online rewards systems. Caching plays a crucial role in reducing latency and improving overall system responsiveness. In a modular architecture, caching strategies can be implemented at various levels to enhance performance[1]. One practical approach is implementing a distributed caching system that spans across modules. This allows frequently accessed data, such as user accounts or current reward balances, to be cached and shared efficiently among different parts of the system. By adopting loosely coupled asynchronous communication patterns, modules can update and retrieve cached data without direct dependencies[10].

Another performance optimization technique involves the strategic use of read models. Maintaining separate read-optimized data structures can significantly improve query performance in scenarios where data is frequently read but rarely updated. This approach aligns well with the Command Query Responsibility Segregation (CQRS) pattern, which can be effectively implemented within a modular monolith architecture[4, 10]. To further enhance performance, the system can employ intelligent pre-fetching mechanisms. By analyzing usage patterns and anticipating data needs, the system can proactively load relevant data into the cache, reducing latency for subsequent requests. This is particularly beneficial for operations that involve complex calculations or aggregations, such as determining a user's eligibility for specific rewards.

4.5 Case Study: Modular Rewards System Architecture

This section presents a practical case study of transitioning an online rewards system to a modular architecture. The case study examines the implementation process, architectural decisions, and maintainability. By analyzing this real-world application, the study aims to provide insights into the effectiveness of modulith architecture in addressing the challenges faced by high-performance online systems.

The new architecture was designed to focus on modularity, clear boundaries between components, and efficient inter-module communication. The system was divided into distinct modules, each responsible for specific functionalities such as user account management, program management, user activity tracking, and reward calculation. These modules were implemented in a separate .NET project within a single solution structure as one deployable unit, maintaining the benefits of a monolithic deployment while introducing a higher degree of organization and separation of concerns. The current design has four modules: a) accounts, b) actions, c) incentives, and d) programs. Figure 3 provides the various modules in the online rewards system and how each module communicates with other modules.

4.5.1 Common Architecture Pattern. The transition to a modular architecture for the online rewards system involved careful consideration of architectural decisions and design patterns. The primary goal was to address the limitations of the existing monolithic structure while avoiding the complexities associated with a fully modular approach. Every module was divided into four projects using the following Visual Studio project structure to ease code management and organization. Figure 4 depicts the project structure and individual modules in a Visual Studio solution.

—ProjectName.API - API project includes the API controller/endpoints and the integration events and handlers.

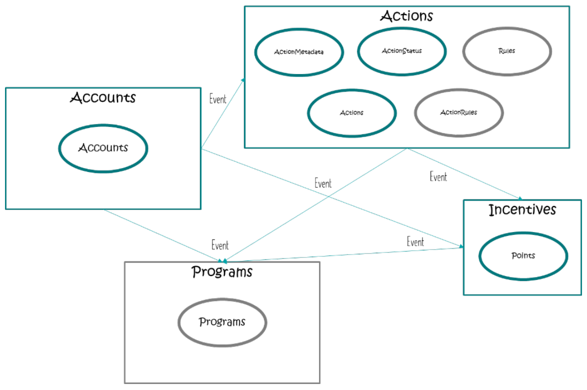


Fig. 3. Various modules included in an online rewards system

—ProjectName.Domain - Domain projects have respective domain models and domain events.

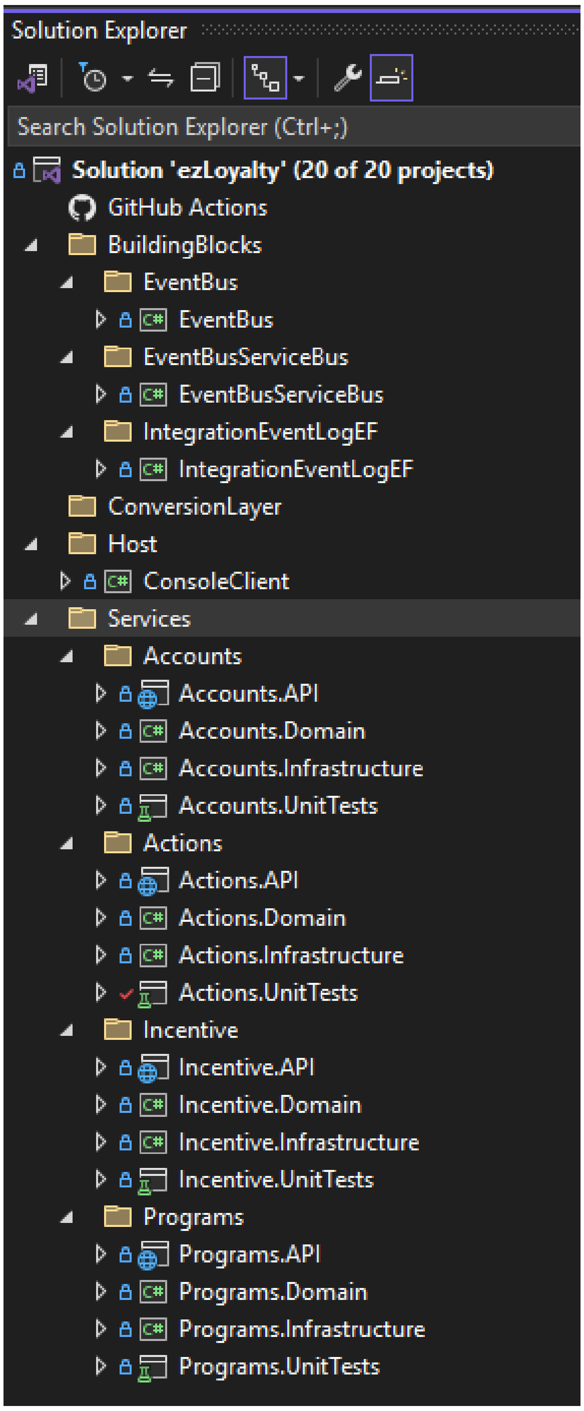
—ProjectName.Infrastructure - The infrastructure project includes the database interaction layer. The case study uses only SQL Server, which is extendable to any database due to its decoupled design.

—ProjectName.UnitTests - Unit test cases for the above three projects.

4.5.2 Event-Driven Architecture. To manage inter-module dependencies, the system employed a combination of dependency injection and mediator patterns. The dependency injection container was configured to handle module components' instantiation and life cycle management, promoting loose coupling between modules. The architecture also incorporated event-driven design principles to handle complex workflows and maintain system responsiveness. The mediator pattern was implemented to facilitate communication between modules through domain and integration events, allowing for a more decoupled and maintainable code base. Domain events were localized to the module; however, critical events, such as incentive or eligibility checks, were published via integration events through an internal event bus, allowing interested modules to react accordingly without tight coupling. Figure 5 provides a view of event communication through the event bus, and Figure 6 depicts the various events and event handlers in a module.
Data access within the modular architecture was carefully designed to balance performance and modularity. Each module was given its dedicated data access layer that was responsible for interacting with the underlying database. The modular and flexible architecture allowed for every module to have its own dedicated database or dedicated schema in a shared database. Figure 7 presents the Infrastructure layer, which represents the data access layer.

5. CONCLUSION AND FUTURE WORK

This study has provided valuable insights into adopting modulith architecture for an online rewards system, focusing on managing inter-service dependencies and data access. This study has demonstrated that the modular monolith approach offers a compelling alternative to microservices architecture, particularly for organizations seeking to balance system modularity, performance, maintainability, and scalability. Implementing a modulith architecture in the context of an online rewards system has shown significant benefits in reducing complexity and improving overall system performance. By encapsulating distinct functionalities within modules



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ezLoyalty' (20 of 20 projects)
  GitHub Actions
  BuildingBlocks
    EventBus
      EventBus
    EventBusServiceBus
      EventBusServiceBus
    IntegrationEventLogEF
      IntegrationEventLogEF
  ConversionLayer
  Host
    ConsoleClient
  Services
    Accounts
      Accounts.API
      Accounts.Domain
      Accounts.Infrastructure
      Accounts.UnitTests
    Actions
      Actions.API
      Actions.Domain
      Actions.Infrastructure
      Actions.UnitTests
    Incentive
      Incentive.API
      Incentive.Domain
      Incentive.Infrastructure
      Incentive.UnitTests
    Programs
      Programs.API
      Programs.Domain
      Programs.Infrastructure
      Programs.UnitTests
```

Fig. 4. A visual studio solution structure for modulith design

while maintaining a unified code base, the study has observed enhanced data consistency, simplified deployment processes, and improved resource utilization. These advantages have translated into a more efficient and responsive rewards system capable of handling high transaction volumes with reduced latency.

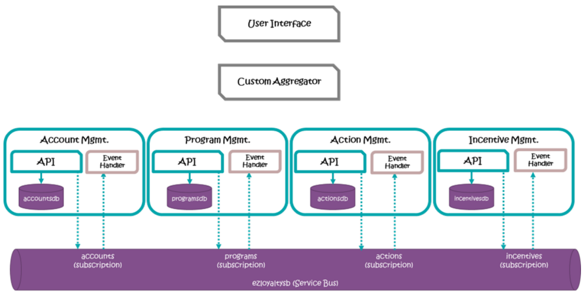
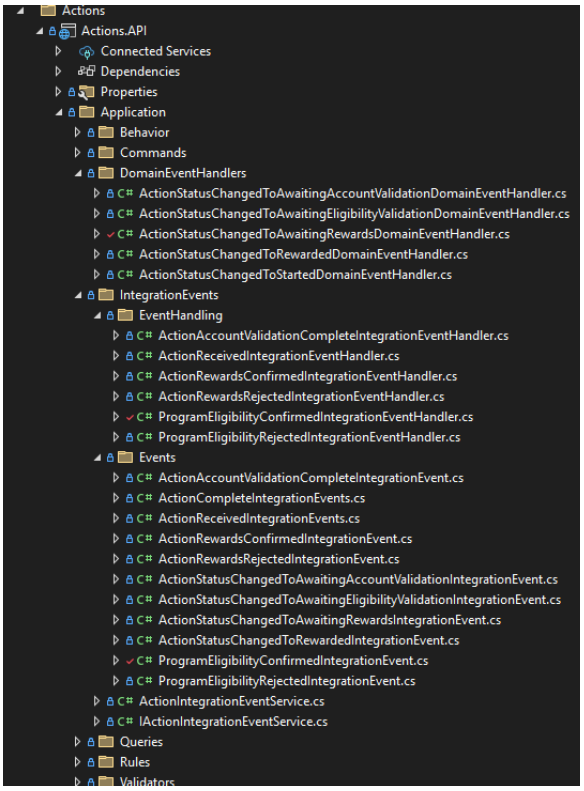


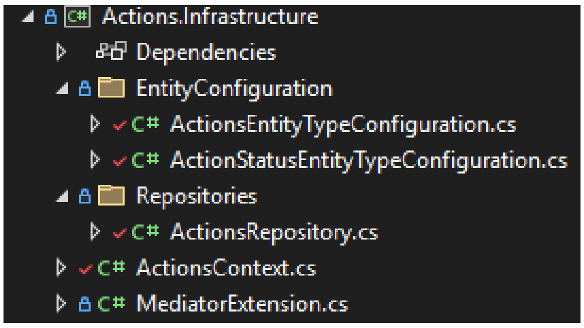
Fig. 5. Rewards system modules and cross-module communication through Events



```
Actions
  Actions.API
    Connected Services
    Dependencies
    Properties
    Application
      Behavior
      Commands
      DomainEventHandlers
        ActionStatusChangedToAwaitingAccountValidationDomainEventHandler.cs
        ActionStatusChangedToAwaitingEligibilityValidationDomainEventHandler.cs
        ActionStatusChangedToAwaitingRewardsDomainEventHandler.cs
        ActionStatusChangedToRewardedDomainEventHandler.cs
        ActionStatusChangedToStartedDomainEventHandler.cs
      IntegrationEvents
        EventHandling
          ActionAccountValidationCompleteIntegrationEventHandler.cs
          ActionReceivedIntegrationEventHandler.cs
          ActionRewardsConfirmedIntegrationEventHandler.cs
          ActionRewardsRejectedIntegrationEventHandler.cs
          ProgramEligibilityConfirmedIntegrationEventHandler.cs
          ProgramEligibilityRejectedIntegrationEventHandler.cs
        Events
          ActionAccountValidationCompleteIntegrationEvent.cs
          ActionCompleteIntegrationEvents.cs
          ActionReceivedIntegrationEvents.cs
          ActionRewardsConfirmedIntegrationEvent.cs
          ActionRewardsRejectedIntegrationEvent.cs
          ActionStatusChangedToAwaitingAccountValidationIntegrationEvent.cs
          ActionStatusChangedToAwaitingEligibilityValidationIntegrationEvent.cs
          ActionStatusChangedToAwaitingRewardsIntegrationEvent.cs
          ActionStatusChangedToRewardedIntegrationEvent.cs
          ProgramEligibilityConfirmedIntegrationEvent.cs
          ProgramEligibilityRejectedIntegrationEvent.cs
        ActionIntegrationEventService.cs
        IActionIntegrationEventService.cs
      Queries
      Rules
      Validators
```

Fig. 6. Rewards system modules and cross-module communication through Events

The study findings suggest that the modular approach effectively addresses many of the challenges associated with distributed systems, particularly in the realm of inter-service dependencies[15][16]. By leveraging well-defined interfaces between modules, the study has achieved a level of loose coupling that facilitates more manageable maintenance and updates without compromising the integrity of the system as a whole. This architectural choice has proven especially beneficial in managing the complex relationships inherent in rewards systems, such as activity tracking, rewards accrual, and account management[16]. Furthermore, the study has highlighted the importance of thoughtful data access patterns within the modular structure[16]. By implementing a centralized data access layer, the study has successfully mitigated many data consistency issues that often plague distributed



```
Actions.Infrastructure
    Dependencies
    EntityConfiguration
        ActionsEntityTypeConfiguration.cs
        ActionStatusEntityTypeConfiguration.cs
    Repositories
        ActionsRepository.cs
    ActionsContext.cs
    MediatorExtension.cs
```

Fig. 7. Infrastructure Module Containing Data Access Layer

systems. This approach has not only improved data integrity but has also simplified the implementation of cross-cutting concerns such as caching and security.

However, as with any architectural approach, some areas warrant further investigation and potential improvement. One key area for future work is the exploration of more sophisticated module communication strategies. While the current implementation has shown promise, there is room for optimization in terms of inter-module messaging and event propagation. Future research could focus on developing more efficient patterns for asynchronous communication between modules, potentially incorporating advanced queuing mechanisms. Another promising direction for future work lies in the realm of dynamic module loading and unloading. As online rewards systems often need to adapt to changing business requirements and seasonal promotions, the ability to hot-swap modules without system downtime could provide significant operational benefits. This area of research could explore techniques for safely introducing or removing functionality at runtime, potentially drawing inspiration from plugin architectures or dynamic library loading mechanisms.

Additionally, as the scale of online rewards systems continues to grow, future studies should investigate strategies for horizontal scaling of modulith architecture. While the current architecture has demonstrated good vertical scaling characteristics, exploring methods for distributing modules across multiple nodes while maintaining the benefits of a monolithic code base could yield interesting insights[16]. This could involve research into partial deployment strategies, state replication techniques, or novel approaches to distributed transactions within a modular context[16].

In conclusion, this case study has demonstrated the viability and advantages of adopting a modulith architecture for a high-performance online rewards system. By effectively managing inter-service dependencies and optimizing data access patterns, the study has shown that this architectural approach can deliver significant benefits in terms of system performance, maintainability, and scalability[16]. As the field of software architecture continues to evolve, the insights gained from this study provide a solid foundation for further research and development in the domain of modular system design for complex, transaction-intensive applications like online rewards systems.

6. REFERENCES

[1] Matheus Felisberto. The trade-offs between monolithic vs. distributed architectures. arXiv preprint arXiv:2405.03619, 2024.

[2] Michail Tsechelidis, Nikolaos Nikolaidis, Theodore Maikantis, and Apostolos Ampatzoglou. Modular monoliths the way to standardization. In Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum, pages 49–52, 2023.

[3] Eric Evans. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

[4] Chandra Prakash and Sunil Arora. Systematic analysis of factors influencing modulith architecture adoption over microservices. In 2024 TRON Symposium (TRONSHOW), pages 1–8, 2024.

[5] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. Flexible and optimal dependency management via max-smt. In 2023 IEEE/ACM 45th International Conference on Software En- gineering (ICSE), pages 1418–1429. IEEE, 2023.

[6] Chandra Prakash. Zero-Trust Architecture Approach to Secure Microservices for the Healthcare Insurance In- dustry. PhD thesis, University of the Cumberlands, 2024. Available at https://www.proquest.com/openview/eef147d67ec912743f9791d236299c6f/ 1?pq-origsite=gscholarcbl=18750diss=y.

[7] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Transactions on Software Engineering, 49(8):4213–4242, 2023.

[8] Sourabh Sethi and Sarah Panda. Transforming digital experiences: The evolution of digital experience platforms (dxps) from monoliths to microservices: A practical guide. Journal of Computer and Communications, 12(2):142–155, 2024.

[9] Mehdi AIT SAID, Lahcen BELOUADDANE, Soukaina MIHI, and Abdellah EZZATI. Modulith architecture: Adop- tion patterns, challenges, and emerging trends. International Journal of Computing and Digital Systems, 16(1):189–203, 2024.

[10] Ruoyu Su and Xiaozhou Li. Modular monolith: Is this the trend in software architecture? In Proceedings of the 1st International Workshop on New Trends in Software Architecture, pages 10–13, 2024.

[11] Taras Shablii and Sergiy Tytenko. Modular monolith as a microservices precursor. Modern engineering and innovative technologies, (29-01):25–32, 2023.

[12] Robert K. Yin. Case study research and applications: Design and methods. SAGE, 2018.

[13] Tianyi Yang, Baitong Li, Jiacheng Shen, Yuxin Su, Yongqiang Yang, and Michael R Lyu. Managing service dependency for cloud reliability: The industrial practice. In 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 67–68. IEEE, 2022.

[14] Luan Lazzari and Kleinner Farias. Event-driven architecture and rest architectural style: An exploratory study on modularity. Journal of applied research and technology, 21(3):338–351, 2023.

[15] Sam Newman. Microservices for greenfield?, 2015.

[16] Martin Fowler. Monolith first, 2015.