Milestone 3

Project: PurrfectMatch

Team: Ana Comeagă & Ruxandra Popa

1) Introduction

PurrfectMatch is a Spring Boot web platform that connects adopters with local shelters. Core features include user authentication, pet browsing/search, adoption requests, and notifications.
This milestone evaluates **three alternative architectures** for the same system:

1. **Monolithic (Layered) architecture** - single deployable, modular codebase.
2. **Microservices architecture** - independently deployable services with their own data.
3. **Event-Driven (distributed) architecture** - components communicate via asynchronous events/message bus.

For each, we describe structure, interactions, and data flow; provide component and deployment diagrams; analyse pros/cons against the project's non-functional requirements; and provide a final comparison and justified choice.

2) Architecture-Significant Requirements Catalogue

Nr.	Non-functional requirement name	Yes /Not architecture significant	If Yes, how may the requirement influence the architecture, according you?
1.	Scalability	Yes	Pet browsing and image retrieval may face high traffic; dictates whether modules must scale independently.
2.	Availability & Reliability	Yes	Adoption and login must remain available; drives need for modular boundaries, retries, or asynchronous queues.
3.	Security & Privacy	Yes	User credentials and shelter data require secure interfaces; motivates a central Auth module / API Gateway.
4.	Maintainability & Modularity	Yes	Domains (Users, Pets, Adoptions, Notifications) must evolve independently; influences package and service separation.
5.	Extensibility	Yes	Future features (analytics, external shelter APIs) should be added without rewriting core logic; encourages event or service-based structure.

6.	Interoperability	Yes	The system exposes REST APIs tested via Postman (e.g., for sending adoption status notifications). Future integrations with other platforms or shelter systems would reuse the same REST interfaces.
7.	Performance / Efficiency	Partial	Read performance improved via caching; does not require structural change.
8.	Usability / Understandability	Partial	Affects front-end, not architecture.

3. Monolithic Architecture

3.1 Description (Structure & Data Flow)

In the **Monolithic (Layered)** architecture, **PurrfectMatch** operates as a single Spring Boot application that includes all functionalities, meaning user management, pet listings, adoption processing, and notifications, within one deployable unit. Although it runs as one application, it is structured into **logical layers**, as illustrated in the **Component Diagram**.

Main Components

- **Presentation Layer:** Includes controllers (AdoptController, UserController, LoginController, AdminController) that handle HTTP requests from the web client or Postman and delegate processing to the business layer.
- **Business Layer:** Contains the service logic in UserServiceImpl and interfaces like UserService, responsible for enforcing application rules, validating input, and coordinating between controllers and repositories.
- **Persistence Layer:** Consists of repositories (UserRepository, PetRepository) that abstract database operations and communicate with **MariaDB** through JPA.
- **Domain Layer:** Defines entities such as User, Pet, Role, and PetSearchCriteria, which model the system's data and relationships.
- **Configuration Layer:** Contains SecurityConfig (authentication setup) and DataInitializer (loads initial users, pets, and roles on startup).
- **Factory Layer:** Implements DefaultPetFactory, which creates predefined Pet objects during initialization for testing and demonstration.

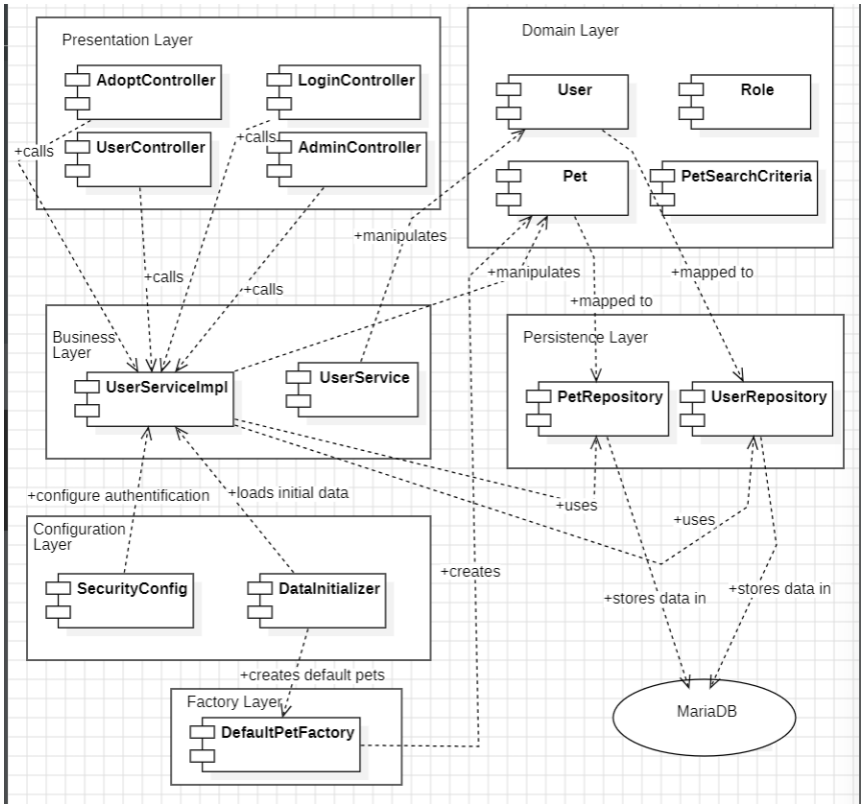
Interactions and Data Flow

1. **User/Admin** sends an HTTP request via a **web browser** or **Postman** to the controller.
2. The **controller** validates input and calls the relevant **service** (UserServiceImpl).
3. The **service** executes the business logic and calls a **repository** to fetch or persist data.
4. The **repository** interacts with **MariaDB**, performing CRUD operations through JPA.

5. The **domain objects** (User, Pet) are populated and returned to the service, which prepares the response.
6. The **controller** sends the HTTP response back to the client.
7. During startup, DataInitializer runs once to populate the database with sample data, using DefaultPetFactory for pet creation.
8. Simulated **notification requests** are triggered internally and tested via Postman.

Component Diagram (Monolithic Architecture)

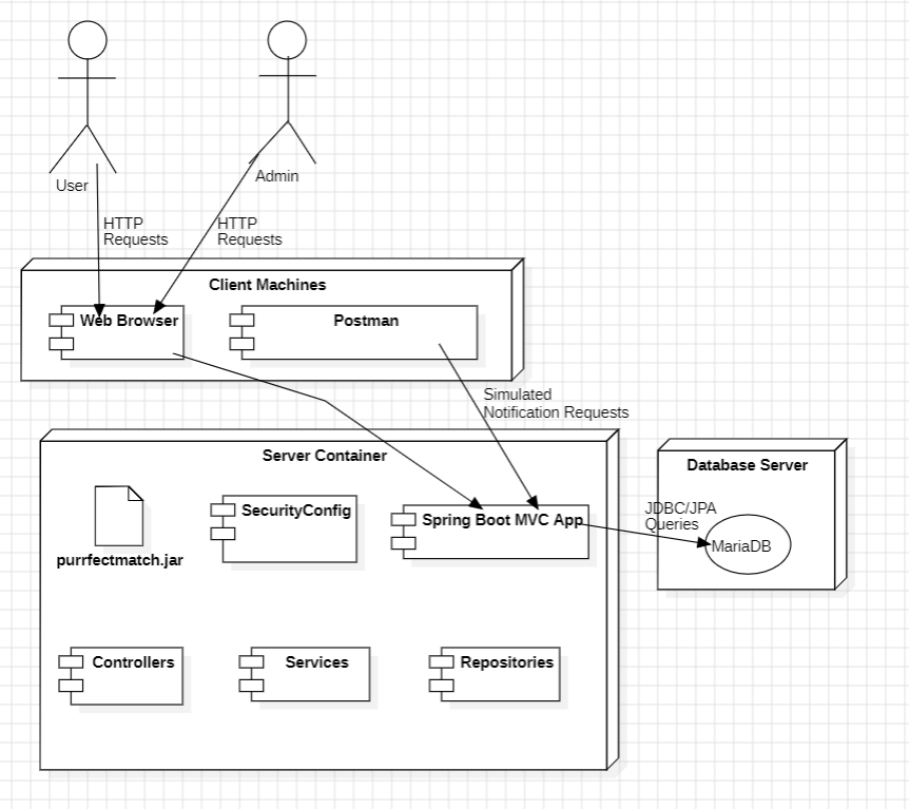
This UML Component Diagram shows the major logical parts of your single Spring Boot application, their dependencies, and external interfaces.



The component diagram illustrates the logical structure of the monolithic Spring Boot application. Controllers handle incoming requests and call corresponding services. Services contain the core business logic and interact with repositories to persist and retrieve data from the MariaDB database. Configuration components such as SecurityConfig and DataInitializer manage environment setup, while DefaultPetFactory demonstrates the Factory Method pattern used to create Pet entities. The Notification component is integrated within the same codebase and is invoked internally by the adoption service to simulate asynchronous communication tested through Postman.

Deployment Diagram (Monolithic Architecture)

A Deployment Diagram shows where components physically run: client → server → database.



As shown in the **Deployment Diagram**, all layers are packaged within the same Spring Boot .jar file. The application runs on a single server container, while clients (Web Browser or Postman) communicate through HTTP. The database server hosts **MariaDB**, which stores all persistent data. This structure ensures simplicity, strong consistency, and minimal deployment overhead.

3.4 Pros & Cons

Advantages	Disadvantages
Simple and stable structure; fully matches current project.	Harder to scale specific modules (entire app must be scaled).
Easy deployment and debugging (single JAR/Docker container).	Single point of failure, one crash affects all features.
Central database ensures strong data consistency.	Difficult to split into microservices later if tightly coupled.
Best fit for small team, short-term academic project.	Not optimal for heavy traffic or parallel feature development.

Advantages

The **Monolithic** architecture offers a simple and cohesive structure in which all modules including authentication, pet management, adoption, and notification handling are integrated into a single Spring Boot application. This unified design makes it easier for developers to understand the overall system and quickly implement, debug, or modify features. The structure promotes fast onboarding for new team members, as the entire codebase and functionality exist within one deployable unit.

Another key advantage is strong data consistency, since all layers of the application share a single MariaDB database and the same transactional context. This guarantees data integrity without requiring distributed transactions. As a result, updates to users, pets, and adoption requests are synchronized in real time, ensuring reliable and accurate records across the system.

The monolithic model also enables faster communication between layers, as interactions between controllers, services, and repositories occur through internal method calls rather than through external network requests. This results in lower latency and improved response times compared to distributed architectures. Testing and debugging are also simplified. Developers can execute full end-to-end tests within the same environment using tools such as Postman, without worrying about network failures or inter-service connectivity issues. Logging and error tracing are centralized, making it easier to isolate and fix issues.

From a deployment and cost perspective, the architecture has low infrastructure requirements. It can run as a single Spring Boot JAR file or Docker container, which simplifies setup and minimizes hosting expenses. This makes it ideal for small academic teams like the **PurrfectMatch** developers, who need a straightforward and stable environment. The approach also supports rapid development and deployment, since all components are compiled and deployed together, allowing for faster iteration cycles during testing and feedback.

Finally, the monolithic structure demonstrates strong alignment with **PurrfectMatch**'s current project scope. The platform does not demand distributed computing, advanced load balancing, or service orchestration. Within this scale, the monolithic architecture provides all the required functionality efficiently while maintaining a clean and organized codebase.

Disadvantages

Despite its simplicity, the monolithic architecture presents several challenges. Its most significant drawback is limited scalability. The entire application must be replicated as a whole to handle heavier workloads, even if only one feature, such as pet search, requires additional resources. This can lead to inefficient resource utilization and scaling costs.

Another concern is the single point of failure. Because all components share a common process, an error or crash in one module such as the adoption workflow can bring down the entire system, impacting all users. As the system grows, long-term maintainability also becomes more difficult. The codebase can become tightly coupled, and introducing new features might unintentionally break existing functionality if modular boundaries are not strictly enforced.

The monolithic structure also makes it harder to adopt new technologies. Since all modules depend on the same framework and stack, upgrading to newer versions or integrating different technologies requires rebuilding and redeploying the entire application. Moreover, parallel development becomes less efficient, as multiple developers working in the same codebase risk merge conflicts and overlapping dependencies.

Lastly, the architecture is not optimized for high traffic or performance peaks. All requests are processed within a single runtime instance, and as concurrency increases, response times can degrade. For a larger-scale platform or production environment, these limitations would hinder system responsiveness and reliability.

4. Microservices Architecture

4.1 Description (Structure & Data Flow)

In the **Microservices Architecture**, **PurrfectMatch** is decomposed into multiple independently deployable services, each encapsulating a single domain area. The services communicate using **REST APIs** and **asynchronous events** via a **Message Broker (RabbitMQ)**, as illustrated in the **Component Diagram**.

Main Components

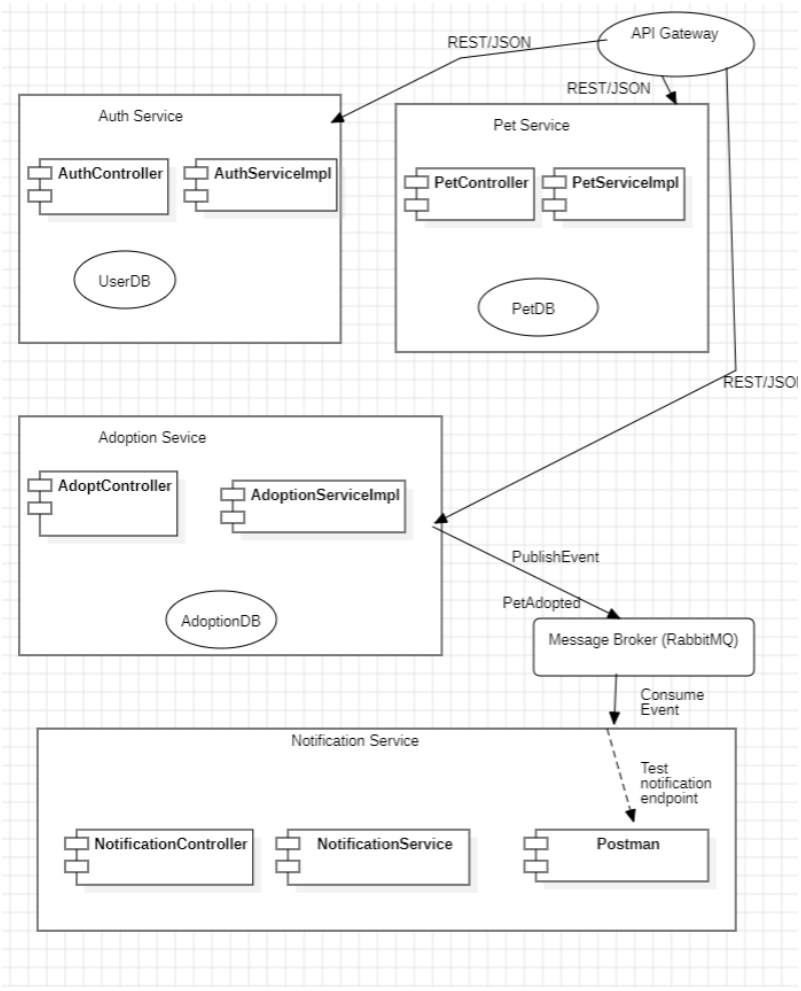
- **Auth Service:** Handles user registration, authentication, and role management. Stores user credentials in UserDB.
- **Pet Service:** Manages CRUD operations for pets, including profile creation, updates, and searches, storing data in PetDB.
- **Adoption Service:** Handles adoption requests, verifies user and pet data, updates pet status, and publishes adoption events (PetAdopted) to the Message Broker. Stores data in AdoptionDB.
- **Notification Service:** Consumes adoption events from the Message Broker and simulates sending notifications to adopters using **Postman**.
- **API Gateway:** The single entry point that routes all client requests (via REST/JSON) to the appropriate service, applies authentication, and provides rate limiting and logging.
- **Message Broker (RabbitMQ):** Manages inter-service event communication asynchronously.
- **Databases:** Each service maintains its own independent database (DB-per-service).

Interactions and Data Flow

1. The **User/Admin** sends an HTTPS request through the **Web UI** or **Postman** to the **API Gateway**.
2. The **Gateway** routes the request to the corresponding microservice (e.g., Auth Service for login, Pet Service for pet search).

3. The **Adoption Service**, when processing an adoption, validates users through the **Auth Service** and fetches pet details from the **Pet Service**.
4. Once the adoption is confirmed, the **Adoption Service** saves data to **AdoptionDB** and publishes a **PetAdopted** event to **RabbitMQ**.
5. The **Notification Service** consumes this event, processes it, and simulates notification delivery through **Postman**.
6. Each microservice interacts only through REST APIs or events, ensuring loose coupling.

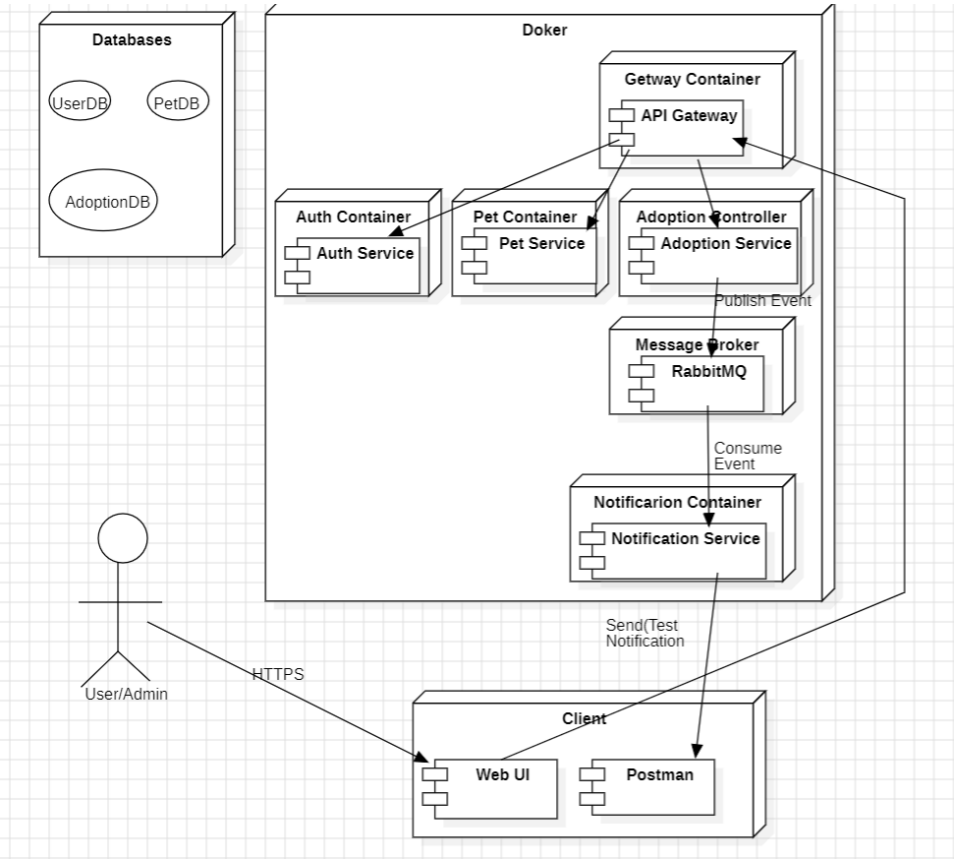
Component Diagram (Microservices Architecture)



The component diagram depicts four independent Spring Boot services: Auth Service, Pet Service, Adoption Service, and Notification Service. Each service manages its own domain and database, following the "database per service" pattern. The API Gateway routes client requests to the appropriate service and centralizes authentication and authorization. The Adoption Service publishes messages to the Message Broker when an adoption event occurs, and the Notification Service consumes those messages to send updates (tested through

Postman). This structure provides clear modular boundaries and allows each service to scale independently.

Deployment Diagram (Microservices Architecture)



The Deployment Diagram shows that each service runs in a separate Docker container:

- Containers: Auth, Pet, Adoption, Notification, Message Broker, and API Gateway.
- Each container communicates via REST (HTTP/JSON) or asynchronous events.
- Databases (UserDB, PetDB, AdoptionDB) are hosted separately but accessible only by their corresponding services.
- Clients access the system through HTTPS via the Gateway Container.

This structure supports independent deployment, scalability, and fault isolation but requires more infrastructure management compared to the monolith.

4.4 Pros & Cons

Advantages	Disadvantages
Scales individual modules independently (Pet, Adoption, Notification).	Complex deployment (multiple containers, databases).
Fault isolation: failure in one service doesn't crash others.	Requires DevOps setup (API Gateway, Broker, Monitoring).
Easier long-term maintainability and extension.	Cross-service transactions require Saga/Event patterns.

Advantages

The **Microservices Architecture** decomposes **PurrfectMatch** into independent, domain-specific services that can be developed, deployed, and scaled separately. This structure provides independent scalability, allowing each microservice such as Pet or Adoption to increase capacity based on demand without affecting the others. For instance, the Pet Service can handle a higher number of requests for browsing and filtering pets, while the Auth Service continues to operate at its normal load.

Another major strength is fault isolation. Failures in one service, such as Notification, do not propagate through the system, preventing complete outages and increasing overall resilience. Improved maintainability and modularity follow naturally from this structure. Each service manages its own codebase and database, making the system easier to update and reducing the risk of introducing errors across unrelated components.

Microservices also provide flexibility in technology choices. Different teams or modules can use different frameworks or database systems based on specific needs. Although **PurrfectMatch** uses Spring Boot across all services, this flexibility offers room for future evolution. Additionally, this architecture supports continuous integration and deployment (CI/CD). Developers can deploy or roll back updates for one service independently, improving development speed and minimizing downtime.

The architecture also enhances system extensibility. Adding new features, such as reporting or analytics, can be achieved by introducing new microservices without modifying the existing ones. Finally, asynchronous communication via a Message Broker (RabbitMQ) reduces dependency between services and supports event-driven operations, such as triggering adoption notifications without direct service coupling.

Disadvantages

While powerful, the microservices model introduces significant operational complexity. Each service must be deployed, monitored, and scaled independently, which requires container orchestration tools like Docker Compose or Kubernetes. This increases setup and maintenance effort, especially for smaller teams.

Another drawback is distributed data consistency. Since each service maintains its own database, global transactions (e.g., verifying user data during adoption) require patterns like Saga or two-phase commits to maintain consistency, adding technical overhead. Furthermore, network latency and communication overhead increase because services interact via REST APIs instead of local calls.

Microservices also consume more resources. Each instance runs in its own environment with its own dependencies, resulting in higher memory and CPU usage. Managing security becomes more complex, as authentication and authorization must be consistently enforced across all services or centralized through an API Gateway.

Debugging distributed systems is another challenge. Error tracing and monitoring require specialized tools for distributed logging and tracing. Finally, the learning curve is higher. Developers and operators must understand service discovery, load balancing, and container networking to manage the full system effectively.

Despite these challenges, microservices provide a clear path toward scalability and modularity, but the trade-offs in complexity and resource consumption make them less practical for the current **PurrfectMatch** implementation.

5. Event-Driven Architecture

5.1 Description (Structure & Data Flow)

In the **Event-Driven Architecture**, **PurrfectMatch** uses asynchronous communication through a central Message Bus (RabbitMQ). Instead of direct REST calls between services, events trigger actions in other services automatically, improving decoupling and scalability.

Main Components

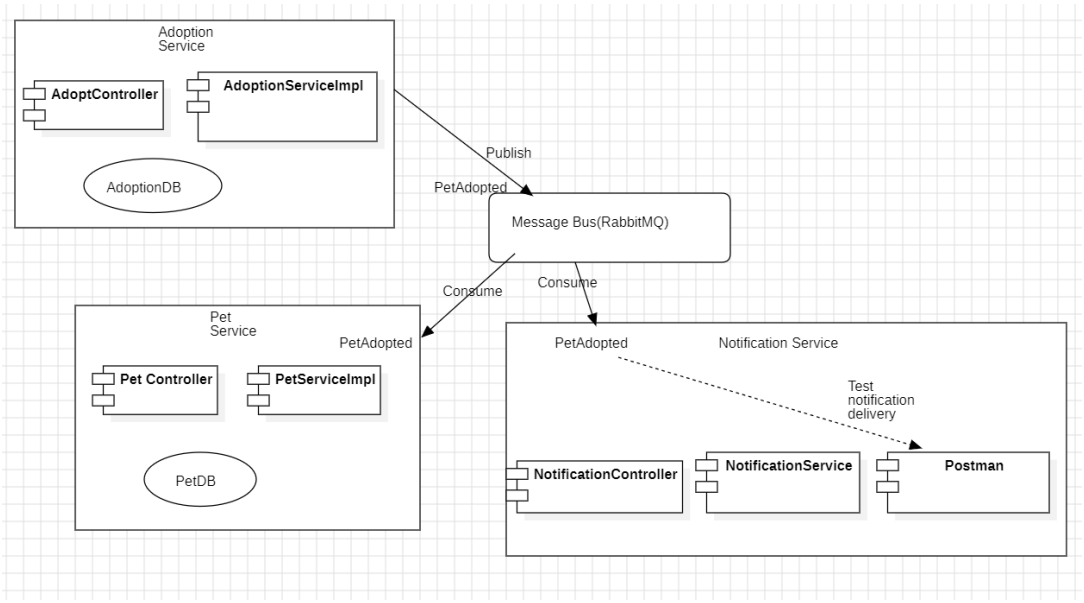
- **Adoption Service:** Publishes an event (PetAdopted) to the message bus when an adoption request is approved. Persists data in AdoptionDB.
- **Pet Service:** Subscribes to the message bus to receive PetAdopted events, updating pet status in PetDB.
- **Notification Service:** Subscribes to the same event and sends simulated adoption confirmations through **Postman**.
- **Message Bus (RabbitMQ):** Routes published events to all subscribed services, ensuring asynchronous communication.
- **Databases:** Each service has its own database (AdoptionDB, PetDB).
- **Client Layer:** The user interacts via **Web UI** or **Postman** to request adoptions.

Interactions and Data Flow

1. A **User** initiates an adoption request through the **Web UI** or **Postman**.
2. The **Adoption Service** processes the request, stores adoption data in AdoptionDB, and publishes a PetAdopted event to **RabbitMQ**.

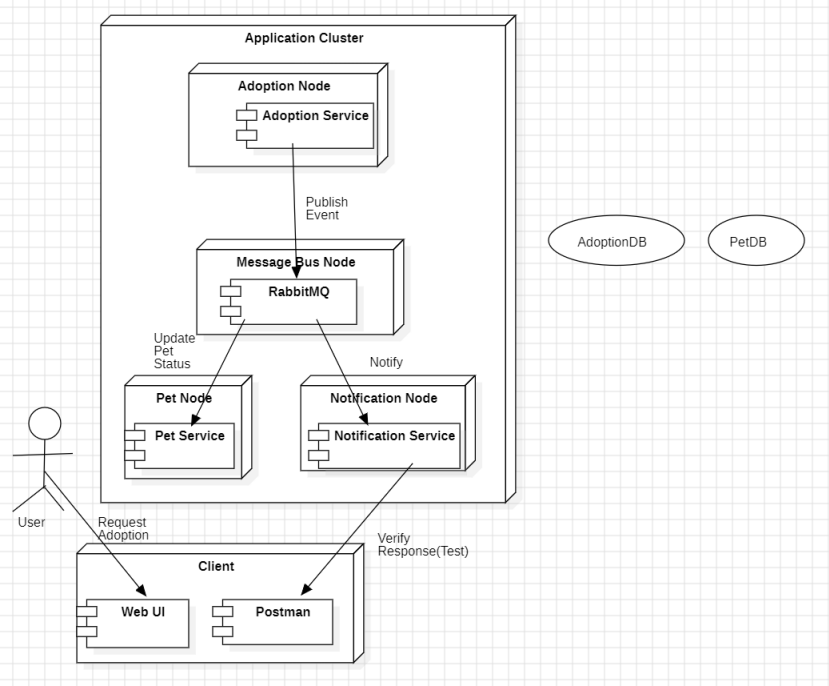
3. The **Message Bus** delivers this event asynchronously to the **Pet Service** and **Notification Service**.
4. The **Pet Service** updates the pet's status in PetDB.
5. The **Notification Service** sends (simulated) confirmation messages through **Postman**, completing the process.
6. All communications occur asynchronously, so services remain operational even if others are temporarily unavailable.

Component Diagram (Event-Driven Architecture)



The component diagram shows the event-driven flow of communication between services. When an adoption request is completed, the Adoption Service publishes an event (PetAdopted) to the Message Bus (Kafka/RabbitMQ). The Notification Service consumes this event and triggers a simulated notification tested via Postman. The Pet Service also subscribes to the same event to update pet availability. This structure demonstrates loose coupling and asynchronous coordination between components.

Deployment Diagram (Event-Driven Architecture)



As shown in the **Deployment Diagram**, services are distributed across multiple nodes within an **Application Cluster**:

- Adoption Node, Pet Node, and Notification Node host their respective services.
- Message Bus Node hosts RabbitMQ for event routing.
- Each node connects to its local database (AdoptionDB, PetDB).
- Clients (Web UI and Postman) interact through event-triggering requests rather than synchronous HTTP responses.

This structure enables high resilience, scalability, and loose coupling but introduces eventual consistency and increased complexity in monitoring asynchronous flows.

5.4 Pros & Cons

Advantages	**Disadvantages**
Highly scalable and loosely coupled.	Eventual consistency; delayed updates.
Ideal for notifications and asynchronous workloads.	More complex debugging and broker maintenance.
Easy to extend (new event consumers).	Requires message broker setup and management.

Advantages

The Event-Driven Architecture introduces high scalability and loose coupling by enabling asynchronous communication between services. Each component, such as Adoption, Pet, or Notification, operates independently and reacts to events distributed via a message bus like RabbitMQ or Kafka. This design allows services to scale independently and continue functioning even if others are temporarily offline.

A major benefit is resilience and reliability. Messages remain in the broker queue until consumers (e.g., Notification Service) are available to process them, preventing data loss during downtime. It also offers faster user experiences, since tasks like sending notifications are handled asynchronously. Users receive immediate confirmation of adoption requests, while background services complete notifications later.

Event-driven systems provide better extensibility than synchronous architectures. New services, such as analytics or monitoring, can subscribe to existing event topics without modifying existing services. This ensures that future expansion of **PurrfectMatch** such as adding recommendation systems or adoption history reports can happen seamlessly.

Additionally, the architecture supports distributed scaling. Each consumer can scale independently based on the number of incoming messages. The event-driven model also encourages a natural evolution path from a monolithic or microservices system, allowing gradual migration while keeping services loosely coupled.

Disadvantages

However, this approach introduces eventual consistency, since events are processed asynchronously. It may take time before pet statuses or adoption confirmations are reflected in all services. Managing error handling and reliability also becomes more complex, requiring dead-letter queues and message replay strategies to avoid duplicates or lost events.

Another major challenge is monitoring and debugging asynchronous systems. Tracing event flow across multiple queues and consumers is difficult, and identifying failed or delayed events requires specialized tools. Setting up and maintaining a message broker such as RabbitMQ adds infrastructure complexity and resource overhead.

The learning curve is also steeper. Developers must understand event serialization, schema versioning, and idempotent consumer design to ensure reliable processing. Finally, testing becomes more complicated, since verifying asynchronous behaviour requires message simulation rather than direct HTTP testing.

Despite these disadvantages, event-driven architecture provides strong advantages in flexibility and responsiveness. It is an ideal candidate for future iterations of **PurrfectMatch** once the system requires real-time processing and higher throughput.

6. Comparison and Evaluation (Numeric)

Quality Attribute	Monolithic	Microservices	Event-Driven	Explanation
1. Scalability	2	5	4	Monolith scales as a whole; microservices and event-driven scale specific modules.
2. Availability / Fault Tolerance	2	5	4	Distributed services isolate failures; monolith does not.
3. Security & Privacy	4	3	3	Centralized auth in monolith; distributed auth harder to manage.
4. Maintainability / Modularity	3	5	4	Services improve modularity and maintainability.
5. Extensibility	3	5	5	Event-driven easiest to extend with new consumers.
6. Interoperability (Postman / REST)	3	4	5	Event-driven supports multiple integration types.
7. Data Consistency	5	2	2	Monolith has single ACID DB; others use eventual consistency.
8. Operational Complexity	1	4	4	Distributed systems require gateway, broker, and monitoring.
9. Implementation Speed (for course)	5	2	2	Monolith fastest for current academic scope.

(Scale 1–5: 1 = lowest, 5 = highest)

7.Final Comparison and Justified Choice

The three investigated architectures (Monolithic, Microservices, and Event-Driven) offer distinct trade-offs for scalability, maintainability, and complexity.

- The **Monolithic architecture** provides strong internal consistency, simple deployment, and straightforward testing, making it ideal for academic or small-team development. All modules (controllers, services, repositories, and configuration) are packaged together in a single deployable Spring Boot application.
- The **Microservices architecture** introduces clear modular separation and scalability but requires additional infrastructure (API gateway, container orchestration, service discovery, inter-service communication, and monitoring). This complexity is unnecessary for the current scope of PurrfectMatch.
- The **Event-Driven architecture** enables high scalability and loose coupling through asynchronous events, but it introduces delayed consistency and complex debugging that exceed the system's current operational needs.

After evaluating all three alternatives, the **Monolithic architecture** has been selected as the final and permanent architecture for the PurrfectMatch project. This structure will remain the foundation for future milestones and implementation because:

- It fully supports all existing functionalities: user management, pet browsing, adoption requests, and notifications.
- It ensures data consistency and simple deployment in a single environment (Docker or local server).
- It allows clear separation of concerns through layered design (controllers, services, repositories, domain) while remaining extensible for future evolution into microservices or event-driven systems if the project expands.
- It aligns with the team's current resources and the course requirements, balancing maintainability and performance without introducing unnecessary complexity.

Consequently, the **PurrfectMatch** system will be architected, implemented, and maintained as a **Monolithic** application, ensuring architectural consistency, reliability, and a solid basis for future scalability and enhancement.