To address the architectural flaw of relying on a monolithic architecture in Yahoo's email and news services, we can propose a transition to a microservices architecture. This approach will enhance scalability, maintainability, and resilience. Below are the basic-level solutions, architectural patterns, and a proposed architecture to implement.

Problem: Monolithic Architecture

Issues with Monolithic Architecture:

1. **Scalability**: The entire application must be scaled together, which can lead to resource inefficiencies.
2. **Deployment Challenges**: Any change in one part of the application requires redeploying the entire system, increasing downtime and risk.
3. **Single Point of Failure**: If one component fails, it can bring down the entire application.
4. **Development Bottlenecks**: Teams may struggle to work independently on different features, leading to slower development cycles.

Proposed Solution: Microservices Architecture

1. Transition to Microservices:

- Break down the monolithic application into smaller, independent services that can be developed, deployed, and scaled independently.

2. Key Microservices:

- **User Service**: Manages user accounts and authentication.
- **Email Service**: Handles sending, receiving, and storing emails.
- **News Service**: Manages news articles, categories, and user preferences.
- **Notification Service**: Sends notifications to users (e.g., new emails, news updates).

Architectural Patterns

1. **Microservices Pattern**:
    - Each service is a standalone application with its own database, allowing for independent scaling and deployment.
2. **API Gateway Pattern**:
    - An API Gateway acts as a single entry point for clients, routing requests to the appropriate microservices. It can also handle cross-cutting concerns like authentication, logging, and rate limiting.
3. **Service Discovery Pattern**:
    - Use a service discovery mechanism (e.g., Eureka, Consul) to allow services to find and communicate with each other dynamically.
4. **Event-Driven Architecture**:

- Implement an event-driven model where services communicate through events (e.g., using a message broker like RabbitMQ or Kafka). This decouples services and allows for asynchronous processing.

5. **Database per Service Pattern**:
   - Each microservice has its own database schema, ensuring that services are loosely coupled and can evolve independently.

A SIMPLE CODE FOR A SCENERIO IN WHICH THIS PROBLEM IS SOLVED:

BOOK STORE APPLICATION:

1. USER SERVICES:

```
package com.example.userservice;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

1.1. package com.example.userservice;

```
import org.springframework.web.bind.annotation.*;


import java.util.HashMap;
import java.util.Map;
```

```
@RestController
@RequestMapping("/users")
public class UserController {
    private final Map<String, String> users = new HashMap<>();

    @PostMapping("/register")
    public String registerUser (@RequestParam String userId,
@RequestParam String userName) {
        users.put(userId, userName);
        return "User  registered successfully!";
    }

    @GetMapping("/{userId}")
    public String getUser (@PathVariable String userId) {
        return users.getOrDefault(userId, "User  not found!");
    }
}
```

2.

BOOK SERVICE:

```
package com.example.bookservice;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BookServiceApplication {
    public static void main(String[] args) {
```

```
        SpringApplication.run(BookServiceApplication.class, args);
    }
}
```

2.2.

```
package com.example.bookservice;

import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/books")
public class BookController {
    private final Map<String, String> books = new HashMap<>();

    @PostMapping("/add")
    public String addBook(@RequestParam String bookId,
@RequestParam String bookTitle) {
        books.put(bookId, bookTitle);
        return "Book added successfully!";
    }

    @GetMapping("/{bookId}")
    public String getBook(@PathVariable String bookId) {
        return books.getOrDefault(bookId, "Book not found!");
    }

    @GetMapping
    public Map<String, String> getAllBooks() {
        return books;
    }
}
```

3. **ORDER SERVICE:**

```
package com.example.bookservice;

import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/books")
public class BookController {
    private final Map<String, String> books = new HashMap<>();

    @PostMapping("/add")
    public String addBook(@RequestParam String bookId,
@RequestParam String bookTitle) {
        books.put(bookId, bookTitle);
        return "Book added successfully!";
    }

    @GetMapping("/{bookId}")
    public String getBook(@PathVariable String bookId) {
        return books.getOrDefault(bookId, "Book not found!");
    }

    @GetMapping
    public Map<String, String> getAllBooks() {
        return books;
    }
}
```

3.3. package com.example.orderservice;

```
import org.springframework.web.bind.annotation.*;
```

```
import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/orders")
public class OrderController {
    private final List<String> orders = new ArrayList<>();

    @PostMapping("/place")
    public String placeOrder(@RequestParam String userId,
@RequestParam String bookId) {
        String order = "Order placed by user " + userId + " for book " +
bookId;
        orders.add(order);
        return "Order placed successfully!";
    }

    @GetMapping
    public List<String> getAllOrders() {
        return orders;
    }
}
```