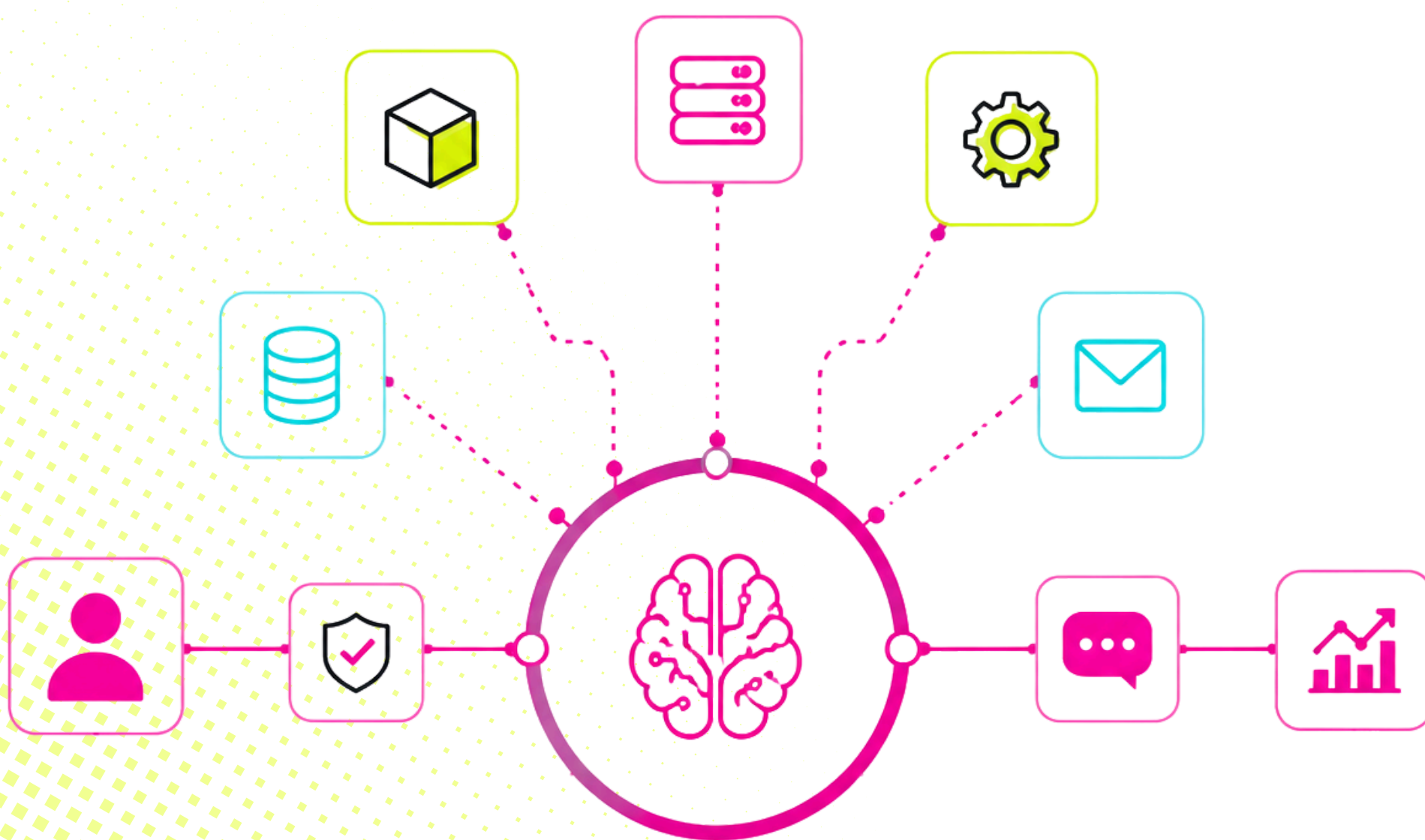




Created by [Robbert van Vlijmen](#)

14 Microservices Patterns Every Developer Must Know

Design better distributed systems with proven architecture patterns

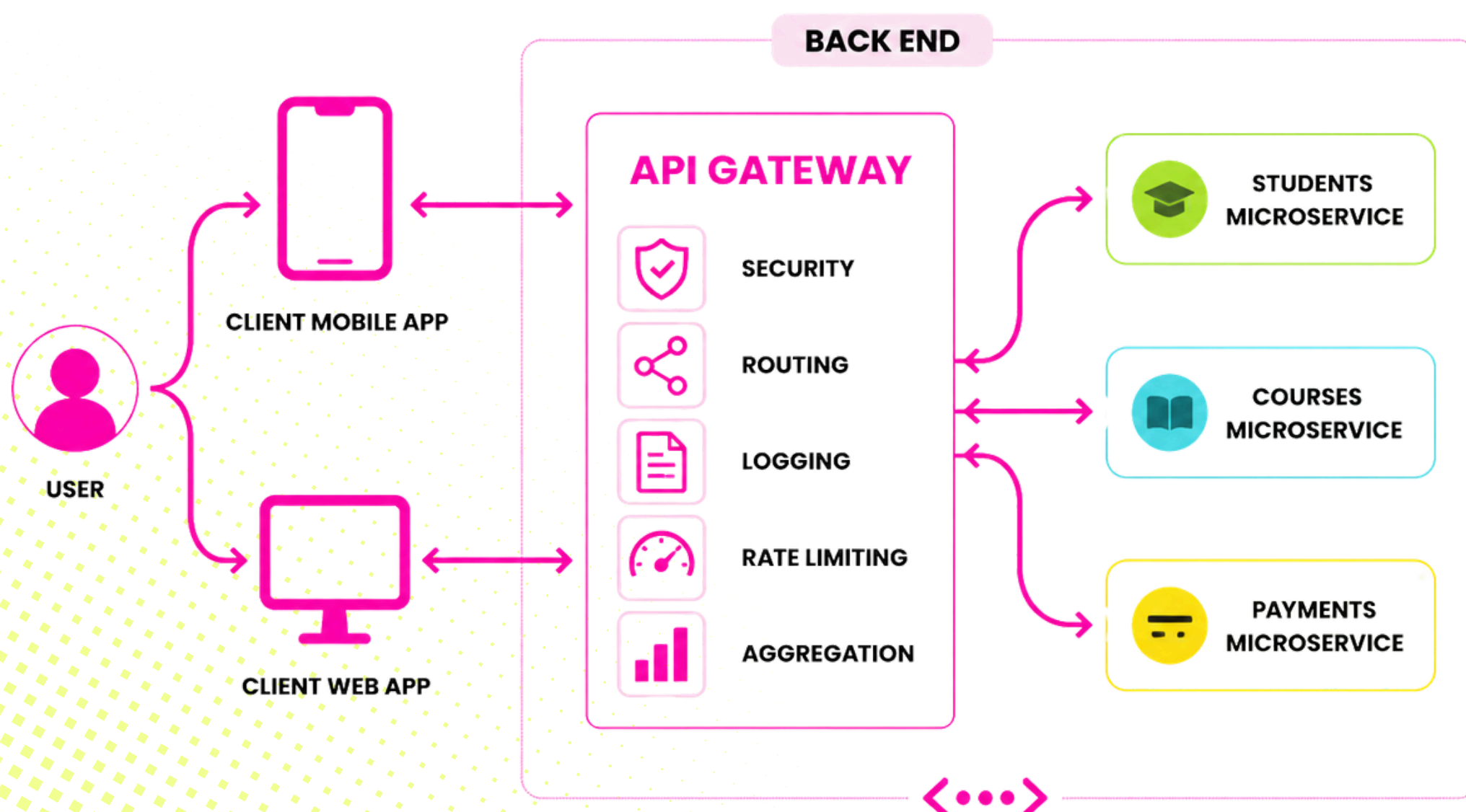




01

GATEWAY PATTERN

- ✓ Routes client requests to the right microservices through a centralized API Gateway layer.
- ✓ Manages key gateway functions like authentication, security, logging, and rate limiting.
- ✓ Improves scalability and reliability with load balancing and request aggregation across services.

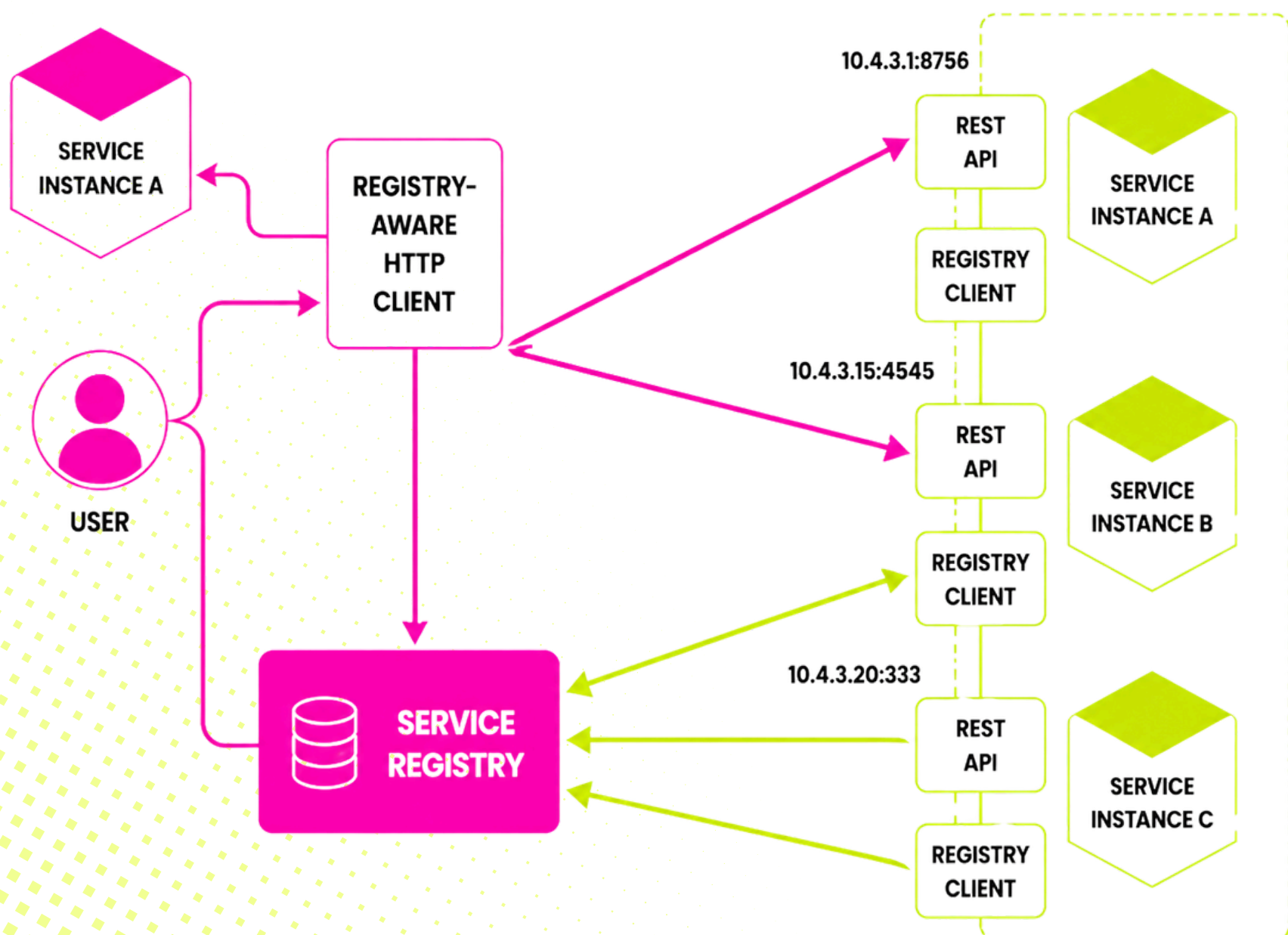




02

SERVICE REGISTRY PATTERN

- ✓ Automatically registers and tracks microservices to maintain an up-to-date service directory.
- ✓ Enables dynamic service discovery by allowing applications to locate available service instances.
- ✓ Improves communication and scalability by simplifying interactions between distributed services.

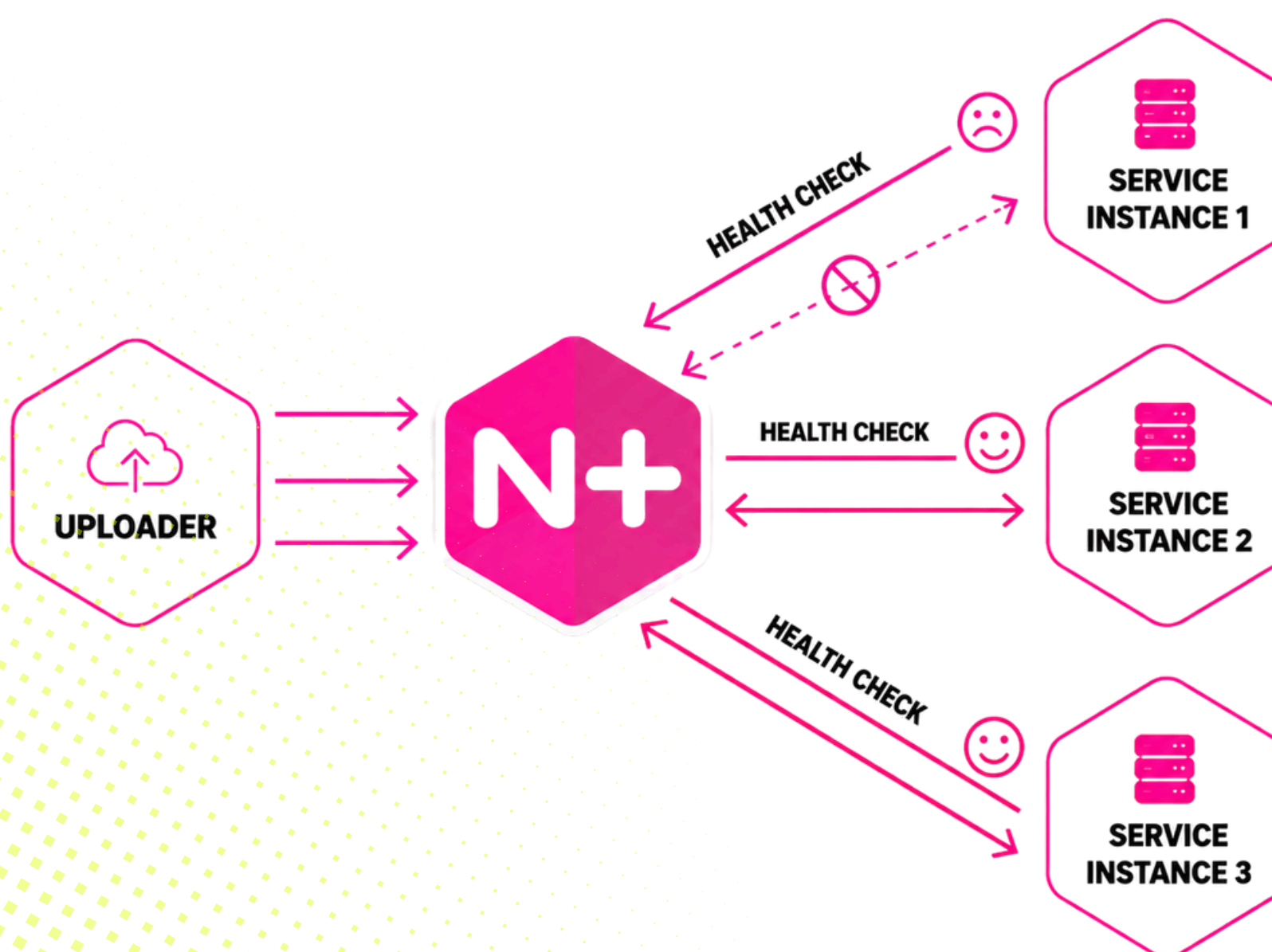




03

CIRCUIT BREAKER PATTERN

- ✓ Prevents cascading failures by monitoring service health and isolating failing components.
- ✓ Temporarily blocks requests to unavailable services to avoid system overload.
- ✓ Provides fallback mechanisms to maintain application reliability during service failures.

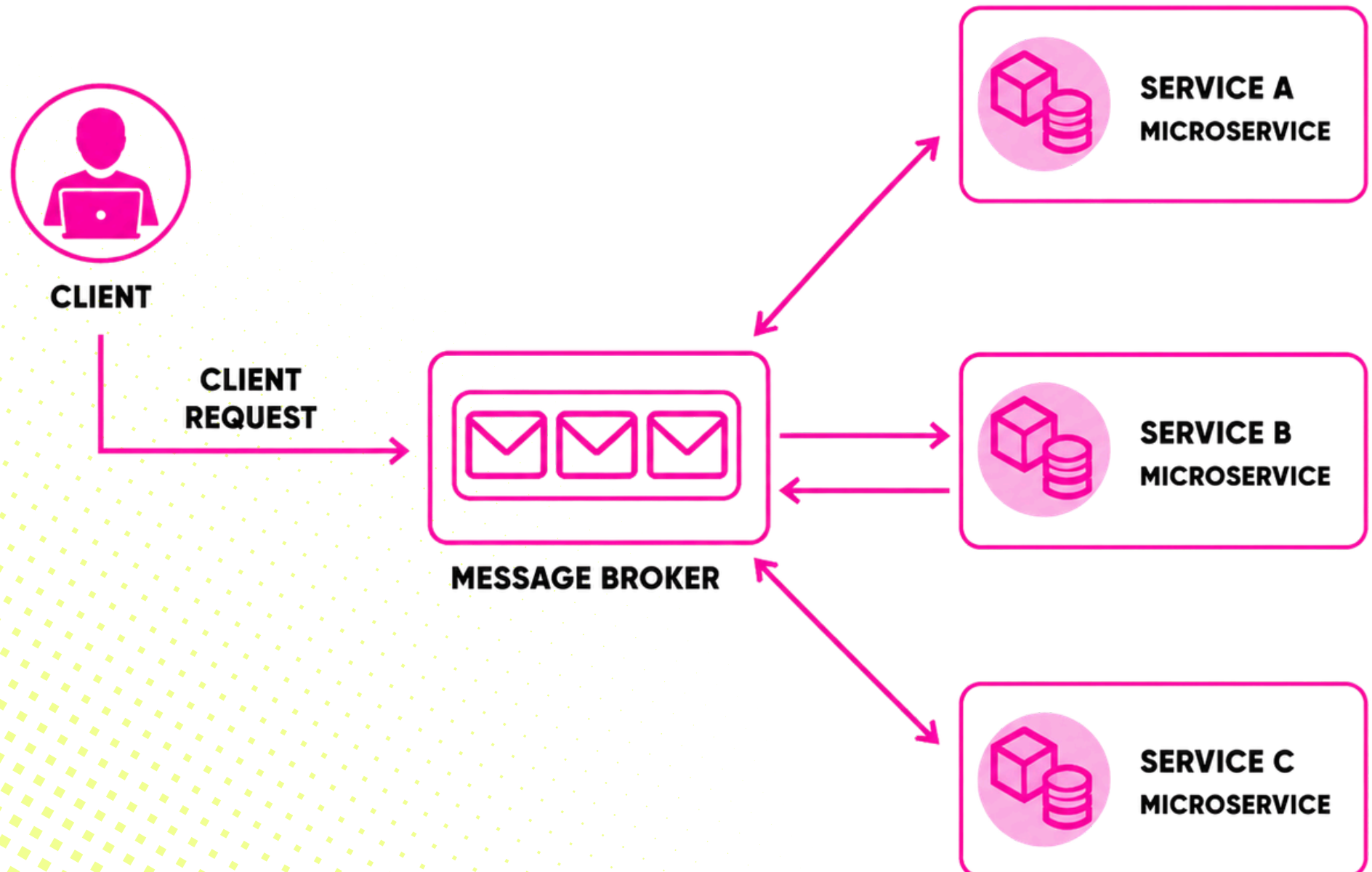




04

SAGA PATTERN

- ✓ Handles distributed transactions across multiple microservices without relying on a single database transaction.
- ✓ Breaks complex workflows into smaller local transactions managed by individual services.
- ✓ Ensures data consistency and reliability through coordinated actions and compensation mechanisms.

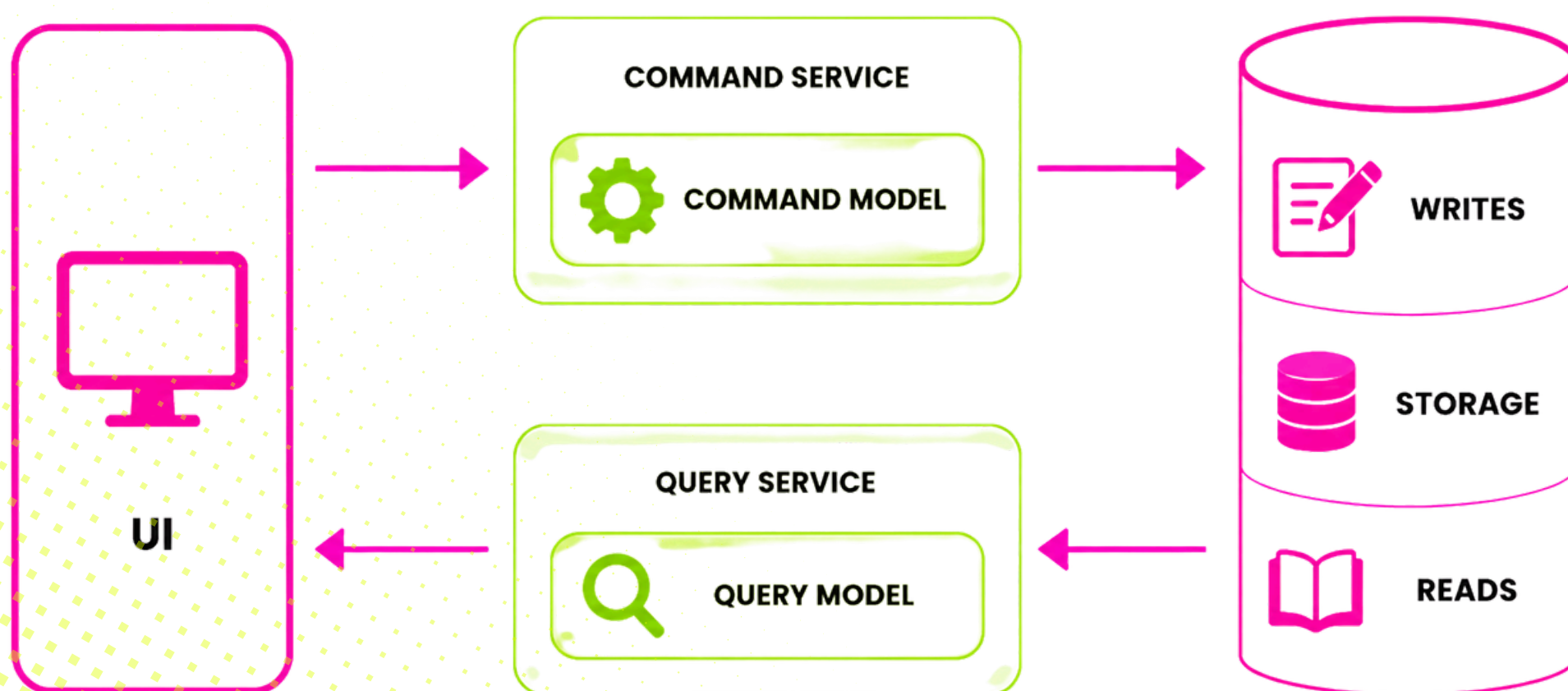




05

CQRS PATTERN

- ✓ Separates read and write operations into independent models for better system organization.
- ✓ Optimizes performance by allowing commands and queries to scale independently.
- ✓ Improves scalability and flexibility by managing data access patterns efficiently.

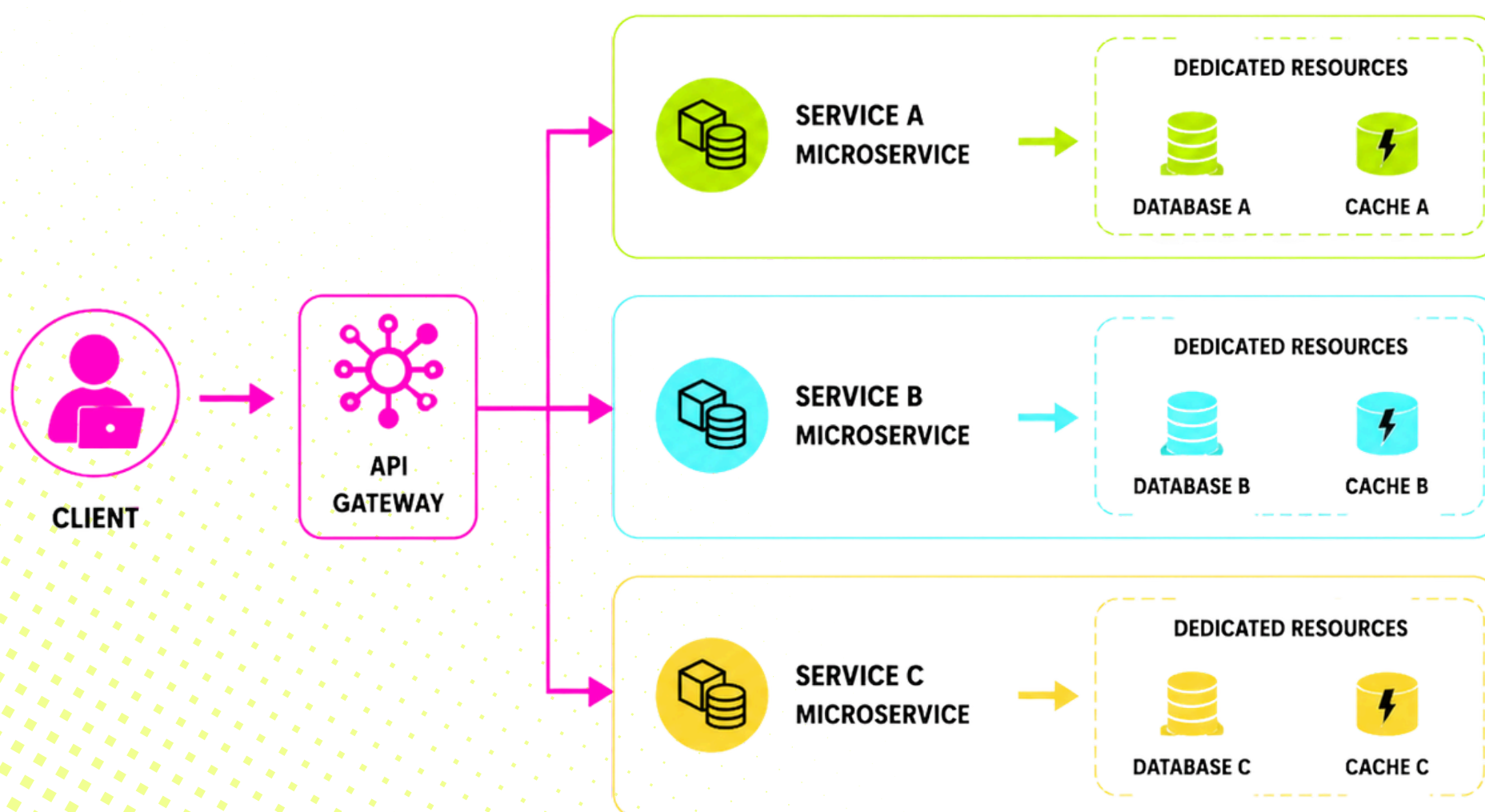




06

BULKHEAD PATTERN

- ✓ Isolates failures by separating system components into independent sections.
- ✓ Prevents cascading issues by limiting the impact of failures to specific services.
- ✓ Improves system resilience and availability by ensuring critical services continue running.

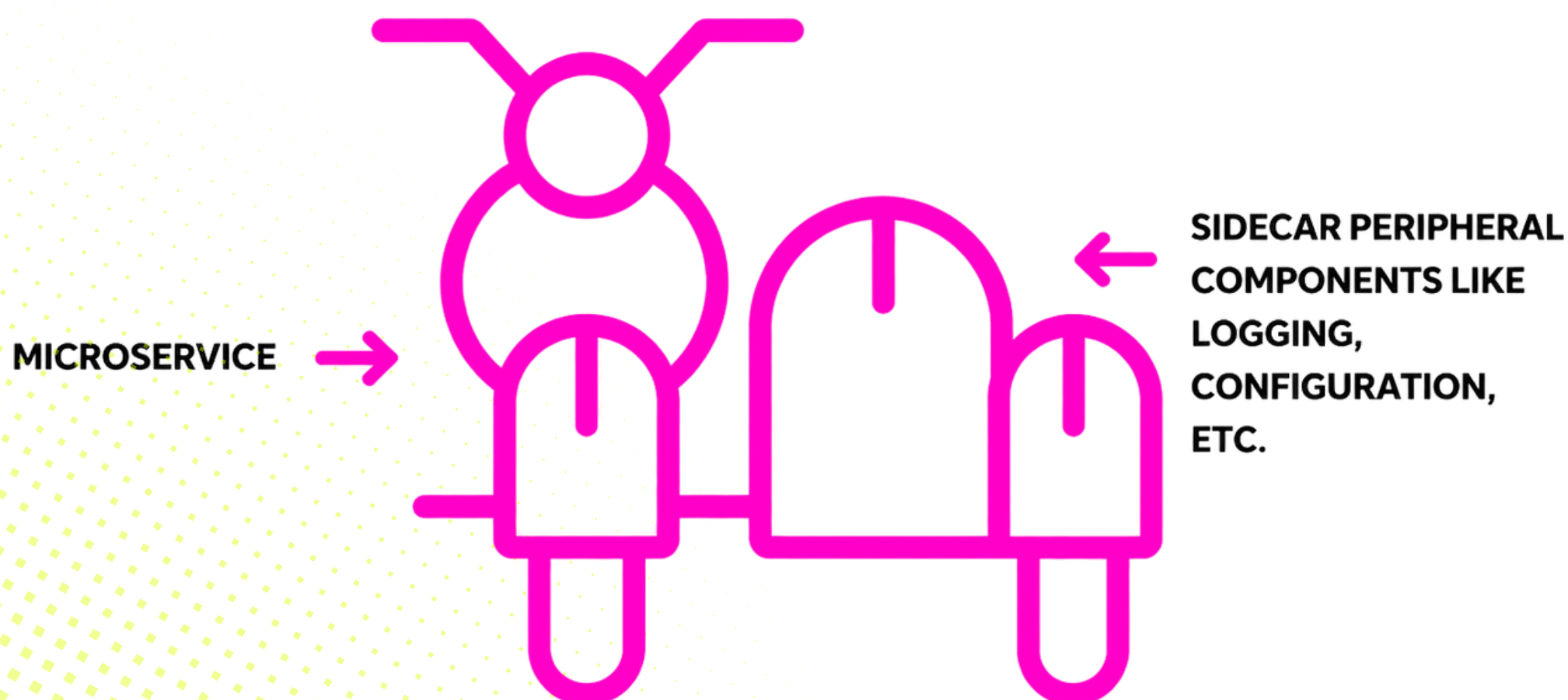




07

SIDECAR PATTERN

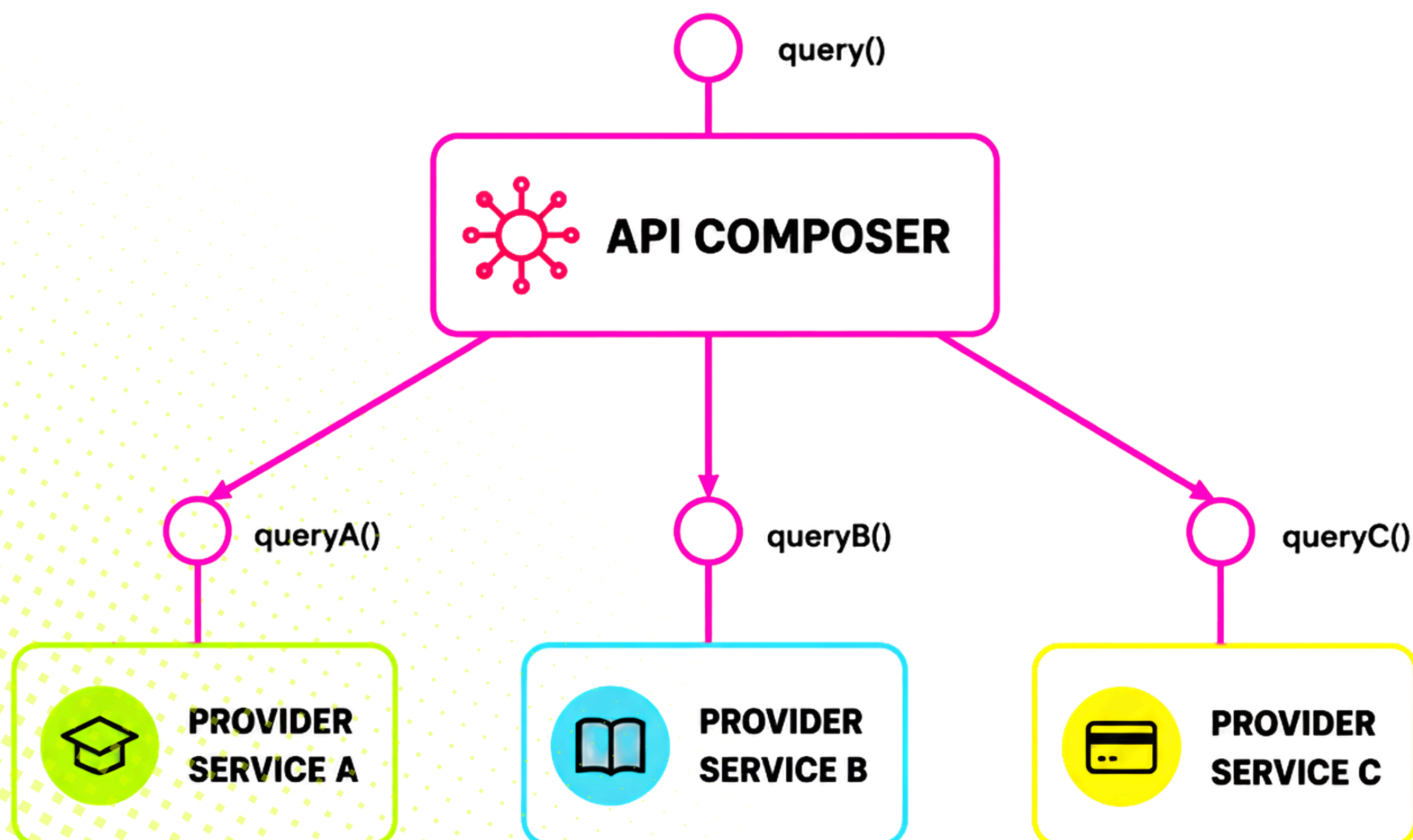
- ✓ Deploys a dedicated sidecar service alongside the main microservice to manage supporting tasks.
- ✓ Handles cross-cutting concerns such as monitoring, logging, configuration, and authentication.
- ✓ Improves service maintainability by separating operational features from core application logic.





API COMPOSITION PATTERN

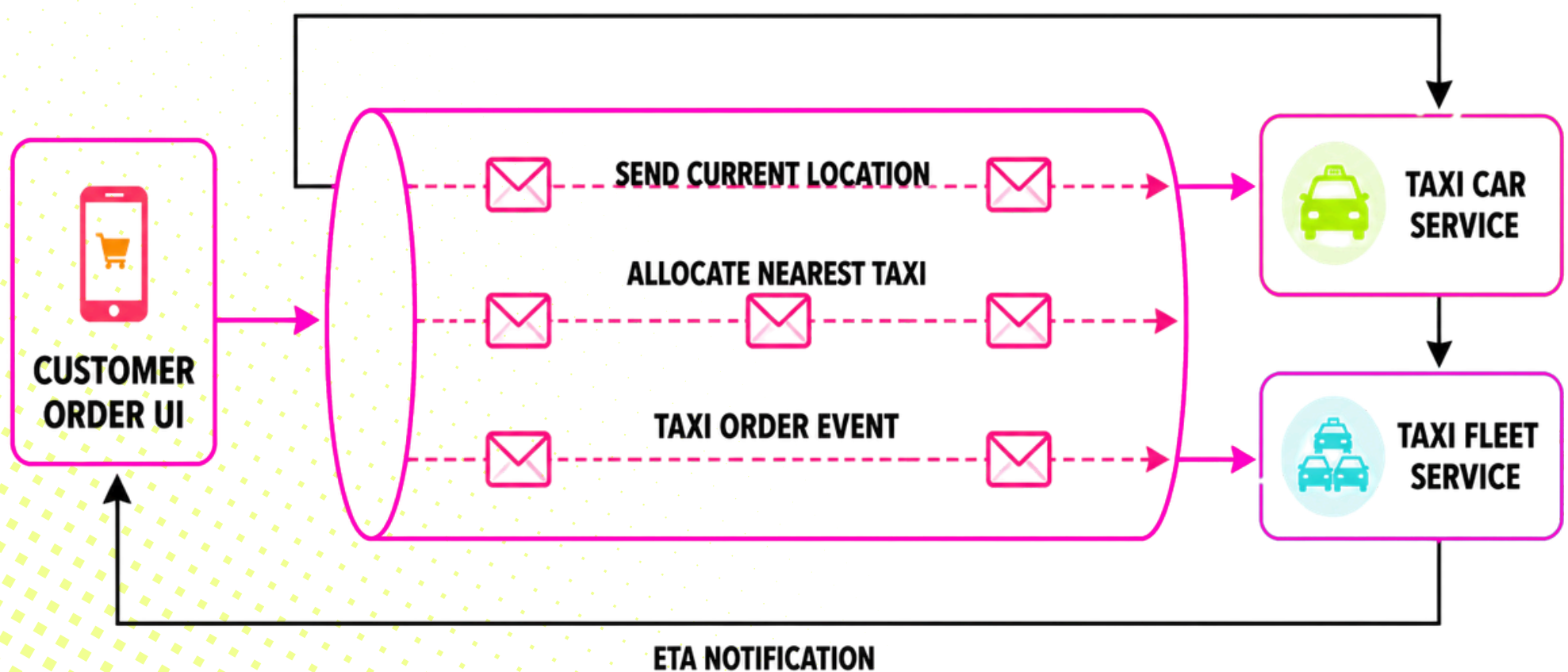
- ✓ Aggregates data from multiple microservices into a single API response for clients.
- ✓ Simplifies client interactions by reducing the need to call multiple services separately.
- ✓ Creates richer and more efficient APIs by combining capabilities across different services.





EVENT-DRIVEN ARCHITECTURE PATTERN

- ✓ Enables communication between microservices through asynchronous event-based interactions.
- ✓ Reduces service dependencies by allowing components to operate independently.
- ✓ Improves scalability and flexibility by efficiently handling distributed workflows.

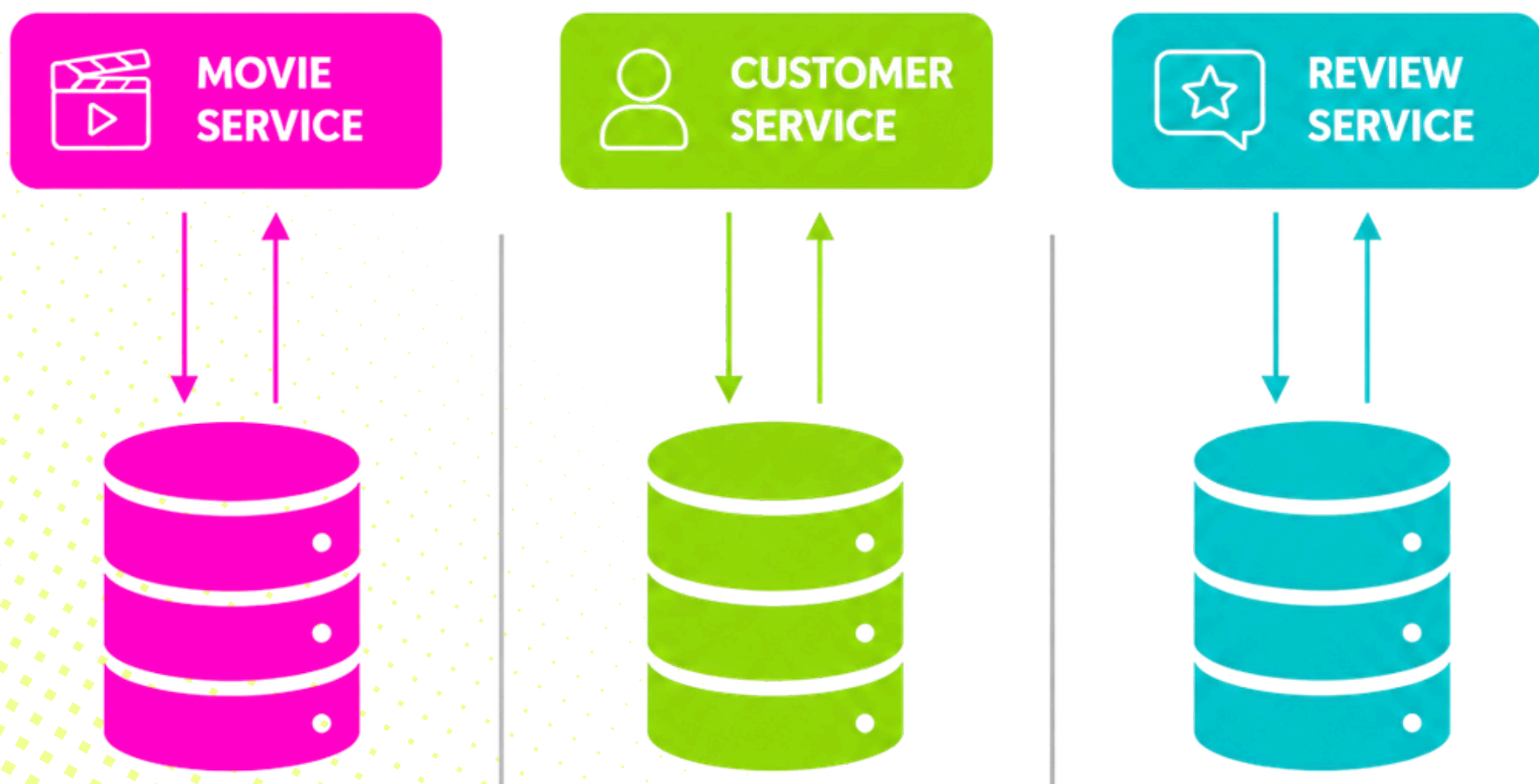


10



DATABASE PER SERVICE PATTERN

- ✓ Assigns a separate database to each microservice for independent data management.
- ✓ Reduces service dependencies by preventing direct database sharing between services.
- ✓ Improves scalability and autonomy by allowing each service to evolve independently.

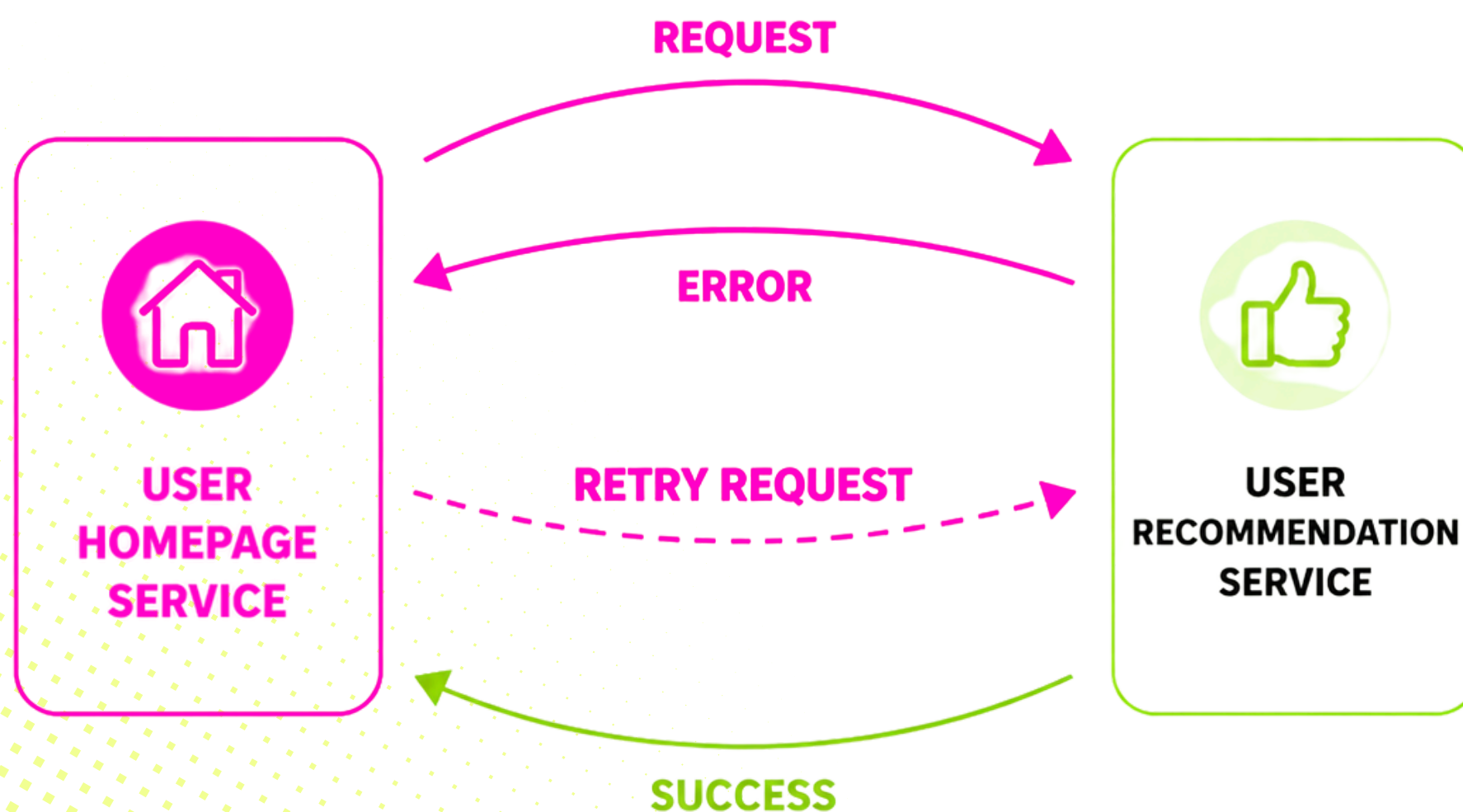




11

RETRY PATTERN

- ✓ Automatically retries failed requests to handle temporary service failures.
- ✓ Improves reliability by increasing the chances of successful operations.
- ✓ Reduces the impact of transient errors through controlled retry mechanisms.

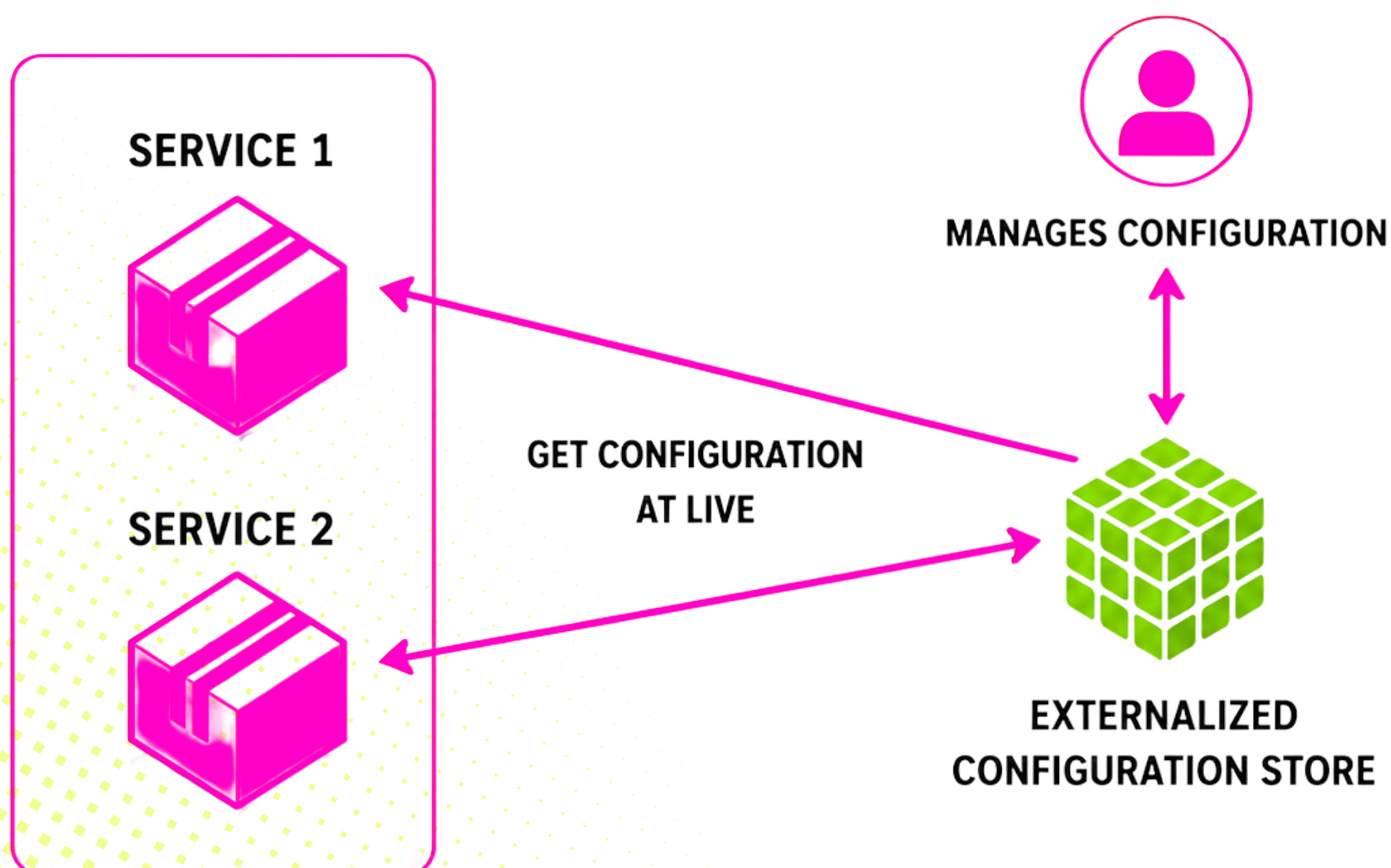




12

CONFIGURATION EXTERNALIZATION PATTERN

- ✓ Stores configuration separately from application code to improve flexibility and maintainability.
- ✓ Allows services to fetch configurations dynamically from a centralized configuration store.
- ✓ Simplifies updates and deployments by changing settings without modifying application code.

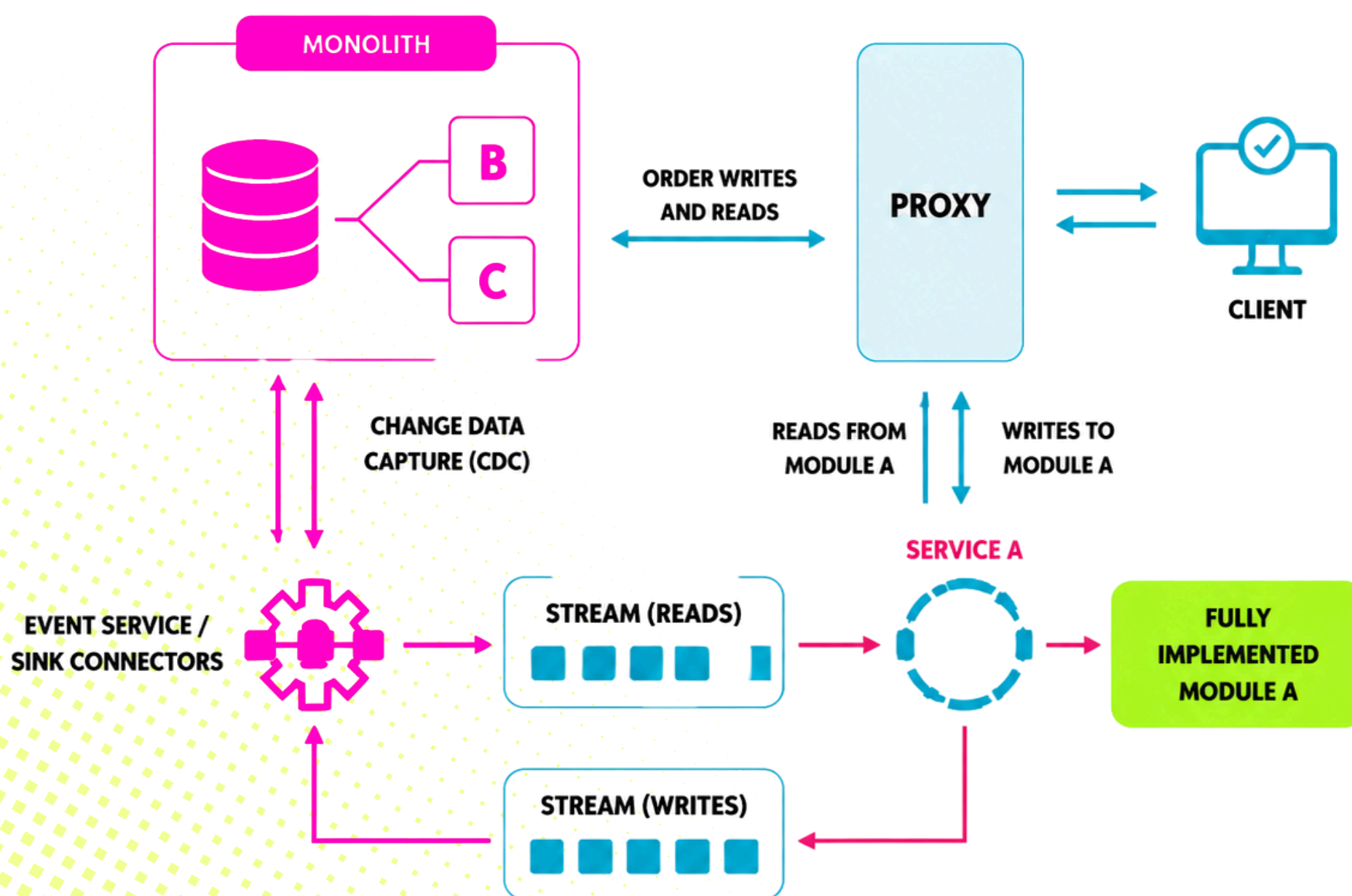


13



STRANGLER FIG PATTERN

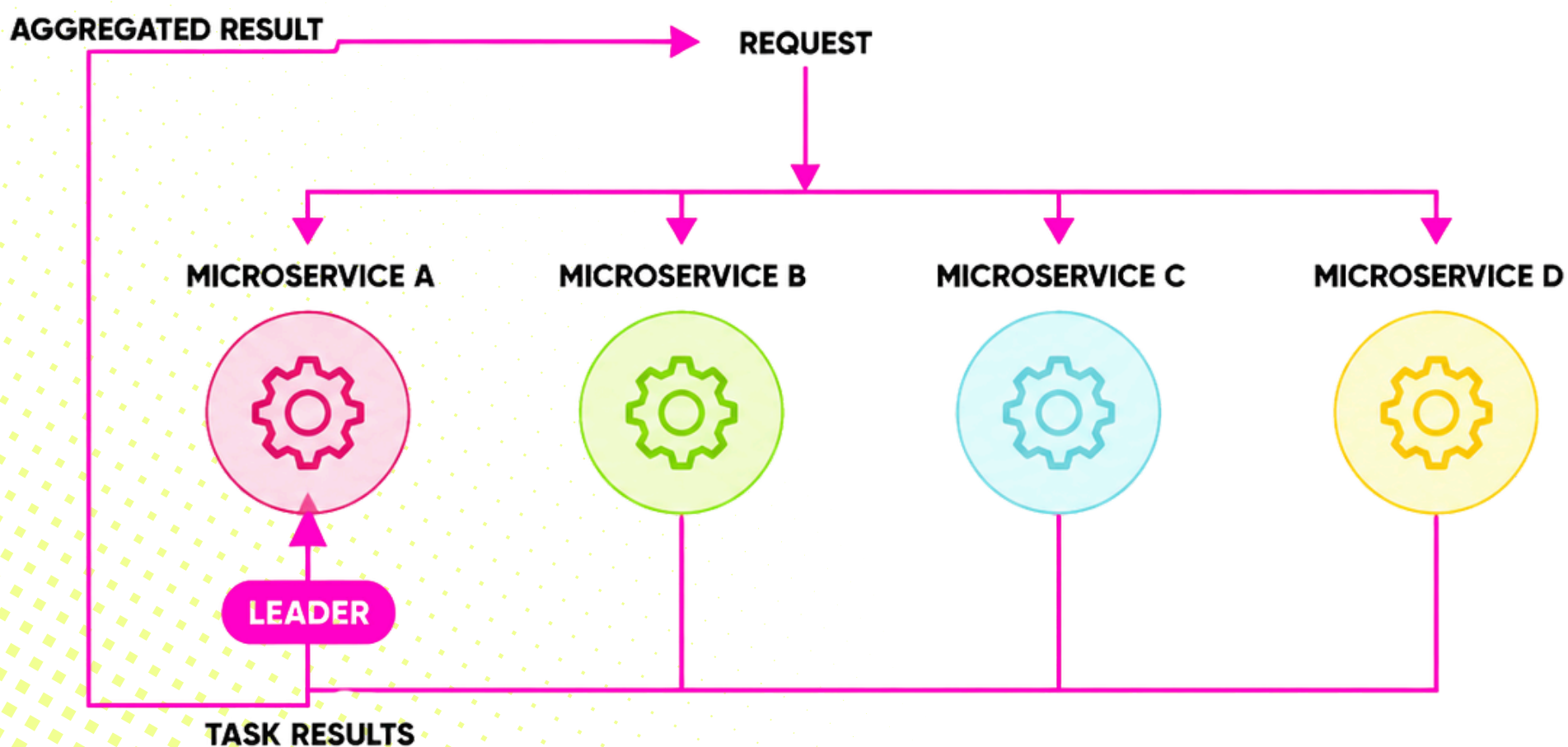
- ✓ Incrementally replaces legacy components with modern services instead of performing a complete system rewrite.
- ✓ Allows gradual migration by running new and old systems together during the transition.
- ✓ Reduces migration risks while completely replacing the legacy system over time.





LEADER ELECTION PATTERN

- ✓ Selects a single leader instance among multiple microservices to coordinate activities.
- ✓ Handles shared responsibilities like task distribution, synchronization, and decision-making.
- ✓ Improves system consistency by avoiding conflicts between multiple service instances.





BUILD SCALABLE. DELIVER RELENTLESSLY.



Design, deploy, and evolve microservices with patterns that power resilience and performance.

Follow for more AI trends