5 CLOUD NATIVE ARCHITECTURE PATTERNS FOR 2024

Becoming “cloud native” is often cited as the end goal for migrating or building applications today. But depending on who you ask, you’ll probably get a lot of different definitions of what exactly it means. Overall, the majority boils down to this: Cloud native is an approach to building and running scalable applications to take full advantage of cloud-based services and delivery models.

WHAT IS A CLOUD-NATIVE APPLICATION?

The cloud native landscape is rapidly evolving, demanding architectures that are both scalable and agile. These architectures need to be designed for distributed environments, embracing microservices and containerization. To meet these demands, cloud native architecture patterns offer proven approaches for building resilient and efficient applications.

In this article, we’ll explore the top 5 cloud native architecture patterns you should know to prepare yourselves for 2024:

Sidecar/Sidekick Pattern

Imagine a tiny companion riding alongside your motorcycle. That’s the essence of the Sidecar/Sidekick pattern. This pattern involves deploying a small container alongside the main application container. Think of it as a “sidecar” providing essential functionality like logging, monitoring, security, or even an API gateway.

Benefits:

Decoupling: Separates core application logic from auxiliary functions, improving modularity and resilience.

Scalability: Sidecars can be scaled independently according to their specific needs.

Flexibility: Different sidecars can be deployed with different applications, offering a modular approach.

Example:

Imagine an e-commerce application with a sidecar container handling payment processing. This sidecar could handle encryption, communication with payment gateways, and fraud detection, keeping the core application focused on order management and product listings.

Ambassador Pattern

Think of an ambassador as a diplomat representing your interests. Similarly, the Ambassador pattern uses a container to handle external traffic before it reaches the main application. This ambassador can handle tasks like authentication, authorization, rate limiting, and load balancing.

Benefits:

Security: Acts as a central point for enforcing security policies and protecting the application.

Scalability: Enables scaling the ambassador independently to handle increased traffic.

Load Balancing: Distributes traffic across multiple application instances for improved performance.

Example:

Consider a social media platform with an ambassador container handling user logins. This ambassador could validate credentials, assign user roles, and perform rate limiting to prevent security breaches and ensure smooth operation.

Scatter/Gather Pattern

Imagine dividing a large task into smaller, manageable chunks and distributing them among workers. That's the essence of the Scatter/Gather pattern. This pattern involves a "scatter" process that distributes tasks across multiple worker processes and a "gather" process that collects the results and returns them to the client.

Benefits:

Parallelization: Enables concurrent execution of tasks, significantly improving performance.

Scalability: Workers can be scaled horizontally to handle increasing workloads.

Fault Tolerance: If a worker fails, others can pick up the slack, ensuring resilience.

Example:

Consider a video streaming platform that utilizes the Scatter/Gather pattern for video transcoding. The scatter process would divide the video into segments and distribute them to worker processes for transcoding. The gather process would then collect the transcoded segments and assemble them into a single video file.

Backend for Frontends (BFF) Pattern

Ever felt frustrated with a website designed for a different device? The BFF pattern addresses this issue. It introduces a dedicated API service for each type of client application (mobile, web, etc.). This API service tailors its responses to the specific needs of each client, providing a more optimal user experience.

Benefits:

Client-Specific Optimization: Tailors data and functionality to each client's unique needs.

Improved Performance: Reduces data transfer by only providing relevant information to each client.

Decoupling: Isolates the main application from client-specific concerns.

Example:

Imagine a news website with a BFF for mobile and web clients. The mobile BFF could deliver optimized content and images for smaller screens, while the web BFF could provide a richer experience with additional features and information.

CQRS (Command Query Responsibility Segregation)

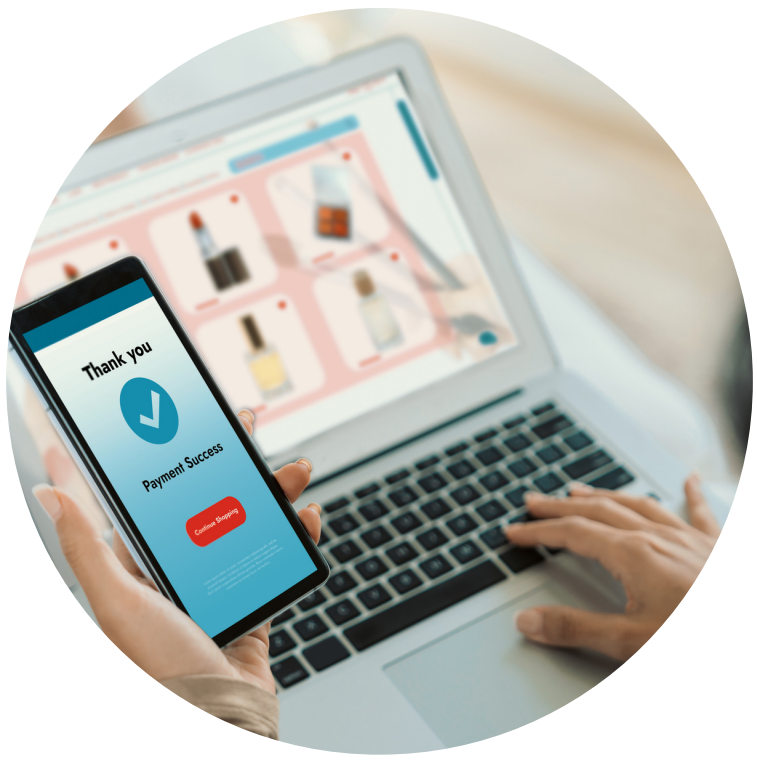
Imagine having separate teams responsible for managing data reads and writes. That's the essence of CQRS. This pattern separates read and write operations into different models and databases. This allows for concurrent read and write operations without conflicts, improving scalability and performance.

Benefits:

Improved Scalability: Reads and writes can be scaled independently according to their specific needs.

Increased Availability: Reads can continue even if the write model is unavailable.

Simplified Development: Separates read and write operations, making the code easier to understand and maintain.



Example:

Consider an online store with a CQRS architecture. The write model would be responsible for managing product inventory and order creation. The read model would be responsible for generating product listings and order status updates. This separation allows for handling high read traffic without impacting write availability.

Taken from: https://danielfoo.medium.com/5-cloud-native-architecture-patterns-for-2024-5bf6dc34e204