S Shashir

Summary

The Sidecar pattern in microservice architecture allows for the decoupling of core business logic from supportive functionalities by running them in separate, but closely integrated, containers within the same pod, facilitating shared resources and enhanced capabilities like logging, configuration, monitoring, and networking.

Abstract

In the realm of microservices, the Sidecar pattern emerges as a key architectural approach to manage common functionalities such as logging, configuration, monitoring, and networking services. This pattern advocates for the separation of these functionalities from the core business logic by encapsulating them in a Sidecar container, which operates in tandem with the primary application container within a Kubernetes pod. This design ensures that the core application remains focused on its primary function while leveraging the Sidecar for extended capabilities. The Sidecar pattern is particularly beneficial when services are implemented in different languages or technologies, require co-location, are owned by external teams, need independent updates, or when resource limits need to be controlled. Examples of its application include adding HTTPS support to legacy services, enabling dynamic configuration, and implementing log aggregation.

Opinions

- The Sidecar pattern is praised for its ability to segregate core logic from supporting functionalities, reducing the risk of one component's failure affecting the entire application.
- It is highlighted that the Sidecar container enhances the primary application by providing shared resources and functionalities, leading to more efficient use of those resources.
- The pattern is seen as advantageous for microservice architectures that need to share common components across services, such as logging and monitoring.

- it allows for independent updates and lifecycle management while maintaining co-location benefits.
- The opinion is expressed that the Sidecar pattern aligns with the principle of containers addressing a single concern and doing so effectively, which is a common best practice in the containerized world.

Microservice Architecture: Sidecar Pattern

Sidecar Pattern

In microservice architecture, it's very common to have multiple services/apps often require common functionalities like logging, configuration, monitoring & networking services. These functionalities can be implemented and run as a separate service within the same container or in a separate container.

Implementing Core logic and supporting functionality within the same application:

When they are implemented in the same application, they are tightly linked & run within the same process by making efficient use of the shared resources. In this case, these components are not well segregated and they are interdependent which may lead to failure in one component can in-turn impacts another component or the entire application.

Implementing Core logic and supporting functionality in a separate application:

When the application is segregated into services, each service can be developed with different languages and technologies best suited for the required functionality. In this case, each service has its own dependencies \libraries to access underlying platform and the shared resources with the primary application. It also adds latency to the application when we deploy two applications on different hosts and add complexity in terms of hosting, deployment and management.

Sidecar Pattern (or) Sidekick Pattern?

Sidecar concept in Kubernetes is getting popular and .Its a common principle in container world, that container should address a single concern & it should do it well. The Sidecar pattern achieves this principle by decoupling the core business logic from additional tasks that extends the original functionality.

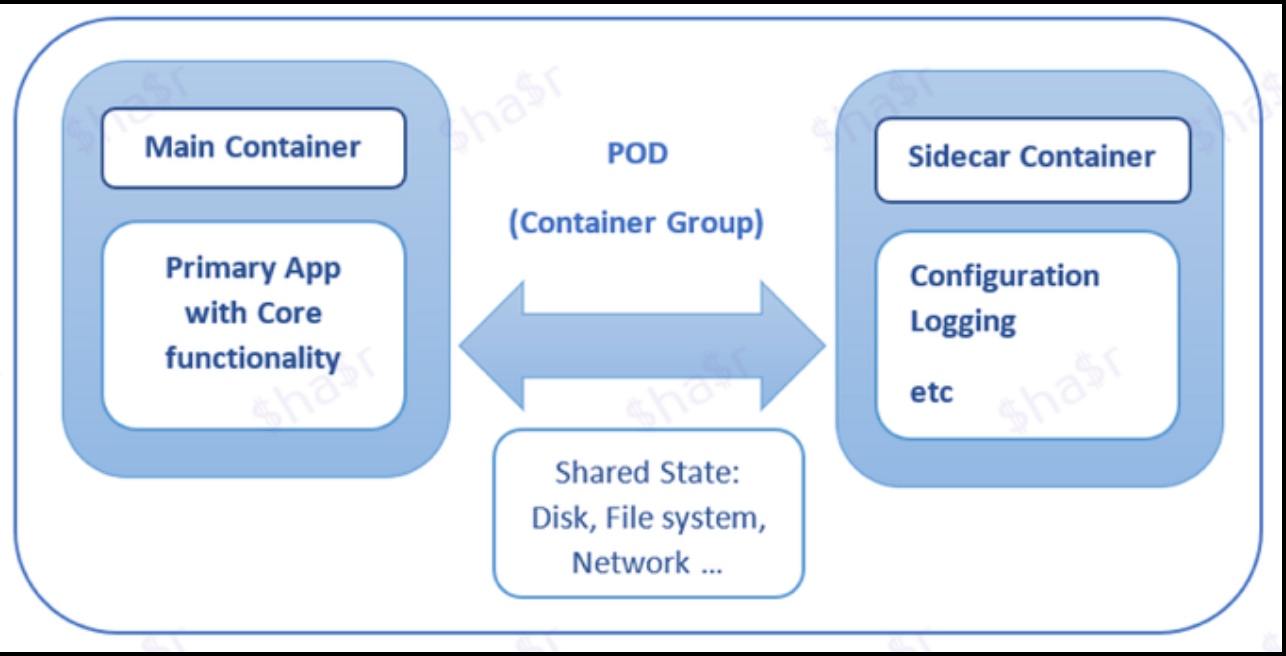
Sidecar pattern is a single node pattern made up of two containers.

The first is the application container which contains the core logic of the application (primary application). Without this container, application wouldn't exist.

In addition, there is a Sidecar container used to extend/enhance the functionalities of the primary application by running another container in parallel on the same container group (Pod). Since sidecar runs on the same Pod as the main application container it

shares the resources — filesystem, disk, network etc.,

It also allows the the deployment of components (implemented with different technologies) of the same application into a separate, isolated & encapsulated containers. It proves extremely useful when there is an advantage to share the common components across the microservice architecture (eg: logging, monitoring, configuration properties etc..)



Eg: Sidecar: Pod with 2 containers

What is a Pod?

Pod is a basic atomic unit for deployment in Kubernetes (K8S).

In K8S, a pod is a group of one or more containers with shared storage and network. Sidecar acts as a utility container in a pod and its loosely coupled to the main application container. Pod's can be considered as Consumer group (in Kafka terms) which runs

multiple containers.

When Sidecar pattern is useful ?

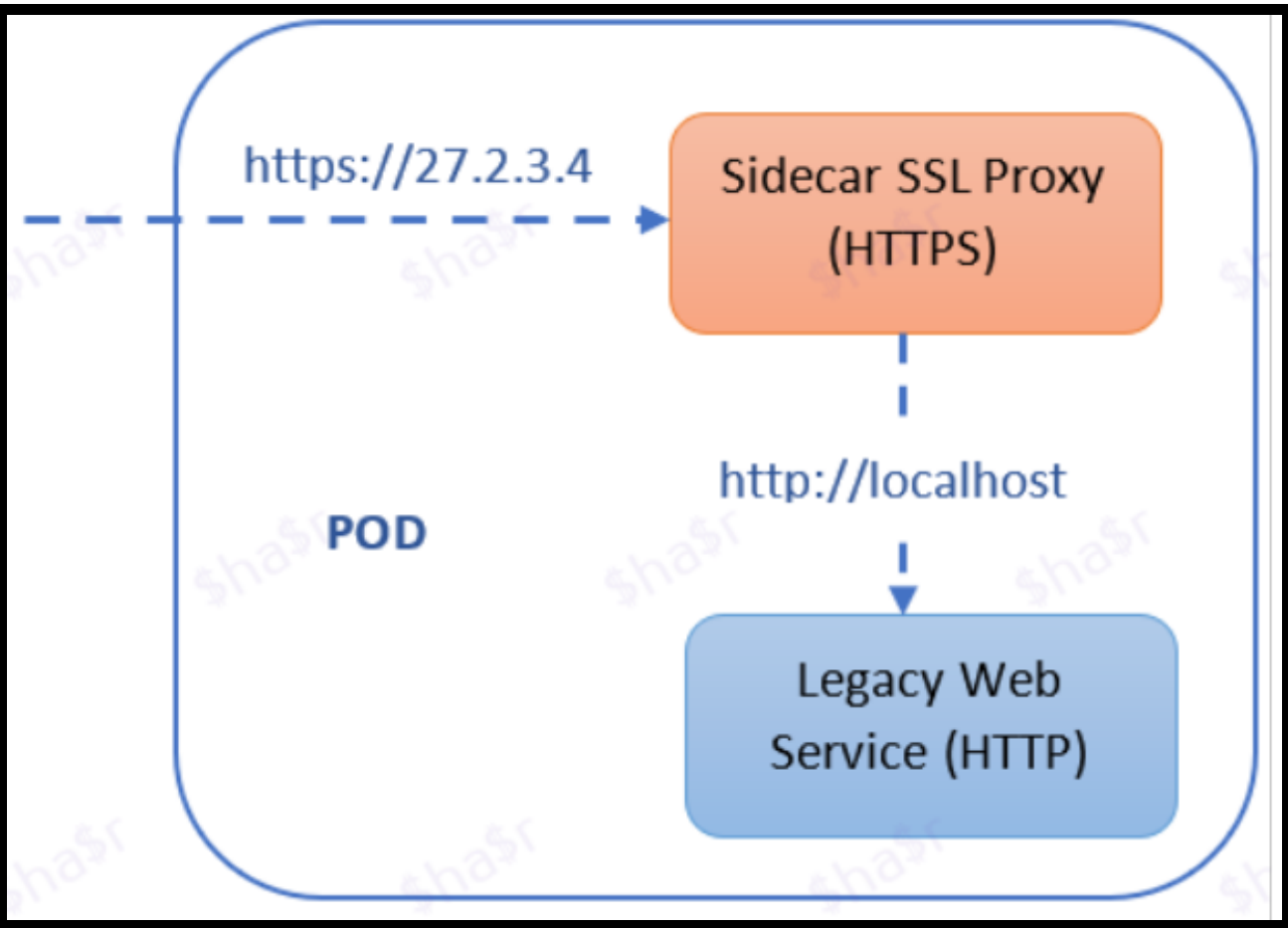
- When the services/components are implemented with multiple languages or technologies.
- A service/component must be co-located on the same container group (pod) or host where primary application is running.
- A service/component is owned by remote team or different organization.
- A service which can be independently updated without the dependency of the primary application but share the same lifecycle as primary application.
- If we need control over resource limits for a component or service.

Examples:

1. Adding HTTPS to a Legacy Service 2. Dynamic Configuration with Sidecars 3. Log Aggregator with Sidecar

Adding HTTPS to a Legacy Service

Consider a legacy web service which services requests over unencrypted HTTP. We have a requirement to enhance the same legacy system to service requests with HTTPS in future.

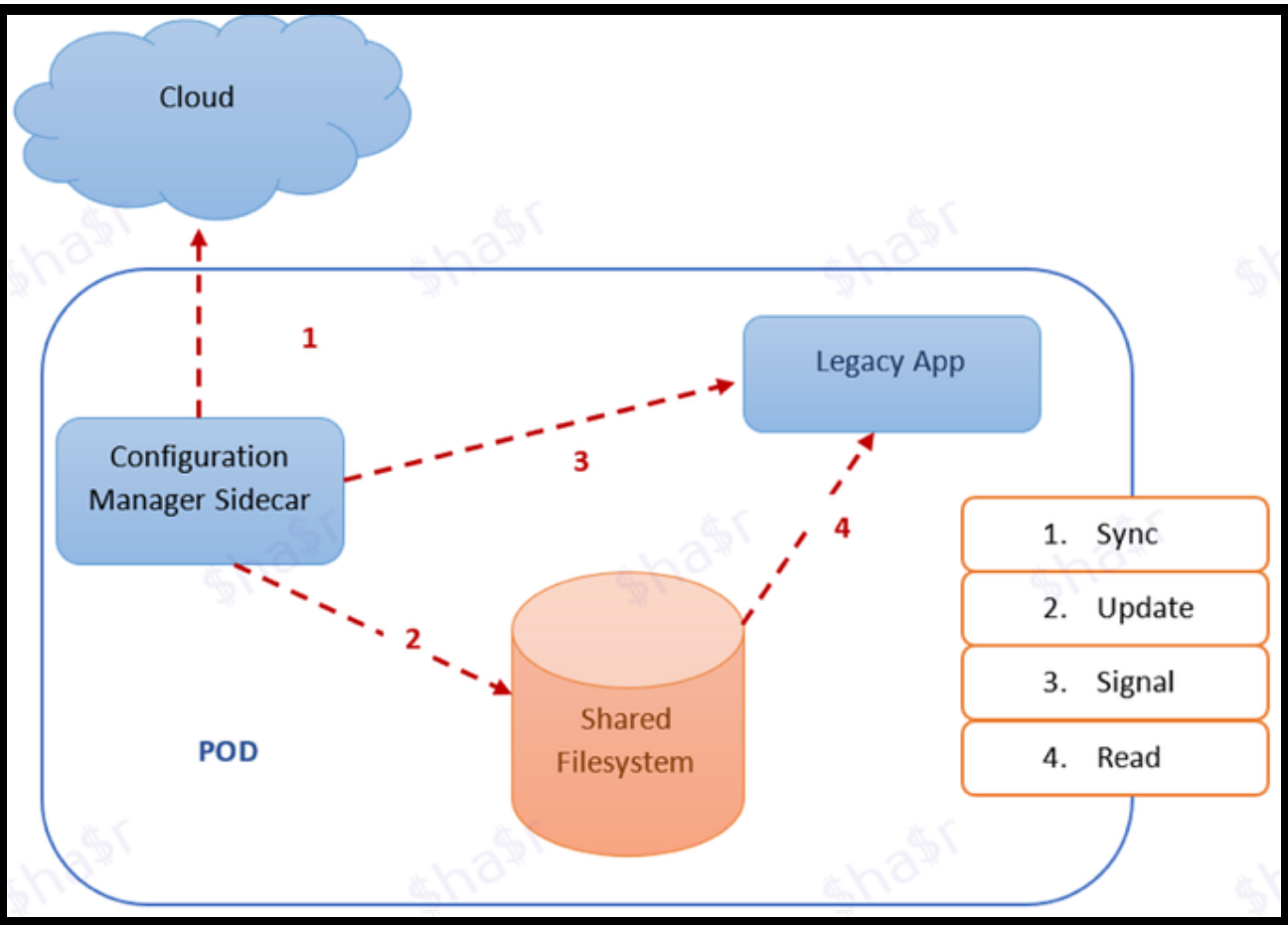


Example: Example: Adding HTTPS to a Legacy Service

The legacy app is configured to serve request exclusively on localhost, which means that only services that share the local network with the server able to access legacy application. In addition to the main container (legacy app) we can add Nginx Sidecar container which runs in the same network namespace as the main container so that it can access the service running on localhost.

At the same time Nginx terminate HTTPS traffic on the external IP address of the pod and delegate that traffic to the legacy application.

Dynamic Configuration with Sidecars



Example: Example: Dynamic Configuration with Sidecars

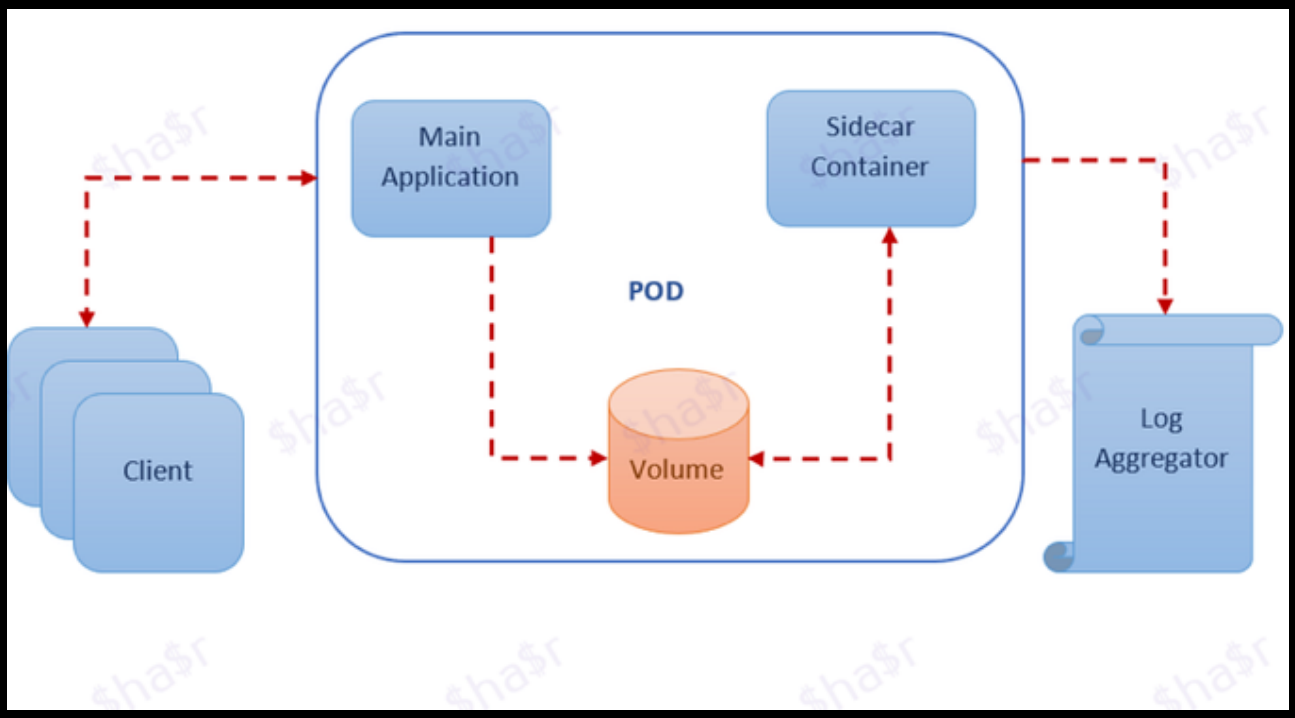
When the legacy app starts, it loads its configuration from the filesystem.

When configuration manager starts, it examines the differences between the configuration stored on the local file system and the configuration stored on the cloud. If there are differences, then the configuration manager downloads the new configuration to the local filesystem & notify legacy app to re-configure itself with the new configuration (Eg: can be EDD or Orchestration mechanism to pick new config changes)

Log Aggregator with Sidecar

Consider we have a web server which is generating access/error logs

which is not soo critical to be persisted on the volume beyond the specific time interval/memory space. However, access/error logs helps to debug the application for errors/bugs.



Example: Log Aggregator with Sidecar

As per separation of concerns principle, we can implement the Sidecar pattern by deploying a separate container to capture and transfer the access/error logs from the web server to log aggregator.

Web server performs its task well to serve client requests & Sidecar container handle access/error logs. Since containers are running on the same pod, we can use a shared volume to read/write logs.

Microservices | Microservices Pattern | Sidecar Pattern | Software Patterns | Design Patterns

Recommended from ReadMedium

Joud W. Awad

Microservices Pattern: Distributed Transactions (SAGA)

Explore the SAGA Pattern: Ensuring Data Integrity in Microservices. Dive into its benefits and applications

23 min read

Love Sharma

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

9 min read

Denat Hoxha

Sharing Data Between Microservices

Robust distributed systems embrace eventual

consistency to share data between their services.

7 min read

Ramesh Fadatare

5 Microservices Design Patterns You Must Know in 2025

Here are five important microservices design patterns you should know in 2025, explained in simple terms with examples. Microservices...

6 min read

Laksh Chauhan

Distributed Transaction Patterns in Event-Driven Microservices

Design patterns to maintain consistency for an event spanning across services in a microservice architecture.

4 min read

Java Interview

Spring Says Goodbye to @Autowired: Here's What to Use Instead

Yes, starting with Spring Boot 3 and Spring Framework 6, Spring has been encouraging constructor-based dependency injection over field...

3 min read