k8s_nemesis

Resources and microservices

Product description

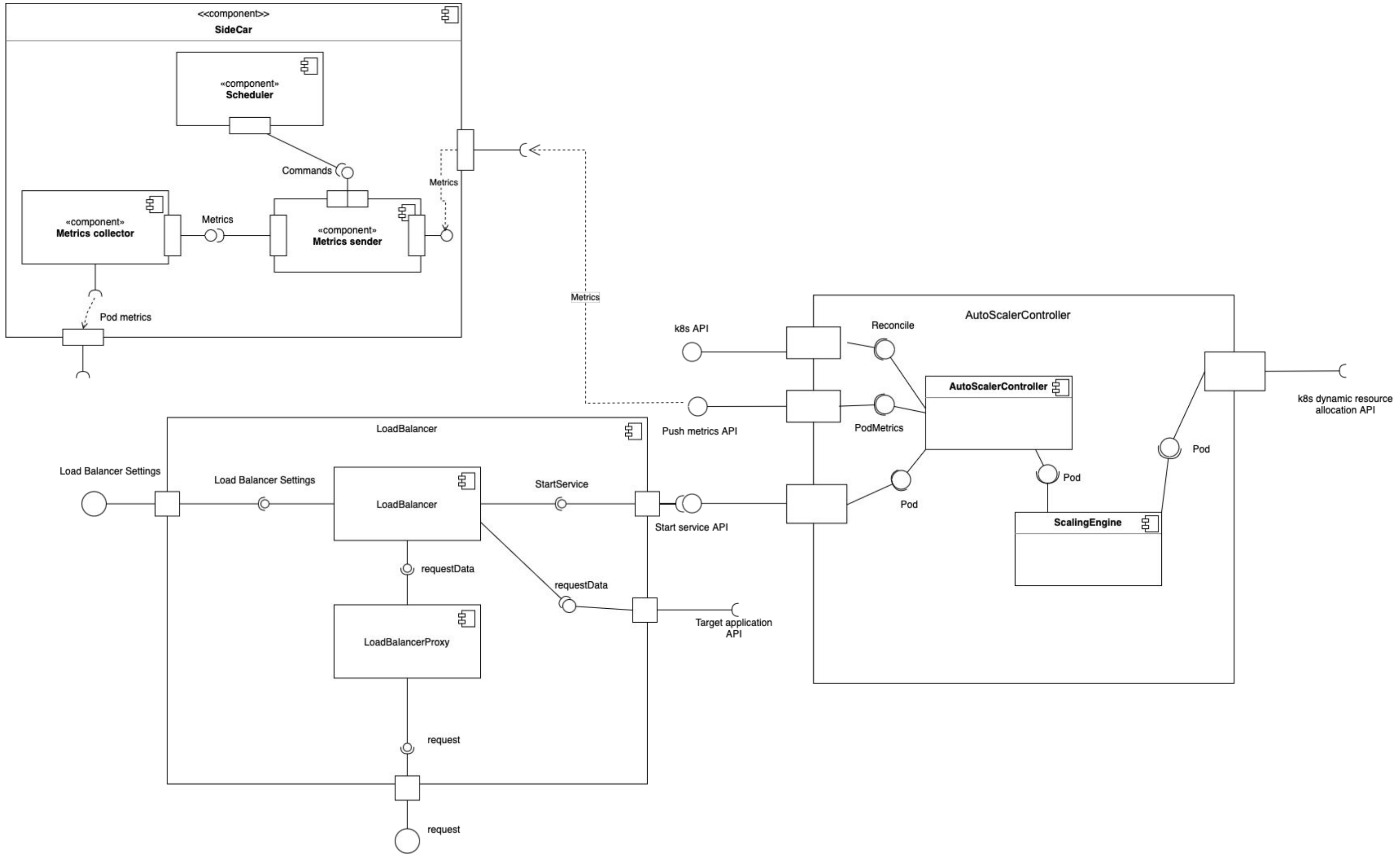
Product Description: A cloud-native service designed to enhance application scalability and performance in Kubernetes (K8s) environments, specifically targeting machine learning (ML) applications with unique scaling and resource requirements.

Team: Sergey Lokhmatikov, Roman Kuzmenko, Andrey Tamplon

Repo: https://github.com/Lokhmat/k8s_nemesis/tree/main

Report: https://github.com/Lokhmat/k8s_nemesis/blob/main/task_12_slides.pdf

System architecture



Separation of Concerns Principle

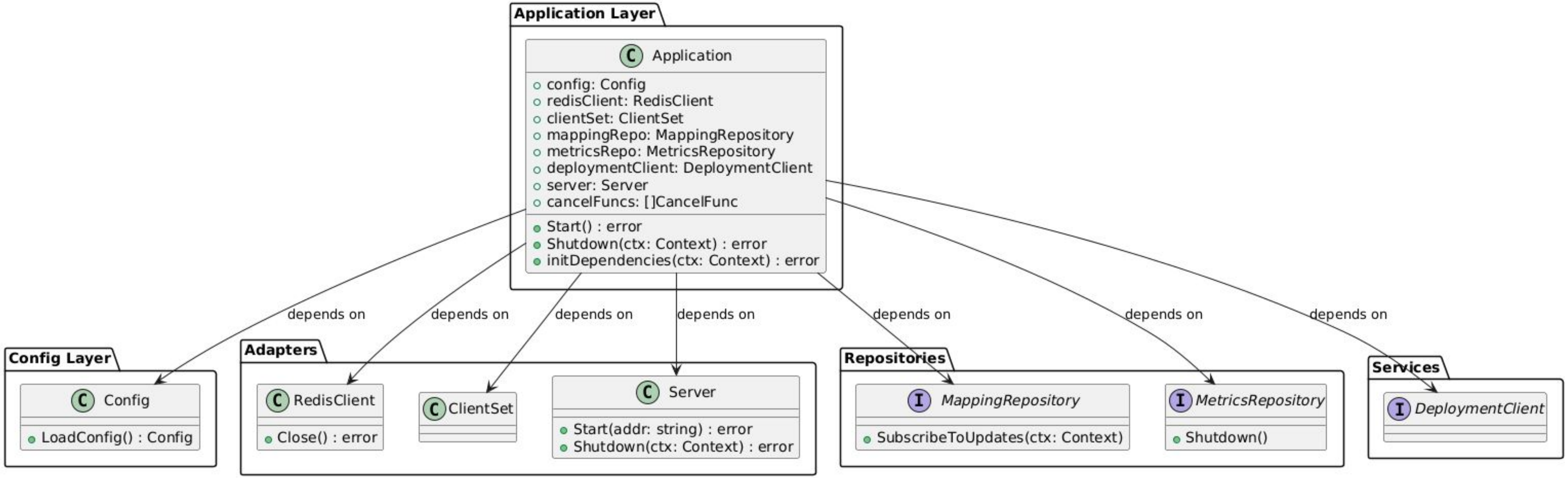
“Each service or component should perform its own clearly defined task. This increases the readability and testability of the system.”

Example: MappingRepository performs one clear task - receiving and subscribing to data updates. Tasks are isolated from business logic and are provided via SubscribeToUpdates.

Patterns and approaches used:

Repository Pattern: Separates data access logic from business logic.

Adapter Pattern: Wraps the data access layer to support different sources.

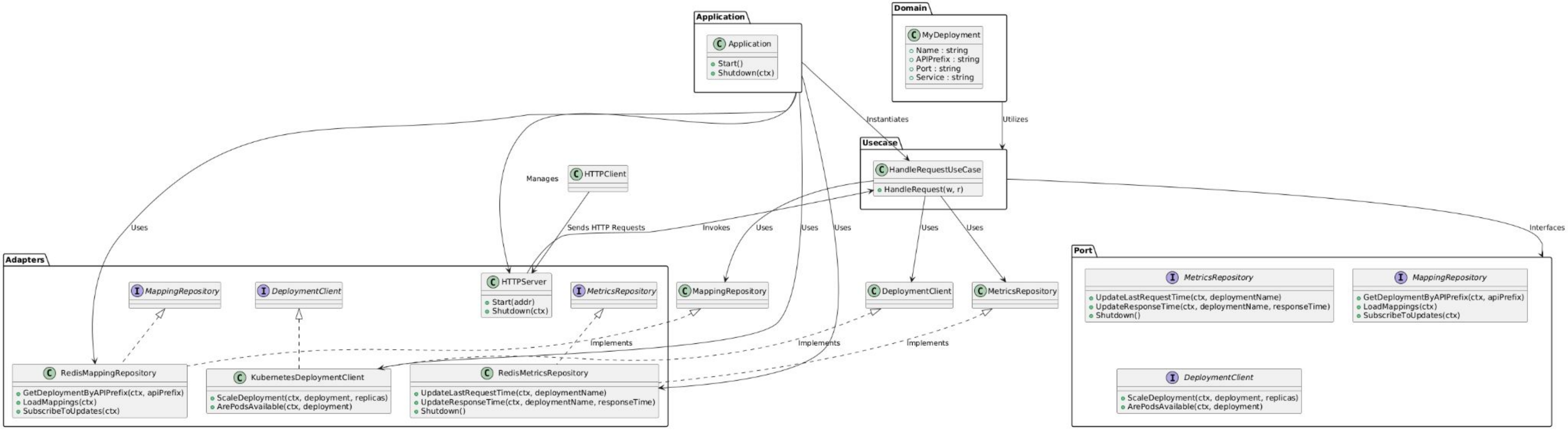

Application Layer
Application
+ config: Config
+ redisClient: RedisClient
+ clientSet: ClientSet
+ mappingRepo: MappingRepository
+ metricsRepo: MetricsRepository
+ deploymentClient: DeploymentClient
+ server: Server
+ cancelFuncs: []CancelFunc
+ Start() : error
+ Shutdown(ctx: Context) : error
+ initDependencies(ctx: Context) : error
depends on
depends on
depends on
depends on
depends on
depends on
depends on
Config Layer
Config
+ LoadConfig() : Config
Adapters
RedisClient
+ Close() : error
ClientSet
Server
+ Start(addr: string) : error
+ Shutdown(ctx: Context) : error
Repositories
MappingRepository
+ SubscribeToUpdates(ctx: Context)
MetricsRepository
+ Shutdown()
Services
DeploymentClient

Hexagonal architecture

Problem: The need to create a flexible and scalable load balancing application that could easily integrate with various external systems (Redis, Kubernetes) and provide a clean and supported code structure.

Solution: The use of a hexagonal architecture to separate business logic from external dependencies through the use of ports and adapters

Hexagonal architecture example in autoscaler



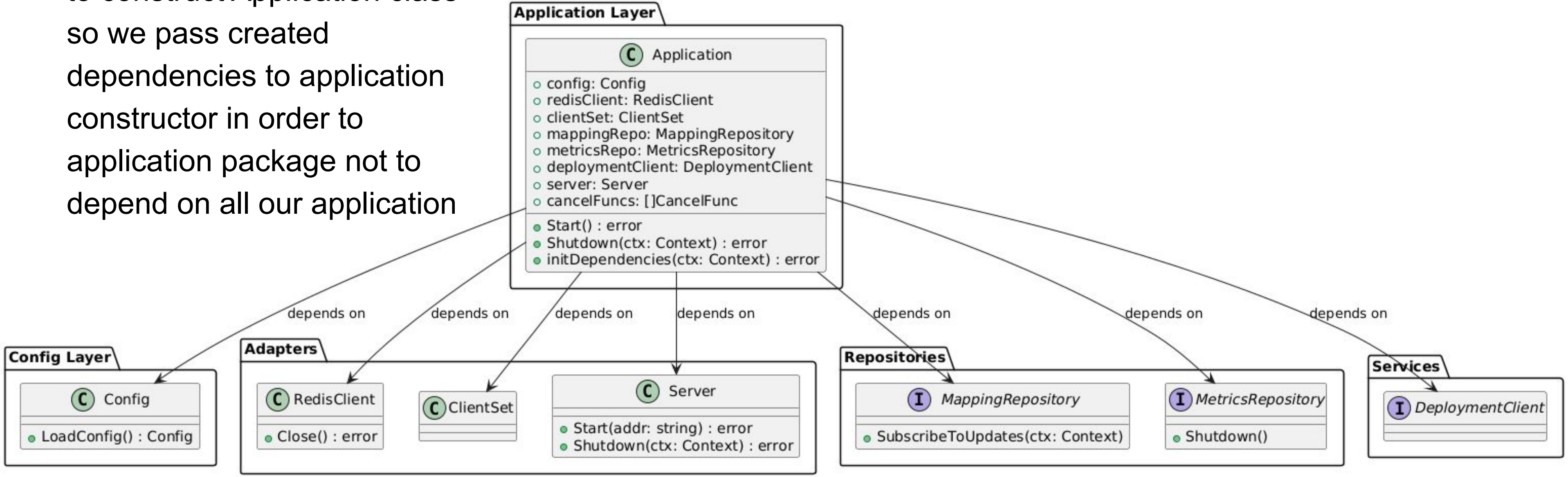
Modularity and low coupling

Problem: System has multiple different components, some of them are dependant on others. Tight coupling and straightforward dependencies impact flexibility and decrease convenience of working with code.

Solution: The use of modularity and low coupling principle is used to enhance process of integration of different components between each other. It helps us to increase flexibility and maintainability of the code.

Example of low coupling: Dependency injection

Dependency injection is used to construct Application class so we pass created dependencies to application constructor in order to application package not to depend on all our application


Application Layer
Application
config: Config
redisClient: RedisClient
clientSet: ClientSet
mappingRepo: MappingRepository
metricsRepo: MetricsRepository
deploymentClient: DeploymentClient
server: Server
cancelFuncs: []CancelFunc
Start() : error
Shutdown(ctx: Context) : error
initDependencies(ctx: Context) : error
depends on
Config Layer
Config
LoadConfig() : Config
Adapters
RedisClient
Close() : error
ClientSet
Server
Start(addr: string) : error
Shutdown(ctx: Context) : error
Repositories
MappingRepository
SubscribeToUpdates(ctx: Context)
MetricsRepository
Shutdown()
Services
DeploymentClient