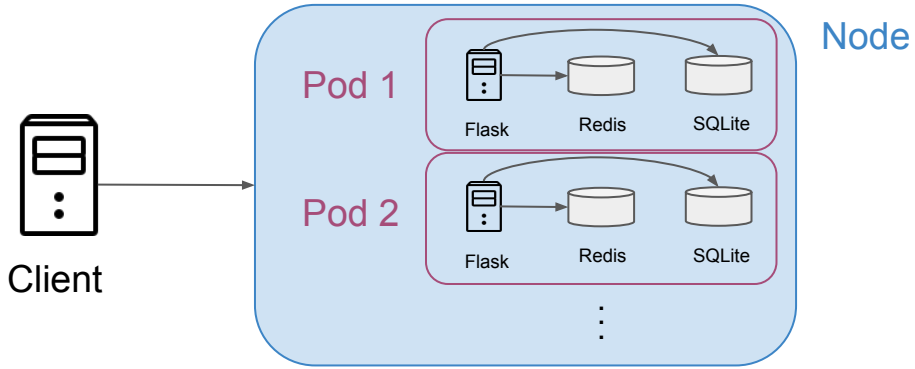
Scalable Species Info Service

Project Outline

Introduction and Architecture

- Prototype for a scalable information service for read-only species data
- Containerized with Docker and deployed locally with Kubernetes (Minikube)
- NodePort service routes requests to one or more species-service pods
  - Each pod contains a Flask app container, a Redis cache container and its own persistent volume for a SQLite database
  - Identical DB seed data across all pods
- Supports horizontal scaling (number of pods) and vertical scaling (CPU/RAM per pod)

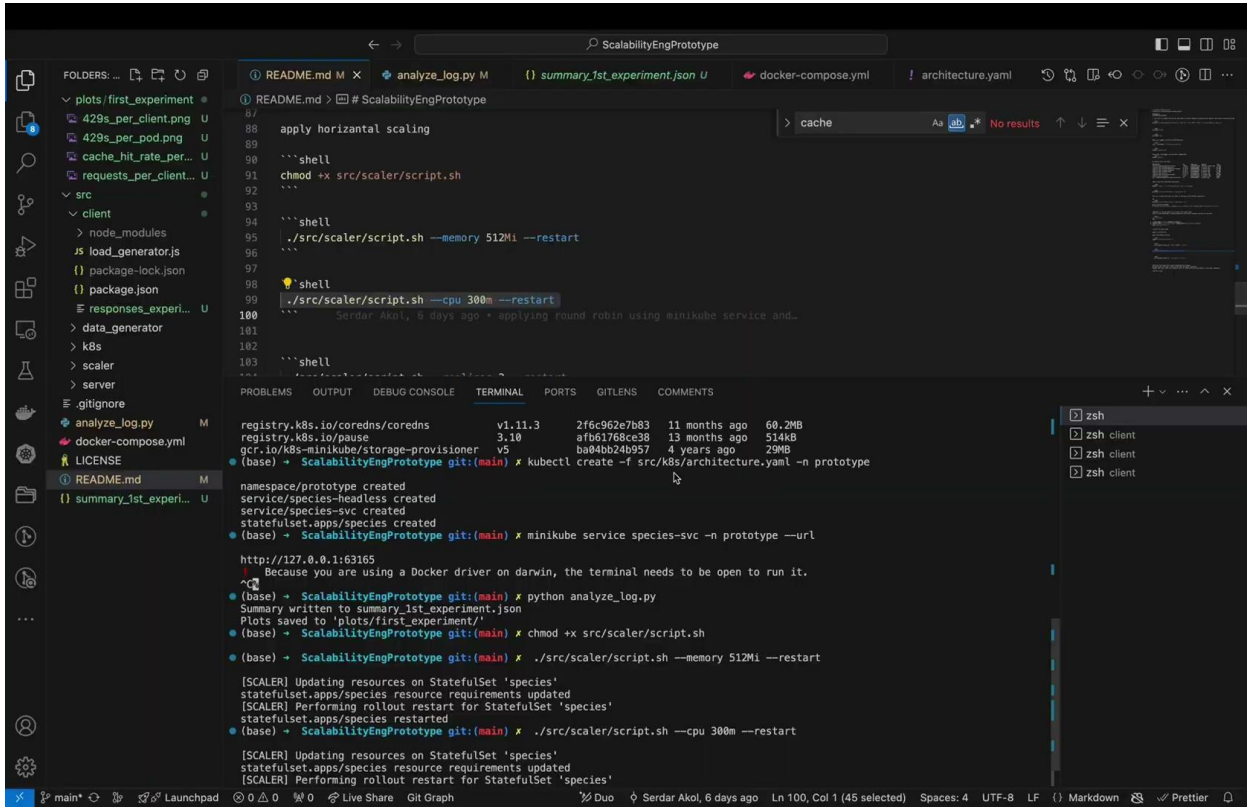


Implemented Features

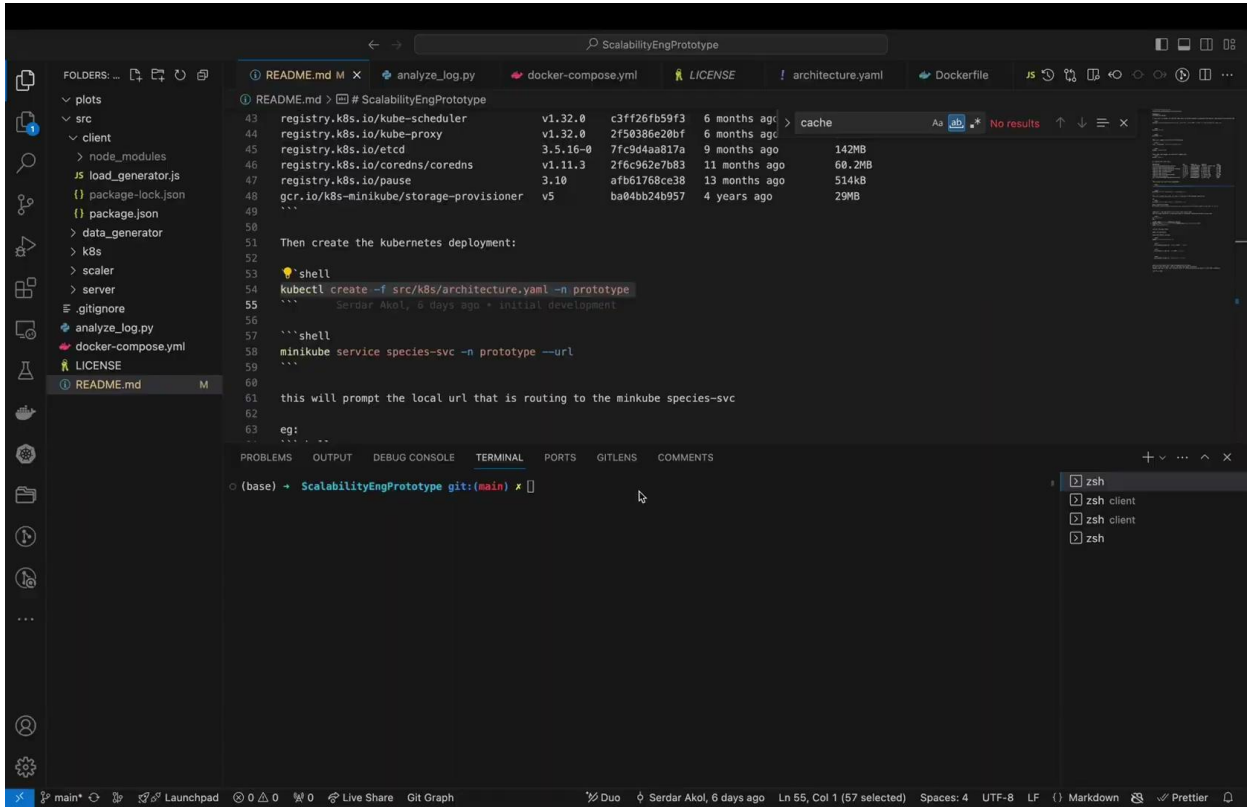
- Flask API is stateless per request
- **Persistent state** (species data) is identical and preloaded for each pod
  → No coordination is needed for managing the state
- The Flask app implements a simple **Load Shedding** mechanism to limit the number of requests per minute
- A **Redis Cache** reduces database reads and improves throughput
- **Scaling Script** (.sh) to adjust the Kubernetes resources while the application is running

Repository:
⇒ https://github.com/serdarakol/ScalabilityEngPrototype/tree/main

Demo Functionality



Demo Scalablity



Results and Limitations

- We use a StatefulSet with per instance data volumes to ensure each replica can operate independently, for dynamic data we might introduce a scalable shared database
- Caching provides crucial performance gains, a shared cache layer could optimize cache usage even more
- Our simple load shedding mechanism protects the server from overload, for more sophisticated rate limiting we could use adaptive limits
- Automatically scaling based on metrics like CPU usage would help ensure sufficient performance under varying load