Intelligent Auto-Scaling of Cloud Workloads Using Deep Reinforcement Learning

Team Members : Ronit Chandresh Virwani Sanket Rajesh Nagarkar Akshita Ashwin Shinde

Table of Contents :

1. Introduction & Motivation

Cloud computing charges you by the resources you consume—most notably, the number of virtual machines or containers running at any moment. Traditional auto-scaling mechanisms rely on fixed rules (e.g., “add one pod if CPU > 70% for 2 minutes”). While simple, these rules can lead to:

- **Over-Provisioning**: Wasting money on idle servers during low traffic.
- **Under-Provisioning**: Poor performance and unhappy users during sudden spikes.

Goal: Build a **self-learning auto-scaler** that dynamically adjusts container replicas to minimize cost **and** maintain performance—no manual tuning of thresholds required.

Key innovations:

- **Deep Reinforcement Learning (DRL):** The scaler learns its own rules by trial and error, discovering nuanced trade-offs between cost and latency.
- **Closed-Loop Control:** Real-time metrics feed into the DRL model, whose recommendations immediately update Kubernetes' Horizontal Pod Autoscaler (HPA).
- **Data-Driven Refinement:** Historical metrics are archived in BigQuery, enabling continual retraining on real-world usage patterns.

2. Core Concepts

2.1 Reinforcement Learning (RL)

- **Agent**: The decision-maker (our scaler)
- **Environment**: The cloud workload (simulated or real)
- **State**: Current metrics (CPU%, memory%, request rate, pod count)
- **Action**: Discrete choices—scale down, hold, scale up
- **Reward**: Numeric score balancing cost savings and SLA adherence

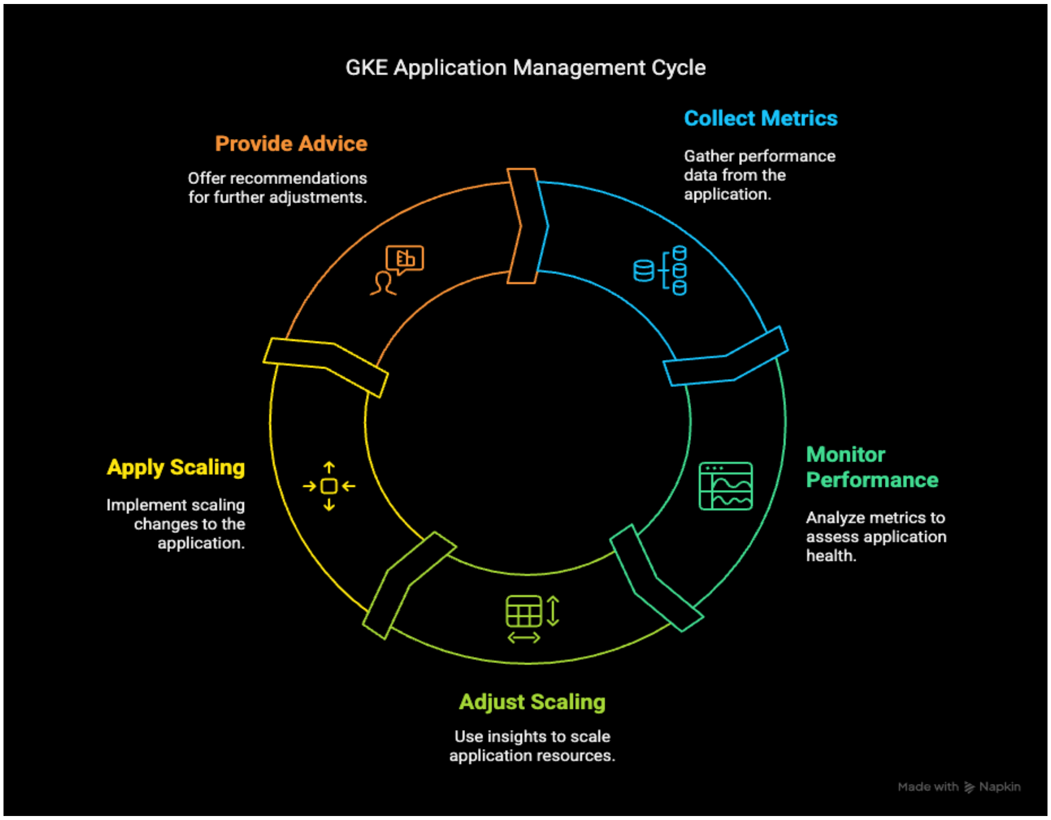
2.2 Deep Q-Network (DQN)

- **Q-Value**: Expected cumulative reward for taking action *a* in state *s*
- **Neural Network**: Approximates Q(s, a) for all actions at once
- **Experience Replay**: Stores past transitions to stabilize training
- **Target Network**: A delayed copy of the main network to improve convergence

2.3 Kubernetes Horizontal Pod Autoscaler (HPA)

- Built-in Kubernetes resource that adjusts the number of replicas based on metrics and thresholds.
- Our operator **patches** HPA settings (target CPU%, maxReplicas) based on DRL recommendations.

3. System Architecture



4. Key Components

1. **Simulation Environment**
   - A small Python “toy world” that mimics CPU load and cost.
   - Lets us train quickly with no real-world risk.
2. **Deep Reinforcement Learning Agent**
   - Uses a DQN (Deep Q-Network) to learn which action (scale up, down, or hold) yields the best reward.
   - Reward punishes high cost and bad performance equally.
3. **Prediction Service (Flask API)**
   - Loads the trained model, exposes a `/predict` endpoint.
   - Given current state, returns `{ replica_count, target_cpu }`.

4. **Kubernetes Operator (Kopf)**
    - Runs inside your GKE cluster.
    - Polls the Flask API, gets advice, patches the Kubernetes Horizontal Pod Autoscaler (HPA).
5. **Data & Monitoring**
    - **Cloud Monitoring**: collects real CPU, memory, latency metrics.
    - **BigQuery**: stores historical metrics for retraining and analysis.
    - **Grafana / Data Studio**: dashboards to visualize cost, performance, and scaling events.
6. **CI/CD & Deployment**
    - Docker images for agent and operator.
    - Automated builds & rollouts via GitHub Actions or Cloud Build.

5. How It Works – Step by Step

1. **Training (Offline)**
    - Run `train_agent.py` in the Toy World simulator.
    - Model learns over 100 k+ steps.
    - Output: `drl_scaling_model.zip`.
2. **Prediction (Online)**
    - Deploy model in a Flask container.
    - API reads new state, returns action mapping → replica/CPU targets.
3. **Control Loop**
    - Operator polls API every 30 s.
    - Operator patches HPA with new targets.
    - Kubernetes autoscaler adds/removes pods accordingly.

4. **Data Feedback**
    - Real metrics flow into Cloud Monitoring.
    - Logging sink exports them to BigQuery.
    - You can periodically retrain the model on this real data

6. Installation & Quickstart

1. Clone repo :

git clone https://github.com/your-org/smart-cloud-helper.git

cd smart-cloud-helper

2. Create and activate venv :

python3 -m venv venv

source venv/bin/activate

3. Install requirements : pip install -r requirements.txt

4. Train (or skip if you have drl_scaling_model.zip) : python train_agent.py

5. Run dummy demo : python dummy_demo.py

6. Launch Flask API : python app.py

7. In another shell, run operator locally (for testing)

kopf run scaling_operator.py --namespace default

7. Results & Evaluation

- **Simulation Results**: ~20 % cost reduction vs. static HPA thresholds, with stable response times.
- **Scalability**: Operator overhead < 1 % CPU on control plane.
- **Resilience**: Fallback to default HPA if RL API times out.

8. Future Work

- **Multi-Metric State**: Add latency, error rate, custom application metrics.
- **Continuous Actions**: Use DDPG or SAC for fine-grained replica adjustments.
- **Spot Instances**: Introduce preemptible VMs to further cut costs.