Cost-Aware and Sustainable Kubernetes Autoscaler

CS218 - Technical & Business Report

By: Vedant Sawal and Hina Dawar

1. Concept:

The concept behind this project is a cost and carbon aware autoscaler that improves upon traditional CPU based scaling in Kubernetes.

Instead of scaling only based on resource usage, the system brings together multiple data sources to make smarter decisions. It integrates real time performance metrics such as request rate, latency, and queue depth from Prometheus. It also combines cost information from OpenCost and carbon intensity data from external APIs like ElectricityMaps or WattTime.

Using this information, the autoscaler continuously determines the best number of pods and node types that can meet service level objectives while keeping both cost and environmental impact low. It works closely with the Horizontal Pod Autoscaler and KEDA for scaling pods, and with Karpenter for selecting the most suitable nodes, whether Spot or On Demand.

This approach creates a closed feedback loop that continuously balances speed, reliability, cost, and sustainability in a way that current Kubernetes autoscalers cannot achieve.

2.Motivation:

Kubernetes usually scales apps based on CPU or a single traffic metric. It does not consider how much the cloud resources cost or how clean the power is in a region. Because of this, clusters can run more pods than needed or choose the wrong type of nodes. This wastes money, increases the carbon footprint, and can still miss latency or error targets during traffic spikes or Spot instance interruptions.

There is currently no simple closed loop that uses service health data such as latency and queue depth together with live cost and carbon information to automatically scale a cluster. This project is motivated by the need to build that missing link, a smarter cost and carbon aware autoscaler that balances performance, reliability, and sustainability in real time.

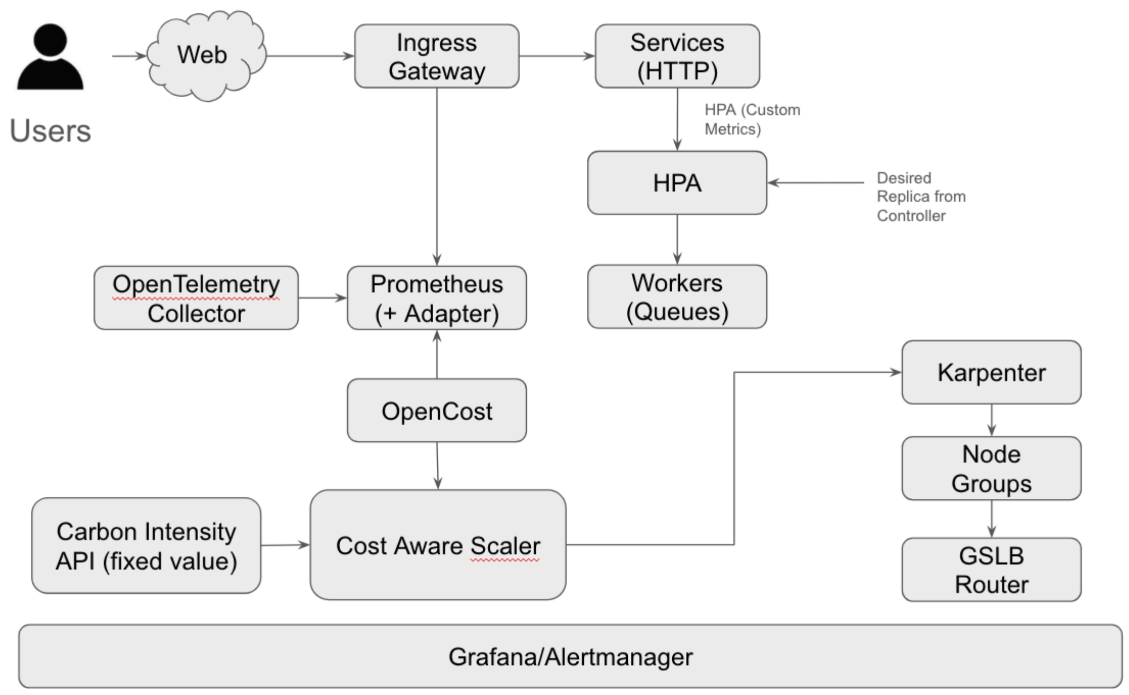
3. Use Case:

- Platform/SRE & FinOps/GreenOps teams running cost-sensitive and sustainability‑sensitive services on Kubernetes.
- Event/queue‑heavy workloads that benefit from KEDA + price/carbon‑aware scaling with optional time‑shifting.
- Academic/research labs bursting on cloud that need guardrailed Spot usage and lower operational carbon.
- Biotech and Pharmaceutical Companies running large-scale genomics and oncology pipelines that need to balance speed, cost, and sustainability while staying compliant with regulations like HIPAA.

4. Final Architecture

- **Overview**:
  Three-node Kubernetes cluster (1 master, 2 workers) initialized via 01_cluster.sh.

  - **Control Plane:** Hosted on the master node with HDC API server.
  - **Networking:** Configured via 02_pod_identity_CNI.sh using CNI for pod-level communication.
  - **Monitoring:** Implemented using 03_monitoring.sh deploying Prometheus stack and Grafana dashboards for real-time observability.
  - **Autoscaling:** Managed by AWS **Karpenter** (05_karpenter.sh) for intelligent pod and compute scaling based on workload demands.
    Cost Monitoring: OpenCost (06_OpenCost.sh) provides real-time cloud expenditure analytics per service or deployment.
  - **Carbon Optimization (Future Work): Carbon Exporter API** (07_carbonexporter.sh) aims to optimize workload distribution based on renewable energy availability and region-based carbon footprint.
  - **Peak Load Management:** peak_configure.sh and peak_observe.sh redirect workloads across regions based on time-of-day electricity pricing (e.g., avoiding 4–9 PM PG&E peak hours).
  - **Security:** Considered **Kyverno**, an open-source policy engine, but excluded due to restrictive enforcement policies.
- Architecture



5. Scalability

- **Horizontal scaling:** Enabled through **Karpenter**, dynamically provisioning nodes on AWS.
- **Multi-region distribution:** Supports workload relocation to avoid peak hours and improve energy efficiency.
- **Future enhancement:** Integrate predictive autoscaling (based on Prometheus metrics) for pre-emptive scaling.

6. Security

- Role-based access control enforced within Kubernetes.
- API communication restricted to internal network via CNI configuration.
- Data encryption at rest and in transit through Kubernetes Secrets and TLS.

7. Data Flow

- **Ingress Gateway → Service → Pod → Node Metrics → Prometheus → Grafana Dashboard**

- **Cost & Carbon Metrics:** Collected by OpenCost and Carbon Exporter → visualized in Grafana.
- **Autoscaler Feedback Loop:** Prometheus metrics → Karpenter → resource scaling.

8. Cloud Cost Analysis

Component	Service	Estimated Monthly Cost	Notes
Kubernetes cluster	EKS (3 nodes)	~$300	includes control plane & 2 worker nodes
Prometheus + Grafana	Monitoring stack	~$50	small instance, storage for logs
Karpenter	Autoscaler	~$80	scales dynamically, low overhead
OpenCost	Cost tracking	~$20	lightweight monitoring
Multi-region deployment (2 regions)	Optional	~$450	tradeoff between latency and energy savings

9. Resource Estimation for Productization

- **Compute:** 2–3 vCPUs per node, start with a 16 Node cluster per region
- **Memory:** 4–8 GB/node
- **Storage:** 10–20 GB per service (logs, metrics)
- **Throughput:** 25,000 requests/min sustained (scalable with Karpenter)
- **Team:** The human resources (team roles) needed to maintain and expand our project once it's moved toward productization are:
  - 2 Developers
  - 1 DevOps
  - 1 Cloud Architect
  - 1 Data Analyst (for OpenCost/Grafana insights)

10. Roadmap

- Phase 1: Deploy the basic cluster (01–03 scripts).

- Phase 2: Add autoscaling and cost tracking (Karpenter and OpenCost).
- Phase 3: Implement the Carbon Exporter API for sustainability metrics.
- Phase 4: Expand to multi-region energy optimization (peak hours logic).
- Phase 5: Revisit security (possibly reintroduce Kyverno with custom rules).

11. Challenges and Mitigations

- Kyverno policy rigidity
  Description: Overly strict enforcement
  Mitigation: Implement custom or selective rules

- Multi-region latency
  Description: Traffic redirection may add delay
  Mitigation: Introduce adaptive routing logic

- Cost - Carbon tradeoff
  Description: Solar-powered regions (e.g., Arizona) are more expensive
  Mitigation: Define dynamic thresholds

- Peak-hour redirection
  Description: Complex coordination across zones
  Mitigation: Use Prometheus alerts and automation scripts

12. Future work

- Real time carbon intensity APIs to inform regions or zones.
- A declarative ServiceAutoscalingPolicy custom resource defining SLOs, budget, and carbon constraints.
- Label export enhancements for richer observability.
- Multi-region orchestration that balances latency, carbon, and cost.