Intermediate Project Report: Experience Scaling

Author(s):

Sid Bansal
bansalsi@usc.edu

Ethan Bootehsaz
bootehsa@usc.edu

Anirudh Rajagopalan
ar92426@usc.edu

Objective

Our goal is to design and implement a proactive autoscaling system for Kubernetes (K8s) clusters that leverages historical traffic patterns to preemptively scale resources ahead of recurring traffic spikes. By replacing purely reactive metrics (e.g., CPU usage) with predictive scaling based on historical data, we aim to reduce latency caused by pod and node spin-up delays during sudden workload surges. Additionally, we will evaluate the efficiency of our system by measuring overprovisioning, ensuring minimal compute resource wastage while maintaining responsiveness.

Goals

Kubernetes (K8s) is a widely adopted container orchestration platform that dynamically manages workloads across distributed environments. While its default Horizontal Pod Autoscaler (HPA) scales applications reactively (e.g., based on CPU utilization), this approach introduces latency during the critical window between a traffic spike and resource provisioning. Though reactive scaling avoids idle resource costs, systems with predictable, recurring traffic patterns (e.g., daily/weekly cycles) can benefit from preemptive scaling.

Our system addresses this gap by:

1. Proactive Scaling: Using historical network traffic patterns to horizontally scale pods before anticipated surges, reducing end-user latency.
2. Efficiency: Balancing resource utilization to minimize overprovisioning (idle pods/nodes) while ensuring readiness for traffic spikes.
3. Empirical Validation: Quantifying success through two metrics:
    a. Latency reduction during scaling events (vs. vanilla HPA).
    b. Overprovisioning rate (percentage of unused pods).

By combining predictive scaling with Kubernetes' reactive HPA, we aim to deliver a hybrid solution that optimizes both performance and cost for workloads with temporally predictable demand.

Tasks Completed

Code: https://github.com/CSCI555-Spring25/Scaler

1. Environment Setup

We host our experiments on CloudLab. We create a 3-node Kubernetes cluster with 1 control plane node and 2 worker nodes, running Ubuntu 22.04.2 LTS. We installed Docker and Docker local container registry to build and pull containers within the K8s cluster. Each Docker container executes a simple web server. At minimum, our Kubernetes creates 1 replica for this web service image, with an Horizontal Pod autoscaler that can increase the replicas upto 10, with CPU usage threshold of 50% triggering scale-up. We created an API service coupled with Ingress and egress specifications, exposing the service to external traffic, allowing us with the

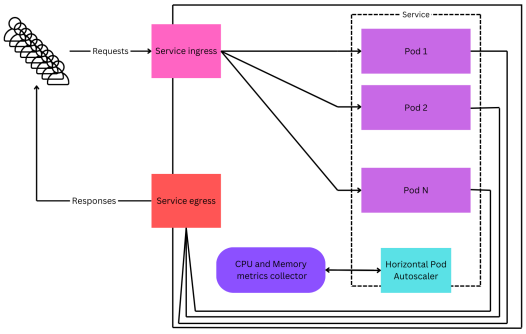
ability to simulate requests and collect responses while tracking metrics within the cluster using Prometheus.

2. **<u>Custom Resource - Experience Scaler</u>:**

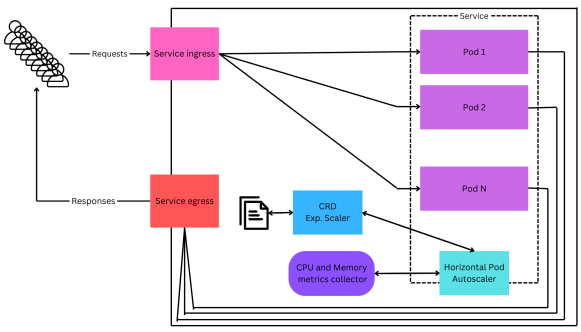
The system enhances Kubernetes' default Horizontal Pod Autoscaler (HPA), which operates reactively based on real-time metrics, by introducing a proactive scaling mechanism. This is achieved through a Custom Resource Definition (CRD) that leverages historical cluster "Experience" (i.e., pod-count patterns over time) to predict scaling needs. The goal is to optimize resource efficiency while retaining adaptability to evolving traffic trends.

- **Key Design Decisions:**
  - <u>Focus on Pod-Count History Over Traffic Metrics</u> :
    - Traffic volume alone does not directly correlate with resource consumption. Different images/applications have workloads which utilize CPU/memory at varying rates, even under identical traffic. The CRD uses pod-count history as an abstraction, decoupling scaling decisions from compute/memory thresholds. This simplifies homogenized scaling across diverse deployments.
  - <u>Two-Pronged Scaling Approach :</u>
    - *Proactive Baseline*: CRD recommends a minimum pod count (*minPods*) based on historical trends, ensuring proactive readiness. The minimum recommendation is defaulted to 1, to ensure no compute wastage during minimal traffic.
    - *Reactive Adjustment*: HPA retains its reactive scaling capability to adjust pod count upward if real-time CPU/memory usage exceeds developer set thresholds. The total pods remain within the developer-defined *maxPods*, ensuring safety guarantees while allowing dynamic adjustments to minPods.
    - *Experience Refinement*: After each scaling action, historical data is updated to reflect new observations, enabling continuous learning. All data is stored within a json inside the K8s cluster's persistent volume, in format timestamp:podCount as key value pair to enable quick lookups and updates.
  - Real world applications might require a more refined granularity, for example including day of the week or time of the year, and will also require more than one value mapped to timestamp when there are multiple services/images present. We have overlooked these necessities for our prototype due to time, data and resource constraints.

2.1. Architecture Diagrams:



An oversimplified HPA only cluster setup



Experience scaling backed HPA cluster setup

2.2. How Experience Scaling Works:

2.2.1. Cold start (no experience / no json with pod count history):

The scaling occurs as per vanilla HPA scaling based on CPU and/or memory usage, with developer defined min and max pod count, this data is added to a json file present within the persistent volume.

2.2.2 With Gained Experience:

There are two major developer defined values, ***predictionWindowMinutes, currentWeight*** and ***historicalWeight***. The ***predictionWindowMinutes*** i.e. look-ahead time (default 5 minutes) dictates how long before a particular timestamp should the scalar be looking-ahead to and scaling up/down based on computed recommendation. The developer can also modify the ***historicalWeight*** and ***currentWeight*** i.e. weightage of old experience v current data (default 0.7 and 0.3 respectively), adjusting how much the existing experience is affected by a different traffic pattern that has currently emerged. This dictates how easily the gathered experience is modified over time when the pattern changes.

2.2.3. Scaling

When looking ahead to scaling up/down the replicas, the Scaler follows:

*Required_pods = int((current_pods / historical_pods_now) * historical_pods_ahead)*

We get a ratio of current traffic (in terms of number of pods) against historical pod counts at current timestamp. The lookahead timestamp's pod count is then adjusted based on this ratio and is set to the minimum required pods. Then, the minimum of maxPods and calculated required pods is then set as the *minPods* in the HPA. The HPA inturn triggers scale up from this minPods value, basing on current compute and/or memory usage.

For instance, if current pods are 5, historical pods at current timestamp (now) is 4 (i.e. traffic is more than expected), and historical pods ahead is 6 for the specified look ahead, then Required_pods = (5/4)*6 = 7.5, rounded down to 7. And hence two more pods are spun up by the HPA. If the *Required_pods* > *maxPods*, then we set recommendation = *maxPods*.

2.2.4. Experience Gain

The value present in the json is adjusted based weighted average of the historical pod counts vs current pods calculated:

*entry["podCount"] = int(historical_weight * historical_count + current_weight * current_pods)*

From the above example, after seeing more traffic at current timestamp, the "Experience" is updated as entry["now"] = 0.7 * 4 + 0.3*5 = 4.3, rounded down to 4.

3. Prototype Testing

For prototyping and testing the Scaler, we have generated an "Experience" with timestamps across a day against the pod count value of just one service with one image present in the cluster every 5 minutes. This data is compiled in a json file present in the persistent volume of the cluster. The json contains a key-value pair of timestamp to pod count. We applied this to the cluster in CloudLabs and monitored the logs for how the scaling of the service is ongoing over time and how the json is getting updated. After debugging and fixing

issues with K8s integration and logic errors during edge cases (like divide by 0 during no traffic), the results show that the Scaler is working as expected.

4. **Documentation and Open Source access**

The project is present as a public repo on Github and we are also actively updating the readme to allow for others to fork and/or generate PRs in the future.

Tasks Pending

1. Writing probabilistic traffic simulation script (in-progress, being tested)
2. Generating Traffic and collecting metrics against a simple HPA only cluster
3. Re-simulating a similar traffic and collecting metrics against the Experience Scaler

With the Scaler CRD completed and tested to be working as expected, we are on track for completion of the project within the timeline, with only evaluation pending.