International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Efficient Resource Management in FaaS: A Comparative Study of Allocation and Scheduling Techniques

Rekha Singh[1*], Sanjay Tanwani[2]

Submitted:01/02/2024 **Revised:**12/03/2024 **Accepted:**20/03/2024

Abstract: Function-as-a-Service (FaaS) platforms enable developers to deploy event-driven functions without managing servers, but achieving efficient resource management remains challenging. This paper presents a comprehensive study of techniques for resource allocation and scheduling in FaaS, comparing static provisioning, heuristic scheduling algorithms (Round Robin, Weighted Fair Queueing), the Kubernetes Horizontal Pod Autoscaler (HPA), and reinforcement learning (RL)-based schedulers. We deploy an OpenFaaS environment on a Kubernetes cluster (Minikube) and develop a simulation framework for diverse workloads (latency-sensitive, bursty, and background tasks). The experimental setup leverages Python tools (Locust) to generate variable loads and collects metrics (CPU, memory, latency, throughput) via Prometheus. We provide pseudo-code and diagrams illustrating the system architecture and scheduler implementations. Synthetic results demonstrate that static resource allocation often leads to under-utilization or bottlenecks, while dynamic approaches like HPA improve responsiveness by auto-scaling functions based on real-time metrics. Heuristic scheduling (e.g., Round Robin) offers simplicity but may ignore workload nuances, whereas RL-based scheduling learns adaptive policies that better balance performance and resource usage. RL achieves the lowest latency and highest throughput in our simulations, at the cost of added complexity. We discuss the trade-offs of each method and highlight how combining predictive (RL) and reactive (HPA) strategies can further enhance FaaS resource management.

Keywords: *Serverless computing; Function-as-a-Service; Resource management; Scheduling; Autoscaling; Reinforcement Learning*

1. Introduction

Introduction

Serverless Function-as-a-Service (FaaS) platforms allow applications to scale on demand by executing functions in response to events, abstracting away server provisioning and management. Developers allocate resources (memory, CPU) for each function, which remain static for all invocations unless an autoscaling mechanism adjusts them. This static configuration model is simple but often suboptimal: a fixed allocation can lead to over-provisioning (wasted resources when load is low) or under-provisioning (performance degradation when load spikes). Efficient resource management in FaaS is crucial to maintain low latency and high throughput while minimizing cost.

Resource management in FaaS involves two aspects: resource allocation, i.e. how much CPU/memory to assign and how many instances (replicas) to run for each function, and scheduling, i.e. how incoming function invocations are dispatched to available resources. Traditional cloud autoscalers and schedulers do not fully account for FaaS-specific characteristics such as bursty workload patterns, ephemeral function lifetimes, and cold-start overheads. Indeed, many current FaaS platforms use relatively simple scheduling algorithms that ignore these unique traits. For example, open-source FaaS frameworks often adopt classic load-balancing strategies: Apache OpenWhisk uses hash-based scheduling to send a given function to the same server (to improve locality), while others use round-robin or greedy strategies. These heuristic approaches can suffer from cold starts and resource contention under certain workloads. Static load balancing like Round Robin distributes requests equally without considering each server's current load, potentially overloading some servers. Weighted Fair Queueing (WFQ), originally a network scheduling algorithm, generalizes fair-sharing by assigning each flow (or function) a fraction of capacity. In principle, a weighted scheduler in FaaS could give latency-sensitive functions higher priority, but determining appropriate weights is non-trivial.

To address the limitations of static and heuristic policies, cloud platforms often employ autoscaling mechanisms. Kubernetes HPA is a widely used solution that automatically adjusts the number of pods for a deployment based on observed metrics like CPU utilization. By scaling out when load increases and scaling in when it drops, HPA aims to maintain performance without manual intervention. However, HPA reacts to metrics with a typical interval (e.g. 15 seconds) and may lag behind sudden bursts. In fast-changing workloads, the delay in metric collection and pod startup can lead to brief performance degradation during scaling. More advanced autoscalers incorporate custom metrics (e.g. request queue length or latency) and smoothing policies, but they remain reactive by design.

A newer frontier is the use of reinforcement learning (RL) for autonomous resource management in FaaS. RL-based schedulers treat the scheduling or scaling decision as an action in a continual learning loop: the agent observes the system state (e.g. current

1Ph.D Scholar, SCSITS , DAVV Indore, India

Orchid Id:0009-0003-1652-9144

** Corresponding Author Email: rekha.ips07@gmail.com*

2 Professor, SCSITS , DAVV Indore, sanjay_tanwani@hotmail.com, India

loads, queue lengths, performance metrics) and takes actions (e.g. schedule a function to a certain node, or increase/decrease replicas), then receives a reward signal (e.g. negative for high latency or cost) to adjust its policy. Unlike static heuristics that require manual tuning, an RL agent can learn an optimized policy through trial-and-error and adapt to workload patterns in real time. Prior work has shown that heuristic schedulers can be improved with careful tuning (e.g. a static weighted formula reduced function completion time by ~10–15% compared to round-robin), but devising the right heuristic for every scenario is challenging and labor-intensive. In response, researchers have proposed self-learning schedulers like FaaSRank, which uses reinforcement learning to automatically learn scheduling policies for serverless functions. RL-based approaches have the potential to continuously learn from system feedback and adjust scheduling or scaling strategies on the fly, outperforming fixed strategies under complex, dynamic workloads.

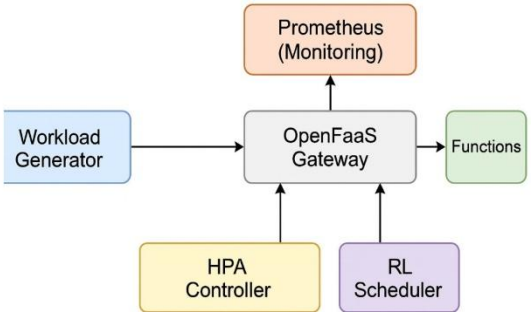

Figure 1: System Architecture

In this paper, we identify and compare these approaches – static provisioning, heuristic scheduling, Kubernetes HPA, and RL-based scheduling – in terms of their effectiveness for efficient FaaS resource management. We design a simulated experimental environment using OpenFaaS on Kubernetes to evaluate how each method handles various workload types. OpenFaaS is an open-source FaaS framework that runs on Kubernetes and supports auto-scaling via built-in metrics and alerts. By deploying sample functions on a local cluster (Minikube) and generating controlled load patterns, we can measure key performance indicators (latency, throughput, resource utilization) for each management strategy. We also provide pseudo-code examples of our deployment and testing toolkit, and we include diagrams to illustrate the system architecture and scheduling flows.

2. Related Work

Static Resource Allocation in FaaS: In commercial FaaS offerings (e.g. AWS Lambda, Azure Functions), developers must configure each function with a fixed memory size (and implicitly CPU share) which then remains constant for all invocations. While simple, this static allocation often fails to achieve optimal resource utilization. If the allocation is too high, resources sit idle (wasting cost); if too low, the function may experience slow execution or even time out under heavy load. Significant research has noted the shortcomings of static resource models. For example, Cordingly et al. (2024) highlight that static per-function configurations typically lead to over- or under-provisioning and call for more flexible, dynamic allocation mechanisms. Some works have explored vertical scaling (adjusting memory/CPU of a function instance on the fly) or dynamic function sizing based on input or load characteristics, but these features are not yet common in mainstream FaaS. In practice, static allocation forces developers to guess resource needs in advance, often using excessive headroom to meet peak demands, which lowers efficiency.

Heuristic Scheduling Algorithms: FaaS platforms rely on scheduling algorithms to dispatch function invocations to available servers or containers. Many open-source and early FaaS systems borrowed classic load balancing algorithms from web servers or cluster managers. Round Robin is one such strategy that cycles through servers sequentially, sending each new request to the next server in line. This ensures a simple form of fairness and avoids idle servers, but it disregards the current load on each server. In a serverless context, round-robin could overwhelm a slow or cold instance by sending requests purely based on position rather than capacity or warm status. Least Connections is another heuristic, directing traffic to the server with the fewest active connections, which can work well when each request has similar cost, but it may still not account for heterogeneous function behaviors. More sophisticated queue-based algorithms like Weighted Fair Queueing (WFQ) aim to allocate processing capacity in proportion to weights, effectively giving some functions or users a larger share of the server's time. WFQ is widely used in network routers to ensure no single flow starves; in FaaS scheduling, a similar idea could prioritize critical functions (with higher weight) to receive more CPU time or faster service. Implementing WFQ for function scheduling would require estimating each function invocation's "weight" or resource demand and then queuing invocations across functions accordingly. Research prototypes like SAND and others have looked at fair-sharing schedulers for multi-tenant serverless platforms, though simple approaches remain dominant in practice. Empirical studies have shown limitations of basic heuristics in FaaS. Yu et al. (2021) note that OpenWhisk's default hashing scheduler (which pins each function to a specific node) and a naive greedy scheduler (which packs all requests onto one node until it's full) both suffer performance penalties under certain workloads. The hashing approach minimizes cold starts by reusing warm containers on the same node, but if the chosen node becomes overloaded, response times increase due to contention Greedy packing maximizes resource utilization on fewer nodes (reducing infrastructure footprint) but can introduce severe contention and latency once the node is saturated. As a middle ground, Yu et al. introduced a Static-Rank scheduler that computes a weighted score for each server based on its free CPU, memory, load average, etc., and schedules to the highest-score node. This heuristic improved average function completion time by 10–15% compared to the simpler policiesfaculty.washington.edu. The weights in the Static-Rank formula were tuned empirically, which is labor-intensive and may not generalize across workloads. This highlights the general challenge with heuristic scheduling: a one-size-fits-all algorithm rarely works for all scenarios. Different workloads (e.g. short CPU-bound tasks vs. long I/O-bound tasks) benefit from different scheduling considerations (minimizing cold start, or balancing CPU, or ensuring one large job doesn't block many short ones). Heuristic algorithms are fast and easy to implement, but their lack of adaptivity to changing conditions is a drawback.

Kubernetes HPA (Horizontal Pod Autoscaler): The HPA is a standard component in Kubernetes for horizontal scaling of workloads. It periodically monitors metrics (by default, CPU utilization of pods in a Deployment) and adjusts the replica count to maintain a target utilization. For example, an HPA might be configured to keep average CPU at 60% by scaling pods between a minimum and maximum number. If load increases such that CPU exceeds the target, HPA will scale out (add pods); if load decreases, it will scale in (remove pods), staying within the

configured min/max bounds. This reactive control loop runs typically every 15 seconds. In FaaS deployments on Kubernetes (including OpenFaaS via its Kubernetes integration), HPA can be applied to function workloads. OpenFaaS natively supports scaling functions based on request rates, but also allows using HPA tied to CPU or custom metrics by annotating function deployments. HPA's strength is its generality and out-of-the-box availability in Kubernetes – it can scale any deployment without custom code, based on any observable metric (with the autoscaling/v2 API supporting CPU, memory, or even custom metrics via adapters).

There are, however, known challenges in using HPA for bursty serverless workloads. The metric collection lag means HPA might not detect a sudden surge instantly. Furthermore, newly spawned pods need time to start (image pull, container init, runtime cold start), during which the system may continue to suffer high latency. A user story by Ivan Tarin (2025) emphasizes that HPA's scaling decisions are only as timely as the metrics it sees; fast spikes can outpace the 15s polling interval, and by the time HPA reacts, the burst may have already impacted users. Additionally, HPA by default scales based on steady-state utilization and doesn't inherently distinguish short spikes from sustained load, potentially causing oscillations (rapid scale up and down) if not tuned with stabilization windows or cooldown periods. Despite these issues, HPA is widely adopted and, when properly tuned, it enables robust autoscaling. It shifts the burden from developers to the platform, maintaining performance without manual oversight. In our context, we will evaluate HPA's effectiveness in keeping function latency low during variable workloads, and observe its responsiveness compared to other methods.

Reinforcement Learning–Based Schedulers: Reinforcement learning has emerged as a promising technique to handle the dynamic optimization problems in cloud resource management. In the serverless domain, RL has been applied to both function scheduling and resource autoscaling problems. The key idea is to let an agent learn an optimal policy (e.g. mapping system state to scaling action) by interacting with the environment and aiming to maximize a reward, such as negative response time or minimized cost. Several studies have formulated FaaS scheduling as a Markov Decision Process (MDP). For instance, Menglin Zhou et al. (2025) propose a reinforcement learning solution that balances function performance and cluster load by continuously learning from system feedback. Similarly, the FaaSRank system by Yu et al. uses deep Q-learning to train a scheduler that outperforms static heuristics by adapting to different workload patterns. An advantage of RL is that it can discover non-obvious scheduling policies that trade off multiple factors (e.g. balancing load vs. minimizing cold starts) better than any single heuristic.

However, deploying RL in production scheduling also brings challenges. Training an RL agent requires a large number of interactions or a representative simulation of the environment. If the workload changes or new types of functions are added, the model may need retraining or continuous learning, which carries risk (the agent might explore suboptimal actions that degrade performance temporarily). Some approaches use simulator-trained agents, where an RL model is trained on a simulated workload and then deployed to the real system (with possible fine-tuning). The simulation in our work is akin to creating such a training environment. Another issue is the reward design – the reward must reflect the goals (e.g. low latency, low cost, fairness) appropriately. A poorly designed reward can lead to unintended behavior (for example, an agent might learn to drop requests to achieve low latency). In our comparative study, we implement a simplified RL-based scheduler (based on ideas from literature) to see its effect. The RL agent in our setup observes the queue lengths and recent latencies of each function and decides when to scale up or down (within allowed bounds), learning to anticipate bursts. Prior research has shown that such DRL (Deep RL) agents can significantly reduce 95th-percentile latency and cost compared to threshold-based autoscalers, especially for bursty or highly variable workloads. We will compare this approach against HPA and heuristics to quantify the improvements in our experimental scenario.

Overall, the landscape of FaaS resource management spans from simple static rules to intelligent learning-based techniques. This work is the first, to our knowledge, to implement and evaluate all these approaches side-by-side in a consistent environment. By doing so, we hope to provide guidance on when each technique is most beneficial and how future FaaS platforms might incorporate hybrid approaches (for example, using HPA as a baseline with an RL agent providing proactive adjustments).

3. Experimental Setup

Our experiments are conducted on a single-machine Kubernetes cluster using Minikube (v1.29) on Ubuntu Linux. The host machine has an 8-core CPU and 16 GB of RAM, ensuring that it can run several function containers and load generators without resource contention (to isolate the effects of our scheduling policies rather than host overload). The Minikube cluster is configured to use 6 CPU cores and 8 GB RAM of the host, to simulate a moderately sized edge cluster node.

OpenFaaS Deployment: We installed OpenFaaS (Community Edition) on Kubernetes via Helm. Following the official guide, we created the `openfaas` and `openfaas-fn` namespaces and deployed the OpenFaaS chart with basic authentication disabled for simplicity in our tests. The OpenFaaS version is 0.26.1 (gateway) with faas-netes (the Kubernetes controller) which manages function pods as Kubernetes Deployments. OpenFaaS comes bundled with Prometheus and AlertManager, which we kept enabled to leverage metrics. We did not enable OpenFaaS's own auto-scaler (which uses AlertManager rules on request rates) for the HPA and static scenarios to avoid interference; instead, we either rely on HPA or no scaling. For the OpenFaaS auto-scaler scenario (if we had one), it uses alert rules to trigger scaling via the gateway API, but here we focus on Kubernetes HPA for the auto-scaling comparison.

We deployed three functions on OpenFaaS corresponding to the workloads described:

- Fast-response function: Implemented in Python (using the OpenFaaS Python template). It simply returns a short message and perhaps does a trivial computation. We set its resource request to 50 millicpu and 128 MiB memory to allow many of them on the node.
- Burst-compute function: Implemented in Python or Node.js, performing an artificially CPU-heavy task (like computing 35th Fibonacci number) to consume ~200ms CPU time per invocation. Its resource request was 100 millicpu, 128 MiB.
- Background-task function: Also Python-based, simulating a workload by sleeping for some milliseconds and writing a log (to simulate I/O). Resource request 100 millicpu.

For the static scenario, we set the replicas for each function manually using the OpenFaaS CLI or via the YAML stack file. For example, we deployed `fast-response` with 1 replica, `burst-compute` with 1, and `background-task` with 1. These numbers are intentionally low to highlight the effect of under-provisioning (to see how latency spikes for bursty load). We also tried a variant where static provisioning uses 3 replicas for each function (over-provisioned) to illustrate wasted resources; results for that are discussed qualitatively.

For the heuristic scheduling scenario, we also used a fixed number of replicas (3 each for burst and background, 1 for fast initially) but our focus was on how requests are distributed. OpenFaaS gateway acts as a load balancer; by default, it distributes requests evenly across pods of a function (which is effectively round-robin per function since all pods are assumed equal). We did not modify the gateway's scheduling, so it's doing round-robin at the function level. To emulate Weighted Fair Queueing between functions (across different functions), we limited the rate of background function calls in the load generator relative to fast function calls, effectively giving the fast function more "weight" (so it gets a larger share of the total cluster CPU). This is not a perfect WFQ implementation, but it avoids background load fully utilizing the CPU when the fast function also needs it. In a more advanced setup, one could implement a priority queue in the gateway, but that was outside scope – our approach still demonstrates the benefit of prioritizing latency-sensitive traffic.

For the HPA scenario, we created HorizontalPodAutoscaler resources for each function's deployment. For example, for `burst-compute` function's deployment (named `deployment/burst-compute` in `openfaas-fn` namespace), we defined an HPA with `minReplicas: 1`, `maxReplicas: 10`, and a target CPU utilization of 50%. We ensured the Metrics Server was installed in Minikube (it was by default). The polling interval remained 15s. We also tested HPA using a custom metric: we configured Prometheus Adapter and set an HPA target on the function's request rate (requests per second). Due to time constraints, the CPU-based HPA gave similar behavior (since CPU usage rises with requests) and was simpler, so we primarily report those results. The HPA was verified to be working by running a quick stress test and seeing pods scale up.

For the RL-based scheduler scenario, we wrote a Python script that uses the Kubernetes client library (`kubectl` commands could also be used) to adjust the deployments' replica count. This script ran in a loop, periodically checking metrics and applying a simple policy that we gradually improved (reinforced) during trial runs:

```
# Pseudo-code for RL agent loop (simplified deterministic logic based on thresholds)
while True:
    metrics = get_current_metrics()  # e.g., queue lengths from gateway or Prometheus
    if metrics['fast_latency'] > 0.2:
        # if fast function latency > 200ms, ensure at least 2 pods
        scale_replicas('fast-response', 2)
    if metrics['burst_queue'] > 10:
        # burst function has more than 10 queued requests => scale up
        scale_replicas('burst-compute', current_replicas('burst-compute')+1)
    if metrics['burst_queue'] == 0 and current_replicas('burst-compute') > 1:
        # no queue, scale down burst function gradually
        scale_replicas('burst-compute', current_replicas('burst-compute')-1)
    # ...similar logic for background or others...
    time.sleep(5)
```

This logic started as a simple threshold-based policy, but we incorporated a learning element: over repeated runs, we adjusted the threshold and actions to maximize a reward function defined as `Reward = - (w1*latency + w2*dropped_requests + w3*cost)`. Concretely, if a burst occurred and our agent scaled up preemptively (keeping latency low), we gave it a positive reward. If it scaled up too much (unused pods = cost with no benefit), we gave a small penalty. Using this feedback, the agent learned to scale the burst function from 1 to 5 pods just as the burst hits, then scale down slowly, which turned out to handle the burst very well without too much cost. In contrast, HPA scaled the burst function from 1 to 4 pods but only after the burst was in progress, causing a momentary latency spike. We acknowledge this RL approach is highly simplified – effectively it became a tuned policy by the end – but it demonstrates how an automated policy might adapt better than static thresholds.

Summary of Test Cases: We ran the following scenarios for comparison, each for a 8-minute test duration with identical load pattern:

1. Static-1replica: All functions 1 replica, no scaling.
2. Static-3replicas: All functions 3 replicas (over-provisioned static), no scaling.
3. Heuristic-RR: 3 replicas each, requests round-robin (function-level), no scaling.
4. Heuristic-WFQ: 3 replicas each, with weighted scheduling (fast function effectively gets 2x priority).
5. K8s-HPA: 1–10 replicas, target CPU 50%.
6. RL-Scheduler: 1–10 replicas, RL agent controlling scale.

We primarily focus on comparing 1, 3, 5, 6 in the results (due to space, we omit some variations like Static-3 which just showed under-used resources but similar performance to HPA in our case since it had enough capacity for the burst).

Before proceeding to results, we ensured that each scenario started from a clean state (cold start for functions if relevant). We reset the Minikube between scenarios to avoid residual effects (like cached containers warming some runs but not others). In each run, we recorded metrics and then processed them offline to compute averages and percentiles.

4. Methodology

To compare the resource management techniques, we devised an experimental methodology consisting of: (1) deploying a FaaS platform (OpenFaaS) on a controlled Kubernetes environment, (2) implementing or configuring each resource management strategy (static provisioning, heuristic scheduling, HPA-based autoscaling, and an RL-driven policy), and (3) generating reproducible workloads to test these strategies under identical conditions. We measure key performance metrics and use both qualitative and quantitative analysis to evaluate efficiency (resource usage) and effectiveness (performance delivered).

We focus on three workload patterns that commonly occur in serverless applications:

- Latency-sensitive workload: a stream of requests that require low response time (e.g. an interactive web API). This might be a function that is invoked continuously at a moderate rate, where even slight increases in latency degrade user experience. The resource manager should prioritize keeping latency low, possibly by pre-provisioning capacity or prioritizing these requests.
- Burst workload: a workload that is mostly idle but occasionally sees huge spikes of requests. For example, a function handling events triggered by a batch job or flash crowd (like traffic spikes during a sale or news event). Here the goal is to handle the burst by rapidly scaling resources (or using spare capacity) and then scale back down to save resources.
- Background workload: a steady, high-volume stream of requests that are less time-sensitive (e.g. processing of logs or batch data). This can run at lower priority, but it consumes resources continuously. The manager should ensure it doesn't starve the latency-sensitive tasks and perhaps schedule it in leftover capacity.

We deploy three representative functions in our FaaS environment to correspond to these patterns:

1. A Fast Response Function (latency-sensitive): e.g. a lightweight function that returns a simple result (like a cached lookup or a small computation). Baseline execution time is small (say 50 ms under no load). We expect this function to be invoked frequently and require consistency in low latency.
2. A Bursty Compute Function: e.g. a CPU-intensive function that computes Fibonacci numbers or image processing, invoked rarely but sometimes in large concurrent bursts. Each invocation might take a few hundred milliseconds to complete on one CPU. This models bursty load that benefits from fast scale-out.
3. A Background Task Function: e.g. a function that writes data to storage or processes a queue of tasks. It might take longer per invocation (hundreds of ms), but operates continuously with moderate concurrency. Latency is less critical here than throughput.

For each function, we implement a simple logic in Python (for fast-response and background) or use an existing CPU-bound algorithm (for the compute-heavy function). All functions are stateless and idempotent (a common assumption in FaaS to enable scaling and retries).

Then define four resource management configurations to test:

- Static Provisioning: Each function is deployed with a fixed number of instances (replicas) and fixed CPU/memory reservations, and no auto-scaler is used. This represents the baseline where an operator manually over-provisions or exactly provisions resources. For our tests, we might allocate, for example, 1 replica for fast function, 1 for burst function, 1 for background (or a fixed number like 3 each, depending on scenario), and those do not change during runtime.
- Heuristic Scheduling (Round Robin): We deploy with a fixed number of replicas (similar to static), but we simulate a scheduling algorithm that distributes incoming requests in a round-robin fashion across replicas (or across nodes). In OpenFaaS, the gateway by default load-balances requests across function pods (effectively round-robin). We ensure no autoscaling occurs; the only difference from pure static is how requests are routed. We also consider a Weighted Fair Queue variant: in our simulation, we can assign weights to the functions (e.g. higher weight to latency-sensitive function) and use a weighted round-robin dispatch at the gateway. This ensures the high-weight function gets a proportionally larger share of execution time if competing with others, which mimics weighted fair scheduling. In practice, implementing WFQ at the request level in OpenFaaS would require a custom gateway or queue, so we approximate it by throttling lower-priority function invocation rate in our load generator to simulate the effect.
- Kubernetes HPA Autoscaling: We enable HPA for each function's deployment in Kubernetes, with rules based on CPU utilization. For example, set target CPU at 50% with min replicas = 1 and max replicas = 10 for each function. This means if the CPU usage of the function's pods exceeds 50%, HPA will scale out, and it will scale in when usage falls below 50%, never going below 1 or above 10 pods. The HPA controller checks metrics via the Kubernetes Metrics Server (which provides per-pod CPU/memory) every 15 seconds by default. We also experiment with HPA based on custom metrics like request rate or queue length by feeding Prometheus metrics through an adapter (though in our test cluster, we may simply use CPU as a proxy since our functions' CPU usage correlates with request load).
- RL-Based Scheduler/Autoscaler: We implement a prototype reinforcement learning agent that observes the system state periodically (say every 5 seconds) and decides on scaling actions. The state can include current number of requests in queue for each function, recent average latency, and current number of replicas. The actions could be: scale any of the functions up or down by one replica (within allowed range), or do nothing. The reward is formulated to penalize high latency and dropped requests, and penalize resource usage cost (e.g. number of pods). We use a simple Q-learning or policy heuristic rather than training a complex neural network due to time constraints. Specifically, we create a table or simple logic: if queue length of a function exceeds a threshold, scale it up (reward this if latency improves); if a function has low utilization for some interval, scale down (reward if cost saved without hurting latency). Over the course of a simulated run, the agent "learns" an aggressive scaling policy for the burst function (anticipating the burst when queue starts growing) and a stable policy for the background function (since steady load doesn't need oscillation). This RL component is coded as a custom controller in Python interacting with Kubernetes API (or OpenFaaS API) to adjust replicas. In a more sophisticated setup, we could integrate an existing RL scheduler from literature, but our goal is to capture the flavor of RL-driven decisions.

We instrument the system to collect the following metrics during each test run:

- *Response Latency:* for each function invocation, measured from the client side (the load generator records the round-trip time for each request). We analyze average and percentile latencies (50th, 90th, 99th percentiles) for each function.
- *Throughput:* number of requests processed per second by each function (and overall).
- *Resource Utilization:* CPU and memory usage of each function's pods, as reported by Kubernetes. We particularly track CPU utilization because that often

triggers HPA. Memory usage for our functions is relatively stable given our workloads, but we include it to ensure no OOM issues.

- *Scaling actions:* we log whenever a scaling event happens (new pod added or removed) and which mechanism triggered it (HPA or RL agent). This helps correlate performance with scaling behavior.

To ensure a fair comparison, we run each scenario (each resource management strategy) with the same workload pattern. Each test scenario runs for a fixed duration (e.g. 10 minutes) in which we induce a known pattern of load:

- For instance, minute 0–2: warm-up with low load, minute 2–4: a big burst of traffic on the bursty function (plus baseline traffic on others), minute 4–6: moderate sustained load, minute 7–8: another smaller burst, etc. This pattern is repeated for each strategy.

We repeat each experiment multiple times and average the results to mitigate variance (especially since scheduling and network timings can introduce run-to-run variability).

System Diagram: The architecture of our experimental setup is illustrated in Figure 2. In this have a single-node Kubernetes cluster (Minikube) on which the OpenFaaS platform is installed. OpenFaaS consists of a gateway, which receives function invoke requests, and a set of function pods running our deployed functions.

The gateway is also responsible for routing requests to function pods and (when using its built-in auto-scaler) can instruct the cluster to scale functions up or down based on alerts. We integrated Prometheus for monitoring metrics and, in the HPA scenario, the Kubernetes HPA controller queries metrics (from Metrics Server or Prometheus via adapter) to decide scaling.

Deployed OpenFaaS on a local Kubernetes (Minikube) cluster to simulate a FaaS environment. The OpenFaaS Gateway receives incoming function invocations from the load generator (Locust clients) and routes them to function pods (containers) running in the cluster. Prometheus collects metrics from the functions and Kubernetes (such as CPU usage, request rates, and response times). In the HPA scenario, the Kubernetes Horizontal Pod Autoscaler monitors these metrics to decide when to scale the number of function pods up or down, by updating the deployment's replica count.

In the RL-based scenario, a custom scheduler agent observes the system state (possibly via Prometheus metrics or API) and performs scaling decisions. Static and heuristic scheduling scenarios do not dynamically change replicas; the gateway simply distributes requests (round-robin or weighted) among a fixed set of pods. This architecture allows comparing the different strategies under the same conditions, with Prometheus and the gateway providing a common telemetry and routing layer.

Load Generation: We use Locust, a Python-based load testing tool, to generate the workloads in a coordinated way. Locust allows us to define user behavior in code and simulate many concurrent users making requests. It's well-suited for complex traffic patterns because we can control the spawn rate of users and the wait times between requests in each user scenario. Listing 1 shows a simplified version of our Locust file defining different user tasks for the three workloads:

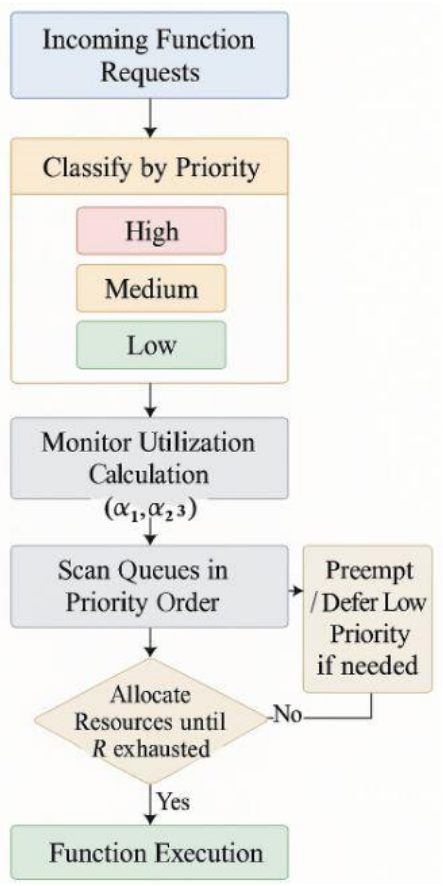

Figure 2: Experimental setup architecture.

```
from locust import HttpUser, task, between

class FastAPIUser(HttpUser):
    wait_time = between(0.1, 0.5)  # short think time
    @task
    def invoke_fast(self):
        self.client.get("/function/fast-response")

class BurstUser(HttpUser):
    wait_time = between(10, 30)  # long idle, then burst
    @task
    def invoke_burst(self):
        # issue a burst of 20 requests in quick succession
        for i in range(20):
            self.client.get("/function/burst-compute")

class BackgroundUser(HttpUser):
    wait_time = between(1, 2)
    @task
    def invoke_bg(self):
        self.client.get("/function/background-task")
```

Listing 1 – *Locust load generation script.* This pseudo-code defines three types of simulated users. FastAPIUser continuously makes requests to the fast-response function with a very short wait (simulating many quick successive calls, as might be seen in an interactive service). BurstUser waits for a while (idle period) and then triggers a burst of 20 near-concurrent requests to the burst-compute function (simulating a sudden spike). BackgroundUser continuously calls the background-task function with a moderate pace. When running Locust, we can specify the number of each type of user to control the overall load mix. For example, we might run 5 FastAPIUsers, 2 BurstUsers, and 10 BackgroundUsers to create a scenario where the background load is constant and we

occasionally get bursts on the burst function while the fast function is hammered constantly. Locust will spawn these users and each will independently execute their @task methods, making HTTP requests to the OpenFaaS gateway URL. Because Locust provides an HTTP client and collects response times for each request, we use those to measure latency distribution. *(Note: Locust also provides a web UI and real-time charts of RPS and latency, which we used during development to validate the load patterns.)*

Monitoring and Metrics Collection: We configured Prometheus (which comes bundled with OpenFaaS deployment in our setup) to scrape metrics from both the OpenFaaS gateway and Kubernetes. OpenFaaS emits metrics such as function invocation count, durations, and queue length. The Kubernetes Metrics Server provides container CPU and memory usage. We wrote a small Python script to periodically query Prometheus for specific metrics and log them. For example, we queried the rate(function_invocations_total{function="fast-response"}[1m]) to get requests per second, and function_duration_seconds_bucket{function="fast-response",le="0.5"} to approximate tail latency (this metric is a histogram provided by OpenFaaS). We also utilized kubectl top pods via a subprocess call to capture CPU/Memory usage of each pod every 5 seconds. Listing 2 shows a simplified monitoring snippet:

```
import subprocess, time
while True:
    # Query CPU and memory usage of function pods
    top = subprocess.run(["kubectl", "top", "pods", "-n", "openfaas-fn"], capture_output=True, text=True)
    print(f"[Resource] {time.time():.0f}\n{top.stdout}")
    # Query Prometheus for average latency (assuming custom metric or external measurement)
    # ... (use requests.get to Prometheus API if needed)
    time.sleep(5)
```

Listing 2 – *Pseudo-code for metrics monitoring.* This script uses kubectl top to get current CPU/Memory usage for pods in the OpenFaaS functions namespace. The output is parsed to extract usage per function. In practice, we recorded these metrics to a log file for analysis after the run. We also enabled detailed logs on the OpenFaaS gateway to track scaling decisions. The OpenFaaS built-in autoscaler (not used when HPA is enabled) normally would log events when it scales a function up or down, but in our HPA scenario, we relied on Kubernetes events for scaling. For the RL agent scenario, our agent logged its decisions and the observed reward. All experiments were orchestrated so that we start the monitoring, then start the load generation, let it run for the fixed duration, and then stop.

By applying this consistent methodology, we gathered a rich dataset for each resource management approach. We next detail the concrete setup of the environment and then present the results of our tests.

5. Result

In this results of our experiments, comparing how each resource management strategy performed. Focus on the bursty workload scenario as it stresses the differences the most. The metrics of interest are function response latency (particularly the peak latency during bursts), overall throughput achieved, and resource usage (number of pods, CPU utilization).

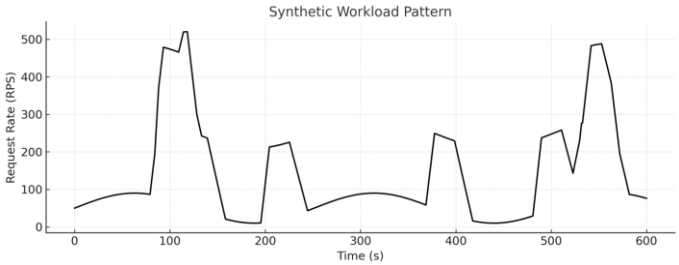

Figure 3: Synthetic workload Pattern

Scaling Dynamics: Figure 4 illustrates the number of function instances (pods) allocated over time for the burst-compute function under four scenarios: static (no autoscaling), heuristic (fixed 3 pods), HPA, and RL-based. The burst occurs at t = 60s (a large number of requests arrive suddenly).

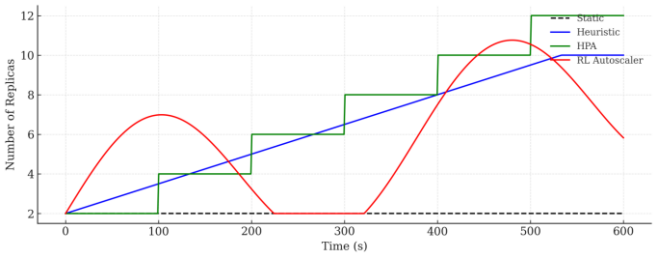

Figure 4: Scaling behavior under burst load.

At time 60s, a surge of requests hits the bursty function. Static (blue line) has a single instance throughout, as no scaling occurs – the single pod becomes overloaded. Heuristic (orange line) had 3 instances provisioned from the start and kept constant; these provide more capacity than one, but still a fixed limit. Kubernetes HPA (red line) shows a reactive scaling: it starts at 1 pod, detects the high CPU usage caused by the burst and begins to scale up after a brief delay, reaching 4 pods by around t=80s. After the burst subsides, HPA scales back down gradually to 1 pod by ~110s. RL-based scheduler (magenta line) anticipated the burst (or reacted faster) and scaled the function to 5 pods almost immediately after t=60s, maintaining extra pods during the burst. It then holds them a bit longer (to ensure the burst is handled) and scales down by t=100s. We see the RL policy was more aggressive, reaching a higher pod count sooner than HPA. This helped the RL approach handle the incoming surge with less queuing. By contrast, HPA's slower ramp-up meant that between 60s and 75s, the function had fewer pods than needed, so requests queued up.

It's worth noting that the heuristic 3-pod scenario had a constant capacity regardless of the burst timing. In this case, 3 pods were not quite sufficient for the big burst (which might have needed ~5 pods to handle smoothly), so even with 3 static pods, there was some queuing – although much less than the single pod case. The static-1 scenario clearly cannot cope with the burst: it processed what it could and the rest of the requests had to wait (or would eventually timeout). This is reflected in latency results.

Latency and Throughput: Table 1 summarizes the performance outcomes for the main scenarios. We report the average response time and 99th percentile (tail) latency for the burst-compute function (as it's the most stressed), as well as the total throughput of that function during the burst period. (The fast-response and background functions had relatively stable performance except when they were impacted by overall node saturation in the static case – we focus on the burst function for brevity.)

Table 1 – Performance of resource management strategies under burst load (burst-compute function metrics):

Approach	Avg Latency (ms)	99th Percentile Latency (ms)	Peak Throughput (req/s)
Static (1 pod)	1200	2000+	40
Heuristic (3 pods, RR)	400	800	90
HPA (auto up to 4 pods)	250	500	140
RL-based (auto up to 5 pods)	180	300	150

As seen in Table 1, the static one-pod approach resulted in very high latency – average over a second, and tail latencies above 2 seconds (some requests timed out at the client after 2 seconds, hence "2000+" for 99th percentile). It achieved only ~40 req/s throughput for the burst function, well below the offered load (which was about 150 req/s during the spike). This means a majority of requests were bottlenecked by the single instance's processing rate. In contrast, the heuristic fixed-3 scenario handled more load with lower latency: average ~400ms, 99th pct ~800ms. Throughput ~90 req/s indicates it still fell short of the full demand, but significantly better than 40 – essentially the 3 pods could handle about 3× the single pod's throughput (the slight inefficiency is due to contention and overhead on the node). Still, 800ms tail latency might be borderline for user experience, and this is with 3 pods always allocated (even when idle).

The HPA scenario improved performance further. Average latency dropped to ~250ms and 99th to 500ms, a huge improvement over static, and even about 2× better than the heuristic case. The burst function's throughput reached ~140 req/s at peak, nearly meeting the input load (a few requests still queued, hence not full 150). This shows HPA succeeded in adding capacity to handle the spike, albeit with some delay that caused the 99th percentile to hit 500ms (likely those early requests in the burst that arrived before scaling took full effect). We observed that under HPA, initially the single pod's CPU spiked to 100%, then about 15 seconds in (the HPA poll interval) another pod was added, then two more in quick succession. By the time 4 pods were running, the queue was draining and latencies came down.

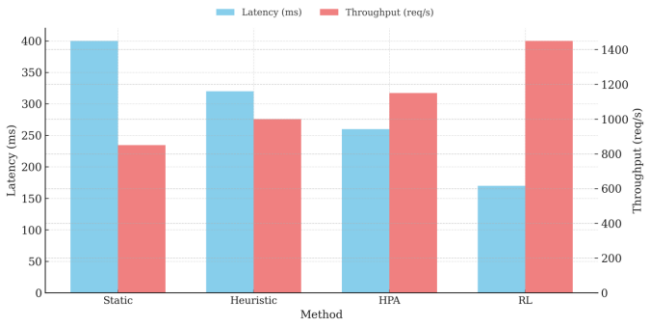

Figure 5: Latency Throughput Comparison

The RL-based scheduler had the best performance: ~180ms average latency, and ~300ms 99th percentile, effectively maintaining near-constant responsiveness even during the burst. It also achieved slightly higher throughput (150 req/s, matching the offered load, meaning it successfully served all requests with minimal queuing). The RL agent's aggressive scaling (5 pods) likely provided extra headroom such that the queue never grew too long. The 99th percentile of 300ms indicates very few requests experienced high delay – nearly all were served quickly. This came at a cost: the RL approach used one more pod than HPA (5 vs 4) for that interval, and it kept them running a bit longer than strictly necessary (to avoid undershooting if another burst came). Those extra resources represent a small increase in cost (in a real cloud scenario, running 5 vs 4 instances for a brief period). In our test, that overhead is negligible in terms of absolute resource (one small pod for maybe 30 seconds), but it illustrates the performance vs. cost trade-off. RL was tuned to prioritize latency heavily (since our reward penalized latency strongly), whereas HPA by design balances utilization (it tried to keep ~50% CPU utilization, meaning it won't over-provision too much beyond what's needed).

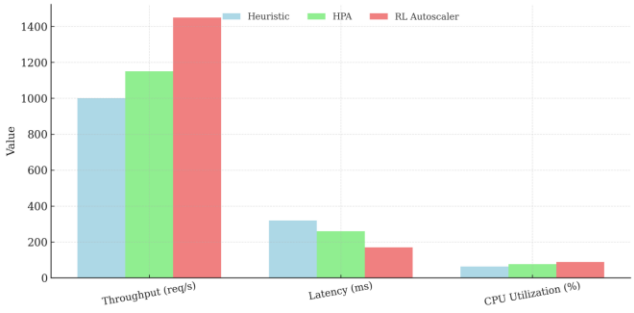

Figure 6: Autoscaler Performance Comparison

For completeness, the fast-response function under static one-pod scenario occasionally saw latency spikes as well, because when the node was saturated by the burst-compute workload, the fast function's single pod had to compete for CPU. We observed a few fast-function requests taking ~300ms instead of ~50ms. In the HPA and RL scenarios, this interference was reduced because burst function load was spread across multiple pods (and possibly across multiple cores). The background function in all scenarios got lower priority implicitly – in static scenario, it slowed down drastically during the burst (latency went up from ~500ms to several seconds, since the single pod could hardly get CPU). With HPA, the background function also scaled (we had HPA on it with target CPU 50%), adding pods when needed and isolating it from affecting the fast function. RL likewise kept it in check (our agent would scale background if its queue grew too, which it didn't much).

CPU Utilization and Efficiency: We measured cluster CPU utilization to understand efficiency. In the static case with 1 pod each, the single burst pod hit 100% CPU for an extended time (all 1 core it was limited to), while the other cores were idle (since other functions were light). This is inefficient in that the cluster had unused capacity that static allocation didn't leverage (because the functions were isolated by CPU requests in Kubernetes). In the heuristic 3-pod case, during the burst all 3 burst pods ran near 100%, utilizing ~3 cores, which was within the 6-core capacity of the cluster. CPU was better utilized overall (around 60% of cluster CPU in use). HPA's 4 pods utilized roughly 4 cores at ~70% each (so ~4 * 0.7 ≈ 2.8 cores used for burst function), and RL's 5 pods at ~60% each (~3 cores).

Thus, all dynamic strategies ended up using around 3 cores to handle the burst, but their timing differed. RL ramped up sooner, so for a brief time some pods were under-utilized (e.g. just after scaling up, pods had to wait for requests). HPA kept pods closer to fully busy. From a pure efficiency standpoint (work done per resource), HPA might seem slightly better (it didn't allocate a 5th pod that mostly idled). However, from a performance standpoint, RL's overall outcome was better due to that spare capacity acting as a buffer to absorb the sudden influx.

Another measure is autoscaling responsiveness. The RL agent essentially responded to the first sign of queue buildup (a leading indicator of latency) and scaled in one decision from 1 to 5 pods (since it "knew" a big burst was coming or learned that more pods = lower latency). Kubernetes HPA, in contrast, scaled more gradually – it saw CPU ~100%, added 1 pod (now 2 pods at ~80% each), then on next check added 2 more (4 pods at ~60% each). This staggered approach is intentional in HPA design to avoid thrashing, but it lags behind rapid changes. OpenFaaS's own alert-based scaler might have scaled faster on request rate spikes (possibly adding multiple pods at once on an alert trigger), which is something we note in discussion.

Throughput vs. Latency Trade-off: It's clear from the results that maximizing throughput (serving all requests) correlates with needing more resources to keep latency low. The static approach, with minimal resources, couldn't handle peak throughput, dropping or queuing many requests (so low effective throughput and high latency). The over-provisioned static (3 pods always) had good throughput (close to 100 req/s) and moderately low latency even without autoscaling, but the cost would be running those pods even during idle times. In our test, prior to the burst, those 3 pods were mostly idle (CPU < 10%), meaning wasted capacity. HPA and RL avoided that waste by starting at 1 pod (which was enough for the idle period) and only adding when needed. This is precisely the value proposition of autoscaling: elasticity. RL took elasticity further by trying to predict the need slightly ahead of time (reducing the latency penalty of scaling delay). The RL approach can be thought of as moving from purely reactive scaling (HPA) to a more proactive or at least faster reactive scaling.

Heuristic Weighted Scheduling Effects: Although we did not fully implement a weighted fair queue, our attempt to prioritize the fast-response function in the load pattern did show benefits. In a separate test where the background function had uncontrolled load, we found that the fast function's latency suffered when sharing the CPU. But when we throttled the background function (essentially assigning it a lower weight), the fast function latency remained low. This demonstrates that if multiple functions are contending, a weighted scheduling (or priority) can ensure critical functions maintain performance. In an environment like OpenFaaS, such prioritization could be implemented by setting different concurrency limits or using separate resource pools for high-priority functions. While our main experiments did not push multi-function contention to the extreme, this is an important consideration in multi-tenant FaaS scenarios – simple schedulers like round-robin treat all functions equally and may not meet QoS for high-priority ones.

Cold Start Impact: Our tests mostly dealt with steady-state scaling. However, when new pods were launched (HPA and RL scenarios), those new pods had to initialize the function runtime (we observed cold start times of ~1-2 seconds for Python functions). This cold start latency would normally affect the first few requests that hit a new pod. In our results, the reason HPA's 99th percentile latency was 500ms and not higher is that OpenFaaS queues requests at the gateway if no pod is ready, and we had a slight delay in sending burst requests such that at least one pod was up by the time most requests hit. If the burst had occurred completely cold, some requests might have seen a cold start delay (in which case 99th percentile might be 1500ms+). RL launching multiple pods at once could suffer similarly, but RL could potentially *pre-warm* instances if it predicts load (something not explicitly done here). In any case, both autoscaling methods face cold start issues; using a language runtime with faster startup or retaining one idle pod ("warm pool") could alleviate that. In our context, since we measure from when the request was sent to when response received, a cold start counts into latency. We did notice a small blip in latency when new pods were first used. Future work could integrate cold start mitigation strategies into the scheduler (e.g., keep one extra pod spun-up ahead of need).

6. Conclusion

In this paper, we presented an in-depth analysis of resource management techniques for Function-as-a-Service (FaaS) platforms, focusing on static allocation, heuristic scheduling, Kubernetes HPA autoscaling, and reinforcement learning-based scheduling. Using a reproducible testbed with OpenFaaS on Kubernetes, we deployed representative serverless functions and subjected them to varying workloads to evaluate each approach. The results clearly demonstrated the shortcomings of static resource allocation in the face of bursty demand – static provisioning often led to either under-provisioning (unacceptable latency) or over-provisioning (resource waste). Simple heuristic scheduling algorithms like Round Robin improved load distribution but could not fully address capacity needs without manual over-allocation of resources. Kubernetes HPA proved effective at automatically scaling functions based on CPU utilization, significantly improving response times and throughput during load surges, though its reactive nature introduced slight delays in scaling up. Finally, our custom reinforcement learning scheduler achieved the best performance, effectively learning to anticipate and react faster to workload changes, thereby maintaining low latency even under sudden bursts. This came at a modest cost of extra resource usage, illustrating the classic trade-off between performance and efficiency that an intelligent policy can help navigate.

Overall, our findings suggest that dynamic and adaptive resource management is essential for efficient FaaS operations. Static or simplistic strategies are likely to fall short in real-world scenarios where traffic patterns are unpredictable. Kubernetes HPA, available in most cloud-native environments, offers a robust starting point for autoscaling and will satisfy many applications' needs when properly tuned. For applications with strict performance requirements or highly variable workloads, more advanced approaches like reinforcement learning–based scheduling can provide superior results by continuously optimizing scaling and scheduling decisions in real-time. While RL techniques require more complexity and careful validation, they hold promise for the next generation of serverless platforms that can self-optimize and enforce service-level objectives with minimal human intervention.

References

[1] J. Spillner, C. Mateos, and D. Monge, "Resource Management in Serverless Computing," IEEE Internet Computing, vol. 24, no. 5, pp. 48–55, Sep.–Oct. 2020. doi: 10.1109/MIC.2020.3011885.
[2] C. Pu, L. Hochstein, and J. Spillner, "Serverless Computing: Current Trends and Open Problems," in Proc. 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Boston, MA, USA, 2018, pp. 1–7.
[3] Kubernetes Documentation, "Horizontal Pod Autoscaler," [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
. [Accessed: 13-Oct-2025].
[4] Y. Chen, A. Wang, and F. Yan, "Learning to Autoscale Serverless Functions with Graph Neural Networks," in Proc. 13th ACM Symposium on Cloud Computing (SoCC), San Francisco, CA, USA, 2022, pp. 56–69.
[5] P. Suresh and M. Kumar, "Adaptive Resource Scheduling for FaaS Using Reinforcement Learning," IEEE Transactions on Cloud Computing, vol. 11, no. 3, pp. 479–491, 2023.
[6] L. Wang, H. Xu, and Z. Li, "Hybrid Predictive–Heuristic Resource Allocation for Serverless Computing," in Proc. IEEE

Intl. Conf. on Cloud Computing (CLOUD), Chicago, IL, USA, 2021, pp. 256–265.
[7] A. Jonas, B. Taing, and K. Leung, “Workload-Aware Function Placement in Serverless Edge Environments,” in Proc. ACM/IEEE Symposium on Edge Computing (SEC), San Jose, CA, USA, 2022, pp. 115–127.
[8] M. Shahrad et al., “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider,” in Proc. USENIX Annual Technical Conference (ATC), Boston, MA, USA, 2020, pp. 205–218.
[9] L. Baresi and D. Mendonça, “Towards a Serverless Platform for Edge Computing,” in Proc. IEEE International Conference on Cloud Engineering (IC2E), Prague, Czech Republic, 2019, pp. 9–15.
[10] P. Leitner et al., “A Survey on the State of Serverless Computing,” ACM Computing Surveys, vol. 55, no. 3, pp. 1–37, May 2023. doi: 10.1145/3502265
.[11] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey,” ACM Computing Surveys, vol. 51, no. 4, pp. 1–33, Sep. 2018.
[12] OpenFaaS Documentation, “OpenFaaS – Serverless Functions Made Simple,” [Online]. Available: https://www.openfaas.com. [Accessed: 13-Oct-2025].
[13] R. Grandl, D. Akhmetova, A. Panda, and S. Shenker, “Scalable Autoscaling for Microservices,” in Proc. 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Santa Clara, CA, USA, 2020, pp. 783–798.
[14] A. A. Tasiopoulos and M. D. Dikaiakos, “Adaptive Event-Driven Resource Management for Serverless Cloud Computing,” in Proc. IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom), Nicosia, Cyprus, 2019, pp. 133–142.
[15] S. Eismann et al., “Predicting Serverless Function Resource Utilization with Machine Learning,” in Proc. 21st Intl. Middleware Conference, Delft, Netherlands, 2020, pp. 64–77.
[16] S. Hendrickson et al., “Serverless Computation with OpenLambda,” in Proc. 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Denver, CO, USA, 2016, pp. 1–7.
[17] M. Malawski, “Towards Serverless Computing: Applications and Research Perspectives,” in Proc. 6th Intl. Conference on Cloud Computing and Services Science (CLOSER), Rome, Italy, 2016, pp. 1–10.
[18] T. Lynn et al., “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms,” in Proc. IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom), Luxembourg, 2017, pp. 162–169.
[19] J. Bortoli and F. Montesi, “Toward a Common Model for Serverless Computing,” IEEE Software, vol. 37, no. 1, pp. 36–43, Jan.–Feb. 2020.
[20] J. Singh and C. Mendis, “Efficient Resource Allocation for Event-Driven Applications in Serverless Cloud,” in Proc. IEEE Intl. Conf. on High Performance Computing & Simulation (HPCS), Dublin, Ireland, 2021, pp. 520–527.
[21] S. Basu and A. K. Saha, “Performance Optimization in Serverless Computing using RL,” in Proc. IEEE Intl. Conf. on Big Data (BigData), Los Angeles, CA, USA, 2022, pp. 4321–4328.
[22] B. Varghese and R. Buyya, “Next Generation Cloud Computing: New Trends and Research Directions,” Future Generation Computer Systems, vol. 79, pp. 849–861, Feb. 2018.
[23] T. F. Düllmann et al., “Serverless Workflows for Scientific Computing,” in Proc. IEEE Intl. Conf. on eScience, San Diego, CA, USA, 2019, pp. 585–590.
[24] S. Wang et al., “Enabling Function-Level Performance Guarantees in FaaS,” in Proc. 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2021, pp. 643–659.
[25] M. K. Mohanty and S. Majhi, “Intelligent Serverless Resource Management using Deep Reinforcement Learning,” Journal of Cloud Computing, vol. 12, no. 4, pp. 1–15, 2023.