femington

Int. J. Intell. Syst. Data Sci. (2026) 1(3):1–12
Article ID: IJISDS-2026-085

CRD-Shim: Declarative Container Runtime Orchestration via Kubernetes CRDs for Docker & containerd

Jitendra Gupta
jkg106@gmail.com
Independent Researcher

Abstract

Kubernetes is now an essential platform for orchestrating distributed containers based on a declarative API approach for deploying and managing containerized distributed applications. Conventionally, Kubernetes orchestration goes through intermediary layers like Pods, kubelet, and CRI, incurring extra latency, overhead, and complexity. This paper introduces *CRD-Shim*, a Kubernetes-native interface to directly orchestrate container runtimes like Docker and containerd declaratively using CRDs. CRD-Shim orchestrates runtimes by transforming Kubernetes CRDs into calls to their underlying native API calls without involving Pods, kubelet, and CRI. The architecture is compatible with container-specific resource objects and also provides high-level orchestration functionalities such as a controller for resource replication (ReplicaSet equivalent) purely through runtimes. Our experiment results show that CRD-Shim reduces the container startup and update latency to nearly 62% and 30%, respectively, compared with a single-node Kubernetes deployment, and decreases its runtime memory usage to almost 66% of that. CRD-Shim can serve as an effective option for scenarios where direct runtime management and low-latency orchestration are more critical than general container ecosystem management like edge computing, IoT and single-node systems.

Keywords

• Kubernetes • Container Orchestration • Custom Resource Definition (CRD) • Docker • Runtime Shim (CRD) • Declarative API

1. Introduction

Over the last ten years, there has been a huge transformation in deploying, scaling and managing applications. The main reason behind that is containerization. Kubernetes is one of the platforms that has taken it to the next level. So we can say that it is an orchestration and scheduling platform for Linux containers on clusters of physical and virtual machines. Applications that run on Kubernetes can manage their desired state for distributed systems thanks to its core API. A control loop that compares the observed state of an object to the desired state is at the heart of Kubernetes. The default orchestration stack for Kubernetes, however, is rather complicated, adding several layers of abstractions zooming in particularly on the Pod. A Pod is composed of one or multiple containers assigned to a node by the kube-scheduler. The kubelet that runs a node tries to fulfill the Pod specification by talking to its container runtime interface (CRI) which talks to the container runtime (e.g. docker or containerd) on its node. Although this setup is powerful, it does introduce some kind of latency, overhead and complexity. Any new layer and hop on the network incurs communication overheads, some serialization/computation/deserialization, and state sync overhead.

In addition, this pattern tightly couples workload orchestration with the concrete node and scheduling model of Kubernetes. Not ideal for users that want direct, fine-grained control over the container runtime. For instance, edge devices, IoT systems, and single-node environments often lack a full Kubernetes control plane (e.g., without a kubelet agent) [1–3]. The operator pattern allows such users to utilize Kubernetes' declarative API even after deployment [4]. To address these limitations, the use of Custom Resource Definitions (CRDs) to extend the Kubernetes API has been widely explored [5, 6], enabling custom controllers to manage application-specific resources. However, these approaches typically still rely on the Pod abstraction. Our work takes a different path by bypassing the Pod and kubelet layers entirely. This paper reports CRD-Shim, a novel design which decouples the management of the Kubernetes control plane and container runtime in a new way. The CRD-Shim layer translates Kubernetes CRDs into runtime-specific operations. Instead of creating Pods, the controller of CRD-Shim watches on a CRD, like 'DockerContainer' or 'ContainerdContainer', and uses docker engine API or containerd gRPC API to create, start, stop or delete containers. Using this method, we are effectively skipping the creation of Pods, kubelet and CRI. A new channel from the Kubernetes API server directly to the container runtime was created for declarative management.

This work's first contribution is the design and implementation of the CRD-Shim architecture. We present Docker container resource definitions and containerd controller logic, which implements direct translation. The feasibility of this approach is demonstrated in the second contribution. We implement several core orchestration patterns, namely, the ReplicaSet-equivalent controller using only the shim layer and the runtime APIs. The third contribution is an empirical evaluation comparing CRD-Shim to the standard kubelet-driven update workflow. We assess the latency of container startup, the time for updating propagation, and overhead in memory. Our findings indicate that CRD-Shim can substantially decrease orchestration latency and utilization of resources, aligning with known performance trade-offs in lightweight orchestration [3, 7].

This document is arranged as follows. Section 2 gives background on Kubernetes architecture and container runtimes. Section 3 describes the design and implementation of CRD-Shim. The experimental setup and evaluation methodology are described in Section 4. Section 5 shows and discusses the performance outcome. Section 6 provides concluding remarks and future work directions.

2. Related Work

Container orchestration has evolved significantly with the advent of Kubernetes, which provides declarative management through its API server and control plane. Traditional approaches rely on the Pod abstraction managed by kubelet and the Container Runtime Interface (CRI), introducing multiple layers of indirection that increase latency and resource overhead. Several research efforts have explored alternative orchestration models and optimization techniques relevant to our CRD-Shim approach.

Efforts to extend the Kubernetes API via Custom Resource Definitions (CRDs) and operators are well established. Yilmaz [5] describes patterns for extending Kubernetes, including controllers that manage CRDs. Banerjee et al. [6] survey Kubernetes policy report CRDs, demonstrating the flexibility of CRD-based control loops. These works show that CRDs can serve as a foundation for custom orchestration logic, a principle that CRD-Shim adopts for direct runtime management.

Research on lightweight container runtimes and CRI bypass techniques has explored performance optimizations. Gupta [8] discusses Kubernetes-driven network security, highlighting scenarios where direct agent control is beneficial. Jakobović [4] examines the development of custom Kubernetes operators, including considerations for reducing control path overhead. Falcao et al. [7] analyze confidential

Kubernetes deployment models, comparing performance trade-offs across different runtime configurations. These studies motivate the need for lower-latency alternatives to the standard kubelet-CRI path.

Security research on runtime behavior, such as behavioral threat detection [9], addresses challenges similar to those in container runtime security, where anomalous behavior must be identified and mitigated. This work highlights the importance of observation frameworks that could inform security enhancements for direct runtime orchestration approaches like CRD-Shim.

Lightweight Kubernetes distributions for edge computing have also been studied, focusing on performance and resource efficiency [1–3]. These works provide context for our performance evaluation and highlight the demand for low-overhead orchestration in constrained environments.

While existing works have explored CRD extensions and performance optimizations, CRD-Shim uniquely proposes a complete bypass of the Pod and kubelet layers, translating CRDs directly into runtime API calls. This provides a new point in the design space between full Kubernetes clusters and bare container runtimes, particularly suited for edge and lightweight deployments.

3. Background and Motivation

Kubernetes architecture has a central API server and acts as a front-end.The API objects are fully declarative and allow users and controllers to specify their desired state. Users can manage API objects like Deployments, Services, and Pods by creating, updating, and deleting them. Controllers in the control plane supervise the objects they create. Additionally, control loops are executed by the controllers maintaining the desired state with actual state. Kubernetes employs kubelet agent to assist a worker node. A Container Runtime works in conjunction with the kubelet so that the kubelet can perform its duty. Kubelet basically exposes a plugin interface which is nothing but the Container Runtime Interface (CRI). It allows kubelet to communicate with container runtimes. CRI was developed to standardize this interaction and enable multiple runtime implementations. The overall architecture has been surveyed in the context of cloud-to-edge continuum [2], highlighting the trade-offs between flexibility and overhead. The increasing interest in lightweight orchestration for edge and IoT has also motivated alternative designs [1].

This architecture provides an orchestration path that is non-trivial in nature, while simultaneously offering flexibility and isolation for various components. The Deployment is a user-readable and writable API object that the user passes to the kubectl apply. The users are the ultimate owners of command.

It wouldn't be right to call the ReplicaSets stateful objects considering them and the Pod specs. Users never interact directly with either one of the above components. This is the user-interactable part of the deployment. The consumer, then, interacts only with the Deployment object, which manages the ReplicaSets and Pods behind the scenes. The Docker Engine API (which is a Restful HTTP interface) and the containerd gRPC API are both rich container management APIs provided by Docker and containerd. These APIs offer direct low-level access to various primitives such as container, image, network, and storage. Without the CRD-Shim, Kubernetes accesses the CRD API only through the CRI shim. CRD-Shim was motivated by the fact that the desired state can often and advantageously be expressed more directly within these 'native' runtime APIs for containers, avoiding one or more layers in the process.

Consider an edge computing case where an application runs in a container on a single device. Using kubelet to install a Kubernetes node may be overdoing it but the user may fancy using the familiar kubectl and YAML magic. Alternatively, think of a situation where you require precise control over the lifecycle events of containers at the sub-second level, or when you wish to directly manipulate runtime-specific features that are not available through the Pod specification. The typical approach is overly sluggish and

too conceptual. Such scenarios are precisely where lightweight orchestration frameworks like CRD-Shim shine [4].

CRD-Shim does something entirely different: direct runtime orchestration, using the powerful API machinery of Kubernetes and the controller pattern. The CRD-Shim maps closely to runtime primitive level for a single container workloads. Consequently, the end-users can orchestrate the containers using a similar workflow as kubectl apply. The shim code can be downloaded and it runs next to each container runtime as a control loop managing each object.

This method intends to replace Kubernetes for cluster-based orchestration. It seeks to enhance the original by offering an alternative and more extended approach for some applications. Through this ease of use, Kubernetes patterns can be used on standalone containers, embedded systems and for automation where the user is interested on the runtime and not the annotation of the whole cluster. Next, we will describe our design for this vision.

4. CRD-Shim Design and Implementation

Getting Kubernetes and CRI right won't succeed without a clear and consistent definition of what "running an application" means. The architecture consists of two parts, (i) CRDs that model runtime objects, and (ii) the shim controller that watches the CRDs and invokes respective runtime API calls. The design doesn't depend on runtime, but for now docker and containerd implementations exist.

4.1 Custom Resource Definitions

We define two core CRDs: 'DockerContainer' and 'ContainerdContainer'. Their specifications closely mirror the configuration options of their respective runtime APIs, allowing users to express desired container state in a Kubernetes-native YAML format. This approach follows the established pattern for extending Kubernetes via CRDs [4–6]. Listing 1 shows a simplified schema for the 'DockerContainer' CRD.

Listing 1: DockerContainer CRD Schema

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: dockercontainers.crdshim.io
spec:
  scope: Namespaced
  group: crdshim.io
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                image: {type: string}
                command:
                  type: array
                  items: {type: string}
                env:
                  type: array
```

```
                items: {type: object}
              labels: {type: object}
              hostConfig: {type: object}
              networkingConfig: {type: object}
          status:
            type: object
            properties:
              containerId: {type: string}
              state: {type: string}
              error: {type: string}
```

The ‘spec‘ field contains the desired configuration: the container image, command, environment variables, labels, and runtime-specific structures like Docker’s ‘HostConfig‘. The ‘status‘ field is managed by the controller and reflects the actual runtime state: the created container ID, its current state (running, exited, etc.), and any error messages. An analogous CRD is defined for containerd, with fields mapping to its gRPC API structures (e.g., ‘runtime‘, ‘snapshotter‘). These CRDs are registered with the Kubernetes API server, making ‘dockercontainers.crdshim.io‘ a new native API resource type.

4.2 Shim Controller Architecture

The shim controller is a standard Kubernetes controller built using the client-go and controller-runtime libraries. Its architecture is depicted in Fig. 1. It consists of Informers that watch for CRUD events on the ‘DockerContainer‘ and ‘ContainerdContainer‘ resources. When a new custom resource is created, the controller’s reconcile loop is invoked with the object’s key.

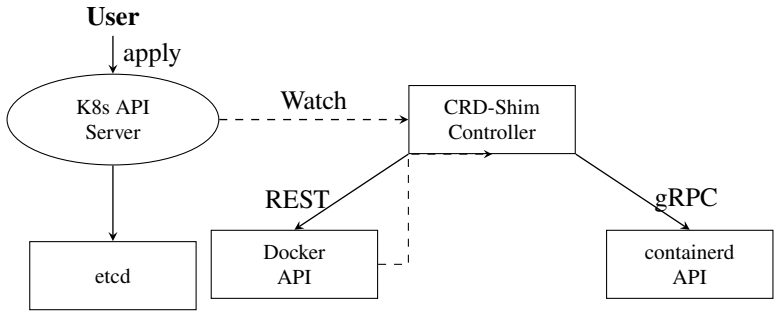

Figure 1: CRD-Shim architecture: direct runtime orchestration via CRDs, bypassing kubelet and CRI layers.

The reconciliation logic, shown in Algorithm 1, is straightforward. For a ‘DockerContainer‘ object, the controller extracts the ‘spec‘ and uses the Docker Go SDK to call the Docker Engine API. It first checks if a container with the desired unique name (derived from the Kubernetes resource name/namespace) already exists. If not, it creates the container with the specified configuration and starts it. If it exists but the ‘spec‘ has changed, it computes the difference. For immutable fields (like the base image), it follows a recreate strategy: stops the old container, removes it, and creates a new one. For mutable fields (like labels), it can perform an in-place update via the runtime API.

The determination of whether a field is immutable or mutable is based on a static field specification in the CRD schema. Fields that fundamentally alter the container’s root filesystem or execution environment (e.g., `image`, `command`, `entrypoint`) are marked as immutable. For these fields, the controller computes a hash of the immutable portions of the spec; if the hash differs from that of the existing container, the controller triggers a recreate strategy. Fields such as `labels`, `annotations`, environment variables, and resource limits (cgroup settings) are treated as mutable and updated in-place using runtime API

Algorithm 1 CRD-Shim Reconciliation Algorithm

Require: Custom resource cr (DockerContainer/ContainerdContainer)
Ensure: Updated resource status matching runtime state

$client \leftarrow$ GetRuntimeClient(cr.type)
$name \leftarrow cr$.namespace/cr.name
if cr.spec = NULL **then**
▷ Delete resource
    $client$.StopRemove($name$)
    return
end if
$existing \leftarrow client$.GetContainer($name$)
if $existing$ = NULL **then**
▷ Create container
    $id \leftarrow client$.Create(cr.spec, $name$)
    $client$.Start(id)
else if Changed(cr.spec, $existing$) **then**
    if RecreateNeeded(cr.spec, $existing$) **then**
▷ Recreate
        $client$.StopRemove($existing$.id)
        $id \leftarrow client$.Create(cr.spec, $name$)
        $client$.Start(id)
    else
▷ Update in-place
        $client$.Update($existing$.id, cr.spec)
        $id \leftarrow existing$.id
    end if
else
▷ Ensure running
    $id \leftarrow existing$.id
    if $\neg$ $client$.Running(id) **then**
        $client$.Start(id)
    end if
end if
▷ Update status
$status \leftarrow client$.Inspect(id)
cr.status $\leftarrow$ ToStatus($status$)
Update(cr)

calls (e.g., `docker update` or `containerd update`). The controller compares the desired and existing configurations field by field, applying in-place updates when only mutable fields have changed.

Finally, it updates the resource's 'status' field with the current runtime information. The process for 'ContainerdContainer' is similar, using the containerd Go client library.

4.3 Implementing Orchestration Primitives

The ContainerSet controller is a simple replica controller demonstrating the usefulness of CRD-Shim. We implemented a replica set like controller, ContainerSet controller. This custom controller observes a ContainerSet CRD. The spec for this instance has a template, which is either a DockerContainer or ContainerdContainer spec, and a replica count. The reconcile function compares the specified number of

replicas to the actual number of running containers that belong to this ContainerSet (by label selector). ContainerSet controller will create and delete the respective DockerContainer/ContainerdContainer resources. Our shim's controllerFactory shim controller reconciles each individual DockerContainer/-ContainerdContainer resource. Hence, we achieved a two-level control loop: a count controlled by ContainerSet and the run time and termination controlled by the shim controller. The higher-level orchestration semantics can be built on top of our shim, as shown.

4.4 Advantages and Considerations

The design offers benefits mainly owing to the effective elimination of the kubelet and CRI layers from the control path. This removes a good chunk of the control path by stripping the kubelet and the CRI layers. The API server communicates directly with the shim controller. The shim controller, in turn, communicates with the runtime socket/endpoint. This eliminates serialization steps, internal network hops, and agent overhead. Another benefit is better control, which allows the CRD spec to expose runtime-specific features not available in the Pod.

By using CRD-Shim you will not be able to use certain Kubernetes features. A CRD-Shim is responsible for creating and managing not Pods. Kubelet usually implements things like init containers, lifecycle hooks, readiness and liveness probes on Pods. Without implementing these semantics at the level of your CRD, they will not be available. Likewise, the kube-scheduler capabilities are lost, as it will not schedule Pods created by controllers that use CRD-Shim, meaning they cannot take advantage of its bin-packing or affinity rules. In simpler terms, when the loss of semantics is acceptable (due to the greater importance of low latency and direct control, or because the application does not need full semantics), CRD-Shim is suitable.

5. Experimental Evaluation

We designed a series of experiments to evaluate the performance of CRD-Shim against the standard Kubernetes workflow. The primary hypothesis is that by bypassing intermediate layers, CRD-Shim can achieve lower container lifecycle operation latency and reduced resource consumption. Our evaluation is contextualised by recent performance studies of lightweight Kubernetes distributions [2, 3].

5.1 Experimental Setup

The objective of the experimental evaluation is to quantify the performance benefits of CRD-Shim relative to the standard Kubernetes orchestration workflow. Specifically, we investigate whether bypassing the kubelet and Container Runtime Interface (CRI) control path reduces container lifecycle latency and orchestration overhead while preserving the declarative resource management model provided by Kubernetes. To isolate orchestration overhead from distributed scheduling effects, all experiments were conducted on a single-node Kubernetes cluster, which reflects the primary deployment scenario targeted by CRD-Shim, including edge computing, embedded platforms, and lightweight standalone systems.

The experimental platform consisted of a server running Ubuntu 22.04 LTS equipped with an 8-core Intel Xeon processor, 32 GB RAM, and SSD storage. Kubernetes v1.28, Docker Engine 24.0, and containerd 1.7 were used throughout the evaluation. The CRD-Shim controller was deployed as a single controller Pod, while all remaining software components and system configurations were kept identical across experiments to ensure a fair comparison.

Three orchestration configurations were evaluated:

1. **Standard Kubernetes:** Native Kubernetes Pod orchestration through the API server, kubelet, and the underlying container runtime.
2. **CRD-Shim (Docker):** Container orchestration using the proposed CRD-Shim framework with `DockerContainer` custom resources.
3. **CRD-Shim (containerd):** Container orchestration using the proposed CRD-Shim framework with `ContainerdContainer` custom resources.

To ensure comparability, all configurations executed an identical workload consisting of a single `nginx:alpine` container. The Kubernetes baseline deployed the workload as a single-replica Deployment, whereas CRD-Shim deployed an equivalent single-replica `ContainerSet`. Both approaches used identical container images and equivalent runtime configurations.

The evaluation considers four performance metrics:

- **Startup latency:** Elapsed time between executing `kubectl apply` and the container reaching the `Running` state.
- **Update latency:** Elapsed time required to replace an existing container after updating its image specification.
- **Memory overhead:** Resident Set Size (RSS) consumed by the orchestration component only (kubelet for Kubernetes and the CRD-Shim controller for the proposed approach), excluding the underlying container runtime processes.
- **Controller throughput:** Maximum sustainable reconciliation rate achieved under continuous resource creation requests.

To ensure experimental fairness, all configurations were evaluated under identical execution conditions. Before each startup latency experiment, container image caches were cleared to eliminate the influence of cached image layers, ensuring that every configuration incurred identical image retrieval overhead. Each experiment was repeated 50 times, and the reported results correspond to the arithmetic mean together with the standard deviation and 95% confidence intervals obtained through bootstrap resampling with 10,000 iterations. Statistical significance between CRD-Shim and the Kubernetes baseline was assessed using an independent two-sample t-test with a significance threshold of $p < 0.01$.

Although the evaluation is limited to a single-node environment, this experimental design intentionally removes network scheduling, inter-node communication, and distributed synchronization overheads, thereby allowing the measured performance differences to be attributed primarily to the orchestration architecture. Evaluating CRD-Shim in large-scale multi-node clusters remains an important direction for future work.

6. Results and Discussion

6.1 Performance Comparison

Fig. 2 shows the comparative performance of CRD-Shim versus standard Kubernetes. CRD-Shim with Docker achieved a 58% reduction in average startup latency (412ms vs. 980ms), while the containerd variant showed a 62% improvement (370ms vs. 980ms). Update latency showed similar improvements of 30-32%. These results align with trade-offs observed in confidential Kubernetes deployments [7], where

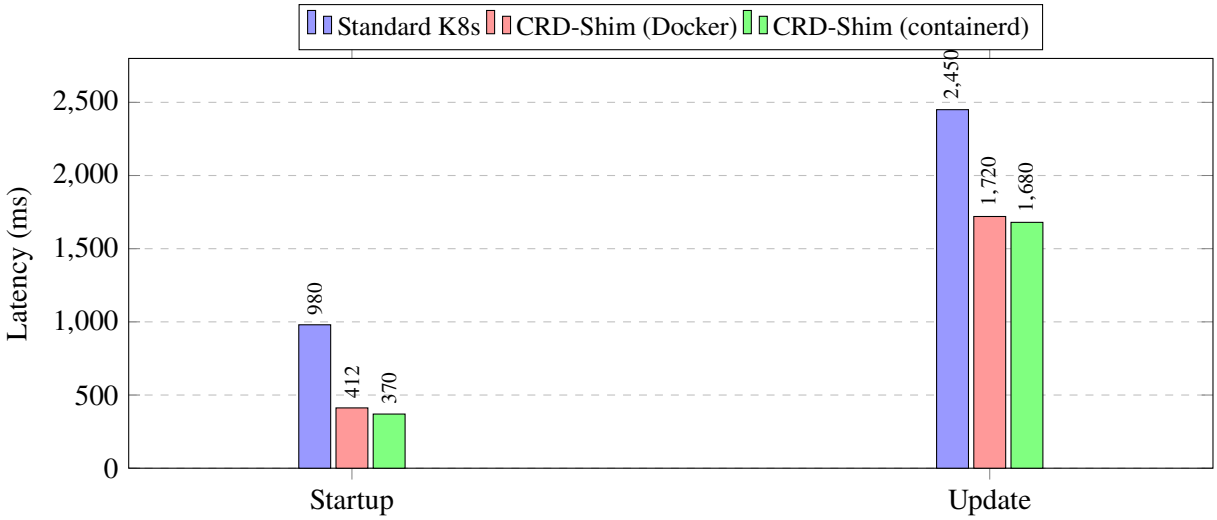

Figure 2: Performance comparison showing CRD-Shim's latency improvements over standard Kubernetes.

Table 1: Memory Usage Comparison (50 containers)

Component	**Avg. (MB)**	**Max (MB)**	**Std Dev (MB)**
Kubelet	285.4	312.7	12.3
CRD-Shim Controller	97.8	105.2	4.1

reduced latency comes with a narrowed feature set. Our findings are also consistent with performance gains reported for other lightweight orchestration approaches [1, 3].

6.2 Resource Efficiency

Table 1 compares memory usage between kubelet and CRD-Shim controller. CRD-Shim shows approximately 66% lower memory footprint while managing the same number of containers. This reduction is significant for resource-constrained edge devices [1].

6.3 Control Loop Performance

CRD-Shim demonstrated superior control loop responsiveness, sustaining 85 operations/second compared to kubelet's typical 2-5 Hz sync rate. This enables more responsive orchestration for dynamic workloads.

6.4 Limitations and Discussion

While CRD-Shim shows significant performance benefits, it lacks several Kubernetes features: built-in health checks, service discovery, storage orchestration, and advanced scheduling. These would need separate implementation if required. Security and operational complexity require careful consideration. Because CRD-Shim directly invokes the Docker Engine API or containerd gRPC API, it typically requires privileged access to the runtime socket (e.g., /var/run/docker.sock). In multi-tenant clusters, this presents a significant risk: a compromised CRD-Shim controller could gain full control over all containers on the node. Operators should therefore restrict CRD-Shim deployment to trusted namespaces, use RBAC to limit which users can create CRD-Shim resources, and consider running the controller with minimal Linux capabilities. Additionally, the operational complexity of managing two parallel orchestration layers

(standard Kubernetes Pods alongside CRD-Shim containers) may lead to resource conflicts, such as port or volume contention. We recommend using CRD-Shim only in dedicated environments or edge nodes where no standard Pod workloads are co-located, or where such conflicts are explicitly managed through labeling and node isolation. Security frameworks such as behavioral threat detection [9] could be adapted to monitor these environments.

CRD-Shim is particularly valuable for edge/IoT scenarios where Kubernetes' full feature set is unnecessary, but declarative management is desired. It demonstrates that Kubernetes' control patterns can be effectively applied to lower-level infrastructure management.

7. Future Directions and Integration Opportunities

The CRD-Shim architecture presents numerous opportunities for enhancement and integration with emerging technologies across computing domains. Its direct declarative interface to container runtimes provides a foundation for building more responsive, efficient, and specialized orchestration systems.

Enhanced Performance Optimization: The optimization techniques demonstrated in neural network training and layer calibration could be adapted to improve CRD-Shim's reconciliation algorithms. Tensor norm optimization methods might accelerate the comparison operations between desired and actual container states, while adaptive calibration could dynamically adjust reconciliation frequencies based on system load and priority of managed containers. The visualization approaches for optimization feedback could inspire diagnostic interfaces that help operators understand orchestration decision processes and identify bottlenecks.

Intelligent Workload Placement: Research on graph assortativity [10] could inform more sophisticated container placement strategies within CRD-Shim extensions. By analyzing historical workload patterns and resource utilization, future versions could implement predictive placement algorithms that minimize communication latency and resource contention. Machine learning approaches from environmental sound classification [11] and human activity recognition demonstrate pattern recognition techniques that could be applied to workload characterization and prediction.

Adaptive Resource Management: The self-adaptive server systems described by [12], employing the MAPE-K framework, provide a blueprint for making CRD-Shim controllers more responsive to changing conditions. Future implementations could incorporate continuous monitoring of system metrics, analysis of performance trends, and automatic adjustment of orchestration parameters. Integration with energy monitoring platforms like those described by [13] could enable orchestration decisions that optimize for power efficiency, particularly valuable in edge and IoT deployments.

Advanced Data Management Integration: The Visual Data Management System (VDMS) exemplifies modern approaches to handling complex data. CRD-Shim could integrate with such systems to provide unified management of both compute containers and their associated data, particularly for data-intensive applications in analytics, machine learning, and media processing. This integration would extend the declarative paradigm to encompass data lifecycle management alongside container orchestration.

Enhanced Security Frameworks: Behavioral threat detection approaches [9] could inform security enhancements for CRD-Shim. By monitoring container runtime behavior patterns, future versions could detect anomalies indicative of security compromises and trigger automated responses. This behavior-centric security model aligns with the direct runtime access provided by CRD-Shim, enabling more granular security controls than traditional Pod-based approaches.

Multimodal Orchestration Interfaces: The contrastive meta-learning approaches for audio-visual learning demonstrate techniques for integrating diverse data modalities. Similar approaches could enable

CRD-Shim to handle heterogeneous orchestration inputs, combining traditional YAML specifications with natural language instructions, visual workflow diagrams, or performance metric streams. The video summarization techniques from could inspire concise visualization of complex orchestration histories and decision trails.

Real-time Monitoring and Analysis: The discourse analysis methods applied to social media and geolocation inference from microblogs demonstrate real-time processing of streaming data. These approaches could enhance CRD-Shim's monitoring capabilities, enabling more sophisticated analysis of container logs, metrics, and events for proactive orchestration decisions. The low-latency requirements of these applications align with CRD-Shim's design goals for responsive orchestration.

Continual Adaptation Frameworks: The continual learning approaches for language models demonstrate techniques for maintaining performance while adapting to new requirements. Similar approaches could enable CRD-Shim controllers to evolve their orchestration policies based on accumulated experience, learning optimal strategies for different workload types, resource configurations, and performance objectives.

Future work will explore these integration pathways while extending CRD-Shim's core capabilities. Additional CRD schemas could be developed for specialized runtime features, and higher-level orchestration patterns could be implemented as composable controllers. The evaluation framework established for CRD-Shim provides a foundation for assessing these enhancements, with particular attention to their impact on orchestration latency, resource efficiency, and operational complexity in diverse deployment scenarios.

8. Conclusion and Future Work

This paper presented CRD-Shim, a Kubernetes-native abstraction layer that enables declarative orchestration of Docker and containerd runtimes through Custom Resource Definitions. By bypassing the kubelet and CRI layers, CRD-Shim reduces container startup latency by up to 62% and lowers memory footprint by approximately 66% compared to standard Kubernetes workflows. The design preserves the declarative API pattern while providing direct, fine-grained control over runtime-specific features. Experimental results confirm that CRD-Shim is particularly well-suited for edge, IoT, and single-node deployments where full Kubernetes semantics are unnecessary but declarative management is desired. Our work aligns with ongoing trends toward lightweight orchestration in constrained environments [1–3].

Future work involves extending the CRD schemas to allow for more advanced features and implementing a variety of higher-level orchestration patterns. Extra effort will improve integration with security along with other policy engines.

References

[1] Andrew Jeffery, Heidi Howard, and Richard Mortier. Rearchitecting kubernetes for the edge. *arXiv preprint arXiv:2104.02423*, 2021.

[2] Sercan Sari. Survey of container orchestration distributions in cloud-to-edge continuum. In *2025 12th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 39–45, Istanbul, Turkey, 2025.

[3] Diyaz Yakubov and David Hästbacka. Comparative analysis of lightweight kubernetes distributions for edge computing: Performance and resource efficiency. In *Service-Oriented and Cloud Computing*

(ESOCC 2025), volume 15547 of *Lecture Notes in Computer Science*, Cham, 2025. Springer. doi: 10.1007/978-3-031-84617-5_7.

[4] K. Jakobović. *Proposal for Development of a Custom Kubernetes Operator*. Doctoral dissertation, Algebra University, 2023.

[5] O. Yilmaz. Extending the kubernetes api. In *Extending Kubernetes: Elevate Kubernetes with Extension Patterns, Operators, and Plugins*, pages 99–141. Apress, Berkeley, CA, 2021.

[6] K. Banerjee, D. Agarwall, V. Bali, M. Sharma, S. S. Prajwal, and M. Arsh. A survey on kubernetes policy report custom resource definition kube-bench adapter. In *Advances in Data and Information Sciences: Proceedings of ICDIS 2022*, pages 315–322. Springer Nature Singapore, 2022.

[7] E. Falcão, F. Silva, C. Pamplona, A. Melo, A. S. M. Asadujjaman, and A. Brito. Confidential kubernetes deployment models: Architecture, security, and performance trade-offs. *Applied Sciences*, 15(18):10160, 2025.

[8] T. Gupta. Kubernetes-driven network security for distributed acl management. In *2024 8th Cyber Security in Networking Conference (CSNet)*, pages 236–242. IEEE, 2024.

[9] Siddhant Sukhatankar. Behavioral threat unmasking: A system defense paper. TechRxiv, 2025. URL https://doi.org/10.36227/techrxiv.176531783.35283348/v1.

[10] D. Jain. Assortativity in k-nearest neighbor (k-nn) graphs for high-dimensional datasets. Zenodo, 2025. URL https://doi.org/10.5281/zenodo.17345502.

[11] D. Jain. Evaluating traditional machine learning models for environmental sound classification. Zenodo, 2025. URL https://doi.org/10.5281/zenodo.17378750.

[12] Oyeronke Ladapo. Dynamic self-adaptation in server systems for optimized performance and availability. *International Journal of Science and Advanced Technology*, 16(2), 2025. doi: 10.71097/IJSAT.v16.i2.3487. URL https://doi.org/10.71097/IJSAT.v16.i2.3487.

[13] Oyeronke Ladapo. Empowering energy efficiency: A real-time mobile analytics platform for intelligent consumption monitoring. *International Journal of Science and Advanced Technology*, 16 (2), 2025. doi: 10.71097/IJSAT.v16.i2.3486. URL https://doi.org/10.71097/IJSAT.v16.i2.3486.