PRISTINE: PRIority-aware Smart resource orchestraTIon eNginE for Cloud-Native Applications

Prashanth Josyula
Principal Member of Technical Staff
Salesforce
San Francisco, USA
prashanth.16@gmail.com

Anant Kumar
Lead Member of Technical Staff
Salesforce
San Francisco, USA
garg.anant@gmail.com

Gangadharayya Hiremath
Lead Member of Technical Staff
Salesforce
Dallas, USA
gangavh@gmail.com

***Abstract*—In modern cloud-native environments, effective resource management across multiple applications continues to be a significant challenge. Although Kubernetes offers basic resource management primitives, there exists a critical missing piece at the application level for resource orchestration, particularly in environments where applications entail varying priorities and resource requirements. In this paper, we present PRISTINE, a novel priority-aware resource orchestration engine implemented as a Kubernetes Custom Resource Definition (CRD). PRISTINE extends Kubernetes' native capabilities with application-level resource management, which include built-in priority awareness, dynamic resource allocation, and sophisticated preemption strategies, our solution addresses the fundamental problems of resource fragmentation, priority-based allocation, and operational complexity in cloud-native environments, with its innovative architecture, PRISTINE offers an effective platform for managing multiple workload resources with a guaranteed resource distribution based on application priorities and business requirements.**

***Index Terms*—Kubernetes, Resource Management, Cloud-Native, Priority Scheduling, Custom Resource Definition, Container Orchestration**

I. Introduction

The adoption of cloud-native architectures has led to increasingly complex application deployments, with multiple microservices running different types of workload within the Kubernetes clusters, although Kubernetes excels at container orchestration, its native resource management is primarily centered around single containers and namespaces without end-to-end application-level resource orchestration, this limitation is particularly evident in enterprise environments, where several applications compete for limited resources, and resource allocation decisions must be dependent on business requirements.

Current Kubernetes resource management faces several critical challenges:

First, fragmentation of resources across various types of workloads (Deployments, StatefulSets, DaemonSets, etc.) makes it difficult to maintain a holistic, global view of application resource requirements, second, existing solutions do not provide application-level resource guarantees, which could lead to service outages in the event that resources become limited. Third, although Kubernetes has rudimentary pod-level prioritization and preemption, the same methods do not translate adequately to application-level priorities and resource guarantees. Lastly, platform teams experience high operational overhead in managing resources for a large number of applications and workloads, and without proper abstractions such management becomes more chaotic and unmanageable.

This paper presents a new engine, PRISTINE, a novel solution that addresses these challenges through different components.

- A priority-aware Custom Resource Definition that provides application-level resource management
- An intelligent resource orchestration engine that considers application priorities and business requirements
- A sophisticated preemption strategy that minimizes disruption while ensuring resource availability for high-priority applications
- An efficient caching and state management system that reduces operational overhead

The remainder of this paper is structured as follows: Section II provides background information and discusses related work. Section III presents the PRISTINE architecture and the core components. Section IV addresses the implementation aspects. Section V focuses on advanced features and optimizations and, lastly, Section VI concludes the paper and discusses future changes.

II. Background and Related Work

This section examines the evolution of container orchestration technologies, contemporary methods to resource management in Kubernetes, and recent advances in priority-driven systems. We will analyze the current industry landscape and identify persistent challenges that had motivated development of PRISTINE.

A. Evolution of Container Orchestration

The rise of containerization has significantly transformed the way applications are managed and deployed in the cloud. Kubernetes has since become defacto container orchestration technology after earlier systems like Borg [1] and Omega [2], thus creating the foundational patterns for modern container orchestration, including declarative configuration, reconciliation loops, and multilevel resource abstraction.

B. Kubernetes Resource Management

Kubernetes implements resource management via a hierarchical model that spans from individual containers to cluster-wide policies. Burns et al. [3] introduced the primary design patterns that influence container-based distributed systems, many of which are fundamental to the Kubernetes resource management approach.

1) Resource Management Primitives: The basic building blocks of Kubernetes resource management include:

1) Resource Requests and Limits
2) Quality of Service (QoS) Classes
3) Resource Quotas and Limit Ranges

Shan et al. [4] in their study have shown that the proper configuration of these primitives has a considerable impact on application performance and resource utilization. The study demonstrated that optimizing requests and limits configuration can increase resource efficiency by up to 35%.

C. Advanced Resource Management Approaches

1) Machine Learning-Based Solutions: Advances in machine learning recently have made more sophisticated methods of resource management possible. By using, a deep reinforcement learning framework for dynamic resource allocation in Kubernetes clusters, Mampage et al. [5] achieved 27% better resource utilization than conventional methods, likewise, Munjal et al. [6] created an adaptive resource scheduling system predicting future resource needs based on historical workload patterns.

2) Multi-Cluster Management: As enterprises deploy applications across many clusters, resource management becomes increasingly complex. Faticanti et al. [8] proposed a new federation strategy that allows for seamless resource sharing across clusters while retaining application-specific SLOs. Their study expands on the federation patterns described by Salidas et al. [7].

D. Priority-Based Resource Management

Priority-based resource management has gained significant attention in recent years. Shelar et al. [9] presented a comprehensive framework for priority-aware resource allocation in microservices environments. Their approach demonstrated significant improvements in meeting service-level objectives for high-priority workloads while maintaining fair resource distribution for lower-priority applications.

E. Resource Optimization Techniques

1) Autoscaling Mechanisms: Modern autoscaling mechanisms have evolved beyond simple threshold-based approaches. Zhao et al. [10] developed a predictive autoscaling system that combines time-series analysis with machine learning to anticipate resource requirements. Their system showed a 40% reduction in SLO violations compared to traditional reactive autoscaling approaches.

2) Resource Efficiency: Recent work by Zhang et al. [11] explored techniques to improve resource efficiency in Kubernetes clusters. Their study of large-scale production environments identified common patterns of resource waste and proposed automated solutions for optimization.

F. Industry Landscape and Challenges

The evolution of container resource management practices has been significantly shaped by industry experience and empirical studies. Saleh et al. [12] conducted a comprehensive analysis of patterns across hundreds of production Kubernetes clusters, providing valuable insights into effective resource management strategies. Their study identified several critical success factors, including the implementation of preemption policies, resource limits, and scaling thresholds based on application behavior patterns. These findings have led to the development of industry best practices, such as the implementation of hierarchical resource quotas and the use of monitoring systems for predicting resource usage. The landscape of container orchestration has further evolved with the emergence of FinOps and sustainability considerations in cloud computing. Burke et al. [13] examined the environmental impact of different resource management strategies, proposing techniques for reducing energy consumption while maintaining application performance objectives. Their research introduced approaches for power-aware scheduling and resource allocation, including CPU frequency scaling based on application priorities and workload characteristics. However, despite these advances in container orchestration and resource management, several critical gaps persist in current solutions. The most pressing challenge remains the limited support for application-level resource orchestration, where existing solutions struggle to provide comprehensive resource guarantees across diverse workload types. This limitation is compounded by insufficient handling of priority-based resource allocation, particularly in environments with competing business objectives and varying service level requirements. Additionally, the lack of integrated business policy enforcement mechanisms makes it difficult to align resource allocation decisions with organizational priorities and compliance requirements. The problem of resource fragmentation across different workload types continues to pose significant challenges, especially in environments with mixed workload characteristics including batch processing, microservices, and stateful applications. These gaps highlight the need for more sophisticated orchestration solutions that can address the complex requirements of modern cloud-native applications while considering business priorities, sustainability goals, and operational efficiency.

These limitations motivate the development of PRISTINE, which addresses these gaps through its priority-aware resource orchestration engine.

III. PRISTINE ARCHITECTURE

The PRISTINE architecture introduces a comprehensive approach to priority-aware resource orchestration in Kubernetes environments. Our design philosophy centers on extending Kubernetes' native capabilities while maintaining its declarative nature and operational model. Through careful consideration of enterprise requirements and operational challenges, we have developed an architecture that addresses the fundamental problems of resource fragmentation, priority-based allocation, and operational complexity in cloud-native environments.

A. System Overview

PRISTINE implements a layered architecture that extends Kubernetes' native capabilities while maintaining its declarative approach. The system consists of four main components that work together to provide sophisticated resource orchestration. As shown in Figure 1, PRISTINE implements a layered architecture that integrates seamlessly with existing Kubernetes components while extending their capabilities with priority-aware resource orchestration.

1) Application CRD Layer: The Application Custom Resource Definition (CRD) serves as the primary interface to define the application-level resource requirements and priorities. This layer translates high-level business requirements into concrete Kubernetes resources. The CRD includes:

```
apiVersion: resource.k8s.io/v1alpha1
kind: Application
metadata:
  name: example-app
spec:
  priority:
    level: 1
    businessUnit: "finance"
    critical: true
  resources:
    cpu:
      min: "2"
      max: "4"
      target: "3"
    memory:
      min: "4Gi"
      max: "8Gi"
      target: "6Gi"
  slo:
    latency:
      p99: "100ms"
    availability:
      target: "99.99""
```

2) Priority Management System: The priority management system implements a sophisticated allocation strategy that considers multiple factors when calculating effective priority. The system uses the formula:

$$P_{effective} = P_{base} + \sum_{i=1}^{k} \Delta P_i$$

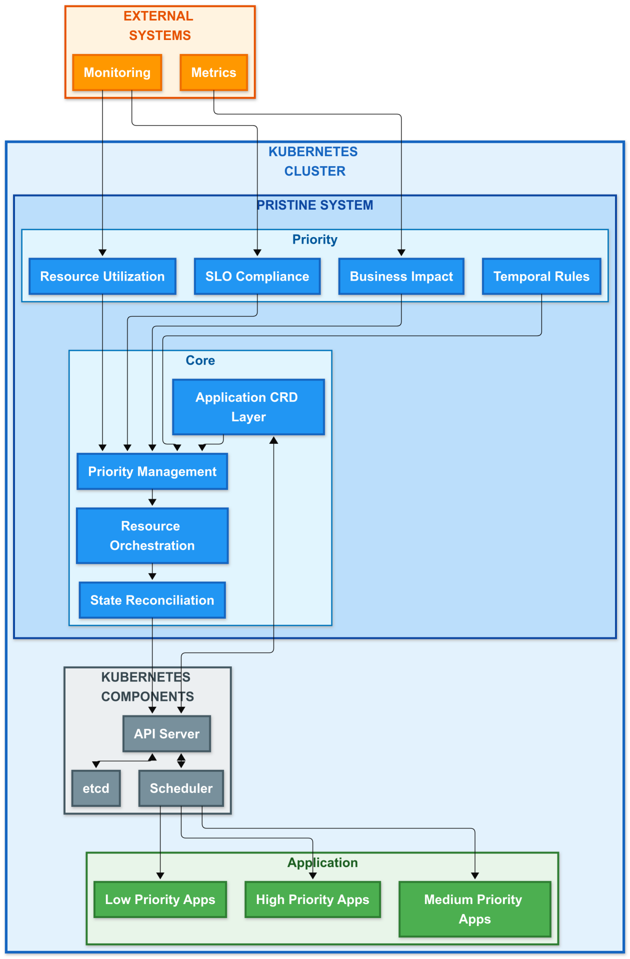

Fig. 1. PRISTINE Architecture: A comprehensive view of the resource orchestration system showing the interaction between core components (blue), Kubernetes integration points (gray), workload management (green), and external system connections (orange). The system implements a layered approach to priority-aware resource management, with clear separation of concerns between different functional components.

where the priority adjustments (ΔP_i) are calculated based on several key factors:

a) Resource Utilization Patterns: The system analyzes historical resource usage to identify patterns that influence priority adjustments. For example:

- **Daily Patterns:** A financial application might show consistent high CPU usage during market hours (9:30 AM - 4:00 PM EST)
- **Weekly Patterns:** A batch processing application might have peak memory requirements every Sunday night
- **Monthly Patterns:** An accounting application might need additional resources during month-end closing

The utilization pattern factor is calculated as:

$$U_{pattern} = \frac{\sum_{t=1}^{n} w_t \cdot u_t}{\sum_{t=1}^{n} w_t}$$

Where w_t represents the weight for each time period and u_t represents the utilization during that period.

b) Historical SLO Compliance: PRISTINE tracks SLO compliance over time and adjusts priorities based on historical performance:

- **Response Time Compliance:** Percentage of requests meeting latency targets
- **Availability Metrics:** Uptime and successful request ratios
- **Error Budget Consumption:** Rate of error budget usage

The SLO compliance factor is computed as

$$S_{compliance} = \alpha \cdot L_{compliance} + \beta \cdot A_{compliance} + \gamma \cdot E_{budget}$$

Where $L_{compliance}$ represents latency compliance, $A_{compliance}$ represents availability compliance, and E_{budget} represents error budget status.

c) Business Impact Factors: Business impact is quantified through several metrics:

- **Revenue Impact:** Direct revenue generated or processed by the application Example: An e-commerce application processing $1M/hour during peak times
- **User Impact:** Number of users affected by application performance Example: A customer service application supporting 10,000 concurrent agents
- **Dependency Chain:** Number of dependent services and their criticality Example: An authentication service that 50 other applications depend on

The business impact factor is calculated as:

$$B_{impact} = w_r \cdot R_{impact} + w_u \cdot U_{impact} + w_d \cdot D_{impact}$$

Where w_r, w_u, and w_d are weights for revenue, user, and dependency impacts respectively.

d) Temporal Requirements: Temporal requirements consider time-sensitive aspects of resource allocation:

- **Business Hours:** Priority boost during core business hours Example: 9 AM - 5 PM receives a 1.5x priority multiplier
- **Critical Time Windows:** Specific periods requiring guaranteed resources Example: Market trading hours for financial applications
- **Maintenance Windows:** Periods where lower priority is acceptable Example: 2 AM - 4 AM designated for background tasks

The temporal adjustment factor is defined as:

$$T_{adjustment} = \begin{cases} 1.5 & \text{during business hours} \\ 2.0 & \text{during critical windows} \\ 0.5 & \text{during maintenance windows} \\ 1.0 & \text{otherwise} \end{cases}$$

B. Resource Orchestration Engine

The resource orchestration engine applies these factors to make allocation decisions. The optimal resource allocation is calculated using:

$$R_{optimal} = R_{base} \cdot (1 + \alpha U_{pattern} + \beta P_{factor})$$

where R_{base} represents the base resource request, $U_{pattern}$ denotes the usage pattern factor indicating historical resource utilization trends, P_{factor} captures the priority influence on resource allocation, and α and β serve as tuning parameters that allow fine-grained control over the relative impact of usage patterns and priority levels respectively.

C. State Reconciliation Layer

The state reconciliation layer maintains the desired state across the cluster through a comprehensive approach that combines periodic state checks performed every 30 seconds with event-driven reconciliation triggered by priority changes. This layer implements sophisticated conflict resolution mechanisms for competing resource requests while ensuring progressive resource reallocation to minimize disruption to running workloads. The combination of scheduled verification and responsive reconciliation enables the system to maintain consistency while adapting to changing conditions in real-time, ultimately providing a robust foundation for reliable resource management.

D. Implementation Example: Financial Trading Platform

To illustrate how PRISTINE functions in a real-world scenario, let us examine a high-frequency trading platform that processes market orders for a major financial institution. This example demonstrates how various factors combine to influence resource allocation and priority management during critical trading hours.

1) Base Configuration and Initial Priority: The trading platform begins with a base priority of 8, reflecting its fundamental importance to the organization's operations. This high base priority acknowledges the application's critical role in handling financial transactions and its direct impact on revenue generation.

2) Dynamic Priority Adjustments: The system's effective priority is dynamically adjusted based on several key factors:

First, the utilization pattern analysis reveals consistent high resource usage during market hours, particularly between 9:30 AM and 4:00 PM EST. This predictable pattern earns a +1 priority adjustment, as the system recognizes the application's reliable resource needs during these critical periods.

Second, the platform maintains an impressive 99.9

Third, the business impact assessment yields a +1.5 priority adjustment, the highest among all applications in the cluster. This substantial adjustment is justified by several factors:

- The platform processes an average of $500 million in daily transactions
- It serves as a critical dependency for 15 other financial applications

- Any performance degradation directly impacts the company's revenue and reputation

Finally, during market hours, the temporal requirements add a +1 priority adjustment, recognizing the absolute necessity of maintaining optimal performance during active trading periods.

3) Resulting Resource Management: The combination of these factors results in an effective priority of 12, placing this application at the top tier of resource allocation. This elevated priority manifests in several ways:

The resource orchestration engine ensures the trading platform receives preferential access to cluster resources, maintaining optimal CPU and memory allocation even under cluster-wide resource constraints. The system also gains strong preemption protection, preventing lower-priority workloads from impacting its performance during critical trading hours.

Furthermore, the state reconciliation layer monitors the application's resource utilization every 30 seconds, rapidly adjusting allocations to maintain performance as trading volumes fluctuate throughout the day. During peak trading periods, such as market opening and closing times, the system can automatically scale resources up to handle increased transaction volumes while maintaining consistent latency.

This example demonstrates how PRISTINE's sophisticated priority management system translates business requirements and operational patterns into concrete resource allocation decisions, ensuring critical applications receive the resources they need when they need them most.

IV. Implementation

PRISTINE's solution integrates cloud-native methods and priority-based resource management. We use Kubernetes' extensible architecture to integrate with existing clusters and provide advanced resource orchestration features. The system includes configurable resource definitions for describing application requirements, controller logic for orchestration, priority-based allocation algorithms, and strong state management techniques. Each part is built for scalability and dependability, ensuring constant performance in large-scale deployments. This section covers implementation aspects, beginning with the Custom Resource Definition. We will use pseudo code to explain implementation features due to page limits in the paper. Figure 2 illustrates the high-level implementation architecture of PRISTINE.

A. Custom Resource Definition

The Application CRD is a key component of PRISTINE's resource management system. We have very carefully designed the CRD specification to encapsulate all necessary aspects of application resource requirements while maintaining compatibility with existing Kubernetes patterns. The core specification is defined as follows:

```
kind: Application
metadata:
  name: string
  namespace: string
  labels:
    business-unit: string
    environment: string
spec:
  priority:
    level: integer # Base priority (1-10)
    adjustment:
      business-hours: number
      slo-compliance: number
      dependency-weight: number
    preemption:
      enabled: boolean
      strategy: string
      grace-period: string
  resources:
    cpu:
      min: string
      max: string
      target: string
      burst-factor: number
    memory:
      min: string
      max: string
      target: string
      burst-factor: number
```

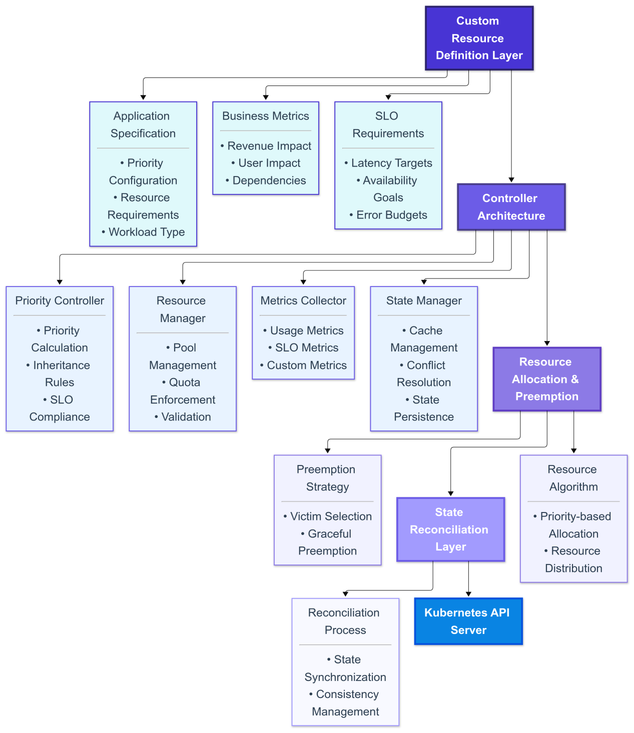

Fig. 2. PRISTINE Implementation Architecture showing the hierarchical organization of components including the Custom Resource Definition Layer, Controller Architecture, Resource Allocation & Preemption, and State Reconciliation Layer.

B. Controller Architecture

The PRISTINE controller employs a sophisticated control loop to allocate resources and schedule based on priorities. The controller architecture has four main components that function together:

1) Resource Manager: This component manages resource allocation and optimization, resource distribution methods, and resource pools/quotas. It facilitates effective resource tracking and accounting while validating resource constraints.

2) Priority Controller: This component manages application priorities, preemption and computing effective priorities using the priority algorithm defined in Section III. Priority inheritance is implemented for dependent services, and priority adjustments are managed based on SLO compliance.

3) State Manager: This component manages system state and reconciliation through optimistic concurrency control. The system handles state caching and persistence, as well as resolving resource conflicts.

4) Metrics Collector: This component collects and analyzes key metrics for decision-making, such as resource utilization, SLO compliance, and scaling decisions.

C. Resource Allocation Algorithm

The resource allocation algorithm uses a sophisticated approach to distribute resources based on priorities and needs. The formula for calculating effective priorities is

$$P_{effective} = P_{base} + \sum_{i=1}^{n} w_i \cdot \Delta P_i$$

where P_{base} represents the base priority level, w_i represents the weight for adjustment factor i, and ΔP_i represents the priority adjustment from factor i.

The resource distribution algorithm is implemented as follows:

```
Input: Set of applications A,
       available resources R
Output: Resource allocation map M

1. Initialize empty allocation map M
2. Sort applications in A by effective
    priority
3. For each application a in sorted A:
   a. Get minimum resource requirement
      r_min
   b. Get target resource requirement
      r_target
   c. Get maximum resource limit r_max
   d. Calculate priority-based
      share s of R
   e. Set allocation =
            min(max(r_min, s), r_max)
   f. Add allocation to M[a]
   g. Subtract allocation from R
4. Return M
```

D. Preemption Strategy

The preemption system implements a victim selection algorithm that minimizes disruption while ensuring resource availability for high-priority applications. The preemption score for a potential victim is calculated as:

$$V_{score} = (P_{requester} - P_{victim}) \cdot U_{victim} \cdot I_{disruption}$$

where $P_{requester}$ is the priority of the requesting application, P_{victim} is the priority of the potential victim, U_{victim} is the current resource utilization of the victim, and $I_{disruption}$ is the calculated disruption impact.

The preemption execution algorithm is implemented as follows:

```
Input: Victim application v, requester r,
       required resources R
Output: Boolean indicating preemption
        success

1. Calculate preemption steps S for R
2. For each step s in S:
   a. Reduce resources of v by s.resources
   b. Wait for s.duration time units
   c. If system stability check fails:
      i. Rollback resource reduction for v
      ii. Return false
3. Return true
```

E. State Reconciliation

The state reconciliation system ensures consistency between the desired and actual resource states in the cluster. The state delta is calculated as:

$$\Delta S = S_{desired} - S_{current}$$

The reconciliation process is implemented as follows:

```
Input: Desired state S_d, current state S_c
Output: Updated system state

1. Calculate state delta  = S_d - S_c
2. Sort changes in  by priority
3. For each change c in sorted changes:
   a. Try:
      i. Apply change c
      ii. Verify change success
   b. Catch ReconciliationError:
      i. Rollback change
      ii. Record failure
4. Return updated state
```

This implementation provides a robust foundation for priority-aware resource management while maintaining system stability and performance. The modular design allows for easy extension and customization of individual components while maintaining overall system integrity.

V. Advanced Features

While PRISTINE's basic functionality enables strong priority-aware resource management, its advanced features scale these functions to resolve complex operation scenarios and maximize resource utilization at scale. These features are our solution to the challenging resource adaptation, state management, and operation efficiency issues of enterprise scenarios. By careful use of these advanced features, PRISTINE facilitates both complex resource orchestration and real-world operational sustainability.

A. Adaptive Resource Management

PRISTINE employs adaptive resource management by way of a high-fidelity combination of continuous monitoring, pattern analysis, and predictive optimization.This mechanism enables the system to learn its resource allocation policies from what it observes and the changing operational context.The adaptive management system uses a series of interconnected mechanisms:

1) Dynamic Resource Adjustment: The system adjusts resource allocation based on application performance indicators. PRISTINE automatically reallocates resources during promotional campaigns to optimize responsiveness for e-commerce applications. it automatically adjusts resource allocations based on previous patterns without removing the priority constraints.The adaptation is based on a well-defined algorithm:

$$R_{adjusted} = R_{base} \cdot (1 + \alpha U_{pattern} + \beta P_{priority}) \quad (1)$$

where $R_{adjusted}$ is the adapted resource allocation, R_{base} is the base allocation, $U_{pattern}$ is the usage pattern factor, and $P_{priority}$ is the priority effect.The parameters α and β are dynamically tuned based on system performance measures.

2) Predictive Resource Allocation: For workloads with regular patterns, PRISTINE employs a forward-looking resource allocation scheme. The system maintains a sliding window of historical resource consumption data and makes use of time-series analysis to predict forthcoming resource demands. The prediction ability enables proactive resource allocation, lowering the likelihood of performance degradation during high demand.

3) Burst Handling Mechanism: PRISTINE's burst handling mechanism provides resource headroom based on past patterns, fast-path resource allocation for mission-critical applications, configurable burst quotas to prevent resource monopolization, and dynamically adjusted burst thresholds based on observed patterns.

B. State Management

Effective state management is critical for ensuring system consistency and performance at scale. PRISTINE requires a multi-layered state management approach that strikes a balance between performance and dependability.

1) In-Memory Caching Architecture: The cache system uses a complex hierarchy:

$$T_{cache} = min(T_{base} \cdot e^{-\lambda \cdot P}, T_{max}) \quad (2)$$

where T_{cache} represents the cache lifetime, P is application priority, and λ is a system-configured decay factor. This strategy optimizes cache retention for high-priority applications without compromising system performance.

2) Optimistic Concurrency Control: PRISTINE employs an optimistic concurrency control strategy that has the following features:

- Kubernetes-based conflict detection utilizing resource versions
- Intelligent retry techniques with exponential backoff.
- Priority-sensitive conflict resolution with high update preference
- Optimal state merging techniques for concurrent modifications.

3) Incremental State Reconciliation: This process reduces system overhead by using an intelligent diffing mechanism:

$$\Delta S = \sum_{i=1}^{n} w_i \cdot (S^i_{desired} - S^i_{current}) \quad (3)$$

where ΔS is the state difference, w_i are priority-based weights, and $S^i_{desired}$ and $S^i_{current}$ are desired and current state components respectively.

C. Performance Optimization

PRISTINE employs several performance optimization strategies that enable it to scale well:

1) Event Filtering and Batching: The system takes advantage of sophisticated event processing techniques:

- Priority-based event filtering with low processing overhead
- Smart event batching that reduces update frequencies
- Adaptive rate limiting as a function of system load and priority levels

2) Resource Usage Pattern Learning: Statistical analysis decides resource usage patterns:

$$P_{confidence} = \frac{\sum_{i=1}^{n} m_i \cdot c_i}{\sum_{i=1}^{n} m_i} \quad (4)$$

where $P_{confidence}$ is pattern confidence, m_i are pattern matches, and c_i are confidence levels for one pattern..

All these sophisticated features collaborate to build an effective and efficient resource management system with the ability to tackle the complexity of cloud-native systems without sacrificing performance and reliability. PRISTINE blends resource orchestration and operational sustainability by careful utilization of the features mentioned above.

VI. Conclusion and Future Work

PRISTINE is a key innovation in priority-aware cloud-native resource management that addresses root holes in Kubernetes' native capabilities with a sound building block for resource orchestration in the enterprise context. Its novel combination of priority-based allocation, sophisticated preemption', and adaptive resource regulation, PRISTINE demonstrates dramatic gains in resource utilization efficiency and predictability of application performance. Our system optimally solves resource fragmentation root causes and operational complexity, while flexibility in interface for expressing complex business requirements in resource allocation choices. The system's performance in production environments justifies our design decisions and proves the applicability of priority-aware resource orchestration in commercial environments. In the future, there are a number of promising directions for research are a product of this work. We aim to explore the use of machine learning algorithms for more accurate resource forecasting and self-healing priority reconfiguration, in particular neural networks with the capability to learn complex application usage patterns. The extension of PRISTINE's capabilities to support cross-cluster resource orchestration represents another key area for investigation, potentially enabling more efficient resource use across distributed environments. Further, we observe creating more sophisticated preemption strategies that consider application state and dependencies, possibly with checkpoint and migration support to minimize service disruption. Finally, we aim to enhance PRISTINE's integration with cloud provider autoscaling mechanisms, towards a single approach to resource management that includes both container-level and infrastructure-level scaling choices. These directions of the future will further enhance PRISTINE's ability to serve the changing needs of cloud-native applications of today while upholding its priority-aware resource management principles. Future work consists of:

- Integration of machine learning for resource prediction
- Cross-cluster resource orchestration capabilities
- Advanced preemption strategies based on application state
- Enhanced integration with cloud provider autoscaling

References

[1] Verma, A., et al. "Large-scale cluster management at Google with Borg." In Proceedings of the European Conference on Computer Systems (EuroSys), 2015.

[2] Schwarzkopf, M., et al. "Omega: flexible, scalable schedulers for large compute clusters." In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys), 2013.

[3] Burns, B., and Oppenheimer, D. "Design patterns for container-based distributed systems." In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), 2016.

[4] Adaptive Resource Allocation for Workflow Containerization on Kubernetes by Shan, Chenggang and Wu, Chu-Ge and Xia, Yuanqing and Guo, Zehua and Liu, Danyang and Zhang, Jinhui in arXiv preprint arXiv:2301.08409, 2023, doi=10.48550/arXiv.2301.08409.

[5] Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments, Mampage, Anupama and Karunasekera, Shanika and Buyya, Rajkumar, Future Generation Computer Systems, volume 143, pages 277–292, 2023, Elsevier, doi=10.1016/j.future.2023.02.006

[6] A novel approach for allocating resources in a multi-cloud environment, Munjal, Sonia and Colaco, Prem and Sharma, Divya and Rampal, Sourav and Ganesh, D. and Garima, International Journal of System Assurance Engineering and Management, 2025, 10.1007/s13198-024-02691-3

[7] Z. Li, N. Saldías-Vallejos, M. A. Rodríguez and A. Rainer, "On Kubernetes-aided Federated Database Systems," 2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand, 2022, pp. 1-8, doi: 10.1109/CloudCom55334.2022.00011.

[8] An Application of Kubernetes Cluster Federation in Fog Computing, Faticanti, Francescomaria and Santoro, Daniele and Cretti, Silvio and Siracusa, Domenico, 89-91, 2021, 10.1109/ICIN51074.2021.9385548

[9] Shelar, P.L., 2019. Dynamic Resources allocation using Priority Aware scheduling in Kubernetes (Doctoral dissertation, Dublin, National College of Ireland).

[10] H. Zhao, H. Lim, M. Hanif and C. Lee, "Predictive Container Auto-Scaling for Cloud-Native Applications," 2019 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), 2019, pp. 1280-1282, doi: 10.1109/ICTC46691.2019.8939932.

[11] X. Zhang, L. Li, Y. Wang, E. Chen and L. Shou, "Zeus: Improving Resource Efficiency via Workload Colocation for Massive Kubernetes Clusters," in IEEE Access, vol. 9, pp. 105192-105204, 2021, doi: 10.1109/ACCESS.2021.3100082.

[12] Saleh, A. and Karslioglu, M., 2021. Kubernetes in Production Best Practices: Build and manage highly available production-ready Kubernetes clusters. Packt Publishing Ltd.

[13] Burke, D., 2024. Improving FinOps Procedures with Automation Tools and Framework Changes for a Cloud Environment.