MC-Kube : SDV 에서의 Mixed-Criticality Task Orchestrator

제출일 : 2025 년 11 월 20 일

지도교수 : 김세화 교수님

정보통신공학과	202002832	이해준
정보통신공학과	202003557	최우인
정보통신공학과	201903749	한원탁

제목 : SDV에서의 Mixed-Criticality Task Orchestrator

대 학 : 공 과 대 학　　학 과 : 정보통신공학과

학 번 : 202002832　　성 명 : 이해준
202003557　　최우인
201903749　　한원탁

이 논문을 제출 하오니 승인하여 주십시오.

2025년　11월　20일

성 명 : 이해준 (서명)

최우인 (서명)

한원탁 (서명)

위 학생의 논문 제출을 승인함.

2025년　11월　20일

지 도 교 수 : 김세화 (서명)

요 약

클라우드와 MSA (Micro Service Architecture) 구조가 확산되고 있다. 대표적으로 쿠버네티스 (Kubernetes) 상에서 산업용 IoT, 자율주행, 로보틱스와 같은 실시간(Real-Time) 워크로드를 실행 있다. 하지만 쿠버네티스의 기본 CFS(Completely Fair Scheduler) 스케줄러는 응답 시간을 보장하지 못하며, 기존의 실시간 연구들 또한 태스크의 중요도(Criticality)를 구분하지 않거나 오버런 (Overrun) 상황 감지 및 처리에 높은 오버헤드를 유발하는 한계가 있었다. 본 논문에서는 혼합 중요도(Mixed-Criticality) 실시간 워크로드를 쿠버네티스 환경에서 안정적이고 효율적으로 오케스트레이션하기 위한 MC-Kube 시스템을 제안한다.

MC-Kube는 hierarchical-CBS 커널 패치를 도입하여, Cgroup v2 환경에서 컨테이너 단위의 SCHED_DEADLINE 스케줄링을 보장한다. 또한, 중요도 기반의 지능형 오케스트레이션 컨트롤러를 통해 높은 중요도 태스크의 자원 확보가 필요할 시, 해당 코어의 낮은 중요도 태스크를 선제적으로 마이그레이션 한다. 특히, eBPF를 활용하여 커널 레벨에서 태스크 오버런을 감지함으로써, API 서버를 이용하는 기존 방식 대비 오버헤드를 획기적으로 줄였다. 실제 72개의 혼합 중요도 컨테이너를 사용한 성능 평가에서, MC-Kube는 기존 RT-kube 대비 높은 중요도 태스크의 데드라인 미스(Deadline Miss) 횟수를 143회에서 19회로 약 86% 감소시켰다. 또한, 오버런 감지 및 대응 지연 시간을 기존 73,770μs에서 299μs로 단축시켜, 제안하는 시스템의 효율성과 실시간성을 입증하였다.

1 서론

최근 Software-Defined Vehicle(이하 SDV), 로보틱스, IoT 와 같은 지능형 엣지 시스템 (Intelligent Edge Systems) 분야에서도 클라우드 네이티브(Cloud-Native) 기술의 도입이 활발히 논의되고 있다. 특히 애플리케이션을 독립적인 서비스로 분리하는 마이크로서비스 아키텍처(MSA)와, 이를 배포·운영하는 컨테이너화(Containerization) 기술은 시스템의 모듈성, 확장성, 그리고 배포 유연성을 획기적으로 향상시킨다. 이러한 이점은 개발자들이 복잡한 시스템을 더욱 빠르고 안정적으로 개발, 테스트 및 배포할 수 있게 한다.

그러나 이러한 지능형 엣지 시스템은 실제 환경과 밀접하게 연결된 미션 크리티컬(Mission-Critical) 특성을 가진다. 일반적인 워크로드와 달리, 시스템 오류는 치명적인 서비스 장애나 물리적 손상, 심지어 인명 사고로 이어질 수 있다. 따라서 해당 도메인에서는 엄격한 서비스 품질(QoS), 높은 신뢰성(Reliability), 그리고 결정론적인 실시간성(Deterministic Real-time performance)의 보장과 같은 엄격한 요구 사항이 필수적이다.

이러한 요구 사항을 만족하며 컨테이너화된 워크로드를 관리하기 위해서는 컨테이너 오케스트레이터의 역할이 중요하다. 대표적인 오케스트레이터인 Kubernetes(이하 K8s)는 사실상 표준으로 널리 사용되고 있지만, 본질적으로는 클라우드 환경에서의 공정성(fairness)과 처리율(throughput)을 최적화하도록 설계되었다. 이로 인해 미션 크리티컬 작업의 요구사항을 충족시키기 어려운 구조적 한계를 가지고 있다.

1. 비결정론적(Nondeterministic) vs 결정론적(Deterministic): K8s 는 기본적으로 fair-sharing 과 throughput maximization 을 목표로 하는 Completely Fair Scheduler(이하 CFS)기반의 자원 스케줄링을 채택한다. 이는 일반 클라우드 서비스에서는 높은 자원 활용률을 제공하지만, 주기적(periodic) 태스크, Worst-Case Execution Time (이하 WCET), deadline 준수를 요구하는 실시간 작업에는 적합하지 않다. 특히 태스크의 응답 지연(latency)을 예측할 수 없게 만들어 컨테이너 내부 실시간 태스크의 결정론적 실행(deterministic execution)을 보장하지 못한다.
2. 리소스 경합 및 노이지 네이버(Noisy Neighbor) 문제: K8s 는 다수의 컨테이너가 동일 노드의 리소스를 공유하도록 설계되어 있으나, 이러한 구조는 노이지 네이버 문제를 유발한다. 특정 컨테이너가 CPU, 메모리 또는 I/O 를 과도하게 점유하면 동일 노드의 실시간 태스크에서 jitter, latency spike, deadline miss 가 발생할 수 있다. 이는 SDV 의 센싱(sencing)–플래닝(planning)–제어(control) 루프나 로보틱스의 SLAM·Perception 과 같은 시간 민감적 파이프라인에서 치명적이다.

SDV에서의 Mixed-Criticality Task Orchestrator (이해준, 최우인, 한원탁)

3. 혼합 중요도 관리의 부재: SDV·로보틱스·IoT 시스템은 서로 다른 중요도(Criticality Level)를 가진 태스크들이 공존한다. 그러나 K8s 는 높은 중요도(High-criticality) 태스크를 보호하고, 시스템 과부하 시 낮은 중요도(Low-criticality) 태스크를 선별적으로 제어하는 중요도 인지(Criticality-aware) 메커니즘이 부재하여 강건한 실행을 보장하지 못한다.

이처럼, 미션 크리티컬한 워크로드에 대한 오케스트레이터는 혼합 중요도를 관리하며, 결정론적인 실행, 독립적인 리소스 보장이 필요하다.

전통적인 실시간 시스템 분야에서는 태스크의 중요도(Criticality)에 따라 실행 보장을 차등적으로 제공하는 혼합 중요도 시스템(Mixed-Criticality System, 이하 MCS) 모델이 널리 연구되어 왔다. MCS 는 High-criticality 태스크의 안정성과 deadline 보장을 위해 실행 모드 전환, 자원 재할당, Low-criticality 태스크의 축소·중단 등을 포함하는 체계적인 실행 모델을 제공한다.

하지만 이러한 MCS 모델은 단일 노드, 단일 운영체제(OS), 태스크 중심의 스케줄링을 기반으로 정의되어 있으며, 컨테이너 기반의 분산 오케스트레이션 환경에서는 그 자체로 활용되기 어렵다. 현대의 엣지·클라우드 네이티브 시스템에서는 실행 단위가 컨테이너로 확장되고, 스케줄링 대상은 클러스터 전역으로 확대되기 때문에, MCS 개념을 오케스트레이터 수준으로 재해석하고 확장하는 혼합 중요도 기반 컨테이너 오케스트레이션이 필요하다.

이러한 배경에서 본 연구는 MCS 이론을 기반으로 한 Mixed-Criticality Orchestrator(MC-Kube)를 설계하여, K8s 환경에서도 High-criticality 워크로드의 결정론적 실행과 자원 격리, 그리고 혼합 중요도 정책을 지원할 수 있도록 한다. 본 연구는 특히 주기성(Periodicity), 마감 기한(Deadline), 그리고 중요도를 가진 컨테이너화된 태스크들을 클러스터 수준에서 안전하게 관리하기 위한 오케스트레이션 메커니즘을 제안한다.

본 논문의 핵심 기여는 다음과 같다.

1. Two-Level Scheduling 기반 리소스 컨트롤러 구현: 선행 연구인 Hierarchical-Constant Bandwidth Server(이하 HCBS)의 커널 기능을 K8s 오케스트레이션 계층과 연동하는 새로운 Resource Controller 를 설계하였다. 이는 cpu.rt_multi_runtime_us 인터페이스를 확장하여, Pod 단위가 아닌 컨테이너 단위로 SCHED_DEADLINE 파라미터를 동적으로 제어함으로써 실질적인 데드라인 보장을 실현한다.

2. 중요도 인지형 오케스트레이션 (컨트롤러 레벨): 본 연구에서는 K8s 상에서 혼합 중요도 정책을 실행하기 위해 전용 Custom Controller 를 설계·구현하였다. 이를 위해 McKube CRD(Custom Resource Definition)를 통해, 사용자가 각 태스크(컨테이너)의 중요도, 주기, 실행 시간(runtime), 자원 요구도 등 정책에 필요한 파라미터를 선언적으로 기술할 수 있도록 하였다. 개발된 McKube Controller 는 eBPF 를 기반으로 각 컨테이너의 상태를 지속적으로 관찰하며, 중요도 기반 정책(Criticality-aware Policy)을 자동으로 실행하는 정책 엔진으로 동작한다.
3. eBPF 기반 초저지연 오버런 감지 및 대응: 태스크 오버런 감지를 위해 API 서버를 경유하는 대신 eBPF 를 사용하여 커널 레벨에서 직접 컨테이너별 이벤트를 감지한다. 오버런 발생 시, 중앙 컨트롤러에 즉각적으로 전송되어 설정된 정책을 통해 각 노드에 배포된 Resource-controller 가 즉각적으로 Cgroup 파라미터를 동적으로 조정하여 데드라인 미스를 방지한다.

제안하는 MC-Kube 시스템의 성능을 입증하기 위해 Vanilla K8s 및 선행 연구인 [1](RT-kube)와 비교 실험을 수행하였다. 72 개의 혼합 중요도 컨테이너를 동시에 실행하는 시나리오에서, MC-Kube 는 High-Criticality 태스크의 데드라인 미스를 20 회로 억제하여, Vanilla K8s (724 회) 및 RT-kube (297 회) 대비 획기적인 성능 개선을 달성하였다. 또한, eBPF 기반 감지 메커니즘은 RT-kube 의 API 폴링 방식(73,770μs) 대비 200 배 이상 빠른 299μs 의 평균 감지 지연 시간을(Fig.n) 보여주었다.

본 논문의 나머지 구성은 다음과 같다. 2 장에서는 관련 연구를, 3 장에서는 MC-Kube 의 상세 설계 및 구현을, 4 장에서는 성능 평가 및 결과 분석을, 5 장에서 결론을 맺는다.

2 관련 연구

본 장에서는 본 연구의 배경이 되는 세 가지 주요 연구 흐름을 기술한다. 첫째, SDV 및 로보틱스 분야의 임베디드 태스크를 컨테이너로 전환하는 동향과 이에 따른 성능적 과제를 분석한다. 둘째, 이러한 컨테이너화된 워크로드를 관리하기 위한 K8s 기반 오케스트레이션 플랫폼의 발전 및 실시간성 확보 연구를 고찰한다. 마지막으로, 자원 효율성을 높이기 위한 혼합 중요도 시스템(MCS)의 적용 사례와 기존 연구들이 갖는 기술적 한계(Research Gap)를 명확히 한다.

2.1 SDV 및 로보틱스 워크로드의 컨테이너화와 성능 과제

최근 자동차 및 로보틱스 산업에서는 하드웨어에 종속적인 기존 임베디드 소프트웨어를 유연한 마이크로서비스 구조로 전환하기 위해 컨테이너 기술 도입을 가속화하고 있다. 특히 산업계에서는 SOAFEE(Scalable Open Architecture for Embedded Edge)[19]와 Eclipse SDV[20] 프로젝트가 주도하여, 클라우드 네이티브 기술을 차량 엣지에 적용하는 표준 아키텍처를 정립하고 있다. 또한, LG전자가 주도하는 오픈소스 프로젝트 'Eclipse Pullpiri' [21]는 차량의 상태, 환경, 서비스 요구사항과 같은 다양한 컨텍스트에 따라 미리 정의된 시나리오를 효율적으로 활성화하는 차량용 서비스 오케스트레이터 프레임워크(Vehicle Service Orchestrator Framework)를 제안하고 있다. 이러한 산업계의 움직임은 단순한 가상화를 넘어, SDV를 위한 차세대 컨테이너 오케스트레이션 기술이 필수적인 요소로 자리 잡고 있음을 시사한다.

학계에서도 이러한 흐름에 발맞추어 다양한 연구가 진행되고 있다. Reke et al. [1]은 ROS 2 기반의 자율주행 차량 아키텍처를 제안하며, 센싱, 판단, 제어와 같은 핵심 주행 태스크들을 컨테이너로 패키징하여 시스템의 이식성과 확장성을 높이는 방안을 제시하였다. 또한 Laclau et al. [4]은 SDV 환경에서 차량 내 인포테인먼트(IVI)와 Vehicle to Everything(V2X) 통신 서비스를 컨테이너 기반의 애플리케이션으로 정의하고, 사용자 경험(UX) 요구사항에 맞춰 서비스 레벨을 동적으로 관리하는 모델을 연구하였다.

그러나 실시간성이 필수적인 임베디드 태스크를 컨테이너화하는 것은 필연적으로 가상화 계층으로 인한 성능 오버헤드를 수반한다. Betz et al. [6]과 Wang and Bao [7]는 자율주행 시스템의 주요 태스크를 컨테이너 및 중첩된(Nested) 컨테이너 환경으로 마이그레이션할 때 발생하는 종단간(End-to-End) 지연 시간과 통신 지터(Jitter)를 실험적으로 분석하였다. 이들의 연구 결과, 컨테이너 레이어는 네이티브 환경 대비 예측 불가능한 지연을 유발할 수 있음을 시사한다. 또한 Kronauer et al. [8]은 분산된 ROS 2 노드 간의 통신 지연을 분석하고, 데이터 분배 서비스(DDS) 설

정과 네트워크 경합이 실시간 성능에 미치는 부정적 영향을 지적하였다. 이러한 연구들은 태스크의 컨테이너화(Containerization) 자체가 시스템의 복잡성을 높이고 시간적 결정성(Determinism) 저해할 수 있어, 단순한 도입을 넘어선 정교한 격리 기술이 필수적임을 역설한다.

2.2 실시간 컨테이너 오케스트레이션 플랫폼 및 스케줄링 연구

컨테이너화 된 워크로드의 배포와 관리를 자동화하기 위해 k8s와 같은 오케스트레이션 플랫폼을 도입하려는 시도가 활발하다. Zhang et al. [2]의 'KubeROS'와 Lampe et al. [3]의 'RobotKube'는 각각 K8s를 활용하여 다중 로봇 시스템을 자동화하고 배포하는 통합 플랫폼을 제안하였으며, Carmona Cejudo and Siddiqui [5]는 V2X 시나리오에서 엣지-클라우드 간의 파드(Pod) 오프로딩을 최적화하는 프레임워크를 제시하였다. 이는 로보틱스와 SDV 분야에서도 클라우드 네이티브 오케스트레이션이 표준적인 관리 방식(De-facto Standard)으로 자리 잡고 있음을 보여준다.

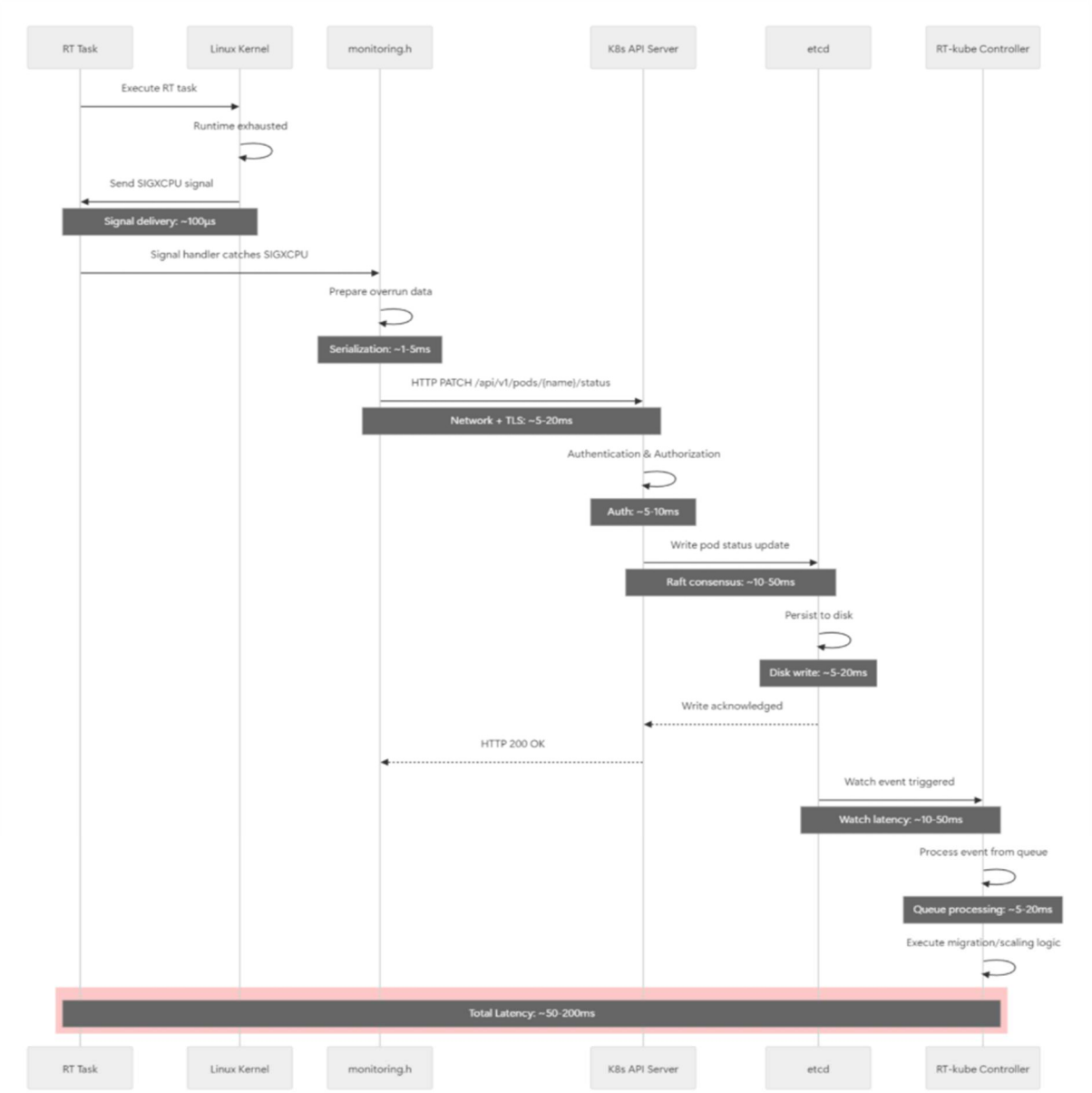
그러나 표준 K8s 스케줄러는 실시간성을 보장하지 못하므로, 이를 보완하기 위해 커널과 스케줄러를 확장하는 연구가 진행되었다. 초기 연구인 Cinque and Tommasi [11]는 'rt-cases' 개념을 통해 실시간/비실시간 컨테이너가 공존하는 아키텍처를 제안하였으며, Abeni et al. [12]은 리눅스 커널의 SCHED_DEADLINE을 Cgroup에 적용하여 컨테이너 단위의 시간적 격리(Temporal Isolation)를 이론적으로 정립하였다. 이를 오케스트레이션 레벨로 확장한 Fiori et al. [13]은 'RT-Kubernetes'를 통해 계층적 스케줄링(Hierarchical Scheduling)을 구현하여 파드 간 CPU 간섭을 줄이고자 했으며, Samimi et al. [14]은 분산 컴퓨팅 프레임워크(Ray) 환경에서 실시간 제약 조건을 관리하는 스케줄링 기법을 제안하였다. 또한 Lee et al. [15]은 자율주행 태스크 그래프에 대해 확률적 분석(Stochastic Analysis)을 적용하여 종단 간 지연 시간을 보장하려 시도하였다. 한편 Lumpp et al. [9, 10]은 로봇 애플리케이션의 정적 구조를 기반으로 Heterogeneous Earliest Finish Time (HEFT) 알고리즘을 적용하여 스케줄링 효율을 높이는 연구를 수행하였다.

2.3 혼합 중요도 시스템(MCS)의 적용 및 한계

제한된 엣지 자원 내에서 효율성을 극대화하기 위해, 중요도가 서로 다른 태스크를 단일 플랫폼에서 통합 운영하는 혼합 중요도 시스템(MCS) 연구가 오케스트레이션 분야로 확장되고 있다. Lee et al. [16]은 SDV 서비스를 혼합 중요도 애플리케이션(MCA)으로 모델링하고 런타임에 동적으로 매핑하는 프레임워크를 제안하였으며, Lumpp et al. [17] (RT-Kube)은 시스템 과부하 시 낮은 중요도(Low-Criticality) 태스크를 노드 마이그레이션하여 높은 중요도 태스크

를 보호하는 메커니즘을 구현하였다. 또한 Casini et al. [18]은 모니터링 기반의 분석을 통해 엣지 오케스트레이션의 제약 조건을 관리하는 기법을 제시하였다.

그러나 기존 MCS 오케스트레이션 연구들은 런타임에 발생하는 자원 간섭 및 교란(Resource Interference and Disturbances)에 실시간으로 대응하는 데 있어 접근 방식에 따른 제약을 가진다. Samimi et al. [14]는 주로 애플리케이션의 배포 단계 (Deployment Phase) 시점에서 자원을 할당하는 데 집중하고 있어, 실행 중(Runtime)에 발생하는 예기치 못한 오버런이나 자원 경합에 즉각적으로 대응하는 메커니즘이 부재하다. 또한 Casini et al. [18]은 모니터링 기반의 분석 프레임워크를 제시하였으나, 분석 결과를 실제 컨테이너 제어(Actuation)로 연결하는 구체적인 런타임 피드백 루프와 그 지연 시간에 대한 고려는 미흡하다. 특히 Lumpp et al. [17] (RT-Kube)은 실질적인 런타임 제어를 구현하였다는 점에서 중요한 기여를 하였다. 다만, 해당 시스템은 오버런 감지 및 대응 과정에서 K8s API 서버를 경유하는 아키텍처를 채택하고 있어, 필연적으로 통신 지연이 발생할 수밖에 없는 구조적 특성을 가진다. 이러한 방식은 수만 마이크로초(μs) 단위의 지연을 유발할 수 있어, 밀리초(ms) 단위의 정밀한 제어가 요구되는 SDV의 고속 제어 루프(High-frequency Control Loop)를 완벽하게 보호하기에는 다소 제약이 따른다. 따라서 실시간 시스템의 강건성(Robustness)을 극대화하기 위해서는 API 서버를 거치지 않는 보다 즉각적인 대응 메커니즘이 요구된다.



[Fig. 1] RT-Kube의 오버런 감지 및 대응 sequence diagram

이 API 서버 중심의 피드백 루프는 상태 저장(stateful)이며 복잡하고, 수만 마이크로초(µs)에 달하는 막대한 지연 시간을 유발한다. 이러한 지연은 실시간 시스템의 데드라인 미스에 즉각적으로 대응하는 것을 불가능하게 만든다.

2.4 본 연구의 차별성 (Research Gap)

본 논문은 상기 분석된 연구들의 한계를 극복하기 위해 MC-Kube를 제안한다. 기존 연구들이 정적 배치 최적화에 머무르거나 높은 지연 시간을 갖는 API 기반 제어 방식을 사용했던

것과 달리, 본 연구는 다음 세 가지 핵심 기여를 통해 차별화를 꾀한다.

커널 레벨의 계층적 실시간성: Abeni [12]와 Fiori [13]의 커널 격리 기술을 확장하여, 오케스트레이터가 컨테이너 단위의 SCHED_DEADLINE 대역폭을 동적으로 정밀하게 제어하는 인터페이스를 구현한다.

중요도 기반 지능형 오케스트레이션: Lumpp [17]의 연구를 발전시켜, 태스크의 중요도(Criticality)에 따라 자원을 동적으로 재할당하고 낮은 중요도 태스크를 즉각적으로 제어(Throttling/Preemption)하는 정책 엔진을 제공한다.

eBPF 기반 초저지연 대응: 기존 연구의 API 서버 병목 현상을 해결하기 위해 API 서버를 우회하는 eBPF 기술을 적용하여, 수백 배 빠른 속도로 오버런을 감지하고 중요도 기반 정책을 커널 레벨에서 집행(Enforcement)한다.

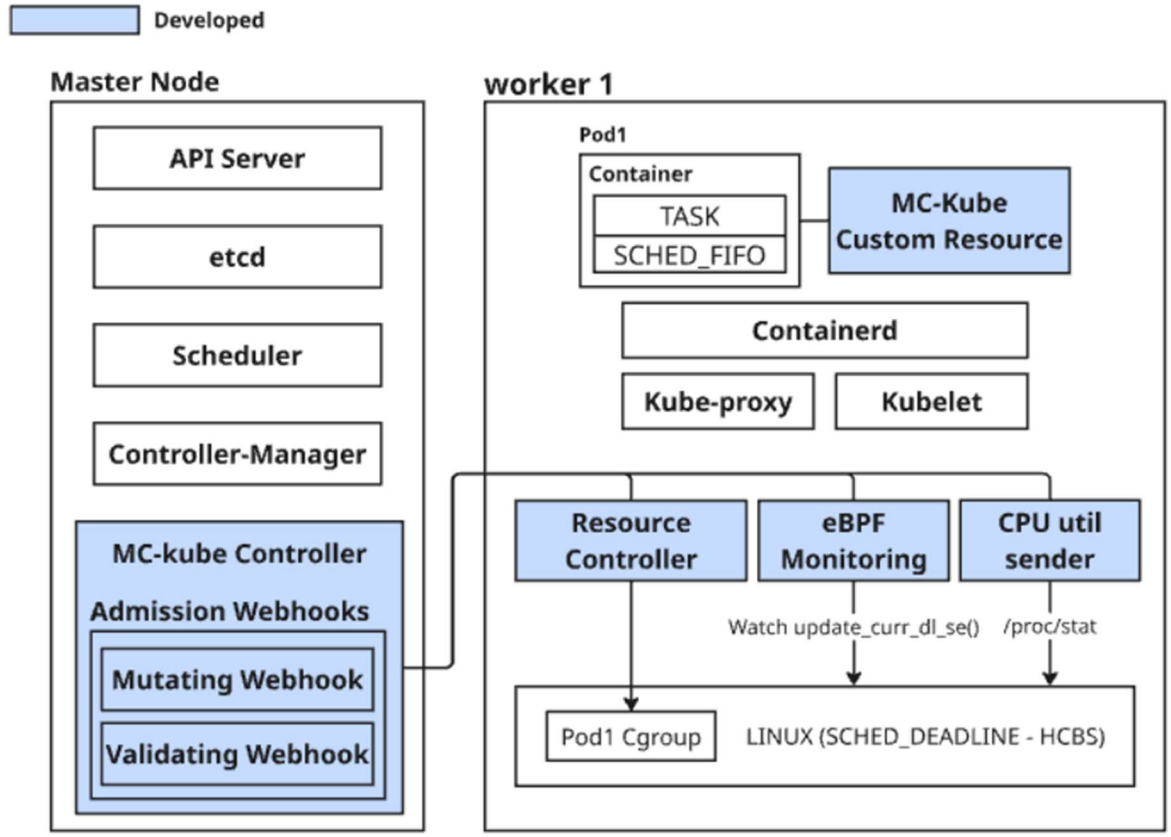
3 MC-Kube 시스템 설계 및 구현

본 연구에서 제안하는 MC-Kube는 K8s 환경에서 실행되는 실시간 워크로드가 주기성, 데드라인, 그리고 혼합 중요도 특성을 동시에 만족하도록 보장하는 오케스트레이션 시스템이다. 특히 자율주행(Software-Defined Vehicle), 로보틱스, 제조 자동화, 고신뢰 엣지 컴퓨팅과 같이 실시간성과 안정성이 동시에 요구되는 도메인을 주요 적용 대상으로 한다. 이러한 시스템들은 작업 간 중요도가 상이하며, 고중요도(High-Criticality) 태스크는 낮은 중요도 태스크로부터의 간섭 없이 반드시 제시간에 수행되어야 한다. 그러나 기존 K8s는 기본적으로 CFS에 기반해 동작하며, 중요도 기반 자원 분리나 데드라인 보장, 초저지연 오버런 대응과 같은 실시간 스케줄링 기능을 제공하지 않아 이러한 요구사항을 충족하기 어렵다.

MC-Kube는 이를 해결하기 위해 (1) 중요도 인지 스케줄링, (2) 코어 수준 격리, (3) 커널 레벨 실시간성 제어, (4) 저지연 오버런 대응을 통합한 아키텍처를 제공한다. CRD 기반 정책 선언, Custom Controller 기반 자원 배치, Resource-controller를 통한 Cgroup 실시간 설정, 그리고 eBPF 기반 오버런 감지·대응의 파이프라인을 통해 고중요도 태스크의 안정적 실행을 보장한다. 이를 적용함으로써 실시간 워크로드의 데드라인 준수율을 크게 향상시키고, 시스템 포화 상황에서도 고중요도 태스크의 실행 가능성을 유지하며, 커널-사용자 공간 간 지연을 최소화하여 기존 방식 대

비 훨씬 높은 신뢰성과 예측성을 확보할 수 있다.

MC-Kube는 K8s를 확장하여 설계되었으며, 제어 평면(Control Plane)의 정책 결정과 워커의 실시간 집행을 분리하면서도 긴밀하게 연동하는 구조를 가진다. 전체 시스템 아키텍처는 Fig. 2와 같다.



[Fig. 2] MC-Kube 전체 시스템 아키텍처

MC-Kube는 다음과 같은 5가지 핵심 컴포넌트로 구성된다.

1. McKube CR (Custom Resource): 사용자가 애플리케이션의 중요도(Criticality), 주기(Period), 실행 시간(Runtime) 등 실시간 요구사항을 선언적으로 기술하는데 사용되는 구성 객체이다.
2. Admission Webhook: 파드 생성 요청시 API 서버와 etcd 사이에 위치하여, 초기 리소스 설정의 유효성 검증 및 MC-Kube 관리 메타데이터 주입을 수행함으로써, RT(Realtime) 워크로드를 시스템 관리 대상으로 초기화(Initialization)하는 게이트키퍼(Gatekeeper) 역할을 한다.

3. MC-Kube Custom Controller: 클러스터 전역의 두뇌이다. CR 변경을 감지하여 글로벌 스케줄링 정책을 수립하고, 특히 자원 경합 시 Escalation(중요도 기반 자원 재할당) 정책을 결정하여 하위 리소스 컨트롤러에 지시한다.
4. Resource Controller: 각 워커 노드에 배포된 실행 에이전트이다. 중앙 컨트롤러의 정책을 수신하여 HCBS 커널의 Cgroup 인터페이스(cpu.rt_multi_runtime_us, cpuset.cpus)를 동적으로 수정함으로써 물리적인 자원 할당을 집행한다.
5. eBPF Monitoring Agent: eBPF 기반의 커널 레벨에서 동작하는 초저지연 감시 모듈이다. 컨테이너의 실행 흐름을 추적하다가 오버런(Overrun) 발생 시 API 서버를 거치지 않고 리소스 컨트롤러에게 즉각 신호를 보낸다.

본 연구는 시스템의 실시간성을 명확히 검증하기 위해 다음과 같은 환경을 전제한다. 우선, 자원 격리의 복잡성을 최소화하고 분석의 직관성을 높이기 위해 하나의 파드가 단일 컨테이너만을 포함하는 1:1 매핑 구조를 채택하였다. 또한, 결정론적 스케줄링 분석을 위해 모든 실시간 태스크는 주기적(Periodic)이며 Runtime, Period, Deadline이 명시적으로 주어진다고 가정한다. 이를 물리적으로 뒷받침하기 위해, 본 연구에서는 개별 컨테이너 단위로 리소스를 정밀하게 설정할 수 있도록 HCBS 커널 패치를 워커 노드에 적용하여 계층적 스케줄링 환경을 구축하였다. 아울러, 워크로드의 실행 시간 보장에 집중하기 위해 네트워크 대역폭이나 전송 지연과 같은 네트워크 요소는 이상적이라고 가정하고 본 모델의 고려 범위에서 제외한다.

3.1 본 연구의 차별성 (Research Gap)

표준 K8s 에서 제공하는 리소스 종류에는 realtime 과 관련된 리소스 지원이 부재하다. 따라서 본 연구에서는 Custom Resource Definition(CRD)을 사용하여, 사용자가 컨테이너별 실시간 제약 조건과 중요도를 YAML 형식으로 명시하고 관리할 수 있도록 확장하였다.

각 필드는 다음과 같이 구성된다.

Criticality	각 컨테이너의 중요도 (Criticality)
Period	각 컨테이너의 리소스 보장 주기
Runtime_low	Escalation 정책의 runtime lower value
Runtime_hi	Escalation 정책의 runtime upper value
Core	컨테이너에 할당될 코어

[Table 1] McKube Custom Resource field 구성

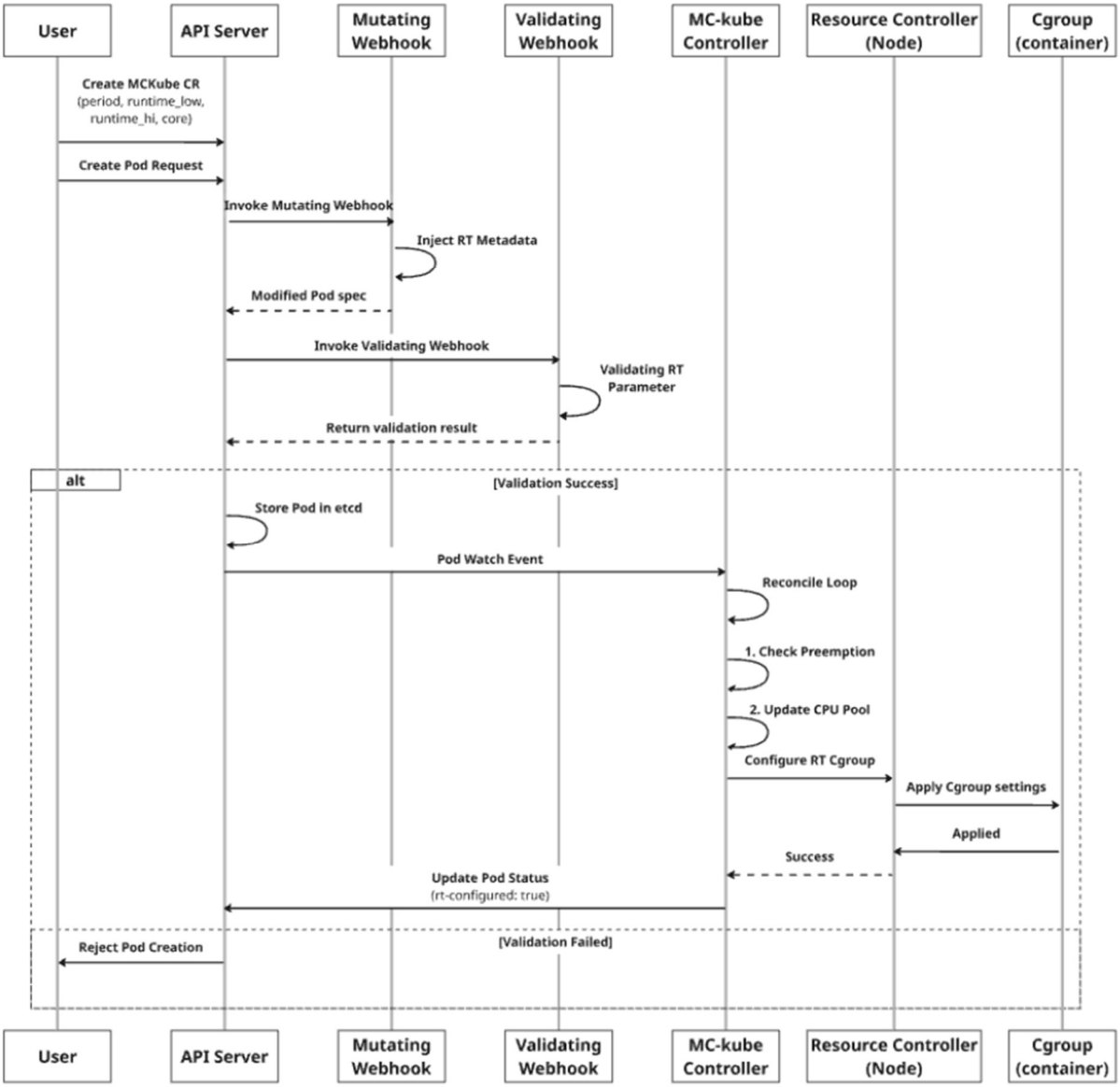
```yaml
apiVersion: mcoperator.sdv.com/v1
kind: McKube
metadata:
  name: rt-test-pod-config
  namespace: default
spec:
  podname: "rt-test-pod"
  criticality: "High"
  rtSettings:
    period: 1000000        # 1 second in microseconds
    runtime_low: 300000    # 300ms initial runtime (conservative)
    runtime_hi: 500000     # 500ms elevated runtime (after overrun detection)
    core: "2"            # Use cores 2
```

[Fig. 3] RealTime 과 관련된 Custom Resource 예시

Criticality(High, Middle, Low)의 경우 태스크의 중요도 수준을 정의한다. 이는 시스템 과부하 시 동작하는 Escalation 및 Core Migration 정책의 기준이 되며, 낮은 중요도(Low) 태스크는 높은 중요도(High) 태스크의 자원 확보를 위해 다른 노드로 축출(Eviction)되거나 다른 코어로 이주될 수 있다. Period는 해당 컨테이너가 리소스를 보장받아야 하는 주기(T)를 마이크로초(μs) 단위로 설정한다. SCHED_DEADLINE의 dl_period에 매핑된다. runtime(low/hi)의 경우 혼합 중요도 시스템의 가변적인 실행 시간을 지원하기 위한 듀얼 런타임 파라미터이다. runtime_low는 일반적인 상황에서의 실행 시간(Nominal WCET)이며, runtime_hi는 오버런 발생 등 비상 상황(Escalation)에서 시스템이 보장해야 하는 확장된 실행 시간(Pessimistic WCET)이다. MC-Kube는 상황에 따라 이 두 값 중에서 자원을 동적으로 조절한다. Core는 컨테이너가 할당 받기를 희망하는 선호 코어(Preferred Core) ID이다. 단, 3.3절에서 후술할 Core Migration 정책에 따라, 자원 경합 시 실제 할당되는 코어는 변경될 수 있다.

3.2 Admission Webhook

MC-Kube 시스템의 Admission Webhook은 클러스터에 새로운 파드가 생성되는 시점에 API 서버가 파드 생성 요청을 Webhook에 전달하여 해당 파드가 실시간 요구사항을 준수하는지 검증하고 MC-Kube의 관리 대상으로 초기화하는 게이트키퍼 역할을 수행한다. 이 과정은 Validating Webhook과 Mutating Webhook의 두 단계로 나뉘어 동작하며, Pod가 최종적으로 etcd에 저장되기 전에 필수적인 초기 RT 설정을 Pod Spec에 주입하는 역할을 한다.



[Fig. 4] Sequence diagram about Webhook Process

3.2.1 Mutating Webhook (Pod Mutator)

Mutating Webhook은 Pod의 사양(Spec)을 변경하고 필요한 RT 메타데이터를 주입하는 역할을 한다. 이 과정은 크게 메타데이터 주입과 비동기 RT 설정 적용으로 나뉜다.

A. RT 어노테이션(Annotation) 주입:

McKube CRD에 정의된 실시간 파라미터(period, runtime, core 등)를 조회한다.

이 값들을 Pod의 어노테이션 필드에 추가한다. 이 어노테이션들은 이후 Resource Controller가 실제로 Cgroup을 설정할 때 참고하는 최종 지침서 역할을 하며, MC-Kube

시스템 내에서 Pod의 RT 설정 상태(mckube.io/rt-configured, mckube.io/rt-pending 등)를 추적하는 데 사용된다.

B. 비동기 RT 설정 적용:

Mutating Webhook은 백그라운드 goroutine을 생성하여 Pod가 Running 상태가 될 때까지 비동기적으로 모니터링한다. Pod가 Running 상태로 확인되면, goroutine이 해당 노드의 Resource Controller에 초기 RT 설정(period, runtime, core)을 HTTP POST로 전송한다.

이 비동기 통신을 통해 Resource Controller가 커널 레벨에서 Cgroup 설정을 완료하도록 트리거하며, 이는 곧바로 HCBS 커널 패치가 적용된 cpu.rt_multi_runtime_us 인터페이스 값을 동적으로 수정하는 단계로 이어진다.

이러한 Mutating Webhook의 역할 덕분에, RT 워크로드는 생성되자마자 검증된 RT 파라미터와 함께 할당 받은 노드에서 즉시 커널 레벨의 실시간 제어를 받을 수 있도록 초기화된다.

3.2.2 Validating Webhook (RT Validator)

Validating Webhook은 Pod 생성 요청이 클러스터의 안전 정책과 McKube CRD 규칙을 준수하는지 사전에 검증한다. 이는 잠재적인 시스템 불안정 요소를 초기에 차단하여 클러스터의 건전성을 유지하는 데 목적이 있다.

Validating Webhook은 failurePolicy: Fail로 설정되어 있어, 검증 규칙을 하나라도 위반하는 Pod 생성 요청은 API Server 단계에서 즉시 거부되고 Pod 생성은 차단된다. 현재 MC-Kube의 검증 과정에는 다음과 같은 규칙이 존재한다.

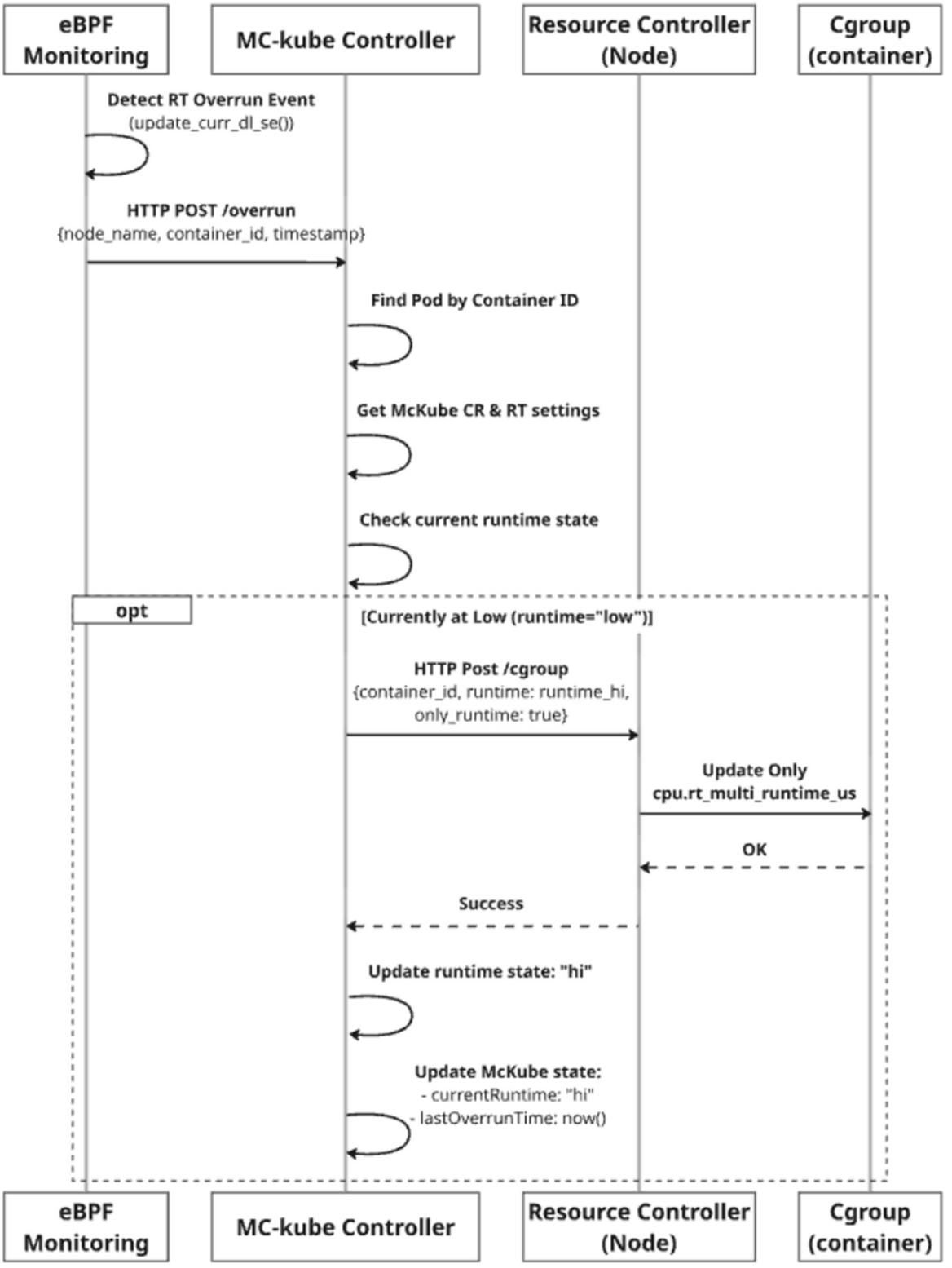
검증 규칙	**설명**
파라미터 유효성	모든 파라미터는 양수이며, 다음 논리적 순서를 만족해야 한다. 0 < runtime_low <= runtime_hi <= period

안전 임계값 검증	**Runtime은 Period의 95%를 초과할 수 없다.** 이 안전 임계값은 전체 CPU 자원의 과도한 예약(Over-reservation)을 방지하고, 나머지 태스크와 커널 작업을 위한 최소한의 유휴 자원을 보장한다.
Core 유효성	할당하고자 하는 Core ID 형식이 유효한지 검증한다.

[Table 2] Validating Webhook 검증 사항

3.3 MC-Kube Custom Controller

MC-Kube Custom Controller 는 K8s 확장 아키텍처에서 동작하며, 실시간(Real-Time, RT) 및 혼합 중요도 워크로드의 CPU 리소스 관리를 담당하는 중앙 제어 루프(Central Control Loop)이다. 표준 K8s 스케줄러는 CPU 요청량(Request)의 총합만을 관리할 뿐, 시간적 제약 조건인 대역폭(Bandwidth)과 마감 기한을 인지하지 못한다. 이를 해결하기 위해 MC-Kube 컨트롤러는 노드별 자원 상태를 모델링하고, 사용자가 정의한 정책(Policy)에 따라 동적 리소스 조정을 수행하는 정책 엔진(Policy Engine)으로 설계되었다.



[Fig. 5] Escalation Sequence Diagram

MC-Kube 컨트롤러는 다양한 사용자의 정책을 적용할 수 있다. 본 연구에서는 혼합 중요도 시스템의 요구사항을 검증하기 위해 다음과 같은 Escalation 및 Core Migration 정책을 구현

하여 적용하였다.

3.3.1 자원 모델링 및 허용 제어 (Resource Modeling & Admission Control)

컨트롤러는 클러스터 내 모든 워커 노드의 실시간 코어 상태를 추적하기 위해 'CPUPool' 이라는 추상화된 인메모리 데이터 구조를 유지 관리한다.

A. 사용률 추적 (Utilization Tracking): CPUPool은 각 물리 코어(CoreID)별로 할당된 SCHED_DEADLINE 대역폭의 총합을 관리한다. 코어의 실시간 사용률(U_{core})은 해당 코어에 배치된 모든 파드(i)의 실행 시간(runtime)과 주기(period)의 비율의 합으로 정의된다.

$$U_{core} = \sum_{i \in \text{Pods}(core)} \frac{\text{runtime\_millis}_i}{\text{period\_millis}_i}$$

여기서 Pods(core)는 해당 물리 코어에 할당된 파드들의 집합을 의미하며, 모든 단위는 밀리초(ms)로 환산하여 계산한다.

B. 임계값 기반 허용 제어 (Threshold-based Admission Control): SCHED_DEADLINE은 이론적으로 100% 이용률을 보장하지만, 컨텍스트 스위칭 오버헤드와 커널 스레드 실행을 고려하여 안전 마진(Safety Margin)이 필요하다. 이에 본 연구에서는 각 코어의 최대 허용 사용률(U_{th})을 90% (0.9)로 제한하였다. 컨트롤러는 새로운 태스크 할당 시, 예상되는 코어 사용률이 이 임계 값을 초과하지 않는 경우에만 배치를 승인한다.

$$\text{Admission Condition: } U_{core} + U_{new\_task} \leq 0.9$$

3.3.2 혼합 중요도 스케줄링 정책 (Scheduling Policies)

본 시스템은 시스템 과부하 및 오버런 상황에 대응하기 위해 다음 두 가지 정책을 유기적으로 연동한다.

A. Escalation 정책 (Criticality-Aware Escalation Policy)

컨테이너 내 태스크의 오버런이 감지되었을 때, 고중요도(High-Criticality) 컨테이너의 실행을 최우선으로 보장하기 위한 정책이다.

- 트리거 조건: eBPF 모니터링을 통해 상대적 중요한 컨테이너의 오버런이 감지될 경우
- 대응 로직: 해당 컨테이너의 런타임 파라미터를 일반 모드인 Runtime_low에서 확장 모드인 Runtime_hi로 즉시 격상(Escalation)시킨다. 이를 통해 컨테이너는 추가적인 CPU 자원을 확보하여 데드라인 미스를 방지할 수 있다.

B. Core Migration (동적 코어 재배치)

Escalation으로 인해 특정 컨테이너의 자원 점유율이 커지거나 새로운 고중요도 컨테이너가 유입되어 코어가 포화 상태가 될 경우, 시스템의 전체 가용성을 유지하기 위해 컨테이너를 재배치하는 정책이다.

- 트리거 조건: 특정 코어의 예상 사용률(U_{core})이 임계 값(U_{th} = 0.9)을 초과하는 경우
- 선점 로직: 경합이 발생한 코어 내에서, 요청된 태스크보다 중요도가 낮은 피해자(Victim) 태스크를 식별한다. 본 구현에서는 High > Middle > Low의 3단계 중요도 모델을 사용하며, 상위 중요도 태스크는 하위 중요도 태스크를 선점할 권한을 가진다. 이주 및 축출(Migration & Eviction): 식별된 Victim 태스크는 해당 코어의 자원 할당 풀에서 즉시 제외되며, 다음 단계에 따라 처리된다.
  I. 대상 코어 선정: 컨트롤러는 마이그레이션 대상 코어를 결정하기 위해 탐색 알고리즘을 수행한다. 본 연구의 현재 구현에서는 최소 부하 휴리스틱(Least Loaded Heuristic)을 적용하여, 노드 내에서 U_{core}가 가장 낮은 코어를 우선적으로 선택한다.
  II. 안전한 이주 (Safe Migration): 선정된 대상 코어에 Victim 태스크를 배치했을 때 임계 값을 초과하지 않는 경우에만 마이그레이션을 확정한다.
  III. 축출 (Eviction): 만약 모든 코어가 포화 상태여서 이주가 불가능할 경우, 최후의 수단으로 가장 낮은 중요도의 컨테이너를 축출(Eviction)하여 시스템 붕괴를 방지한다.

결정된 배치 정보는 해당 파드의 AllocatedCore 상태에 업데이트되며, 이는 즉시 노드의 Resource Controller로 전송되어 cpuset 및 cpu.rt_multi_runtime_us 설정을 변경함으로써 물리적인 실행 위치를 런타임 중에 변경한다.

3.4 Resource Controller

표준 K8s 의 Kubelet 은 cpu.shares(Request)와 cpu.quota(Limit)를 통해 CFS 파라미터만을 관리할 뿐, 실시간 스케줄링을 위한 커널 인터페이스는 제어하지 않는다. 이로 인해 파드가 생성되더라도 실시간 속성(cpu.rt_period_us 등)은 초기화되지 않거나 기본값으로 남게 되어 결정론적 실행을 보장할 수 없다.

본 연구는 이를 해결하기 위해 각 워커 노드에 Resource Controller 를 데몬셋(DaemonSet)으로 배포하여, 중앙 컨트롤러의 정책을 커널 레벨의 물리적 설정으로 변환하는 실시간 액추에이터(Real-Time Actuator) 역할을 수행하게 하였다. Resource Controller 는 MC-Kube Custom Controller 로부터 CgroupRequest (ContainerID, Period, Runtime, Core)를 수신하면, 해당 컨테이너의 Cgroup V2 경로를 식별하고, HCBS 커널 패치가 제공하는 확장 인터페이스인 cpu.rt_multi_runtime_us 를 조작한다.

이때 Cgroup V2 의 경우 하위 그룹의 리소스 총량이 상위 그룹의 제한을 초과할 수 없는 계층적 제약 조건을 가진다. 따라서 런타임 값을 변경할 때 순서가 잘못되면 설정 오류가 발생할 수 있다. 이를 방지하기 위해 본 시스템은 변경 유형에 따라 업데이트 순서를 제어하는 순차적 업데이트 로직(Ordered Update Logic)을 구현하였다.

할당량 증가(Increase)의 경우 상위 계층(Pod Slice)의 제한을 먼저 확장한 후, 하위 계층(Container Scope)의 런타임을 증가시킨다.

할당량 감소 (Decrease)의 경우 하위 계층(Container Scope)의 런타임을 먼저 축소한 후, 상위 계층(Pod Slice)의 제한을 줄인다.

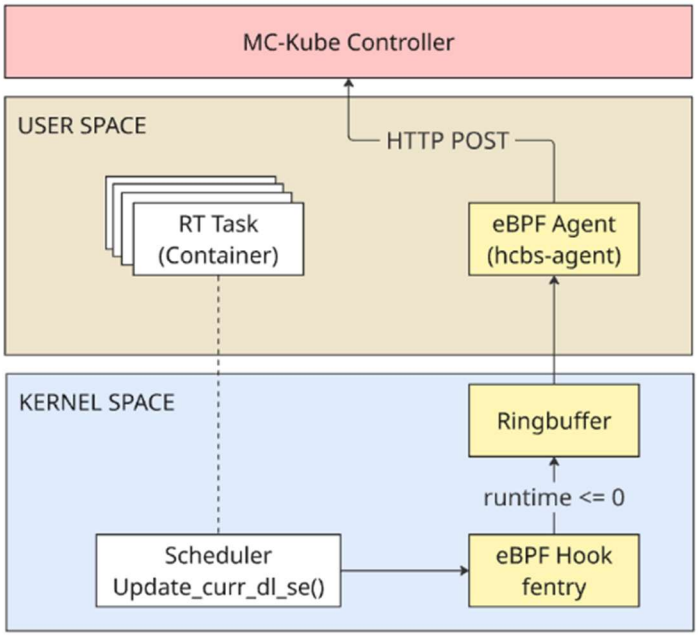
이러한 순차적 업데이트 로직은 런타임 중에 발생할 수 있는 설정 오류를 방지하고, 시스템의 안정적인 리소스 재구성을 보장한다. 또한, 3.3 절의 정책에 따라 코어 마이그레이션이 결정될 경우 cpuset.cpus, cpu.rt_multi_runtime_us 를 실시간으로 수정하여 Low-Criticality 컨테이너를 유휴 코어로 즉시 이주(Migration)시키는 메커니즘의 기반이 된다.

3.5 eBPF 기반 초저지연 감지 및 대응

MC-Kube 는 고중요도(High Criticality) 태스크의 WCET (Worst-Case Execution Time) 오버런을 컨테이너 그룹 단위에서 초저지연으로 탐지하고 즉각적으로 대응하기 위해 eBPF(Extended Berkeley Packet Filter) 모듈을 활용한다.

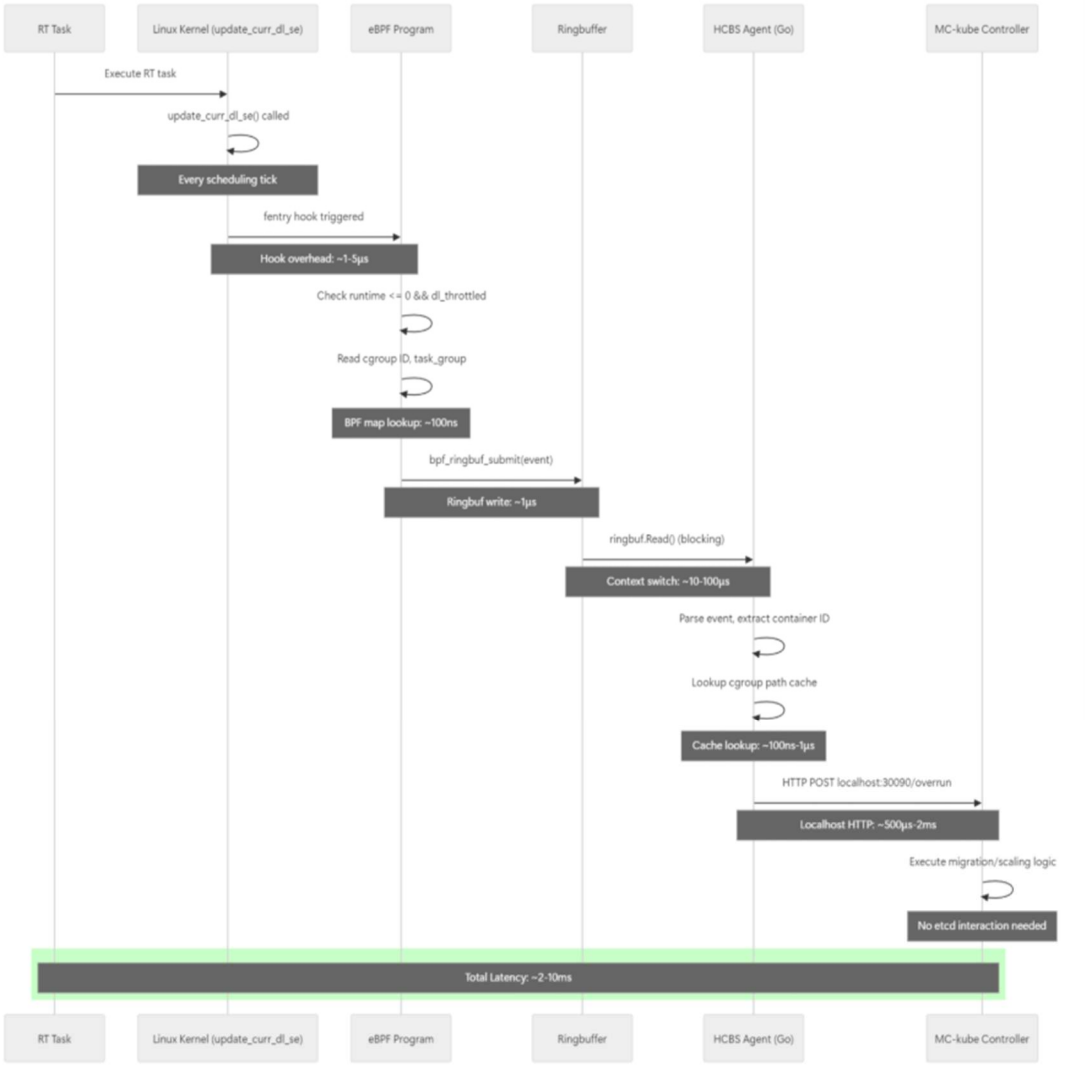
기존 선행 연구(RT-Kube)는 SCHED_DEADLINE 을 스레드 단위로 적용하였기에, 리눅스 커널의 SIGXCPU 시그널을 통해 WCET 오버런을 감지할 수 있었다. RT 스레드 당 하나의 모니터링 스레드를 붙여서 시그널을 처리하는 구조를 채택한 것이다. 그러나 본 연구의 MC-Kube 는 HCBS 패치를 적용하여 SCHED_DEADLINE 을 컨테이너 그룹 (cgroup) 단위로 사용한다. 이 구조에서는 WCET 오버런 감지는 개별 스레드 단위가 아닌 그룹 단위에서 주기적으로 할당된 리소스 예산이 모두 소진되었는지를 확인해야 한다.

이러한 요구사항을 충족시키기 위해, 리눅스 커널 내부에서 SIGXCPU 가 발생하던 흐름의 코드를 분석하여, 우리는 SCHED_DEADLINE 스케줄러 내에서 런타임 갱신을 담당하는 함수인 update_curr_dl_se()에 eBPF 프로그램을 부착하는 방식을 채택하였다.



[Fig. 6 eBPF] 기반 오버런 탐지 구조도

eBPF 프로그램은 커널 스케줄러(update_curr_dl_se())에서 발생한 오버런(runtime ≤ 0) 상황을 실시간으로 감지하며, Cgid(cgroup inode ID)를 포함한 메타데이터를 Ring Buffer를 통해 사용자 공간으로 전달한다. 이를 통해 스레드 단위가 아닌 컨테이너 단위로 오버런 발생 지점을 정확히 식별할 수 있다.



[Fig. 7] eBPF 기반 오버런 탐지 sequence diagram

감지된 이벤트는 K8s API 서버를 완전히 우회하는 Fast-Path 메커니즘을 통해 처리된다. eBPF 에이전트는 수신된 Cgid를 로컬 PathCache 맵을 통해 컨테이너 ID로 변환한 후, 이를 컨트롤러의 대응 엔드포인트로 HTTP POST 요청을 통해 직접 전송한다. 이러한 직접 통신 경로를 통해 시스템은 기존의 통신 오버헤드를 제거하고, 수백 마이크로초(μs) 이내에 런타임 보충(Replenishment)이나 에스컬레이션 정책을 즉각적으로 수행한다.

4 성능 평가 (Evaluation)

4.1 실험 환경 (Experimental Setup)

1) 하드웨어 및 소프트웨어 (Hardware and Software): 모든 실험은 16GiB RAM 을 탑재한 Intel Core i7-10700 CPU (@ 2.90GHz) 머신에서 수행되었다. 클러스터는 Kubernetes v1.34.1 로 구성하였다.

2) 워크로드 구성 (Workload Configuration): 총 72 개의 단일 스레드 실시간 컨테이너가 생성되었으며, 중요도는 High(16 개), Medium(24 개), Low(32 개) 세 종류로 구분하였다. 각 태스크의 (period, runtime)은 randfixedsum 알고리즘을 이용해 생성하였고, 전체 시스템 부하는 80~95% 수준으로 유지하였다.

3) 비교 대상 (Baselines): MC-Kube 의 기여도를 명확히 분리하여 입증하기 위해 다음과 같은 네 가지 시스템 구성을 비교한다.

- Vanilla K8s (CFS): 표준 K8s. 실시간 커널 패치나 SCHED_DEADLINE 없이 기본 CFS 스케줄러를 사용하는 하한선(lower-bound) 베이스라인이다.
- RT-kube [5]: SCHED_DEADLINE 을 K8s 에 적용한 대표적인 최신 선행 연구(state-of-the-art)이다. 본 실험에서는 HCBS 커널 상에서 동작하며, API 서버 Patch 를 통한 오버런 관리를 수행한다.
- MC-Kube (HCBS only): MC-Kube 의 핵심 기여도를 분리하여 평가하기 위한 절제 연구(Ablation Study)용 구성이다. HCBS 커널은 적용되었으나, III-C, III-D 절에서 제안한 지능형 오케스트레이션 로직(eBPF 기반 오버런 감지 및 중요도 기반 Escalation)은 비활성화된 상태이다.
- MC-Kube (HCBS + Escalation): 본 논문에서 제안하는 최종 시스템 (Full System)이다. HCBS 커널, eBPF 기반 오버런 관리, 중요도 기반 Escalation 로직이 모두 활성화된 상태이다.

4.2 평가 지표 (Evaluation Metrics)

- Overrun Detection Delay (μs): 실시간 태스크가 runtime을 초과(overrun)했을 때, 시스템이 이를 인지하고 Cgroup 수정과 같은 대응 조치를 완료하기까지 걸리는 평균 시간. 시스템의 반응성과 효율성을 나타낸다.

- Missed Deadlines: 10,000번의 주기 중 태스크가 데드라인을 준수하지 못한 총 횟수의 평

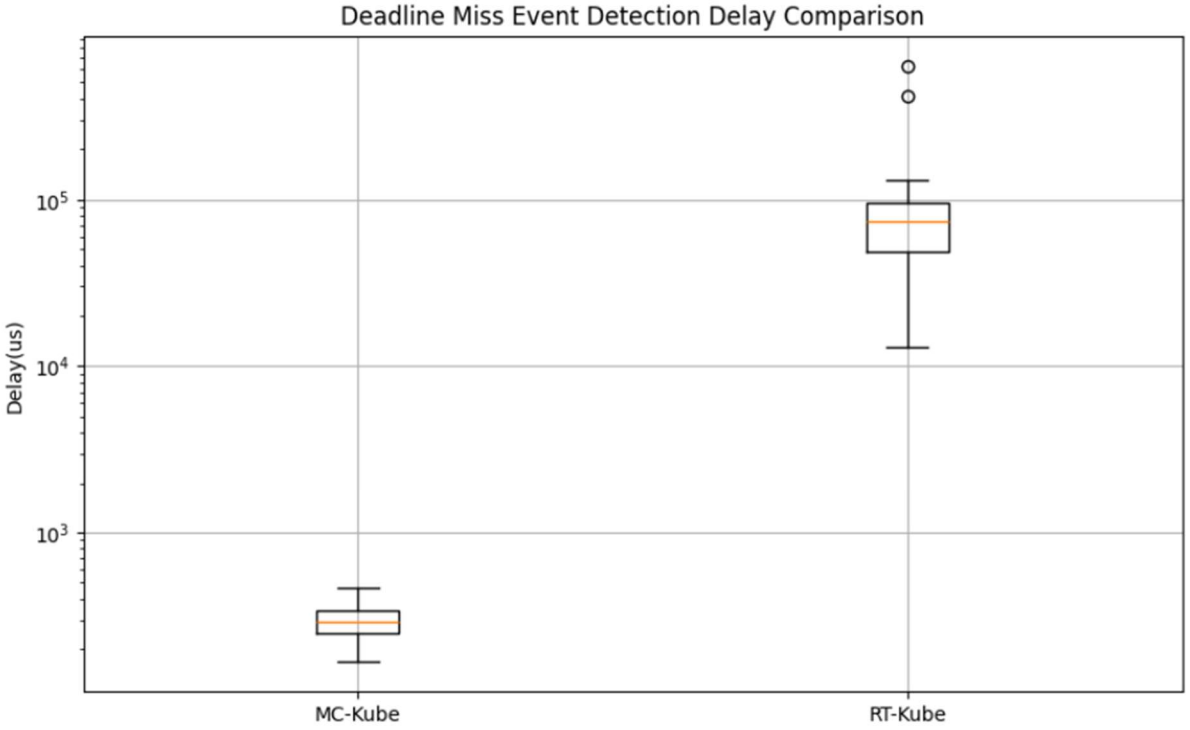
균값. 실시간 성능(correctness)을 측정하는 핵심 지표이다.

- Response Time (ms): 태스크 완료 시간. Max(최대)와 Std dev(표준 편차)를 통해 시스템의 안정성(stability)과 예측 가능성(Jitter)을 평가한다.

4.3 Result 1: System Efficiency (Overrun Detection Latency)

1) Objective: 본 실험은 MC-Kube 의 eBPF 기반 오버런 관리 메커니즘(III-D 절)이 기존 RT-kube 의 API 서버 방식 대비 갖는 효율성 차이를 검증한다.

2) Methodology: 태스크가 할당된 runtime 을 초과(overrun)한 시점부터, 각 시스템의 에이전트(Resource-controller 또는 RT-kube 에이전트)가 이를 인지하고 Cgroup 파라미터 수정과 같은 대응 조치를 완료하기까지 걸리는 평균 시간(μs)을 측정하였다.



[Fig. 8] 데드라인 미스 이벤트 감지 지연 시간 비교 (Log Scale)

Deadline Miss 관측 delay	Avg latency(us)	ΔCPU%
RT-kube [5]	73,770	<0.1%

MC-Kube	299	<0.1%

[Table 3] 데드라인 미스 이벤트 감지 지연 시간 및 CPU 사용량 비교

3) Results and Analysis: Fig. 8 은 로그 스케일(Log Scale)로 표현된 두 시스템 간의 지연 시간 차이를 보여준다. RT-kube 는 커널 이벤트를 API 서버를 경유해 처리하는 피드백 루프로 인해, 지연 시간의 중앙값(median)이 약 70,000μs (평균 73,770μs)정도의 지연 시간을 보이며, 변동성 또한 매우 크다.

이는 10~50ms 내외의 주기를 갖는 많은 실시간 태스크 입장에서, 시스템이 대응하기도 전에 이미 수차례의 데드라인 미스가 발생할 수 있음을 의미한다. 즉, API 서버 기반 방식은 고속 제어 루프에 대응하기에는 구조적 한계가 있다.

반면, MC-Kube 는 eBPF 를 통해 API 서버를 완전히 우회(bypass)하고 커널 이벤트를 노드 내에서 즉각 처리한다. 그 결과, 지연 시간의 중앙값이 약 300μs 수준(평균 299μs)에서 매우 안정적인 지연 시간을 기록하였다. 이는 MC-Kube 가 기존 방식 대비 약 246 배 빠른 속도이며, 99.6%의 지연 시간을 감소시킨 것이다. 이러한 초저지연 특성은 MC-Kube 가 데드라인 미스를 실시간으로 다룰 수 있게 하는 핵심 동인(Enabling Factor)이다.

4.4 Result 2: High-Load Scenario (Stress Test)

1) Objective: 본 실험은 72 개의 컨테이너가 경쟁하는 고부하(Stress) 환경에서 각 시스템의 성능 안정성을 평가한다.

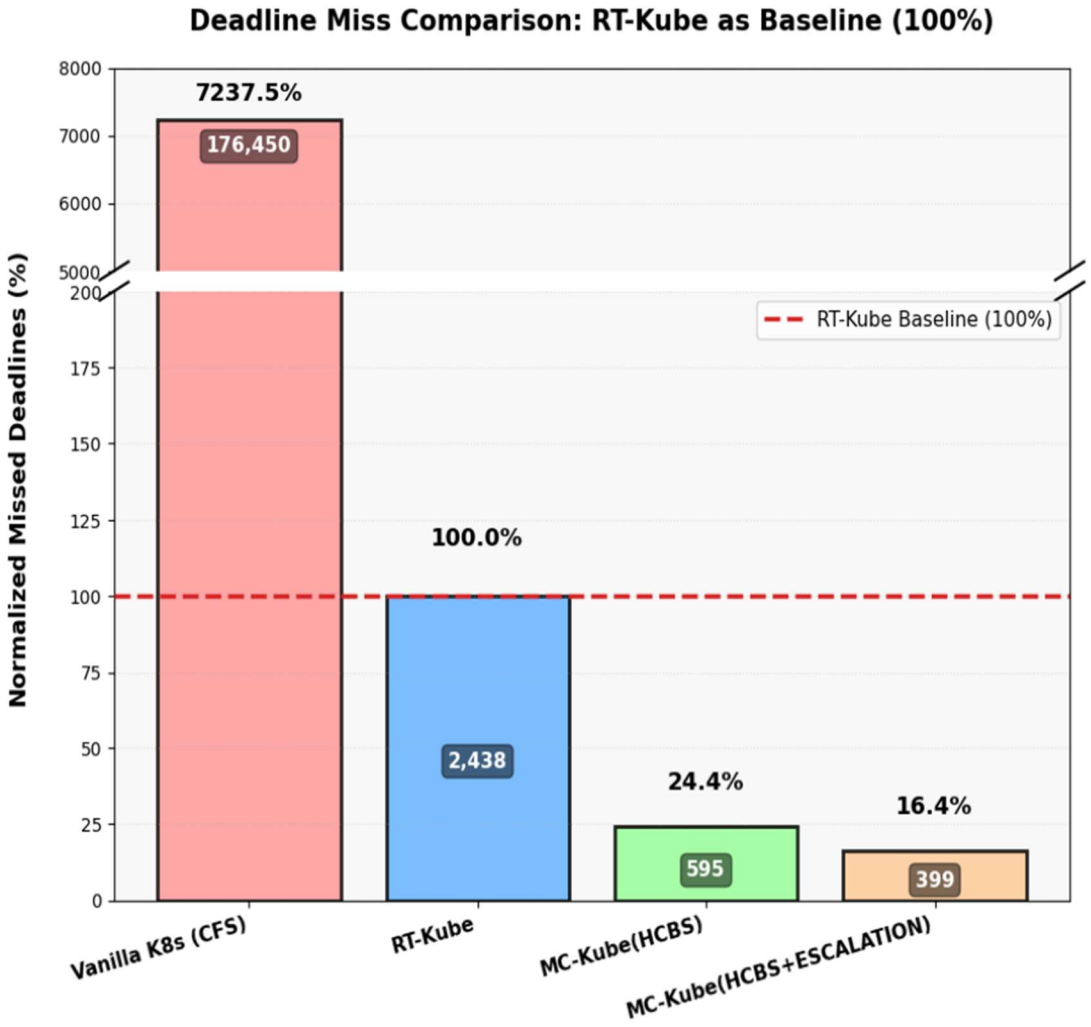
2) Results and Analysis: 전체 실험 결과는 TABLE IV 에 요약되어 있다.

System Configuration	Criticality	Avg Resp. (ms)	Std dev (ms)	Max Resp. (ms)	Missed Deadlines
Vanilla K8s (CFS)	High (16)	19.79	23.04	250	50,503
	Medium (24)	19.95	23.26	270	63,588
	Low (32)	19.87	23.14	296	62,359
RT-kube [5]	High (16)	5.13	0.55	21	143
	Medium	5.09	0.95	26	428

	(24)				
	Low (32)	4.82	1.60	38	1,867
MC-Kube(HCBS only)	High (16)	4.78	0.80	21	108
	Medium (24)	5.25	2.06	30	207
	Low (32)	6.26	3.85	38	280
MC-Kube(Full)	High (16)	4.83	0.70	23	19
	Medium (24)	5.30	2.04	31	70
	Low (32)	6.19	3.70	43	310

[Table 4] HIGH-LOAD (STRESS TEST) SCENARIO 성능 비교

2-1) Deadline Miss Ratio (High-Load): Fig. 9은 TABLE IV의 총 데드라인 미스 횟수(Total Missed Deadlines)를 시각화한 것이다.



[Fig. 9] 총 데드라인 미스 횟수 및 비율 비교 (RT-Kube=100%, High-Load)

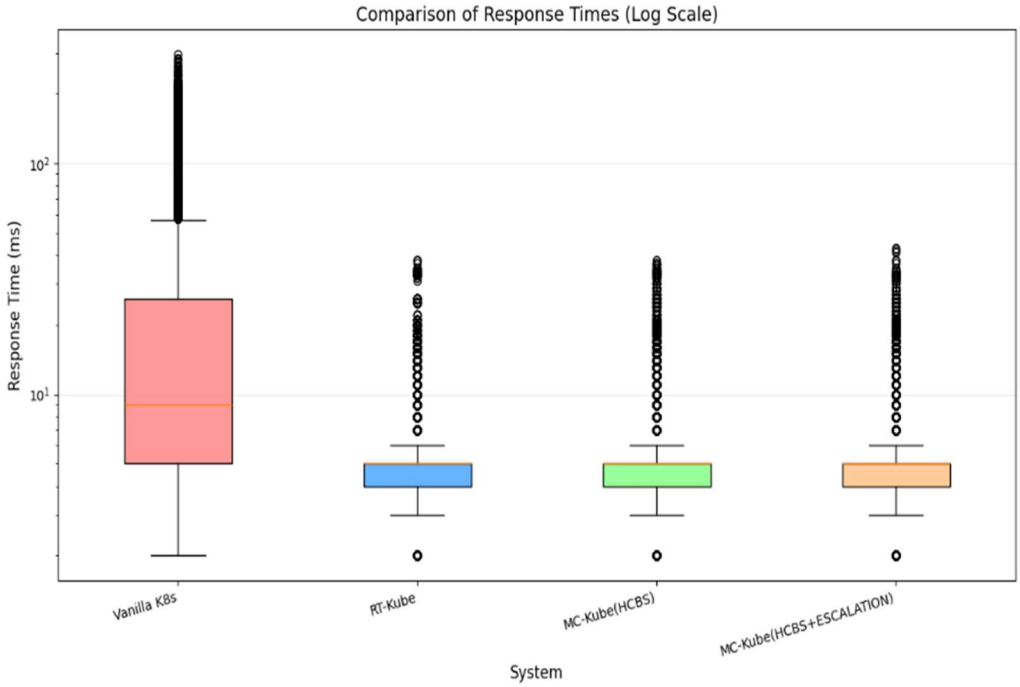
- Vanilla K8s의 한계: Vanilla K8s는 총 176,450회의 미스를 기록하여 실시간 워크로드를 전혀 처리하지 못함을 보였다. 이는 CFS가 공정성(Fairness)을 우선시하기 때문에 발생한 필연적 결과이다.
- RT-kube는 SCHED_DEADLINE 도입을 통해 전체 데드라인 미스를 대폭 감소시켰으나, 여전히 2,438회의 미스를 기록하여 베이스라인을 형성했다.
- Ablation Study (절제 연구):
  - MC-Kube(HCBS)는 커널 패치와 코어 고정(core pinning)만으로도 전체 미스 횟수를 RT-kube 대비 약 75.6% (2,438회 → 595회) 감소시켰다. 이는 HCBS 커널의 계층적 제어 구조가 기본적으로 우수한 성능을 제공함을 시사한다.
  - MC-Kube(HCBS + Escalation)는 eBPF 감지와 Escalation 로직을 더해, 전체 미스 횟수를 399회까지 추가로 낮추었다. 즉, 지능형 오케스트레이션 로직이 커널 패치

이상의 최적화를 이끌어냄을 확인하였다.

- High-Criticality Protection (핵심 성능 분석): 미션 크리티컬 시스템에서 가장 중요한 것은 High-Criticality (HC) 태스크의 무결성이다. TABLE I은 High-Criticality (HC) 태스크 결과이다.
    - RT-kube와 MC-Kube(HCBS)는 각각 143회, 108회의 HC 미스를 기록하여, 고부하 상황에서 중요 태스크를 완벽히 보호하지 못하는 한계를 드러냈다.
    - MC-Kube(HCBS + Escalation)는 eBPF를 통한 즉각적인 오버런 감지와 중요도 기반 자원 재할당(Escalation)을 통해 HC 태스크의 미스를 단 19회로 억제하였다.
    - 이는 제안하는 MC-Kube 시스템이 고부하 환경에서 선행 연구 (RT-kube) 대비 HC 태스크의 데드라인 미스를 86.7% 감소시켰음을 의미한다. 결론적으로, 본 논문이 제안하는 Escalation 알고리즘이 시스템 과부하 상황에서도 낮은 중요도(LC) 태스크를 희생하여 높은 중요도(HC) 태스크의 자원을 성공적으로 확보했음을 명확히 증명한다.

2-2) Response Time Stability (High-Load):

Fig. 10 은 TABLE IV 의 응답 시간 (Response Time) 분포를 로그 스케일의 Box Plot 으로 시각화한 것이다.



[Fig. 10] 응답 시간 분포 비교 (Log Scale, High-Load)

- Jitter (변동성) 제어: Vanilla K8s는 응답 시간의 분포가 극단적으로 넓고 (Std dev:

23.04ms), 100ms를 초과하는 수많은 이상치(outlier)를 보이며 (Max: 250ms), 완전한 예측 불가능성을 나타낸다.

- SCHED_DEADLINE 시스템의 안정성: 반면, RT-Kube와 MC-Kube 시스템들은 모두 매우 낮고 좁은 박스(Box)를 보여주며, 대부분의 응답 시간이 10ms 이하에서 안정적으로 처리됨을 시사한다. TABLE I의 Std dev 수치(0.55ms ~ 0.80ms)가 이를 뒷받침한다.
- 핵심 결론: Fig. 9와 Fig. 10을 종합하면, MC-Kube는 RT-Kube와 대등하거나 더 우수한 안정성(Low Jitter)을 유지하면서도, 데드라인 미스율은 획기적으로 감소시켰다. 이는 Escalation과 같은 동적 오케스트레이션 로직이 시스템 안정성을 해치지 않으면서도 성능을 극대화했음을 의미한다.

5 결론 (Conclusion)

5.1 평가 지표 (Evaluation Metrics)

최근 SDV 및 로보틱스 분야에서 컨테이너 기술의 도입이 가속화되고 있으나, 표준 K8s 는 실시간성과 혼합 중요도 관리에 한계를 보인다. 본 논문은 이를 해결하기 위해 MCS 이론을 오케스트레이션 계층으로 확장한 MC-Kube 를 제안하였다. 본 연구의 핵심 기여는 다음과 같다.

1. 중요도 인지형 오케스트레이션 (McKube Custom Controller): K8s 상에서 MCS 정책을 실행하기 위한 핵심 두뇌(brain)로서 McKube Custom Controller를 설계 및 구현하였다. McKube CRD 를 통해 사용자가 컨테이너의 중요도(Criticality), 주기(Period), 실행 시간(Runtime)을 선언적으로 정의할 수 있게 하였으며, 컨트롤러는 이를 기반으로 시스템 전역의 중요도 정책을 관장하는 정책 엔진으로 동작한다.

2. Two-Level 스케줄링 실행 (Resource Controller): 중앙의 Custom Controller 가 수립한 정책을 각 노드에서 실질적으로 집행하는 Resource Controller 를 구현하였다. 이는 HCBS 커널의 cpu.rt_multi_runtime_us 인터페이스와 연동하여 컨테이너 단위로 SCHED_DEADLINE 파라미터를 동적으로 제어함으로써 정밀하고 계층적인 실시간성 보장을 실현한다.

3. eBPF 기반 초저지연 오버런 감지: 기존 K8s 의 API 서버를 경유하는 방식이 갖는 근본적인 지연 문제를 해결하기 위해, eBPF 를 메커니즘을 도입하여, 커널 레벨에서 컨테이너의 오버런 이벤트를 직접 감지하였다. 감지된 이벤트는 API 서버를 거치지 않고 McKube Custom Controller 및 Resource Controller 로 직접 전달되므로, 통신 지연 없는 즉각적인 대응(Escalation)이 가능하다.

5.2 성능 평가 결과 및 시사점

정량적 성능 평가는 MC-Kube 시스템의 실질적인 우수성을 입증하였다. (IV-D)

첫째, 시스템 효율성(Efficiency) 측면에서 MC-Kube 의 eBPF 기반 오버런 감지 지연(평균 299μs)은 기존 RT-kube 의 API 서버 방식(평균 73,770μs) 대비 약 246 배 더 빠르거나 99.6%의 지연 시간을 감소시켰다. 이는 73.8ms 에 달하는 RT-kube 의 지연이 많은 태스크의 주기(period)보다 길어 사실상 실시간 대응에 실패하는 반면, MC-Kube 는 즉각적인 대응이 가능함을 시사한다.

둘째, 실시간성(Correctness) 및 안정성(Stability) 측면에서 MC-Kube 는 80~95%의 극한 고부하 환경에서도 뛰어난 성능을 보였다. Vanilla K8s(CFS)가 17 만 회 이상의 데드라인 미스로 완전히 붕괴한 반면, MC-Kube(Full)는 선행 연구 RT-kube 대비 High-Criticality 태스크의 데드라인 미스를 86.7% (143 회 → 19 회) 획기적으로 감소시켰다.

특히, 절제 연구(Ablation Study)를 통해 MC-Kube(HCBS only) 대비 MC-Kube(Full)의 성능 향상을 입증함으로써, 이러한 성과가 단순한 커널 패치(HCBS)의 효과가 아닌, 본 논문이 제안하는 eBPF 감지 메커니즘과 두 컨트롤러(Custom/Resource)의 유기적인 결합을 통해 달성되었음을 명확히 증명하였다.

5.3 한계 및 향후 연구

본 연구는 MC-Kube 의 핵심 성능을 입증하였으나 다음과 같은 한계점을 가지며, 이는 향후 연구 방향이 된다.

다중 노드 클러스터로의 확장: 현재 평가는 단일 노드 환경에서 집중적으로 수행되었다. 실제 K8s 환경은 다중 노드로 구성되므로, 향후 연구에서는 네트워크 지연 시간을 고려한 클러스터 레벨의 중요도 기반 배치(Placement) 및 노드 간(inter-node) 마이그레이션 정책을 연구할 것이다.

I/O 및 다중 스레드 워크로드 고려: 본 논문은 CPU 중심의 단일 스레드 태스크를 가정하였다. 그러나 실제 SDV 환경에서는 카메라 센서(I/O-bound)나 복잡한 연산(multi-threaded) 워크로드가 혼재한다. I/O 대기(blocking)가 SCHED_DEADLINE 에 미치는 영향을 분석하고, 이를 오케스트레이션에 반영하는 연구가 필요하다.

지능형 오케스트레이션 고도화: 현재의 Escalation 로직을 넘어, 클러스터의 전역 상태(global state), 태스크 간 의존성, 네트워크 토폴로지 및 에너지 효율성까지 고려하는

적응형(adaptive) 오케스트레이션 알고리즘을 개발하여 시스템 전체의 효율성을 최적화할 계획이다.

6 참고문헌

[1] M. Reke et al., "A Self-Driving Car Architecture in ROS2," in 2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring), 2020, pp. 1–5.
[2] Y. Zhang, C. Wurll, and B. Hein, "KubeROS: A unified platform for automated and scalable deployment of ROS2-based multi-robot applications," in 2023 IEEE International Conference on Robotics and Automation (ICRA), 2023, pp. 10791–10797.
[3] B. Lampe et al., "RobotKube: Orchestrating Large-Scale Cooperative Multi-Robot Systems with Kubernetes and ROS," in 2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC), 2023, pp. 2862–2869.
[4] P. Laclau, S. Bonnet, B. Ducourthial, X. Li, and T. Lin, "Enhancing Automotive User Experience With Dynamic Service Orchestration for Software Defined Vehicles," IEEE Transactions on Intelligent Transportation Systems, vol. 26, no. 1, pp. 824–838, 2025.
[5] E. Carmona Cejudo and M. S. Siddiqui, "An optimization framework for edge-to-cloud offloading of kubernetes pods in V2X scenarios," in 2021 IEEE Globecom Workshops (GC Wkshps), 2021, pp. 1–6.
[6] T. Betz et al., "A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation," in 2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2024, pp. 1–10.
[7] Y. Wang and Q. Bao, "Adapting a Container Infrastructure for Autonomous Vehicle Development," in 2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE), 2019, pp. 1–6.
[8] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis, "Latency analysis of ROS2 multi-node systems," in 2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), 2021, pp. 1–7.
[9] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Containerization and Orchestration of Software for Autonomous Mobile Robots: a Case Study of Mixed-Criticality Tasks across Edge-Cloud Computing Platforms," in 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2022, pp. 11224–11231.
[10] F. Lumpp, F. Fummi, and N. Bombieri, "Optimizing Kubernetes Deployment of Robotic Applications with HEFT-based Container Orchestration," in 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2024, pp. 1–6.
[11] M. Cinque and G. De Tommasi, "Work In Progress: Real-Time Containers for Large-Scale Mixed-Criticality Systems," in 2017 IEEE Real-Time Systems Symposium (RTSS), 2017, pp. 1–4.
[12] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the Linux kernel," in Proceedings of the Embedded Operating Systems Workshop (EWILI), 2018, pp. 1–6.
[13] S. Fiori, L. Abeni, and T. Cucinotta, "RT-Kubernetes: Containerized real-time cloud computing," in Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022, pp. 1449–1452.
[14] N. Samimi et al., "Enabling Containerisation of Distributed Applications with Real-Time Constraints," in 37th Euromicro Conference on Real-Time Systems (ECRTS 2025), vol. 335, 2025, pp. 3:1–3:29.
[15] H. Lee, Y. Choi, T. Han, and K. Kim, "Probabilistically guaranteeing end-to-end latencies in autonomous vehicle computing systems," IEEE Transactions on Computers, vol. 71, no. 12, pp. 3361–3374, 2022.
[16] N. Lee, S. Hong, and S. Kim, "Dynamic Mapping of Mixed-Criticality Applications onto a Mixed-Criticality Runtime System with Probabilistic Guarantees," in 2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS), 2024, pp. 1–6.
[17] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots," IEEE Transactions on Robotics, vol. 40, pp. 540–553, 2024.
[18] D. Casini, P. Pazzaglia, and M. Becker, "Managing real-time constraints through monitoring and analysis-driven edge orchestration," Journal of Systems Architecture, vol. 163, p. 103403, 2025.

[19] SOAFEE Special Interest Group, "Scalable Open Architecture for Embedded Edge (SOAFEE)," 2025. [Online]. Available: https://soafee.io.
[20] Eclipse Foundation, "Eclipse Software Defined Vehicle (SDV) Working Group," 2025. [Online]. Available: https://sdv.eclipse.org.
[21] Eclipse Pullpiri Project, "Eclipse Pullpiri: Vehicle Service Orchestrator Framework," 2025. [Online]. Available: https://github.com/eclipse-pullpiri/pullpiri.
[22] Alan Burns and Robert I. Davis. 2017. A Survey of Research into Mixed Criticality Systems. ACM Comput. Surv. 50, 6, Article 82 (November 2018), 37 pages. https://doi.org/10.1145/3131347
[23] Abeni, Luca & Balsini, Alessio & Cucinotta, Tommaso. (2019). Container-based real-time scheduling in the Linux kernel. ACM SIGBED Review. 16. 33-38. 10.1145/3373400.3373405.
[24] K. Baruah and S. Vestal. 2008. Schedulability analysis of sporadic tasks with multiple criticality specifications. In Proc.ECRTS. 147–155.