CMS: Memory Fabric Orchestration Architecture

Version 1.0
Date: 12/19/2025

Author (s):
Adam Manzanares, Samsung Semiconductor Inc.
Anjaneya “Reddy” Chagam, Intel Corporation
Grant Mackey, Jackrabbit Labs
Hongjian Fan, Seagate Technology, LLC
Manoj Wadekar, Meta
Navneet Rao, Altera
Pankaj Mehra, The Ohio State University and Elephance Memory
Sakul Gupta, Micron
Siamak Tavallaei, Samsung Semiconductor Inc.

Executive Summary

This white paper presents the Open Compute Project (OCP) Composable Memory Systems (CMS) **Memory Fabric Orchestration (MFO) Logical Architecture** - a comprehensive framework designed to address the evolving demands of modern data center infrastructure. Composable Memory Systems represent an emerging paradigm that enables dynamic and unified memory management across heterogeneous environments, seamlessly integrating diverse memory technologies, device types, interconnects, and fabric hierarchies into a cohesive, manageable system.

This document serves as an authoritative reference for key stakeholders in the data center ecosystem, including system architects, data center operators, infrastructure teams, and enterprise customers evaluating memory solutions. The framework provides essential guidance for selecting optimal combinations of host processor platforms, memory devices, memory fabrics, chassis configurations, and rack deployments. This CMS Memory Fabric Orchestration architecture demonstrates a practical implementation featuring Kubernetes orchestration with general-purpose applications running on CPU infrastructure, utilizing Compute Express Link (CXL®) interconnects to enable pooled memory use cases as a reference implementation [1].

This white paper is designed as a living document that will evolve to incorporate emerging technologies, advanced fabric architectures, and next-generation use cases, including memory sharing capabilities and artificial intelligence workloads.

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Table of Contents

1. Introduction 6
- 1.1 Market Drivers and Memory Challenges 6
- 1.2 System Design Imperatives 6
- 1.3 Composable Memory Systems Overview 6

2. Compliance with Open Compute Project Tenets 7
- 2.1 Openness 7
- 2.2 Efficiency 7
- 2.3 Impact 7
- 2.4 Scale 7
- 2.5 Sustainability 7

3. CMS Memory Fabric Logical System Architecture 8
- 3.1 Direct-Attached Memory Expansion 9
- 3.2 Multi-Headed Memory 10
- 3.3 Alternative Memory Transports 11
- 3.4 Fabric Attached Memory 11
- 3.5 Bridges between Native Scale-up and Memory Fabrics 12
- 3.6 Memory Fabric Manager (MFM) 15
  - 3.6.1 Architectural Role and Purpose 15
  - 3.6.2 Dynamic Resource Management 15
  - 3.6.3 Security and Monitoring Capabilities 15
  - 3.6.4 Agent-Based Communication Architecture 15
  - 3.6.5 High Availability and Integration 16

4. Datacenter Fabric Management Interfaces 16
- 4.1 Redfish model for CMS memory fabric logical system 17
  - 4.1.1 Redfish resource mapping 18
  - 4.1.2 Redfish model chart examples 20
- 4.2 DMTF Redfish Interfaces for Memory Life-Cycle Management 22
  - 4.2.1 View Host server info 24

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

4.2.2 View switch and its attached Memory device info 24
4.2.3 View logical device info 24
4.2.4 Manage Memory (de-)Provisioning 25
4.2.5 Manage Memory Binding for VHS 25
5. OS Sub System Overview – Linux and CXL 25
5.1 Driver and Hardware Interface Layer 26
5.2 OS Memory Management Integration 26
5.3 Application Layer and Memory Consumption 27
5.4 Dynamic Memory Management in Production 27
5.4.1 Memory Addition and Integration 27
5.4.2 Memory Removal and Safety 27
5.5 Fabric Management Coordination 28
5.6 Platform Integration and Compatibility 28
6. QEMU Overview for CXL Memory and Fabrics 28
6.1 Planned Enhancements 29
7. Kubernetes Integration with CMS MFM 30
7.1 Kubernetes Control Plane Components 31
7.2 Kubernetes Data Plane Components 31
7.3 Kubernetes Add-On and Extension Components 32
7.4 MFM Implementation Details for CXL Pooled Resources 32
7.5 Future Milestones and Enhancements for Composable Memory in Kubernetes 36
8. Future Directions for Disaggregated Memory Systems 38
9. Conclusion 38
10. Glossary & Appendix 40
10.1 Glossary 40
10.2 References 44
11. License 44
12. About Open Compute Foundation 46

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

1. Introduction

1.1 Market Drivers and Memory Challenges

The enterprise and hyperscale computing markets are experiencing unprecedented growth, positioning memory as a critical performance bottleneck[1]. Memory-intensive applications—including Artificial Intelligence (AI), Machine-Learning (ML), and high-performance databases—are driving computational demands that exceed the capabilities of traditional server architectures.

To address these escalating requirements, the industry is rapidly innovating on the memory technology stack. Advanced semiconductor devices, optimized memory interfaces, high-performance interconnects, and sophisticated memory hierarchies are being developed to deliver enhanced bandwidth, increased capacity, and improved system-wide performance and scalability.

1.2 System Design Imperatives

System architects are actively pursuing balanced solutions that harness these emerging technologies alongside enabling software frameworks. The dual pressures of deeper data processing requirements and exponentially growing computational performance are straining memory performance limits, while expanding data volumes and model sizes are challenging system memory capacity boundaries.

These evolving hardware capabilities, combined with new software abstractions and system architectures, promise to unlock significant technical and economic value for memory-intensive workloads. Key innovations include location-transparent memory access, automated tiering across hybrid memory technologies, and pooled memory capacities managed through a unified control plane abstraction as a major focus of this white paper.

1.3 Composable Memory Systems Overview

Composable Memory Systems (CMS) represents an emerging paradigm that enables dynamic and unified memory management across heterogeneous memory technologies, interconnects, and hierarchical structures. This approach addresses the complex performance, capacity, latency, throughput, data sharing, scalability, and security requirements of next-generation applications—all without requiring system reboots or service interruptions.

[1] Memory roofline dominates AI performance https://www.rambus.com/blogs/memory-systems-for-ai-part-3/ and *It's the Memory, Stupid!* in http://cva.stanford.edu/classes/cs99s/papers/architects_look_to_future.pdf

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

The CMS logical system architecture provides essential guidance for system designers, data center operators, and enterprise customers navigating the landscape of commercial memory solutions. This framework assists in selecting optimal combinations of host processor platforms, memory devices, chassis configurations, and rack deployments. The initial CMS logical architecture focuses on general-purpose applications running on CPU platforms and leverages CXL interconnects to enable advanced pooled memory use cases.

2. Compliance with Open Compute Project Tenets

2.1 Openness

The OCP CMS Memory Fabric Architecture was developed in dialogue with the entire OCP CMS community membership and the resulting document represents the initial logical architecture. CMS usage models are nascent, so this document will evolve over time, and this architecture document represents the wisdom of the collective ecosystem on the deployment scenarios for creating CXL based composable memory systems.

2.2 Efficiency

The OCP CMS Memory Fabric Architecture lays the foundation for developing OCP vendor reference architectures for various deployments highlighting hardware/software co-designs.

2.3 Impact

Establishing a unified foundational logical architecture enables architectural consistency across future white papers, specifications, case studies, and vendor reference implementations.

2.4 Scale

The CMS Memory Fabric Orchestration Architecture promotes vendor-neutral composable solutions while enabling extensive reusability through open ecosystem software, standardized hardware form factors, and industry best practices.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

2.5 Sustainability

CMS software and hardware co-design solutions are designed to optimize performance and resource utilization for underlying physical memory technologies thereby achieving data center energy efficiency.

3. CMS Memory Fabric Logical System Architecture

The foundational principle guiding this architecture is minimal disruption to existing infrastructure, ensuring that composable memory systems integrate seamlessly without compromising established operational workflows. The logical architecture incorporates essential abstractions and interfaces between hardware and software components, enabling advanced composability functionality while preserving existing control flows in deployed infrastructure environments. This section establishes a reference framework for composable memory fabric logical architecture, leveraging Compute Express Link (CXL) standards and protocols as the foundational technology.

CXL is an open, industry-standard, cache-coherent interconnect protocol designed to enable seamless communication between processors, memory expansion devices, and specialized accelerators. In this white paper, Composable Memory Systems (CMS) takes advantage of two essential capabilities from the CXL specification: the CXL.mem protocol enables memory expansion using native Load/Store semantics, providing direct integration with host systems and underlying operating system or hypervisor runtimes; while the CXL.io protocol delivers comprehensive device and fabric control plane management capabilities, facilitating integration with data center orchestration platforms for enterprise fleet management. In a later version of this white paper, we will explore how CXL.cache, the third CXL protocol, will advance accelerators' direct access to host memory for a converged memory experience.

The CMS Logical Architecture presented in Figure 2 defines five distinct composable system configurations that leverage CXL.mem data-plane capabilities and CXL control-plane abstractions. These architectural patterns address diverse deployment scenarios and performance requirements: Direct-Connected Host Solutions with single and multi-headed device configurations using direct host attachment; Fabric-Connected Host Solutions featuring single and multi-headed devices accessed through fabric infrastructure; System Network Configurations enabling distributed memory access across network-connected resources; and Alternative Transport Solutions supporting direct-attached devices utilizing non-CXL/PCIe® physical layer transports. Each configuration is

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

optimized for specific use cases, performance characteristics, and operational requirements, providing flexibility in composable memory system deployment strategies.

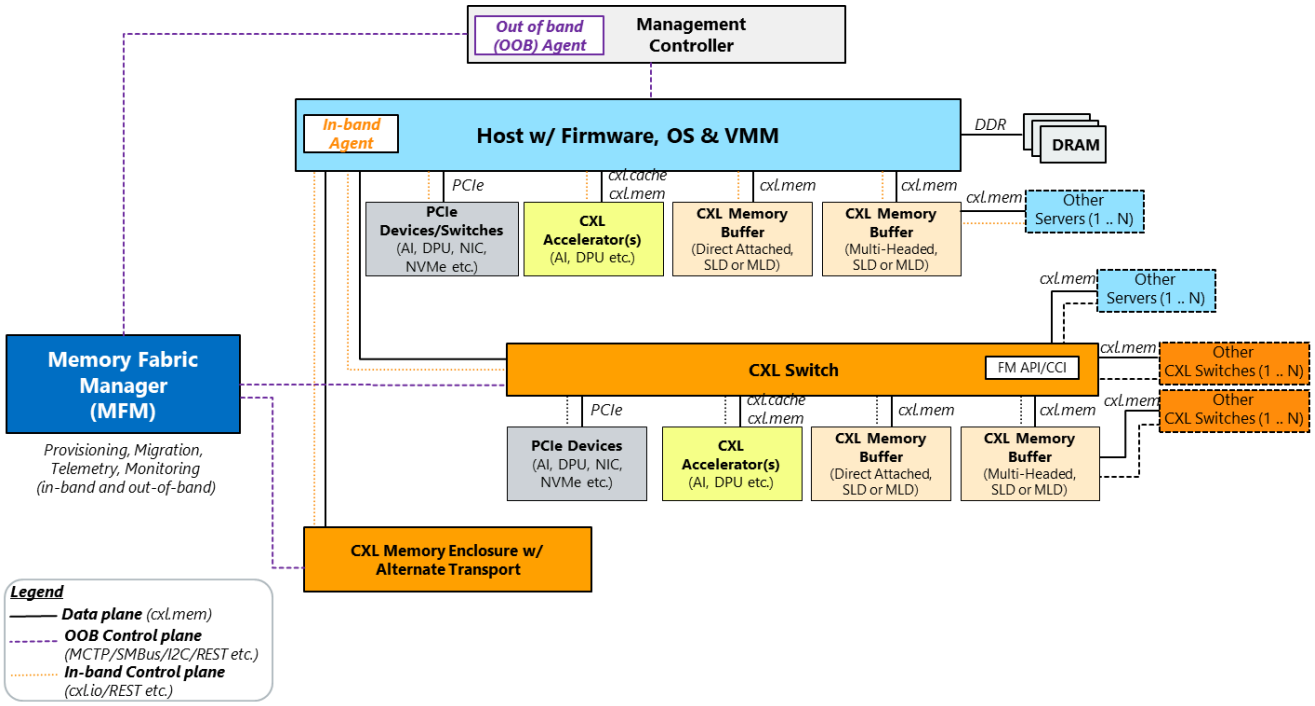


Figure 1: CMS Memory Fabric Orchestration Logical System Architecture

The following subsections provide detailed description of the functional layers within the CMS Fabric Orchestration Management (FOM) logical architecture outlined in above picture, progressing from direct-attached single devices through Data Center Memory Fabric Management (MFM) and host operating system interactions.

3.1 Direct-Attached Memory Expansion

Direct-attached memory expansion establishes the fundamental composable memory configuration, featuring CXL devices that connect directly to host platforms for exclusive use by that specific host (sometimes on behalf of remote initiators). This approach implements CXL Type 3 devices (memory buffers) according to specifications outlined in Section 2.3 of the Compute Express Link Specification Revision 3.2, Version 1.0 [12].

CXL memory buffers may accommodate various memory technologies including DDR4, DDR5, High-Bandwidth Memory (HBM), and Storage Class Memory (SCM). Management implementation

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

offers flexibility through in-band agents operating within host systems or out-of-band protocols using dedicated management controllers, allowing system vendors to select the most appropriate approach for their specific requirements.

CXL memory buffers present **H**ost-managed **D**evice **M**emory (HDM) that host firmware integrates into the system's coherent address space, enabling standard write-back memory semantics for host access. HDM delivers significant flexibility through partitioning functionality, allowing memory division into multiple isolated Logical Devices where each partition becomes accessible through virtual hierarchies as specified in Section 2.4 of the CXL Specification. Host firmware can configure HDM memory for either special-purpose applications requiring dedicated resources or conventional memory supporting general system operations.

The operating system manages CXL HDM regions as zero-core, memory-only NUMA domains. During system initialization, the OS kernel establishes special-purpose CXL HDM memory as Direct Access (DAX) devices, with DAX utilities providing dynamic memory promotion and demotion capabilities between kernel-managed and application-managed memory pools after boot completion [8]. Operating systems and applications utilize CXL memory's capacity, bandwidth, and latency properties to develop intelligent tiering algorithms that enhance data movement efficiency between host and CXL memory regions.

3.2 Multi-Headed Memory

Multi-headed CXL buffers establish physical connections to multiple host systems, facilitating advanced memory pooling and shared resource utilization across distributed computing environments. This architecture utilizes CXL's logical abstractions to enable dynamic memory allocation that responds to real-time workload demands and computational requirements.

When memory regions become accessible to multiple hosts simultaneously, cache coherency management requires coordinated responsibility across host systems, software layers, and associated hardware devices. This collaborative framework maintains data integrity and consistency throughout multi-host access scenarios, ensuring reliable operation across distributed computing environments.

The multi-headed logical abstraction adapts effectively to alternative transport implementations, delivering consistent functionality across various physical layer technologies while preserving identical management and operational characteristics. This flexibility allows organizations to deploy

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

multi-headed configurations using diverse transport mechanisms without compromising system behavior or administrative processes.

3.3 Alternative Memory Transports

This configuration expands the direct-attached logical architecture to accommodate transport mechanisms that extend beyond traditional CXL/PCIe physical layers. Alternative transport options encompass Universal Chiplet Interconnect Express™ (UCIe™) for chiplet-based memory expansion implementations, photonic memory extension technologies that enable flexible physical placement through **extended** distance capabilities, memory **expansion** solutions supporting diverse media types with integrated tiering functionality, and memory pooling and sharing architectures for distributed resource utilization.

Non-CXL memory buffers employ identical management paradigms as standard CXL devices, maintaining support for both in-band and out-of-band management methodologies. These alternative transport devices present Host-managed Device Memory (HDM) that host firmware integrates into the coherent address space, preserving standard write-back memory semantics while accommodating both special-purpose application requirements and conventional memory configurations for general system use.

3.4 Fabric Attached Memory

Fabric-connected systems distinguish themselves from previous deployment models by positioning CXL memory buffers downstream from CXL switching infrastructure, establishing a disaggregated memory architecture. CXL switches function as intermediaries connecting multiple host systems while facilitating dynamic memory resource provisioning across the connected infrastructure.

Memory allocation operates dynamically with resources provisioned to hosts according to real-time computational demands. CXL switches leverage Fabric Management APIs to coordinate physical memory mapping and control host access to CXL devices through centralized Fabric Manager supervision. This coordinated control mechanism enables responsive resource distribution that adjusts to changing workload requirements.

CXL fabric architectures accommodate varying complexity levels, as demonstrated in the accompanying figure. Implementations span from simple single-switch configurations serving

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

multiple hosts to complex multi-tier hierarchical fabric topologies. This deployment strategy provides pooling and sharing capabilities similar to multi-headed configurations while delivering enhanced scalability through improved fan-out capacity and expanded host connectivity options.

Fabric management functionality supports multiple implementation approaches: switch-integrated management embeds fabric management capabilities directly within CXL switch hardware, dedicated management controllers provide standalone fabric oversight functions, and distributed software agents deploy fabric management as software services on orchestrator host systems.

In multi-rack data center environments, fabric managers facilitate sophisticated workload-driven memory provisioning by integrating with distributed CXL fabric management agents throughout the facility. This distributed coordination strategy optimizes resource utilization and ensures responsive memory allocation across data center infrastructure, supporting large-scale composable memory deployments that dynamically adapt to cloud workload patterns.

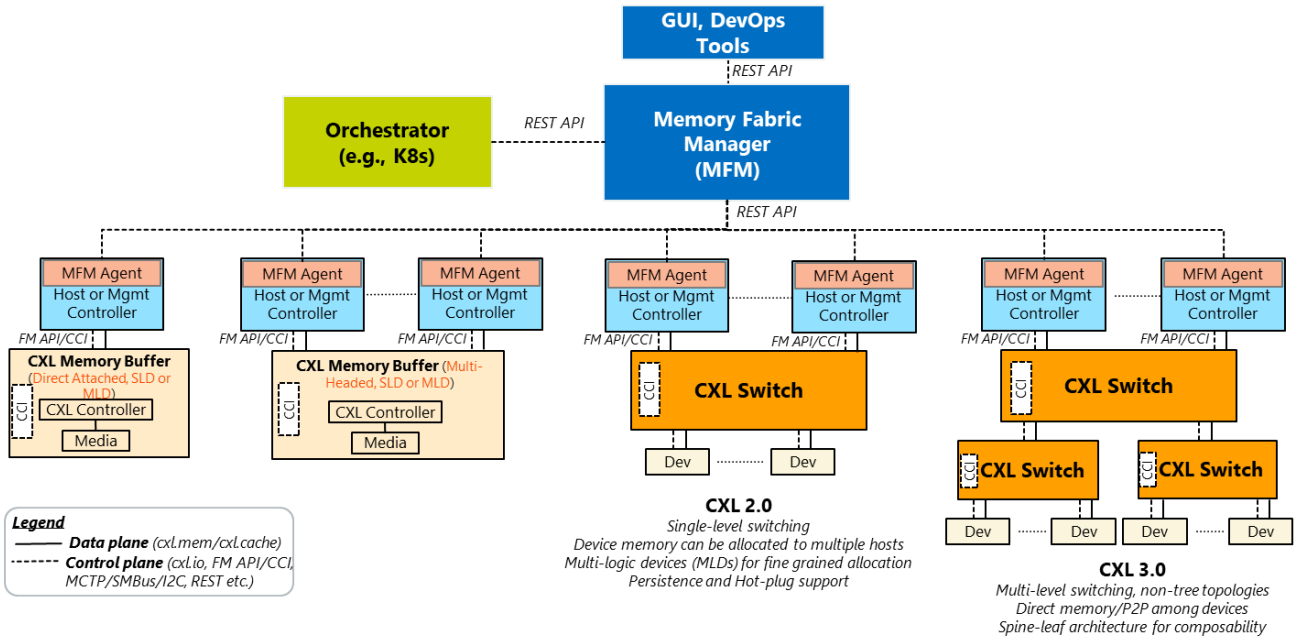


Figure 2: Fabric-Attached Memory (Sample deployment models with CXL)

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

3.5 Bridges between Native Scale-up and Memory Fabrics

Previous overview sections outlined several methods for interconnecting CPUs to memory components via buffers, memory controllers, and fanout switches to increase the available memory footprint for the operating systems and applications.
Traditional **scale-up** fabrics couple CPUs to each other to build large symmetric multi-processing systems (**SMP**) to run large, **parallel processing** workloads under one operating system (OS). Such workloads require the SMP fabric to provide efficient protocols for **Load/Store** semantics and to maintain memory/**cache-coherence** in hardware with low-latency for small payloads.

CXL is an example of a standard interconnect that aims to fulfill these requirements. While Gen-Z, OpenCAPI, and CCIX organizations have transferred their specifications to CXL Consortium, major CPU manufacturers offer Root Ports which natively support CXL protocol over PCIe physical interface to couple to end devices. These factors have fueled the computer industry businesses to build and offer several CXL-enabled memory controllers and solutions.

Emerging scale-up fabrics for AI/ML natively couple xPUs to each other and to switches which use the same protocol used for **native** xPUs. Grown out of the **distributed processing** paradigm, these workloads demand better memory response times than NVMe®/SSDs offer, but they do not require the hardware-based cache-coherence or the low-latency features that CXL specification offers. Instead, AI/ML workloads require high-throughput interconnects for their **emerging** software to **move data ahead of use** (to hide data-movement overhead) via software methods for memory/cache-coherence.

The flexibility of these new software models afford the emerging AI/ML workloads not to require the full potential of the CXL interconnect protocol for efficient data-movement via hardware-based cache-coherence. Yet, the fact that the industry is producing standardized, commercially-available, CXL-connected memory controllers for high-speed memory such as DRAM enables a **hybrid** usage model.

Compelling fabric topologies emerge to "**bridge**" between **native** scale-up fabric protocols (xLink used by xPUs) and the **standard** CXL protocol for scaling-up disaggregated memory pooling and sharing.

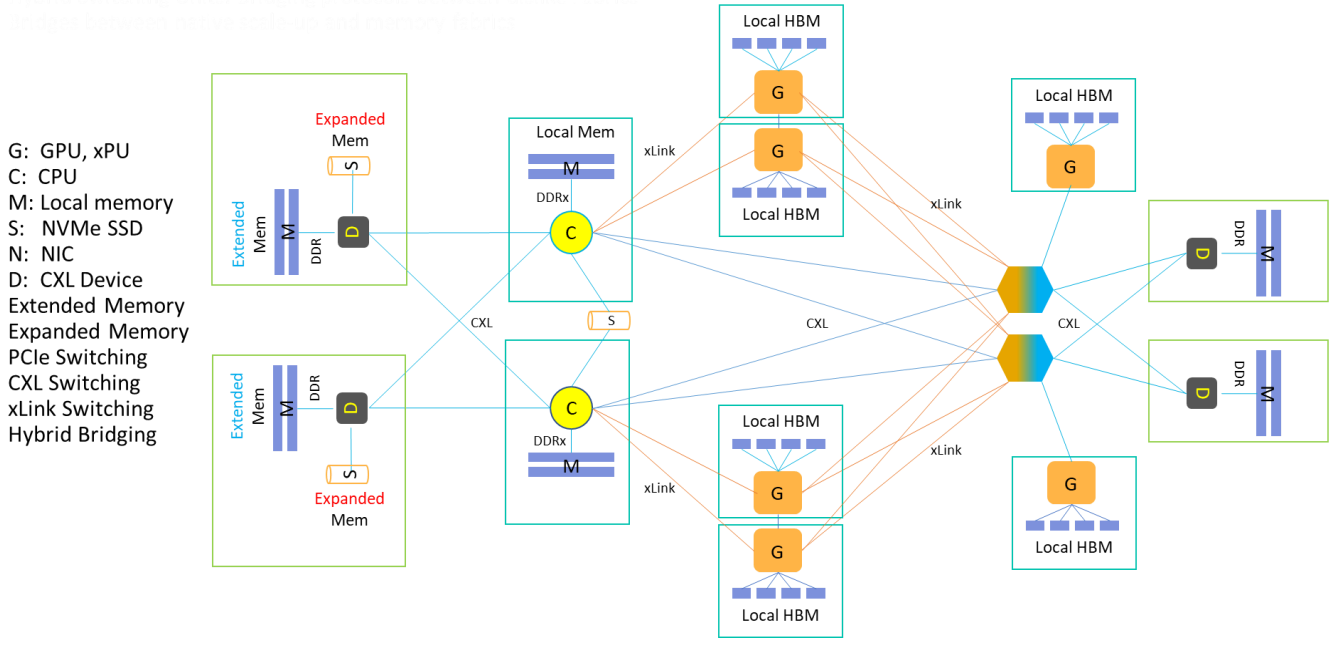
Using dual-ported devices, the following figure illustrates a topology with several memory attach points, memory pooling/sharing at different tiers, **switches** for coupling fabrics running **the same**

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

protocol, and **bridges** for coupling fabrics running **different protocols**. (Consider multi-ported devices and switches for a generalized case.)

In this model, **Hybrid Switching Units** (dual-colored hexagon) bridge interconnect protocols between dislike Fabrics to enable scale-up using native interconnects and protocols plus pooling/sharing using Fabric techniques for Memory Fabrics.

The following figure further shows several memory attach points such as Local HBM to GPU, Local DDRx memory to CPUs, CXL-connected **Extended** DDRx Memory attached to CPUs, **Expanded** Memory tier offered by a CXL Device via a PCIe/NVMe port, pooled memory provided via multi-ported CXL memory controllers for multiple CPUs, and CXL-connected DDRx memory coupled to CPUs via a CXL Switch (blue links). This diagram also shows how xPUs scale up via their native xLink protocol when they couple to each other, to CPUs, and to xLink Switches (yellow links). While other CPUs and GPUs may couple to CXL Switches (blue links), **Hybrid Switching Units** allow devices with native PCIe protocol (blue link) to reach CXL devices (blue link). Hybrid Switching Units also allow GPUs with xLink to reach CXL-connected devices. The following figure also shows how CPUs may serve pooled Storage or pooled/shared Memory resources to devices on the CXL Fabric or to the devices on the xLink Fabric.



Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Figure 3: Hybrid Switching Units: Bridging protocols between dislike Fabrics (for example between native scale-up and memory fabrics)

In summary, multi-ported devices, switches, bridges, and CPUs may serve pooled/shared memory and storage resources.

3.6 Memory Fabric Manager (MFM)

3.6.1 Architectural Role and Purpose

The Memory Fabric Manager (MFM) represents a foundational logical component within the CMS architecture, serving as the central orchestration engine for memory resource management. The MFM's core responsibility encompasses comprehensive memory resource lifecycle management and optimization of memory infrastructure utilization across data center environments.
As depicted in the CMS Logical Diagram, the MFM functions as the primary control plane, providing centralized management for critical memory infrastructure elements including memory buffers, CXL switches and fabrics, and memory enclosure systems.

3.6.2 Dynamic Resource Management

The MFM provides intelligent oversight of memory bandwidth and capacity resources while dynamically reconfiguring memory fabric topology based on workload requirements specified by orchestration platforms. This adaptive capability enables seamless integration with advanced orchestration frameworks such as Kubernetes, delivering a unified administrative interface for policy establishment, resource allocation, and memory fabric behavior governance.

3.6.3 Security and Monitoring Capabilities

Security enhancement represents a critical MFM function, implemented through comprehensive access control mechanisms and continuous memory usage monitoring. These capabilities contribute significantly to overall data center performance, scalability, and adaptability, ensuring alignment with evolving service requirements and operational demands.

3.6.4 Agent-Based Communication Architecture

The MFM communicates with physical memory devices through dedicated MFM agents, which can be implemented as either software or firmware-based solutions supporting both in-band and

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

out-of-band operational modes. Agent interaction is facilitated through standardized REST APIs, including DMTF Redfish CXL models, along with other communication mechanisms.
For CXL-specific implementations, agents leverage the device's cxl.io protocol and Component Command Interface (CCI) to execute control plane operations on CXL devices. This standardized approach ensures consistent device management across diverse hardware implementations.

3.6.5 High Availability and Integration

The MFM incorporates robust high availability measures designed to eliminate single points of failure, ensuring continuous operation in mission-critical environments. Northbound integration is achieved through REST APIs that interface with orchestration platforms such as Kubernetes, enabling efficient and scalable memory resource management across distributed computing environments.
This comprehensive approach positions the MFM as the central nervous system for composable memory operations, providing the intelligence and coordination necessary for cloud-scale memory fabric deployments.

4. Datacenter Fabric Management Interfaces

Fabric orchestration significantly enhances data center operational efficiency by automating network configuration and management processes, thereby reducing manual intervention requirements and minimizing the risk of human error. This automation enables dynamic resource allocation capabilities that allow infrastructure to scale resources in real-time based on demand fluctuations, optimizing bandwidth utilization while eliminating resource waste.

The orchestration approach delivers enhanced scalability for managing large-scale, complex environments while providing continuous performance monitoring to rapidly identify and address bottlenecks or system failures. Through centralized management platforms, fabric orchestration streamlines operational workflows including system updates, troubleshooting procedures, and infrastructure monitoring, resulting in reduced operational overhead and improved administrative efficiency. Furthermore, fabric orchestration facilitates seamless integration with virtualization platforms and cloud services, supporting hybrid deployment models that enhance both flexibility and operational efficiency across diverse data center environments.

Data center management employs two fundamental approaches: in-band management operates through the same network infrastructure that handles data traffic, utilizing established protocols including Simple Network Management Protocol (SNMP) and Secure Shell (SSH). This approach

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

provides efficiency for routine management operations but faces limitations during network failures. Conversely, out-of-band management functions independently of the primary data network through dedicated management interfaces and specialized protocols such as Intelligent Platform Management Interface (IPMI) and Redfish, ensuring continuous administrative access even during primary network outages.

The Redfish API has established itself as a comprehensive solution for data center hardware management, delivering a standardized and secure interface developed by the Distributed Management Task Force (DMTF). Redfish provides a RESTful architecture with JSON formatting and HTTP protocol support, creating an accessible, developer-friendly environment that facilitates management task automation. The API's capability to provide detailed hardware component information and real-time status monitoring makes it an essential tool for administrators focused on optimizing data center operations.

The Redfish API is currently extending support to include CXL devices, which represent increasingly critical components in contemporary data center architectures. This integration enables data center fabric orchestration to achieve superior efficiency and operational flexibility through dynamic CXL resource allocation and management capabilities. Redfish-enabled CXL device management facilitates optimized workload distribution and enhanced overall system performance through intelligent resource allocation algorithms [3].

The following sub-sections present practical examples demonstrating how a data center memory fabric logical system can be represented as a Redfish model, along with detailed illustrations of how users interact with RESTful APIs to access Redfish services and obtain comprehensive memory fabric information for operational decision-making.

4.1 Redfish model for CMS memory fabric logical system

To illustrate how memory devices are mapped into Redfish schema under different topology, we present a redfish model reference design for the CMS logical system architecture shown as Figure 2 from section 2. This system incorporates mixed components including CXL memory configurations such as directly attached devices, CXL switch-connected systems, multi-headed memory appliances, and alternative transport mechanisms for memory extension. The redfish model for such a combined system with linked connection is shown in Figure 5. More details about mapping resources into the redfish model will be explained in the next sub section.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

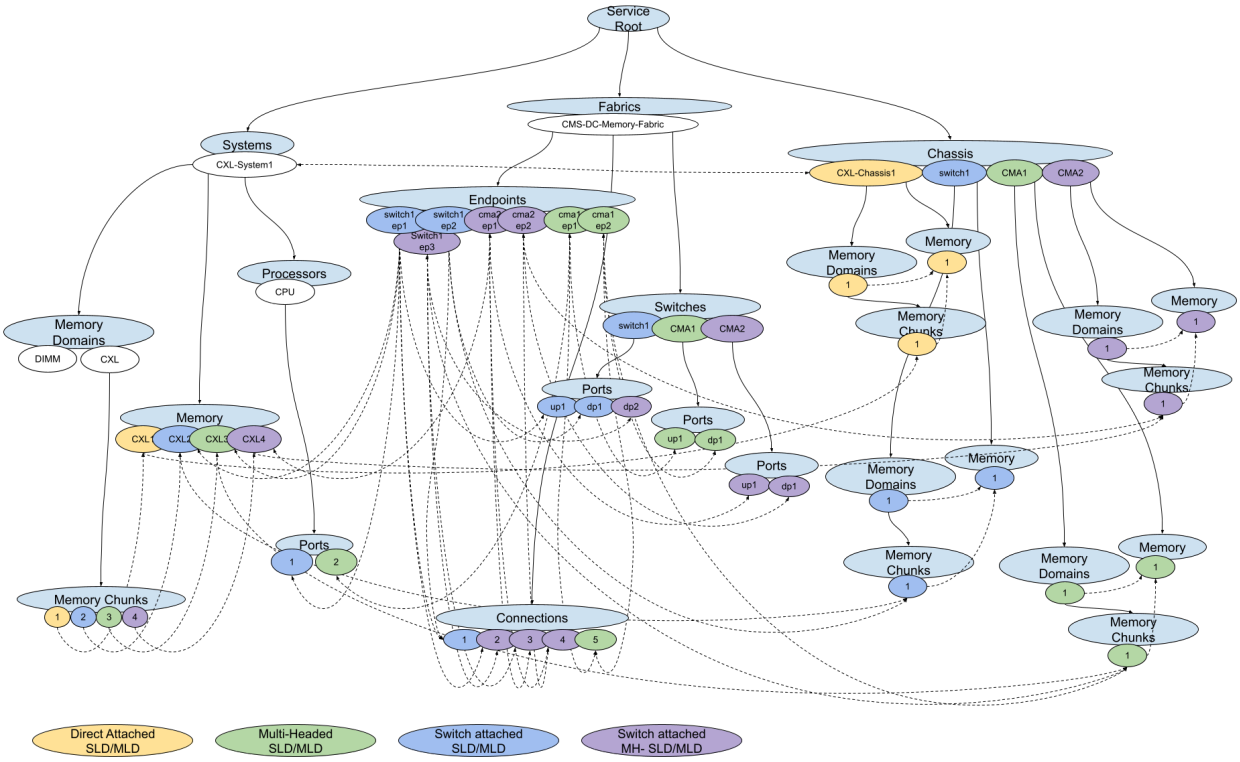


Figure 4: A redfish model chart representing the 4 types of memory buffer devices defined in CMS logical system architecture

4.1.1 Redfish resource mapping

In the Redfish model, different types of Resources are grouped by either physical or logical functionality under collections. Below is a brief description of the main resource collections used in the Redfish reference model for the CMS logical system architecture.

The Service Root establishes the foundational API layer for Redfish implementations. Its attributes typically contain essential management system information and provide navigation paths to subsequent resource levels within the service hierarchy.

Systems, Fabrics, and Chassis Resources constitute the primary collections containing memory fabric information. System resources represent logical abstractions that users can directly interact with through the operating system, Chassis resources represent physical

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

hardware components, and Fabrics resources provide logical views of connections between composable system components.

Memory Resources display devices that present memory functionality to the system. This typically represents individual DIMM modules within servers. For memory devices such as CXL Type 3 memory extenders or multi-headed memory appliances, firmware may present each installed DIMM or memory component as a distinct "memory" resource to support serviceability requirements. Multiple "memory" resources typically operate behind memory controllers, preventing users from associating devices through memory addresses.

Memory Domain Resources function as controllers that translate system memory addresses to memory component addresses, respond to load/store operations, and provide Quality of Service (QoS) and Reliability, Availability, and Serviceability (RAS) information to management systems. Memory domains typically present address ranges that map linearly into host system memory maps. From a system perspective, this may appear as a NUMA node where multiple memory channels are interleaved by CPU memory controllers, or as a CXL device presenting Host-Managed Device Memory (HDM) space to the host.

Memory Chunk Resources represent composable and interleavable units controlled by memory domains. For DIMMs installed on motherboards, NUMA nodes function as memory domains, individual DDR channels serve as memory chunks, and DIMMs constitute memory resources. For CXL multi-logical devices, the CXL device itself operates as a memory domain, each logical device functions as a memory chunk, and DIMMs installed in the CXL module represent memory resources.

Fabric Resources describe connectivity between composable components. Switch structures appear within this category with comprehensive port information, where switch ports and host ports are abstracted as endpoints for connection management.

PCIe Devices and CXL Logical Devices reflect the fact that CXL devices build upon PCIe standards and appear to host as PCIe devices. Components that CXL devices share with PCIe standards are presented as PCIe device resources, while CXL-specific functionality appears as CXL logical device resources. Individual CXL physical devices may support multiple CXL logical devices.

For detailed information regarding individual schemas and expected request and response formats, refer to the comprehensive Redfish schema guide.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

4.1.2 Redfish model chart examples

This section breaks down the Figure 5 combined redfish model for the CMS memory fabric logical system into individual models based on the 4 different types of memory device's fabric representation.

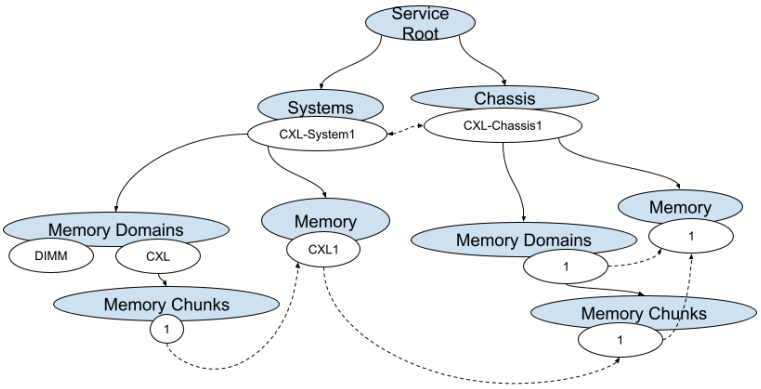


Figure 5: A redfish model chart representing a direct attached SLD/MLD device

Figure 6 shows a single memory device directly plugged into the target system. Since the device is physically plugged, there is no fabric or switch structure shown in the redfish model. The memory as a physically existing device shows under the Chassis collection and the abstracted memory for the Operation System to consume is listed under the Systems collection. There is not much for fabric manager to configure the memory but will monitor the telemetry and serves RAS features through the management system,

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

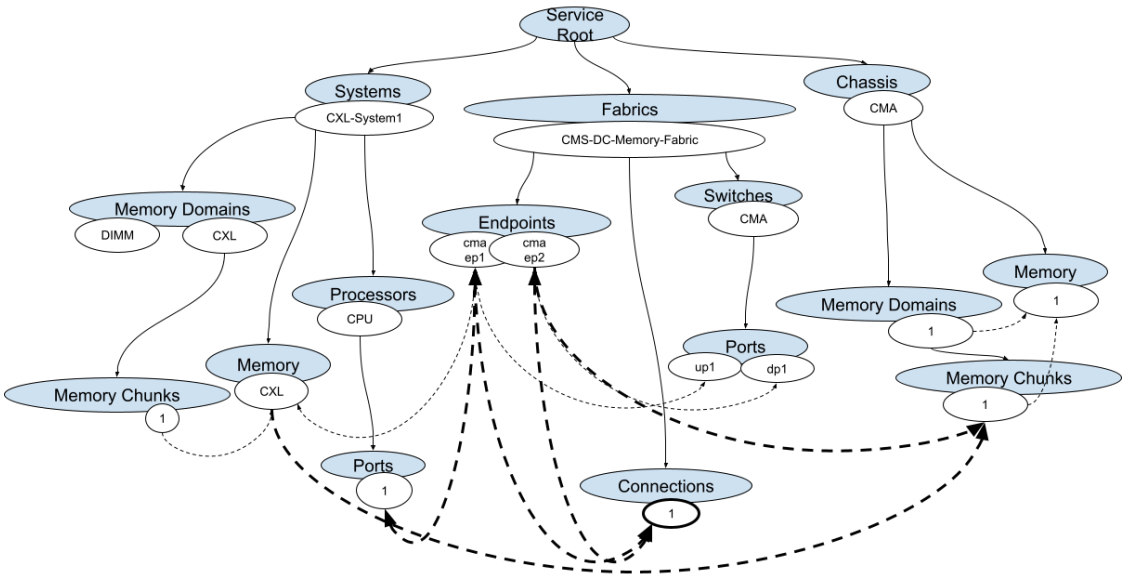


Figure 6: A redfish model chart representing a direct attached Multi-Headed SLD/MLD device

Figure 7 represents the Multi-Headed memory appliance. Memory chunks and connections are composable elements to be managed. Memory needs to be composed into memory Chunks from the memory pool and binded to the target ports. Users would see memory chunks and connections being created and removed as the result of memory provisioning and deprovisioning.

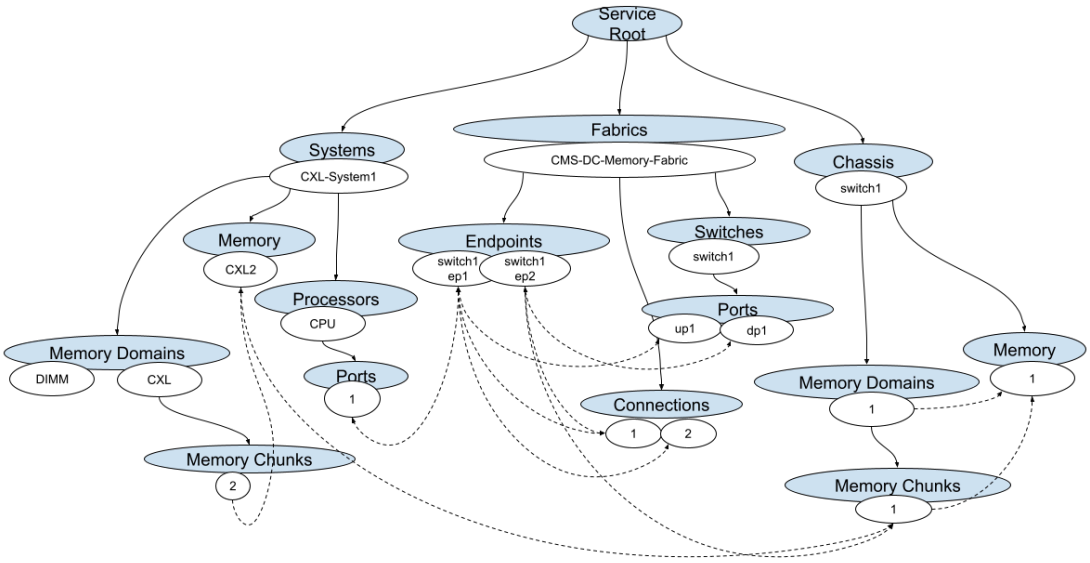


Figure 7: A redfish model chart representing a SLD/MLD device attached through a switch

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Figure 8 shows a memory extender device providing the memory through a switch. In general, the connections between the host port and switch's upstream ports are binded statically and the upstream ports to the downstream ports in the switch are manageable. Users could post connections between upstream and downstream ports to change the binding of the memory to different host servers.

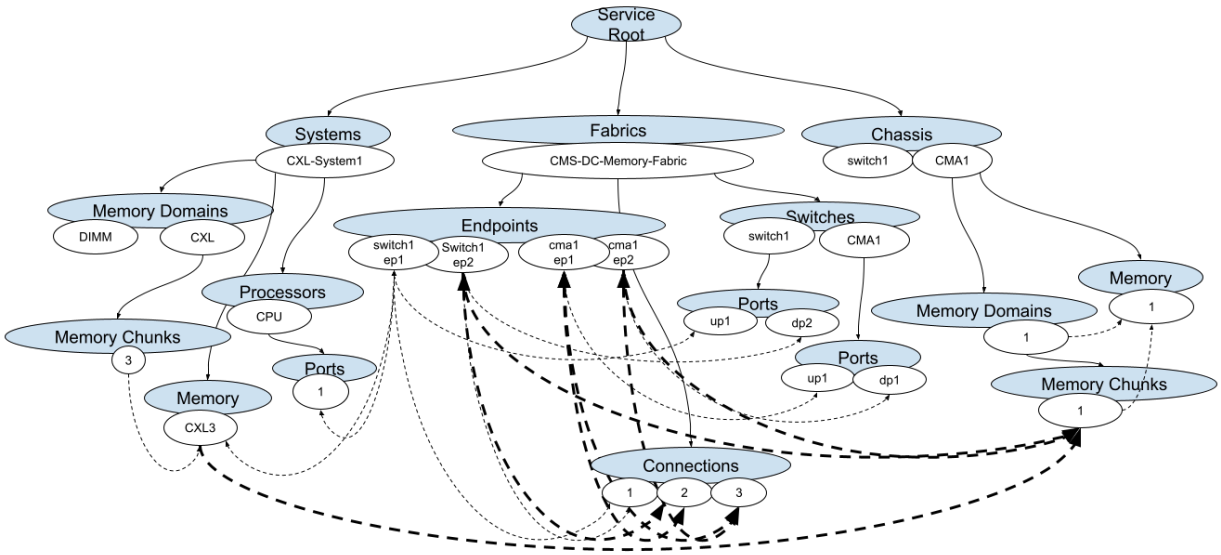


Figure 8: A redfish model chart representing a Multi-Headed SLD/MLD device attached through a switch

Figure 9 replaces the direct attached memory device with a multi-headed device from Figure 8. This is a combination of switch and multi-headed device, which leads to one more layer of connection.

4.2 DMTF Redfish Interfaces for Memory Life-Cycle Management

This section provided memory life cycle management interfaces using a simplified memory topology featuring a single CXL switch connected to two host systems, illustrating memory management processes throughout the complete lifecycle including discovery, provisioning, and deprovisioning stages.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

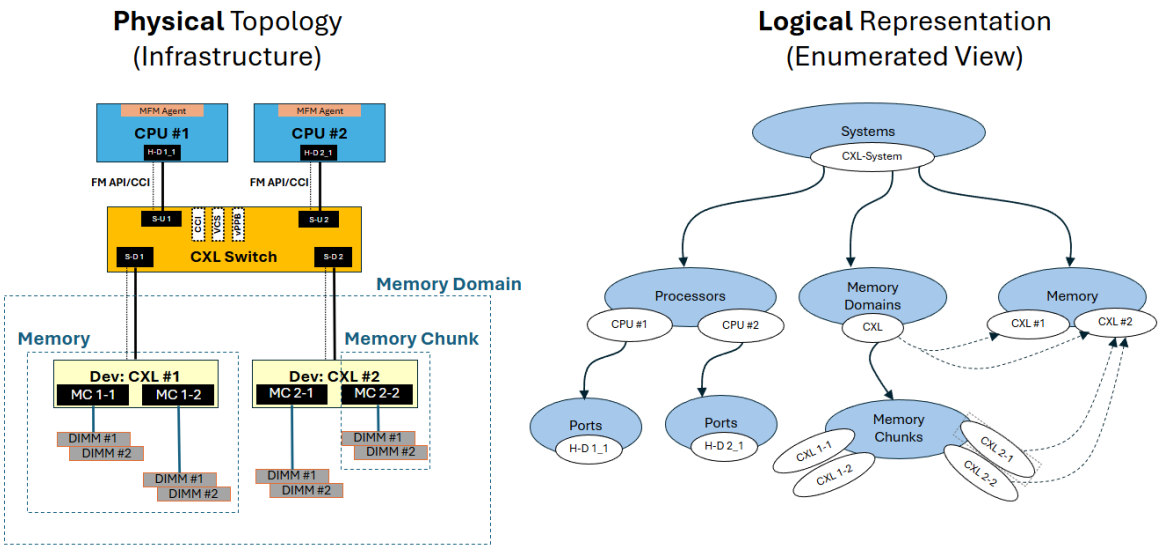


Figure 9: Simplified Physical and Logical views of a CXL memory fabric

The left graphic in the above picture demonstrates a physical infrastructure example for memory deployment using CXL protocol, featuring CXL connectivity between hosts and CXL switches. The CXL switch extends both memory capacity and bandwidth by enabling multiple device connections within a CXL hierarchy as specified in the CXL standard.

The right side graphic presents a logical representation of this enumerated physical infrastructure using CMS Fabric Manager APIs that integrate multiple entities including CXL systems, processors, memory domains, ports, and memory chunks. An example API implementation using Redfish specification is available through the provided GitHub link.

Different application requirements necessitate varied memory device profiles, represented as memory domains with distinct characteristics. For instance, memory and memory chunks may be available as DDR5, HBM, or Storage-Class memory devices, but when applications require capacity beyond what single devices can provide, capacity and bandwidth aggregation becomes necessary. Switch ports can serve as aggregation points that present unified Memory Domain views—such as Memory Domain A aggregating all HBM resources or Memory Domain B consolidating all DDR5 resources.

Memory tiering represents another aggregation approach focused on matching memory accesses with latency considerations. Through physical infrastructure topology mapping, agents can discover and enumerate system characteristics. For example, CXL switch ports 1, 4, 5, and 6 might connect directly to memory devices, providing predictable and shorter latencies, while ports 2, 7, and 8 may

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

require additional hops for memory device access. Consequently, memory devices connected to ports 1, 4, 5, and 6 can be aggregated under a Memory Domain that delivers consistent latencies when applications can accommodate tiered memory access patterns compared to host-based execution.

Memory domain aggregation enables provisioning for interleaved or block allocation, class of service requirements, chunk sizing, chunk deployment increments, and aggregation of discrete homogeneous and heterogeneous memory objects. This approach allows CMS Memory Fabric to support multiple memory systems through CXL protocol implementation.

4.2.1 View Host server info

Method: GET
APIs:
/redfish/v1/systems/{systemId}/MemoryDomains/
/redfish/v1/systems/{systemId}/MemoryDomains/{memoryDomainId}/MemoryChunks
Usage: The user issues the command to the System memory domain API with GET method, to obtain a list of the memory domains available on the system named {systemid} from the response body of the Redfish service. The response from the {systemid} request will contain a list of memory Domains and the user can further issue a GET request to the memory chunk resource for its detailed information. System memory and related media information could be found from the properties of the memory chunk response.

4.2.2 View switch and its attached Memory device info

Method: GET
APIs:
/redfish/v1/Fabrics/{fabricId}/Switches/
/redfish/v1/Fabrics/{fabricId}/Switches/{switchId}/Ports
/redfish/v1/Fabrics/{fabricId}/Endpoints/
/redfish/v1/Fabrics/{fabricId}/Connections/
Usage: The user issues the command to the fabric switch API with GET method, to obtain a list of the switches available on the fabric named {fabricId} from the response body of the Redfish service, and further check the port information following the returned switch id and its returned information about the ports it has. Detailed information about the status of the ports is expected to be obtained from the port resource. More detail about how redfish properties are mapped into CXL CCI commands can be found from the DMTF site. The user can also check the memory device and host binding information through the resources under Endpoints and Connection collection.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

4.2.3 View logical device info

Method: GET
APIs:
/redfish/v1/Chassis/{chassisId}/PcieDevices/{pcieDeviceId}/CxlLogicalDevices
/redfish/v1/Chassis/{chassisId}/MemoryDomain/{memoryDomainId}/MemoryChunks
Usage: The user can iterate through the PCIe Device collection under Chassis to obtain a list of devices and check its property to see if it is a memory device, and backed by certain transport. Further, the user can use the link from the CXL logical device to trace to the memory chunk and memory resource provided by this device.

4.2.4 Manage Memory (de-)Provisioning

Method: POST & DELETE
APIs:
/redfish/v1/Chassis/{chassisId}/MemoryDomain/{memoryDomainId}/MemoryChunks
Usage: In order to provision a certain amount of memory from a memory pool, the user needs to compose a memory chuck with desired configuration such as capacity, interleave, QoS from a list of available resources. Available memory pool information could be obtained with GET command with the APIs to the memory devices. Some hardware may have static memory chunks hard coded by the controller. Memory provisioning might not be available on every hardware. To de-provisioning the memory, the user issues DELETE method to the memory chunks to be de-provisioned.

4.2.5 Manage Memory Binding for VHS

Method: POST & DELETE
APIs:
/redfish/v1/Fabrics/{fabricId}/Connections/
Usage: To assign the composed memory chunk to a host is by POST a connection to the endpoints between the memory chunk and the target host endpoint in the fabric. Some hardware may have static fabric binding. Changing port binding through fabric management might not be available on every hardware. To remove the existing binding, the user issues DELETE method to the connection to be unbinded.

5. OS Sub System Overview – Linux and CXL

Linux has evolved to support fabric-attached memory through a three-layer software architecture as shown in below picture, that seamlessly integrates CXL devices into existing memory management frameworks. This implementation demonstrates how modern operating systems can effectively manage distributed memory resources while maintaining compatibility with existing applications.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

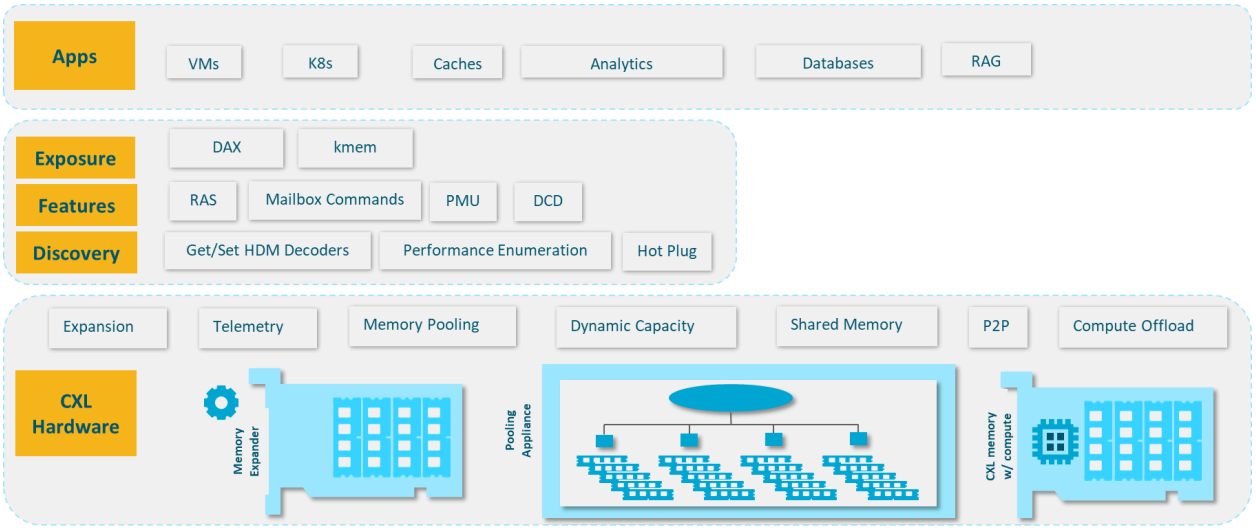


Figure 10: Linux CXL Sub-system Overview

5.1 Driver and Hardware Interface Layer

At the foundation, Linux implements CXL support through specialized drivers that interface directly with CXL devices. The CXL driver leverages the technology's PCIe foundation to handle device discovery and management using familiar hot-plug mechanisms similar to existing PCIe drivers like NVMe. This approach provides a robust foundation for managing fabric-attached memory devices.

The driver layer seeks to support advanced CXL features including Dynamic Capacity Devices (DCD), which enables runtime addition and removal of memory extents. When a MFM (Memory Fabric Manager) initiates memory provisioning requests, the CXL driver recognizes these events and coordinates with upper OS layers to integrate the new memory resources. The driver intelligently presents CXL memory as either application-specific or general system memory based on device timing (boot vs. runtime) and configuration parameters.

5.2 OS Memory Management Integration

Linux integrates CXL memory into its existing memory management subsystem through two primary mechanisms, both designed to work within established OS abstractions without requiring fundamental architectural changes.

NUMA Node Integration: The most common approach exposes CXL memory as CPU-less NUMA nodes. Linux's NUMA-aware memory management automatically incorporates these nodes into

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

system-wide memory allocation decisions. The OS uses performance characteristics provided by the hardware to make intelligent data placement decisions, with memory tiering capabilities that account for the typically higher latency of fabric-attached memory compared to local DDR.

DAX Subsystem Integration: For applications requiring more direct control, Linux leverages its existing DAX (Direct Access) subsystem to provide application-specific or filesystem-specific access to CXL memory. This approach allows controlled consumption of fabric-attached memory without impacting general system memory management, giving applications fine-grained control over memory placement and usage patterns.

5.3 Application Layer and Memory Consumption

Applications running on Linux can consume fabric-attached memory through multiple pathways, depending on their specific requirements and CXL memory configuration. When CXL memory is exposed as general-purpose system memory through NUMA nodes, applications automatically benefit from the additional memory capacity without requiring modifications.

For performance-sensitive applications, Linux provides NUMA-aware interfaces that allow applications to explicitly control memory placement across different memory tiers. CXL-aware applications can leverage NUMA node affinity, direct DAX filesystem access, or direct DAX device management to optimize their memory usage patterns based on workload characteristics.

5.4 Dynamic Memory Management in Production

Linux has successfully implemented dynamic memory management capabilities that address the operational requirements of fabric-attached memory systems. The OS handles memory addition seamlessly when sufficient physical address space is reserved, automatically integrating new memory resources into the system's memory pools.

5.4.1 Memory Addition and Integration

When the MFM provisions additional memory to a host, Linux's hot-plug infrastructure recognizes the new resources and integrates them into the appropriate memory management subsystem. For NUMA-exposed memory, the OS updates its memory topology and begins including the new memory in allocation decisions. For DAX-managed memory, the resources become available to applications through filesystem or direct device interfaces.

5.4.2 Memory Removal and Safety

Linux addresses memory removal through differentiated approaches based on how the memory is managed. For application-managed memory accessed through the DAX subsystem, the responsibility

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

for ensuring safe removal lies with user space applications, providing them with the control and flexibility needed for specialized workloads.

For kernel-managed memory integrated as general system RAM, Linux employs its existing memory migration and hot-remove capabilities. While removal success depends on current memory usage patterns and system state, the OS provides mechanisms to attempt safe memory removal when possible.

5.5 Fabric Management Coordination

Linux effectively coordinates with external fabric management systems while maintaining clear separation of responsibilities. Fabric managers, typically running on BMCs or dedicated management systems, handle fabric-wide resource allocation and coordination. These systems communicate with Linux hosts through standard interfaces, triggering appropriate responses from the OS memory management layers.

The Linux implementation ensures that fabric management actions are properly recognized and handled by the memory management subsystem, while maintaining the separation between fabric-level coordination and host-level memory management. This architecture allows for scalable fabric management while preserving the reliability and performance characteristics that applications expect from the operating system.

5.6 Platform Integration and Compatibility

Linux demonstrates robust integration with system platforms that support CXL hierarchies. The OS can consume CXL memory resources even when system firmware places them in standard platform tables without requiring CXL-specific drivers, providing backward compatibility while still enabling advanced features when CXL drivers are available.

This flexible approach allows Linux to adapt to various platform implementations and deployment scenarios, from simple memory expansion use cases to complex multi-host fabric configurations. The OS continues to evolve its CXL support as the technology matures, with ongoing development focused on optimizing performance, reliability, and ease of use for both system administrators and application developers. Refer to CXL kernel documentation for more detail[8].

6. QEMU Overview for CXL Memory and Fabrics

QEMU provides comprehensive emulation support for CXL fabric memory development, including Multi-Head Single Logical Device (MH-SLD) emulation and switch Component Command Interface (CCI) functionality [5,6]. While QEMU's CXL implementation focuses primarily on enabling software

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

development and prototyping rather than performance optimization, it offers valuable capabilities for experimenting with fabric memory management concepts before hardware becomes widely available.

The emulation environment includes mechanisms for sharing memory between multiple guest systems with hardware-coherent access patterns. Importantly, this shared memory framework supports both coherent and non-coherent memory sharing models, providing developers with a flexible platform for prototyping software solutions across different memory consistency models. See Figure 12 as an example fabric that can be created with QEMU.

To illustrate QEMU's capabilities for fabric management experimentation, the following reproducible example demonstrates the dynamic memory allocation possible when one leverages a memory fabric that is connected between two virtual hosts [7].

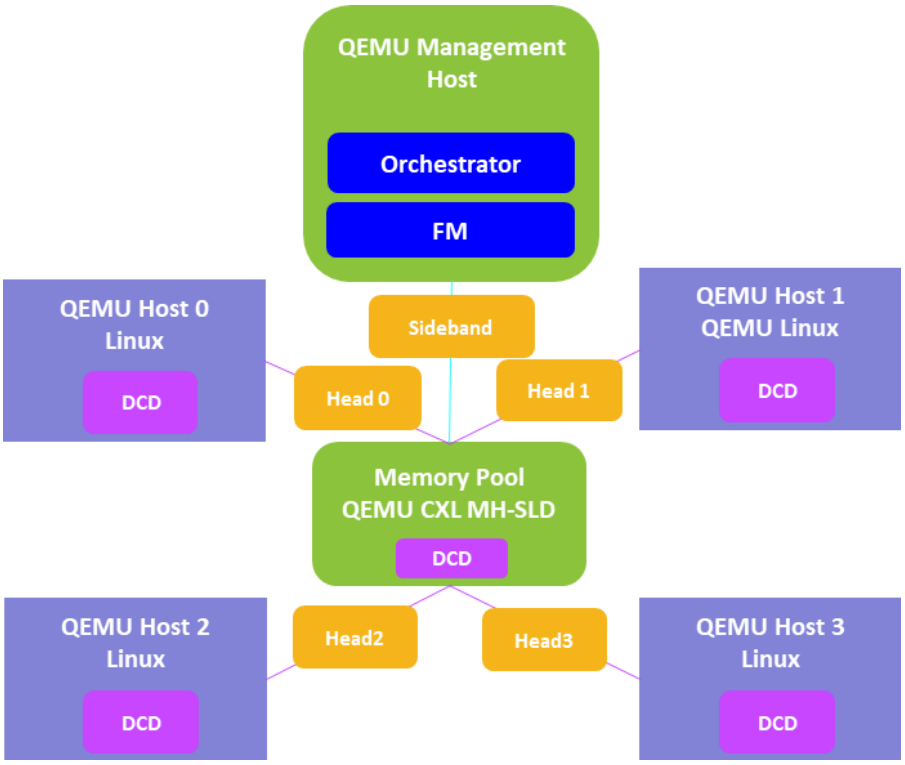


Figure 11: QEMU emulation capabilities with CXL MH-SLD device

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

6.1 Planned Enhancements

Current QEMU memory pooling and sharing capabilities are constrained to multi-headed single logical device configurations. While QEMU provides foundational CXL switch emulation with a switch CCI supporting device identification and physical port state commands, more advanced features remain under development. Physical port control support is currently under review within the QEMU CXL development community. A valuable enhancement for the OCP community would be implementing multi-host shared switch functionality, enabling developers to prototype and validate more sophisticated fabric topologies and management scenarios.

7. Kubernetes Integration with CMS MFM

Kubernetes treats memory today as a static, node-local resource, which limits its ability to manage shared or disaggregated memory. Composable memory technologies like CXL, which enable dynamic, hot-swappable, devices with memory semantics, do not integrate cleanly with Kubernetes without some software integration effort.

There are *many* different ways to integrate new types of hardware into a kubernetes (k8s) cluster ('new' for the k8s is something other than CPUs, local DRAM, network cards, and persistent storage). For example, hardware resources can be treated as abstract labels in user space, they can be expressed as hardware types in different k8s resource allocation plug-ins in the control plane, or they can be resource types in the core k8s scheduler. The takeaway for the reader is that as the integration to k8s gets tighter, the more complexity and community work needs to take place. (As an example, the OCP2024 global summit showed a reference implementation which sat mostly in user-space for k8s [9,10]).

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

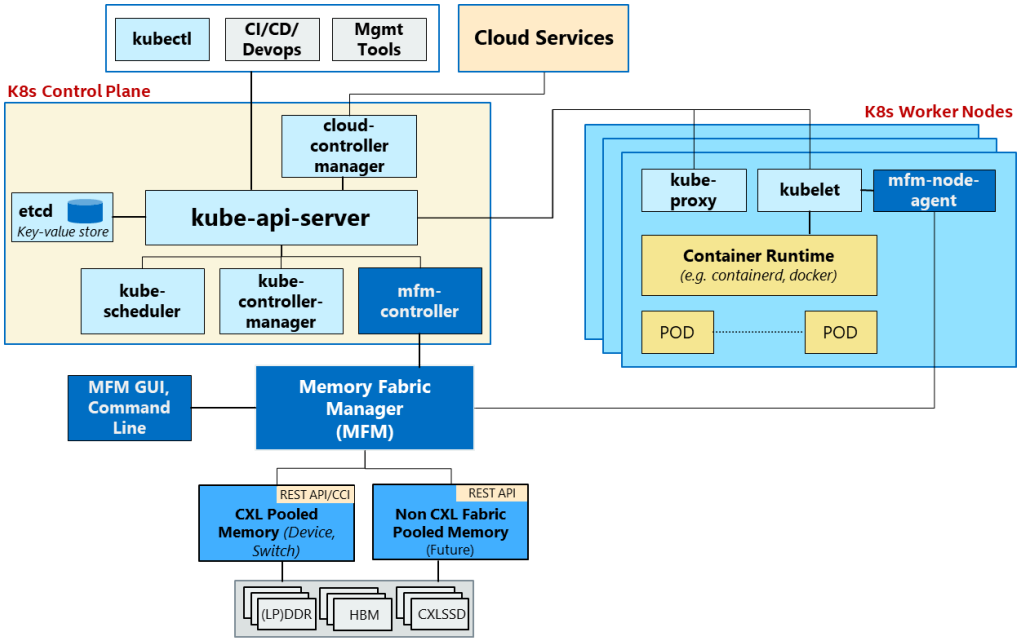


Figure 12: MFM integration with Kubernetes for cluster wide memory management

This section looks forward to describe how the MFM can leverage the Dynamic Resource Allocation (DRA) mechanism to manage CXL-backed pooled memory. The approach integrates fabric-aware resource scheduling and runtime memory mapping while preserving Kubernetes' declarative and secure resource management model. As illustrated below, the DRA controller–node architecture enables dynamic allocation and attachment of CXL-backed pooled memory to Kubernetes workloads. The same architectural framework can be extended to support future fabric technologies that offer memory pooling capabilities with minimal changes to the MFM and Kubernetes integration.

Kubernetes is composed of two primary planes: the **control plane** and the **data plane**, each containing specialized components that collaborate to manage the lifecycle of containerized applications at scale.

7.1 Kubernetes Control Plane Components

The control plane governs cluster-wide decisions such as scheduling, monitoring, and maintaining desired state. Key components include:

- **kube-apiserver:**
  The API server is the front end of the k8s control plane. It is responsible for validating and configuring objects which are used to manage and orchestrate kubernetes resources like

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

pods, services, controllers, etc. In short, it is the gateway to funnel tasks to the appropriate k8s component.

- **kube-controller-manager:**
  The core of kubernetes management, this component runs a collection of control loops, or *controllers*, each responsible for maintaining specific aspects of the cluster's desired state. Examples include the ReplicaSet controller (which ensures the desired number of pod replicas), the Node controller (which monitors node health), and the Service controller.

- **kube-scheduler:**
  The scheduler is responsible for selecting the optimal node on which to run each unscheduled pod. It evaluates resource requirements, affinity and anti-affinity rules, topology hints, and other constraints to make placement decisions. In advanced setups (such as DRA or CXL-integrated systems), the scheduler collaborates with device and resource plugins to perform topology-aware scheduling.

7.2 Kubernetes Data Plane Components

The data plane executes workloads and enforces control plane decisions. Each node in the cluster runs the following components:

- **kubelet:**
  The kubelet is the node-level agent that ensures pods assigned to the node are running and healthy. It communicates with the API server, starts and stops containers as needed, and interfaces with local system resources and plugins. For DRA-enabled systems, the kubelet interacts with node-side device plugins that prepare and expose specialized resources, such as CXL-backed memory regions.

- **Container Runtime:**
  This is the software responsible for running containers—typically `containerd`, `CRI-O`, or another implementation of the Container Runtime Interface (CRI). It downloads images, sets up namespaces, and enforces isolation.

7.3 Kubernetes Add-On and Extension Components

Kubernetes supports numerous extensions through **Custom Resource Definitions (CRDs)**, **Controllers**, **Dynamic Resource Allocation (DRA), Node Resource Interface (NRI),** and **Device Plugins**. These mechanisms allow integration of new resource types—like CXL-based pooled memory—without modifying the Kubernetes core.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

The NRI framework is a public (user space) interface that allows for the customization of K8s resource management by insertion of domain-specific logic into the container runtime. The NRI provides a framework for extending k8s capabilities in resource allocation and management by allowing custom logic to influence the container behaviour at the node level. An example of a working implementation of CXL and k8s using the NRI is discussed in [11].

DRA is a built-in feature of kubernetes that is designed to streamline the process of managing custom, pluggable resources for pods (containers). DRA enables applications to adapt to varying loads, and presents a management interface similar to volume claims, which is a mature interface in k8s. DRA builds on these extensibility primitives, enabling external controllers to dynamically provision and attach hardware or logical resources to workloads at runtime.

7.4 MFM Implementation Details for CXL Pooled Resources

The coexistence of a cluster resource scheduler like Kubernetes with a fabric resource manager such as the MFM (Memory Fabric Manager) enables coordinated orchestration across compute and memory fabrics. The cluster resource manager abstracts datacenter resources for users while remaining aware of all connected hardware resources.

The MFM presents a unified interface that abstracts datacenter resources from end users while maintaining awareness of all connected hardware resources. The MFM operates alongside resource management daemons when CXL resources are present, providing the necessary coordination between management frameworks to deliver composable memory fabric capabilities to end users.
Kubernetes operates with a Control (management) plane responsible for workload preparation and execution management, and a data plane where actual user work occurs through pods (one or more containers collaborating to complete user workloads). MFM integrates with Kubernetes using DRA for pooled memory resource management, ensuring Kubernetes remains standard and maintainable without requiring specialized, unsupported modifications. Key components of MFM Kubernetes include:

- **MFM Controller** – A cluster-level service that allocates memory slices from the CXL pool and exposes eligible nodes through topology hints.
- **MFM Node Agent** – A daemonset running on each node that maps the allocated CXL memory slice into the pod's namespace using standard mechanisms (dax devices, mem slabs, etc.). MFM serves as the central control point for fabric daemons deployed on any server within the Kubernetes cluster that contains CXL attachment points. This architecture supports both direct-attached and fabric-attached CXL configurations, with the fabric orchestrator and daemons remaining transport-agnostic.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

- **MFM** – An external service maintaining topology, zone, and capacity information for all memory pools and connected nodes.
- **Kubernetes Scheduler with DRA Plugin** – The built-in scheduling extension that consumes topology hints from the controller and places pods accordingly.

The workflow for allocating CXL-backed pooled memory in Kubernetes proceeds as follows:

To allocate CXL-backed pooled memory in Kubernetes, an admin would define a `DeviceClass` during server bring-up once, and an end-user would define a `ResourceClaim/ResourceClaimTemplate` whenever CXL resources are needed for a pod. The DeviceClass links the new device type to k8s so that it is schedulable, while the ResourceClaim expresses the need of the user to the kube-apiserver for scheduling of pods.

For example, imagine that you have a multi-headed, multi-logical Type 3 CXL device connected to multiple servers in your k8s cluster with a pool and shared memory space. Your device driver would define how to interact with the regions, as well as the granularity of access, called a `ResourceSlice`.

After a successful deployment of the driver, checking with kubernetes would produce the following, where a `DeviceClass` named `mfm-pooled-memory` might specify a CXL pool in a `ResourceSlice` called `fabric0`:

```
apiVersion: resource.k8s.io/v1
kind: DeviceClass
metadata:
  name: mfm-MHMLD-memory
Spec:
  Selectors:
  ...
    Device.driver == “memory.cxl.example.com”
```

Where `kubectl get resourceslice fabric0 -o yaml` Outputs the following information

```
apiVersion: resource.k8s.io/v1
kind: ResourceSlice
metadata:
  name: fabric0
spec:
  devices:
  - attributes:
```

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

```
      type:
        string: mfm-MHMLD-memory
      capacity:
        size:
          requestPolicy:
            default: 2Gi
            validRange:
              min: 2Gi
              step: 2Gi
          value: 64Gi
      memory:
        value: 64Gi
      name: pooled-memory
  - attributes:
      type:
        string: mfm-MHMLD-memory
      capacity:
      memory:
        value: 64Gi
      name: shared-memory
    driver: memory.cxl.example.com
    nodeName: [Kubernetes assigned node name(s)]
```

The above yaml says that there is a MH-MLD CXL device attached to x nodes in your kubernetes cluster, with two types of resources, a general pool and a shared memory region. The pool can be requested in increments of 2Gi and the shared pool is a region with no request policies (meaning nodes just use it free for all).

A corresponding `ResourceClaim/ResourceClaimTemplate` requests a specific slice from the pool, such as 16 GiB: ResourceClaimTemplates will likely be used as this removes additional admin management responsibilities from a user.

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimTemplate
metadata:
  name: mem-pool-claim-small
spec:
  spec:
    devices:
```


This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

```
    requests:
    - name: req-0
      exactly:
        deviceClassName: pooled-memory.memory.cxl.example.com
        capacity:
          requests:
            size: 16Gi
```

Once the claim is submitted via the template request, the MFM Controller handles allocation. It interacts with the MFM to carve out the requested memory slice, identifies which nodes can access it, and populates the claim's `AllocationResult` with topology hints (`AvailableOnNodes`) that guide the scheduler. Kubernetes' scheduler then filters and scores candidate nodes based on these hints to determine the optimal placement for the pod.

After a node is selected, the MFM Node Agent on that node executes `NodePrepareResources`, mapping the allocated memory slice into the pod's runtime environment. This can be exposed to the pod via environment variables or mount paths, such as `/mnt/cxl/mem-alloc-12345`.

A user can check on an allocation with kubectl get semantics and would get the following output

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaim
...
status:
  allocation:
    devices:
      results:
      - consumedCapacity:
          size: 16Gi
        device: pooled-memory
        shareID: [some uuid]
```

When the pod terminates, the plugin performs `NodeUnprepareResources` to unmap and release the memory slice back to the pool and the claim will disappear from the list of resources in the kubernetes control plane. If a user chose to manually submit a `ResourceClaim` however, they would need to manually remove the request before kubernetes would recall the claim for use elsewhere, this is the primary difference between templates and direct claims.

The MFM controller implements the `Allocate` and `Deallocate` interfaces defined in `k8s.io/kubelet/pkg/apis/dra/v1`. By generating topology hints and leveraging the

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

fabric-aware mapping capabilities of node plugins, this architecture allows multiple Kubernetes nodes to securely consume memory from the same disaggregated CXL fabric while maintaining scheduler awareness, automated cleanup, and strict isolation between workloads.

This architecture allows multiple Kubernetes nodes to consume pooled memory from the same disaggregated CXL fabric while maintaining secure isolation, automated cleanup, and scheduler awareness.

7.5 Future Milestones and Enhancements for Composable Memory in Kubernetes

The current architecture for managing CXL-backed pooled memory in Kubernetes provides a solid foundation for dynamic, fabric-aware memory allocation. Future, general enhancements to fabric attached memory will extend both functionality and flexibility, enabling Kubernetes to evolve into a fully composable datacenter orchestrator. Key directions include:

NUMA-Aware Co-Scheduling
Scheduling decisions will increasingly consider NUMA topology, aligning CPU cores with memory slices that are physically closest on the CXL fabric. This minimizes access latency for latency-sensitive workloads and optimizes overall performance.

Dynamic Resizing of Memory Allocations
Pods can already adjust their memory footprint if configured properly, however kubernetes does not know how to ask for more memory because it treats memory as a fixed resource. Fabric attached memory makes memory composable and dynamic. Enhancing k8s to understand this will increase workload efficiency by removing the likelihood of workload thrashing due to insufficient memory on a node.

Fabric-Aware Metrics and Telemetry
Integrating Kubernetes resource metrics with fabric-level telemetry—such as memory utilization, access latency, and bandwidth—enables smarter scheduling, load balancing, and predictive allocation strategies across nodes.

QoS and Bandwidth-Aware Scheduling
Different memory slices may offer varying performance characteristics. Scheduling that incorporates QoS and bandwidth awareness allows the creation of memory tiers tailored to workload requirements, supporting high-performance, latency-sensitive, or best-effort applications. With DRA, this is potentially low hanging fruit if devices support.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

CXL Security Domain Integration
Multi-tenant isolation can be strengthened by leveraging CXL security domains. Memory slices can be protected with fully-trusted, secure, or default access policies, ensuring safe sharing of the fabric across tenants without compromising data integrity.

Expanded Allocation Parameters
Future iterations will move beyond simple capacity-based provisioning to consider additional allocation factors, including:

- Distance metrics for latency and access optimization
- Interleaving capabilities within and across zNUMA nodes
- Reliability, Availability, and Serviceability (RAS) profiles
- Comprehensive security configurations

The reference implementation demonstrates that Kubernetes readily accommodates composable memory fabric scheduling. By introducing an init pod and supplementary JSON configuration, workloads can dynamically leverage CXL-backed memory, provided appropriate abstraction layers and middleware components are implemented.

By combining Kubernetes' DRA framework with CXL hardware capabilities, this architecture enables dynamic binding of compute and memory across disaggregated infrastructure. Pods can consume pooled memory in a scheduler-aware, secure, and high-performance manner, with automated cleanup and elastic scaling. These enhancements collectively position Kubernetes as a **truly composable datacenter orchestrator**, capable of integrating advanced memory fabrics while maintaining declarative management, multi-tenant isolation, and optimal resource utilization.

8. Future Directions for Disaggregated Memory Systems

The 2024-2025 timeframe has witnessed significant evolution in AI infrastructure economics and CXL adoption, driving the emergence of **hybrid switches** that enable non-CXL xPUs to access CXL memory through high-bandwidth switching with mixed-protocol support. These systems exploit the ability of large-scale AI/ML workloads to tolerate larger memory access latency in return for high-throughput processing and flexible memory provisioning from shared pools. To bridge the gap between xPU scale-up fabrics and CXL-equipped memory devices, new bridge architectures are being developed that translate between different protocols while maintaining the "first, do no harm" principle— ensuring disaggregated systems behave identically to monolithic servers from a software perspective. These solutions will support both memory **extension** (where CXL memory attaches at

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

extended distances with predictable latency) and memory **expansion** (incorporating tiering between DRAM and lower-performance storage).

The ultimate vision represents a fundamental shift from traditional server-centric designs toward rack-level resource disaggregation, mirroring concepts from earlier rack-scale architectures and modern AI rack designs where dedicated shelves house different resource types—CPUs, memory banks, accelerators, and storage—with the entire rack functioning as a unified server. Contemporary AI rack implementations are already demonstrating this approach through specialized configurations that optimize GPU clusters with shared memory pools and high-bandwidth interconnects. Various topologies are emerging to support this transition, from simple dual-ported devices enabling memory pooling between CPUs and xPUs, to complex multi-layered fabric architectures supporting fully-connected memory sharing across distributed AI clusters. While this transformation requires significant changes across hardware, firmware, and operating systems, the design ensures that end users and resource schedulers remain neutral and unconcerned with the underlying infrastructure evolution, maintaining compatibility with existing software ecosystems and operational workflows.

9. Conclusion

The rapidly expanding enterprise and hyperscale markets have positioned memory as the fundamental performance bottleneck in modern computing infrastructure. Memory-intensive applications including Artificial Intelligence (AI), Machine Learning (ML), and high-performance databases are driving computational demands that stress traditional server memory architecture, necessitating revolutionary approaches to memory system design.

Composable Memory Systems (CMS) represents a transformative paradigm that enables dynamic and unified memory management across heterogeneous memory technologies, interconnects, and hierarchical structures. This approach addresses the complex performance, capacity, latency, throughput, data sharing, scalability, and security requirements of next-generation applications while maintaining the foundational "first, do no harm" principle that ensures seamless integration with existing infrastructure.

The CMS logical system architecture provides essential guidance for system designers, data center operators, and enterprise customers navigating diverse commercial implementations. From direct-attached CXL Type-3 devices to sophisticated fabric-connected topologies with multi-headed configurations, the architecture supports flexible deployment strategies that optimize resource

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

utilization across varied use cases. The integration of standardized management interfaces like Redfish APIs and orchestration platforms such as Kubernetes demonstrates that composable memory systems can be deployed without disrupting existing operational workflows.

The emergence of hybrid switches and bridge architectures during 2024-2025 has further expanded the ecosystem, potentially enabling non-CXL xPUs to access CXL memory resources while supporting the unique requirements of AI/ML workloads that can tolerate higher latencies in exchange for massive throughput and flexible memory provisioning. These developments point toward a future of rack-scale disaggregated memory architecture where dedicated resource shelves for CPUs, memory banks, accelerators, and storage function as unified systems, mirroring both historical rack-scale design concepts and contemporary AI rack implementations.

As data center server memory deployment continues evolving toward this ***disaggregated*** future, CMS provides the foundational framework for ***composing*** and managing complex memory fabrics while ensuring compatibility with existing software ecosystems. The flexibility of the architecture in supporting diverse transport mechanisms, management approaches, and deployment topologies positions it as the essential reference for organizations seeking to harness the full potential of composable memory technologies in their data center infrastructure.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

10. Glossary & Appendix

10.1 Glossary

Term/Acronym	Definition
CXL	Compute Express Link, a low-latency, high-bandwidth link that supports dynamic protocol muxing of coherency, memory access, and I/O protocols, thus enabling attachment of coherent accelerators or memory devices.
CXL CCI	CXL Component Command Interface
CXL.io	PCIe-based non-coherent I/O protocol with enhancements for accelerator support
CXL.mem	Memory access protocol that supports device-attached memory.
DMTF	Distributed Management Task Force
HBM	High-Bandwidth Memory
HDM	Host-managed Device Memory. Device-attached memory that is mapped to system coherent address space and accessible to the Host using standard write-back semantics.
HMAT	Heterogeneous Memory Attribute Table as defined in ACPI Specification
PCIe	PCI Express
SLD	Single Logical Device
SRAT	System Resource Affinity Table as defined in ACPI Specification
Fabric	An interconnect system for composing disaggregated systems (via Switches or multi-headed devices for example)
Disaggregate Memory	Disaggregated Memory is not directly connected to the CPU or GPU but is directly available to CPU and GPU via Load/Store commands. This includes CXL-connected

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

	DRAM as long as the latency is no more than three times the latency of locally-attached DRAM.
CXL FM	The Fabric Manager is an entity separate from the switch or host firmware that controls aspects of the system related to binding and management of pooled ports and devices.
CXL FM API	Application Programming Interface for running and executing CXL Fabric Manager commands
CXL Device	CXL is a cache-coherent interconnect for coupling CPU Hosts to peripheral devices such as the traditional non-coherent I/O devices, memory devices, or accelerators either directly or via switches. The types of devices that may attach via CXL include CXL Type 1, Type 2, and Type 3 as well as Hosts and Switches. Hosts use CXL.mem or CXL.io protocols to initiate requests on their CXL Root Ports (RP). Memory devices follow CXL Type 3 model and respond to commands based on CXL.mem and CXL.io protocols. CXL Type 3 devies may behave as a PCIe Device and initiate cycles using CXL.io. Accelerators may use CXL.cache, CXL.io, or PCIe UIO protocol to initiate cycles and follow CXL Type 1 Device model to provide a fully coherent cache in their device. Basic cache coherency allows an accelerator to implement various ordering models and atomic operations. Type 2 Devices are typically accelerators which offer some memory and respond to commands based on CXL.mem and CXL.io while they may initiate cycles using CXL.cache, CXL.io, or PCIe UIO. CXL Switches couple Hosts to PCIe Devices and CXL devices such as other Hosts, Type 3 memory devices or accelerator Type 1 or Type 2 devices.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Memory Fabric Manager (MFM)	An external service to help Kubernetes maintain topology, zone, and capacity information for all memory pools and connected nodes. Its primary responsibility is to fulfill provisioning requests made against memory resources behind ports and devices that it helps to discover and configure.
Memory Expansion	A technique that utilizes software and hardware memory-tiering mechanisms to extend total memory capacity beyond DDR DRAM through CXL-attached devices. It incorporates not just additional ways of attaching DDR DRAM for sharing flexibly among sockets but also lower-cost memory media (e.g., NAND Flash, compressed memory, and emerging memory technologies) that often have higher latency and reduced bandwidth compared to traditional DRAM, which enables a larger memory capacity but requires performance trade-offs through intelligent tiering.
Server Node	A logical computing unit comprising a CPU and its associated resources including memory, storage, networking, and specialized processing units (xPUs), all operating under a single Host Operating System or Hypervisor. These resources may be physically co-located within the same chassis, locally disaggregated within the same enclosure, or remotely disaggregated across external chassis while remaining logically unified under the host CPU's operating system control.
Disaggregated System	A scalable computing architecture that can dynamically expand or contract by adding or removing devices such as switches, memory modules, and processing units through fabric management interfaces like CXL Fabric Manager. It enables flexible resource allocation and reconfiguration without requiring physical system modifications or service interruptions.
SmartNIC/DPU/IPU (Smart Network Interface Card/Data Processing Unit/Infrastructure Processing Unit)	Specialized processing devices that manage data-plane operations within server infrastructure by interfacing with internal components (CPUs, BMCs) and external systems (Rack Managers, Data Center Fabric Managers). These devices offload infrastructure processing tasks from server CPUs for performance or power, and optionally provide enhanced data security through encryption capabilities or enable centralized server management functions on behalf of data center fabric management systems.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

BMC (Baseboard Management Controller)	A management controller responsible for individual server node operations including inventory management, power and cooling control, and out-of-band management functions such as security and RAS features.
Scale Up	Building larger interconnected systems using native CPU or GPU (xPU) interconnects, optimized for parallel computing in a shared memory architecture.
Scale Out	System expansion beyond xPU-native interconnects through bridge devices that provide protocol conversion for distributed processing on a cluster of scale-up systems.
Bridge	A device that interconnects xPUs and components using different native interconnect protocols, potentially providing switching functions between multiple devices.
Memory Provisioning	Servers, hypervisors, or virtualization frameworks using a process of assigning physical memory resources through Memory Fabric Orchestrator.
Performance Metrics	● Bandwidth:Data moving capacity of a physical channel or communication layer ● Throughput: Effective portion of bandwidth utilized for data movement ● Goodput: Fraction of useful data within the cache-lines fetched by CPU cores
Memory Allocation	The process of assigning address space ownership backed by provisioned server physical memory resources to requesting applications and services. It often involves multiple software layers including operating system kernel system calls, language or database runtime environments, and hypervisors with hardware acceleration support.
Memory Tiering	The approach of management of memory resources organized as tiers of performance, capacity, or cost characteristics, including but not limited to near memory and far memory tiers.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

10.2 References

1. Compute Express Link (CXL) 1.1/2.0/3.X specifications: https://www.computeexpresslink.org/
2. Compute Express Link (CXL™) Memory Module Base Standard JESD317: https://www.jedec.org/standards-documents/docs/jesd317a
3. Redfish CXL Device Management: https://www.dmtf.org/standards/redfish
4. Cachelib open-source software: https://cachelib.org/
5. QEMU CXL device emulation: https://www.qemu.org/docs/master/system/devices/cxl.html
6. Jonathan Cameron QEMU CXL device emulation staging tree: https://gitlab.com/jic23/qemu
7. QEMU MH-SLD DCD Example: https://gist.github.com/anisa-su993/4b1df2cca6973aabd9cd22b7a4d03dac
8. The Linux Kernel Documentation: https://docs.kernel.org/driver-api/cxl/index.html
9. Deep Dive into CMS Composable Memory Architecture with Kubernetes Fabric Attached …
10. cms-seq-diag.pdf
11. KubeCon + CloudNativeCon North America 2024: Architecting Tomorrow: The Heterogeneous...

11. License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

OCP encourages participants to share their proposals, specifications and designs with the community. This is to promote openness and encourage continuous and open feedback. It is important to remember that by providing feedback for any such documents, whether in written or verbal form, that the contributor or the contributor's organization grants OCP and its members irrevocable right to use this feedback for any purpose without any further obligation.

It is acknowledged that any such documentation and any ancillary materials that are provided to OCP in connection with this document, including without limitation any white papers, articles, photographs, studies, diagrams, contact information (together, “Materials”) are made available under the Creative Commons Attribution-ShareAlike 4.0 International License found here: https://creativecommons.org/licenses/by-sa/4.0/, or any later version, and without limiting the foregoing, OCP may make the Materials available under such terms.

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

As a contributor to this document, all members represent that they have the authority to grant the rights and licenses herein. They further represent and warrant that the Materials do not and will not violate the copyrights or misappropriate the trade secret rights of any third party, including without limitation rights in intellectual property. The contributor(s) also represent that, to the extent the Materials include materials protected by copyright or trade secret rights that are owned or created by any third-party, they have obtained permission for its use consistent with the foregoing. They will provide OCP evidence of such permission upon OCP's request. This document and any "Materials" are published on the respective project's wiki page and are open to the public in accordance with OCP's Bylaws and IP Policy. This can be found at http://www.opencompute.org/participate/legal-documents/. If you have any questions please contact OCP.

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

12. About Open Compute Foundation

The Open Compute Project (OCP) brings at-scale innovations and hyperscaler best practices to all, spanning technology domains from the data center to the edge, and the technology stack from silicon, to systems, to site facilities and services. The international OCP Community is made up of organizations and people from hyperscale and tier-2 cloud data center operators, communications providers, colocation providers, diverse enterprises, and technology vendors. The OCP Foundation fosters collaboration within the Community to tackle market challenges in an array of OCP Projects that create open contributions such as specifications, designs, and more. The OCP Solution Provider Program then recognizes the application of those contributions in compliant products, solutions, and data center facilities and services with designations like OCP Inspired, OCP Accepted and OCP Ready. With the tenets of openness, impact, efficiency, scale and sustainability, the OCP engages and educates thousands of engineers every year through many events and webinars. Across many initiatives the OCP Foundation and Community are meeting the market today and shaping the future.

Learn more at www.opencompute.org.

Date: 12/19/2025

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.