

Under the Hood of Hardware Systems for Accelerator-Aware Workloads: From Lab to Fleet

Vidya Samyuktha, Ketki Haridas, and Chockalingam A

Microsoft

Abstract - AI-driven, memory-intensive, and data-centric workloads place sustained pressure on processor, memory, and network resources. Shared-resource contention, scheduling interference, and co-tenant activity can increase latency, reduce predictability, and limit throughput. This paper presents an architecture that integrates and virtualizes on-chip accelerators across the lifecycle from laboratory validation to large-scale deployment. Memory movement, compression, decompression, streaming I/O, and intermediate transformations are offloaded to dedicated accelerator data paths, while hardware operation chaining reduces processor involvement, memory traffic, software overhead, and data movement across the on-chip interconnect. The design separates high-frequency data processing from host management and applies queue-level policies for resource partitioning, bandwidth control, prioritization, and congestion-aware throttling. Telemetry covering memory use, queue occupancy, completion status, and errors enables contention detection, policy adjustment, and fault isolation in shared environments. The resulting framework combines scalable accelerator integration, performance isolation, lifecycle validation, and structured error handling to improve throughput, energy efficiency, predictability, and system reliability.

I. INTRODUCTION

Modern cloud workloads increasingly depend on the movement and transformation of large volumes of data. AI pipelines, distributed analytics, storage services, networking functions, and memory-intensive applications place sustained pressure on processor cores, memory systems, and host interconnects. Although general-purpose processors provide flexibility, repeatedly using host cores for data copying, compression, decompression, streaming, and related operations can consume computational capacity that could otherwise be assigned to application workloads.

The problem extends beyond aggregate throughput. Hyperscale infrastructure must provide predictable behavior across diverse workloads sharing the same physical platform. Host scheduling activity, control-plane services, interrupt handling, memory contention, and co-tenant traffic can produce latency variation. A design that reaches high peak throughput under isolated laboratory conditions may therefore behave differently when deployed in Data Centers.

Integrated accelerators offer an opportunity to address these challenges. Accelerator performance should be evaluated as an end-to-end system property rather than as the isolated throughput of an individual engine. Accelerators must support discovery, virtualization, configuration, observability, and diagnosis, while integrating correctly with host and control processors, memory subsystems, firmware, OS, device driver stacks, and fleet management infrastructure.

This paper presents an accelerator-aware system architecture that spans validation, deployment, operation, and reliability. The key contributions of this paper are:

1. **Virtualized accelerator integration architecture:** We present an on-chip accelerator integration model that separates slow-path resource management from the high-frequency data path, enabling direct work submission and per-function/per-queue resource allocation for memory movement and data-transformation.
2. **Policy-driven resource management and observability:** We define a unified model that connects per-VF/per-queue bandwidth allocation, prioritization, and performance isolation with accelerator observability, including PMU counters,

queue state, completion records, address translation, and error telemetry.

3. **Staged system-level validation methodology.** We develop a validation methodology that progresses from capability discovery and basic functional operation to concurrency, NUMA locality, malformed descriptors, address-translation faults, PMU correlation, and representative workload validation.
4. **Fleet-readiness and reliability architecture.** We establish an end-to-end reliability framework that connects command-level completion status and accelerator-specific telemetry with interrupt and error handling, PCIe AER/RCEC reporting, platform diagnostics, reset and recovery mechanisms, and deployment gates for fleet-scale operation.

II. MOTIVATION AND BACKGROUND

System-on-chip (SoC) architectures increasingly rely on domain-specific fixed-function accelerators to address the fundamental inefficiencies of general-purpose CPUs when executing high-throughput, deterministic workloads. General-purpose processors, optimized for control-dominated code with unpredictable branching, incur significant energy and area overhead through instruction fetch/decode pipelines, branch prediction logic, out-of-order execution engines, and multi-level cache hierarchy mechanisms that provide negligible benefit for data-parallel or streaming operations.

Fixed-function accelerators for compression/decompression, encryption/decryption, DMA-based data movement, network packet processing, and graphics/compute (GPUs) exploit algorithmic predictability and data parallelism to deliver 10-100× improvements in performance-per-watt compared to software implementations on CPUs.

Compression and cryptographic engines eliminate instruction overhead by implementing algorithms directly in silicon, achieving multi-gigabyte-per-second throughput while freeing CPU cores for higher priority latency-sensitive tasks.

DMA engines and intelligent data movers reduce CPU intervention in memory transfers and optimize cache

coherency protocols, hiding data movement latency behind useful computation.

Network accelerators offload protocol stack processing, header parsing, and checksum validation to sustain 100+ Gbps line rates without saturating general-purpose cores.

GPUs leverage massive SIMD parallelism and 5-10 $\times$ higher memory bandwidth (via HBM) to dominate bandwidth-bound floating-point workloads in graphics, machine learning, and high-performance computing, delivering 10-50 $\times$ superior FLOPS-per-watt for data-parallel operations. The strategic adoption of fixed-function accelerators represents a deliberate architectural tradeoff: sacrificing programmability for workloads with stable, high-volume computational patterns to achieve orders-of-magnitude gains in energy efficiency and throughput, a necessity as power constraints and specialized workloads (ML inference, video transcoding, secure communication) increasingly dominate datacenter and edge computing environments. Figure below illustrates the comparison between CPU centric execution and domain specific accelerators execution.

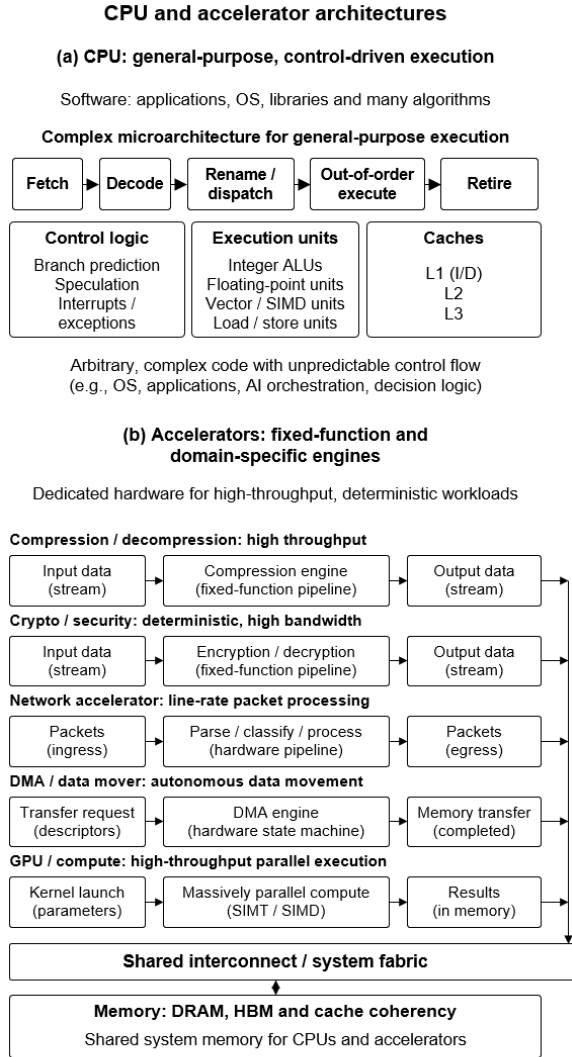


Figure 1. Comparisons CPU, GPU, Network accelerator, On-Chip Accelerator

A CPU implements an algorithm by executing a sequence of software instructions on general-purpose hardware. For example, compression performed by a CPU consists of repeated instruction fetch, decode, scheduling, arithmetic or vector operations, memory accesses, branches, and state updates. A fixed-function accelerator instead maps the dominant algorithm or operation directly onto a specialized hardware data path or state machine. For compression, input data flows through dedicated match, transform, and encoding stages without requiring the CPU to execute the corresponding sequence of instructions for every portion of the input stream.

The CPU consequently provides flexibility and programmability, whereas fixed-function hardware exploits predictability, repetition, and workload-specific dataflow to improve execution efficiency. In a heterogeneous SoC, these architectures are complementary rather than mutually exclusive. The CPU performs control-oriented and irregular computation, while specialized engines execute frequently occurring compute or data-movement functions suited to hardware specialization. The realized system-level benefit also depends on integration factors such as memory bandwidth, interconnect contention, data movement, synchronization, and cache-coherency behavior.

III. ACCELERATOR-AWARE SYSTEM ARCHITECTURE

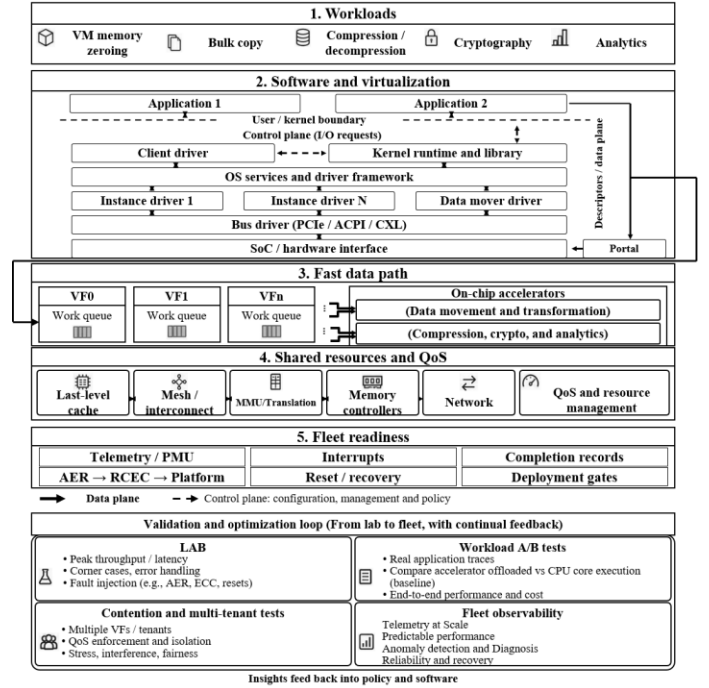


Figure 2. Accelerator-Aware System Architecture

The proposed architecture explains how fixed-function acceleration is most effective when treated as an end-to-end system. The architecture considered in this work consists of five interconnected components: workload specialization, control-plane and data-plane separation, user-visible accelerator instances, shared-resource management, and fleet-level observability.

The first component establishes the workload rationale for fixed-function acceleration by identifying the system scenarios in which specialized execution provides a meaningful advantage over CPU execution. Memory initialization and zeroing are driven by VM provisioning, page recycling, security sanitization, and large-memory lifecycle operations, where substantial memory regions must be prepared without consuming host CPU cycles. Data-movement engines are appropriate for large or repeated copy, fill, compare, scatter-gather, and checksum operations, particularly when CPU occupancy or software orchestration becomes the limiting factor. Compression and decompression are valuable when storage capacity, memory footprint, network bandwidth, or cross-die traffic is constrained, and the expected reduction in data volume exceeds accelerator submission and data-movement overhead. Cryptographic acceleration is driven by sustained encryption, decryption, authentication, hashing, or integrity-processing demand where software execution would consume significant CPU capacity or fail to sustain the required data rate.

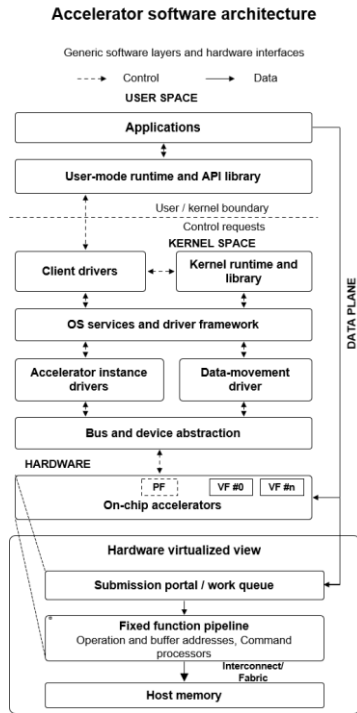


Figure 3 Accelerator software Architecture

The second component separates control-plane management from steady-state accelerator execution. The control plane performs infrequent operations such as device initialization, resource provisioning, virtualization, configuration, policy enforcement, and recovery. PF/VF abstractions extend this separation to virtualized environments: the PF manages shared physical accelerator resources, while VFs expose virtualized and isolated, workload-visible accelerator instances.

Once provisioned, accelerator operations transition to a streamlined data plane. Applications submit operations to buffer descriptors through a user-mode runtime to a VF-associated submission portal or work queue. The fixed-function pipeline processes the request and returns completion status through the corresponding completion mechanism.

With direct access to assigned VF resources, the steady-state path can avoid per-command IOCTL transitions and repeated hypervisor intervention, reducing software and virtualization overhead.

The third component exposes accelerator instances to software. Each logical instance can provide access to resources within their computing boundaries (TCB or GP-VMs), such as work queues, MMIO-visible state, command descriptors, completion records, interrupts, and associated memory. Software invokes the accelerator by configuring the assigned instance and submitting a descriptor that specifies the operation, data buffers, and completion information. The accelerator then retrieves the input data, performs the fixed-function transformation, writes the output, and reports completion status. This division preserves host control while moving repetitive, high-throughput processing from general-purpose instruction execution to a specialized hardware pipeline.

The fourth component addresses shared-resource access and quality of service (QoS). Accelerator generated transactions share the SoC memory and communication hierarchy with CPUs, I/O devices, and other accelerator instances. Requests may traverse the last-level cache, on-chip mesh or coherent interconnect, address-translation and protection mechanisms, memory controllers, and scale-out interfaces such as PCIe. Because multiple VFs and accelerator queues can concurrently consume these shared resources, accelerator performance depends not only on the execution engine but also on available memory and interconnect bandwidth. The architecture therefore incorporates per-VF resource controls, including bandwidth allocation, priority and scheduling, congestion-aware throttling, and isolation mechanisms. These controls provide a means to arbitrate competing accelerator traffic and maintain predictable behavior under multi-tenant and high-contention conditions.

Data-transformation accelerators can also change the traffic presented to downstream resources. For example, compression consumes accelerator and memory-system resources to transform the source data, but a sufficiently reduced output representation can decrease the amount of data subsequently transferred through a constrained interconnect or network. Accelerator performance must therefore be evaluated at the system level, considering execution efficiency together with memory traffic, interconnect utilization, contention, and QoS behavior. Quality of service mechanisms use work-queue policies, prioritization, bandwidth control, congestion response, and telemetry to prevent resource monopolization and protect latency-sensitive workloads. These controls turn shared accelerator resources into a predictable, performance-isolated service for multi-tenant fleet deployment.

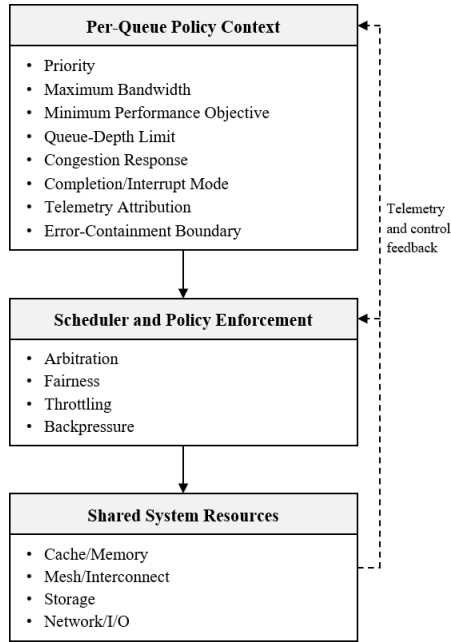


Figure 4: Work-Queue policy context and accelerator resource sharing

Work-Queue policy Context: The policy context may define:

1. **Maximum bandwidth allocation:** Maximum bandwidth allocation establishes an upper resource-consumption boundary for a queue. The limit may regulate the accelerator or associated data-path capacity consumed by a workload.
2. **Minimum performance objective:** A minimum performance objective expresses a lower service target, such as required throughput or latency behavior, for an important workload class. The scheduler can use this target when making resource-allocation decisions under contention.
3. **Queue-depth limits:** Queue-depth limits control the amount of outstanding work that an execution context can place into the accelerator. Provides an early mechanism for controlling pressure before congestion propagates into execution engines, memory, or the system interconnect.
4. **Congestion response:** Congestion response determines how the queue behaves when accelerator or downstream resource pressure increases. Responses may include throttling, backpressure, or changes in scheduling behavior.
5. **Completion and interrupt behavior:** This policy controls how completed accelerator operations are communicated back to software. Queue contexts can use polling, interrupt-driven notification, or a combination of mechanisms depending on workload requirements.
6. **Telemetry:** Telemetry attribution associates accelerator activity with the originating queue or execution context. Relevant observations can include queue occupancy, operations processed, data volume, latency, accelerator utilization, throttling, and errors.
7. **Error-containment boundaries:** Error-containment policy defines the scope at which failures are isolated and recovered. Queue-specific failures should, where the

architecture permits, remain associated with the affected execution context rather than automatically disrupting unrelated workloads. The work queue becomes a natural reliability boundary as well as a performance boundary, supporting targeted diagnosis and recovery.

- A. Bandwidth cap:** limits accelerator and memory consumption while protecting critical workloads under contention. Policies should govern the full data path across compute, memory, accelerator, and network resources.
- B. Congestion-Aware Throttling:** Congestion-aware throttling uses telemetry across queues, engines, memory, interconnects, and downstream devices to regulate workload admission and execution. It reduces background activity under pressure and gradually restores throughput as congestion subsides.
- C. Noisy-Neighbor Detection:** Per-context telemetry identifies workloads that disrupt others through excessive resource use. The policy system then applies corrective controls to maintain performance isolation.

The fifth component provides observability and recovery. Successful accelerator execution progresses from command submission through hardware processing to a corresponding completion record, with counters providing additional visibility into operation and performance. Failure cases require equally explicit reporting through error status, interrupts, architectural error records, or other platform mechanisms. This information enables software to distinguish successful execution from retries and failures and to initiate the appropriate recovery flow. Together, these five components establish a system architecture in which the benefits of specialization extend beyond the accelerator data path itself. Effective fixed-function acceleration requires coordinated design of hardware execution, software control, virtualization, memory and interconnect access, QoS, completion handling, error recovery, and fleet-level telemetry. The resulting architecture preserves the CPU for complex control-oriented computation while assigning stable, high-volume operations to hardware engines optimized for those functions.

IV. VALIDATION FROM LAB TO FLEET

A staged lab-to-fleet validation process is needed to verify functionality, performance, isolation, and fault recovery under progressively more realistic conditions before broad deployment. The figure below shows how each validation phase applies measurable exit criteria and feedback loops to reduce deployment risk and establish fleet readiness.

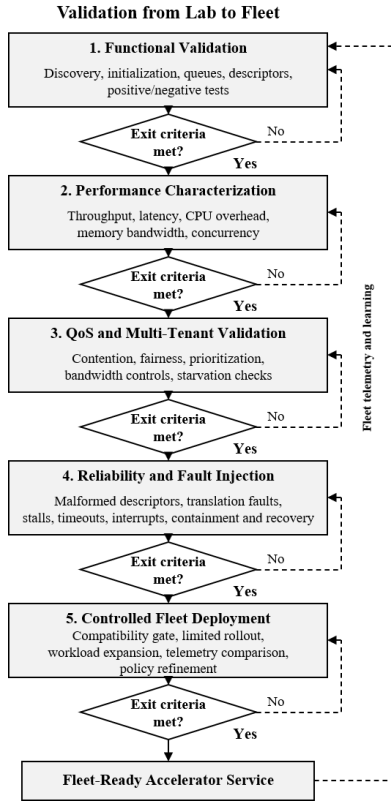


Figure 5 .Validation flow from laboratory qualification to fleet-ready deployment

1. Functional Validation

Laboratory validation begins with device discovery, initialization, queue creation, descriptor submission, operation execution, completion processing, and teardown. Positive and negative tests should cover supported operations, boundary conditions, invalid descriptors, inaccessible buffers, unsupported configurations, and interrupted execution.

2. Performance Characterization

Performance characterization must extend beyond peak throughput to capture how the accelerator behaves across realistic operating conditions. Key dimensions include operation size, queue depth, submission concurrency, host-processor overhead, completion latency, memory-bandwidth pressure, and interference from concurrent workloads. Tests should compare host-based and accelerator-based execution under controlled, reproducible conditions. Each result should document the hardware configuration, software stack, workload definition, measurement method, and observed variability so that performance claims can be interpreted and reproduced accurately.

3. QoS and Multi-Tenant Validation

Run latency-sensitive and throughput-oriented workloads concurrently to reproduce realistic multi-tenant contention. Vary queue policies, bandwidth limits, and priority settings to determine whether each control preserves the intended service behavior. Measure fairness, starvation risk, congestion response, and recovery after a workload exceeds its assigned limits. Validation should confirm that protected workloads remain within their service objectives, and that constrained workloads continue to make forward progress.

4. Reliability and Fault-Injection Testing

Use controlled fault injections to verify error detection, propagation, diagnostics, containment, and recovery. Representative scenarios include malformed descriptors, translation failures, queue stalls, completion timeouts, interrupt anomalies, and simulated internal accelerator faults.

5. Controlled Deployment

Staged rollout: Introduce the accelerator through controlled deployment phases. Early stages should prioritize compatibility, stability, and diagnostic visibility; later stages can expand workload diversity, scale, and resource sharing after predefined exit criteria are satisfied.

Deployment decisions: Advance or pause rollout based on measured evidence from functional, performance, QoS, and reliability validation. Compare fleet telemetry with laboratory baselines to identify regressions, refine policies, and confirm readiness before broader deployment.

V. FLEET TELEMETRY AND OBSERVABILITY

1. Telemetry Requirements

Fleet operation requires telemetry that supports performance analysis, capacity planning, failure diagnosis, and policy tuning. Fleet telemetry and observability are required because isolated host measurements cannot reveal whether an anomaly is workload-specific, platform-specific, or systemic across a hardware generation. A consistent fleet view correlates accelerator, queue, memory, firmware, driver, and error signals across systems, enabling early anomaly detection, capacity planning, noisy-neighbor identification, performance-regression analysis, and rapid fault isolation. It also closes the operational feedback loop: measured behavior can be classified, mapped to the affected execution context, and converted into actions such as policy tuning, throttling, reprioritization, targeted recovery, or deployment rollback.

2. Feedback-Control Loop

Observability becomes more valuable when it informs system action. The proposed architecture uses telemetry as a feedback loop for detecting contention, adjusting queue policy, controlling bandwidth, and identifying abnormal behavior. Avoid unstable policy changes. Adjustments should account for measurement of windows, transient bursts, and interactions between resources.

The control loop includes four conceptual stages:

1. **Observe:** Collect accelerator, memory, queue, completion, and error signals.
2. **Classify:** Determine whether activity reflects normal demand, congestion, unfair consumption, or failure.
3. **Act:** Apply throttling, reprioritization, resource reassignment, reset, or escalation.
4. **Verify:** Confirm that the action improved system behavior without creating a new bottleneck.

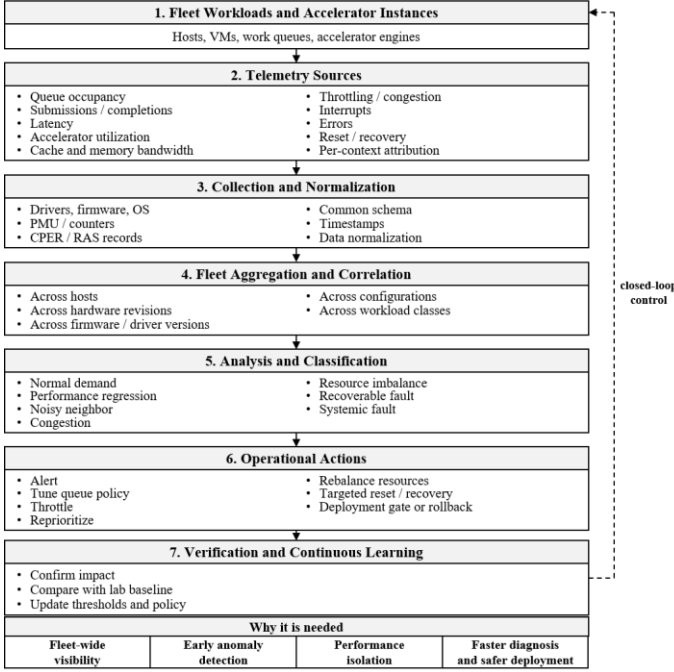


Figure 6. Fleet telemetry, observability, and closed-loop operational control

4. Fleet-Level Correlation

A single host may not provide enough context to distinguish a local fault from a broader implementation issue. Fleet-level correlation enables platform operators to compare failures, performance anomalies, firmware versions, device configurations, and workload characteristics across systems. Diagnostic identifiers and telemetry schemas should remain consistent across validation and production environments. This makes it possible to trace a behavior observed in the fleet back to the corresponding laboratory test and implementation component.

5. Reliability, Availability & Serviceability (RAS)

Reliability, Availability, and Serviceability (RAS) enable accelerator systems to classify faults by scope, severity, and recoverability so software can apply the narrowest effective response, from rejecting an operation to escalating platform recovery. Robust interrupt handling preserves diagnostic state, prevents interrupt storms, and propagates structured error details including the affected resource, execution context, severity, and recovery action through hardware, firmware, drivers, operating systems, and platform-management interfaces. Standardized Common Platform Error Records support fleet-wide diagnosis and targeted recovery.

6. Recovery and Fault Containment

Containment prevents an affected execution context from destabilizing unrelated workloads, while recovery must leave partially completed operations, buffers, and software state in a defined and consistent condition.

1. Reject or fail the affected operation
2. Drain or reset the affected work queue
3. Reset the affected virtual accelerator context
4. Reset the accelerator function
5. Escalate to platform-level recovery

VI. EVALUATION METHODOLOGY

The evaluation methodology uses controlled comparisons of host-based and accelerator-based execution across representative workloads, varying operation size, concurrency, queue policy, memory placement, and contention while measuring performance, efficiency, isolation, and recovery. This methodology is needed to produce reproducible evidence, identify operating limits and bottlenecks, and verify that laboratory improvements remain reliable under realistic multi-tenant and fleet conditions. The evaluation methodology uses controlled comparisons of host-based and accelerator-based execution across representative workloads, varying operation size, concurrency, queue policy, memory placement, and contention while measuring performance, efficiency, isolation, and recovery. This methodology is needed to produce reproducible evidence, identify operating limits and bottlenecks, and verify that laboratory improvements remain reliable under realistic multi-tenant and fleet conditions.

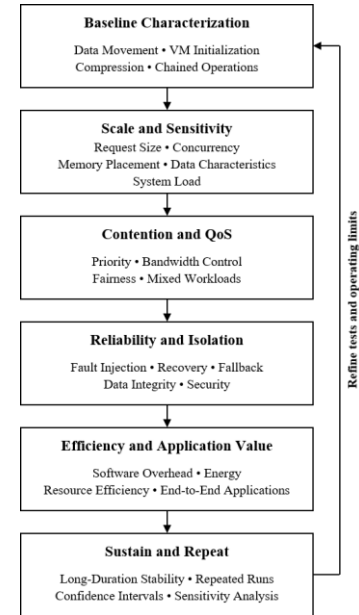


Figure 7. Recommended evaluation flow from baseline characterization through sustained and repeatable validation

Dimension	Evaluation details
Offload Effectiveness	Scope: Move, initialize, compress, chain
	Comparison: CPU vs. accelerator; single vs. chained
	Metrics: Throughput, latency, CPU, bandwidth, energy
	Outcome: Offload gain and CPU savings
Scale and Sensitivity	Scope: Concurrency, size, memory, load
	Comparison: Low vs. high; small vs. large; local vs. remote
	Metrics: Throughput, latency, scaling, saturation
	Outcome: Range, limits, overhead
QoS and multi-tenancy	Scope: Contention, priority, bandwidth, workload mix
	Comparison: Isolated vs. shared; QoS off vs. on
	Metrics: Slowdown, tail latency, fairness, throughput
	Outcome: Isolation and predictable sharing
Reliability and Isolation	Scope: Fault injection; recovery; fallback; integrity; security
	Comparison: Normal/faulted; accelerator/fallback; independent/shared
	Metrics: Coverage; recovery time; completion; output match; violations
	Outcome: Containment, degradation, correctness, isolation
Efficiency and Application Value	Scope: Software overhead; energy; resource cost; applications
	Comparison: Standard/optimized; CPU/accelerator; app baseline/enabled
	Metrics: Submission latency; cycles/request; energy/GB; throughput/W; end-to-end latency
	Outcome: Software, energy, resource, and application benefit
Stability and Confidence	Scope: Sustained load; repeated runs; sensitivity analysis
	Comparison: Short/sustained; repeated trials; controlled variation
	Metrics: Stability; latency drift; errors; power; SD; CI; CV
	Outcome: Robustness, sensitivity, reproducibility

Table 1: Evaluation Metrics

VII. FUTURE WORK

Future work can extend the architecture in several directions. Adaptive policies could use telemetry to adjust queue limits and priorities according to workload conditions. Scheduling decisions could become topology-aware by incorporating memory locality, accelerator placement, and network demand.

Security analysis can examine descriptor validation, denial-of-service resistance, memory isolation, and access-control boundaries. Power and energy measurements can determine whether reduced processor participation translates into system-level efficiency benefits under representative cloud workloads.

VIII. CONCLUSION

This paper presents a system architecture for integrating and virtualizing on-chip accelerators from laboratory validation through fleet-scale operation. The framework moves memory-intensive and data-movement operations from host processors to policy-managed accelerator data paths. Hardware operation chaining reduces repeated host interaction and can limit unnecessary movement of intermediate data. Work-queue-level QoS provides the foundation for bandwidth control, queue prioritization, congestion response, and performance isolation. Telemetry creates a feedback loop for detecting contention, attributing resource use, and refining policy. Structured error propagation, interrupt handling, fault containment, crash diagnostics, and Common Platform Error Records support reliable operation and fleet-level diagnosis.

The central conclusion is that accelerator readiness is a system property. Hardware throughput is necessary, but production deployment additionally requires virtualization, policy, observability, isolation, reliability, and evidence-based validation. When these elements are designed together, integrated accelerators can evolve from specialized offload engines into scalable cloud platform services.

IX. REFERENCES

1. Official processor or accelerator architecture
2. documentation.
3. Peer-reviewed research on accelerator virtualization.
4. Published research on data movement and memory bandwidth bottlenecks.
5. Peer-reviewed work on QoS and performance isolation in multi-tenant systems.
6. Official documentation for Common Platform Error
7. Records.
8. Research on telemetry-driven resource management.
9. Publications describing compression or data transformation acceleration.
- 10.
- 11.
- 12.