

OCP GPU & ACCELERATOR RAS REQUIREMENTS V1.7

Author (s):

[Bhushan Mehendale, Carlos Vallin, Drew Walton, Rama Bhimanadhuni, Venkatesh Ramamurthy, **Microsoft**]

[David Nieto **Google**]

[Linda Wu, Qian Wang, Philip Shirvani, Vishal Jain, Varinder Singh, **NVIDIA**]

[Anil Agrawal, Balaji Vembu, Sukay Luhadia **Meta**]

[Peter VanNess, Taniya Siddiqua, Vilas Sridharan **AMD**]

Contact: hyperscale-gpu@ocproject.net

Table of Contents

Table of Contents 2

Table of Figures 5

Executive Summary 5

License 6

About Open Compute Foundation 7

1 Introduction 8

1.1 Problem Statement and Goals 8

1.1.1 Problem Statement 8

1.1.2 Goals 10

1.2 High-level Requirements for Goals: 10

1.3 GPU RAS System Level View 12

2 Scope 14

3 RAS Feature Requirements 14

3.1.1 Baseline Reliability Metrics for Concurrent Applications 15

3.1.2 Classic Reliability Metrics 16

3.2 System Level RAS Requirements 17

3.2.1 Error Containment 17

3.2.2 Uncorrected Error Handling 18

3.2.3 Corrected Error Storm Handling 20

3.2.4 PCIe Subsystem Fault Handling 21

3.2.5 Other System-Level RAS Features 23

3.3 Memory Fault Handling 24

3.3.1 Memory Error Correction Capabilities 24

3.3.2 Memory Sparing Repair Capabilities 25

3.3.3 Memory Scrubbing 26

3.3.4 Protection from Corruption on the Memory Bus 27
3.3.5 Data Link Parity Protection 27
3.3.6 Containment (e.g., Data Poisoning) 28
3.3.7 Memory Error Reporting 28
3.4 GPU Internal RAS Features 28
3.4.1 Internal Error Coverage 28
3.4.2 Other Radiation Tolerance Features 29
3.4.3 Other Influences on Failure Rates 29
4 Error Reporting 29
4.1 Error Reporting Requirements 30
4.1.1 Error Logging Overwrite Rule Requirements 30
4.2 Error Record Requirements 31
4.3 In-band Vs Out-of-Band Error Logging 32
5 Error Injection 32
5.1 Overview 32
5.1.1 Error Implanting 33
5.1.2 Error Triggering 33
5.1.3 Error Spoofing 34
5.2 Requirements for Testing Error Handling 34
5.3 Error Injection Coverage 35
5.4 Error Injection Security Requirements 36
5.4.1 Secure Unlock 37
5.4.2 Alternate Authorization 38
6 Crash and Device Debug Dumps 39
6.1 Introduction 39
6.2 Crash Dumps 39

6.2.1 Crash Dump Requirements 40
6.3 Debug Dumps 41
6.3.1 Requirements for Debug Dumps 42
6.4 Dump Data Sensitivity 42
6.5 Dump Reliability 43
7 Analysis Tool Requirements 43
7.1 Analyzer Code Requirements 45
7.2 Input Format Requirements 46
7.3 Output Format Requirements 46
8 Compliance tools 48
9 Confidential Compute and RAS Considerations 49
10 OCP GPU RAS document - Future Revision Plan 49
11 References 49
12 OCP Tenets 50
13 Appendices 51
13.1 Appendix A - Implementation Considerations 51
13.1.1 Rationale for hardware error format standardization 51
13.1.2 Using CPER as hardware error format 51
13.1.3 Reporting CPER records via Redfish 53
13.2 Appendix B – Error Injection Support in Redfish 54
13.2.1 Device Identification using Redfish device tree 56
13.2.2 Error Injection Schema Overview 57
13.3 Appendix C – Debug Dump Support in Redfish 58
13.4 Appendix D – Debug Data Taxonomy 61
13.4.1 Level 1: 63
13.4.2 Level 2: 64

13.4.3 Level 3: 64
13.4.4 Level 4: 64
13.4.5 Level 5: 65
13.4.6 Level 6: 65
13.5 Appendix E – RAS Feature Requirements Table 65
13.6 Appendix F - Glossary of Terms 67

Table of Figures

Figure 1: System Level View 12
Figure 2: Various combinations of RT and MTTI that are negotiable between Hyperscalers & Suppliers 16
Figure 3: CPER Format 31
Figure 4: Overview of Secure Unlocking 37
Figure 5: Overview of Error Analysis 44
Figure 6: Error Analysis Process 45
Figure 7: High-level flow for Redfish-based Error Injection 54
Figure 8: Redfish Device Tree for the UBB Component 57

Executive Summary

This document standardizes certain RAS (Reliability, Availability, and Serviceability) capabilities for GPU and accelerator devices. With the explosion of AI/ML workloads and rapid adoption of GPUs and Accelerators in hyperscale Data Centers, ensuring a standard set of critical RAS capabilities is necessary to improve time-to-market and interoperability. The primary goals of this spec are to (a) ensure that GPU/accelerator devices are sufficiently reliable, available, and serviceable to meet hyperscale data center requirements; (b) increase interoperability by standardizing interfaces and mechanisms for certain features; and (c) allow data center operators to validate firmware and software stacks and verify end-to-end error handling and resiliency workflows (e.g., to identify faulty field replaceable units (FRUs) and drive service actions).

This document specifies requirements for error injection, error reporting, debug dumps, and GPU fault management. The goal is to leverage the Open Compute Project to disseminate these requirements to GPU and accelerator suppliers and Hyperscalers to provide mutual benefits and propel the industry forward.

License

Open Web Foundation (OWF) CLA

Contributions to this Specification are made under the terms and conditions set forth in **Modified Open Web Foundation Agreement 0.9 (OWFa 0.9)**. (As of October 16, 2024) ("Contribution License") by:

AMD, Google, Meta, Microsoft, NVIDIA

Usage of this Specification is governed by the terms and conditions set forth in **Modified OWFa 0.9 Final Specification Agreement (FSA)** (As of October 16, 2024) **("Specification License")**.

You can review the applicable Specification License(s) referenced above by the contributors to this Specification on the OCP website at https://www.opencompute.org/contributions/templates-agreements. For actual executed copies of either agreement, please contact OCP directly.

Notes:
The above license does not apply to the Appendix or Appendices. The information in the Appendix or Appendices is for reference only and non-normative in nature.

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS
OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

About Open Compute Foundation

The Open Compute Project Foundation is a 501(c)(6) organization which was founded in 2011 by Facebook, Intel, and Rackspace. Our mission is to apply the benefits of open source to hardware and rapidly increase the pace of innovation in, near and around the data center and beyond. The Open Compute Project (OCP) is a collaborative community focused on redesigning hardware technology to efficiently support the growing demands on compute infrastructure. For more information about OCP, please visit us at http://www.opencompute.org

1 Introduction

This document standardizes Reliability, Availability and Serviceability (RAS) capabilities of GPUs and accelerator devices for use by Hyperscalers. With the growth of multi-supplier GPU and accelerator fleets in data centers, ensuring a standard set of RAS features across all devices will improve time-to-market and interoperability.

For the remainder of this document, the terms "GPU," "accelerator," and "device" will be used interchangeably. In all cases, the term refers to the product shipped by the device supplier, which may include multiple components residing on an OCP Universal Baseboard (UBB) as well as the firmware and software stacks (e.g., device drivers) used to operate the device.

1.1 Problem Statement and Goals

1.1.1 Problem Statement

Hyperscalers face challenges when integrating accelerators, partly due to software complexity and the heterogeneous nature of their implementation. One goal of this document is to provide commonality in certain RAS features and interfaces to reduce the burden of adopting new products. Additionally, inadequate provisioning or adoption of RAS feature sets can lead to insufficient reliability, availability, and serviceability in hyperscale data centers.

The following table summarizes typical pain points experienced by Hyperscalers, which this specification aims to address:

Table 1: Example Problems

Example Problem	**Customer Impact**	**Desired Solution**
GPU not detectable on boot	Capacity Availability MTBF TTM Delays	A reason code that specifies why the GPU was not detectable. Some possible causes include: -PCIe* training issues

Example Problem	Customer Impact	Desired Solution
		-Boot ROM corruption -Bad Microcode or firmware update
Performance impact due to hardware errors	Workload Execution SLAs	Sufficient visibility into hardware errors to understand that they impact performance. Possible causes include: -Error storms -Link degradation -Performance throttling due to thermal excursions or power problems -Driver mishandling error conditions
GPU falling off the bus during run-time	Capacity Availability MTBF TTM Delays VM Crashes	A reason code that specifies why the GPU is falling off the bus. Some possible examples include: -Uncontained memory Errors (SRAM/DRAM) -PCIe Issue (link-state down, uncorrectable) -Re-Timer failures -Voltage Regulator failures
Silent Data corruption (SDC) due to hardware errors	Loss of Confidence Workload Execution Stale/Unavailable Data SLAs	Sufficient detection (in silicon, with test tools) to minimize the SDC rate. Sufficient error containment for detected errors to prevent the use of corrupted data. Possible causes include: -Latent defects in silicon -Soft errors -Aging of components -Lack of containment in design
Uncorrectable errors due to hardware faults	Capacity Availability MTBF TTM Delays VM Crashes	Fault management solutions that minimize downtime, including: -Error logging and analysis which drive RAS actions that minimize impact of Detected Uncorrected Errors (DUEs). -Better error correction capabilities to turn more UEs into CEs. -Error handling policies
Inability to accurately	Capacity Availability MTBF	Fault management solution that enables Hyperscalers to diagnose hardware failures and repair the failures, including:

Example Problem	Customer Impact	Desired Solution
diagnose some hardware failures	TTM Delays VM Crashes NFF replacements	-Telemetry adequate for diagnosing problems (error logs and crash dumps), -Analysis tools to process telemetry (identify nature of failure, identify repair actions)

*PCI-SIG®, PCI Express®, and PCIe® are registered trademarks of PCI-SIG

1.1.2 Goals

To address the problem statement described, the primary goals of this specification are as follows:

1. Ensure that GPUs and accelerator devices are sufficiently reliable, available, and serviceable to meet Hyperscalers' requirements.
2. Improve node uptime, by achieving lower interruption rates due to hardware crashes and better mean time between failures (MTBF).
3. Improve serviceability by reducing mean time to repair (MTTR).
4. Increase interoperability by specifying common interfaces and mechanisms for certain RAS features.
5. Allow data center operators to validate their own RAS firmware and software stacks against end-to-end data center RAS flows (e.g., to identify faulty field replaceable units (FRUs) and drive service actions).
6. Improve Root-Cause Analysis (RCA) time– ability to provide Hardware & Firmware error telemetry to RCA issues and meet Service Level Agreements as per severity of the issue.

1.2 High-level Requirements for Goals:

- **Hardware error resiliency**

  Devices shall minimize the probability of a VM/Node crash due to hardware faults by detecting, correcting, signaling, and containing errors rather than forcing system resets. Fault handling requirements are described in Chapter 3, RAS Feature Requirements, which describes how error mitigation and recovery capabilities are required to reduce the blast radius of uncorrected errors and

how known corrupt data, e.g. uncorrected errors or poison, must be contained and not consumed. Error Reporting requirements are in Chapter 4, Error Reporting.

- **GPU system level reset cause identification**

  To identify all the GPU subsystem reset causes, provide the telemetry related to reset.

- **GPU system level crash dumps**

  The GPU subsystem may encounter a fatal error due to a failure in any of the components in the UBB. In such cases, a crash dump must be provided with sufficient fault information to help locate the source of the crash and perform root cause analysis (RCA). To meet hardware error RCA and SLA times, Hyperscalers require GPU system level crash dumps with capability API support, crash dump triggering mechanisms and tools support to analyze dumps.

- **Fault-isolation to Field Replaceable Unit (FRU) level**

  System FW/SW shall report hardware errors using industry standard specifications like Redfish, IPMI SEL, ACPI APEI error record formats defined in UEFI CPER (Common Platform Error Record). These shall allow for:

  1. FRU Isolation for all types of Hardware Failures.
  2. Use structured data that describes hardware errors.
  3. Support filtering and categorization based on error severity and types of errors.
  4. Allow entries to contain an unbounded amount of data.

- **Error testing capabilities**

  Device suppliers should provide hardware error injection and spoofing capabilities that can be integrated into cloud infrastructure to verify error detection, signaling, logging, containment workflows and to validate RAS features.

- **GPU screening tools**

  GPU screening tools which can detect defects which can be integrated into manufacturing and production environments.

- **Compliance Tools**

  Create open-source software that shall behave like a compliance check for suppliers and system integrators to ensure GPU compliance with the RAS requirements described in this document. These checks will be described in Chapter 8, Compliance tools.

1.3 GPU RAS System Level View

At a system level, GPU RAS features are implemented across multiple components, as shown in the figure below.

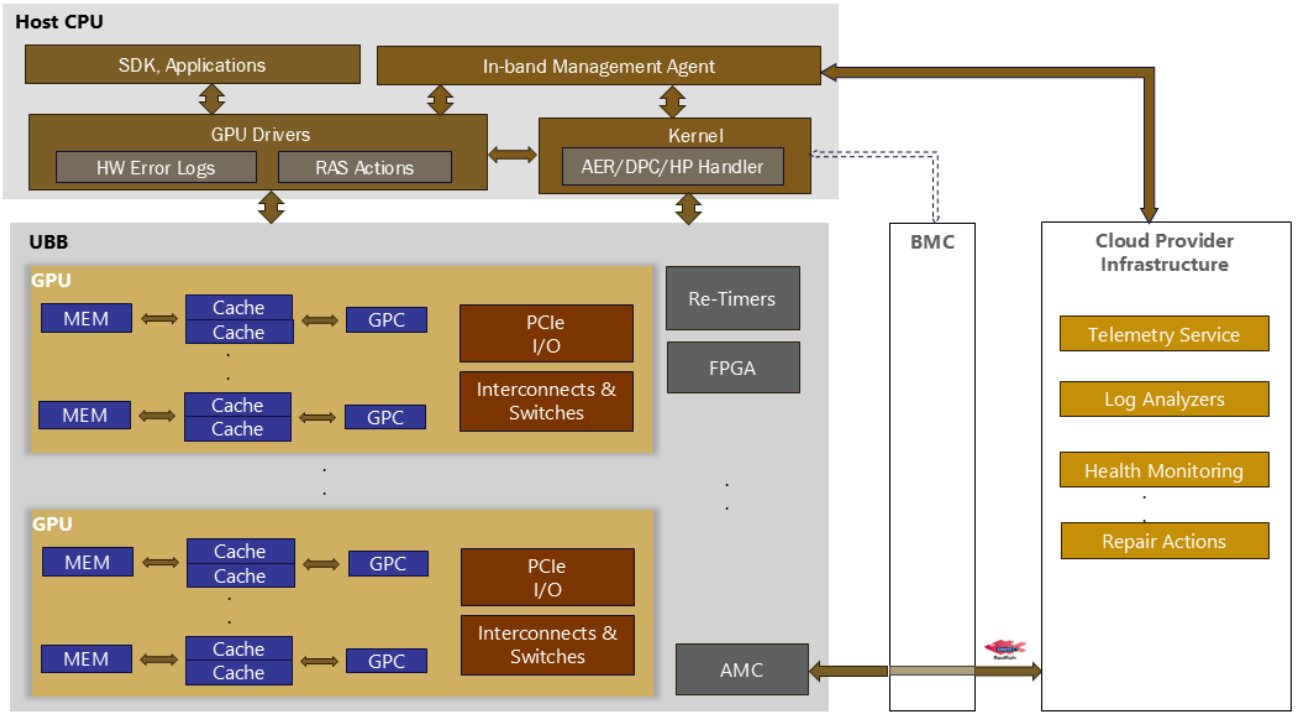


Figure 1: System Level View

The following list gives a high-level overview of RAS requirements for components in the drawing. More detailed RAS requirements are found in Chapter 3, RAS Feature Requirements.

- **MEM (Memory)**

This includes requirements such as memory error detection, correction, and reporting. In addition, system level features such as page off lining, row-remapping, bad-page map out, and error containment are required. The OCP Fleet Memory Fault Management team is working through the details of memory error logging and analysis.

- **Cache**

  This includes cache error detection, correction, and reporting.

- **Accelerator Core/GPC**

  This includes various types of error detection, correction, and reporting within the device's core logic.

- **PCIe I/O**

  This includes PCIe specification defined error detection, correction, and reporting as part of Advanced Error Reporting (AER), and Root Port Programmed I/O (RPPIO) error reporting. In addition, system level features such as 'Downstream Port Containment (DPC)' and 'Surprise Hot Plug' are required. The device must fail operations that consume poisoned TLPs and must prevent the host from consuming corrupted data, either by sending the host poison or failing any operation that uses the corrupted data.

- **Interconnect (XLinks) and Switches**

  UBB designs may have additional Switches and Interconnects, and support supplier specific error detection, correction, and reporting. Additionally, telemetry to detect link integrity issues is required.

- **Re-timers**

  Systems like UBB design may have re-timers and require error collection related to PCIe link signal integrity and protocol errors.

- **Kernel (AER/DPC/HP Handler)**

  This includes Kernel components required to handle UBB specific failure events, such as AER handler, DPC handler, and Hot-plug handler. This also includes an interface to a device driver and the In-band Management Agent.

- **GPU Driver (Error Handler)**

  This includes the UBB specific error handlers that are managed via the driver, and various UBB supplier specific RAS actions.

- **Accelerator Management Controller (AMC)**

  This includes error collection and storage in persistent space for off-line error log collection via BMC and hyperscale infrastructure tools.

- **In-band Management Agent (IMA)**
  This includes collecting error events in-band from UBB and presenting the logs to hyperscaler Infrastructure using Redfish schema.
- **Baseboard Management Controller (BMC) for Out-of-band Management**
  A node-level entity responsible for node management, telemetry and error reporting. This may include collection of error events out-of-band from UBB in pass-through mode where Hyperscalers' infrastructure can access telemetry data from the AMC.
- **Hyper-scale Infrastructure**
  This includes various levels of telemetry data aggregation, RAS action policy such as auto-repair, or manual-repair, FRU isolation hints requirements for both in-band and out-of-band.

2 Scope

This document aims to address the pain-points listed in Section 1.1 Problem Statement and Goals by proposing standardization for GPU & Accelerators-based systems level Error Injection, Error Reporting, crash/Debug dumps. In addition, it provides GPU/Accelerator reliability metrics and describes system-level, PCIe, memory, and silicon-internal RAS requirements that help meet the metrics. The requirements defined in this document will affect the definition of the OCP RAS API specification[11] and vice versa. As the RAS API becomes more mature, future versions of this document will be revised to align with it.

Error reporting uses CPER based error records. Error reporting and mitigation aspects of this specification must apply to production level hardware and firmware.

3 RAS Feature Requirements

Suppliers must first understand how Hyperscalers measure the RAS impact of GPUand accelerators-based systems. This requires defining the metrics used by Hyperscalers to measure the effectiveness of the RAS features outlined in this document.

In a typical hyperscaler setup, these modules are installed in computation nodes that form clusters of many racks. Applications running on these GPUs and accelerators in such clusters can be classified as either

concurrent (e.g., used for training LLMs) or non-concurrent (e.g., used for inference). The definition of metrics and targets is dependent on these categories. For example, uncorrected hardware errors in concurrent application workloads can result in application interruptions, which are particularly undesirable. System crashes due to other types of faults, whether transient or permanent, can also cause interruptions. Restarting an interrupted application requires recovering via a check-point state, which impacts cluster efficiency. The following section lists baseline reliability metrics particularly targeted for concurrent applications.

3.1.1 Baseline Reliability Metrics for Concurrent Applications

1. **Application Interruption Rate (IR):** It is defined as an acceptable mean application interruption rate per unit time due to UBB hardware faults, e.g., a job crash running on the target UBB. For example, if an application is interrupted every 4 hours, then Application IR = $1/4 = 0.25$ interruptions/hour/cluster.
2. **Mean Time To Interrupt (MTTI):** It is defined as a mean time between two interruptions of application execution running on the target cluster. For example, if the application is interrupted every 6 hours, then Cluster MTTI = 6 hrs. MTTI and IR are interrelated as $IR = \frac{1}{MTTI}$
3. **Application Restart Time (RT):** It is defined as the time it takes to restart the job once it is interrupted. The time is measured from the point of interruption to the point where it resumes normal operating state (i.e., it includes detection, restore, restart, and re-sync time).
4. **Cluster Efficiency (CE):** It is a measure of efficiency of the cluster where useful work is performed. It is defined as $CE = 1 - \frac{RT}{MTTI}$
5. **Cluster Useful Availability (UA):** It is a measure of the availability of the cluster for performing useful work. It is defined as $UA = 1 - \frac{RT}{RT+MTTI}$

More specifically, to build a 100K GPU or Accelerator cluster for training, **baseline requirements** could be as follows. These are negotiable between hyperscaler and suppliers. An example shown below:

1. **Cluster Efficiency:** 95% or better.
2. **Cluster Useful Availability:** 95% or better
3. **Interruption Rate:** 2 per day per 1K gpu/acc (MTTI = 12 hrs.) or lower
4. **Application RT:** 3 min or lower

It is also possible to design a cluster with various combinations of IR and MTTI. The following figure shows a graph of cluster efficiency as a function of `RT` (min) and `MTTI` (hours):

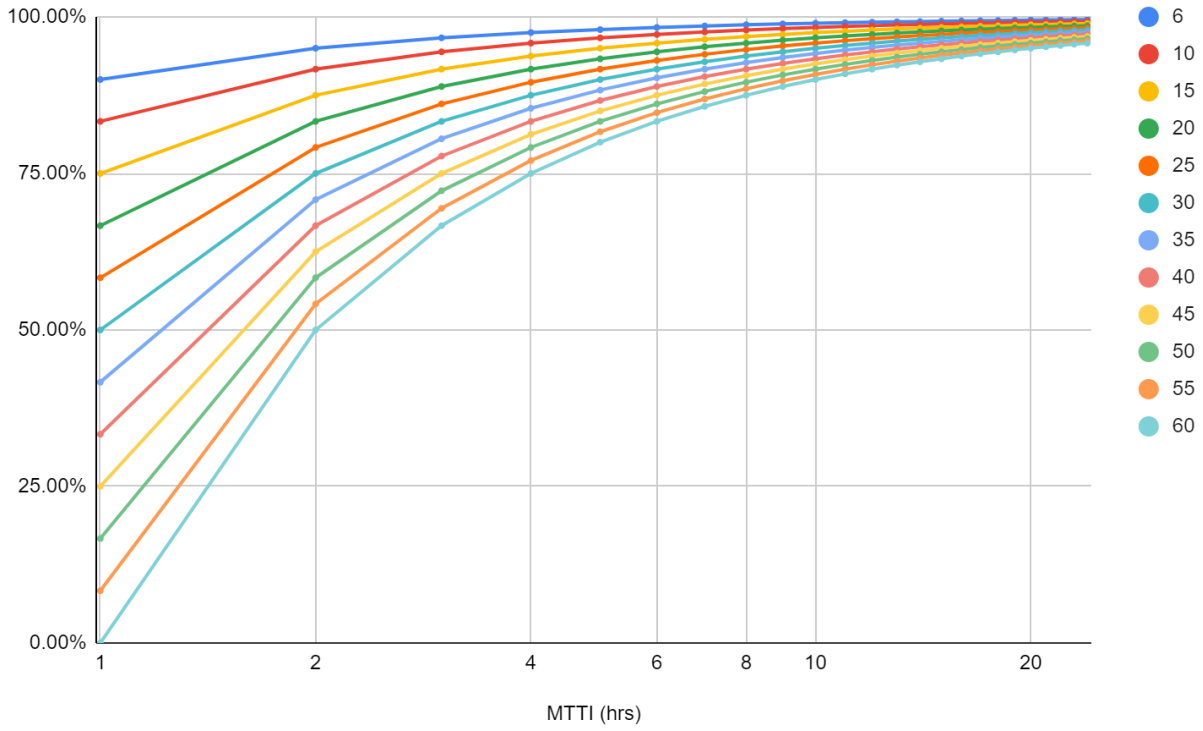


Figure 2: Various combinations of RT and MTTI that are negotiable between Hyperscalers & Suppliers

Note: This reliability modeling worksheet is used to create the above plot as a reference.

With this, Hyperscalers and suppliers can speak the same language in terms of RAS metrics and work out the metrics that best suit their needs.

3.1.2 Classic Reliability Metrics

There are classic reliability metrics that apply to any workload, such as:

1. **`Mean Time To Fail (MTTF):`** It is defined as a mean time to failure.
2. **`Mean Time To Repair (MTTR):`** It is defined as the mean time to repair prior to bringing the system back into production.

3. **`Mean Time Between Failure (MTBF):`** It is derived from the above two metrics (MTBF = MTTF+MTTR).
4. **Target UBB 'Annual Swap Rate' (ASR):** It is defined as the total number of UBB swaps per year due to failures.
5. **Target GPU/Accelerator Failure In-time (FIT):** It is defined as number of GPU/Accelerator failures within 10^9 hours.
6. **Silent Data Corruption (SDC) –** Guidelines and requirements refer to [9]

The classic reliability metrics are being mentioned to draw out the difference between the 2 sets of metrics, and to point out how they play a role in the goals to achieve baseline requirements.

3.2 System Level RAS Requirements

This section defines requirements for proper functioning of GPUs and accelerators with system-level Fault Management Solutions. This includes ensuring interoperability with software/firmware, hardware interfaces between the host and the accelerator devices, and adherence to overarching RAS goals where these devices play a role. High level requirements are listed here; the following sections have detailed requirements:
Error Reporting requirements are listed in Chapter 4.

Error injection capabilities are required. Hyperscalers use them to test error logging, error analysis and RAS actions. Detailed requirements are listed in Chapter 5.

3.2.1 Error Containment

Hardware errors may cause data corruption. To minimize the impact of corrupted data, GPUs and accelerator devices must contain the propagation of corrupted data, including:

1. Log and report an error when data corruption is detected (e.g. log poison creation), following the requirements in Chapter 4 Error Reporting. This allows identification of the faulty FRU and software mitigation actions to prevent use of corrupted data (e.g. mapping out resource).
2. Track data or transactions, including ingress data, marked as corrupted across all applicable places (e.g. caches, memory, storage, on the GPU) to prevent later consumption by applications.
    a. Includes when the corrupt data is not immediately consumed

   b. This may be implemented by marking the corrupted data as poisoned.
   c. Poison signatures in memory must be robust such that they survive a multi-bit memory failure or such that the corruption is re-detected and re-marked with poison.
3. Fail operations that are not successfully completed due to an error (e.g. PCIe completion with UR/CA, ingress transactions, or DMA operations).
4. Prevent consumption of data when corruption is detected by the GPU.
   a. Includes preventing internal compute units from using the corrupted data
   b. Includes indicating data is corrupted using poisoned transactions egressing from the GPU
5. Log and report an error when data either marked or detected as corrupted is consumed (e.g. log poison consumption), following the requirements in Chapter 4 Error Reporting. This indicates what caused the application to fail.
   a. Hyperscalers strongly desire that the location of the consumption get logged in a manner that it can be correlated with a prior poison generation event.

GPU suppliers are strongly encouraged to recover from uncorrected errors through retries or other software recovery mechanisms that are adequate to the workload type to reduce the blast radius of these errors. For instance, AI inference workloads may be recoverable by accelerator core only resets or Function Level Resets, while AI training workloads may be recoverable only by device level resets, resulting in a rollback to the previous saved checkpoint to resume operation after the reset.

However, in cases where the scope of the corruption cannot be bounded, other containment mechanisms such as viral or device resets must be used to prevent consuming and committing corrupted data to non-volatile storage.

3.2.2 Uncorrected Error Handling

The GPU/accelerator must provide a recovery mechanism that limits workload disruption/blast radius to reasonable extents based on the nature of the workload and the hardware implementation.
For instance, the recovery mechanism for workloads that are synchronous in nature may result in hyperscaler solutions to roll back to the previous checkpoint. Other possible recovery methods are quiescing and resetting the accelerator core, functional level reset, DPC, or machine reset in increasing blast radius order.
This document recommends recovery from uncorrected errors in two manners:

- Resume execution with spare resources

- Continue execution with degraded resources

3.2.2.1 Resumption with Spare Resources

Some failures may be addressed by off lining the defective resource and resuming with spare resources available in the device. For instance, if certain memory locations are defective, pages or entire rows can be off lined and swapped with healthy memory. When supported, the offline-and-swap process must minimize workload disruption.

3.2.2.2 Resumption with Degraded Performance

When spare resources are not available, certain failures may be addressed by continuing execution with reduced resources. However, this may impact performance to prohibitive levels depending on the type of workload. For instance, applying this recovery option for large training jobs may not be acceptable since its completion time will be impacted by the slowest device in the cluster.

For cases where continuing execution with degraded performance is viable, GPUs/accelerators shall notify the event to software and provide configurable options to allow Hyperscalers to:

- Stop execution immediately.
- Resume execution from nearest checkpoint after the faulty device is swapped out.
- Continue execution at degraded performance.

3.2.2.3 Graceful Handling of Memory Failures

This section provides system-level requirements for gracefully handling memory failures that are visible to the system. Note that corresponding error containment (3.2.1 Error Containment), and memory RAS requirements (3.3, Memory Fault Handling) have been previously covered and will not be repeated in this section.

If the GPU/Accelerator can detect failing memory ranges in its internal memory (e.g. HBM), it shall prevent the use of the faulty range. Examples include:

- Trigger a RAS action that mitigates the failure. Recommended actions are described in Section 3.3, Memory Fault Handling.
- Map out of the faulty memory range by the software to prevent any application from using it.
- Report an error with failing FRU identification and notify software that the operation failed.

The options above are listed in order of preference. RAS actions that mitigate failure are strongly preferred to replacing a FRU.

3.2.3 Corrected Error Storm Handling

Corrected error storms can be caused by many sources, such as memory and PCIe. These sources can generate corrected errors at a rate that is too high for the system to handle without impacting its performance. For instance, high rates of error interrupts may prevent the OS from making forward progress or overwhelm the Hyperscalers fault management infrastructure. Therefore, GPUs and accelerator devices must support capabilities to limit the impact of error storms across all possible sources of corrected errors.

Recommended options for handling corrected error storms include:

1. Rate-limiting the logging of new errors in hardware, which allows the oldest error to be logged but inhibits logging of new errors until a programmable time interval elapses. The rate limits should be chosen to still allow collection of a representative sample of the errors.
2. Switching to polling once an excessive rate of interrupts or notifications is detected. This approach requires a method to disable interrupts/notifications during an error storm, and to re-enable them once the storm subsides. Implementations may require more sophisticated software, but it is a common approach that still allows reporting of a representative sample of errors.
3. Implementing thresholding counters, which only generate interrupts or notifications when a specific threshold is reached. These counters may include leaky-bucket functions where error counts are periodically decremented at a pre-defined rate. This approach is not optimal for reporting errors that happen at low rates and may result in a non-representative sample of errors.

Hyperscalers strongly prefer programmable rate-limiting capabilities for corrected errors over the other two solutions. GPU suppliers are strongly advised against predicting crashes using corrected error counts alone since the counts itself do not provide information about the nature of the errors. For instance, many single-

bit errors in memory are generally benign and do not indicate that a crash is likely to happen, while a single error that affects a large area of the memory array may be more useful in predicting a crash. Robust crash predictions from memory errors evaluate the location of the errors in memory and the pattern of bit flips that represent specific failures in the memory.

Note that this section does not cover uncorrected error storms, since a high rate of uncorrected errors typically indicates that the hardware is about to crash.

3.2.4 PCIe Subsystem Fault Handling

GPUs/accelerators shall handle PCIe link faults in a manner that minimizes system-level service interruptions and data corruption. Additionally, GPUs/accelerators must support logging and reporting of system performance events such as PCIe error storms and PCIe link downgrades. This document assumes that systems support error handling capabilities defined in the PCI Express Base Specification, such as: PCI Express Link CRC, PCI Express CRC Check and Retry, PCI Express Retraining and Recovery, and Poison TLP handling.

The following subsections cover specific requirements for properly reporting and handling faults in the PCIe subsystem.

3.2.4.1 PCIe Advanced Error Reporting (AER)

GPUs/Accelerators must support PCIe error reporting as defined in the PCI Express Specifications, including Advanced Error Reporting (AER) capabilities. Although PCIe AER support is optional in the PCI Express specification, it is mandatory for GPU/Accelerator-based system designs aiming to be compliant with this specification.

3.2.4.2 Outbound Completions with UR/CA Error

GPUs/Accelerators shall complete non-posted requests with Unsupported Requests/Completer Abort (UR/CA) in case of a fault as defined in the PCIe Express Base Specification (e.g. section 6.2.11.3).

3.2.4.3 Inbound Completions with UR/CA Error

GPUs/Accelerators shall treat incoming UR and CA responses as errors. The UR or CA response must not be treated as valid data and must be handled as per the PCIe Specification (e.g. section 6.2.3.2.5). For instance, if the GPU receives a completion with UR while receiving data as part of DMA operation, the GPU must not consume the DMA buffer data as good data.

3.2.4.4 Handling Completion Timeout (CTO) Errors

When GPUs/Accelerators do not receive a response for a non-posted request, it shall handle it as a CTO error as per the PCI Express Base specification. Additionally, GPUs/Accelerators shall not resume normal execution if recovery attempts are exhausted.

3.2.4.5 PCIe link Bandwidth Change Reporting

As per PCI Express Base specification, link speed and width are allowed to change if the link is not reliable. The PCIe sub-system e.g., root ports and downstream ports shall report such events utilizing the "Link Bandwidth Management" and "Link Autonomous Bandwidth Management" capabilities as described in the PCI Express Base specification (Link Speed Management section).

3.2.4.6 PCIe Downstream Port Containment (DPC)

The GPU/Accelerator PCIe sub-system shall support Downstream Port Containment (DPC) where applicable (e.g. PCIe switches) and trigger as close as possible to the source of the error. Additionally, the GPU and corresponding device drivers should make every effort to successfully retrain the faulty PCIe link when the DPC event is cleared.

GPU-level requirements are not sufficient to support the full DPC flow. There are additional aspects of DPC functions that shall be considered while developing the solution as follows:

- Host-OS based AER/eDPC/Hot-plug drivers must support "PCIe FW specification (e.g. _OSC)" requirements including EDR (Error Disconnect Recover Supported). This is bit 7 "Error Disconnect Recover Supported" in _OSC support. See page 50 of the PCI Firmware Specification, version 3.3, for

more information. This requirement is to support logical surprise removal of the device when a DPC event occurs.

- Host-OS/Drivers need to ensure they handle cases where a configuration read fails and returns all 1's such as when doing a read after DPC brings the link down as described in the implementation note titled "DATA VALUE OF ALL 1'S" from section 6.2.11, "Downstream Port Containment (DPC)".

3.2.4.7 End-to-end Cyclic Redundancy Check (ECRC)

GPU/Accelerator systems are often designed with a multi-level PCIe switch hierarchy to interconnect multiple devices. As such, GPUs/accelerators must support PCIe End-to-end Cycle Redundancy Check (ECRC) as specified in the PCI Express Base specification ("End-to-End Data Integrity" section), which includes support for ECRC generation, ECRC check, and ECRC error reporting. ECRC is required even though this feature is optional in the PCI Express specification.

3.2.5 Other System-Level RAS Features

This section lists additional system level requirements for other interfaces on the UBB.

3.2.5.1 PCIe Retimer Error Handling

PCIe retimers must allow Hyperscalers to retrieve error logs and configure error handling features supported by the device. Any detectable link anomaly should be reported per the requirements in Chapter 4, Error Reporting. Additionally, GPUs/accelerators that include PCIe retimers should exercise diagnostic features in the retimers, whenever possible, and expose the results to the Hyperscalers' fault management infrastructure.

For instance, if a retimer negotiates a lower-than-expected link speed, the event should be logged and made visible to the hyperscaler. Information such as the eye opening and equalization levels may help in root-cause problems related to PCIe retimers.

3.2.5.2 XLink and Switch Error Configuration and Handling

Hyperscalers also require managing faults in vendor-specific Xlinks and switches. GPUs/accelerators and Xlink switches must log Xlink failures with sufficient information to identify which link(s) failed and what type

of failure occurred. Suppliers must provide means (e.g. tooling, APIs) to access and configure Xlink error handling features when supported.

Xlinks must have sufficient reliability to meet the applicable goals described in Section **Error! Reference source not found.**, **Error! Reference source not found.**. Features like link retries, link width reductions, link frequency reductions, and the ability to route around failing links may assist in meeting these goals.

3.3 Memory Fault Handling

This section describes recommended Memory RAS features and considerations for GPU/Accelerator designs to meet the reliability targets outlined in Section **Error! Reference source not found.**, **Error! Reference source not found.**. These features are not mandatory, as GPU suppliers are responsible for ensuring reliability while making cost tradeoff decisions. Different memory technologies, such as HBM, GDDR, LPDDR and DDR may have unique RAS features that are best left to the supplier to assess. However, Hyperscalers encourage GPU suppliers to develop new RAS features beyond the baseline requirements covered in previous sections of this document to make memory more reliable.

Memory on the host is outside the scope of this section.

3.3.1 Memory Error Correction Capabilities

GPU & Accelerator modules are required to support robust memory error correction capabilities to meet Hyperscalers' reliability targets as outlined in section 3.1.1 Baseline Reliability Metrics for Concurrent Applications. For example, single bit and single symbol errors are recommended to be 100% corrected. Where applicable, error detection and correction coverage are expected to comply with industry specifications, e.g., JEDEC's HBM specification.

GPU designers are encouraged to innovate to create better memory error detection and correction solutions. GPU suppliers should consider the strength of the ECC schemes in their designs and ensure they select ECC schemes that are sufficiently strong.

For example, about 77% of DDR4 errors are single bit errors, reference [12] and could be corrected by Single Error Correct, Double Error Detect (SECDED) ECC for a roughly 10x improvement over unprotected memory. This is significantly weaker than ECC that can correct errors from a single device such as DDR5 x10 chip kill

ECC which is roughly 1000x more reliable than unprotected memory. For example, for large-scale GPU clusters, SECDED based ECC may not be sufficient for fault coverage.

It is imperative that GPU designers understand the expected failure mode and rates of different memory technologies from small buffers to large main memories and that they design appropriate memory error correction into their products necessary to meet the reliability goals described in this document.

3.3.2 Memory Sparing Repair Capabilities

Some memory technologies support repair capabilities where a spare element can be swapped in over a failing memory element. These are valuable because:

1. The failing memory can be repaired (swapped) with no human intervention
2. It may be possible to repair the element at runtime, preventing an interruption

There are usually hard and soft versions of these capabilities:

1. Hard repair
    a. Blows a fuse and is permanent
    b. More hard sparing operations may be possible than soft sparing depending on the memory technology
    c. Used mainly at boot time
2. Soft repair
    a. Sets a latch that spares the element until the next reset or power cycle
    b. Since it does not blow a fuse, it is faster
    c. Used at runtime if the memory controller supports it
    d. Useful for testing error handling code since it is reversable

Requirement of these features is subject to availability as per the industry specifications, e.g., JEDEC's HBM specification though suppliers are strongly encouraged to innovate in this area. Post Package Repair (PPR) which is supported in DDR4 memory and later, is an example of a memory repair capability.

Runtime repairs need to be fast enough to prevent triggering transaction timers in the hardware.

The repairs should be used when the GPU supplier determines that a memory error:

1. Represents a risk of causing a crash
2. Causes a noticeable performance degradation
3. The failure condition would be remedied by the repair.

For example, a row repair might remedy a word-line failure but not a sense amplifier failure.

It is recommended that GPU memory controllers provide:

- A capability where memory is tested and repaired as appropriate during boots that the GPU boots with healthy memory.
    - Memory suppliers might have tests specifically designed to detect failures in their parts
    - These often detect failures that other memory tests miss and can drive repairs
- A capability to monitor and repair memory errors during run-time.
    - This may be implemented with Memory Scrubbing as described below.

3.3.3 Memory Scrubbing

Hyperscalers recommend that GPU designers provide functionality that:

- Prevents errors from accumulating in memory that is not used often
- Detects uncorrected errors to:
    - Provides some warnings before applications try to use the affected data
    - Allows the OS/driver/application to recognize bad pages of memory and avoid using them
- Supports innovative RAS features that helps us mitigate memory failures

Memory scrubbing was originally designed to sweep through memory periodically, reading the memory, checking ECC and writing back corrected data for any bits that were found to have been flipped. The intent was to prevent single bit errors from accumulating in memory that was used infrequently. If too many bits were flipped in infrequently used memory, you might get enough of them to cause a crash when you start using the memory. Frequently used memory was less of a concern because it was accessed frequently, and corrections for bit flips were written back whenever the memory was accessed.

A 24-hour scrub rate is typically used, but Hyperscalers recommend that it be programmable to be much faster if desired. 24-hour scrub periods were originally suggested for the first hardware scrubbers because Hyperscalers were confident that the slow rate would not noticeably impact performance but was fast enough to prevent accumulation of corrected errors.

Memory scrubbing has become a more important feature over time. It is also used to detect uncorrected errors in memory before they are consumed. It is easier to recover from the scrubber finding an uncorrected error than recovering from an application attempting to consume poison. Since the memory scrubber is the

primary way Hyperscalers can access memory independently of the applications running on the machine, its uses have expanded to support other RAS features and is sometimes used by firmware to independently assess the scope of a memory failure being used to determine which addresses have errors and if the errors are persistent.

Hyperscalers also recommend an on-demand function to scrub the memory prior to allocating it to an application.
An example implementation as specified in the JEDEC specification is called ECS (Error Check and Scrub) (HBM3 spec JESD238A). Errors detected during the scrubbing function could be corrected or uncorrected and shall be reported as defined in section 7.2.1.8 and 7.2.1.9, respectively. However, ECS as defined in DDR5 does not provide the locations of corrected bit in the DRAM, obscuring the failure patterns and making it harder to determine the scope of the failure. Hyperscalers strongly desire that the ECS data be collected and that the data about which bits are corrected is as detailed as possible.

3.3.4 Protection from Corruption on the Memory Bus

Hyperscalers need a way to distinguish between errors caused by the memory bus and errors caused by bit flips in the memory array. A bit flip in the command/address bus sent to memory may corrupt data. The GPU/accelerator memory controller shall include protection for command/address bus to minimize the risk of data corruption.

The most common method to protect against corrupted bus transactions is by supporting Command/Address Parity per JEDEC specifications (e.g., HBM3 spec JESD238A section 6.4.1). However, GPU designers are encouraged to innovate to add more protection.

3.3.5 Data Link Parity Protection

Some memory technologies support parity checks on data sent across the link to the memory. This helps prevent silent data corruption.
Hyperscalers expect GPU suppliers to provide protection against data corruption on the link whenever the memory technology supports it. See JEDEC specifications (e.g., HBM3 spec JESD238A section 6.4.2) for more information about data link parity.

3.3.6 Containment (e.g., Data Poisoning)

Uncorrected errors in memory must be contained such that applications do not consume corrupted data. These requirements are elaborated in section 3.2.1, Error Containment.

3.3.7 Memory Error Reporting

Memory failures cause a significant number of service interruptions and must be visible to Hyperscalers. See 4.2, Error Record Requirements for more information about error logging expectations.

3.4 GPU Internal RAS Features

GPUs/accelerators shall meet the baseline RAS feature requirements, including:

1. Meet the reliability requirements outlined in Section 3.1.1, Baseline Reliability Metrics for Concurrent Applications
2. Support error containment as described in Section 3.2.1, Error Containment
3. Log errors as described in Section 4, Error Reporting
4. Support error injection as described in Section 5, Error Injection.

Additionally, Hyperscalers strongly encourage suppliers to innovate and add RAS features beyond said baseline RAS requirements. The following subsections provide examples of additional RAS features desired by Hyperscalers. These are provided as guidelines and are not intended to be normative.

3.4.1 Internal Error Coverage

GPU suppliers are expected to be responsible for evaluating their design, identifying common failures from previous designs and ensuring current designs meet the reliability requirements listed in Section 3.1.1, Baseline Reliability Metrics for Concurrent Applications. This may involve evaluating common sources of errors such as soft errors. GPU suppliers are expected to understand the expected error rates of their fabrication technology. Large structures in silicon may have significant FIT rates and may need features such as parity to help prevent SDC, or ECC to enable soft errors to be corrected.

GPU suppliers should consider error detection and correction for large structures, including buffers and caches, and ensure that detected uncorrected errors are contained. These errors should also be logged so that the supplier can measure actual error rates reported via error logs and use the measured rates to drive improvements.

3.4.2 Other Radiation Tolerance Features

Hyperscalers encourage GPU/accelerator suppliers to evaluate the use of physical register structures that benefit from radiation-hardened latches. Error data from previous designs, if detailed enough, can guide decisions on when it may make sense to use this type of latches.

Hyperscalers also encourage GPU/accelerator suppliers to consider the effects of radiation on the physical logic layout. For example, ionizing radiation may affect adjacent memory cells. If an ECC protected structure, like a cache, is physically designed such that memory cells for the same ECC domain are adjacent, it is possible for a single radiation strike to flip multiple bits and cause an uncorrected error reference [13]. GPU/accelerator suppliers are encouraged to ensure the physical design of ECC protected structures prevent single radiation strikes to cause uncorrected errors in said structures.

3.4.3 Other Influences on Failure Rates

It is possible for other sources of semiconductor errors, beyond soft errors, to impact reliability. Examples include bulk silicon defects, electromigration, and hot carrier migration. GPU designers are expected to understand the sources of errors applicable to their design and ensure their products handle them adequately to meet the reliability goals in this document.

As silicon manufacturing geometries continue to become smaller, there are growing concerns about faster silicon wear out. GPU/accelerator suppliers should consider ways to monitor and report silicon wear out to make data-driven decisions on adding features to mitigate wear out.

4 Error Reporting

Hardware error record formats currently used on hyperscale deployments lack standardization and vary across Hyperscalers. Additionally, hardware errors can be reported by different agents, such as UEFI Firmware, the Accelerator Management Controller (e.g., AMC), or the Operating System.

This section standardizes error reporting capabilities and error record formats for accelerators. The goal is to allow common (supplier- and device-agnostic) record formats that are usable across different error reporting agents and data center operators, while allowing accelerated deployment of heterogenous hardware with minimal implementation and maintenance costs.
This document proposes using CPER as the hardware error record format and Redfish as the standardized transport protocol.

4.1 Error Reporting Requirements

Devices shall:

1. Report applicable hardware errors using the Common Platform Error Record (CPER) format defined in the Unified Extensible Firmware Interface (UEFI) specification[2].
2. Report error records over Redfish to the node's Management Controller (xMC) as the point of aggregation for all error records, specifically via LogEntry and/or Event resources as defined by DMTF[5].
3. Collect all errors that have been detected in a coalesced error log.

4.1.1 Error Logging Overwrite Rule Requirements

Ideal hardware supports error logging queues for each error severity. This allows us to apportion error logging storage to each error severity and allows us to ensure that we can always collect logs for the first error of a particular severity. In the case where there is a common error logging register used for all severities, the silicon must:

- Prevent error data from being overwritten by a new error of the same severity until the error is explicitly cleared. This prevents error logging registers from logging a new error while software is in the middle of reading them and thus prevents error logs from being corrupted.
- If an error logging register is used to log errors of multiple severities, it is permissible for the error logging register to be overwritten when a more severe error occurs. Software can prevent corrupting error logs by reading the severity of the error first, gathering data from the error logging registers into the error log and then verifying that the error severity did not change from the original read. If the error severity changed to a higher severity, the error logging software needs to throw away the previous data it read and re-read the error logging registers to capture the new, more severe error.

When an error log cannot be collected because of previous, uncleared errors, the error logging registers must set an overflow bit. This allows us to see that another error occurred but was not logged.

4.2 Error Record Requirements

CPER format consists of a record header followed by one or more fixed-length error record section descriptors, as described in the figure below. For each section descriptor, there is an associated variable-length section that contains either error data or informational data.

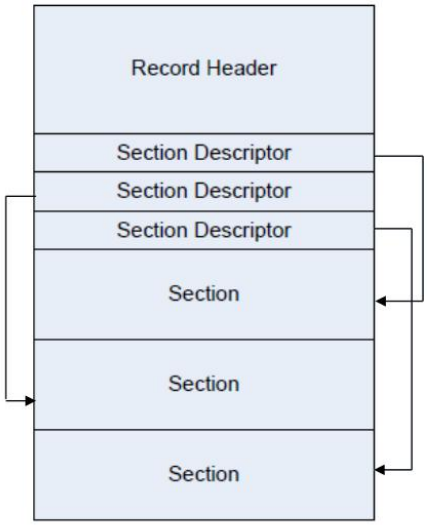


Figure 3: CPER Format

The CPER records shall:

- Allow for identification of the faulty Field Replaceable Unit (FRU).
- Contain all information necessary to fully categorize the error (hermetic) and does not require harvesting of additional error information.
- Implement versioning schemes.
- Be capable of containing an unbounded amount of data.
- Provide sufficient information to aid in root-cause analysis of hardware failures
- Include error records for all error severities supported by the hardware
- Cover all major silicon on the GPU device including memories

For instance, if a single accelerator fails to initialize during system boot, the device generates an error record in CPER format that includes: which accelerator(s) failed to initialize; the cause of this failure (e.g., failed PCIe

training, failed memory training, or boot ROM corruption); and any additional information relevant to diagnosing the issue (e.g., which step in PCIe training or memory training failed, any parameters necessary for the device supplier to further diagnose the issue). This CPER record can be communicated to the data center operator over Redfish via Event and/or Log services.

4.3 In-band Vs Out-of-Band Error Logging

Hyperscalers support Bare-Metal, Infrastructure and Application usage models as described in OCP Fault Management Infrastructure Requirements[6]. Hyperscalers that only support application usage models may want to log more in-band. Hyperscalers that support bare metal may want to log more OOB. Supporting these models requires that applicable error logs need to be available both in-band and Out of Band (OOB).

- The same error events need to be logged both in band and out of band
- Error logs must be sent to the driver and/or application when software needs to react to the error
    - Drive a software-controlled RAS feature
    - Handle a failure, e.g. recover from poison consumption
- Error logs must be logged out of band when software is unable to run on the host machine
    - Crash dumps
    - Fatal errors where the machine or GPU cannot come back up
- Error logging solutions must allow Hyperscalers to direct errors by error type and severity to in band and/or out of band logging agents.

5 Error Injection

5.1 Overview

Error handling and recovery code paths on Accelerator systems require thorough testing in Hyperscalers' environments. Accelerator systems have complex topologies with multiple components in their path, like switches, re-timers, and GPUs. The ability to exercise end-to-end error propagation and containment flows at the fleet level is critical for Hyperscalers to validate mitigation and repairability workflows.

Fault management features must be testable at scale, without requiring known bad hardware or special error injection hardware. Hyperscalers need error testing tools that can:

- Inject errors into hardware

    - Error injection typically requires several steps, including error implanting, detecting, triggering, and logging. The hardware detects the injected failure and exhibits the same behavior as when detecting naturally occurring errors.
- Spoof errors
    - Simulates error logging without implanting the error in the hardware.
    - System may not exhibit the same behavior as when detecting naturally occurring errors. However, error detail logging and signaling would still occur and allow Hyperscalers to validate infrastructure behavior.

The term "error injection" is used in this specification to imply two functions, unless otherwise explicitly specified:

1. Error Implanting
2. Triggering the implanted error

5.1.1 Error Implanting

Error implanting configures the device to cause an error condition to occur. For example, when injecting a memory error, implanting could configure logic to cause bits to be flipped when memory is accessed.

5.1.2 Error Triggering

Error triggering causes a previously implanted error to occur. For example, if a memory error was implanted such that bits from a memory access would be flipped upon accessing a particular address, accessing that address would trigger the memory error. In some cases, there may not be a distinction between implanting and triggering the error.

When an error is triggered, hardware acts as if a real failure occurred, including logging the failure and causing any side effects that would be expected with a real error. For example, if an uncorrected memory error is injected and triggered, Hyperscalers expect the memory controller to poison the access if the memory controller supports poison.

5.1.3 Error Spoofing

Spoofing is useful for testing the flow of error data and analysis of the error data but does not cause the hardware side-effects of the error condition. For this reason, error injection is preferred over spoofing for testing the behavior of the system and for testing RAS features.

Since spoofing only causes error logging registers to contain data as if an error occurred, there is no such thing as a persistent spoofed error. Spoofed errors are cleared when the hardware error logging registers are cleared.

5.2 Requirements for Testing Error Handling

1. GPUs/accelerators must include error injection capabilities for Hyperscalers to develop, refine, and validate system-level error detection, signaling, logging, containment, and mitigation/RAS Feature workflows.
2. UBB subsystem must include error spoofing capabilities for similar purposes as requirement 1. above.
3. GPUs shall provide a way to enable/disable the error injection feature. If early development GPUs do not have a mechanism to enable error injection, error injection shall be enabled by default until the mechanism is implemented.
4. During the system development phase, error spoofing features in the UBBs shall be enabled by default.
5. During the system production phase, GPUs/accelerators shall provide a secure mechanism for querying and configuring the error injection and spoofing features. Section 5.4 Error Injection Security Requirements covers the recommended mechanisms to meet this requirement. Please refer to Figure 7: High-level flow for Redfish-based Error Injection for a reference implementation compliant with this requirement.
6. GPUs/accelerators shall provide a mechanism to discover and query error injection/spoofing capabilities via in-band and out-of-band interfaces, including:
    a. Support for different HW error types at the IP Level, SOC Level, and on the platform level (e.g., HBM, PCIe).
    b. Supported error severities for each error type.
    c. Support for transient or permanent injections for each error type and severity.

7. Error injection/spoofing methods and interfaces shall be abstracted from supplier-specific implementations from the Hyperscalers point of view. Keeping error injection payloads supplier-specific is acceptable if the solution allows Hyperscalers to develop supplier-agnostic tests.
8. The solution shall NOT depend solely on installing unverified error injection tool binaries on the host to perform the injection requests to the accelerator. CSPs and vendor security teams shall define tool security requirements based on the approved error injection threat model. In the absence of a standardized threat model at the time of publication, vendors and CSPs may address this independently of this specification.

Figure 8 illustrates an example framework that would meet these abstraction requirements for error injections during testing of the hyperscaler's fault management infrastructure. In the diagram, the OCP standardized Error Injection API allows hyperscaler automation to orchestrate error injection tests remotely to target machines via the network. In-band or out-of-band services forward the error injection payloads to the corresponding GPU/Accelerator components. Error injection payloads contain a sequence of error injection entries that may be supplier specific.

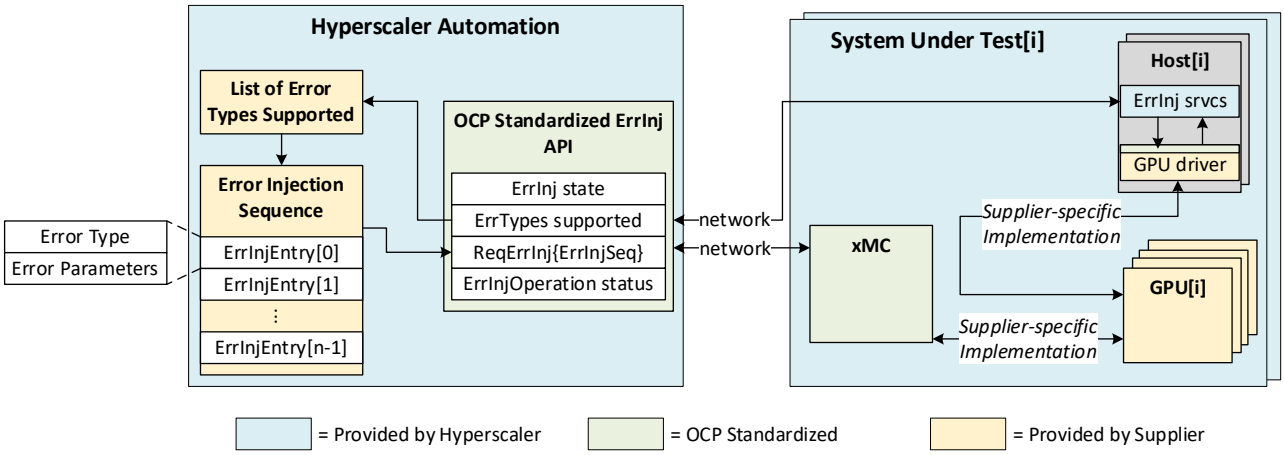


Figure 8: Example of Error Injection Framework

Please refer to 13.2 Appendix B – Error Injection Support in Redfish for a reference implementation that meets these requirements.

5.3 Error Injection Coverage

1. GPUs/accelerators shall provide error injection coverage for a representative set of the errors expected to occur naturally in the production environment. This coverage shall include error types

required for triggering corresponding mitigation workflows supported by the RAS feature set of the device. Please refer to Table 2: Error Injection Coverage below for more details.

2. GPUs/accelerator suppliers shall make reasonable efforts to support error injections that exercise the error reporting flows of naturally occurring errors, as this exposes system behavior under these errors.
3. For error injection types that cannot meet requirement 2 above, GPUs/accelerator suppliers shall provide alternative mechanisms for Hyperscalers to verify the error handling flows, such as error spoofing.
4. Spoofing is optional when error injection is available but still recommended.

The following table defines the minimum set of error coverage expected by Hyperscalers. Refer to Chapter 3, RAS Feature Requirements for more details.

Table 2: Error Injection Coverage

Error Type	Component Coverage	Notes
Memory errors	GPUs/accelerator Memory (e.g. HBM) errors	Host DDR memory is excluded.
PCIe errors	PCIe Switches, PCIe Re-timers, PCIe Endpoint Devices (e.g. PCIe Network devices)	Support for PCIe AER error injection support. Where applicable, PCIe Re-timers require special attention as error handling is outside the scope of the PCIe specification.
GPU Internal Errors	Internal GPU micro controller exceptions, GPU SRAM, Cache, accelerator core, and firmware faults	
XLinks and Switches	GPU Interconnects, Interconnect Switches	
Platform Specific Errors	FPGA Specific Errors, Thermal, USB VNIC, I2C, I3C	
Supplier-defined errors	Other supplier-specific IP errors	As recommended by GPU/Accelerator supplier

5.4 Error Injection Security Requirements

1. Production-grade devices must have error injection disabled by default.

2. Production-grade devices capable of error injection must provide a mechanism to securely enable error injection mode. Two methods are discussed in this document revision:
    a. Secure Unlock
    b. Alternate Authorization

5.4.1 Secure Unlock

This method modifies production-like firmware to enable error injections in the GPU/Accelerator. Figure 4: Overview of Secure Unlocking provides a high-level overview of the secure unlock flow.

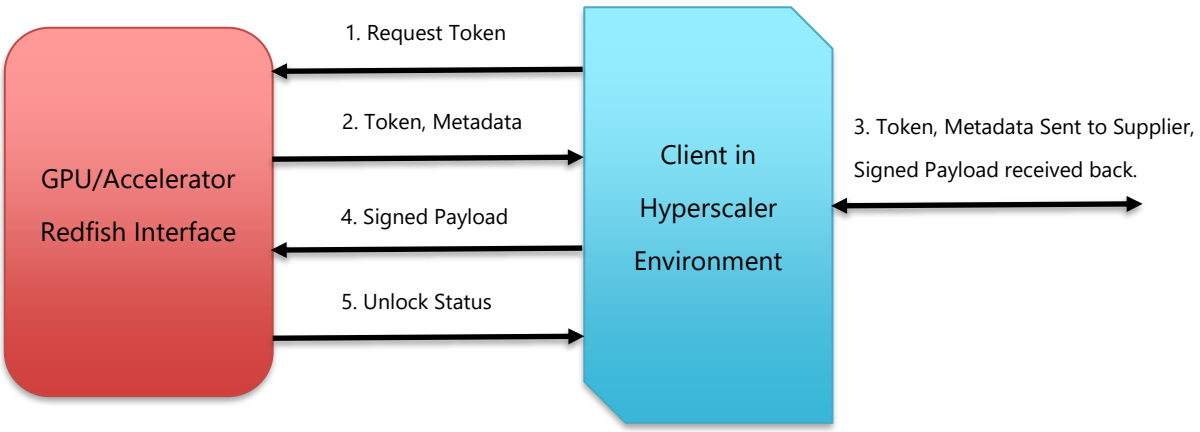


Figure 4: Overview of Secure Unlocking

To support the secure unlock method, the device must meet the following requirements:

1. The device must provide a mechanism to query current lock status, such as “locked,” or “unlocked.”
2. The device must provide a mechanism to request and retrieve an unlock token or challenge payload from the accelerator complex (e.g., accelerator or AMC).
3. The device must provide a mechanism to send the unlock payload or challenge response to the accelerator. What this payload is, is left up to the supplier. It may be a small unlock token, or an entire firmware drop. This payload or response must only be able to unlock the device for which it is requested (e.g., per-part).

Additionally, the device supplier must enable an out-of-band process to receive unlock token requests and send the corresponding unlock payloads to the hyperscaler. The definition of this out-of-band process is outside the scope of this document.

When testing cycles are complete, the hyperscaler may choose to return the device to locked mode. One method is by re-flashing new firmware following the standard firmware update process defined in the OCP GPU FW Update Specification[4]. Another method is to issue appropriate command(s) to the point of authorization (BMC or accelerator complex) to invalidate the token and end the unlock session.

5.4.2 Alternate Authorization

For cases when fully unlocking the device or contacting the supplier to enable error injection is not feasible, an alternate authorization method, such as using pre-provisioned Hyperscaler-owned keys or certificates in the platform, may be used to authenticate the enablement of error injection mode. The actual implementation may be supplier-specific to meet appropriate product-level threat models, but must meet the following requirements:

1. The implementation must provide a mechanism to discover the error injection authorization method and any required attributes.
2. The implementation must provide an interface to control error injection mode, including enabling and disabling error injection mode and querying the current status of error injection mode.
3. The implementation must provide a mechanism to notify the Hyperscaler infrastructure that error injection mode has been enabled on production grade hardware and firmware.
4. The authorization mechanism must be part-specific and only applicable to the device for which it is requested. It must NOT authorize error injection in any other GPU accelerator instance.
5. The authorization mechanism must be scalable and deployable in Hyperscale environments.

6 Crash and Device Debug Dumps

6.1 Introduction

When hardware faults occur at scale, time to root-cause is critical to meet Service Level Agreements (SLA) and reduce the impact on customer experience. Certain hardware faults only manifest in production when exposed to specific workloads. They are often spread across the fleet and tend to show up as we scale the number of devices deployed and scale the number of workloads on the devices. Debug situations almost never persist across a reset and generally require considerable effort to reproduce. Since the time to reproduce these faults in debug or quarantine pools is often non-deterministic, it is critical to gather the necessary debug data at the time of failure.

There are two kinds of dumps: Crash dump and Debug Dump. This section outlines the requirements for crash and debug dumps in terms of reliability, standardization of dump records, sensitivity of dump data, and dump analysis requirements.

6.2 Crash Dumps

Failure events that require crash dumps:

- Fatal errors that caused system crash, e.g.,
    - GPU or accelerator hangs – Hardware asserts.
        - Triggered by internal conditions defined by the supplier when required for root-cause analysis.
    - CPU hangs.
    - X-links fatal errors.
    - X-switch fatal errors.

Crash dumps for fatal errors involve the following:

- Dump collection executed at the time of failure.
- Part of normal fatal error flow (e.g., no special unlocks or other pre-configurations).
- Must not contain customer or workload data.
- The output from the analysis of crash dumps must enable automated responses (e.g., repair actions, self-healing RAS actions) to fatal errors when appropriate.
    - Hyperscalers may request assistance from suppliers when automated responses are not feasible.
- Better data captured in crash dumps reduces the need for debugging/debug dumps.

Crash dump trigger options:

Collection of crash dumps should be initiated based on the following conditions:

a. Automatic collection at the time failure – On uncorrectable, fatal error severity
b. On-demand based on a standardized API request. More details on this API will be included in a future revision of this document.
    i. Used for debugging unexpected behavior that doesn't cause a crash.
    ii. Collects contextual data useful for interactive debugging.

6.2.1 Crash Dump Requirements

This section defines the requirements for crash dumps:

- Must contain detailed information about errors in hardware and must not contain sensitive information, e.g. workload data, customer data and Personal Identifiable Information (PII).
    - This allows for crash dumps to be available even in confidential compute environments.
- Should be persistent across resets, need to store the dumps in non-volatile storage.
- Should be useable across the NPI lifecycle:
    - Including early in development, when Hyperscalers first get parts.
    - Initial crash dump tools should be useful for debugging.
    - Capabilities may increase as we gain experience with the tool.
    - In production, the output of crash dump tools must enable automated responses to errors.
- Should contain all the data needed for diagnostics, including data from components on the UBB (e.g. clock, power, thermal, PCIe/Xlink switches and re-timers).
- Crash dumps must be collected at the time of Uncorrectable, Fatal errors automatically; this should be set as the default behavior in the system configuration. In addition, it is necessary to have support for on-demand crash dumps using an API for all the device dumps listed in table 3.
- If there are any other errors detected beyond the error(s) that triggered the crash dump, they should also be collected and included in the error logs as described in 4 Error Reporting.
- Crash dumps must be decoded and analyzed like any other non-fatal errors reported. Designers may refer to the OCP Fault Management Infrastructure Requirements ver 1.0[6] for further guidance.

The following table details the system and component-level crash dump standardized in this specification, assuming a generic OCP Universal Baseboard (UBB) design. Note that component-level debug dumps may vary across platforms. The following table lists crash dump types and GUIDs used for on-demand collection. These will be described when the API for on-demand collection is defined in a future revision.

Table 3: Dump Types and GUIDs for on-demand collection API

Dump type	Description	GUID for on-demand dump collection.
GPU crash dump	Includes GPC, cache, HBM status information	522820b3-7e0f-4176-9863-a3282788a5cd
PCIe Retimer dump	Includes LTSSM, figure of merit (FoM) data	4ba4100c-e35f-4cb5-a6cd-717f09a0df1d
FPGA dump	Includes FPGA-specific register dumps	5d935107-7c49-4e8c-9d83-9e85b1889de0
ROT dump	ROT logs	68067014-e157-4d96-a325-94989e633e1b
X-link Switch crash dump	Switch error register dumps (driven by suppliers)	b647d9ac-9388-47a6-9712-d044a7da09e2
PCIe Switch crash dump	Switch error register dumps (driven by suppliers)	f8598ac6-6a7c-4b31-bbb2-a6143018d540
Manager level dump	AMC OS level logs to report health of AMC fatal errors	26f72767-bf32-43fe-9879-4ad51c323405

The OCP Hardware Fault Management team has introduced the concept of a Common Platform Action Descriptor (CPAD)[16] that provides a standardized way for analyzers to suggest RAS actions, including actions for crash dumps. The CPAD format will be defined by the RAS API team[11] and this document will be updated when it is sufficiently defined.

6.3 Debug Dumps

- Debug dumps
  - Debug dumps, such as scan dumps and trace dumps, may be provided, in addition to crash dumps, for exceptional failure cases where crash dumps are insufficient for root-cause analysis.
  - Debug dump collection may require prior configuration (e.g., unlocking) before execution.
    - The data captured in a debug dump is tuned for the problem being debugged.
    - For example, an in-silicon logic analysis block might be configured to trigger and capture a trace log for a particular problem.
    - Hyperscalers may require help from suppliers for configuring debug dumps.
  - Debug dumps contain more detailed data than crash dumps.
    - May contain state of flip-flops, internal logic, that is not visible to software.
    - State dumps (e.g., scan dumps) are static snapshots of hardware internal state.

  - Trace logs capture a sequence of events or instructions executed by the device, often stored in a buffer (e.g., a trace buffer).
  - Suppliers may come up with other types of dumps as needed and will define how detailed the dumps need to be.

In cases where existing crash dumps do not provide resolution, additional improvements should be implemented in order of priority below:

1. Crash dump improvements.
2. Analysis tool improvements.
3. Supplier-specific debug tools.
4. Debug dumps.

6.3.1 Requirements for Debug Dumps

- Debug dumps should have a CPER wrapper.
- Debug dumps should use the same software interfaces as non-fatal error logs and crash dumps, when feasible.
- Analysis tools should indicate that the CPER contains debug data that needs to be passed to the supplier.
- Debug dumps are not intended to be analyzed by the Hyperscaler. Instead, device suppliers are the primary consumers of these dumps for analysis and issue resolution.
- Support for custom automatic debug collection on certain hardware conditions (e.g., performance issues, other internal error conditions), may require hyperscaler approval. Each hyperscaler will have their own process for approving debug data collection.
- Trigger based on internal conditions determined by suppliers when required for root-cause analysis.
- Firmware deployments to execute debug dumps should be avoided.

6.4 Dump Data Sensitivity

Retrieving debug dumps from deployments running production workloads may be dependent on the sensitivity of the data collected. In many cases, debug dumps are not be acceptable to be used on devices that are running end-user code. If the problem is severe enough, Hyperscalers may create a quarantine pool of machines that run end-user workloads with their explicit permission. This tends to occur when a particular issue reproduces only when the end-user workload is running.

- Debug dumps should not contain customer information by default.

- Configurable options can be supported under specific conditions, e.g., first-party workloads or with end-user approval.
- End-user approval may be based on authentication artifacts that indicate that the machine is configured to allow dumps. End-users may choose to proceed with execution with the knowledge that dumps are allowed (to replicate specific scenarios where crashes are observed only in confidential execution mode).
    - The authentication details will not be covered in this specification.

Future revisions of this specification will include security guidelines corresponding to the sensitivity of the data extracted by a particular debug dump type. Please refer to Appendix D – Debug Data Taxonomy for a sample classification of debug data sensitivity.

6.5 Dump Reliability

Crash and debug dumps are used to gather error data for the most severe failures. Special consideration must be given to ensuring that dumps can be reliably collected. Dump collection must work under significant hardware failures:

- When memory devices like DDR or HBM have failed.
- When I/O devices such as Accelerators or NVMe storage have failed.
- When high speed links between subcomponents (chiplets, devices, etc.) have failed.
- When compute engines that run applications are too broken to run code.

This implies that a management agent or out-of-band path is used to collect the dumps. It also implies sufficient error injection to test collection of dumps.

7 Analysis Tool Requirements

Analysis tools are strongly encouraged to comply with the OCP Fault Management Infrastructure Requirements[6]. Hyperscalers expect analysis tools that analyze all severities of errors from corrected errors to crash dumps. The analysis tools should recognize failures, determine which FRUs are involved, and provide suggestions for mitigating the failure. Time series analysis may be useful for analyzing error logs, including crash dumps where the analyzer processes errors prior to the crash dump to help determine what caused the crash.

Error analysis should be automated whenever possible, prioritizing frequently occurring errors and errors with a large blast radius. The breadth and quality of crash dump analysis results serve as an important

competitive differentiator. Crash dump analysis should describe the error and how to mitigate it, including RAS actions and repair actions.

Crash dump analysis can request that a CSP contact a supplier to look at a particular error to assist with debugging.

Error analysis tools should recognize debug dumps but are not expected to always analyze them. The analysis tool should, at a minimum, indicate that it detected a debug dump and suggest that the CSP contact the supplier for help in understanding the debug dump.

Hyperscalers integrate and deploy supplier analysis tools into their infrastructure to interpret the CPER records reported. The following figure provides an overview of error analysis.

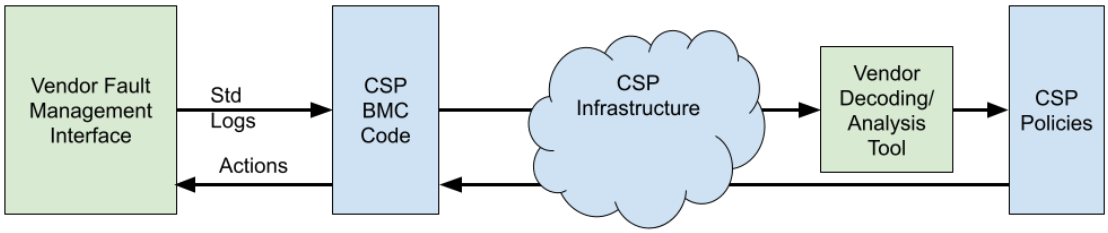


Figure 5: Overview of Error Analysis

CPER records are expected to differ across devices from different suppliers, requiring Hyperscalers to properly route them to the corresponding analysis tool instance. The figure below demonstrates an example scenario where hypothetical CPER Error Logs are appropriately routed and decoded for vendor analysis.

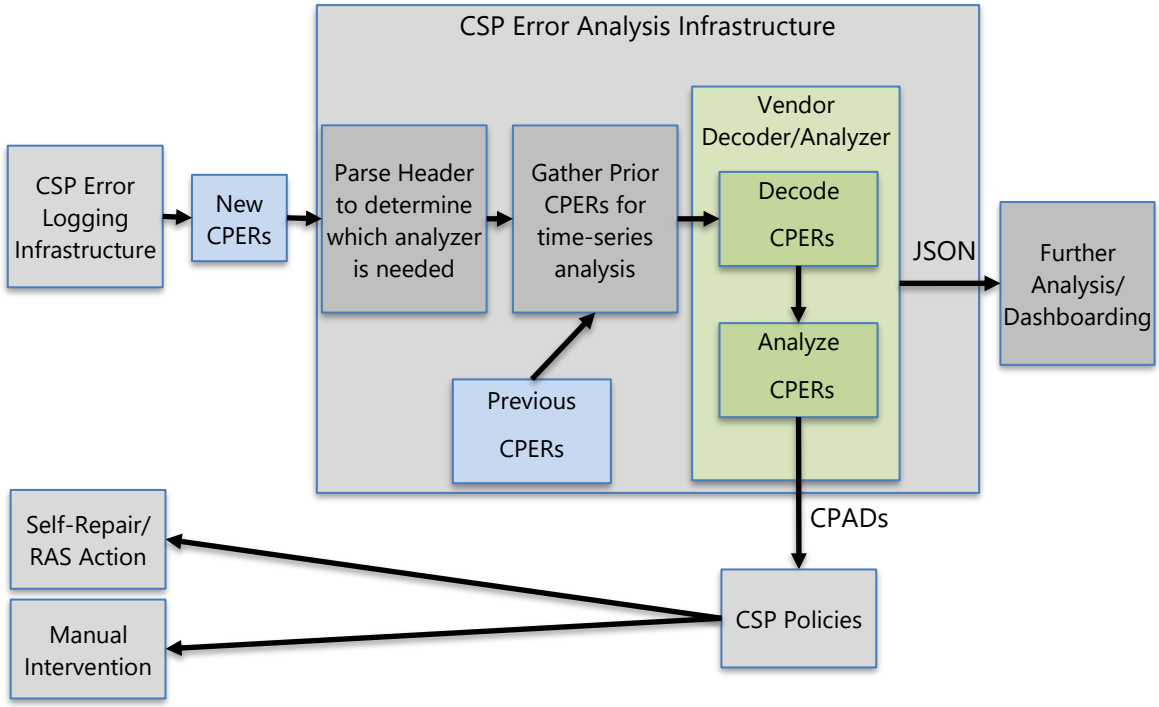


Figure 6: Error Analysis Process

The requirements for supplier-provided analysis tools shall build on the OCP Fault Management Infrastructure Requirements[6], which has been submitted by the OCP Hardware Fault Management project. There are three categories of analysis tool requirements to ensure interoperability within hyperscaler infrastructure:

1. Analyzer code requirements
2. Analyzer input format requirements
3. Analyzer output format requirements

7.1 Analyzer Code Requirements

1. Analysis tool shall be self-contained, with all dependencies included in the package delivered.
2. Analysis tool implementation shall be compatible with Cloud deployment practices such as packaging and running within a container.

3. The supplier analysis tool shall provide an API interface to enable the hyperscaler infrastructure to discover its capabilities:
    a. The analyzer must be able to describe the types of CPER records it can analyze so the end user can send the appropriate CPERs to it; supplier “A” CPERs should be sent to supplier “A” analyzer.
    b. All severities of errors must be logged in CPER format
    c. Whether time-series data is needed by the analysis tool to properly identify the cause of the error
    d. Any additional parameters that are required to enable hyperscaler infrastructure to automate feeding CPER records to the analyzer tool.
    e. This API will be standardized in future releases of the OCP RAS API specification.
    f. The analyzers must be able to analyze all severities of errors including crash dumps.
    g. CPERs must be collected by the same interface to the same pipeline so that we can correlate errors of all severities in our analysis tools.

7.2 Input Format Requirements

Hyperscalers are targeting the use of the CPER format for the supplier analysis tool, as defined in previous sections of this document.

7.3 Output Format Requirements

Analyzer outputs consist of two structures:

1. Mandatory: Decoded CPER data in JSON format
    a. Used for looking at errors, discovering new signals
2. Conditional: A Common Platform Action Descriptor (CPAD) to drive automated responses
    a. Contains information required to drive a self-repair action.
    b. May contain information needed to physically repair a machine.

The CPAD output must be generated by Analyzers whenever required by the corresponding automated response. The CPAD format is currently under definition in the OCP RAS API workstream[11]. Future versions of this document will leverage this definition once finalized.

The following requirements apply to the decoded CPER data in JSON format:

1. Output should be in JSON format that is both machine-parseable and human-readable.
2. Output shall include a section containing decoded data from the CPER records, with raw register-level values as a minimum, and prefer providing a register with additional field-level decoding.
    a. Register-level decoding example: `"aer_uncor_status" : "0x00001000"`
    b. Register with field-level decoding added:

```
"aer_uncor_status" : {
     "dlp_err" : "0",
     "surpdn_err" : "0",
     "poison_tlp_err" : "1",
       …
   }
```

3. Output shall include a section containing error data analysis results, including:
    a. Metadata decoded from the CPER record (e.g., CPER version, FW version, timestamp, device type identifier, etc.) and metadata from the analyzer tool (e.g. tool version, tool-related timestamps, etc.).
    b. A failure identifier that Hyperscalers can refer to when talking to suppliers.
    c. The identifier of the IP block that logged the error.
    d. Suggested actions to take (e.g., replace part, update FW, invoke RAS feature, contact supplier), along with pertinent information to complete the action (e.g. FRU to replace, FW version to update to, RAS feature ID to invoke).
    e. Urgency of suggested action (e.g., as soon as possible, as convenient, before the next reboot, etc.).
    f. Confidence level of the analysis result.
        i. Used to guide hyperscaler policies.
        ii. Desire that it is consistent with other analysis results from same analyzer.
4. Fields in the output report intended for machine-readability must serve a single purpose and not require regular expression parsing to pull multiple pieces of information out of a JSON value.
5. The analysis tool shall have the capability to identify design defects known by the supplier and report them accordingly.
6. The decoded JSON may contain proprietary details depending on the hyperscaler-supplier business relationship:

a. A hyperscaler that has a trusted business relationship with the supplier may see more detail in the decoded output.
b. A hyperscaler that does not have a relationship with the supplier may see much less detail.

To facilitate standardization of these requirements, this document recommends following the JSON schema defined by the OCP HW Fault Management sub-project.

8 Compliance tools

The OCP Test & Validation Initiative is a collaboration between datacenter Hyperscalers having the goal of standardizing aspects of the hardware validation/diagnosis space, along with providing necessary tooling to enable both diagnostic developers and executors to leverage these interfaces. Based on this, a tool called CTAM (Compliance Tool for Accelerator Management) has been developed to work with this and other related specifications and provides compliance testing for Accelerator Management in cloud data centers.

The tool will check compliance of accelerators to as many parts of this RAS specification as possible in an automated manner. The idea being that Hyperscalers and suppliers of accelerators can measure the level of compliance of their hardware with their stacks. The compliance tool will focus on using Redfish to validate the following areas of this specification. The scope will increase in future specifications.

1. Redfish based discovery of supported RAS features, categorized as PCIe, Memory & GPU.
2. Discovery of available error reports / debug dump mechanisms (see 13.3, Appendix C – Debug Dump Support in Redfish). At least one debug dump mechanism should be discoverable.
3. For the discovered error reports / debug dump mechanisms, perform the debug dump actions successfully. Refer to 13.4, Appendix D – Debug Data Taxonomy for examples.
4. Verify the mandated formats, such as CPER, for the error reports / debug dumps / LogService entries.
5. Discovery of error injection mechanisms (see section 13.2, Appendix B – Error Injection Support in Redfish). At least one error injection mechanism should be discoverable.
6. For the discovered error injection mechanisms, perform the injection followed by validation of action and response (see section 5, Error Injection).

9 Confidential Compute and RAS Considerations

This section provides initial guidelines for applying the RAS requirements detailed in earlier sections of this specification when Confidential Compute is enabled. Additional requirements will be included in future revisions of the document.

- Error reporting (Section 4) should not be impacted when Confidential Compute is enabled.
- Error injection (Section 5) should not be impacted when Confidential Compute is enabled under the appropriate test conditions.

10 OCP GPU RAS document - Future Revision Plan

The following list summarizes the topics planned to be included in future versions of this specification:

- Screening and diagnostic tools requirements
- Blast radius reduction based on cluster efficiency scaling
- Confidential Compute and its applicability to each section of the spec
- RAS Compliance Tool Requirement Improvements
- CPAD – Requirements update
- RAS API – convergence ideas
- On-demand Crash Dump API
- Assess if this spec should define a Redfish "Platform Error" message registry to define a set of messages to cover common hardware error cases. This is intended to provide human-readable text along with machine-parseable logs.

11 References

1. PCI Express Base Specification, Revision 4.0, September 27, 2017.
2. Unified Extensible Firmware Interface Specification, version 2.9,
3. OCP GPU & Accelerator Management
4. FW Update Specification.
5. CPER record reporting through Redfish are referenced here: https://www.dmtf.org/sites/default/files/Redfish_LogEntry_for_CPER_WIP.pdf.

6. OCP Fault Management Infrastructure Requirements Ver 1.0
7. OCP HW Fault Management sub-project
8. JEDEC JESD238A: High Bandwidth Memory (HBM) Specification
9. https://www.opencompute.org/documents/external-ver-0-3open-compute-specification-server-component-resilience-workstream-pdf
10. OCP Fleet Memory Fault Management Requirements Version 0.1
11. OCP RAS API Team
12. Corrigendum to A Systematic Study of DDR4 DRAM Faults in the Field." *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*
13. To Detect or to Correct?" *24th International Symposium on On-Line Testing and Robust System Design*
14. OCP GPU & Accelerator Management Interfaces Version 1.0
15. Specification: Firmware Update Requirements for GPU Revision 1.0
16. Common Platform Action Descriptor (CPAD)

12 OCP Tenets

Openness

This specification has been developed using close and open collaboration between industry partners and competitors. Interface and Specification will be open to all OCP members.

Efficiency

The Goal of this specification is to enable seamless integration of GPUs into Hyperscaler solutions. Companion compliance tooling will enable high quality products.

Impact

This is the first Industry initiative to standardize GPU RAS requirements. It is expected to have a significant impact on quality and TTM for GPU Systems by Hyperscalers. Furthermore, it is expected to be applicable to Enterprise deployments as well.

Scale

This specification will apply to very large-scale GPU system deployments in Hyperscale Data Centers.

Sustainability

Reduce the component replacements rates through enhanced Availability and Fault Code improvements.

13 Appendices

The appendices contain useful background details and will be revised as part of the next version of this specification.

13.1 Appendix A - Implementation Considerations

13.1.1 Rationale for hardware error format standardization

There are two existing Industry Standards relevant to hardware error reporting in this document's scope. The first one is the IPMI-based System Event Logs (SEL). However, IPMI SEL has some limitations including reporting very primitive data for hardware errors, lack of support for new technologies such as CXL, and limited industry maintenance. The alternative is Redfish, which better caters to modern datacenter environments. While Redfish does not directly define hardware error formats, it provides an infrastructure upon which retrieval of hardware error records can be standardized.

13.1.2 Using CPER as hardware error format

The second Industry Standard relevant to hardware error reporting is the Common Platform Error Record (CPER) defined in the Unified Extensible Firmware Interface (UEFI) specification. UEFI firmware implementations use CPER records to share hardware error information with the OS. The current industry direction is shifting towards sending CPERs records OOB through the BMC, which is one of the most widely used mechanisms to harvest error data reliably at scale in both bare-metal and non-bare-metal use cases. The details in CPER error records can be opaque to error logging infrastructure but must support analysis by supplier-supplied tools that can look at error trends and suggest RAS actions and FRU replacements to address failures.

The following table shows examples of how CPER format can be used to report hardware errors in this framework.

Table 4: Example CPER Error Record Types

Error Type	Reference of Error Record Format	Comments
Memory Error	Section N.2.5 in UEFI spec Ver 2.9	9.2.2
PCIe Errors	Section N.2.7 in UEFI spec Ver 2.9	9.2.4

CXL Protocol Errors	Section N.2.13 in UEFI spec Ver 2.9	9.2.6
CXL Component Errors	Section N.2.14 in UEFI spec Ver 2.9	9.2.8
Platform Errors and other errors	Section N.2.3 in UEFI spec Ver 2.9	9.2.10 Non-standard Section Body

In addition to standard CPER sections defined in the UEFI spec, suppliers can also use non-standard sections to provide additional information to enhance the Error records.

The following provides an example of PCIe error data reported in the CPER PCIe error section:

```
PCIe Advanced Error Reporting
                    Corrected
------------------------------------------------------------
  0 - PCIe Error Section (Primary)
    Port Type : [4] Root Port
    Version   : 1.1
   Cmd/Status : 0x0010/0x0547
   Device ID  :
        VenId:DevId : 0x8086:0x352a
          Class Code: 0x030400
     Function Number: 0x00
       Device Number: 0x01
             Segment: 0x00
  Primary Bus Number: 0x15
Secondary Bus Number: 0x15
           Reserved1: 0x0
         Slot Number: 0x0000
           Reserved2: 0x00
Device Ser # : 0x0000000000000000
Bridge Ctl/Sts: not supplied
Exp Capability: 0x0142
Dev Caps      : 0x00008022
Dev Control   : 0x2127
Dev Status    : 0x0001
Link Caps     : 0x017a4105
Link Control  : 0x0040
Link Status   : 0xf101
Slot Caps     : 0x00080000
Slot Control  : 0x03c0
Slot Status   : 0x0040
Root Caps     : 0x0001
Root Control  : 0x0008
Root Status   : 0x00000000
Dev Caps2     : 0x007317f7
Dev Control2  : 0x0049
Dev Status2   : 0x0000
Uncorrectable Error Status   : 0x00000000
Uncorrectable Error Mask     : 0x00000000
Uncorrectable Error Severity : 0x00062010
```

```
Correctable Error Status     : 0x00000001
    Receiver Error
Correctable Error Mask       : 0x00000000
Caps & Control               : 0x000010a0
Header Log                   : 0x4a000001 0x16000004 0xfd000000
0x00000000
Root Error Command           : 0x00000007
Root Error Status            : 0x00000001
Correctable Err Source ID    : 0x15, 0x01, 0x00
Uncorrectable Err Source ID  : 0x00, 0x00, 0x00
```

13.1.3 Reporting CPER records via Redfish

The following provides a LogEntry object example that reports a small CPER record as inline diagnostic data:

```
{
"@odata.type": "#LogEntry.v1_14_0.LogEntry",
"Id": "3",
"Name": "CPER Log Entry with small diagnostic data",
"EntryType": "Event",
"Severity": "Critical",
"Created": "2022-03-07T14:45:00Z",
"Message": "A platform error has occurred.",
"MessageId": "Platform.1.0.PlatformError",
"Links": {
"OriginOfCondition": {
"@odata.id": "/redfish/v1/Systems/1"
}
},
"CPER": {
"NotificationType": "902834BC-AD67-0BAD-BEEF-123456789012"
},
"DiagnosticDataType": "CPER",
"DiagnosticData":
VGhlIGNha2UgaXMgYSBsaWUhCg==ASDEWIhnqn55Qe924MFAFHDFOIAFHEDANHV4582bAIYQN"
,
"@odata.id": "/redfish/v1/Systems/1/LogServices/Log1/Entries/4",
}
```

Large CPER files shall use the "AdditionalDataURI" property as proposed in

https://www.dmtf.org/sites/default/files/Redfish_LogEntry_for_CPER_WIP.pdf

13.2 Appendix B – Error Injection Support in Redfish

This section provides a reference implementation in Redfish. The following diagram provides a high-level flow for Redfish-based error injection. This appendix contains useful background details and will be revised as part of the next version of this specification

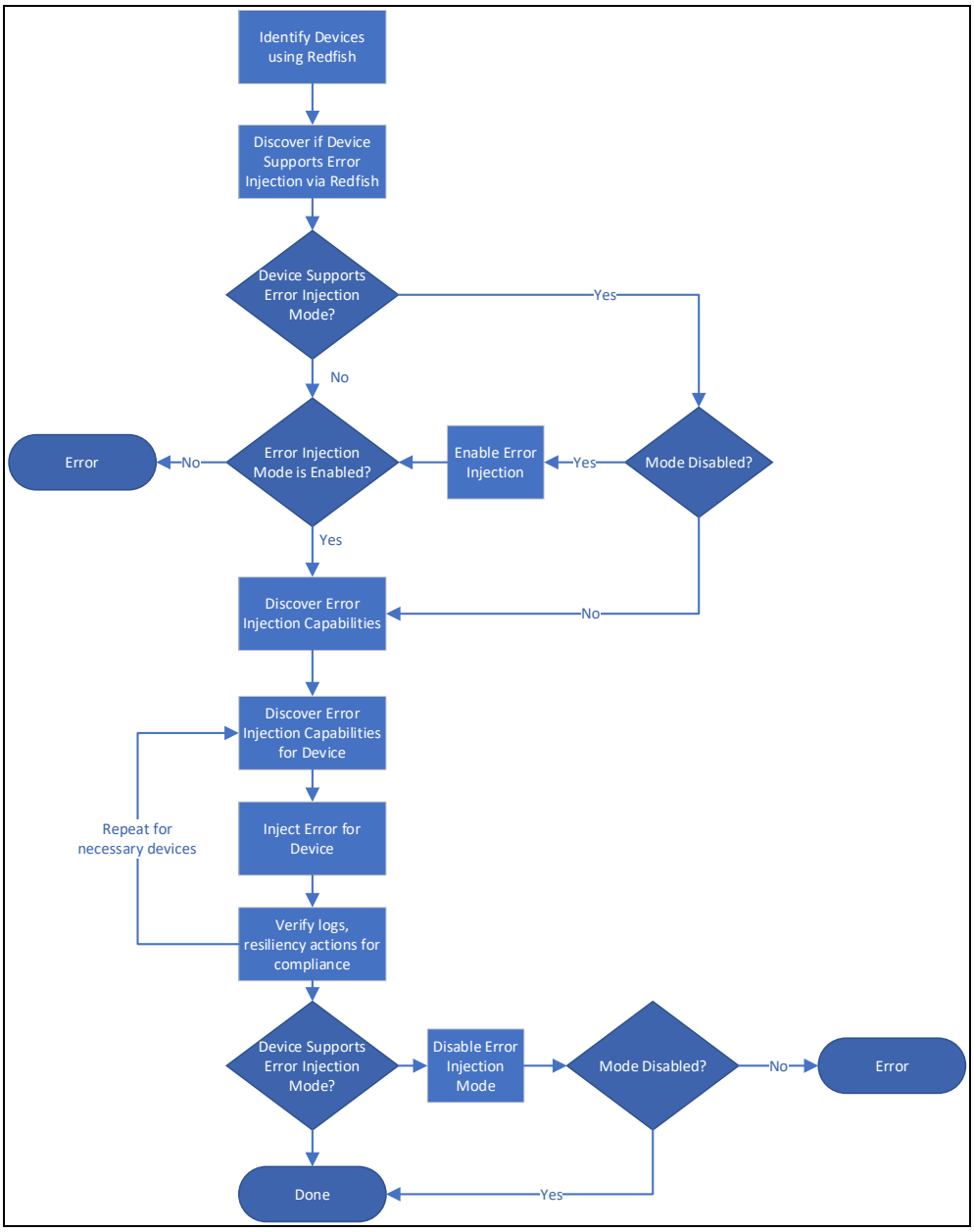


Figure 7: High-level flow for Redfish-based Error Injection

1. Device Identification – Using Redish Device Tree

2. Discover error injection capabilities, including methods supported to enable error injection mode. The discovery mechanism shall be covered in future versions of this specification.
3. Enable error injection mode. Refer to section 4.9 for more details.
4. Perform error injection to a particular Device, which may require execution outside of Redfish
5. Compliance - Verify the error logs and error resiliency actions. This may require executing a workload to trigger the error injected.
6. Disable error injection mode, if desired.

An OCP standardized API should be developed to support the following capabilities:

1. Query ErrInj State: Returns the current state of the Error Injection mode:
    a. Disabled
    b. Enabled
    c. Pending to enable – some action is required before enabling. For example: reset, authentication, unlock, etc.
    d. Pending to disable – some action is required before disabling. For example: FW reflash.
2. Enable ErrInj: action to request enabling/disabling the Error Injection feature
    a. True
    b. False
3. Query Supported ErrTypes: Returns a list of supported error types that can be injected. Please refer to section 5.3 for error coverage requirements.

These APIs may be defined as OEM extensions under “OCP" to keep them supplier neutral. Redfish schemas to be defined in future versions of this document.
For instance, Redfish Clients can traverse to the GPU resource and do a “GET”. The response includes error injection action details.

PCIe Errors Attributes

Attributes	Details	Comments
Device Identification	Uses Redfish based URL	Specifies the Device targeted for error injection
Error Severity	PCIe Correctable PCIe Non-Fatal and PCIe Fatal	All PCIe error severities shall be supported
Error Type	Correctable e.g., Bad TLP, Bad DLLP, Receiver Error, Reply Timeout Ref [1]	Multiple error types for each applicable error severity need to be supported

Non-Fatal e.g., Poisoned TLP received, Completion Timeout, Unexpected Completion Ref [1]

Fatal e.g., Malformed TLP, Flow control Protocol Error, Training Error, Receiver Overflow. Ref [1]

Memory Errors Attributes

Attributes	**Details**	**Comments**
Physical Device Identification	Redfish-based URI	Specifies the Device targeted for error injection
Sub-Device Identification	Rank, Column, Row Level	Specifies the sub-device level targeted for error injection
Error Severity	Correctable Uncorrectable	All error severities shall be supported
Address	Memory Physical Address targeted for error injection	Optional if using Sub-Device Identification attribute

Note: Memory Error Poison and other errors are planned to be defined in the next revision of this document

13.2.1 Device Identification using Redfish device tree

The following figure provides an example of a Redfish device tree for the UBB component:

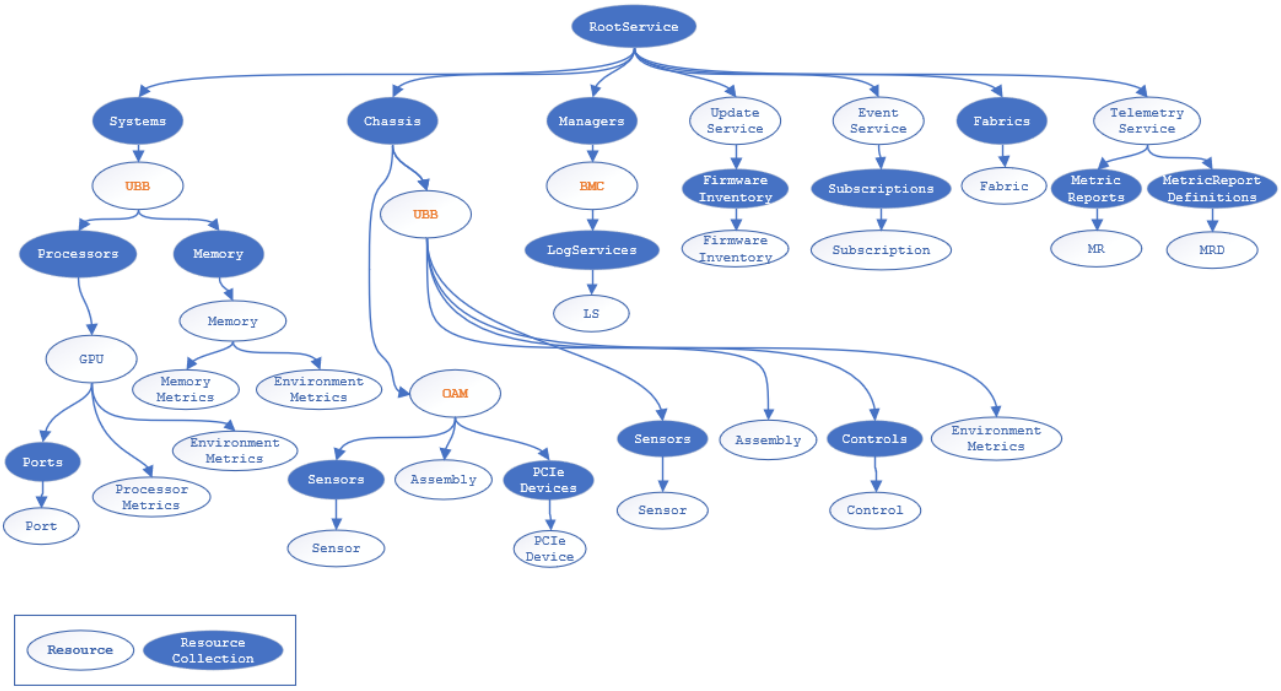


Figure 8: Redfish Device Tree for the UBB Component

Please refer to the [OCP GPU & Accelerator Management] for the latest on this.

13.2.2 Error Injection Schema Overview

This section is work-in-progress and does not constitute normative requirements.

The redfish resource proposal aims to introduce a new schema that could be utilized as an attribute to both physical and logical components and allows the resource to expand in its capabilities using a standard resource, including allowing OEM extensions as necessary.

ErrorInjection Resource collection allows multiple ErrorInjection endpoints and to ensure that the ErrorInjection endpoints are loosely coupled to the resource it could be added as "Links" in parent resources (Does not require to add new properties for each Schema). The initial intent is to limit the standardization to the ErrorInjection Schema properties with an expectation that the OCP team will continue to standardize the enumerations values for each of these properties

Actions (POST)

- "StartErrorInjection":
- "ComponentGroup": string representing ComponentGroups (Enums can be standardized later)
- "ComponentType" : string representing ComponentTypes (Enums can be standardized later)
- "ErrorType" : string representing ErrorTypes (Enums can be standardized later)
- "Severity" : string from Enum of Error Injection Severity
- "ErrorAttributes" : array, this can be array of objects or strings

ErrorInjection (GET)

- Error Injection Status and Current Error Injection can be presented

13.3 Appendix C – Debug Dump Support in Redfish

System debug dump capabilities will vary based on the system design based on the GPU UBB and Discrete accelerator device. Need to support the Redfish based API to list the system debug dump capabilities. System debug dump capability API must return the following information.

1. For each entry, a URL with details about information and tools to parse the dump.
2. For each entry, a list of triggers, i.e. if the dump collection is triggered automatically (on error) or needs to be requested explicitly by the user.
3. Estimated maximum size details will be a part of the next revision of the spec.
4. Estimated max time taken for the dump to be generated will be part of the next revision.
5. Redfish-Publications/csdl/LogService_v1.xml at ce5f0dc950120af8856b7aa10ae3c29fcae89c4f · DMTF/Redfish-Publications · GitHub defines estimated sizes and times for dumps.

Application codes traverse all `LogService` URIs defined in the `LogService` schema (https://redfish.dmtf.org/schemas/v1/LogService_v1.xml).. Then, do a GET on the "`@Redfish.ActionInfo`" value in the `CollectDiagnosticData` action object.

This is an example of the flow for discovering `LogService` subordinate to a `Manager` resource:

1. In a service root object `/redfish/v1`, look for a property called "`Managers`"
    a. Find the "`@odata.id`" value. This is the URI of the Manager collection.
    b. Do a `GET` on that URI.
2. Look at the "`Links`" object in the "`Members`" array.
    a. Find the "`@odata.id`" of the Manager in question (this may require GETs on each manager and looking at properties in the system to determine the right manager).
    b. Do a `GET` on that URI.
3. Look for an object called "`LogServices`".
    a. Find the "`@odata.id`" value.
    b. Do a `GET` on that URI.
4. Look at the "`Links`" object in the "`Members`" array.

   i. Find the "`@odata.id`" of the "`LogServices`" in question (this may require GETs on each LogService and looking at properties in the LogService to determine the right log service).
   ii. Do a GET on that URI.

5. Look in the “Actions” object for an object called "#LogService.CollectDiagnosticData"
   a. Find the "@Redfish.ActionInfo" value.
   b. Do a GET on that URI – this provides the allowable values of the CollectDiagnosticData action.

Example response of a LogService resource:


```
{
    "@odata.id": "/redfish/v1/Managers/1/LogServices/Log1",
    "@odata.type": "#LogService.v1_7_0.LogService",
    "Id": "Log1",
    "Name": "Manager Log Service",
    "MaxNumberOfRecords": 1000,
    "OverWritePolicy": "WrapsWhenFull",
    "DateTime": "2015-03-13T04:14:33+06:00",
    "DateTimeLocalOffset": "+06:00",
    "ServiceEnabled": true,
    "LogEntryType": "Event",
    "Status": {
        "State": "Enabled",
        "Health": "OK"
    },
    "Actions": {
        "#LogService.CollectDiagnosticData": {
            "target":
"/redfish/v1/Managers/1/LogServices/Log1/Actions/LogService.CollectDiagnos
ticData",
            "@Redfish.ActionInfo":
"/redfish/v1/Managers/1/LogServices/Log1/CollectDiagnosticDataActionInfo"
        }
    },
    "Entries": {
        "@odata.id": "/redfish/v1/Managers/1/LogServices/Log1/Entries"
```

```
        }
        "DiagnosticDataDetails": [{
              "DiagnosticDataType": "Manager",
              "EstimatedDuration": "PT1H",
              "EstimatedSizeBytes": 256000
         },{
              "DiagnosticDataType": "OEM",
              "OEMDiagnosticDataType": "<OEM Diagnostic Data Type>",
              "EstimatedDuration": "PT1H",
              "EstimatedSizeBytes": 256000
         }]
    }
```

Example response from a Manager CollectDiagnosticDataActionInfo resource:

```
    {
       "@odata.id":
    "/redfish/v1/Managers/{ManagerId}/LogServices/{LogServiceId}/CollectDiagno
    sticDataActionInfo,
       "@odata.type": "#ActionInfo.v1_0_0.ActionInfo",
       "Id": "CollectDiagnosticDataActionInfo",
       "Name": "Collect Diagnostic Data Action Info",
       "Parameters": [{
          "Name": "DiagnosticDataType",
          "Required": true,
          "DataType": "String",
          "AllowableValues": ["Manager"]
       }]
    }
```

Example response from a System CollectDiagnosticDataActionInfo resource:

```
/redfish/v1/Systems/{ComputerSystemId}/LogServices/{LogServiceId}/CollectDiagnos
ticDataActionInfo

{
   "@odata.id":
"/redfish/v1/Systems/{ComputerSystemId}/LogServices/{LogServiceId}/CollectDiagno
sticDataActionInfo

",
   "@odata.type": "#ActionInfo.v1_0_0.ActionInfo",
   "Id": "CollectDiagnosticDataActionInfo",
   "Name": "Collect Diagnostic Data Action Info",
   "Parameters": [{
```

```
      "Name": "DiagnosticDataType",
      "Required": true,
      "DataType": "String",
      "AllowableValues": ["OEM"]
   },{
      "Name": "OEMDiagnosticDataType",
      "Required": true,
      "DataType": "String",
      "AllowableValues": ["<OemDiagnosticType_1>", "<OemDiagnosticType_2>",
...],
      "AllowableValueDescriptions": ["Tarball of OemDiagnosticType_1 logs",
"Tarball of OemDiagnosticType_2 logs", ...],
   }]
}
```

The following table shows an example curl commands to trigger a crashdump request:

Dump type	Triggers	Redfish API
PCIe Retimer	automatically on-error (default) Manual on demand	`curl -X POST http://${ManagerIP}/redfish/v1/Systems/{ComputerSystemId}/LogServices/{LogServiceId} /Actions/LogService.CollectDiagnosticData/ -d '{"DiagnosticDataType":"OEM", "OEMDiagnosticDataType":"DiagnosticType=4ba4100c-e35f-4cb5-a6cd-717f09a0df1d"}'`

Note that the GUID in DiagnosticType determines the type of dump as defined in **Error! Reference source not found.**

13.4 Appendix D – Debug Data Taxonomy

The following table describes a sample view of the taxonomy of debug data sensitivity and usefulness for root-cause analysis. **This content is not final and is not intended to be normative**.

Table 5: Debug Data Taxonomy

Debug Data Collected	Data Sensitivity			Data Usefulness for RCA	Examples/Recommendations
	Supplier	CSP	End-user		
Error event data from standard logs implemented	Low	None	None	Lowest: basic information about the failure, helpful for triage and HW fault	IPMI SEL log, Redfish Alerts, Linux dmesg logs and Windows event logs.

by the FW, OS or management infrastructure				management, but not sufficient for RCA	
Error logs with non-micro-architectural supplier data	Med	None	None	Low: additional information about the failure, helpful for HW fault management and pointing to known fixes without RCA	Crashdumps, FW-collected logs
Error logs with micro-architectural supplier data	High	Low	None	Medium: helpful for RCA of certain failure modes not covered by previous levels	Enriched Crashdumps and FW-collected logs, custom register dumps, CPER logs defined by OCP HWFM sub-project, trace logs of control fabric of individual Ips. **Recommendation:** OCP HW FW management work group recommendation is to go default support with Level3 to help with node serviceability and fleet actions.
Supplier debug logs with CSP-only SW context	High	High	None	High: helpful for failure modes that require Kernel/application SW context for RCA or failure correlation/reproduction	HW/FW/SW Trace logs, compute-IP register dumps and system memory dumps while executing CSP-internal workloads **Recommendation:** Level4 is required to debug HW and HW/SW combination of issues. Until Level4 logs should not

					have any end User customer data.
Supplier debug logs with end-user workload characteristics and no end-user data	High	High?	Low?	High: helpful to debug failure modes that only reproduce with end-user specific workloads	Trace logs of perf monitors & counters, opcodes, branches and/or micro-arch specific events in compute Ips. **Recommendation:** Level5 is recommended when issue RCA requires understanding the workloads and should not contain any end-user data.
Full data capture with supplier, CSP and end-user data	Highest			Highest: best debug visibility. Helpful for RTL-level RCA and failure reproduction in supplier's debug environment	Full SoC scandumps including data flops, full memory array dumps, trace logs of all compute IP traffic **Recommendation:** Level6 is Not a recommended option on production nodes where end user workloads running. Useful for lab and early cluster level issue RCA.

The taxonomy consists of six levels, with each level adding more useful data for RCA, but also increasing the risk of data sensitivity. This specification does not dictate how these requirements may be implemented by suppliers but encourages them to develop solutions that fit into one of these levels.

13.4.1 Level 1:

This level collects error event data, typically from standard logs provided by the FW, OS, or management infrastructure. These do not contain sensitive data from the CPU supplier, CSP provider, or end-user. They

provide basic information about the failure, which may be useful for triage and HW fault management, but not for root-cause. Examples include IPMI SEL log, Redfish Alerts, Linux dmesg logs and Windows event logs.

13.4.2 Level 2:

This level collects data enriched with supplier-specific information, usually defined by the CPU supplier. This level may or may not contain supplier data bound by Non-Disclosure Agreements (NDA). However, it does not contain sensitive data from CSP providers or end-users. This level of data provides more details about the failure, which may be useful for HW fault management and pointing to a known fix but may not be sufficient for root-cause in certain failure modes. Examples include crashdumps and FW-collected logs.

13.4.3 Level 3:

This level collects Si (GPU, CPU, PCIe Switch) supplier micro-architectural sensitive data that is usually bound by Non-Disclosure Agreements (NDA). However, this level does not contain sensitive data from CSP providers or end-users. This level of data is useful for HW fault management and for root-cause of certain failure modes not covered in the previous levels. Examples include enhanced crashdumps, enhanced FW-collected logs, CPER logs defined by OCP HWFM sub-project, custom register dumps, and trace logs of control fabric of individual IPs.

13.4.4 Level 4:

This level contains both supplier micro-architectural sensitive data and CSP provider sensitive data, such as OS context or custom kernel data. Resorting to this level of data is typically beyond the scope of HW fault management and gets more into the domain of debug and root-cause analysis. Enabling this level of data collection may require supplier authorization. CSP release of this data for supplier debug may require a data audit process and/or additional Non-Disclosure Agreements. Examples include trace logs of SW, FW and/or HW stack, memory array dumps, and custom dumps that includes data registers of compute IPs while executing CSP internal workloads.

13.4.5 Level 5:

The level of data collected may be sufficient to infer or characterize the end-user workload, which poses a risk to end-user confidentiality. Enabling this level of data collection may require supplier authorization. CSP release of this data for supplier debug may require additional agreements, data audits, and implicit/explicit permission from the end-user. Resorting to this level usually happens when the failures only reproduce with the end-user workloads in high-enough failure rates that violate CSP Service Level Agreements (SLA) and/or service interruption rate targets. Examples include trace logs of perf monitors & counters, opcodes, branches and/or micro-arch specific events in compute IPs, etc.

13.4.6 Level 6:

This level is similar to Level 5, but may also include end-user data, such as Personally Identifiable Information (PII). This level poses the highest risk to end-user confidentiality and data privacy. Enabling this level of data collection may require supplier authorization. CPS release of this data for supplier debug may take longer due to additional agreements, data audits, and explicit permission from the end-users as well as meeting legal and regulatory requirements. Resorting to this level usually happens when the failures only reproduce with the end-user workloads in high-enough failure rates that violate CSP Service Level Agreements (SLA) and/or service interruption rate targets. Examples include full SoC Scandumps including data flops, full memory array dumps, trace logs of all compute IP traffic, etc.

13.5 Appendix E – RAS Feature Requirements Table

The table below lists the RAS feature requirements as described in Section 3. RAS Feature Requirements

Section	Theme	RAS Feature
Error! Reference source not found.	Metrics and Target Definitions	

3.2.1	System level RAS Requirements	Error Containment
3.2.2	System level RAS Requirements	Uncorrected Error Handling
3.2.2.1	System level RAS Requirements	Resumption with Spare Resources
3.2.2.2	System level RAS Requirements	Resumption with Degraded Performance
3.2.3.3	System level RAS Requirements	Graceful Handling of Memory Failures
3.2.3	System level RAS Requirements	Correctable Error storm handling
3.2.4	PCIe Subsystem Fault Handling	
3.2.4.1	PCIe Sub-system Fault Handling	PCIe Advanced Error Reporting (AER)
3.2.4.2	PCIe Sub-system Fault Handling	Outbound Completions with UR/CA Error
3.2.4.3	PCIe Sub-system Fault Handling	Inbound Completions with UR/CA Error
3.2.4.4	PCIe Sub-system Fault Handling	Handling Completion Timeout (CTO) Errors
3.2.4.5	PCIe Sub-system Fault Handling	PCIe link Bandwidth Change Reporting
3.2.4.6	PCIe Sub-system Fault Handling	PCIe Downstream Port Containment (DPC)
3.2.4.7	PCIe Sub-system Fault Handling	End-to-end Cyclic Redundancy Check (ECRC)
3.2.5	Other System-level RAS Features	
3.2.5.1	Other System-level RAS Features	PCIe Retimer Error Handling
3.2.5.2	Other System-level RAS Features	XLink and Switch Error Configuration and Handling
3.3	Memory Fault Handling	
3.3.1	Memory Fault Handling	Memory Error Correction Capabilities
3.3.2	Memory Fault Handling	Memory Sparing Repair Capabilities
3.3.3	Memory Fault Handling	Memory Scrubbing
3.3.4	Memory Fault Handling	Protection from corruption on the memory bus
3.3.5	Memory Fault Handling	Data Link Parity Protection
3.3.6	Memory Fault Handling	Containment (e.g. Data Poisoning)
3.3.7	Memory Fault Handling	Memory Error Reporting
3.4	GPU Internal RAS Features	
3.4.1	GPU Internal RAS Features	Internal Error Coverage
3.4.2	GPU Internal RAS Features	Other Radiation Tolerance Features
3.4.3	GPU Internal RAS Features	Other Influences on Failure Rate

4	Error Reporting	
4.1	Error Reporting	Error Reporting Requirements
4.2	Error Reporting	Error Record Requirements
4.3	Error Reporting	In-band vs Out-of-Band Error Logging
5	Error Injection	
5.1	Error Injection	Overview
5.2	Error Injection	Requirements for Testing Error Handling
5.3	Error Injection	Error Injection Coverage
5.4	Error Injection	Error Injection Security Requirements
6	Crash and Device Debug Dumps	
6.1	Crash and Device Debug Dumps	Introduction
6.2	Crash and Device Debug Dumps	Crash Dumps
6.3	Crash and Device Debug Dumps	Debug Dumps
6.4	Crash and Device Debug Dumps	Debug Dump Sensitivity
6.5	Crash and Device Debug Dumps	Dump Reliability

13.6 Appendix F - Glossary of Terms

Term	Definition
Accelerator Core	The element within the GPU/accelerator that executes workloads and performs operations.
BMC-based error logging	Error logging using firmware running on the BMC, and not involving the operating system or platform firmware (may use management core firmware)
Challenge	A security mechanism in which a system issues a prompt to a user or device to verify identity, typically as part of authentication (e.g., one-time code).
Containment	Error containment prevents corrupted data from being used by applications on the system and prevents corrupted data from being transmitted over I/O (e.g., stored to disk or sent over a network).

	In systems that had a bus-based architecture, resets could implement error containment since the reset was sent to all the devices on the bus at once. Modern systems do not use buses, and the lack of a reset wire means there it takes a potentially long time for resets to propagate. So Modern systems transmit containment signals via their interconnect fabric (e.g., UPI or HyperTransport). See poison and viral, for example containment signals. Stopping an OS with an interrupt is not sufficient for containment. DMA may continue after the OS is interrupted and can send corrupted data to I/O.
CPER	Common Platform Error Record, as defined in UEFI specification.
CSP	Cloud Service Provider - The term Cloud Service Provider is intended to be a broad term for any organization that runs a large fleet of machines and provides services using that fleet. The use cases in this document are broad and are not strictly limited to organizations that provide cloud services. This proposal also supports use cases that include server operators that are not cloud providers.
DUE	Detected and Uncorrected Error. This definition is in the context of hardware detected error. These are also sometimes called Uncorrected Errors (UCEs).
Error Logging	Error logging is used very broadly to refer to collecting error data about a failure. Error logging is the process of latching information about an error in the chipset, collecting the error data in software and sending the data to tools where it can be analyzed. When software reads the error logs, it often generates a binary structure such as a Common Platform Error Record (CPER) log that can be passed to other, higher-level software for analysis.
Error Handling	Error handling is where software logs errors, analyzes them, and determines how to minimize their impact. Error handling software can take actions like driving RAS features, telling the operating system to stop using resources, and sending messages to get the server repaired.
Fault	A fault is something that causes hardware to malfunction. If the hardware detects the fault, it can log errors that aid in diagnosing the fault and servicing the hardware.

FRU	Field Replaceable Unit - an FRU is a part that can be changed in the data center to repair a broken system. Components like DIMMs or PCIe cards are FRUs. A DRAM soldered to a mainboard would not be considered an FRU; in that case, the mainboard would be considered an FRU
GPU	A Graphics Processing Unit. It can be a discrete unit or something like a GPU card over PCIe.
GPU Accelerator	A GPU is not necessarily used not as a Graphics processing unit, but as a processing unit for some unit of work by Hyperscalers.
Hardware component	A component designed by a supplier. Examples of hardware components include CPU chips/SoCs, DRAMs, and PCIe devices.
Hyperscalers	Cloud service providers around the world that emphasize scaling and rapid time-to-market of new accelerator products.
Node	A platform designed by a supplier integrating multiple components that run an operating system. These are sometimes also called compute nodes.
NFF	No Fault Found – occurs when a part is returned to the supplier, and the problem cannot be reproduced on the returned part. NFFs can result from failures that are hard to reproduce, insufficient diagnostic tests and replacing the incorrect part. Effective fault management solutions reduce NFFs.
Platform firmware	Firmware running on mission-mode cores on the processor. On x86 CPUs, platform firmware is sometimes called BIOS or UEFI firmware.
OS	Operating System - that runs on the processor. The operating system may run on a virtual machine on a hypervisor or directly on the machine (bare metal).
RAS Features	RAS features mitigate hardware faults. They can be implemented in hardware, software, or a combination of both.

	Some examples of RAS features include memory ECC, DRAM Post Package Repair, marking pages as bad in the OS, and poison recovery. RAS features like Post Package Repair (PPR) need to be triggered. For example, on a memory controller that supports run-time PPR, the software might see a series of corrected errors, all from the same row address. The software could decide that the PPR would mitigate the fault causing the corrected row errors and could trigger hardware to do the PPR.
Signaling	Error signaling refers to how software is notified that an error has occurred.
SDC	Silent Data Corruption. This is caused by the class of failures that are undetected by the hardware logic and resulted in application/SW performing incorrect computation. This does not include the hardware undetected and application/SW undetected faults.
SLA	Service Level Agreement
SLO	Service Level Objective
TTM	Time to Market. Reliability problems in New Product Introductions can cause delays. Hyperscalers need fault management solutions that enable them to get products to General Availability predictably with the required reliability.
UBB	OCP Universal Baseboard
UEFI	Unified Extensible Firmware Interface
Xlink	Generic term for GPU-to-GPU links.
Xswitch	Generic term for a switch used in GPU-to-GPU links.