



OCP FAULT MANAGEMENT INFRASTRUCTURE REQUIREMENTS

Version 1.5.0

Acknowledgement

This document is a result of collective effort of several OCP participants, who came together in the weekly meetings of the Hardware Fault Management (HWFM) sub-project over a period of last two plus years, to share their expertise in the field of data-center reliability, availability and serviceability (RAS), to distill their decades of collective RAS experience and pain-points into a common set of fault management solution requirements.

Contributors: (Sorted alphabetically by company name then last name)

Name	Title	Company
Hank Bruning	Principal Member of Technical Staff	AMD (was ZT)
Vilas Sridharan	Senior Fellow RAS Architect	AMD
Majed Beigi	Senior Member of Technical Staff	AMD
Sebastien Hily	Distinguished Architect	Ampere Computing
Dean Liberty	Distinguished RAS Architect	Arm
Charlie Slayman	Technical Leader	Cisco
Kevin Cross		Dell
Clint Smullen	Engineer	Google
Jon Pan-Doh	Engineer	Google
Shubhada Pugaonkar	RAS Architect	Intel
Yogesh Varma	RAS Technologist	Formerly Intel
Anil Agrawal	Engineer	Meta
Panos Christeas	Engineer	Crusoe
Sukay Luhadia	RAS Systems Engineer	Meta
Evan Pearson	DRAM Systems Eng Manager	Micron
Tony Geidl	DRAM Systems Eng Director	Micron
Dale Hiscock	DRAM Systems Eng Principal Engineer	Micron



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

Owen Straub	DRAM Systems Eng Engineer	Micron
Drew Walton	Fault Management Lead	Microsoft (formerly Google)
Rama Bhimanadhuni	RAS lead	Microsoft
Carlos Vallin	Principal Member of Technical Staff	Microsoft (formerly AMD)
Aimer Bhat	Sr. Principal Engineer	Oracle
Meeta Saggi	Member of Technical Staff	Pure Storage
Sakul Gupta	Member Of Technical Staff, Firmware Engineering	Micron
Roy Chung	RASM Lab	SK Hynix
Joshua Mora	CEO	SYSTEM-STACK

Version History

Date	Version #	Author	Description
06 Oct 2023	1.0		First revision
15 Aug 2025	1.5	See table above	Added in-band method Updated diagrams



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

Executive Summary

This proposal outlines a framework that addresses challenges that [Server Fleet Operators](#) (SFO, a general term for anyone that operates a server) have had with prior hardware fault management solutions. These prior solutions do not provide the tools SFOs need to diagnose hardware failures and act to minimize their impact. CPUs, chipsets, memory and IO devices are complex with thousands of registers and understanding how to interpret those registers requires specialized knowledge of the design of the silicon; something that the silicon vendor is best placed to provide. Prior approaches to error logging resulted in insufficient data collection, had negative impacts on security and performance, and were inconsistent, making it difficult to collaborate with silicon vendors. With these prior solutions, Reliability, Availability and Serviceability features are difficult to implement and this difficulty blocks SFOs from using the features to minimize the impact of failures. In addition, there was no standard way to report the failure data back to silicon vendors in order to identify and mitigate new or low-[FIT](#)-rate failure modes that are hard to observe in the vendor's testing.

This Open Compute Requirements document describes a framework for solving these challenges. This framework guides other OCP efforts including the RAS API. The silicon vendors provide RAS API compliant analysis tools that interpret errors and suggest actions that could help mitigate the issue. The SFOs have the option to deploy a solution built on this framework using in-band and/or out-of-band methods. Such a solution would facilitate integrating vendor specific tools within the SFO's infrastructure such that vendor specific information can be captured in a standardized manner and analyzed efficiently to diagnose complex problems faster. Such a solution would also facilitate observing and analyzing the error data across a large fleet to find any common failure modes and speed up the remediation/repair. The RAS API makes it easy to invoke many RAS features via a simple RedFish® call using either the IMA (Inband Management Agent) and/or the OMA (Out-of-band Management Agent). By lowering the barrier to integrate RAS features, we believe the industry will move from each company focusing on their favorite RAS features to trying new ones and basing the use of the features on data that demonstrates the value of the feature. The solution outlined in these requirements handles bare-metal, application and infrastructure use cases and benefits any server operator that wants to improve the RAS of their servers.

Earlier versions of this requirement document primarily focused on out-of-band management using the BMC. This version of the requirement document describes both out-of-band management and in-band management methods for implementing RAS features (including error reporting).



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

License

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

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

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

*Other names and brands may be claimed as the property of others

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>



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

Table of Contents

Acknowledgement	2
Version History	3
Executive Summary	4
License	5
About Open Compute Foundation	5
1. Introduction	8
2. Problem Statement	8
2.1. Creation of Component Error Logs	9
2.2. Interpretation of Component Error Logs	9
3. Compliance with Open Compute Project Tenets	9
3.1. Openness	10
3.2. Efficiency	10
3.3. Impact	10
3.4. Scale	10
3.5. Sustainability	10
4. Background	11
4.1. Primary data center usage-models	11
Bare-metal	11
Infrastructure	11
Application	11
4.2. In-band and Out-of-band Methods Overview	12
4.3. Hardware Component Error Classification	14
5. Fault Management System Architecture	16
5.1. In-band Fault Management System Goals	16
5.2. Out-of-band Fault Management System Goals	17
5.3. Integration into the SFO infrastructure	18
5.4. In-Band and Out-of-Band Methods are not Mutually Exclusive	19
5.5. Goals of the OCP Fault Management Infrastructure	19
6. Proposed System-Level Error Handling Capabilities	20
6.1. Corrected Errors	21
6.2. Non-Fatal/Recoverable Errors	21
6.3. Non-fatal/Non-recoverable Errors and Reset feasible	22
6.4. Fatal - System Firmware Can Execute	23
6.5. Fatal - System Firmware Cannot Execute	24
7. Solution Architecture	25
7.1. SFO Protocol	26



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

7.2. Inband Management Agent (IMA)	26
7.3. Inband RAS Plugins (INB)	26
7.4. RAS API	26
7.5. Kernel	26
7.6. CPU + Directly Attached Memory and HSIO (High Speed IO) devices	27
7.7. Firmware	27
7.8. Device Firmware	27
7.9. Device Driver	27
7.10. Proposed fault management using inband methods	28
7.11. An example of the PCIe* error handling flow for OS-first, In-band Methods	28
7.12. Overall view of fault handling combining in-band and out-of-band methods	30
7.13. Error Logging Flow for OS Based Methods	34
7.14. Considerations for supporting both IB and OOB Methods	35
8. Error Handling Flow Examples	35
8.1. Data Poisoning Example	36
8.2. Illustration of Framework's OOB Error Handling (Poison Generation to Fault Analysis)	38
8.3. Illustration of Framework's In-Band Error Handling (Poison Creation to Fault Analysis)	40
8.4. PCIe* corrected error reporting example using In-Band method	42
8.5. PCIe* corrected error reporting using OOB	43
9. Complementary Enablement in the Industry	44
9.1. BMC-Based Fault Management Enablement	44
9.2. Interface between the BMC and the Kernel	44
9.3. Interface between the BMC and Analysis/Decoding Tools	44
9.4. Interface for configuring and triggering RAS features Actions from the BMC	45
9.5. Error format for the logs in OS-first method of error signaling	45
9.6. Standardize minimum error data set in Hardware logs	45
9.7. Additions to Redfish Interfaces (Reference)	45
9.8. Hardware Error Reporting Standardization Requirements	46
9.9. Hardware Error Report Standardization Solutions	46
9.10. Error Injection capabilities	47
10. Conclusion	47
11. References	48
12. Appendix A - Glossary of Terms	48
13. Appendix B - Implementation Considerations	55
13.1. Considerations for preventing race conditions	55
13.2. Cases where error registers are overwritten while being read	55
13.3. Cases where In-Band and Out-Of-Band error logging solutions clear errors while they are still being read	56



1. Introduction

This document defines requirements for a system architecture to handle hardware faults so that desired reliability, availability, and serviceability can be achieved. In case of a large cluster of machines, additionally desired efficiency can be achieved. Handling hardware faults implies reporting errors, RAS-actions, and remediating faulty components by the [Server Fleet Operators' \(SFO\)](#) management infrastructure. Desired availability implies both system reliability and serviceability matrices. Desired efficiency implies maximizing the useful time while an application recovers from a hardware fault.

Hardware fault handling could be implemented using in-band methods or out-of-band methods or a combination of both methods. An earlier version of this document already described out-of-band methods and this version of the doc adds system architecture requirements for in-band methods. For in-band methods, It first provides an overview of existing methods, then clarifies the gaps that exist in Server Fleet Operator infrastructure adoption, and finally recommends addressing these gaps.

The goal of this system architecture is to solve Reliability, Availability, and Serviceability problems for Server Fleet Operators. Throughout this document, 'hardware fault management' means recognizing failure conditions and deciding what to do about them. This document describes system architecture requirements that all cloud infrastructure stakeholders shall adhere to for designing a vendor and silicon agnostic hardware fault management infrastructure. The vendor and silicon interoperability is achieved by proposing amendments to existing industry standard specifications in section [Complimentary Enablement in Industry](#).

2. Problem Statement

The system architecture in this document addresses the following challenges in current data centers:

- Creation of standardized hardware component error logs
- Analyzing component error logs - to take RAS actions including repair actions
- Observability into every error detected and logged within the platform (system)

The remainder of this section provides details on these challenges.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

2.1. Creation of Component Error Logs

The approaches employed by the existing data center [nodes](#) to create error logs may be sub-optimal:

- Server fleet operators do not know what needs to be logged for hardware errors
 - Modern silicon can have thousands of registers
 - It is hard to determine which ones are needed to diagnose and act upon failures
- Server fleet operators have different data center usage models
 - These models affect how error logging is implemented
 - These models are described in more detail in the Background section

For example, some implementations use an operating system-based error logging approach (“[OS-first](#) error logging”). OS-first error logging has limitations in the case of severe errors where the node is unable to run software. Another example is related to the bare-metal use-case where the end user can install their own operating system. In these nodes, OS-first error logging will report errors to the end user rather than the server fleet operator.

To address this issue, many nodes support a [platform firmware](#)-based error logging approach (“[FW-first](#) error logging”) where an interrupt is sent to runtime host firmware to perform error logging. FW-first error logging can cause problems for server fleet operators due to both the possible performance impact on application software and potential security vulnerabilities.

2.2. Interpretation of Component Error Logs

It is difficult for server fleet operators to understand [hardware component](#) errors in nodes in their data centers. It is difficult to know what data needs to be logged when there is a failure, it is often hard to gather that data and the server fleet operators may lack sufficient understanding of the hardware component needed to interpret the error logs. This can result in incorrect, insufficient, or inefficient service actions.

3. Compliance with Open Compute Project Tenets

This section describes how this document complies to the OCP tenets.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

A full explanation of the OCP core tenets can be seen [here](#).

3.1. Openness

These requirements propose a fault management architecture that promotes interoperability and collaboration between server fleet operators and ecosystem players including silicon suppliers. It provides requirements that will help third parties build components that fit within this framework.

3.2. Efficiency

This document describes how to gather and analyze data for hardware errors more effectively, improving our ability to mitigate failures and repair failing hardware. Improved accuracy of error analyzers and augmented self-repair capabilities described in this document will result in fewer part replacements, better utilization and reduced capital expenses.

3.3. Impact

The framework described in this document addresses key fault management challenges for server fleet operators. It is extendable to other ecosystem stakeholders and enables more innovation.

3.4. Scale

The framework described in this document solves pain points that server fleet operators face with managing large fleets. Accurately diagnosing failures enables server fleet operators to better scale their fleet by automating repairs, including enabling self repairs. The framework makes it easier for server fleet operators to integrate new products in their fleets by providing standard interfaces for collecting hardware error data, analyzing it and acting upon it.

3.5. Sustainability

This infrastructure enables more accurate diagnosis of hardware failures, enables RAS features that mitigate failures and enable self repair. This enablement reduces the environmental impact of server fleet operators by reducing the number of parts replaced, enabling self-repairs, and extending the useful life of the infrastructure.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

4. Background

The RAS solutions may vary based on the data center usage-models and the software the SFO controls. Therefore, it is important to define data center usage-models prior to defining the error handling methods design requirements. For the sake of simplicity, this document will focus primarily on three data center usage-models.

4.1. Primary data center usage-models

Server fleet operators deploy infrastructure supporting many types of usages. The primary usage-models can be summarized into three broad categories as described below.

Bare-metal

- The server fleet operator owns hardware and firmware, including [management core firmware](#).
- Customer (tenant) is responsible for the OS and rest of the software stack (services and applications); the server fleet operator cannot rely on services from the OS

Infrastructure

- Server fleet operator owns hardware, firmware, OS/hypervisor and some part of the software stack (services).
- The customer is responsible for the rest of the software stack (e.g., guest OS, applications).

Application

- The server fleet operator owns hardware, firmware, OS, and software stack (services and applications); the entire infrastructure.
- End users use the applications and have no visibility into the rest of the infrastructure.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

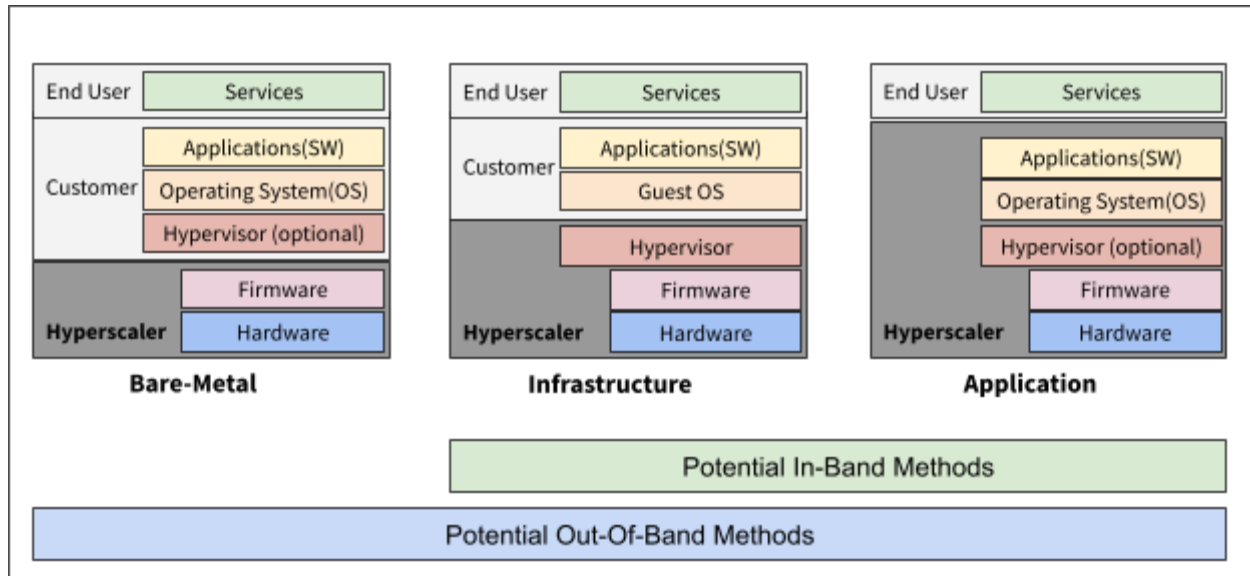


Figure 1: Data Center Primary Usage Models

There are various types of error handling methods feasible within the hardware components used in modern data centers and can be broadly categorized into in-band and out-of-band methods. The next section clarifies the scope of such methods.

4.2. In-band and Out-of-band Methods Overview

In-band and out-of-band methods are complementary and can co-exist, even on the same node. The criteria used to decide between the two methods depends upon the usage-models and error classes. The figure below attempts to provide a high level view of these two methods:

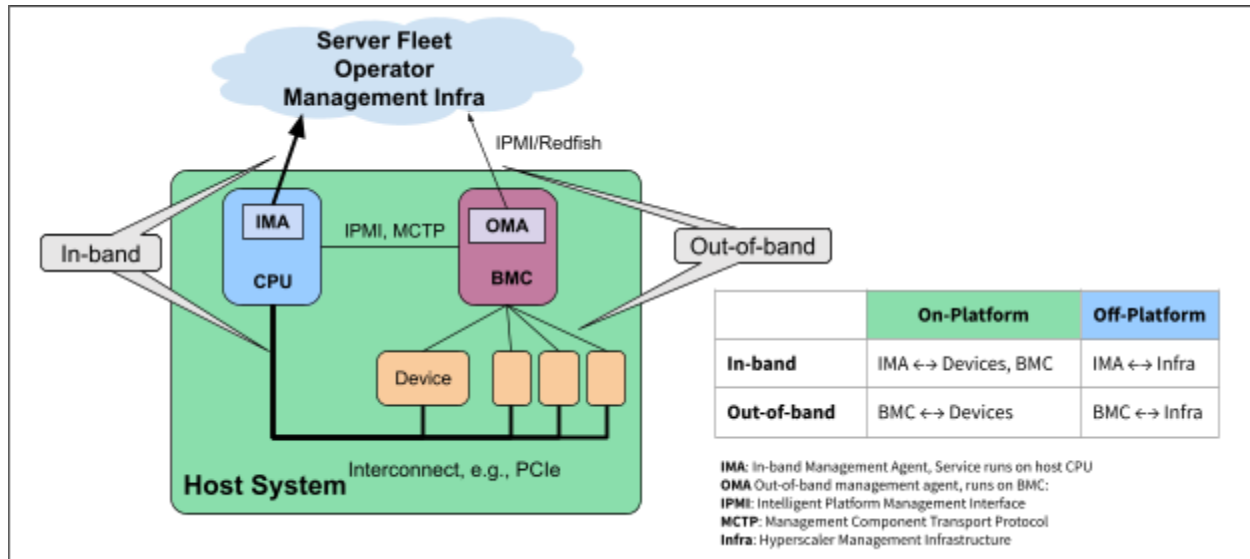


Figure 2: In-band and out-of-band methods for HW Fault Management

There is an assumption that all the devices including CPU, Accelerator/GPU, NIC, NVMe™ are interconnected through high speed in-band interfaces such as PCIe®¹. An Inband Management Agent (IMA) has access to each device over the same high speed interface (e.g., CCI plug-in, ACPI plug-in). Similarly, an Out-of-band Management Agent (OMA) located within the BMC would have access to the devices via an out-of-band interface (e.g., PLDM plug-in, CCI (MCTP) plug-in). In-band and out-of-band methods are defined in the context of the point of view. For example:

1. From the on-platform point of view, the interface between the IMA and the devices is defined as in-band. The interface between BMC (with Out-of-band Agent) and the devices can be routed using out-of-band method or in-band method depending upon the implementation.
2. From the point of view of off-platform, the interface between the Server Fleet Operator Management Infrastructure (Infra) and the IMA is defined as in-band and the interface between the HS and the BMC is defined as out-of-band. The HS is expected to use DMTF standards based Redfish semantics to communicate with BMC. There is no industry standard/semantics for the HS to communicate with the IMA and it is desirable to define a standard or adopt Redfish semantics to communicate with the IMA.

¹ Other names and brands may be claimed as the property of others



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

4.3. Hardware Component Error Classification

The HW error handling solution needs to address errors occurring within HW components (e.g., CPU, memory, NIC, high speed IO links, and cables). This classification includes hardware error severities.

The severities in this section are defined from the hardware's perspective and software may decide to treat the error as a different severity as necessary. For example, hardware might treat attempting to use a poisoned cache line as a recoverable error since the OS might be able to handle the error by killing the affected process or by the process recovering from the data that was lost to poison. But in some cases, the poisoned data might be in a critical OS structure and the OS may decide that it cannot continue running. We still classify the poison consumption error as recoverable in hardware because software might have been able to recover.

The table below lists and briefly describes hardware error types:

HW Error Type	Severity	Example Cases	System Crash	Comments
Corrected	Corrected	Memory, PCIe* link, UPI* link corrected error.	No	1. The error shall not affect correctness of software running on the host. 2. OS/BMC utilities shall report the errors. 3. Appropriate error storm handling is required, e.g., revert back to sampling. 4. Any race conditions shall be avoided in handling corrected errors. 5. SFO Infrastructure should determine the remediation or repair actions.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

HW Error Type	Severity	Example Cases	System Crash	Comments
Non-fatal - recoverable	Non-fatal, Recoverable	Memory/Cache Uncorrected Error, PCIe* Link non-fatal (with E/DPC)	No	1. The platform shall capture error logs in a way that makes them visible to the SFO. 2. Platform shall return the control to OS context. 3. OS/BMC utilities shall report the errors. 4. Appropriate error storm handling is required, e.g., revert back to sampling. 5. There must be no race conditions between in-band vs out of band error logging, e.g. The kernel must not be able to clear the error before the BMC FW reads the logs or vice-a-versa. 6. The Kernel either recovers or panics, depending on the nature of the failure and kernel's capabilities. 7. SFO Infrastructure should determine the remediation/repair actions.
Non-fatal - Non-recoverable	Non-fatal, Non-recoverable	PCIe* Link non-fatal error (Without E/DPC)	Yes	1. The platform shall capture error logs in a way that makes them visible to the SFO. 2. Platform shall return the control to OS context. 3. Kernel should gracefully panic because it is unable to recover. Graceful implies the platform collects error logs prior to reset. 4. SFO Infrastructure should determine the remediation/repair actions.
Fatal - system firmware can execute	Fatal	CPU internal parity error, PCIe* Link fatal error (without E/DPC)	Yes	1. The system firmware shall capture error logs in a way that makes them visible to the SFO and reset. 2. SFO Infrastructure should determine the remediation and/or repair actions.



HW Error Type	Severity	Example Cases	System Crash	Comments
Fatal - system firmware cannot execute	Catastrophic	VR failed, the host CPU cannot run code, but BMC is still active. CPU/System FW crash or hang preventing further code execution. Boot time failures, e.g. attestation failures.	Yes	1. The platform shall capture error logs in a way that makes them visible to the SFO. 2. SFO Infrastructure should determine the remediation and/or repair actions.
Fatal - BMC not available	Catastrophic	BMC fails, loss of AC or BMC VR faults	Yes	1. BMC cannot capture any logs, hence out of scope for this requirement document. The hardware must have capabilities for diagnosing the issue (e.g. Power LED does not light).

5. Fault Management System Architecture

This document aims to address the challenges listed in [section 2](#) by proposing a Hardware Fault Management architecture. It describes both in-band and out-of-band fault management system goals.

5.1. In-band Fault Management System Goals

This section focuses on the requirements for in-band hardware fault management. This includes RAS configuration, error logging/reporting, RAS actions, and error injection interfaces managed through the ‘In-band Management Agent’. Such in-band-methods are useful for hardware fault management usage-models such as ‘infrastructure’ or ‘Application’. The figure below further clarifies a high level view of in-band error flows.

It is still possible for either IMA or Firmware to save the error logs to BMC or any other out-of-band agent, allowing Server Fleet Operator management infrastructure to pull the error logs. In this case, from the Server Fleet Operator management infrastructure point of view it would be using an out-of-band method to collect the log.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

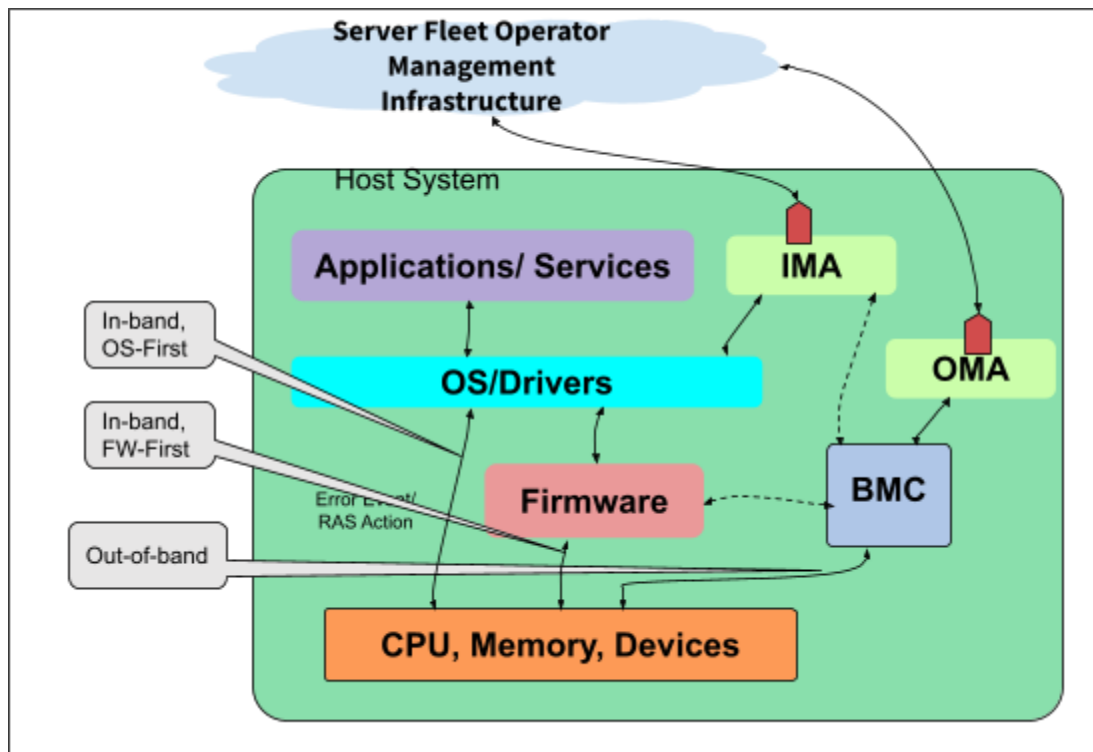


Figure 3: In-band Methods for HW Fault Management

It is important to note that “Firmware” does not necessarily imply ‘System Management Mode (SMM)’. In the majority of the public cloud infrastructures, SMM based methods are not desirable.

5.2. Out-of-band Fault Management System Goals

This section focuses on the requirements for the out-of-band hardware fault management solution. This includes RAS configuration, error logging/reporting, RAS actions, and error injection interfaces managed through the Base-board Management Controller (BMC). The BMC uses standard formats to capture data necessary to diagnose and repair HW failures. Standard interfaces allow the BMC to communicate these error logs to the SFO’s management infrastructure, and allows component vendors to provide analyzer tools that interpret error logs in the SFO’s [management infrastructure](#). For example, out-of-band fault management allows SFOs to log and handle errors in ‘Bare-metal’ usage-models.

5.3. Integration into the SFO infrastructure

The goal of this system architecture is to achieve improvements in node reliability, availability, and serviceability by standardizing behavior. This goal can be achieved by allowing Server Fleet Operators to more quickly validate node behavior, discover and invoke RAS features more quickly when needed, and seamlessly deploy updates to improve data center infrastructure reliability. In order to achieve this, the hardware fault management solution requires the ability to integrate vendor specific fault handlers along with vendor specific decoders and analysis tools. The figure below clarifies the flow at a high level showing both in-band and out-of-band methods.

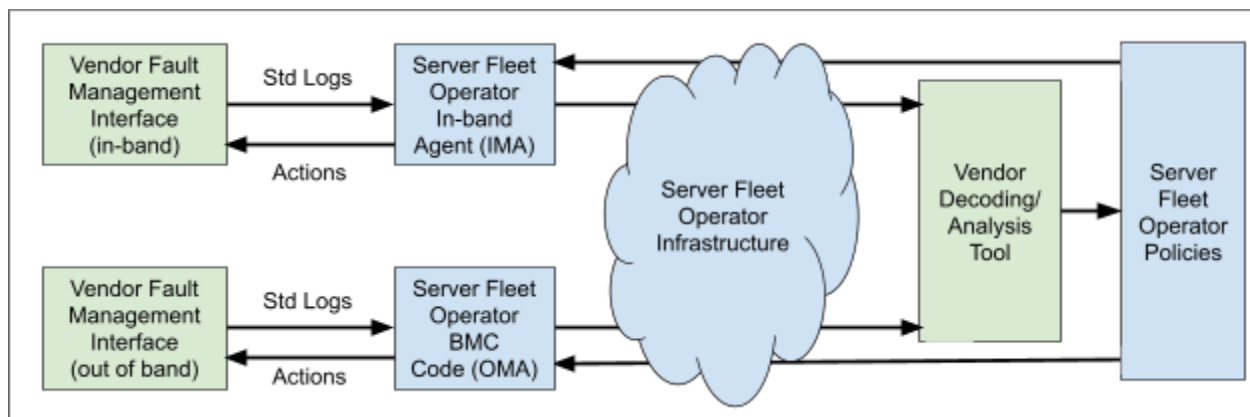


Figure 4: High Level Concept of integrating vendor specific fault handlers

The above figure shows a high-level view of the system architecture for in-band and out-of-band error handling. The green boxes represent component vendor deliverables: an out-of-band (OOB) error logging interface running in BMC firmware and/or in-band error handlers via kernel and driver modules; plus an analysis tool to run in the management infrastructure. The blue boxes represent Server Fleet Operator infrastructure requirements: BMC code for OOB handling and In-band Management Agent (IMA) running within each host; plus the datacenter infrastructure to transmit error logs and service actions from the node to the analysis tools; plus a policy engine to choose when and how to apply service actions. The interfaces between all of the boxes are standardized to allow a Server Fleet Operator to seamlessly deploy components from multiple vendors, and to allow a component vendor to develop a single infrastructure that is used by multiple Server Fleet Operators.

5.4. In-Band and Out-of-Band Methods are not Mutually Exclusive

Server Fleet Operators will use both in-band and out-of-band methods to manage faults in their fleets, depending on their business needs, OS support, and the severity of the error. In some cases, a machine might simultaneously support both in-band and out-of-band methods.

Server Fleet Operators that use OOB methods to drive RAS features and diagnose failures will also need error logs to go to the OS so that the OS can react as needed. For example, an OS needs to decide how to respond when software attempts to consume poison. Another example is when PCIe Downstream Port Containment is triggered; a Server Fleet Operator that is using OOB methods needs to see the error in order to diagnose the system, but the OS and drivers are required to recover when a PCIe link goes down.

In some cases, an OOB method may also need to pass actions to the OS for the OS to execute on its behalf. For example, a vendor-specific DDR DRAM analyzer might recognize a failure mode and suggest that the OS map out pages of memory that have not yet seen an error. In such cases, the OOB methods will send a Common Platform Action Descriptor (CPAD) containing the action to the vendor's fault management interface and that interface may be able to send an error message to the OS, telling the OS about the requested action.

In some cases, a system might experience a fatal error where it is no longer able to run software. In such cases, a Server Fleet Operator that is using in-band methods may need to use an OOB method to collect error logs and determine what failed.

In-Band and out-of-band methods are intended to be complementary and be used as options that allow Server Fleet Operators to apply the methods as needed for their business needs and as needed for different severities of errors as shown in [Section 6](#) of this document.

5.5. Goals of the OCP Fault Management Infrastructure

- Standardize error monitoring and reporting between platform components and management infrastructure (Error Monitoring Agent, e.g. IMA and/or OMA);
- Standardize input format for error analysis tools in the management infrastructure (Vendor [Decoding](#) and Analysis Tool);



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

- Standardize the format of the Analysis Tool output that is consumed by the management infrastructure. This shall include Vendor specific information about the error, including the decoded information.
 - Analysis results - information about what is going wrong
 - Recommended action to take for the error. This includes the platform specific actions (e.g., PPR) and the remediation/repair actions (e.g., reset, re-seat, replace). Such actions would then be consumed by the management infrastructure to apply fleet level policies and then further actions would be initiated including passing the action back to host, or initiating the remediation/repair.
- Standardize interfaces for a BMC or hardware component to communicate error actions to an operating system; Likewise standardize the interfaces for the IMA to communicate error actions to the BMC via OMA.
- Allows the SFOs to share data in a standardized way that enables hardware vendors to improve diagnosis accuracy
- Improve the performance and security of application software by eliminating the use of processor compute cycles for error handling; and
- Be applicable to all primary cloud usage-models

6. Proposed System-Level Error Handling Capabilities

The system architecture described above can apply to specific classes of errors such as corrected, non-fatal/recoverable, non-fatal/non-recoverable that are visible to the IMA/OMA and do not result in system crash or kernel-panic. For example in some CPU architectures, FW-First implies SMM based error handling which is not recommended due to security concerns.

Out-of-band error logging means that error logs are somehow collected and delivered to the SFO infrastructure by the BMC. This is in parallel with in-band error logging. When both the BMC and in-band entities are logging the error, care must be taken to avoid race conditions between them. In some cases, silicon may have dedicated registers and microcontrollers that help provide error logs to the BMC and prevent race conditions.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

The tables below summarize the proposed error handling actions. In these tables, a scheme is described as “Not Feasible” if there are any scenarios where it does not meet the SFO requirements. Additionally ‘FW-first’ and ‘OS-first’ are part of In-band Methods.

6.1. Corrected Errors

Usage	FW-first	OS-first	BMC-based
Bare-metal	Feasible	Not Feasible Because the OS is not owned by the Server Fleet Operator and there is no way to pass logs from the OS to the Server Fleet Operator’s management infrastructure	Feasible
Infrastructure	Feasible	May not be Feasible Concern: Infrastructure providers sometimes use third party hypervisors that do not run the OS-based agents that the providers tailor for the rest of their fleet.	Feasible
Application	Feasible	Feasible	Feasible

6.2. Non-Fatal/Recoverable Errors

Usage	FW-first	OS-first	BMC-based
Bare-metal	Feasible	Not Feasible Because the OS is not owned by the Server Fleet Operator and there is no way to pass logs from the OS to the Server Fleet	Feasible Concern: Though feasible, the BMC will need a race-free way to get the error logs without interference with the OS/Platform.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

		Operator's management infrastructure	
Infrastructure	Feasible	Feasible	Feasible
Application	Feasible	Feasible	Feasible

6.3. Non-fatal/Non-recoverable Errors and Reset feasible

Usage	FW-first	OS-first	BMC-based
Bare-metal	Feasible	Not Feasible Because the OS is not owned by the operator and there is no way to pass logs from the OS to the operator's management infrastructure	Feasible
Infrastructure	Feasible	Feasible Assuming that the OS and IO are healthy enough to write the logs.	Feasible
Application	Feasible	Feasible Assuming that the OS and IO are healthy enough to write the logs.	Feasible



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

6.4. Fatal - System Firmware Can Execute

In this case, firmware is able to collect the error logs after the fatal error. This might be implemented via a warm reset (that preserves error logging registers), crawling out from the error condition, getting the error logs from a management processor or via some other method.

Usage	FW-first	OS-first	BMC-based
Bare-metal	Feasible Firmware may be able to log the error and pass error logs to the server fleet operator.	May not be Feasible If the CPU is able to run code well enough to boot the OS, System Firmware may be able to send logs to the server fleet operator. The OS cannot be relied upon to pass logs to the server fleet operator.	Feasible Since the BMC can still run, collect the errors if accessible and forward them to the operator's management infrastructure
Infrastructure	Feasible Firmware may be able to log the error and pass error logs to the server fleet operator.	May not be Feasible The OS crashed and was not able to log the error directly. But some platforms might log the error in their BIOS and write to the BERT for the OS to pick up on the next boot.	Feasible Since the BMC can still run, collect the errors if accessible and forward them to the operator's management infrastructure
Application	Feasible Firmware may be able to log the error and pass error logs to the server fleet operator.	May not be Feasible The OS crashed and was not able to log the error directly. But some platforms might log the error in their BIOS and write to the BERT for the OS to pick up on the next boot.	Feasible Since the BMC can still run, collect the errors if accessible and forward them to the operator's management infrastructure



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

6.5. Fatal - System Firmware Cannot Execute

In this case, the system is unable to run code on the host CPUs and neither firmware nor the OS can run. Management processors might be able to continue running and either the management processors or the BMC may be able to log the error. This case includes situations where there is a security issue that prevents firmware from running. It also includes cases where there are environmental problems such as VR failures that prevent system firmware from running but, even in such cases, the BMC may be able to collect logs if it runs on a separate voltage rail.

Usage	FW-first	OS-first	BMC-based
Bare-metal	Not Feasible Neither system firmware nor the OS is able to run.	Not Feasible Neither system firmware nor the OS is able to run.	Feasible The BMC may be able to log errors from silicon such as the host CPU and from other components such as voltage regulators.
Infrastructure	Not Feasible Neither system firmware nor the OS is able to run.	Not Feasible Neither system firmware nor the OS is able to run.	Feasible The BMC may be able to log errors from silicon such as the host CPU and from other components such as voltage regulators.
Application	Not Feasible Neither system firmware nor the OS is able to run.	Not Feasible Neither system firmware nor the OS is able to run.	Feasible The BMC may be able to log errors from silicon such as the host CPU and from other components such as voltage regulators.

In the case of a full power loss, the platform shall try to signal the problem as it goes down. It may be possible for the BIOS or BMC to send a message before they go down. Otherwise, repair technicians will need to investigate it and will see that power LEDs are not lit. In the case of BMC VR faults, error handling behavior would be implementation specific and is outside of the scope of this document.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

7. Solution Architecture

The prior section demonstrated the advantages for both in-band and out-of-band methods for hardware fault management. This section proposes an architecture that encapsulates both these methods.

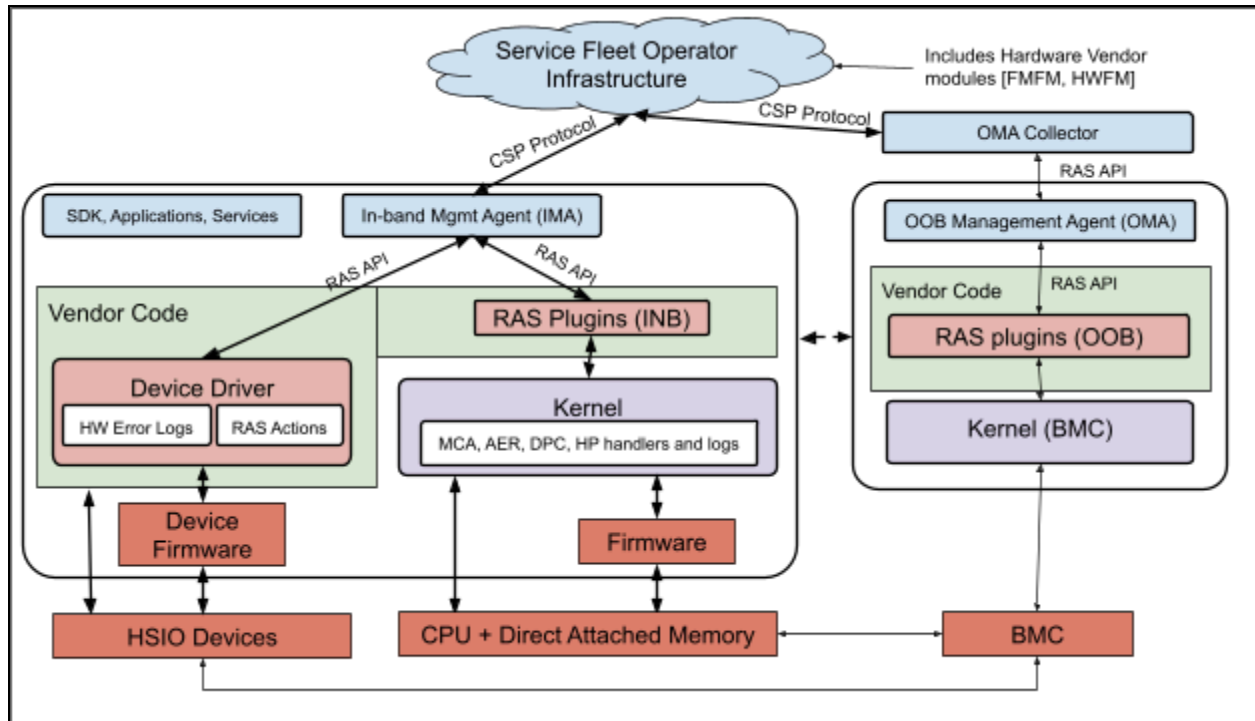


Figure 5: High-level view of the Proposed Fault Management Framework

The HW components such as CPU, memory and devices are the sources of errors and events. Based on the firmware-first or OS-first policy, the errors or events could be handled by the FW (such as via SMM or an embedded uController), or by the SW (such as device drivers or an OS kernel) running in the OS context. In this diagram we propose that the IMA should be isolated from the details of the vendor specific HW components and an abstraction layer such as RAS API be added which can provide the error data as well as a path for RAS actions. The kernel handles the errors and events based on the notification and logging mechanisms based on industry standards(e.g. x86, RISC, PCIe*) and propagates them to the upper layers of software. Device drivers expose the error data to the IMA in a standardized format hence satisfying the requirement of SW scalability for the Server Fleet Operator infrastructure. The RAS plugins in the kernel may also provide a path for RAS actions. Examples of the RAS drivers/plugins are RASDaemon (uses traces), EDAC driver (uses sysfs) or a RAS API driver.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

7.1. SFO Protocol

The SFO Protocol is a Server Fleet Operator's specific implementation that allows the SFO infrastructure to communicate with a platform/machine/host containing an Inband Management Agent (IMA).

7.2. Inband Management Agent (IMA)

The Inband Management Agent is a software service located at the head-node or the host or simply the main CPU subsystem. The purpose of the IMA is to provide an interface between the Server Fleet Operator's management infrastructure and the platform used in the datacenter. The IMA is expected to be developed by Server Fleet Operators. The IMA expects to interface with the rest of the platform using a standard API (e.g., RAS API).

7.3. Inband RAS Plugins (INB)

Inband (INB) RAS Plugins are the software modules whose purpose is to provide a communication channel for the RAS API towards the IMA on one side and existing kernel or HW on the other side. There could be more than one plugin depending upon the implementation choices. RAS Plugins may contain standardized or platform specific modules and are accessible to the Server Fleet Operator. It is recommended to ensure backward compatibility of the RAS Plugins to ease the adoption of the new capabilities without disrupting the existing systems already deployed, but that is not a strict requirement.

7.4. RAS API

RAS API is defined as part of a separate OCP workstream and offers a standard way to interact with the platforms to collect error logs, to initiate RAS actions, and to monitor the RAS actions taken autonomously by the platform. The API prescribes the standardization of the content and action commands that are exposed via the RAS interface, and RAS plugins prescribe how a communication channel could enable the RAS API. RAS plugins and components below it are expected to implement the RAS API.

7.5. Kernel

The kernel is part of the software stack that provides a standard way to interface with the host CPU subsystem. It includes the RAS related modules and handlers for architectural and standardized error reporting mechanisms such as MCA, AER and APEI. The OS/kernel may contain an in-band error logging agent, which may also be able to trigger RAS actions on the node it runs on.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

7.6. CPU + Directly Attached Memory and HSIO (High Speed IO) devices

CPU+Memory and HSIO devices are part of CPU node hardware. The memory can also be implemented as part of HSIO, for example CXL[™] Type 3 Memory Module (Memory Expander with Memory Controller) devices. HSIO devices can be any high speed interface based device including GPUs, Accelerators, CXL Endpoints, NICs, NVME Drives, etc.

7.7. Firmware

Firmware components initialize the platform in order to operate the services hosted. During runtime firmware may perform low level system management tasks while the OS is running, such as error handling, power management and vendor designed code. The Firmware First mode of error handling is handled by this layer.

7.8. Device Firmware

Device-specific firmware runs on the device to control and manage devices. The device firmware's responsibility is to initialize and configure a device and ensure proper operation and functionality. Such firmware typically implements low-level protocols as per the device's usage model including graphics, networking, storage, etc. Health/error monitoring and telemetry data is logged by device firmware. Upon detection of adverse events or threshold violations in hardware, device firmware can signal errors to the system for further handling and potential remediation.

7.9. Device Driver

Device software drivers enable the system OS to interface directly with individual hardware components. Device drivers provide the OS a mechanism to utilize and manage specific hardware resources through an abstracted layer. Error handling at the device driver layer includes leveraging OS error-signaling routines and associated error message logging. In the presence of failures, endpoint device drivers can invoke RAS Actions such as retry and reset flows to attempt data and/or device recovery. In particular, PCIe^{*} AER device drivers coordinate with the system AER driver to provide granular hardware error logs and perform targeted error recovery actions.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

7.10. Proposed fault management using inband methods

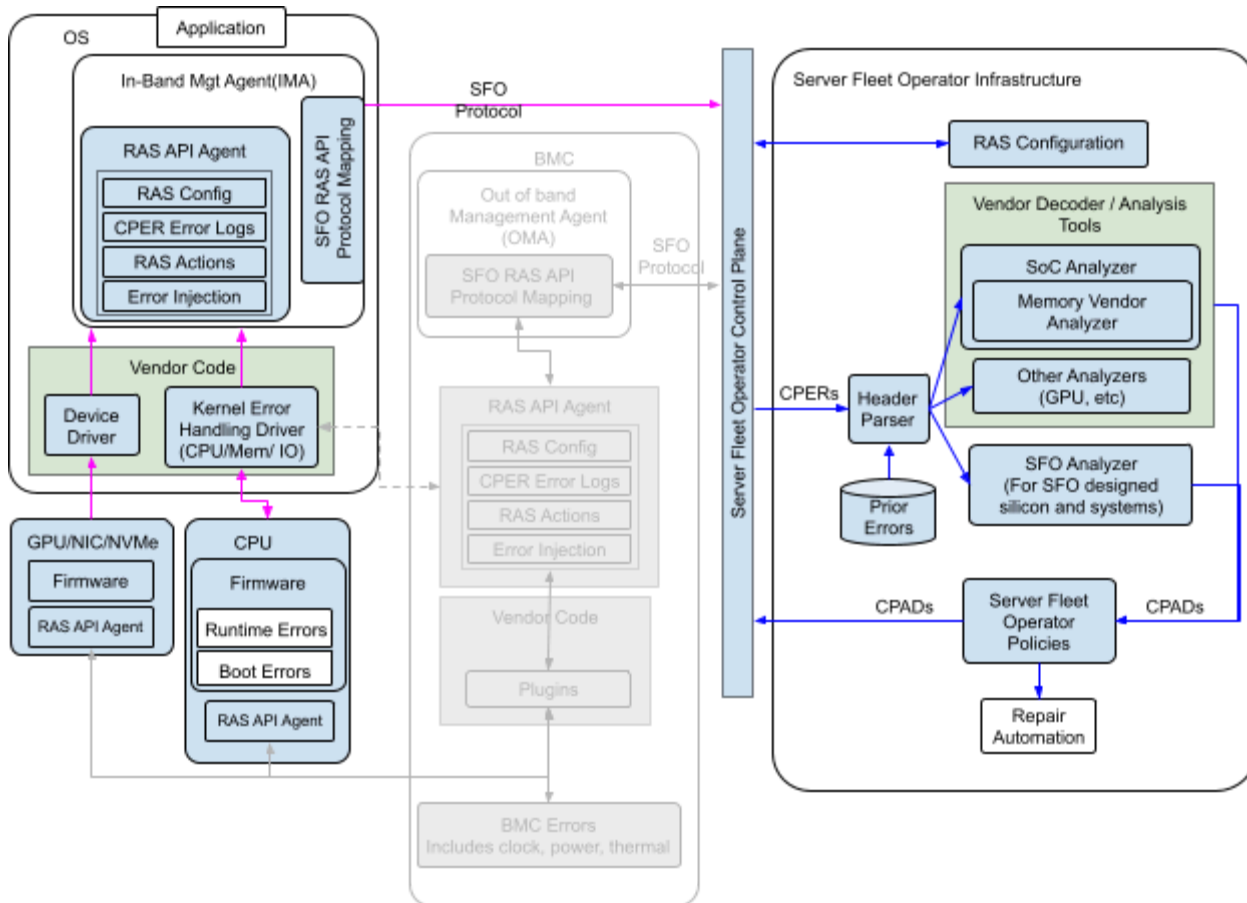


Figure 6: Proposed Fault Management Framework using in-band methods

As part of the inband methods, the next section provides an example of the existing Linux OS implementation for PCIe* error handling flows and various possible improvements.

7.11. An example of the PCIe* error handling flow for OS-first, In-band Methods

This section describes a Linux OS-based PCIe* link error handling flow. The figure below shows a Linux OS-based PCIe* link error handling flow, including associated RAS Plugins: RASdaemon, RAS API Agent, and the In-band Management Agent.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

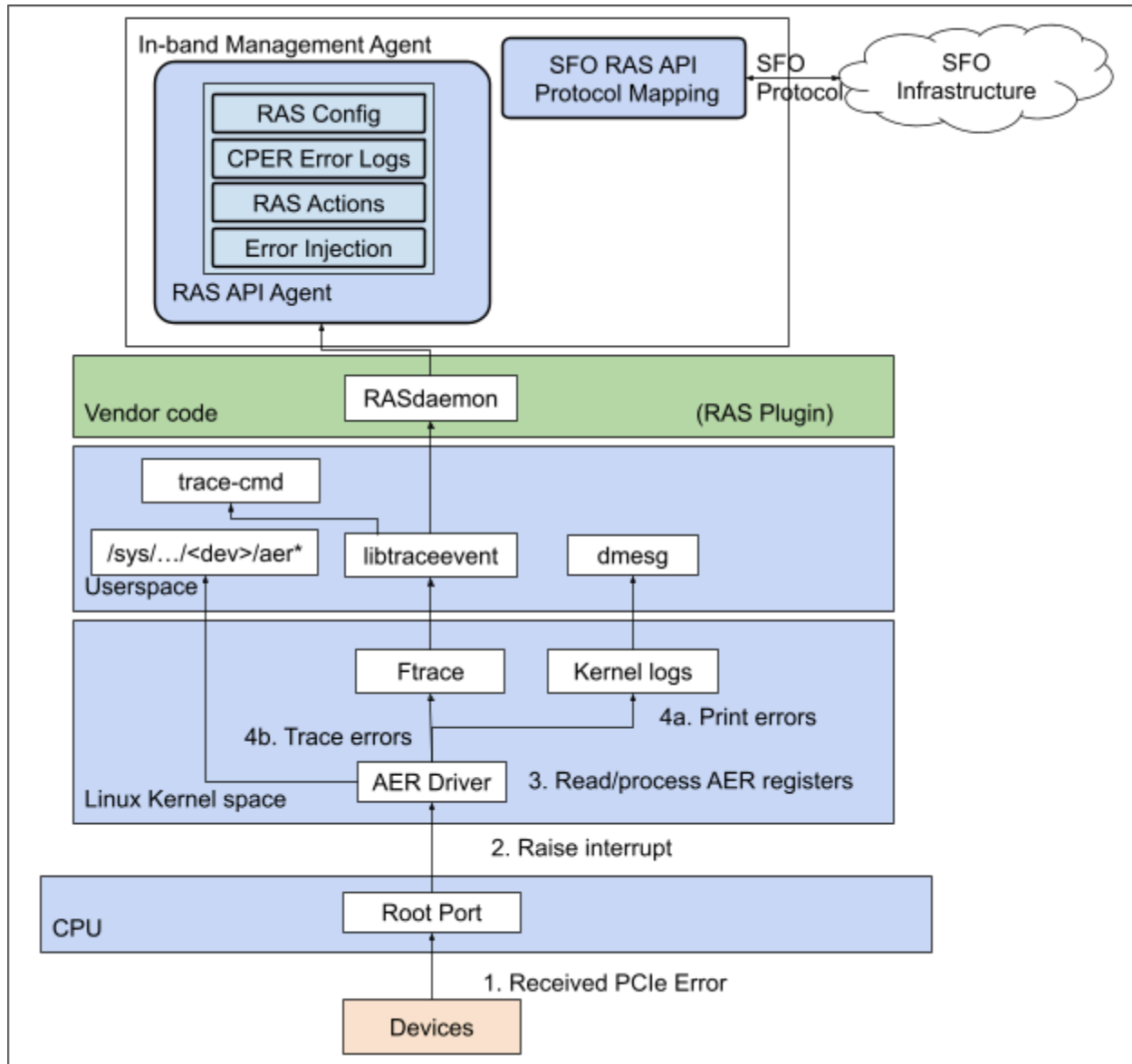


Figure 7: **PCIe*** Error flow between HW and SW using in-band method

PCIe* errors received at the root port of AER-compliant devices raise an interrupt to the AER driver. The driver reads various AER registers to gather info (e.g. type, severity, status, mask, TLP header) and logs it in multiple locations accessible by userspace as shown in the table below:



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

Location	Type	Notes
sysfs	Statistical counters	Grouped by device, severity, and type
Kernel logs	Full debug information	Human-readable text (requires parsing). It is highly desirable to use the standard CPER format that can be processed by the IMA and can also be presented in human readable format using JSON.
Tracepoints	Full debug information	• Only generated if tracing enabled • Exported as fields and format string • Ingested by upstream tools ('libtraceevent')

Fault management at the datacenter-scale may use a userspace daemon (e.g. [RASdaemon](#)) to monitor and gather telemetry (to surface errors for repairability flows) and set error containment/telemetry configurations across different platforms (e.g. overrides/defaults).

The Downstream Port Containment (DPC) driver acts similarly, except it only prints to kernel logs (no tracepoints or sysfs entries) and therefore further enhancements are required. Refer to [PDF PCIe Error Handling in Kernel.pdf](#) for more details of the proposed Linux Kernel support required to enhance it further.

7.12. Overall view of fault handling combining in-band and out-of-band methods

[Figure 8: Proposed Overall Fault Management Framework](#) shows an example of how the vision can be met using IMA, OpenBMC™, and RedFish® interfaces to the Server Fleet Operator Infrastructure. The light blue boxes are components that are being influenced as part of this proposal. Our intent is to have an open-source framework that supports closed source modules where needed to protect vendor IP. Server Fleet Operators might also implement extensions to the infrastructure that are not upstreamed such as the Server Fleet Operator Policies that determine how the Server Fleet Operator reacts to the output of all of the analyzers.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

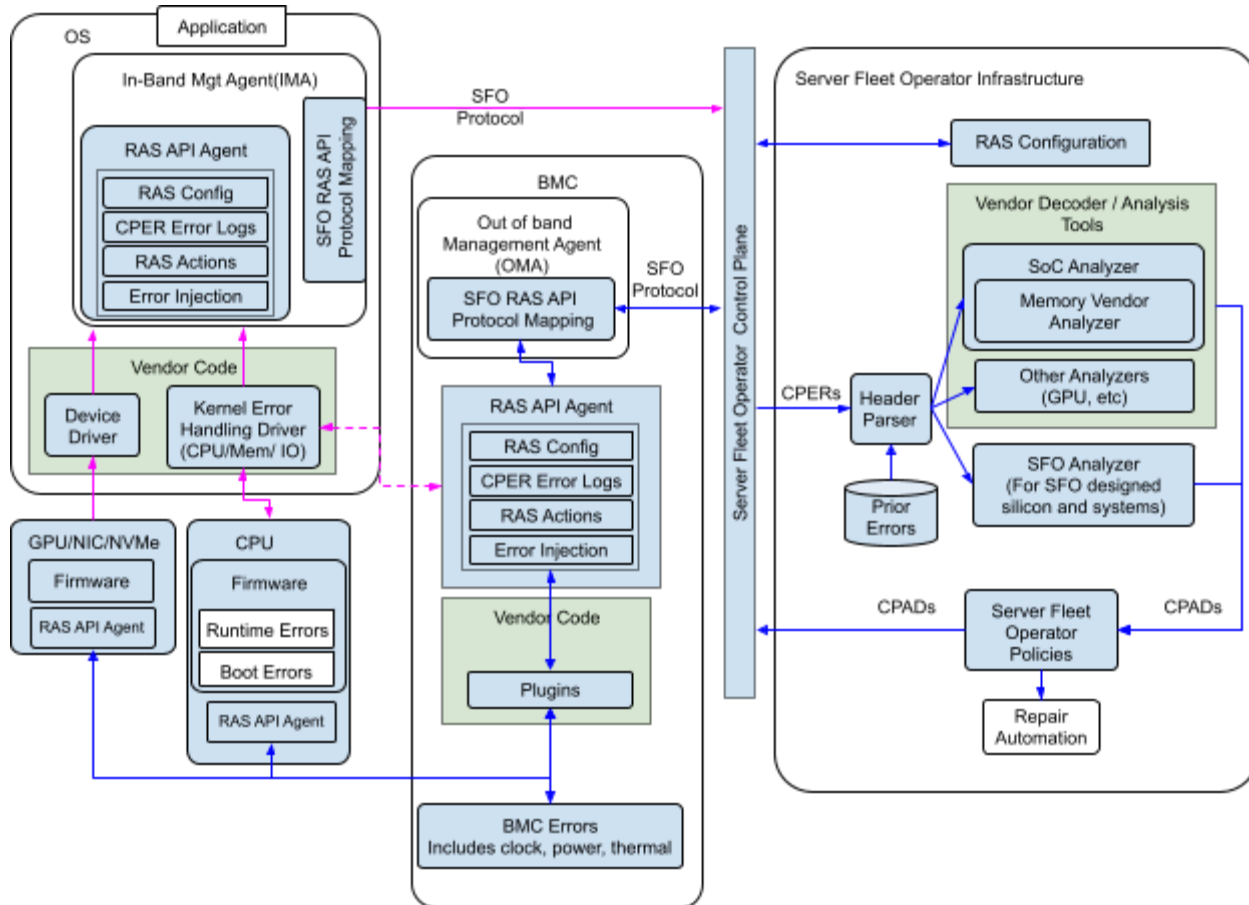


Figure 8: Proposed Overall Fault Management Framework

The figure shows the following:

- Kernel (Operating System): The OS kernel may contain an in-band error logging agent, which may be covered in a future contribution from the OCP HW Fault Management team.
- Out-of-Band Support: Each hardware component is required to provide OOB error logging capabilities that can be utilized by the BMC. This hardware must enable automated logging of hardware events in a manner that
 - (a) avoids race conditions with in-band logging agent(s) in accessing accurate error state;
 - (b) provides a mechanism to limit error log access/actions by an agent that is not managed by the fleet operator (e.g. node OS in bare-metal use-case)



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

- (c) does not contain any Personally Identifiable Information (PII). The OOB Hardware may also apply supported RAS actions requested by the BMC.
- The hardware component should also support the population of errors logged by the OOB hardware in the HEST for consumption by an in-band agent such as kernel without the use of vulnerable FW (e.g. SMM) or stealing compute cycles from the compute node.
- RAS API Support in BMC: Each component vendor provides interfaces that support the RAS API and enable OOB Fault Management including discovering RAS API endpoints, configuring RAS features, logging errors and triggering RAS Actions.
- BMC Web (Redfish): For the purposes of this document, the primary goals of the Redfish interfaces in the BMC is to implement the RAS API protocol over Redfish.
- Vendor Code in Fault Analyzer Server Fleet Operator Infrastructure. This code receives the error logs from the BMC, decodes them, analyzes them, and provides diagnostic information with suggested service actions to the Server Fleet Operator Infrastructure, which includes initiating RAS actions. Based on Server Fleet Operator policy, these actions can be sent back to the node to be applied via the BMC.
- Error logs collected in-band or out-of-band are expected to be formatted as Common Platform Error Records (CPER)
- RAS actions sent back to the RAS API are expected to be formatted as Common Platform Action Descriptors (CPAD)
- The dotted line between the BMC RAS API Agent and the Kernel Error Handling driver enables communication between the OS and OOB Fault Management
 - An analyzer may want to pass a RAS action back to the OS (e.g. mark a memory page as bad)
 - An OS that supports this interface could take common OS-driven actions, even on bare metal machines where there is no In-band management agent
 - There may also be value in passing data from the OS to the OOB fault management tools, but this requires scrutiny to ensure that the OS never passes PII/customer data to the BMC

The following section shows how errors captured out of band may be synchronized with the operating system.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

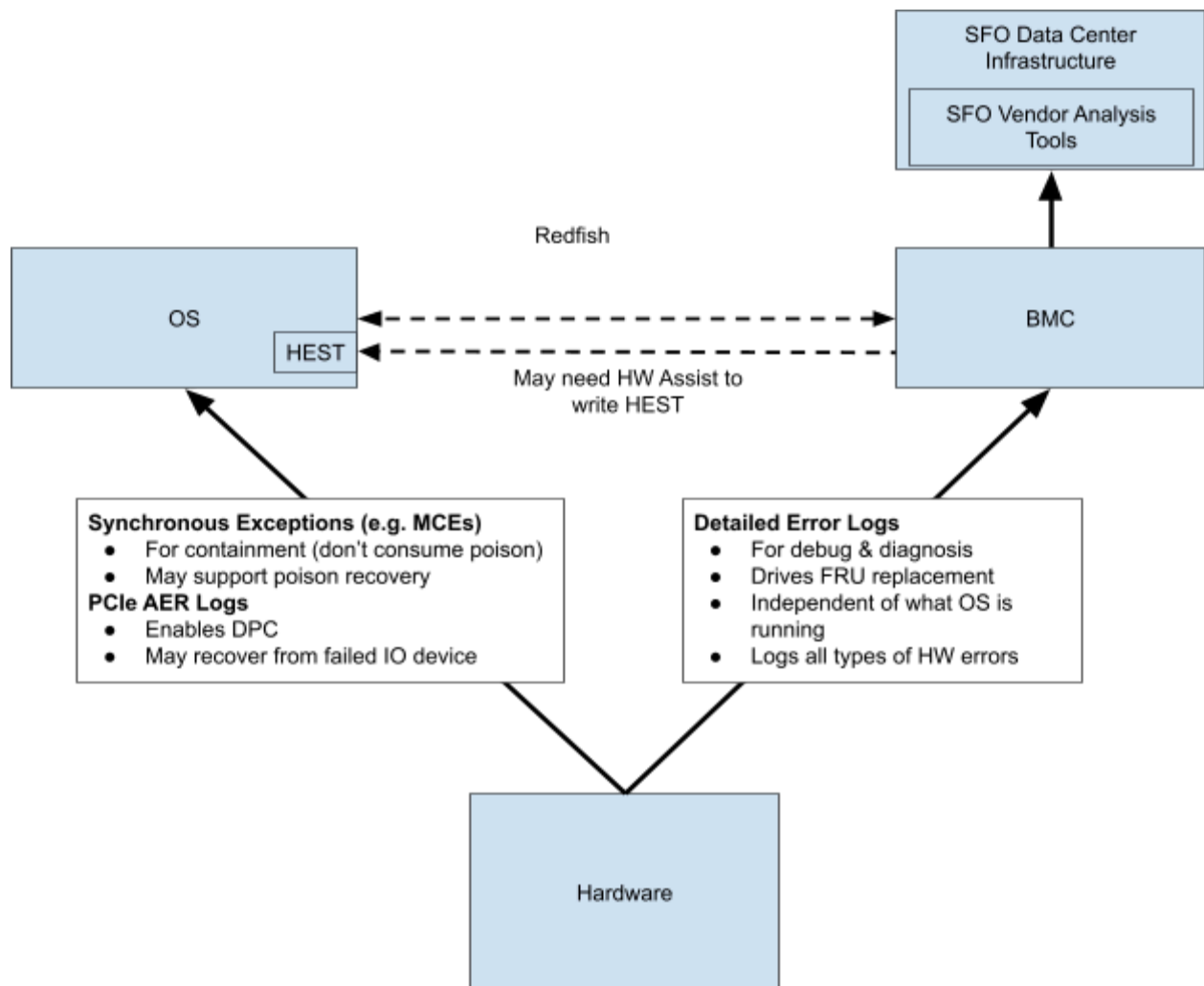


Figure 9: BMC-based error logging and OS error visibility

The OS needs to be synchronously interrupted when it and/or an application tries to use corrupted data or there is a failure where the machine is no longer able to run code reliably. The OS may also contain handlers for errors which can correct an error condition or limit its exposure to the system. There are several ways the OS could see the error. One way is for the BMC to pass the error information to the OS. When the BMC logs an error that the OS needs to see, it will have a way to send the information to the HEST, perhaps with ASL support, but NOT requiring an SMI. For example, if there is poison generated by a patrol scrub, the BMC could pass the error to the OS via the HEST and trigger an SCI so that the OS could map out the poisoned page.

When there is an uncorrected non-fatal error, the OS will get a synchronous error like a machine check to avoid using corrupted data. The BMC will need to get an error log that tells the BMC that the error occurred but the BMC does not need to be synchronously interrupted.

There are two error conditions where the operating system is unable to recover from an error nor collect error data; those conditions are uncorrected fatal reset-path and uncorrected fatal power loss errors. In those cases the BMC is the sole agent collecting error data.

7.13. Error Logging Flow for OS Based Methods

The analysis tools and error logging formats used in BMC-based logging could also be used for OS first logging. Figure 10 describes an OS based architecture. The CPU Vendor Analysis tool could be a plug-in to a tool like EDAC. This does not support bare-metal error logging, but that is OK for server fleet operators that have products in the infrastructure or application tiers [described earlier in the document](#). Some MCEs may be severe enough that the OS is not able to run or send error logs. There are a few workarounds for this issue, but OSs will always be more limited in their ability to log severe errors than a BMC which can log even when the OS is not working. For the most severe errors where the CPUs are stopped, out of band logging from the BMC may be the only alternative.

Note that these drawings are not mutually exclusive. Code running in the OS always needs to be synchronously interrupted when the code tries to use data that has been corrupted and the OS needs to be able to look at the error to determine if it can continue executing. For the worst errors, the OS will not be able to run and the BMC may be the only way to gather error information. Server fleet operators can mix these error logging approaches as needed with our goals supporting the use cases described [above](#).



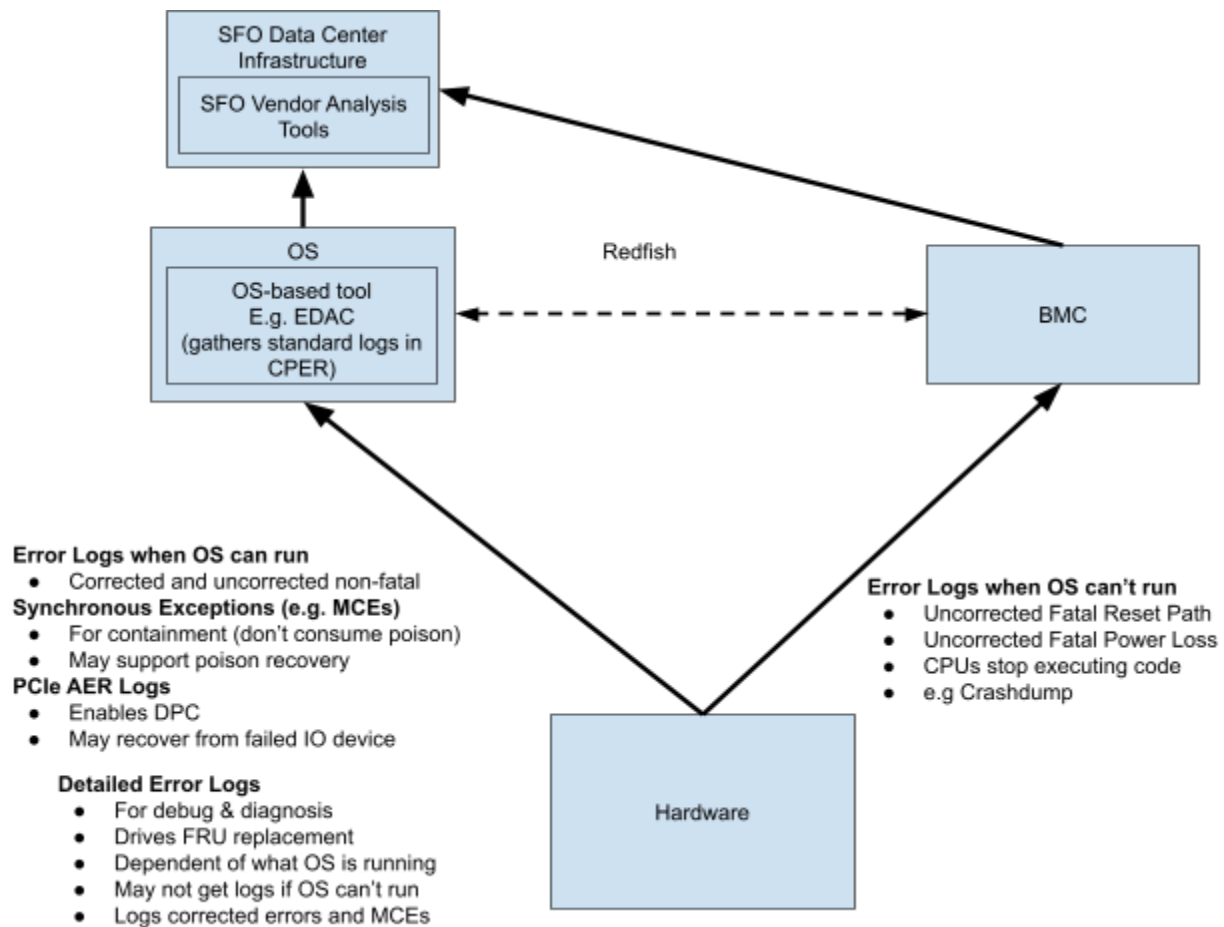


Figure 10: Integrating BMC logging and In-band logging

7.14. Considerations for supporting both IB and OOB Methods

In-band and out-of-band methods shall be implemented in a manner such that they can coexist on the same machine and be used simultaneously. Race conditions between in-band and out-of-band methods shall be prevented. Please see Appendix B - Implementation Considerations for details.

8. Error Handling Flow Examples

This section describes several key error handling flow examples. See the [OCP Fault Management Scenario List for additional examples](#).



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

8.1. Data Poisoning Example

Errors that generate poison require the ability to correlate poison creation (e.g. from an uncorrected ECC error) and poison consumption. In this document we use the term poison consumption to refer to cases where a CPU core or IO device tries to use poisoned data.

When poison is generated, we need to log information about the failure in order to determine what failed and to potentially initiate replacing the failing part.

When poison is consumed, the OS will either crash or recover from poison. Recovery can be done in a variety of ways such as an application handling the error and recreating the affected data, by killing a process that consumed poison or by killing the VM that consumed poison. Recovery is not guaranteed. The corrupted data might be critical to the OS, or errors may be happening at a rate that recovery is not possible. Server fleet operators want to see that the consumptions are occurring in order to monitor the health of customer workloads. A machine that is constantly recovering from poison may not crash a customer's workload, but could affect the quality of their experience on the machine. For example, there may be a loss in efficiency since jobs and tasks that failed need to be relaunched.

After the OS has dealt with the error recovery, the Server Fleet Operator must find the failing component that created the poison and analyze the error that caused the poison so the appropriate actions can take place. On a compute node that supports in-band error logging, the OS looks for the poison creation event in the event logs and determines the source of the poison. With BMC-based logging, the BMC logs the poison consumption event (most likely after the OS has recovered from it) and poison creation events.

The BMC-based logging implementation may not be able to log the errors as quickly as the poison-generation errors occur. In such cases, it is critical to know that the error logs overflowed. This requirement is similar to current OS-based logging solutions that have overflow indications. It is safe to assume that the error logs that are lost because of the overflow are related to the error logs that were successfully captured. For example, a memory failure can cause multiple uncorrected errors to be logged rapidly and possibly overflow the error logging hardware that logs the poison creation. We should be able to tell, from the error logs that we captured before the overflow, that the failure was in memory and determine which FRU(s) were involved. The more severe the failure becomes, the less likely OS recovery is possible and the less likely we are missing something important because of the overflow.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

The following diagram shows some possible sequences where poisoned data is handled:

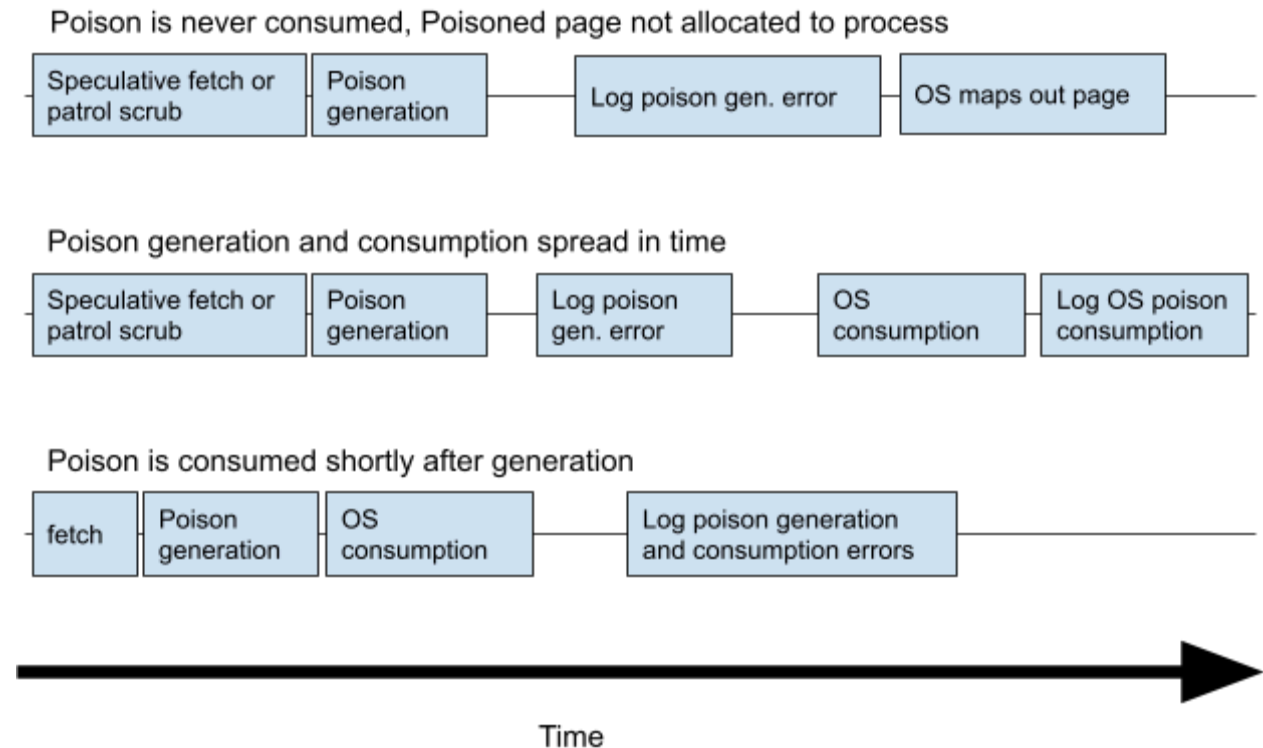


Figure 12: Poison Handling Sequence

In the top example, the poison is generated by a speculative read and is never consumed. It is possible to increase the odds of poison never being consumed by mapping out the poisoned page. It is also possible that an application using the memory could overwrite poisoned data before the poison is consumed and in such a case, there would be no poison consumption event.

In the middle example, poison generation and consumption happen far apart in time. This is possible when poison is generated by a speculative read or a patrol scrub. Server fleet operators would like to be able to correlate the poison generation and consumption even though they may be far apart in time. The most obvious way to correlate them is to use the OS physical address.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

In the bottom-most example, the OS immediately consumes the poison. So there are two errors that happen rapidly in time.

For all of these scenarios, BMC-based error logging needs to see the poison generation and any poison consumption in a race-condition-free manner from the OS. The BMC would see the error logs from two places: the memory controller and the core that tried to consume the poison. The OS-first error logging has to see the attempted poison consumption and might see the poison generation error if the system is configured to expose it. Whichever entity is responsible for logging the poison generation is responsible for sending the data to the server fleet operator's tools that identify failed FRUs.

8.2. Illustration of Framework's OOB Error Handling (Poison Generation to Fault Analysis)

As shown in Figure 13 and enumerated in the steps below, the proposed framework is capable of handling faults from the error detection and poison creation to poison consumption. In doing so, required signaling is supported for page-offlining and OOB hardware-logging allows system visibility post-crash reset or timeout. An additional benefit is that the framework allows for vendor provided error decoders for precise FRU isolation and correlation of errors with ambient system telemetry for better fleet-scale understanding of hardware errors. There is an in-band flow for these errors, and it is described in the next section.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

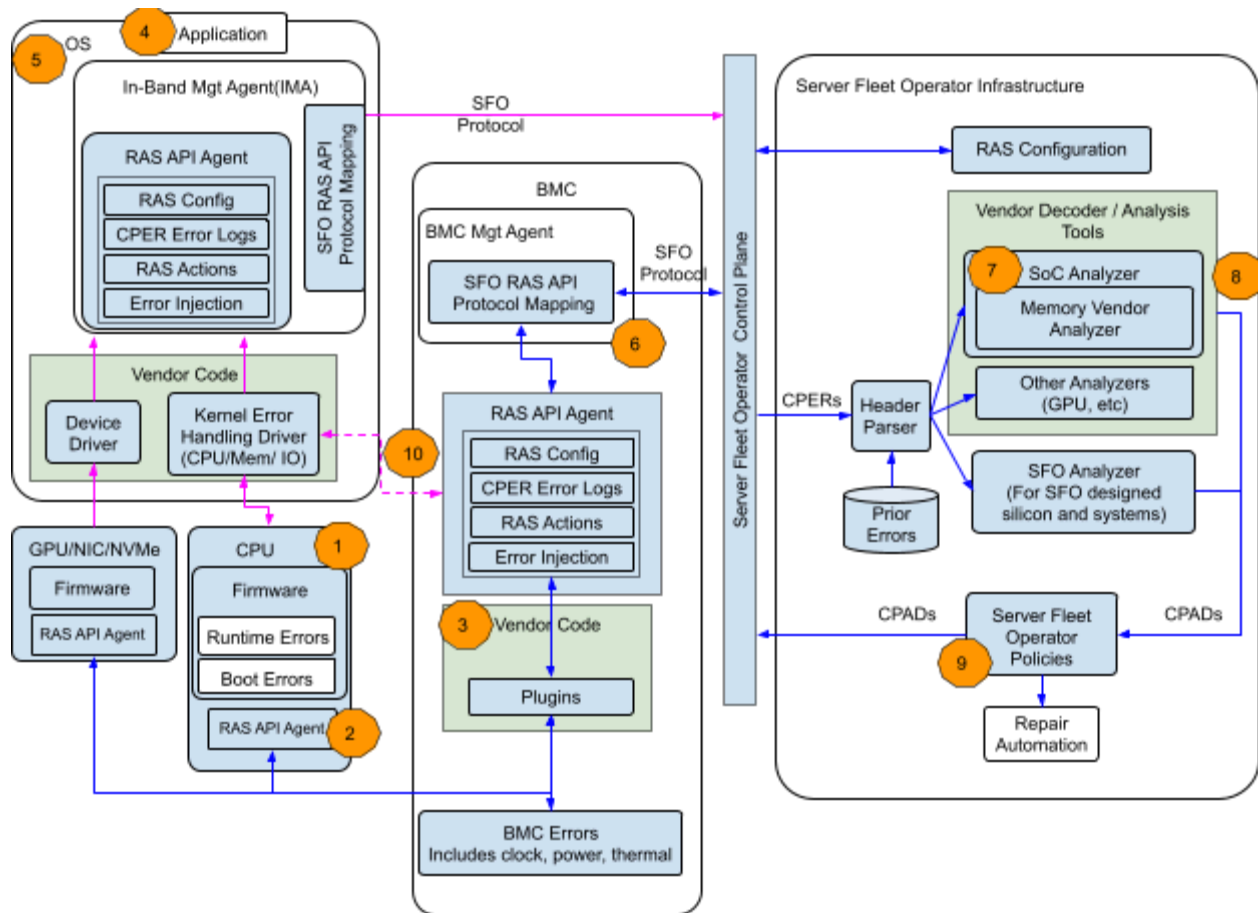


Figure 13: Illustration of Framework's OOB Handling of Poison Creation, Consumption and Fault Analysis

Poison Generation:

1. Memory controller detects and logs Non Fatal, Recoverable error, and poisons the cache line
2. HW RAS API agent's Out of band (OOB) interface sends detailed error information to BMC RAS API agent
3. BMC RAS API Agent forwards the CPER error log to the RAS-API Agent's vendor-specific code

Poison Consumption:

4. An application running on a CPU attempts to consume poisoned data from the cache
5. Before the application can use the poison, the CPU takes an exception into the OS, preventing the application from consuming poisoned data and may be able to recover the system operation. The OS may also offline the bad page.

Fault Analysis and RAS Actions:

6. BMC's RAS API Agent exposes the hardware error information to the vendor analyzer and/or to the Server Fleet Operator's Control Plane through the OMA.
7. Vendor Decoder and Analysis Tools indicate whether CPU or DIMM is the faulty part and suggests self-repair actions if applicable.
8. If the analyzer determines that the memory is safe to use, i.e. not the cause of the poison, it can emit a CPAD that tells the OS that the memory is safe to use.
9. The CPAD is evaluated and Server Fleet Operator Policies may decide whether to allow the CPAD to be sent back to the BMC so that it can be acted upon.
10. The CPAD is sent to the host OS, via the BMC and CPU. If possible, the OS may clear the poison and start using memory again.

8.3. Illustration of Framework's In-Band Error Handling (Poison Creation to Fault Analysis)

As shown in Figure 14 and enumerated in the steps below, the proposed framework is capable of handling faults from the error detection and poison creation to poison consumption. In doing so required signaling is supported for page-offlining and OOB hardware-logging allows system visibility post-crash reset or timeout. An additional benefit is that the framework allows for vendor provided error decoders for precise FRU isolation and correlation of errors with ambient system telemetry for better fleet-scale understanding of hardware errors.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

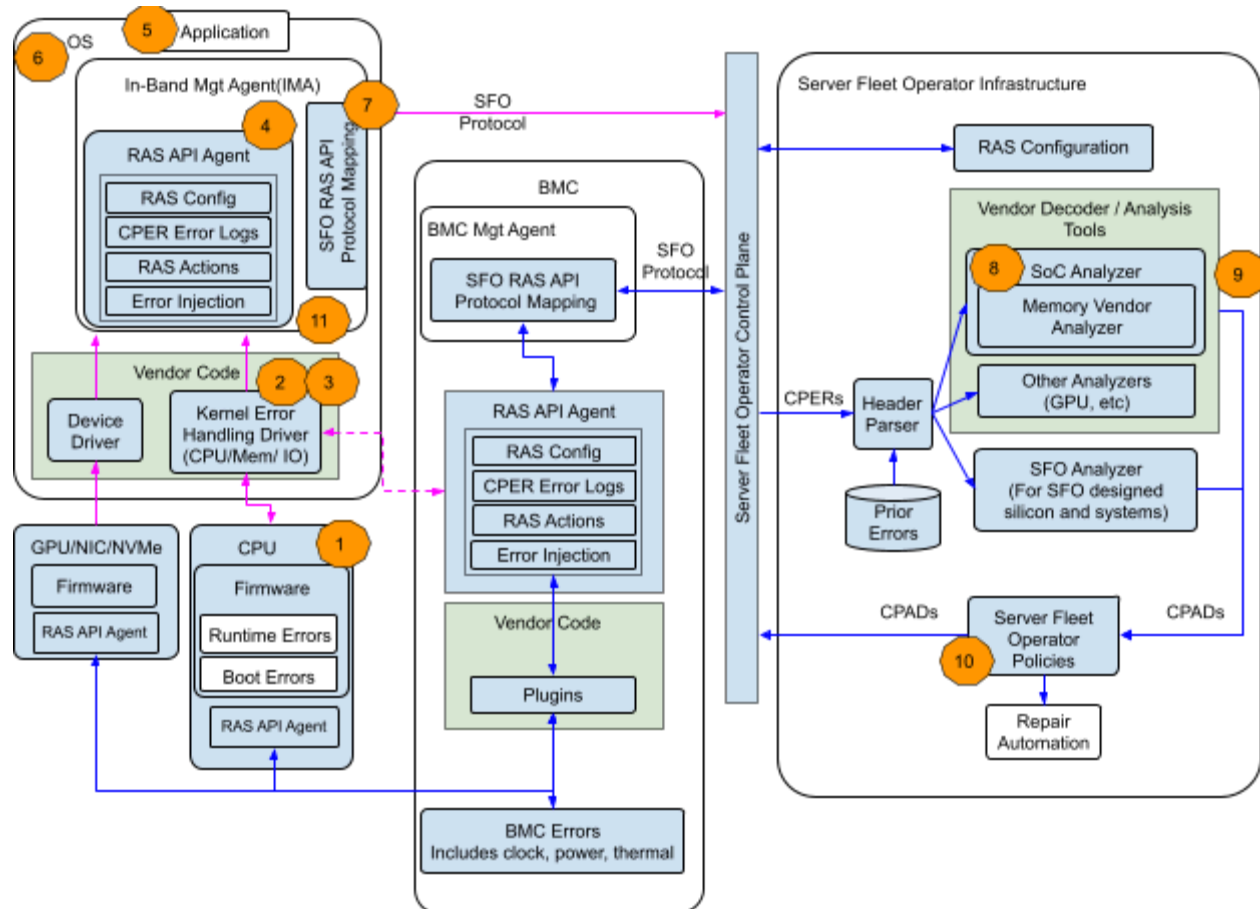


Figure 14: Illustration of Framework's In-Band Handling of Poison Creation, Consumption and Fault Analysis

Poison Generation:

1. Memory controller detects and logs Non Fatal, Recoverable error, and poisons the cache line
2. Kernel Error Handling Driver logs the error
3. Kernel Error Handling Driver can mark the bad page for off-lining
4. Kernel Error Handling Driver passes the error data to RAS-API Agent in the In-Band Management Agent (IMA)

Poison Consumption:

5. An application running on a CPU attempts to consume poisoned data from the cache


 This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

6. Before the application can use the poison, the CPU takes an exception into the OS, preventing the application from consuming poisoned data and may be able to recover the system operation. The OS may also offline the bad page.

Fault Analysis:

7. IMA's RAS API Agent exposes the hardware error information to the vendor analyzer and/or to the Server Fleet Operator's Control Plane
8. Vendor Decoder and Analysis Tools indicate whether CPU or DIMM is the faulty part and suggests self-repair actions if applicable.
9. If the analyzer determines that the memory is safe to use, i.e. not the cause of the poison, it can emit a CPAD that tells the OS that the memory is safe to use.
10. The CPAD is evaluated and Server Fleet Operator Policies may decide whether to allow the CPAD to be sent back to the IMA so that it can be acted upon.
11. The CPAD is sent to the host OS, via the IMA. If possible, the OS may clear the poison and start using memory again.

8.4. PCIe* corrected error reporting example using In-Band method

PCIe* links are expected to be resilient to various types of faults. For example, when a PCIe* link receiver detects a CRC mismatch error, it can be corrected by retrying the transaction and logging the received error as a corrected error. Similarly, there are various types of faults that can be detected and corrected within the PCIe* subsystem. Once such corrected errors are logged by the hardware, reporting of such errors may be achieved via in-band methods or out-of-band methods. The scope of this section is to provide an example of PCIe* corrected error reporting using the in-band method.

PCIe* corrected errors can be detected at various types of PCIe* devices that exist within a system, such as root port, downstream ports, device endpoints, etc. Once an error is corrected and logged, a message is sent to the PCIe* root port and it is the responsibility of the root port to further signal the event either directly to the Operating System or via 'Firmware First method' to [platform firmware](#). Either the Firmware or the Operating System further handles the errors by capturing the error logs and clearing the log registers to re-arm the hardware logic for subsequent errors. In the case of the Linux Operating Systems as an example, error logs are further propagated to the 'sysfs' data structures, dumped on the console (dmesg), and in the Ftrace as described



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

in [Figure 7: PCIe Error flow between HW and SW using in-band method](#). Once the error logs are saved in the Ftrace, daemons such as RASdaemon may further process the logs and present them to the ‘Inband Management Agent IMA’ via ‘RAS API’. And the IMA may interface with the rest of the SFO infrastructure for managing the PCIe* corrected errors at the fleet level such as monitoring the error rate and preemptively initiating PCIe* device remediation flows.

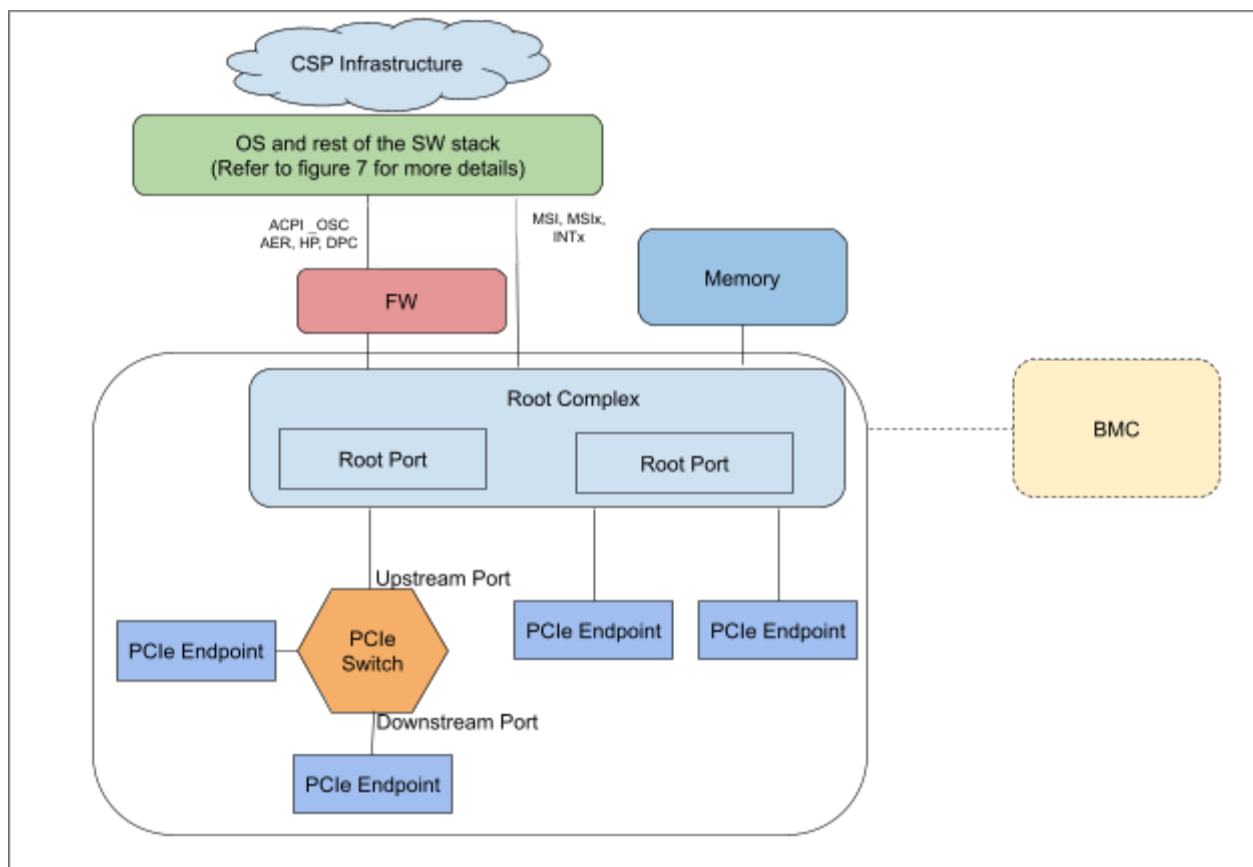


Figure 15: PCIe* corrected error reporting using in-band method

8.5. PCIe* corrected error reporting using OOB

One of the goals for this document is that in band and out of band error handling methods coexist. In Figure 14, the PCIe errors could also be sent to the BMC in addition to the OS. The OS may need to handle PCIe* errors and OOB logging may monitor PCIe* links in order to generate dashboards of systemic behavior across the fleet and



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

to analyze the PCIe* errors in order to trigger RAS actions or repair automation; PCIe* corrected errors shall be reportable both in and out of band. OOB error logging and in-band logging of corrected PCIe* errors shall not be mutually exclusive.

For example, it may be possible for some implementations to have duplicated AER logs such that the OS sees the PCIe* AER registers and such that the OS can clear the AER registers without affecting OOB copies of the registers. Other implementations may have a way to log the AER registers into a queue to be sent out-of-band. The key requirement is that the in-band and OOB logging entities will not clear the error logging registers such that they interfere with each other.

9. Complementary Enablement in the Industry

The hardware fault management system architecture laid out in this requirements document is illustrative of a cloud infrastructure stakeholder to design a vendor and silicon agnostic fault management infrastructure. These requirements in turn rely on industry standard specifications for interoperability. This section outlines new features and potential amendments to Industry standards specifications that may be required to enable the framework proposed in this document.

9.1. [BMC](#)-Based Fault Management Enablement

- Add OOB fault management features to eliminate the need for SMI (x86) or similar methods that interrupt the CPU to implement firmware first logging
- Comprehensive out-of-band error logging to the BMC
- Enablement to make error state persistent, if applicable
- BMC exposes CPER error logs via Redfish (described below)

9.2. Interface between the BMC and the Kernel

- Need a BMC-to-OS interface to pass error information
- Need an OS-to-BMC interface to pass error information

9.3. Interface between the BMC and Analysis/Decoding Tools

- Must be able to support vendor-specific decoding tools for identifying faulty FRUs



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

- Potentially standardize the way these tools report faulty FRUs

9.4. Interface for configuring and triggering RAS features Actions from the BMC

- Define flows where BMC can tell BIOS/UEFI how to configure silicon vendor RAS features at boot time
- RAS actions that do not depend on OS involvement may be triggered directly by the BMC
 - BMC activates features in the HW using an OOB interface

9.5. [Error format](#) for the logs in OS-first method of error signaling

- EDAC is an example of the type of tool that is needed
- Driver format for memory data CE/UC-R
- Driver format for memory command/address parity CE
- Driver format for PCIe* links CE/UC-NF
- Driver format for CXL link CE/UC-NF
- Driver format for device specific CE/UC-NF

9.6. Standardize minimum error data set in Hardware logs

- This workgroup's initiative standardizing "hardware fault logs" associated with various fault domains
- Common Error Record definition
- Any updates to UEFI specification to include new CPER type to standardize the Common Error Record adopted by this framework

9.7. Additions to Redfish Interfaces ([Reference](#))

- Includes Redfish Log, Event and Telemetry services:
 - Any required modifications to Redfish's Log Service to enable:
 - Getting new logs since they were last read
 - Filter based on ID, timestamp, etc
 - Scalability from simple to complex/distributed system constructs
 - Any required modification or additions to Redfish's Event Service to enable:
 - Subscription to new error/RAS event types detected by the BMC
 - Notification of said error/RAS events to the Server Fleet Operator Infrastructure to trigger supported RAS actions



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

- Any required modifications or additions to Redfish's Telemetry Service to enable correlation of platform sensor data with HW error logs (i.e. hermetic logs)
- Any required modifications or new definitions to profiles and message registry
 - Common Error Record
 - Generic event for other errors
 - Specific systems might have other registries
 - CPER class on PLDM

9.8. Hardware Error Reporting Standardization Requirements

The standard for error reporting must satisfy the needs that would enable the faster deployment of heterogeneous hardware and the minimum cost to implement and maintain.

The error record format needs to enable effective error triaging. This is to provide complete and accurate information necessary to fully categorize the error and provide the first level response to the field. This includes identifying the FRU (Field Replaceable Unit) which caused the error.

The error format must also be structured, which simplifies the programming of tools to filter and categorize the errors. The structure for the errors must provide extensibility in the sense of extending the categories and from the perspective of including more data into the message. In some cases, full error root cause may require hardware and software state, and this will create architecture specific unbounded blobs of information that will be required with the error message. Also, the extension of categories will keep the standard alive, by allowing new categories for new technologies or methodologies to be added. Finally, it is important to consider the current footprint of tools and APIs available in the industry for the standard.

9.9. Hardware Error Report Standardization Solutions

The proposed standardization consists of CPER format for the platform error logging with Redfish as the transport protocol.

- CPER based error Records - As CPER based Hardware errors are already standardized as part of the UEFI specification, adopting this solution for out-of-band methods also gets a wider adoption.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

- Redfish schema additions - Allow an individual LogEntry to reference a CPER Record or individual CPER Section as AdditionalData. Support can be easily provided using “CPER” to the list of supported formats for attached diagnostic data
- Redfish “Platform Error” Message Registry – Define a set of messages to cover common hardware error cases associated with CPER data. The message registry provides a human-readable text message along with the programmatic means to identify specific errors without requiring text parsing.

As part of OCP and DMTF collaboration, error reporting standardization implementation details are referenced here:

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

9.10. Error Injection capabilities

- Enable error injection from the BMC
 - Using BMC based tools, e.g., EINJ extensions.
- Improve error injection from the Compute Node
 - Expand EINJ to allow testing of RAS features
 - Injection from the Compute Node is especially important for testing OS-based RAS Features
 - e.g. Poison recovery, mapping out bad pages, etc.
- Having both OOB and in-band error injection is an essential requirement to validate end-to-end error handling flows.
- Leverage EINJ interface between OS-based and BMC-based error injection

10. Conclusion

This proposal outlines a framework that solves problems SFOs have with prior hardware fault management solutions. It provides the tools SFOs need to diagnose hardware failures and minimize their impact, including:

- Standard interfaces for logging errors and driving RAS actions
 - Out-of-Band, from the BMC and Out-Of-Band Management Agent (OMA)
 - In-Band through an In-Band Management Agent (IMA)
- Support for bare metal, application and infrastructure use cases
- A framework for analyzing errors



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

This proposal outlines a direction and proposes changes that affect hardware, management interfaces such as Redfish, BMC software and Server Fleet Operator infrastructure.

11. References

1. ACPI Specification: https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf
2. UEFI Specification: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf
3. PCI Express Specification: <https://pcisig.com/pci-express-6.0-specification>
4. Hardware Fault Manager: <https://github.com/openbmc/docs/blob/master/designs/hw-fault-monitor.md>
5. Redfish Data Model Specification:
https://www.dmtf.org/sites/default/files/standards/documents/DSP0268_2023.1.pdf
6. OCP Fault Management Scenarios: [document](#)
7. [OCP Fault Management Glossary of Terms](#)
8. <https://people.cs.rutgers.edu/~rmartin/teaching/spring03/cs553/readings/avizienis00.pdf>

12. Appendix A - Glossary of Terms

ACPI - Advanced Configuration and Power Interface

ACPI is an open industry specification that defines a standard interface between an operating system and the underlying hardware platform for discovering and configuring hardware components, managing power, monitoring system status and controlling the system.

AER - Advanced Error Reporting

The “Advanced Error Reporting” (AER) nomenclature originates from the PCIe* specification that specifies two primary error-reporting methods: standard and AER. The standard error reporting paradigm is implemented by all PCIe* devices and provides rudimentary error information. AER provides a much richer set of error information and data including error type, location, and transaction specific details.

ASL - ACPI Source Language code run by the ACPI Runtime FW.

BMC Firmware

Firmware running on the BMC.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

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)

CCI - Command Control Interface

A CCI represents a command target that is used to process management and configuration commands that are issued to the component (e.g. Mailbox Registers or MCTP-based).

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 of 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. Modern systems therefore 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.

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 Service Providers.

DPC

Down Stream Port Containment. It is a PCI Express specification based extended capability associated with downstream and root ports.

Decoding

Decoding is the first step in analyzing an error log. The binary error log is parsed and broken into fields to which rules can be applied. A decoded error log is often in a human-readable, but machine parsable format such as



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

JSON. Decoded error logs allow server fleet operators to analyze the errors and use ML tools to look for signals, for humans to consume the logs and to build dashboards for tracking failures in their fleet.

EDAC - Error Detection And Correction

EL3 (Exception Level 3)

EL3 is the highest level of execution state for ARM CPUs. Also called Secure Monitor, it is a privileged mode that can only be entered by firmware. It is used for managing security and other low-level features of the CPU.

Error

An error is incorrect data or execution due to a fault. The following items may occur when there is an error:

Error Detection

Error detection is when hardware indicates the presence of an error.

Error Logging

Error logging is the process of saving information about an error.

Error Signaling

Error signaling refers to the means by which error handlers are notified about errors.

Error Handling

Error handling is the operational actions the system takes in response to an error.

Fault

A fault is an underlying hardware condition that causes a data or execution error.

Fault Management

A system engineering discipline that takes a first-principles approach to managing hardware failures, including understanding how errors occur, designing error logging solutions, analyzing error logs, driving actions to minimize the impact of the error, and determining how to repair systems that have failing hardware.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

FIT (Failures in Time)

The Failures In Time (FIT) rate of a device is the number of failures expected in one billion (10^9) hours of operation.

FRU (Field Replaceable Unit)

A FRU is a part that can be changed in the data center in order to repair a broken system. Components like DIMMs or PCIe* cards are FRUs. A DRAM soldered to a mainboard would not be considered a FRU; in that case, the mainboard would be considered a FRU.

FW-First error logging

Error logging using runtime firmware (e.g SMM). Firmware first error logging allows a server vendor to see all error logs from hardware, even when the vendor does not have a daemon or driver on the operating system. With firmware-first error logging, the firmware can log hardware errors, determine whether it can address the error or whether it needs to expose the error to the operating system. If the firmware can handle the error, it can clear the error and the OS will not see it.

Hardware component

A component designed by a vendor. Examples of hardware components include CPU chips/SoCs, DRAMs and PCIe* devices.

Hermetic error logs

Upon detecting an error, hardware should log all the relevant information and allow ‘error monitoring agent’ to read entire logs prior to updating the logs with any new error event. The error logs should be complete so that they can be analyzed without needing to gather additional data. Hardware should not taint the error logs by overwriting some part of logs before previous logs are read by the ‘error monitoring agent’. This is important where error detection rate could be higher than the rate at which ‘error monitoring agent’ could read the logs, e.g., main memory corrected error logs.

In-Band Management

In-band Management allows OS/software running on the host to collect error logs, clear error logs, and take appropriate RAS actions. In-band error logging may read errors out of architecturally defined registers, may use firmware interfaces such as ACPI or may use drivers. The fault management software running on the host is



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

responsible for communicating information about the health of the system to the server operator's infrastructure.

IMA

In-band Management Agent is a software service running on the host, the main CPU subsystem. The purpose of IMA is to provide an interface between the Server Fleet Operator's management infrastructure and the hardware fault handling solution implemented within the platform..

MCE - Management Check Exception**Management core firmware**

Firmware running on management cores on the processor.

Management infrastructure

Software, typically deployed by the server fleet operator, that manages the fleet operator's hardware. This software manages workloads, monitors the health of the fleet, drives repair actions and manages the flow of data. This software enables server fleet operators to automate the management of their fleet.

Node

A platform designed by a vendor integrating multiple components that run an operating system. These are sometimes also called compute nodes.

Operating system (OS)

An operating system running on the processor. The operating system may run in a virtual machine on a hypervisor or directly on the machine (bare metal).

OS-First error logging

Error logging using the operating system. Hardware detects an error, logs the error, and directly signals to the Operating System for further handling. This usually requires an agent that runs as a daemon in the operating system that handles the error and then communicates with external system management software.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

Out Of Band (OOB) Management

Out-of-band management provides interfaces so that the system may be managed independently of software running on the host processor. With OOB management, the BMC provides interfaces that allow external software to manage the system. OOB error logging means that the BMC can access error logs, with no race conditions, and provide them to external consumers.

Platform firmware

Firmware running on mission-mode cores on the processor. On x86 CPUs, platform firmware is sometimes called the BIOS or UEFI firmware.

Poison

Poison is an example of an error containment signal that can be transmitted over interconnect fabrics. Poison is used when the scope of corrupt data is known (e.g. an uncorrected memory error). The corrupted cache line is marked as being poisoned. When an application attempts to use poisoned data it is interrupted. When poison data is sent over I/O, either the I/O device needs to block the poisoned transaction or the I/O link needs to block it by bringing the link down (e.g. PCIe* Downstream Port Containment). Poison must be tracked in each ECC domain; if poison is generated by memory UCEs, any cache that fetches it must remember that it received corrupted data so that any consumption of the poison is blocked. It is critical that hardware log errors when poison is generated since the server fleet operator will use those errors to determine what FRU needs to be replaced.

Poison Consumption

Poison Consumption implies software attempting to read data that is marked with poison signature. It does not imply that poisoned data is used by the software and instead indicates that software was blocked from using the poisoned data. Refer to the [‘Poison’](#) for more details of the data flow.

PPI - Personally Identifiable Information

RAS

RAS (Reliability Availability Serviceability) solutions mitigate hardware faults. They can be implemented in hardware, software or a combination of both. Some examples of RAS features are memory ECC, DRAM Post Package Repair, marking pages as bad in the OS and poison recovery. Some RAS features like Post Package



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

*Other names and brands may be claimed as the property of others

Repair(PPR) need to be triggered. For example, on a memory controller that supported run-time PPR, software might see a series of corrected errors, all from the same row address. That software could decide that PPR would mitigate the fault causing the corrected row errors and could trigger hardware to do the PPR.

RAS Actions

RAS Actions are actions that a system provides that minimize the impact of the suspected fault. For example, if error analysis tools determine that a subword line has failed in a DRAM, they may suggest that the problem could be resolved with a Post Package Repair. Post Package Repair would be the RAS action.

RAS Configuration

RAS configuration involves various BIOS, OS, and HW configurations to enable RAS features. In general the RAS configurations are done during the boot time, however, some configurations may require the machine to be in operating state.

SFO - Server Fleet Operator

The term SFO 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. Alternative terminologies used as ‘Cloud Service Provider (CSP)’, or hyperscalers.

SMI - System Management Interrupts are high priority unmaskable hardware interrupts which cause the CPU to immediately suspend all other activities

System Firmware

Firmware that boots the SoC and launches the operating system. BIOS, UEFI, and TrustedFirmware-A are examples of system firmware. Firmware-first error logging uses system firmware to log errors.

Viral

Viral is an error containment strategy that is useful when the scope of the corrupted data is not known such as when there is a corrupted packet header or a timeout. When an error causes a loss of coherency, the scope of the corruption is not known. We cannot associate a signal with the affected transactions so we must signal the failure on all transactions. When a block in a chip goes viral, every transaction through the block is marked viral



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

and any block that receives a viral packet also goes viral. As CPUs go viral, they either take an interrupt or halt. I/O is stopped when the I/O controllers go into viral mode. Viral support has typically only been supported on larger systems, but as core counts grow, we need to monitor the potential for SDC and ensure that data corruption is still contained.

13. Appendix B - Implementation Considerations

This section describes possible implementation considerations for the fault management solution described in this document and is non-normative.

13.1. Considerations for preventing race conditions

It is critical to ensure that the in-band and out-of-band error logging solutions do not have race conditions. Some race conditions occur because error logging registers are accessed non-atomically; there can be many registers to log for a particular error and those registers need to be accessed sequentially. Some of the most important race conditions to handle are:

- When error registers get overwritten with a new error while software is in the process of reading the registers
- When in-band and out-of-band error logging solutions might clear errors while they are still being read

13.2. Cases where error registers are overwritten while being read

Most error handling architectures have rules for handling cases where another error occurs before a previous error has been cleared. These overwrite rules prevent corruption when another error occurs while software is in the middle of reading the error logging registers.

For example, some commonly implemented error logging overwrite rules only allow errors to be overwritten when:

- The error logging registers have been cleared by software or a reset
- A higher severity error occurs after a less severe error was captured in error logging registers

With these overwrite rules, software reads a status register that contains the severity of the error first. It then reads and records all of the other registers. After reading all of the registers, it reads the status register and checks to see that the severity of the error has not changed. If it has changed, a more severe error has



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others

overwritten a less severe error and the registers need to be reread. If it has not changed, the error log has been captured and the error log can be cleared.

13.3. Cases where In-Band and Out-Of-Band error logging solutions clear errors while they are still being read

There will be race conditions if both the in-band and out-of-band solution log and clear errors in shared error logging registers. If the OS were faster than the OOB logging entity and they shared the same error logging registers, the OS could clear the error log while the OOB entity was still reading it, causing the OOB entity to see an inconsistent error log.

There are many options for preventing race conditions between in-band and out-of-band logging and may include:

- Firmware First error logging using firmware interrupts
 - Firmware copies error data to the HEST
 - Requires SMM on x86 which is undesirable
 - The security and performance concerns with this solution make it undesirable
- Shadow registers
 - A set for the kernel and a set for the BMC
 - May get out of sync for CEs
 - UCEs will be easier since they are infrequent and we mainly care about the first one
- Embedded error logging microcontroller
 - Microcontroller reads logs from HW and sends them to the BMC
 - Need to ensure OS and microcontroller can both read the data without race conditions
 - Cannot change the programming model of the CPU for legacy OS's

This is not an exhaustive list and there may be other implementations that prevent conflicts between in-band and out-of-band error logging.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

*Other names and brands may be claimed as the property of others