



OCP FAULT MANAGEMENT INFRASTRUCTURE REQUIREMENTS

Version 1.0

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.



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

Executive Summary

This proposal outlines a framework that solves problems [Cloud Service Providers](#)(CSPs) have with prior hardware fault management solutions. These prior solutions do not provide the tools CSPs need to diagnose hardware failures and 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 that the silicon vendor is best placed to provide. Prior approaches to error logging gather insufficient data, have negative impacts on security and performance, and are 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 CSPs from using the features to minimize the impact of failures. These prior solutions have no standard way to report the failure data back to silicon vendors that is needed to identify and mitigate new or low-[FIT](#)-rate failure modes that are hard to observe in their testing.

This Open Compute Requirements document describes a framework for solving these challenges. The silicon vendors provide a RAS API in the Baseboard Management Controller(BMC) and Analysis tools that interpret errors from the RAS API and suggest commands to the RAS API that could help mitigate the issue. It provides a way for silicon vendors to capture the data and provide tools to diagnose failures in their silicon. By having the silicon vendors gather the logs, the format will be standardized and easier to share with them, helping them see trends that require large amounts of error data. The RAS API makes it easy to invoke many RAS features as a simple RedFish call to the BMC. By lowering the barrier to trying RAS features, we believe the industry will move from each company focusing on their favorite RAS features to trying many of them 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.

This proposal focuses on Out of Band management and does not address any requirements a CSP may have for in-band error reporting. This proposal will be adapted to in-band fault management in a future version.



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.



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/).

Table of Contents

Executive Summary	2
License	3
About Open Compute Foundation	4
1. Introduction	7
2. Problem Statement	7
1. Creation of Component Error Logs	7
2. Interpretation of Component Error Logs	8
3. Background	8
Primary data center usage-models	8
Bare-metal	8
Infrastructure	8
Application	8
Hardware Component Error Classification	9
4. Fault Management System Architecture Goals	11
Specific Goals of the Infrastructure	12
5. Proposed System's Error Handling Capabilities	13
Corrected Errors	13
Non-Fatal/Recoverable Errors	13
Non-fatal/Non-recoverable Errors	14
Fatal - Reset Feasible	15
Fatal - VR Faults	15
6. Solution Architecture	16
Node View	18
Error Logging Flow for OS Based Methods	20
7. Error Handling Flow Examples	21
1. Data Poisoning Example	21
2. Illustrative Error Handling Walk-through (Poison Creation to Fault Analysis)	24
8. Complementary Enablement in the Industry	26
BMC-Based Fault Management Enablement	26
Interface between BMC and Kernel	26
Interface between the BMC and Analysis/Decoding Tools	26
Interface for configuring and triggering RAS features from the BMC	26
Error format for the logs in OS-first method of error signaling	26
Standardize minimum error data set in Hardware logs	27
Additions to Redfish Interfaces (Reference)	27
Hardware Error Reporting Standardization Requirements	27

Hardware Error Report Standardization Solutions	28
Error Injection capabilities	28
9. Conclusion	29
10. References	29
Appendices	30
11. Appendix A - Glossary of Terms	30
12. Appendix B - Implementation Considerations	34

1. Introduction

This document defines requirements for a system architecture to monitor, report, and remediate hardware component errors. The first version of this document is focused on the out-of-band, [BMC-based](#) error logging architecture. The goal of this system architecture is to ease the adoption and management of new data center equipment for Server Fleet Operators. Throughout this document, it may be called an 'error handling method'. 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 two challenges in current data centers:

- Creation of standardized hardware component error logs - for all errors experienced by [nodes](#) and addressing Heterogenous error reporting formats
- Analyzing component error logs - to take node level repair and RAS actions

The remainder of this section provides details on these challenges.

1. Creation of Component Error Logs

The approaches employed by existing data center [nodes](#) to creating error logs can be sub-optimal. For example, many nodes implement an operating system-based error logging approach ("[OS-first](#) error logging"). OS-first error logging is a problem for 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. It is often important to correlate error logs gathered by software, such as Machine Check Banks with hardware error logs.

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 performance impact on application software and potential security vulnerabilities. As a result, FW-first error logging is being phased out by most CSPs.

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. Background

Hardware errors are existential within any system deployed in a modern data center. Therefore ‘Error handling methods’ are required in all the systems deployed within modern data centers whether the systems are computation, networking, or storage. Existing methods are developed using a combination of standards based as well as proprietary techniques and are complex in nature. Additionally, the solutions may vary based on the data center usage-models. 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 on primarily three data center usage-models as stated below and illustrated in figure 1:

Primary data center usage-models

Bare-metal

- Cloud Service Provider (server fleet operator) owns hardware and firmware, including [management core firmware](#).
- Customer owns OS and rest of the software stack (services and applications)

Infrastructure

- Server fleet operator owns hardware, firmware, OS, hypervisor and some part of the software stack (services).
- Customer owns the rest of the software stack (e.g., applications).

Application

- Server fleet operator owns hardware, firmware, OS, and software stack (services and applications). Basically the entire infrastructure.



- Customers use the applications and own their own personal goal.

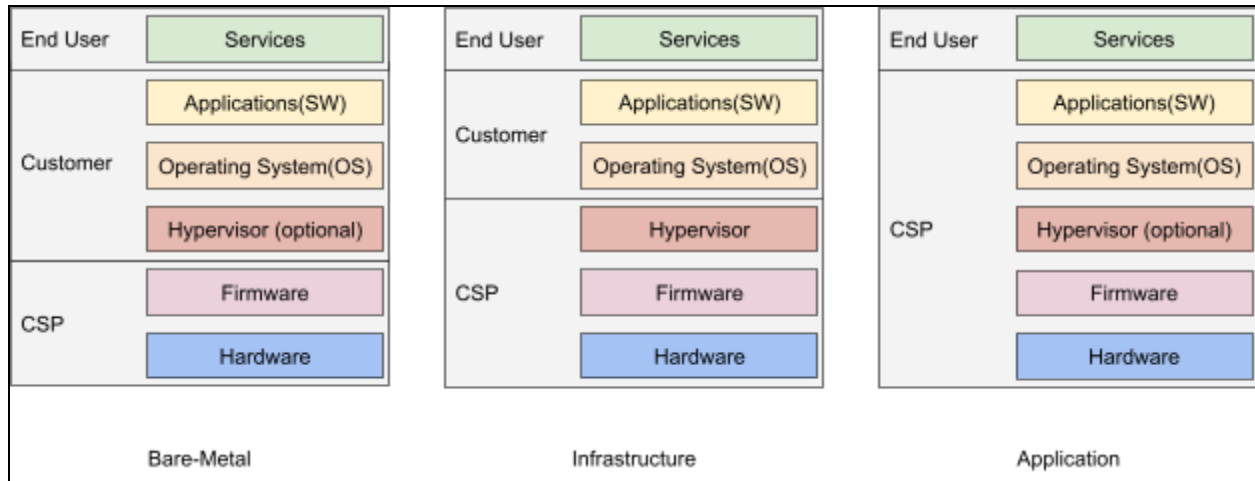


Figure 1: Data Center Primary Usage Models

There are various types of errors feasible within the hardware components used in modern data centers. Next section describes the base-line error classification used in the rest of this document.

Hardware Component Error Classification

This HW error handling solution needs to address various types of errors as listed and briefly described in the below table:

HW Fault Type	Severity	Example Cases	System Crash	Comments
Corrected	Corrected	Memory, PCIe link, UPI link corrected error.	No	1. Platform shall return the control to OS context. 2. OS/BMC utilities shall report the errors. 3. Appropriate error storm handling is required, e.g., revert back to sampling. 4. There must be no race conditions between in-band vs out of band error logging, e.g. The kernel must not be able to

HW Fault Type	Severity	Example Cases	System Crash	Comments
				clear the error before the BMC FW reads the logs or vice-a-versa. 5. CSP Infrastructure should determine the remediation or repair actions.
Non-fatal - recoverable	Non-fatal, Recoverable	Memory/Cache Uncorrected Error, PCIe Link non-fatal (with E/DPC)	No	1. Platform shall capture error logs in a way that makes them visible to the CSP.. 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. CSP Infrastructure should determine the remediation/repair actions.
Non-fatal - reset feasible	Non-fatal, Non-recoverable	PCIe Link non-fatal error (Without E/DPC)	Yes	1. Platform shall capture error logs in a way that makes them visible to the CSP. 2. Platform shall return the control to OS context. 3. Kernel should panic. 4. CSP Infrastructure should determine the remediation/repair actions.
Fatal - reset feasible	Fatal	CPU internal parity error, PCIe Link fatal error (without E/DPC)	Yes	1. Platform shall capture error logs in a way that makes them visible to the CSP.

HW Fault Type	Severity	Example Cases	System Crash	Comments
				2. Platform shall collect error logs and reset. 3. CSP Infrastructure should determine the remediation/repair actions.
Fatal - VR faults	Catastrophic	VR failed, host CPU cannot run code, but BMC is still active.	Yes	1. Platform shall capture error logs in a way that makes them visible to the CSP. 2. CSP Infrastructure should determine the remediation/repair actions.
Fatal - BMC VR faults	Catastrophic	BMC fails, loss of AC or BMC VR faults	Yes	1. BMC cannot capture any logs, hence out of scope for this proposal. The hardware must have capabilities for diagnosing the issue (e.g. Power LED does not light).

4. Fault Management System Architecture Goals

This document aims to address the [pain-points listed in section 2](#) by proposing a Hardware Fault Management architecture. HW components in a datacenter node use Out Of Band (OOB) error logging to the Base-board Management Controller (BMC). This document refers to this as “[BMC-based](#) error logging”, which enables CSPs to log and handle errors in ‘Bare-metal’ usage-models, for example. 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 CSP’s management infrastructure, and allows component vendors to provide analyzer tools that interpret error logs in the CSP’s [management infrastructure](#).

Even though the solution presented in this version of the requirements document focuses on OOB, BMC-based fault management, the OCP Fault Management Team acknowledges that some server fleet operators may want to use In-Band fault management solutions for usage-models such as ‘infrastructure’ or ‘Application’. Compatible In-Band fault management solutions will be addressed in a future revision of the requirements document.

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 CSPs 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.

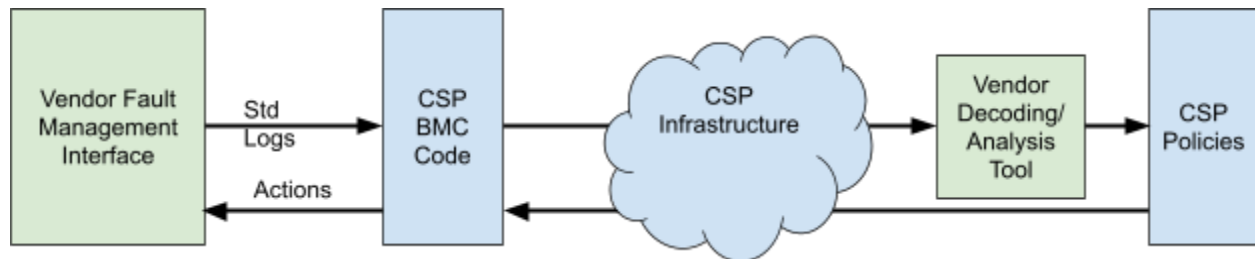


Figure 2: High Level Concept

The above figure shows a high-level view of the system architecture for BMC-based error logging. The green boxes represent component vendor deliverables: an OOB error logging interface running in BMC firmware; plus an analysis tool to run in the management infrastructure. The blue boxes represent CSP infrastructure requirements: BMC code and datacenter infrastructure to transmit error logs and service actions from the node to the analysis tools; and a policy engine to choose when and how to apply service actions. The interfaces between all of the boxes are standardized to allow a CSP to seamlessly deploy components from multiple vendors, and to allow a component vendor to develop a single infrastructure that is used by multiple CSPs.

Specific Goals of the Infrastructure

- Standardize error monitoring and reporting between all hardware components and management infrastructure (Error Monitoring Agent);
- Standardize error event decoding and analysis in the management infrastructure (Vendor [Decoding / Analysis Tool](#));
- Standardize interfaces and mechanisms to drive actions from management infrastructure to hardware components via Redfish interfaces provided by the BMC based on the CSP's policy choices and/or end user's Service Level Objective (SLO);
- Standardize interfaces for a BMC or hardware component to communicate error actions to an [operating system](#);
- Allows CSPs to share data that enables silicon 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

This proposal does not address any requirements a CSP may have for, or mechanisms for a hardware component to report to, an in-band error reporting agent (i.e., OS-First or FW-First error reporting). The proposal also does not address component hardware requirements such as error containment. Those requirements are beyond the scope of this document.

5. Proposed System's Error Handling Capabilities

The system architecture described above can apply to all component errors that occur in all primary cloud usage-models. BMC-based error logging is the only error handling scheme that is feasible in all scenarios and thus is the only solution that can be common across all scenarios. While FW-First error handling is feasible in many scenarios, it is not recommended by CSPs due to the security and performance concerns described earlier. The tables below summarize this. In the tables, a scheme is described as “Not Feasible” if there are any scenarios where it does not function.

Corrected Errors

Usage	FW-first	OS-first	BMC-based
Bare-metal	Feasible	Not Feasible Because the OS is not owned by the operator	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

Non-Fatal/Recoverable Errors

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 BMC will need a race-free way to get the error logs without interference with the OS/Platform.
Infrastructure	Feasible	Feasible	Feasible
Application	Feasible	Feasible	Feasible

Non-fatal/Non-recoverable Errors

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 Since the crash is not directly visible to any end user. Assuming that the OS and IO are healthy enough to write the logs.	Feasible
-------------	----------	--	----------

Fatal - Reset Feasible

Usage	FW-first	OS-first	BMC-based
Bare-metal	Feasible Firmware can usually log the fatal error after a warm reset if the CPU is able to reset and run code and if the necessary data is preserved in sticky registers that survive warm reset.	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 and forward them to the operator's management infrastructure
Infrastructure	Feasible Firmware can usually log the fatal error after a warm reset if the CPU is able to reset and run code and if the necessary data is preserved in sticky registers that survive warm reset.	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 and forward them to the operator's management infrastructure
Application	Feasible Firmware can usually log the fatal error after a warm reset if the CPU is able to reset and run code and if the necessary data is preserved in sticky	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	Feasible Since the BMC can still run, collect the errors and forward them to the operator's management infrastructure

	registers that survive warm reset.	the OS to pick up on the next boot.	
--	------------------------------------	-------------------------------------	--

Fatal - VR Faults

In this case, the host CPUs are unable to run, but the BMC still has housekeeping/aux power.

Usage	FW-first	OS-first	BMC-based
Bare-metal	Not Feasible Since without power, code cannot run on the node	Not Feasible Since without power, code cannot run on the node	Feasible Since the BMC can run and log the error on housekeeping power. Assumes BMC can detect the VR failure.
Infrastructure	Not Feasible Since without power, code cannot run on the node.	Not Feasible Since without power, code cannot run on the node	Feasible Since the BMC can run and log the error on housekeeping power
Application	Not Feasible Since without power, code cannot run on the node	Not Feasible Since without power, code cannot run on the node	Feasible Since the BMC can run and log the error on housekeeping power

In the case of a full power loss, the platform will 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, people servicing the machine 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.

6. Solution Architecture

In the prior section we demonstrated the need for BMC and OS interaction for the successful recovery from error conditions. We would now like to propose an architecture that meets those needs. Figure 3 shows an example of how the vision can be met using OpenBMC and RedFish interfaces to the Cloud Provider 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. Cloud providers might also implement extensions to the infrastructure that are not upstreamed such as the Cloud Provider Policies that determine how the CSP reacts to the output of all of the analyzers.

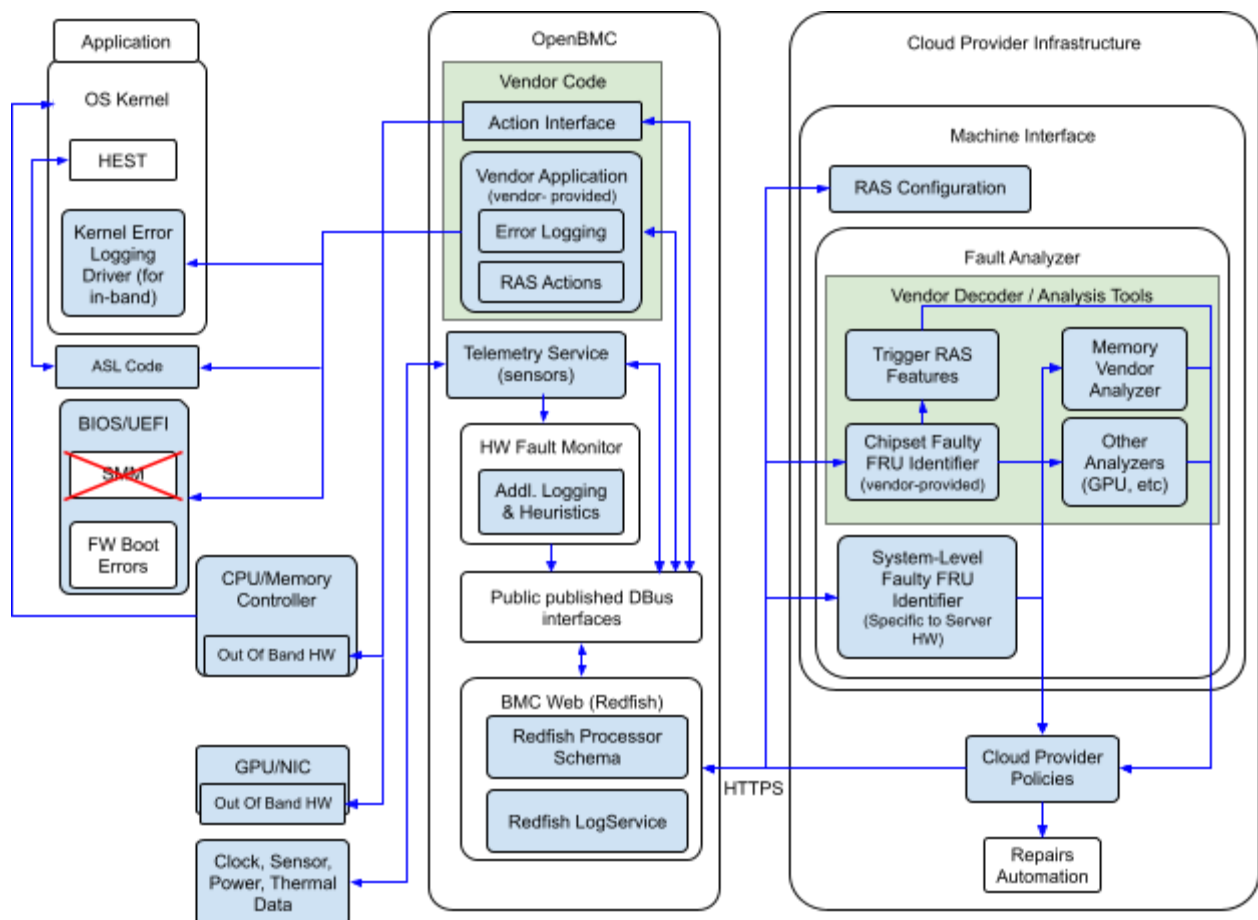


Figure 3: Proposed Fault Management Framework

The figure shows the following:



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

- **Kernel (Operating System):** 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. Improvements to In-Band error logging will be covered in a future contribution for the OCP HW Fault Management team.
- **Out-of-Band Hardware:** 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)
 - (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.
- **Vendor Code in OpenBMC:** Each component vendor provides code that can be integrated into OpenBMC. This code detects and retrieves the error state from the OOB Hardware and packages the state into logs that follow a standard format, such as Common Platform Error Record (CPER). These logs must be parsable by tools deployed in the Cloud Provider Infrastructure. This code also receives RAS Feature Triggers from Dbus/Redfish and communicates them to the OOB Hardware.
- **DBus and BMC Web (Redfish).** For the purposes of this document, the primary goals of the DBus and Redfish interfaces in the BMC are to (a) take the error logs from the Vendor Code and pass them to the Cloud Provider Infrastructure; and (b) receive RAS actions from the Cloud Provider Infrastructure and pass them to the Vendor Code.
- **Vendor Code in Fault Analyzer Cloud Provider Infrastructure.** This code receives the error logs from the BMC, decodes them, analyzes them, and provides diagnostic information with suggested service actions to the Cloud Provider Infrastructure, which includes initiating RAS actions “[Trigger RAS Features](#)” block). Based on Cloud Provider policy, these actions can be sent back to the node to be applied via the BMC.

Node View

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](#).

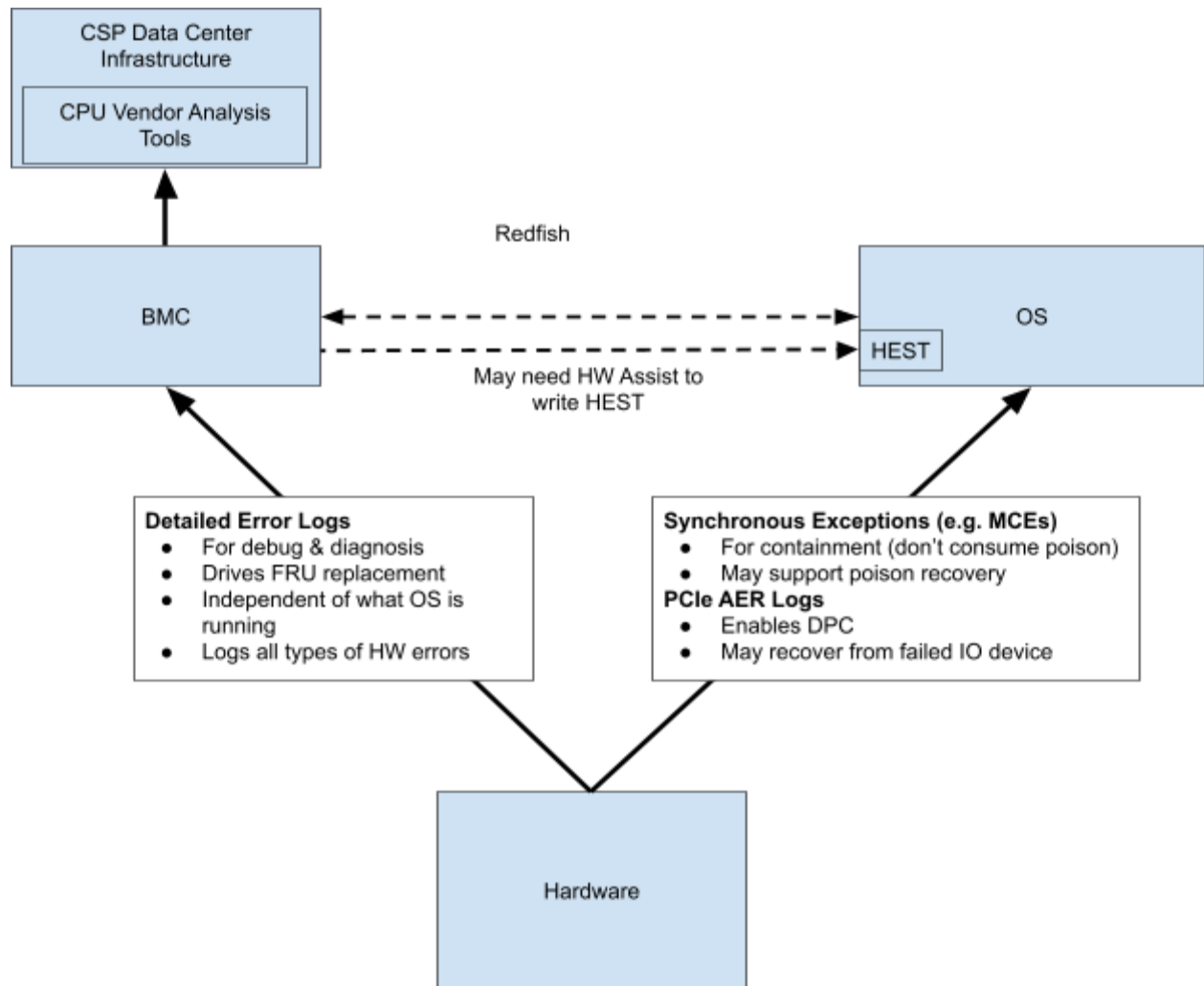


Figure 4: BMC-based error logging and OS error visibility

The OS needs to be synchronously interrupted when it and/or 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 the exposure to the system of an error. 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 SMM. 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.

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 5 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 subject to it than a BMC which can log 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](#).

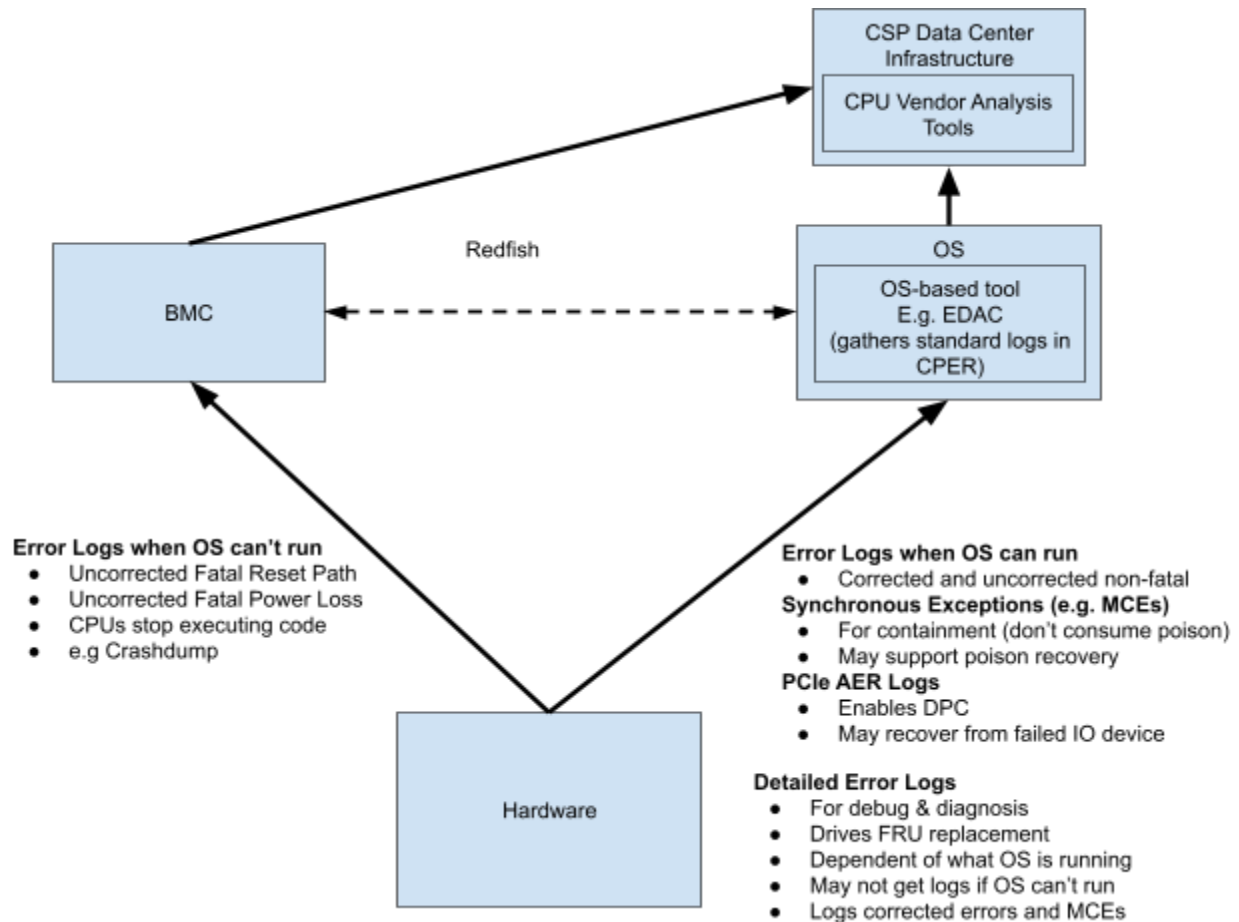


Figure 5: Integrating BMC logging and In-band logging

7. Error Handling Flow Examples

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 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 system must find the failing component that created the poison and log it 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.

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

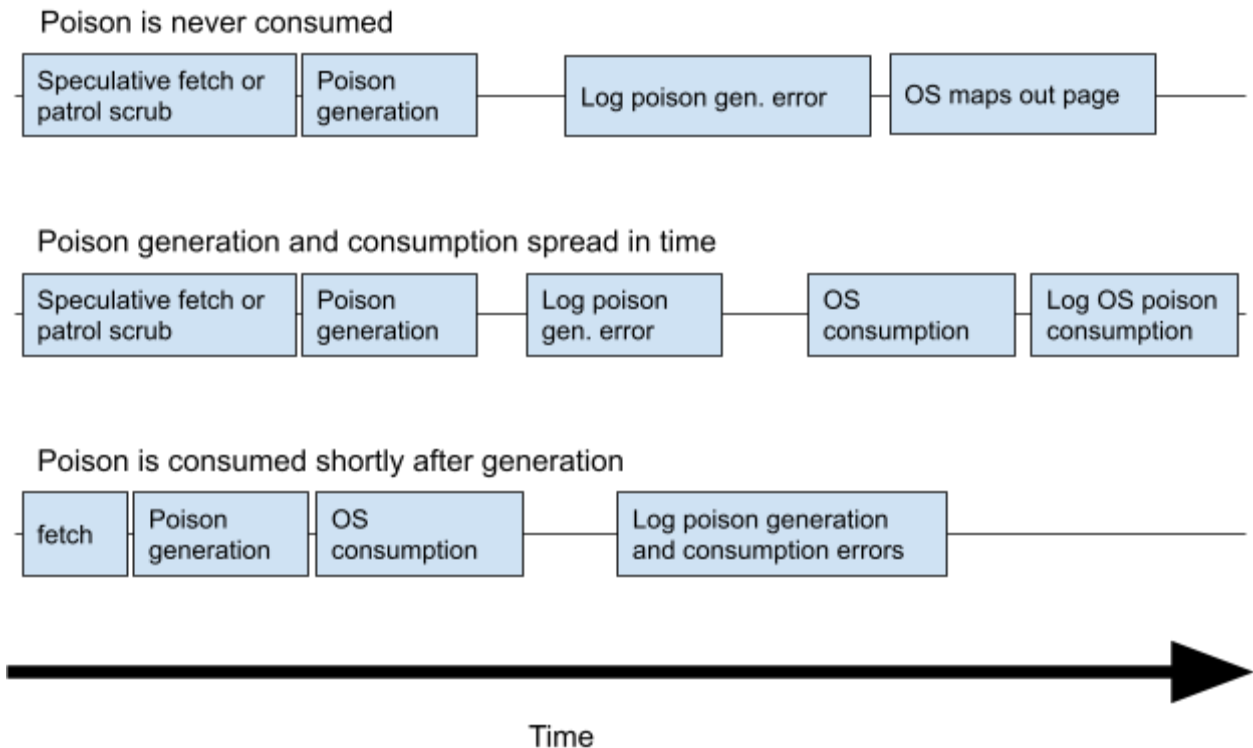


Figure 6: Poison Handling Sequence

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 poison before it 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.

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.

2. Illustrative Error Handling Walk-through (Poison Creation to Fault Analysis)

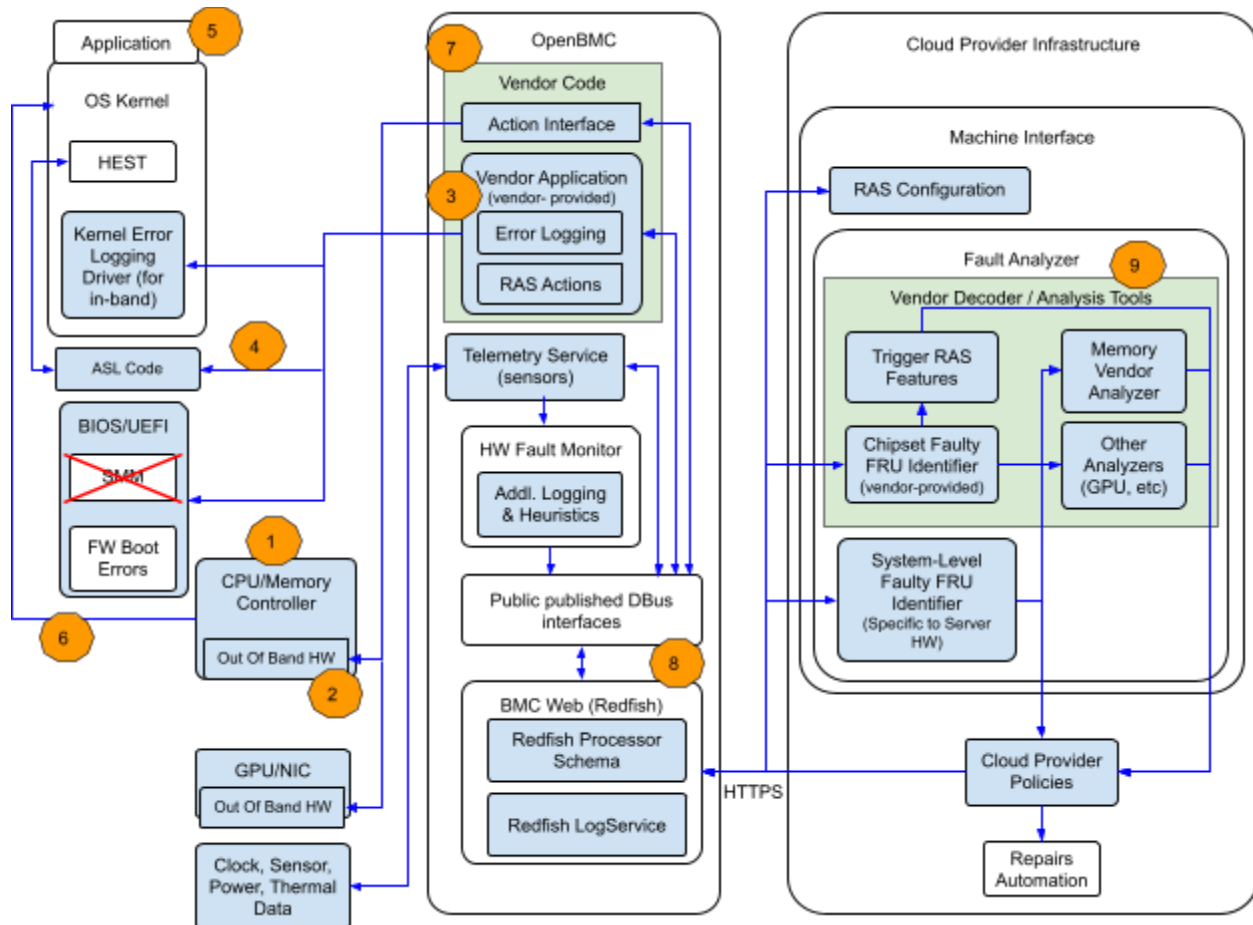


Figure 7: Illustration of Framework's Handling of Poison Creation, Consumption and Fault Analysis

As shown in Figure 7 and enumerated steps described 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.

Poisoned Page-Off-lining:

1. Memory controller detects and logs UC error, poisons cache line
2. Out of band (OOB) HW sends detailed error information to BMC
3. BMC vendor-specific code ensures that it has collected enough data to identify faulty FRU
4. BMC can write HEST (via ASL code) to warn OS of bad page for off-lining

Poison Consumption:

5. An application running on a CPU attempts to consume poisoned data from the cache
6. OS gets a Machine Check Exception due to poison consumption and may be able to recover

Fault Analysis:

7. Out of band (OOB) HW logs machine check information and exposes it to the BMC
8. Out of band HW and Memory Vendor FRU analysis code(s) clearly indicate whether CPU or DIMM is the faulty part
9. BMC correlates error logs with platform telemetry logs (also exposed using RedFish)
10. CSP Infrastructure analyzers perform fleet level failure analysis and CSP policy-based RAS optimizations

8. Complementary Enablement in the Industry

The hardware fault management system architecture laid out in this requirements document is prescriptive for 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.

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)

Interface between BMC and Kernel

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

Interface between the BMC and Analysis/Decoding Tools

- Must be able to support Vendor-specific Decoding Tools for identifying faulty FRUs
- Potentially standardize the way these tools report faulty FRUs

Interface for configuring and triggering RAS features from the BMC

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

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

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

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 Cloud Provider Infrastructure to trigger supported RAS actions
 - 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

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 triaging errors. 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.

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.
- 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 referenced here:

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

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

9. Conclusion

This proposal outlines a framework that solves problems [Cloud Service Providers](#)(CSPs) have with prior hardware fault management solutions. It provides tools CSPs 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
- Support for bare metal, application and infrastructure use cases
- A framework for analyzing errors

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

10. 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

Appendices

11. Appendix A - Glossary of Terms

BMC Firmware

Firmware running on the BMC.

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)

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 providers.

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 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.



EL3 (Exception Level 3)

EL3 is the highest level of execution state for ARM CPUs, called Secure Monitor. It is a privileged mode that can only be entered by firmware and has some similarity to System Management Mode (SMM) on x86. It is used for managing security and other low-level features of the CPU.

Error Logging

Error logging is used very broadly to refer to collecting error data about a failure. Error logging is the process of latching information about an error in the chipset, collecting the error data in software and sending the data to tools where it can be analyzed. When software reads error logs, it often generates a binary structure such as a Common Platform Error Record(CPER) log that can be passed to other, higher level software for analysis.

Error Handling

Error handling is where software logs errors, analyzes them and determines how to minimize their impact. Error handling software can take actions like driving RAS features, telling the operating system to stop using resources and sending messages to get the server repaired.

Fault

A fault is something that causes hardware to malfunction. If the hardware detects the fault, it can log errors that aid in diagnosing the fault and servicing the hardware.

FIT(Failures in Time)

The Failures In Time (FIT) rate of a device is the number of failures expected in one billion (10^9) device-hours of operation. (E.g. 1000 devices for 1 million hours, or 1 million devices for 1000 hours each, or some other combination.)

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.

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.

Platform firmware

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

Operating system (OS)

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

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. This usually requires an agent that runs as a daemon in the operating system that communicates with external system management software.

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.

RAS Features

RAS features 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 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.

Reporting

Signaling



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

Error signaling refers to the means by which software is notified that an error has occurred. Machine Check Exceptions(MCEs) or Corrected Machine Check Exceptions(CMCEs) are ways the hardware can let software running in the compute node know that an error has occurred. The BIOS on x86/x64 machines can be notified of errors via System Management Interrupts (though that has security and performance concerns). The CPU can send the BMC interrupts when an error occurs in order to trigger Out-Of-Band error logging.

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 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.

12. Appendix B - Implementation Considerations

This section dives into some implementation considerations any BMC-based error logging solution would need to support.

Shared Error Logging Registers with Kernel and BMC

The error logging registers are read non-atomically; there are many registers we need to log for a particular error. Error logging registers frequently do not log the next error until the previous error log in them has been cleared. These overwritten rules are fundamental and are essential to getting a complete error log. There will be race conditions if something in both the kernel and the BMC are logging and clearing errors. They will get incomplete logs if the log is cleared while they are reading it and will get an incomplete picture of the errors that are occurring.

There are several options for enabling sharing:

- Firmware First error logging using firmware interrupts
 - Requires SMM on x86



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

- Firmware copies error data to the HEST
 - 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