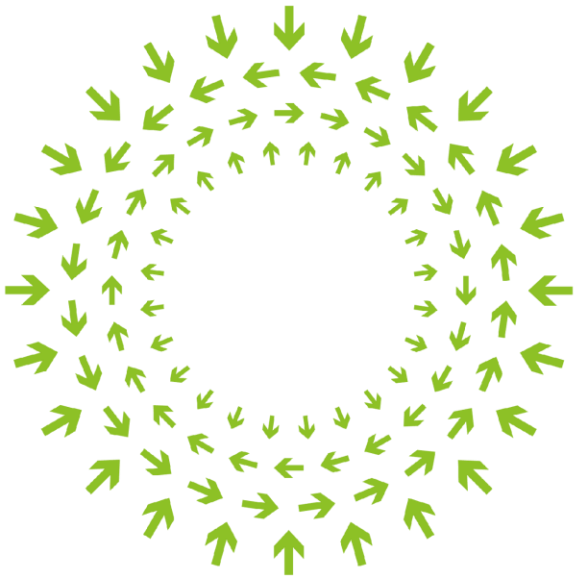


OPEN Compute Project

HYPERSCALE CPU DEBUG AND RAS REQUIREMENTS

Specification version 0.7

Effective 9/15/2025

Authors:

[Rama Bhimanadhuni, Carlos Vallin, Alex Akesson, Rajat Madhusudan **Microsoft**]

[Jicksen Joy, Sean Malley **Google**]

[Vilas Sridharan, Justin King, Nitish Paliwal **AMD**]

[Dean Liberty, John Horley, Bhattu Bhattacharya **Arm**]

[Kapil Dave, Anil Agrawal, Kiran Malwankar **Meta**]

[Jonathan Bentley, Allison Goodman, John Leung, Shubhada Pugaonkar **Intel**]

Contact: hyperscale-cpu@ocproject.net

Table of Contents

Table of Contents 2

Table of Figures 4

Executive Summary 5

License 7

About Open Compute Foundation 8

1 Introduction 9

2 Scope 9

3 Definitions 10

4 Problem Statement and Goals 12

4.1 Problem Statement 12

4.2 Goals 15

5 Standard Debug Workflow 16

5.1 Overview 16

5.2 Configuration Phase 19

5.3 Collection Phase 21

6 Crashdump Requirements 24

7 Debug Dump Requirements 25

7.1 General Requirements 25

7.2 Extended RegDump Requirements 27

7.3 Trace Dump Requirements 28

7.4 State Dump Requirements 28

8 Diagnostics 30

8.1 Diagnostics Standardization Overview 30

8.2 Scope 32

8.3 Diagnostics Environment Requirements....32
8.4 Diagnostics Tools Package Standardization....33
8.5 Diagnostics Recipe Requirements....34
8.6 Diagnostics Compliance Tool....36
9 Future Work....37
9.1 Standardized interfaces for feature configuration and control....37
9.2 Security Requirements....38
9.3 Debug Dump Data Sensitivity....38
9.4 On-demand dump triggering....38
9.5 Diagnostics Tools....39
9.6 Bi-Directional Data Sharing....39
9.7 Additional Dependencies....39
9.8 Reliability, Availability, Serviceability (RAS)....39
9.9 Compliance tools....40
10 References....41
11 OCP Tenets....42
12 Appendices....43
12.1 Appendix A - Glossary of Terms....43
12.2 Appendix B – Diagnostics Compliance Diagrams....46

Table of Figures

Figure 1: Diagram reflecting relationships between Register Dumps, Debug Dumps, and Diagnostics. 11
Figure 2: Crashdump Flow 17
Figure 3: Debug Dump Configuration Flow 20
Figure 4: Standardized Debug Workflow 23
Figure 5: Pain-points and desired solutions for end-to-end Diagnostics. 30
Figure 6: Diagnostics Standardization Components 37
Figure 7: Diagnostics Compliance Components 46
Figure 8: Diagnostics Recipe Standardization Components 46
Figure 9: Diagnostics Recipe Step Types 47
Figure 10: Diagnostics Result Code Components 47
Figure 11: Sample Illustration of the Diagnostic Compliance Tool Architecture 48

Executive Summary

This specification standardizes debug capabilities for CPUs in hyperscale environments, addressing the growing complexity in data centers with multi-supplier CPU fleets. It outlines a standard set of debug feature requirements and production workflows to enhance time-to-market, operational efficiency, and interoperability. By harmonizing these processes, the specification aims to streamline operations across diverse hardware ecosystems, ensuring a more predictable and reliable debugging experience.

The specification introduces a layered debug architecture centered on three foundational components:

- **Crashdumps**: Establish a uniform approach to capturing and analyzing system crashes, enabling faster root cause identification and resolution.
- **Trace logs**: Provide a standardized format for recording event sequences and system activities, offering deeper insights into system behavior and interactions.
- **State dumps**: Capture snapshots of the system's current state, facilitating detailed analysis of operational anomalies.

Together, these elements support a comprehensive debug workflow that includes structured data collection, secure transmission, unified analysis, and actionable feedback. This approach ensures that hyperscale deployments can maintain robust performance and resilience under varied operational pressures.

In addition to debug standardization, this specification introduces a framework for diagnostics standardization across manufacturing, development, and production environments. It defines requirements for diagnostic packages, recipes, and compliance tooling—enabling suppliers to deliver containerized, interoperable solutions that hyperscalers can deploy at scale. By aligning with the OCP Test & Validation (OCPTV) standards, the specification ensures diagnostic outputs are structured, actionable, and automation-friendly, reducing onboarding friction and improving fault isolation across CPU fleets.

The specification also emphasizes collaborative innovation under the guidance of the Open Compute Project (OCP). By leveraging open-source principles, it fosters an ecosystem of shared knowledge and rapid technological advancements. This approach aligns with OCP's mission to accelerate progress in hardware design, addressing the expanding demands of compute infrastructure effectively and sustainably.

While this version (v0.7) focuses on CPU debug and diagnostics standardization, future revisions will expand to include:

- Standardized APIs for debug feature configuration and control Enhanced security frameworks for debug data protection
- Broader diagnostics tooling and compliance coverage
- Extended Reliability, Availability, and Serviceability (RAS) requirements for fault prediction, recovery, and telemetry-driven analysis

By harmonizing debug and diagnostics across architectures and vendors, this specification lays the foundation for resilient, efficient, and scalable compute infrastructure in hyperscale data centers.

License

Open Web Foundation (OWF) CLA

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

AMD, Arm, Google, Intel, Meta, Microsoft

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

You can review the applicable Specification License(s) referenced above by the contributors to this Specification on the OCP website at https://www.opencompute.org/contributions/templates-agreements.

For actual executed copies of either agreement, please contact OCP directly.

Notes:

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

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY

CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

About Open Compute Foundation

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

1 Introduction

This specification standardizes Debug capabilities for CPUs in a hyperscale environment. With the growth of multi-supplier CPU fleets in data centers, ensuring a standard set of Debug features requirements and production workflows will improve time-to-market, operational efficiency, and general interoperability.

For the remainder of this specification, the terms “processor”, “CPU”, and “device” will be used interchangeably. In all cases, the term refers to server processor products shipped by the device supplier, which may include hardware as well as the firmware and software stacks (e.g., device drivers) used to operate the device. Additionally, the terms “vendor” and “supplier” will also be used interchangeably as both accurately represent their function in the supply chain.

2 Scope

This document aims to address the pain-points listed in Section 4 Problem Statement and Goals by proposing standardization for CPU crashdumps, debug dumps, and a debug workflow for collecting them in production. Debug dumps include extended register-based dumps, state dumps, and trace dumps.

This specification also aims to set reliability, availability, serviceability (RAS) error reporting requirements in addition to quantitative requirements specific to CPUs. This creates industry alignment in the adoption and metrics for RAS in CPUs. More details will be developed in future revisions of this document.

3 Definitions

Table 1: Definitions

Term	Definition
Crashes	System state where workloads executing in CPUs cannot make further progress, typically because of an uncorrected/fatal error.
Register based dumps	Snapshot of CPU named register structures (versus lower-level logic state) at a specific moment.
Crashdumps	Any register-based dump collected while the CPU is in a crashed state, primarily for hardware fault management in the data center. Crashdumps represent error state from fatal errors.
Debug dumps	Any debug data collecting methods used primarily for root-cause analysis. These include extended register-based dumps, state dumps, and trace dumps. Unlike crashdumps, debug dumps are used for specific failure signatures and may not be collected on every failure.
Extended Register dumps	Any static register-based dump collected at the time of failure primarily for debug and root-cause analysis. Unlike crashdumps, extended register dumps capture a broader set of register data that may not be exposed to crashdumps and are focused on a specific failure signature.
State dump	Methodology to capture and retrieve flip-flop and/or array/memory binary state for debug purposes, typically by leveraging standards-based test access mechanisms.
Trace dumps	Time-series data that is collected on a regular basis while the CPU is running in production mode, even when no problem has been detected. Trace data may be organized by time into a Trace Log.
Sensitive dumps	Any debug dump whose data, when taken individually or in aggregate, may reveal sensitive information about the operation or end-user of the CPU. Sensitive information may include personally identifiable information (PII), personal health information (PHI), intellectual property, workload computational data, and cryptographic keys, as well as sequences of events that may form a side channel revealing independently sensitive information. Instruction and physical bus traces are examples of sensitive Trace Logs.
Non-sensitive dumps	Any debug dump whose data is not believed, when taken individually or in aggregate, to reveal sensitive information about the workload or end user of the CPU.

Term	Definition
Diagnostic	A structured methodology comprising tools, scripts, and workflows designed to assess the health, integrity, and performance of systems. Unlike debug tools, which are primarily used for root-cause analysis after a failure occurs, diagnostics are employed to screen for issues, verify correct system behavior, and support ongoing reliability. Diagnostics play a critical role across manufacturing, development, and production environments, where they are used to detect faults, validate system functionality, and inform repair or maintenance actions.

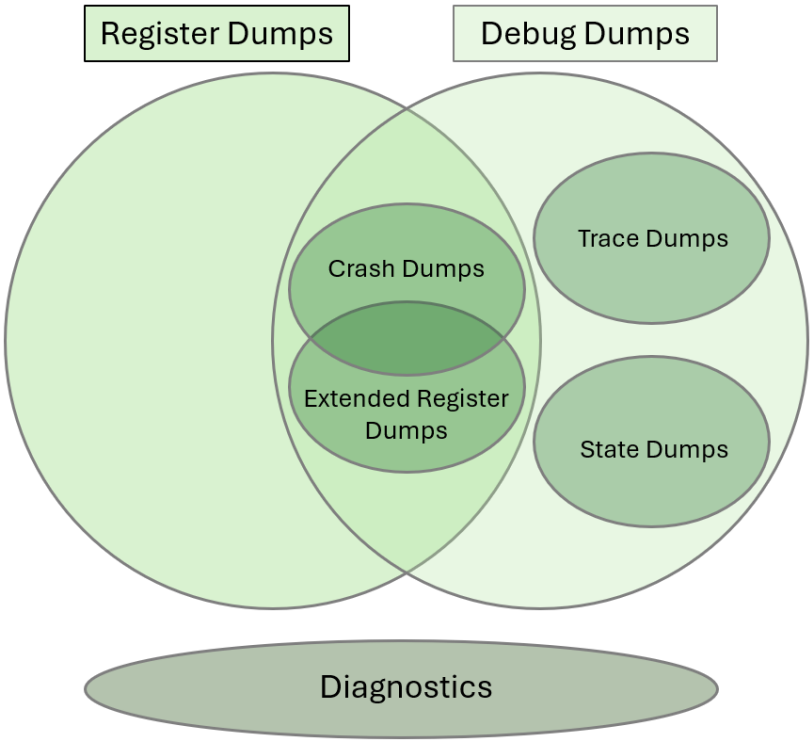


Figure 1: Diagram reflecting relationships between Register Dumps, Debug Dumps, and Diagnostics.

4 Problem Statement and Goals

4.1 Problem Statement

Hyperscalers face challenges when integrating CPUs, in part due to their complexity and the heterogeneous nature of their implementation. One goal of this specification is to provide commonality in certain debug features and interfaces to reduce the burden of adopting new products. This includes register-based dumps, state dumps, and trace logs.

Debugging issues in the fleet becomes increasingly challenging with the introduction of new silicon architectures and components, as they bring new failure modes that may only reproduce while in the presence of production workloads, which are often not visible to the hyperscaler or CPU supplier.

Furthermore, this specification sets reliability, availability, serviceability (RAS) error reporting requirements in addition to quantitative requirements specific to CPUs. This creates industry alignment in the adoption and metrics for RAS in CPUs.

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

Table 2: Pain Points

Pain point	**Description**	**Desired Solution**
Debug feature security access controls and sensitivity of data collected are inadequate for use in production	Access controls for debug features do not meet hyperscaler security standards for deployment and data sensitivity in production, leading to their disablement or complex enablement procedures that are significantly different from their use in development phases (i.e. in lab). Existing debug features may collect end-	Debug feature access controls that adhere to latest industry security standards, offering granular access based on the least privilege principle, while maintaining consistency across development and production phases. Data exposure for debug collection features that are clearly described by

	user sensitive data that should not be visible to either the hyperscaler or CPU supplier. This data may include Personally Identifiable Information (PII), Personal Health Information (PHI), payment card information, or Intellectual Property (IP). Additionally, exposure of this data may be regulated by tenant contracts, hyperscaler policies, or government regulations.	the CPU vendor, including an analysis of the feature's ability to expose end-user data. This description should be machine-readable so that it may be fed into the hyperscaler's policy engines for analysis prior to enabling the feature.
Challenges debugging workload-specific problems at scale often lead to Root-Cause Analysis (RCA) Service Level Agreements (SLA) violations	Certain hardware failure signatures only occur at cloud-scale running production workloads, making them difficult to reproduce in debug environments and delaying root cause analysis beyond SLA times. While existing telemetry solutions help diagnose some failures, they often fall short in identifying RCA for all failure signatures.	Standardized debug interfaces and protocols that streamline enablement, configuration and retrieval of CPU debug data in production without compromising security or performance. To ensure security, access to sensitive CPU data, including silicon IP, customer workloads, and CSP intellectual property— must be strictly controlled.
Debug features and tools do not scale well from development to large-scale production environments.	Traditional in-lab debugging relies on manual processes, physical access, custom firmware updates, and debug tool applications that are unsuitable for production environments. Supplier-provided debug tool binaries and applications for the development phase are impractical for use in production. Examples include security restrictions preventing deployment of applications not signed by the hyperscaler, incompatibility with additional OS versions used in production, integration challenges into modern microservice architectures,	Use of remote, automation-friendly debug workflows that securely invoke debug features and collect debug data without human intervention. Adoption of frameworks that enable suppliers to provide debug tool source code under proper software licensing (NDA and/or Open Source). Standardizing these solutions can enable scalable deployment of debug features and tools in productions, which facilitates debug data sharing with suppliers to accelerate RCA.

	inability to audit for vulnerabilities, and software licensing issues.	
Triaging failures Across the System stack: From Hardware, Firmware, OS and drivers	A certain percentage of failures stem from system-level issues that cannot be immediately traced to a specific layer, such as hardware, firmware, OS, or drivers. These issues require extensive root cause analysis (RCA), often taking longer to diagnose. Debugging such failures involves collecting and analyzing debug dumps across the entire system stack to isolate the problem to a specific sub-area.	Effective troubleshooting requires advanced hardware debugging techniques capable of capturing silicon state registers, OS and driver stack data, and real-time runtime states. For example, state dump features like JTAG-based debugging can be instrumental in identifying these complex failures.
Supplier-based custom solutions	Hyperscaler fleets include systems based on several hardware architectures (e.g., x86, ARM), each with custom debug interfaces and protocol implementations. This fragmentation requires hyperscalers to integrate multiple debugging solutions, increasing complexity. Additionally, multiple ARM CPU providers offer customized solutions, further complicating standardization and integration efforts.	Develop standards-based interfaces and solutions would streamline debugging across x86 and ARM architectures, reducing integration complexity for Hyperscalers. A unified framework for debug interfaces and protocols would enable seamless interoperability across vendors, improving efficiency, security and scalability.

By collaborating within OCP and engaging between hyperscalers and silicon vendors, we can address these pain points and develop robust and standardized solutions for debugging CPU-related issues across different architectures. This will contribute to a more reliable and efficient infrastructure for the broader community.

4.2 Goals

To address the problem statement, the primary goals of this specification include:

1. Enhance Debug Feature Security and Data Sensitivity Controls: Implement debug feature access controls that adhere to the latest industry security standards, offering granular access based on the least privilege principle, while maintaining consistency across development and production phases.
2. Improve Root-Cause Analysis (RCA) for Workload-Specific Problems: Develop standardized debug interfaces and protocols that streamline the enablement, configuration, and retrieval of CPU debug data in production to meet SLAs without compromising security.
3. Scale Debug Features and Tools from Development to Production: Use remote, automation-friendly debug workflows that securely invoke debug features and collect debug data without human intervention.
4. Enable CSPs to Operate Independently to Diagnose Hardware Failures: Develop analysis tools that process telemetry to identify the nature of the failure and recommend repair actions.
5. Improve Triaging of Failures to Drivers, Software, Firmware, Hardware Issues: Enhance diagnostic tool output to provide high-confidence triage instead of generic advice like “no issue found”.
6. Develop Standards-Based Interfaces and Solutions for Supplier-Based Custom Solutions: Develop standards-based interfaces and solutions to streamline debugging across x86 and ARM architectures, reducing integration complexity for hyperscalers.
7. Compliance Tools: Create open-source software that performs compliance checks for suppliers and system integrators to ensure CPU compliance with the requirements described in this document.

5 Standard Debug Workflow

All hardware has known failure rates, but the patterns and impacts of these failures can vary in production. Effective and quick debug-ability in production is essential to minimize impact on end-customers. Identifying root causes quickly allows hyperscalers to implement mitigation measures that enhance system resilience to known failures. Therefore, a pre-approved debug workflow is crucial for efficient data collection during failures in production, ultimately saving time and resources, and reducing the overall time to root cause (Mean Time to Debug, Mean Time to Resolve).

This section proposes a standardized debug workflow to address this issue. By providing a single framework to use in production, it simplifies the logistics, security and other aspects associated with deploying multiple vendor-specific solutions that may differ significantly from each other.

5.1 Overview

Current debug methods in production data centers primarily rely on OS/CPU crashdumps and telemetry from system firmware (e.g., UEFI/BMC System Event Logs). While effective for most failures, they fall short for about 20% of cases, which are often complex and hard to diagnose. These issues may stem from silicon defects, logic bugs, firmware/software bugs, hypervisor hangs, or OS partition deadlocks.

Additional debug workflows are necessary to address the remaining percentage of failures. In production hyperscale environments, these workflows must: a) adhere to strict security requirements, and b) minimize disruption to data center operations, enabling hyperscalers to meet uptime targets and service level agreements (SLAs).

This section outlines a layered approach that integrates with the CPU crashdump flows used for managing fatal errors. This allows deeper levels of debug data collection to be enabled precisely and only when required by the specific failure signature being debugged. The following diagrams illustrate this proposal.

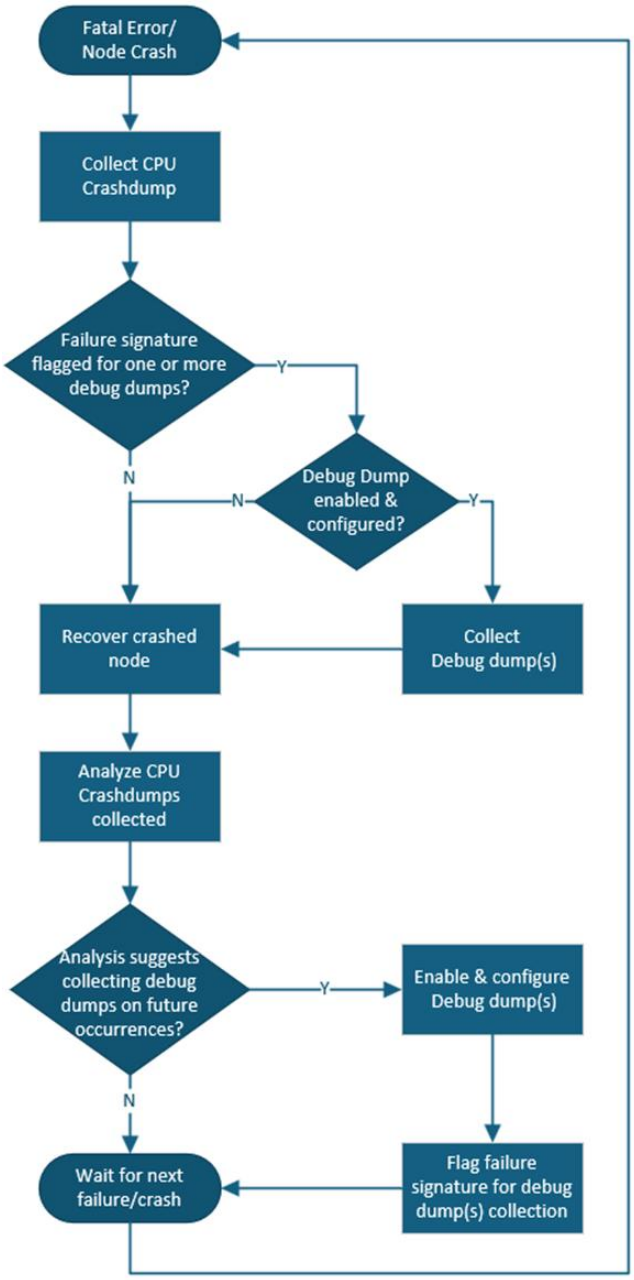


Figure 2: Crashdump Flow

1. The flow starts every time a fatal error that results in a node crash occurs. As a reminder to the reader, this document release focuses on fatal error severities due to their disruptive impact on hyperscaler service availability goals.
2. The CPU crashdump flow is executed. Implementations may vary, but it typically involves a platform monitoring agent (e.g. a BMC) detecting the node crash and initiating the crashdump routine.

3. The CPU crashdump data can be used to determine whether the current fatal error event requires additional debug dump collection. One implementation example involves computing a failure signature and comparing it against a list of qualifying failure signatures provided by the vendor. This computation may be performed by services in the data center (DC) control plane or may be offloaded to the platform monitoring agent. The vendor may provide this list via different means, although the method preferred by hyperscalers is via a vendor analyzer tool that can process crashdumps.
4. If the fatal error event qualifies, the next step is determining if the corresponding debug dump is currently enabled and configured in the crashed node. Otherwise, it may be challenging to disambiguate if failed attempts to collect debug dumps are unexpected (error) or expected (disabled).
5. Assuming that all debug dumps are initially disabled by default, no debug dumps are collected. The crashdump is saved in a persistent manner and the crashed node can follow hyperscaler-specific recovery flows. Most of the time, this may result in a node reset, although other service or repair flows may apply.
6. At this point, hardware fault management services in the DC control plane have been notified of the fatal error event and the presence of the associated crashdump. These services proceed to decode and analyze the crashdump via vendor-provided Analyzers, which may suggest collecting one or more debug dumps on future instances of this fatal error event.
7. If permitted by hyperscaler policy, the DC control plane services may follow the Analyzer's suggestion and opt-in to enable and configure the suggested debug dumps in a subset of nodes considered likely to fail again. This step is referred to as the "Configuration Phase" throughout this document.
8. Once the debug dump features are successfully enabled and configured, DC control plane services may subscribe to these fatal error events and wait to be notified on any occurrence.
9. The flow restarts at step #1 in any node encountering the flagged fatal error event, with the following differences:
    a. In step #5: If one or more debug dumps are enabled under the current fatal error event, then attempt to collect all applicable debug dumps. This step is referred to as the "Collection Phase" throughout this document.
    b. In step #6: hardware fault management services in the DC control plane are also notified about the presence of one or more debug dumps, in addition to the associated crashdump.

These services typically will not attempt to decode debug dumps, as these need to be sent back to the vendor for further root-cause analysis.

It is acceptable to change the order of certain steps in the high-level workflow as long as they meet the goal of a layered approach on a per-failure signature basis. For example, a Hyperscaler with a real-time crashdump processing infrastructure may not want to recover the crashed node before determining if an extended register dump is recommended by the analysis results, as this removes the waiting cycle for the next failure occurrence to reproduce.

Note that between configuration and collection phases, there needs to be an associated “triggering” phase that produces the debug dump. Implementation of triggering in hardware and/or platform monitoring agent is vendor-specific and is outside the scope of this specification.

5.2 Configuration Phase

This section covers the debug dump Configuration Phase in more detail. The purpose of this phase is to enable and properly configure the desired debug dump features before the fatal error event of interest occurs. The following diagram illustrates this concept.

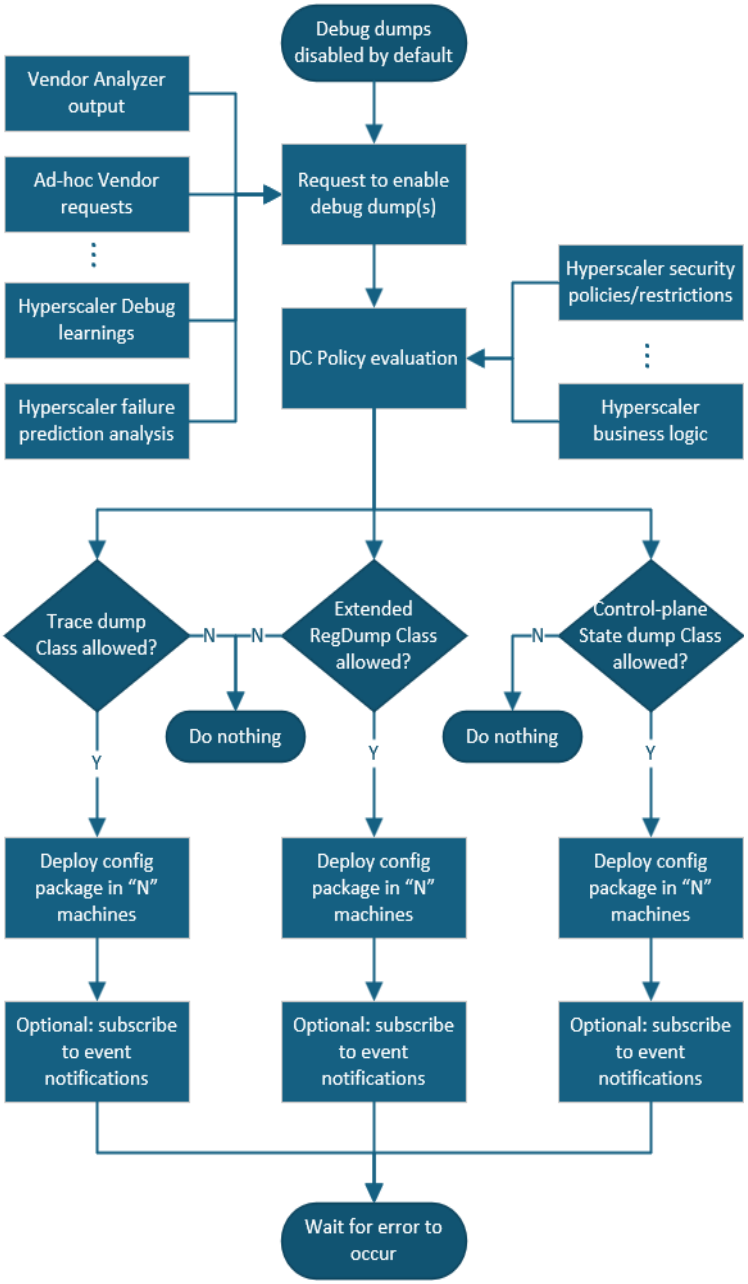


Figure 3: Debug Dump Configuration Flow

1. Debug Dumps are initially disabled by default and there is a request to enable them. Sources of the request may include vendor Analyzer output, (e.g., collect Extended RegDump on 3rd strike), ad-hoc vendor request, (e.g., task-force requests collecting trace logs on future error instances), hyperscaler debug learnings, (e.g., root-causing previous timeouts required sending state dumps to vendor), and proprietary failure prediction algorithms.

2. The request is evaluated against hyperscaler-specific Data Center Policy, which may grant or deny the request at that particular moment based on security restrictions (e.g., confidential workload, sensitive dump, capacity constraints, business logic, etc.).
3. If a particular debug dump class is allowed, then data center control plane services opt-in to enable and program debug dumps in a subset of nodes. Data center control plane services may also subscribe to notifications on debug dump events.

Enabling and programming debug dumps may require deploying a configuration package or "recipe" to the selected nodes. Hyperscalers expect all the necessary components (e.g., commands, configuration files, vendor-specific payloads, etc.) to be provided by the vendor, via vendor Analyzer output (preferred) or other distribution means. Details of configuration packages are vendor-specific and are outside the scope of this specification. However, this specification requires these packages to a) be self-contained, such that they can be exchanged as payloads within the hyperscaler data center control plane, and b) not require a deployment method that interrupts end-user workloads currently running (e.g., hot-loadable). The configuration package must include everything that is needed to properly program the hardware and/or platform monitoring agent, including trigger conditions and any flagging that future occurrences of a fatal error event qualify for additional debug data collection.

Please note that this proposed flow assumes the debug dumps configured are NOT sensitive dumps; they are guaranteed to exclude user data. Configuring sensitive dumps is part of future work as of this specification release.

5.3 Collection Phase

This section covers the debug dump Collection Phase in more detail. The purpose of this phase is to attempt to collect non-sensitive debug dumps available. There may be cases where non-sensitive debug dumps may be unavailable or insufficient for root-cause analysis, requiring collection of sensitive debug dumps. Collecting sensitive dumps is part of future work as of this specification release.

Once a fatal error event that qualifies for additional debug dumps occurs, their collection phase proceeds according to the following sequence:

1. After collecting the corresponding CPU crashdump, the next step is determining if any non-sensitive debug dumps are enabled and configured in the crashed node. This evaluation must be performed

in this phase since the collection services may not have state saved from the previous debug dump configuration phase (i.e. stateless).

2. If one or more non-sensitive debug dumps are enabled, then execute any required pre-collection steps. Examples include disabling watch-dog timers (WDT) used in mission-mode, enabling debug-mode WDTs, performing security authorization/authentication flows, establishing secure channels, launching additional debug services, scripts or tools, etc.
3. Collection agents attempt to collect the non-sensitive debug dumps enabled and handle any collection errors encountered. These agents may be implemented as debug services in the data center control plane, services running in the platform monitoring agent, or a combination of both. Having the platform monitoring agent (e.g, BMC) iterate through all debug dumps that should be collected and then inform the data center control plane upon task completion is the preferred method by hyperscalers participating in this workstream.
4. All collected dumps are saved in a persistent manner. The choice of storage should ensure the debug dumps persist through node recovery or node state changes in the data center. For instance, state dumps may require a complete AC cycle of the node to recover, so storage persisting only through warm reset would be inadequate. Similarly, a debug dump stored on the hard disk of a node that transitions from production to diagnostics state, where a newly flashed hard disk image overwrites previous data in the hard disk, may also be inappropriate. The recommendation is to save these dumps outside of the node, preferably in a database with medium-long term retention policies.
5. At the end of the flow, hyperscaler policy may decide to keep the node in the failed state for more interactive debug, which is outside the scope of this specification. Otherwise, the node is recovered, and non-sensitive debug dumps can be sent to the vendor for further analysis.

The following diagram illustrates this flow.

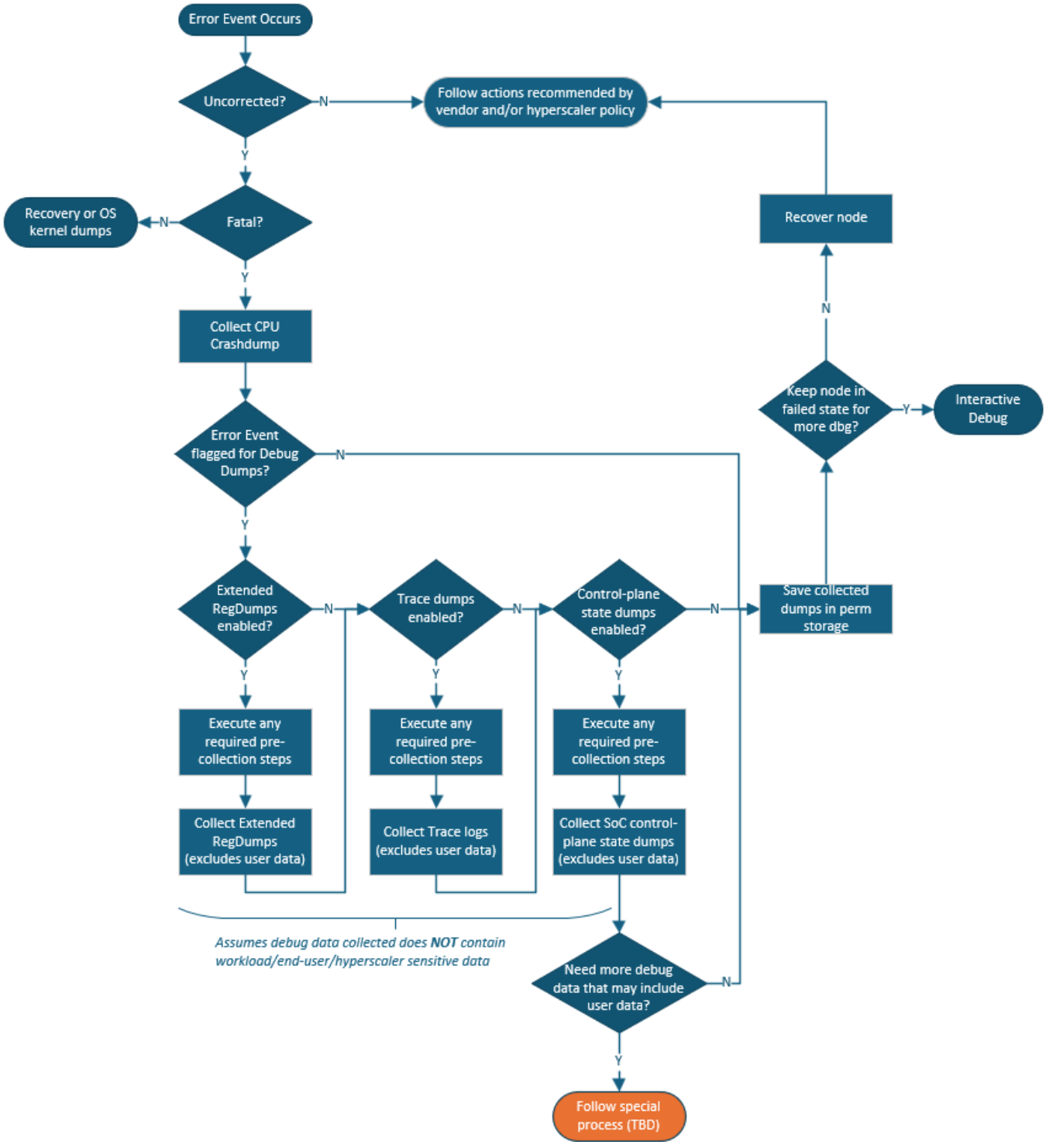


Figure 4: Standardized Debug Workflow

6 Crashdump Requirements

This section details crashdump requirements for day-to-day hardware fault management of fatal errors in hyperscale production environments.

1. The Crashdump feature shall:
    a. Be enabled by default at the node level.
    b. Expose standardized interfaces for configuration and control. More details will be provided in future versions of this document.
    c. Include post-processing tools that decode and analyze the data collected, as defined in the OCP Fault Management Infrastructure Requirements[1].
2. Crashdump data collection process shall:
    a. Trigger automatically at the node level upon detecting the crash state without dependencies on Hyperscaler management/control plane.
    b. Notify Hyperscaler management/control plane upon completion.
    c. Include survivability features to recover from errors or hangs in the data collection procedure (e.g., timeouts, retries, error codes, etc).
3. Crashdump data collection process shall NOT:
    a. Rely on data-plane hardware components that may not operate reliably during a fatal error (e.g., system memory, high-speed data links, hung compute cores, etc).
    b. Require “phone home” methods for vendor authorization prior to collection.
    c. Use hardware interfaces that provide unbounded access to the processor state (e.g. JTAG).
4. Data collected shall:
    a. Meet the standard record format defined in OCP Fault Management Infrastructure Requirements[1].
    b. Be sufficient for triaging to a faulty FRU for hardware fault management.
5. Data collected must NOT contain Hyperscaler sensitive data, end-user sensitive data (e.g., PII, PHI), or data collected directly from the workload that was executing. This allows crashdumps to be available even in confidential compute environments.
6. Data collected should be sufficient to guide further root-cause analysis if the crashdump data is insufficient for this purpose. It may contain vendor-specific data bound by Non-Disclosure Agreements (NDA) for guiding next-level debug dumps or further root-cause analysis.

7 Debug Dump Requirements

This section details debug dump requirements for use in hyperscale production environments. These requirements target the use case of debugging issues that only manifest under certain production workloads in large, automated hyperscale data centers.

7.1 General Requirements

1. Debug Dumps shall:
    a. Be disabled by default at the node level.
    b. Implement modern Data Center production-grade secure authentication mechanisms to enable the feature. More details will be provided in future versions of this document.
2. Debug Dump configuration process shall:
    a. Expose standardized interfaces for configuration and control. More details will be provided in future versions of this document.
    b. Implement a mechanism to programmatically identify if vendor-specific configuration packages or tools are required for debug dump collection.
3. Any required configuration package shall:
    a. Be self-contained, such that they can be exchanged as payloads within the Hyperscaler DC control plane.
    b. Include everything that is needed to properly program the hardware and/or platform monitoring agent, including trigger conditions and any flagging that future occurrences of a fatal error event qualify for additional debug data collection.
    c. Be protected with up-to-date encryption schemes (e.g. password protection is not acceptable). More details will be provided in future versions of this document.
4. Any required configuration package shall NOT:
    a. Have a total size larger than 50MB.
    b. Require a deployment method that interrupts end-user workloads currently running (e.g., FW updates or early-boot windows that require system reset).
5. Debug Dump data collection process shall:
    a. Trigger automatically at the node level upon detecting specific failure modes or signatures previously opted-in or pre-programmed by the Hyperscaler management/control plane.
    b. Notify Hyperscaler management/control plane upon triggering.

   c. Notify Hyperscaler management/control plane upon completion.
   d. Include survivability features to recover from errors or hangs in the data collection procedure (e.g., timeouts, retries, error codes, etc).
6. Debug Dump data collection process shall NOT:
   a. Require "phone home" methods for vendor authorization prior or post collection.
   b. Require human intervention or manual procedures to execute once enabled and configured.
   c. Require human intervention or manual procedures to recover the system after completion. Reset operations such as physically unplugging the AC power cord are unfeasible in hyperscale environments. The required level of reset to the CPU must be performed at the node level by the platform monitoring agent (e.g. BMC). This requires that the reset logic and power rails of the platform monitoring agent survive the CPU reset.
7. Debug Dump data collection process should NOT:
   a. Have an execution time longer than 15 minutes. Longer execution times may be treated on a case-by-case basis by each Hyperscaler.
8. Debug Dump data collected shall:
   a. Be cryptographically protected in-transit and at-rest, following minimum security standards. More details will be provided in future versions of this document.
9. Debug Dump data collected should not contain Hyperscaler sensitive data, end-user sensitive data, or data collected directly from the workload that was executing. In case this requirement cannot be guaranteed by the vendor, then vendor must facilitate a data-auditing process before the debug data can be released by the Hyperscaler.
10. It is recommended for debug dumps to also use the CPER format required for crashdumps. The OpenBMC framework provides standard libraries to encapsulate any data blob into the CPER format. However, this may not always be practical. For example, converting a short trace dump into the CPER format may be easier than converting a JTAG-based scandump due to the large size of the data dump and the absence of firmware agents to "CPER-ize" the data. Therefore, using the CPER format is strongly encouraged but not mandatory.
11. Vendor-provided sources of debug dump requests (e.g. analyzer outputs, ad-hoc requests) shall include the following properties for Hyperscaler policy evaluation:
   a. Dump class identifier (e.g., extended RegDump, Trace dump or State dump).
   b. Estimated dump time, in seconds.
   c. Estimated final dump size, in Bytes.

d. Dump data sensitivity classification (e.g., non-sensitive, sensitive).

12. Any required tools or scripts to execute and/or collect debug dumps shall:
    a. Be provided in source code form. Hyperscaler production environments typically prohibit the deployment of third-party applications not coded and signed by the Hyperscaler.
    b. Use memory safe languages such as Python (preferred), RUST, or C#.
    c. Be designed to function in isolated execution environments.
    d. Follow adequate coding practices for errors, including but not limited to:
        i. Exceptions, exit codes get properly propagated to the caller.
        ii. Exceptions and error-related output reported to stderr.
        iii. Include logging capabilities for debugging integration in Hyperscaler environment.
    e. Follow customary revision control practices and include a changelog.
    f. Adhere to best practices of backwards and forward compatibility, including API, features and structures.
13. Any required tools or scripts shall NOT:
    a. Rely on hardcoded filenames, absolute paths, shared resources for operation. No folder/directory names with illegal characters.

7.2 Extended RegDump Requirements

Extended register-based dumps may be used to augment debug data for diagnosing complex issues that may not be evident from basic crashdumps alone. These may use the same interfaces used for crashdump (e.g., I2C, I3C, PECI), bare-metal debug/test interfaces (e.g., secure JTAG, USB), or a combination of both. Extended register dumps are strongly encouraged before resorting to state dumps due to simpler data auditing logistics. Auditing or certifying exclusion of sensitive data in arbitrary chains of flip-flops or arrays is significantly more challenging compared to doing so in spec-defined registers.

In addition to the general requirements in Section 7.1, Extended Register dumps are expected to meet the following requirements for use in production hyperscale environments.

1. Extended Register dumps shall:
    a. Be configurable to increase or decrease the amount of data (e.g. registers) collected in the dump.
2. Extended Register dumps shall NOT:

a. Exceed 50MB in dump size. If compression is needed to meet this target, *gzip* is the recommended compression algorithm.
a. Use hardware interfaces that provide unbounded access to the processor state. If using interfaces like JTAG, it shall be Secure JTAG and be granular enough to only expose the state dump capability. The definition of what qualifies as Secure JTAG will be provided in future releases of this document.

3. Extended Register dumps should NOT be destructive to maintain correctness of subsequent dumps collected or interactive debug activities.

7.3 Trace Dump Requirements

Trace dumps are a result of regular collection of trace data that shows the operation of the CPU over time, which logs the sequence of events that leads to a CPU failure. This data may need to be correlated with end-state Crashdumps or State Dumps. Examples include Processor/microcontroller instruction trace, external bus trace, and internal signal logic traces.

In addition to the general requirements in Section 7.1, trace dumps are expected to meet the following requirements for use in production hyperscale environments.

2. Trace dumps shall:
    a. Operate while the CPU's production workloads are executing.
    b. Be configurable to increase or decrease the amount of Trace Data collected in the Trace Log.
    c. Be reversible (i.e. allow the system to return to its original state).
3. Trace logs shall NOT:
    a. Stop execution of the CPU's production workloads.
    b. Alter the state or correct execution of the CPU's production workloads.
    c. Affect the performance of the CPU's production workloads beyond a very minimal overhead.
    d. Exceed 50MB in log size. If compression is needed to meet this target, *gzip* is the recommended compression algorithm.

7.4 State Dump Requirements

State dumps provide a static snapshot of the CPU state, offering increased visibility for root-cause analysis when other types of dumps are insufficient. These dumps capture extended CPU state, such as flip-flops, state machines, internal status signals, SRAM, or queue contents. State dumps are typically executed and

collected via bare-metal debug or test interfaces such as secure JTAG and USB. Examples include Scan dumps and Latch/Array dumps. Collecting state dumps at the time of failure can accelerate vendor-driven root-cause analysis, especially when failures are hard to reproduce or the workload causing the issue isn't accessible.

In addition to the general requirements in Section 7.1, state dumps are expected to meet the following requirements for use in production hyperscale environments.

4. State dumps shall:
    a. Be configurable to increase or decrease the amount of data collected in the dump.
5. State dumps shall NOT:
    e. Exceed 300MB in log size. If compression is needed to meet this target, *gzip* is the recommended compression algorithm.
    a. Use hardware interfaces that provide unbounded access to the processor state. If using interfaces like JTAG, it shall be Secure JTAG and be granular enough to only expose the state dump capability. The definition of what qualifies as Secure JTAG will be provided in future releases of this document.

8 Diagnostics

This section covers the standardization of Hyperscale CPU Diagnostics, this includes diagnostics for manufacturing screening.

8.1 Diagnostics Standardization Overview

Diagnostics standardization follows the following end-to-end flow, targeting the complete development cycle.

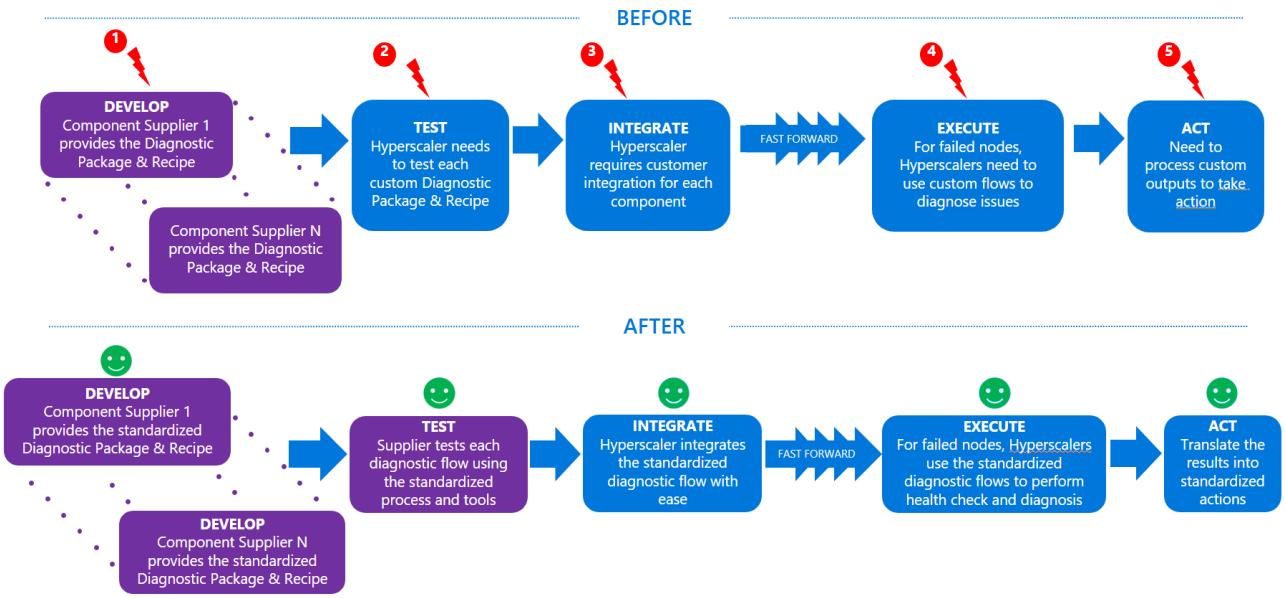


Figure 5: Pain-points and desired solutions for end-to-end Diagnostics.

The diagram shows pain points and solutions in the end-to-end Diagnostics Flow, with supplier effort in purple and hyperscaler in blue. Each stage reveals challenges paired with workflow or technology solutions.

Table 2: Diagnostics pain points, impacts, and desired solutions by stage in the end-to-end flow.

Stage	Pain points	Impact	Desired Solution
1	Suppliers are starting from scratch to define diagnostics package and need to redesign to cater to each CSP.	Engineering Effort and Delays	Standardized Diagnostics Package & Recipe
2	Heavy lift needed by the hyperscaler to learn, test, and integrate each diagnostics tool, Recipe and workflow for various suppliers components	Engineering effort and Quality	Suppliers provide the Diagnostic Package & Recipe based on standardization spec requirements
2	Assumptions regarding presence of HW/FW and SW interfaces within the fleet diagnostic environment. e.g., software versions, interfaces, packages, or modules (e.g., kernel version).	Product Intercept delays	Supplier tests each diagnostic flow using the standardized Interfaces and Open source Linux based Kernels. Supplier Diagnostics tools should not automatically abort based on software version mismatch.
3	Implicitly reflects the pain point for the silicon/component vendor needing to handhold each hyperscaler through the integration steps.	Silicon vendor resources	Standardized diagnostics recipe.
3	Diagnostic tool output requires bespoke interpretation and analysis. Output may not be actionable.	Repair Actions not clear	Broadly defined Standardized Diagnostics Analysis & Actions.
3	Potential for, or presence of, security vulnerabilities and no industry standard frameworks	Security impact	Suppliers become aware of hyperscaler security requirements. Streamlined security reviews given standard structure.
3	Support for Bare metal/ OOB Diagnostics and log availability	Supports multiple Implementations	Encourage suppliers to standardize OOB diagnostics, including details on system disruption/ impact.
4	Redundant and excessive Diagnostics workflows increases downtime.	Out of Service/ Revenue	Compliance requirements will allow for evaluation of ROI. Optimization of Diagnostics workflows.
5	Every hyperscaler needs to define supplier-specific actions, even if similar.	Engineering Effort/ Costs, Product Intercept Delays	Broadly defined Standardized Diagnostics Actions.

8.2 Scope

This section formalizes the diagnostics requirements and capabilities to cover hyperscaler diagnostic needs and provides requirements for:

- Diagnostic Environment Requirements
- Diagnostic Tools Package Standardization
- Diagnostic Recipe Standardization
- Diagnostic Compliance Tool

Diagnostics play a critical role across manufacturing, development, and production environments, where they are used to detect faults, validate system functionality, and inform repair or maintenance actions. The requirements listed in this document are applicable across manufacturing screening used at system integrators and production phase stages like Out for Repair (OFR), empty nodes.

8.3 Diagnostics Environment Requirements

This section of the document specifies the minimum diagnostic environment requirements. It will cover all the hardware and software environment requirements which the standardized diagnostic flow will depend on. The diagnostics environment describes infrastructure, configuration, and conditions required to successfully run diagnostic processes on target system.

1. There will be a Device Under Test system which is accessible by a diagnostic invoker/machine or diagnostic host machine.
2. Operating System and Platform Compatibility
    a. The diagnostic environment must operate on a Linux-based operating system (per Compliance tool CPACT requirements).
    b. The Linux OS on the target system must support containerization, as diagnostic packages are typically deployed within containers.
3. Preloaded Software Requirements
    a. The target system must be preloaded with essential applications required by the diagnostic compliance tool to execute diagnostic flows. Please refer to section 8.6 for compliance tooling dependencies.
    b. These applications should be compatible with the diagnostic tools and recipes provided by the supplier.

4. Containerization and Dependency Management
    a. Diagnostic packages should be delivered as self-contained packages similar to Linux OS-based containers.
    b. All dependencies required by the diagnostic flow must be encapsulated within the container.
    c. For non-containerized diagnostic packages, all dependencies must be bundled within the package to ensure standalone operability.
5. The target system should be accessible by the diagnostic machine for in-band commands, to be run on the target OS.
6. The target system should be accessible by the diagnostic machine over out-of-band interfaces for observability and, optionally, for controllability as well. Access to diagnostic output logs through out-of-band interfaces is required to support troubleshooting when the system fails to boot.

8.4 Diagnostics Tools Package Standardization

Diagnostics tools package describes a bundle of diagnostic tools, scripts, binaries, and supporting files that are version-controlled and uniformly deployed across all systems.

While the diagnostic recipe will define how to use the diagnostic tools package, below are some of the requirements of the package itself:

1. Diagnostic tools package should comprise of the actual system exerciser & health check tools and collaterals needed to run the steps outlined by the diagnostic recipe.
2. Though suppliers have flexibility here, suppliers are encouraged to use Linux containers and/or open-source tools that are vetted by industry to be safe for the hyperscalers to deploy.
3. The diagnostic tools shall be functional across a range of Linux distributables to permit compatibility with different hyperscaler platforms.
4. If not generally available, the diagnostic tools package or the container should include other software dependencies within.
5. The diagnostic tools package should be well tested by the supplier in conjunction with the diagnostic recipe prior to being released to the hyperscalers.
6. The package should comply with regulations and data privacy laws ensuring that the diagnostic data is handled securely and responsibly.

7. The diagnostic tools should generate logs that are easy for the recipe to determine the result of the diagnostic run and the action required for each result (align with OCP Test & Validation workstream requirements).
8. In the event of unexpected results, the diagnostic logs should assist with additional logs that will help debug the issue and call out the next steps.

8.5 Diagnostics Recipe Requirements

Suppliers are expected to provide a standardized recipe that outlines the diagnostic flow steps for using the associated diagnostic package. The standardized recipe format should also specify how to interpret the output and detail the required actions. Below are the requirements for the recipe.

The input recipe contains a sequence of steps to be followed in a standardized format including command lines, interface dependencies, and output expectations of each step. The recipe output will help the supplier provide guidelines and a format on how to call out the result of each diagnostic recipe along with the logs to assist in describing the result. Along with each result, the supplier will be able to call out the corresponding action among a group of standardized actions. As the specification evolves, we will include an analyzer section that will permit suppliers to provide additional information that can analyze results across recipes and generate separate actions. This output will comply with the OCP Test & Validation (OCPTV) specification, which provides standards diagnostics tool output.

Diagnostic recipe standardization will help the supplier in detailing recipe requirements, recipe steps, and recipe output. The diagnostic recipe describes the sequence of steps to test a system, including input parameters, expected outputs, and pass/fail criteria. If a diagnostics tool failure results in a crashdump, the procedure outlined in Section 6 on crashdumps will be followed.

Assuming the system under test has been set up with the needed tools and other dependencies, the recipe will define the flow to use them on the target system under test. Below are the requirements for the diagnostic recipe:

1. The diagnostic recipe shall be provided in a format that is suitable for diagnostic application use (i.e. yaml or json). For more details, consult Section 8.6 Diagnostics Compliance Tool.

2. The diagnostic recipe must include an integrity check to verify that its structure, syntax, and dependencies are complete and uncorrupted prior to execution.
3. The recipe must include a header section which will list the recipe metadata and call out dependencies (e.g. location of the diagnostic packages) before starting the diagnostic flow.
4. Post the header section, the diagnostic recipe specifies the series of steps which MUST either involve a command execution OR a log analysis OR an invocation of another child-recipe. Currently, these are the only three permitted step types. There is no mandate on the ordering of these step types.
5. Each command execution step OR log analysis step in the recipe may involve an analysis that may either result in a diagnostic result code (DRC) being generated OR variables being set. While variables can be used for recipe flow control, DRCs get accumulated through the recipe to be returned at the end of the recipe.
6. Additionally, suppliers shall provide a separate map file (JSON) linking all possible DRCs from the recipe to diagnostic actions, with relevant instructions and messages.
7. Each diagnostic recipe step may define the entry criteria as a function or expression of variables defined in the prior steps. For example, a PCIe Link Check tool command in a command execution step may have an entry criteria set which checks if PCIe_Link_Error_Found, a variable, is set to true.
8. Each command execution and log analysis step must define the connection and connection type required for step execution.
9. A command execution step shall be permitted to be initiated and proceed with the next step of the recipe without waiting for completion. Such a step shall be allowed to be revisited for status at a later stage in the diagnostic recipe.
10. Each log analysis step must work on only one log file pointed to by a log path. The analysis step may look for diagnostic result codes (DRCs) or patterns in the log file to assign DRCs or variables, upon a successful match.
11. An invoke scenario or invoke recipe step can be used to construct a modular recipe flow. These steps shall have an optional entry criterion but must include the path to the child-recipe. The DRCs collected by the invoked recipe are returned to the invoking recipe or scenario.
12. The recipe may utilize regex based string search of diagnostic result codes OR for search strings in a log file OR for log file names OR for command execution output analysis & expected outcomes.
13. A recipe shall return a FAIL if recipe dependencies are not met OR if a log file is missing from the log path of a log analysis step OR if the output analysis of any command execution step fails. If an

invoked recipe returns a FAIL for any of the listed reasons, the invoker recipe shall also return a FAIL. The diagnostic tool consuming the recipe shall generate special DRCs for these recipe failures.

8.6 Diagnostics Compliance Tool

The final objective is to allow suppliers and hyperscalers to use an industry open-source compliance tool to vet the recipe and package. The tool will also execute the diagnostic flow using the same recipe and package while helping hyperscalers build confidence in the supplier provided diagnostic flows.

1. The compliance tool shall be invokable from inside the target system or from outside, like from a host machine.
2. Suppliers shall check for the goodness of the recipe using the compliance tool's recipe compliance check.
3. Suppliers and hyperscalers shall verify the validity of the diagnostic recipe and package by running it on the target system assuming the diagnostic environment requirements are met.
4. Compliance tool shall generate sufficient logs to help with diagnostic recipe results analysis, to assist suppliers with building robust diagnostic recipes and packages. The primary execution log must be aligned with OCPTV logging requirements.
5. The compliance tool shall house a folder which will have the schema and sample yaml or json files as examples of input recipe, DRC mapper, recipe result & output logs.
6. The compliance tool will be open sourced along with sample generic diagnostic flows for suppliers and hyperscalers across the industry to use. This will also encourage hyperscalers to build their diagnostic tools along similar lines.

Figure 6 covers the components of the Diagnostics Standardization. Additional breakdown of the components for each underlying section as well as a tentative diagnostic compliance tool architecture can be found in Appendix B – Diagnostics Compliance Diagrams.

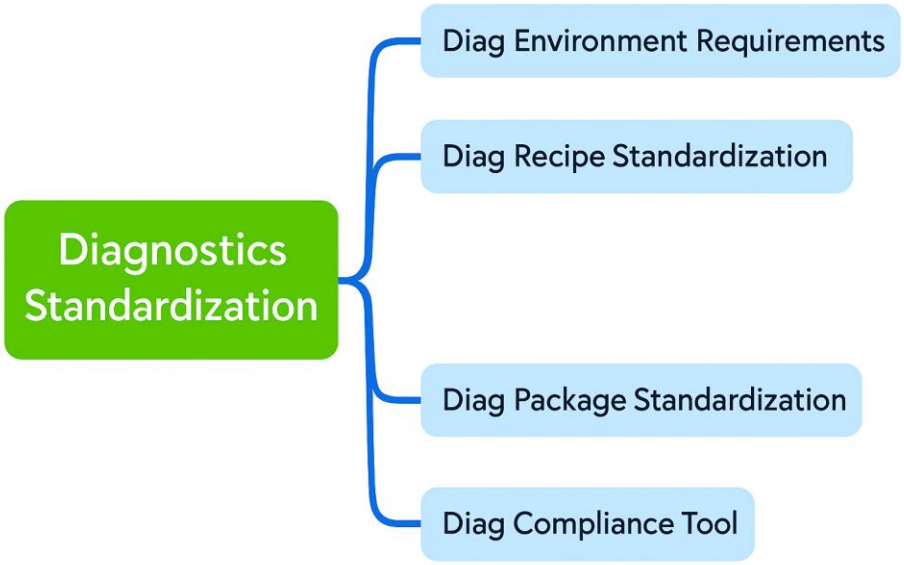


Figure 6: Diagnostics Standardization Components

9 Future Work

This section covers topics under discussion for future revisions of this document.

9.1 Standardized interfaces for feature configuration and control

Standardizing debug workflows in hyperscale data centers will require using common interfaces (e.g., APIs) for dump configuration and access control. The OCP RAS API subproject[2] is working on standardizing interfaces for similar actions. There is an opportunity to leverage this effort, particularly the flows and APIs for requesting RAS actions via *ActionIDs*. Each ActionID for debug dumps can remain vendor-specific as long as the Hyperscaler SW does not need to know its value. For instance, crashdump analysis results by a vendor analyzer may detect a failure signature that needs a trace log for root-cause analysis. The vendor analyzer may suggest enabling and collecting trace dumps on nodes sensitive to this failure signature. The analyzer provides the ActionID and required config file for the Trace dump. Hyperscaler infrastructure simply forwards the ActionID and payload to the nodes where tracing can be enabled per hyperscaler policy. The platform

monitoring agent (e.g., BMC) may perform additional steps to request the device to enable the ActionID for Trace dumps and forward the provided config file.

9.2 Security Requirements

Future updates will reference security requirements for production-grade data centers, including access control (e.g. authorization, authentication), encryption, data retention, and configuration persistency across resets for debug dumps. These security measures are paramount to protect sensitive data and maintain operational integrity. Adopting a secure framework approved by the security community will ease integration of debug dumps in hyperscale production data centers.

9.3 Debug Dump Data Sensitivity

The end goal of collecting debug dumps is to share them with hardware vendors for further root-cause analysis. Releasing debug dumps collected from deployments running production workloads is highly dependent on the sensitivity of the data collected. Future revisions of this specification will include guidelines corresponding to the sensitivity of the data extracted by a particular debug dump type, as well as a process to share sensitive dumps with vendors.

It is important to recognize that non-sensitive data may create a side-channel exposing sensitive information. This is beyond the scope of this initial specification but an area that can be expanded upon in future revisions.

Each Hyperscaler has the ability to define the processes to gather approvals to collect non-sensitive or sensitive information. These vary based on Hyperscaler-specific requirements, customers, and other policies (e.g., confidential compute). Given the industry variations, this specification does not prescribe exact procedures or rules for sharing debug data with vendors.

9.4 On-demand dump triggering

The capability to trigger debug dump collection on-demand for offline debugging purposes is being considered. Appropriate safeguards need to be put in place to prevent misuse of this feature, including measures such as rate limiting, authentication/authorization, and other controls to minimize unauthorized use that may impact the performance of the system.

9.5 Diagnostics Tools

Future revisions of this specification aim to address the current challenges faced by hyperscalers in diagnostics. These revisions will seek to standardize various aspects of diagnostics, including the tooling environment, physical interface requirements, input and output parameters, protocols and APIs, tool results (JSON-based), performance metrics, quality metrics, and effectiveness of diagnostic & screening tools. This will cover both CPUs, as well as GPUs/accelerators, storage devices, and network components. The objective is to develop a standardized methodology for diagnostic tools and processes, thereby enhancing fault isolation and detection, minimizing manual diagnostic efforts, and improving the efficiency of hyperscale data centers.

9.6 Bi-Directional Data Sharing

Future revisions will cover the debug data sharing between CSPs and silicon vendors. This will provide a standardized way of sharing data, avoiding the need for additional data reviews. This also streamlines the process for CSPs to extract the correct debug data to then share with silicon vendors.

9.7 Additional Dependencies

The current debug workflow proposed relies on successful crashdump execution and collection. There may be a subset of fatal error events that are missing crashdumps, or that the execution of crashdumps may change the failure state of the CPU. These scenarios are likely good candidates for collecting debug dumps. Future releases will consider a solution to enable, trigger and collect debug dumps in the absence of crashdumps.

Other dependencies in scope for future releases include logistics such as BMC development and standardization, upstreaming to the OpenBMC repository, etc.

9.8 Reliability, Availability, Serviceability (RAS)

Focus areas of RAS include

- CPU Fatal Error Resiliency – Minimizing Blast Radius – Currently, CPU fatal error recovery is very limited compared to memory error recovery flows.

- CPU Quality and Design Improvements - Expand workload-specific standardization and tooling to enhance CPU reliability and performance.
- Fatal Error Prediction & Analysis - Implement telemetry-based prediction (e.g., CPU SMART) to detect potential failures in advance.

9.9 Compliance tools

The OCP Test & Validation (OCPTV) initiative is a collaboration between hyperscalers aimed at standardizing hardware validation and diagnosis, along with providing necessary tooling[3] for diagnostic developers and executors to leverage these interfaces. In support of this, the Cloud Processor Accessibility Compliance Tool (CPACT) is being developed to allow hyperscalers and suppliers to check for CPU compliance with the specifications. The tool will be a combination of compliance scenarios and the underlying framework, categorized by domains. RAS & Debug requirements will be a domain by itself. The compliance scenarios for each domain will be dictated by the respective specifications.

The CPACT compliance tool will support a range of access methods to the target, both inband and out-of-band. It will also provide a standardized way to build the results for all the scenarios run on the target. All logs collected will be available for detailed analysis. This tool bridges hyperscalers and suppliers, assisting in the onboarding of supplier diagnostic capabilities with confidence. Once the specifications are public, the tool will become available publicly on GitHub. The tool will leverage OCPTV's philosophy and underlying *pytest* based framework to create each of the different compliance scenarios.

10 References

1. OCP Cloud Processor Accessibility Compliance Tool (CPACT) GitHub
2. OCP Fault Management Infrastructure Requirements V1.0
3. OCP RAS API Subproject
4. OCP Test & Validation Project
5. Cloud Processor Accessibility Compliance Tool - CPACT
6. OCP Test & Validation GitHub
7. OCP Hyperscale CPU RAS and Debug Requirements V0.5
8. OCP Hyperscale CPU Impactless Firmware Updates Requirements V0.5

11 OCP Tenets

Openness

This specification was developed collaboratively among industry partners and competitors, and is open to all OCP members.

Efficiency

The goal is seamless CPU integration into hyperscaler solutions, with compliance tools ensuring high-quality products.

Impact

As the first initiative to standardize CPU Debug & RAS requirements, it will significantly improve quality and TTM for CPU Systems by hyperscalers.

Scale

Applicable to large-scale CPU deployments in hyperscale Data Centers.

Sustainability

Enhanced debug and root cause analysis will reduce component replacement rates.

12 Appendices

The appendices contain useful background details and will continue to be revised as per each new version of the specification.

12.1 Appendix A - Glossary of Terms

Term	Definition
AI	Artificial Intelligence.
BMC	Baseboard Management Controller.
CPER	Common Platform Error Record, as defined by UEFI specification.
CPU	Central Processing Unit.
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.
DC	Data Center.
Fault	A fault is something that causes hardware to malfunction. If the hardware detects the fault, it can log errors that aid in diagnosing the fault and servicing the hardware.
FRU	A Field Replaceable Unit is a part that can be changed in the data center to repair a broken system, for example a DIMM or a CPU.
FW	Firmware.
GPU	Graphics Processing Unit.
HPC	High-Performance Computing.

Hyperscalers	Cloud service providers of the world, who value scaling and rapid time to market of new hardware products.
Hardware component	A component designed by a supplier. Examples of hardware components include CPU chips/SoCs, DRAMs and PCIe devices.
I2C/I3C	[Improved] Inter-Integrated Circuit. Bare metal interface to access on-chip debug and fault management resources.
IP	Intellectual Property.
JTAG	Joint Test Action Group. Bare metal interface to access on-chip debug and test resources.
MTTD	Mean Time to Debug
MTTR	Mean Time to Resolve
NDA	Non-Disclosure Agreement.
Node	A platform designed by a supplier integrating multiple components that run an operating system. These are sometimes also called compute nodes.
OpenBMC	An open-source project aimed at providing a common framework for managing and controlling BMCs.
OS	Operating System - that runs on the processor. The operating system may run in a virtual machine on a hypervisor or directly on the machine (bare metal).
PECI	Intel's proprietary Platform Environment Control Interface.
PHI	Personal Health Information.
PII	Personally Identifiable Information.
Platform firmware	Firmware running on mission-mode cores on the processor. On x86 CPUs, platform firmware is sometimes called BIOS or UEFI firmware.
RAS	Reliability, Availability, and Serviceability.

RCA	Root Cause Analysis.
SLA	Service Level Agreement.
SMM	System Management Mode. Privileged execution mode of x86 architecture.
SPDM	Security Protocol and Data Model, defined in www.dmtf.org.
SW	Software.
TTM	Time to Market.
UEFI	Unified Extensible Firmware Interface, defined in www.uefi.org.
USB	Universal Serial Bus
WDT	Watchdog Timer

12.2 Appendix B – Diagnostics Compliance Diagrams

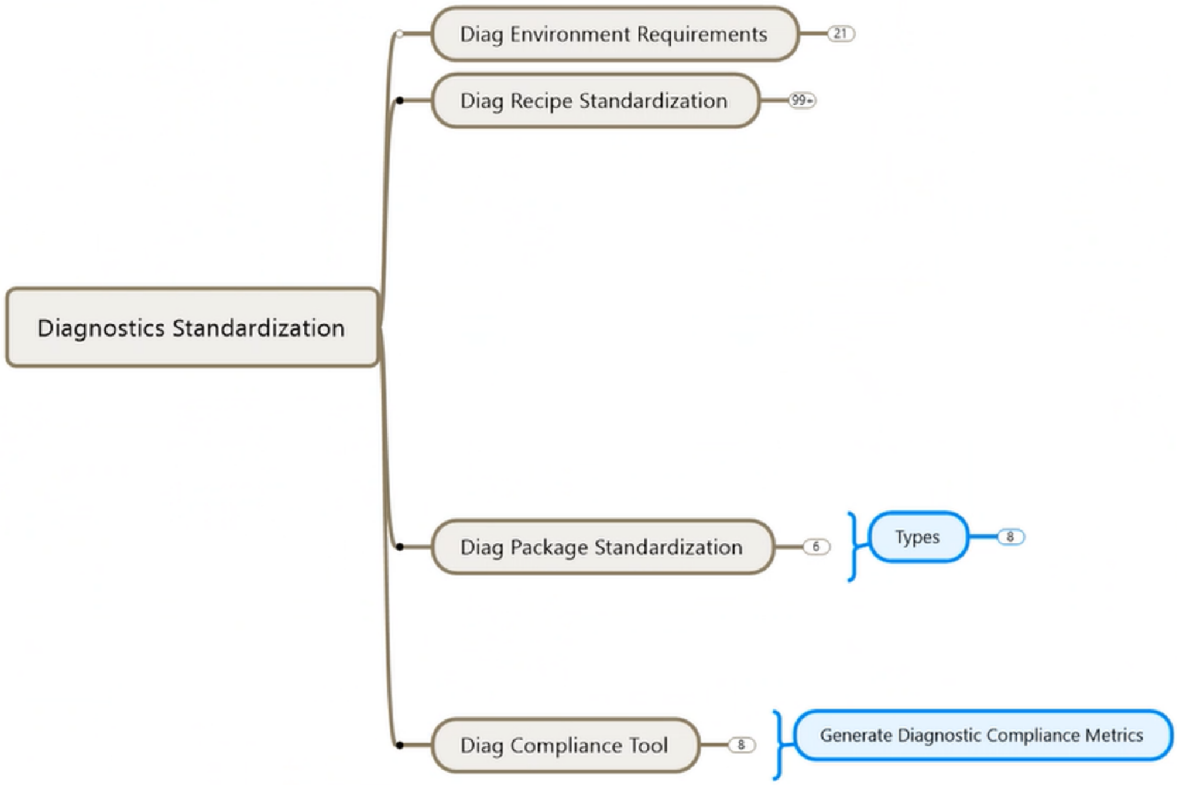


Figure 7: Diagnostics Compliance Components

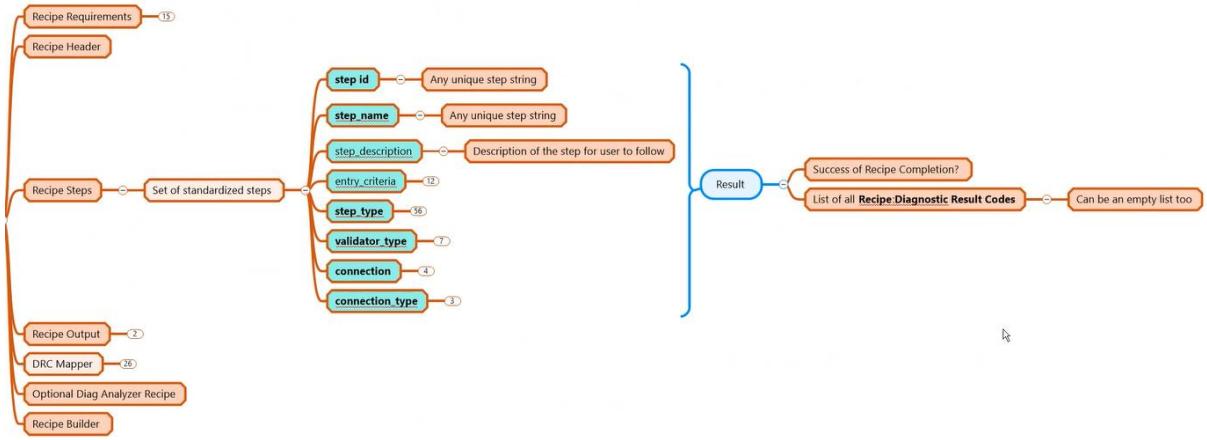


Figure 8: Diagnostics Recipe Standardization Components

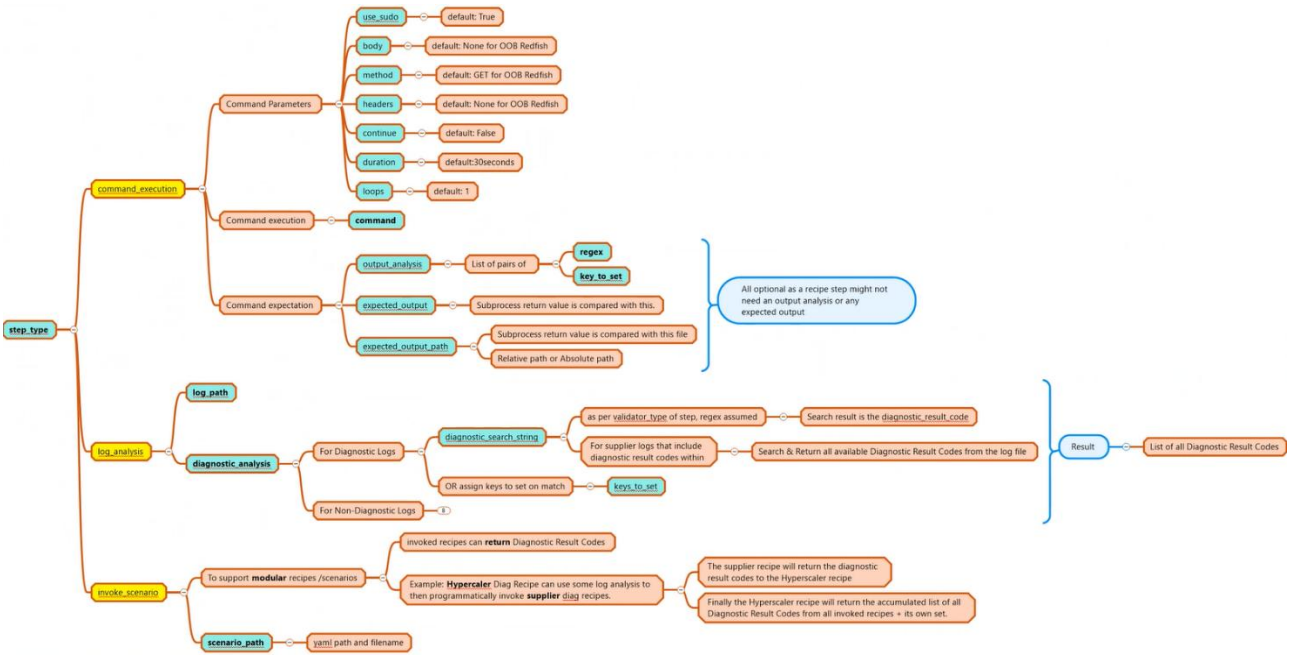


Figure 9: Diagnostics Recipe Step Types

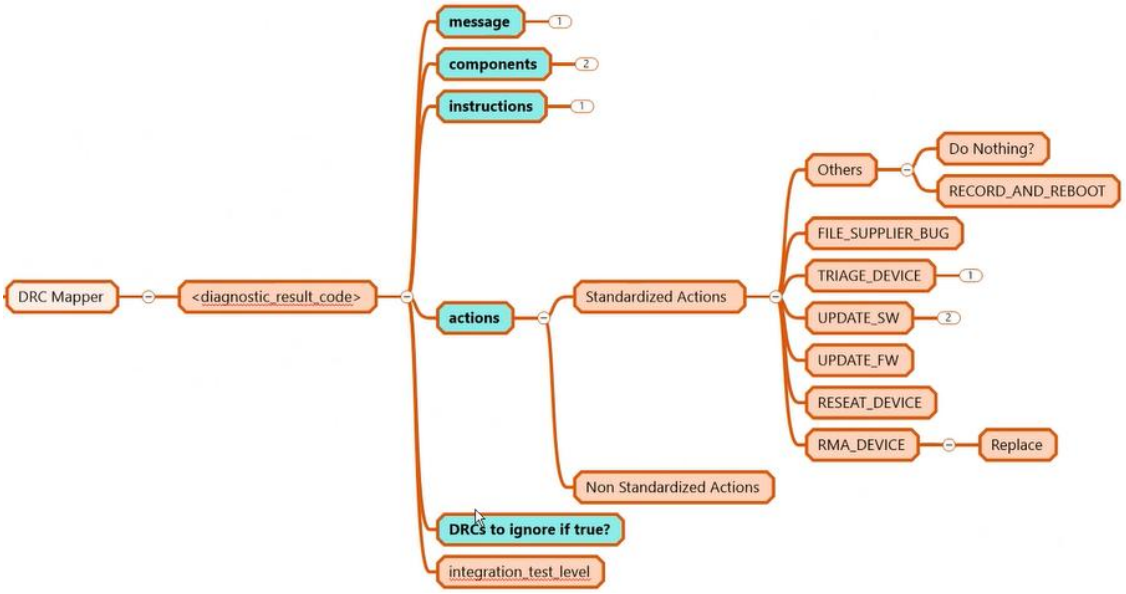


Figure 10: Diagnostics Result Code Components

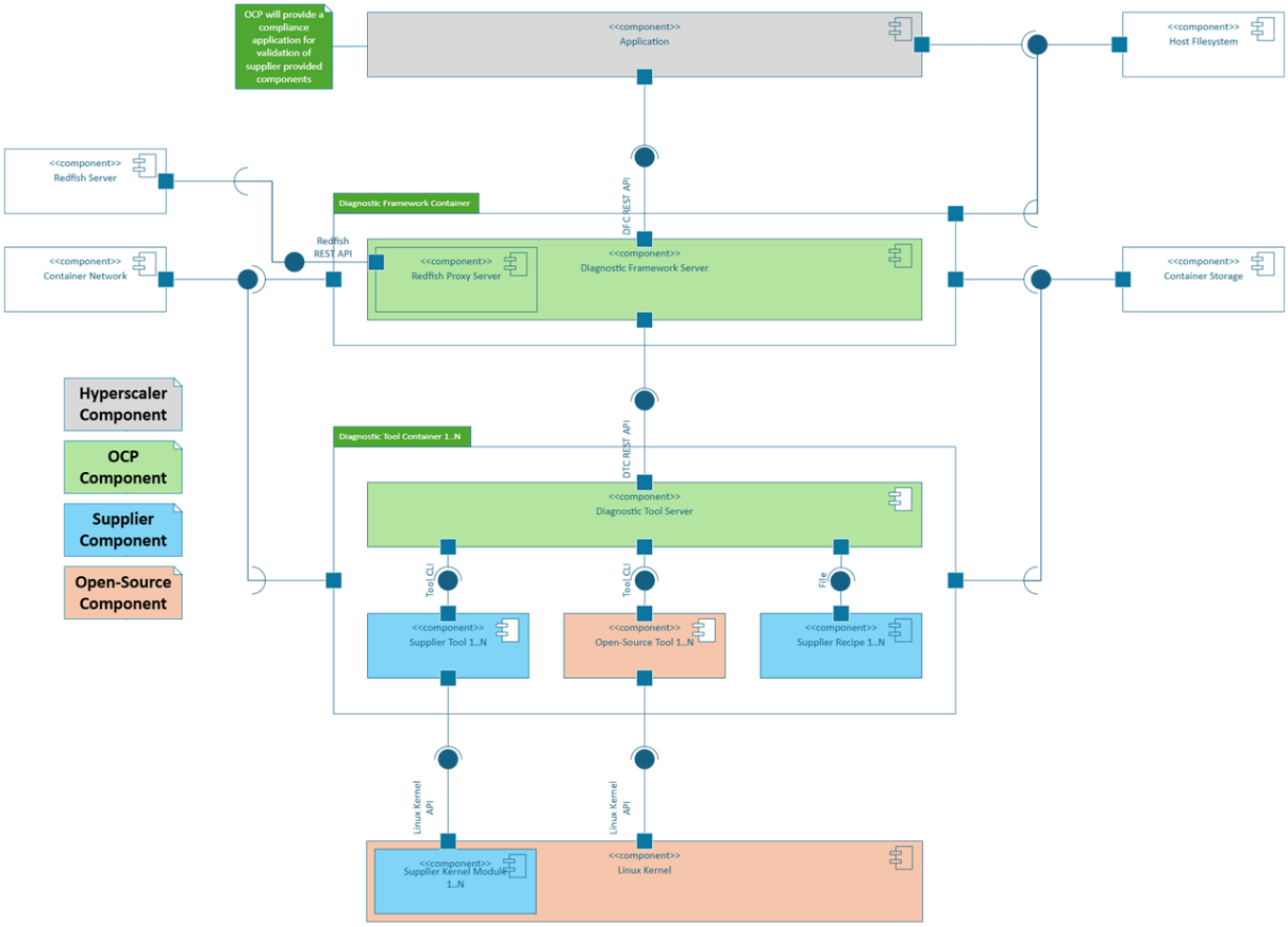


Figure 11: Sample Illustration of the Diagnostic Compliance Tool Architecture