The Software Safety Ceiling: Why Physical AI Cannot Be Made Safe in Software Alone

Mati Melchior
Independent Physical AI Safety Researcher
mati@physical-ai-safety.com

May 2026

Abstract

Physical AI systems control actuators that can cause physical harm. Current safety practice in this domain leans heavily on software mechanisms: monitors, watchdogs, redundant channels, and formal methods. This paper argues that software-only safety has a fundamental architectural limit. The mechanism is fate-sharing: a software safety monitor and the components it monitors share a substrate — CPU, memory, operating system, firmware, power, clock, supply chain, and design assumptions. Failures in any shared resource can affect both the monitor and the monitored. The probability of correlated failure can be reduced by careful engineering, but cannot be eliminated by software design alone.

Three high-stakes physical domains have, by formal decision, mandated hardware-isolated safety mechanisms: aviation (DO-178C / DO-254 / ARP4754A, 1990s onward), nuclear (IEC 60880, 1986; IEC 61513), and rail (EN 50128 / EN 50129, 2001 onward). Each domain converged on this conclusion independently, after evidence that software-only approaches did not provide assurance equivalent to physical safety requirements. Physical AI exhibits the three features that drove this convergence: high-stakes physical outcomes, complex software stacks, and adversarial environments. The historical base rate predicts the same architectural conclusion.

This paper names the limit — the Software Safety Ceiling — and the underlying mechanism — fate-sharing. It addresses five common objections and finds each insufficient. The implication is direct: hardware-level safety mechanisms are necessary for Physical AI, not optional, not nice-to-have, not a future research direction.

Keywords: Physical AI Safety, software safety, hardware safety, common-cause failure, IEC 61508, functional safety, fate-sharing

---

1 Introduction

In March 2018, an autonomous test vehicle struck and killed a pedestrian in Tempe, Arizona [16]. The vehicle's perception software detected an object approximately

5.6 seconds before impact. The classification flipped repeatedly across frames — vehicle, then unknown, then bicycle, then unknown again. Path prediction reset with each reclassification. Emergency braking was suppressed by design to reduce false-positive interventions during testing. The vehicle did not stop.

The relevant fact is not that perception failed. Perception will sometimes fail. The relevant fact is this: the safety response ran on the same software stack that produced the unreliable perception. There was no independent layer that could detect that the primary control path had degraded.

This pattern generalizes. A safety monitor that runs on the same substrate as the system it monitors inherits the failure modes of that substrate. When the substrate fails, the system and its monitor fail together. The implications for Physical AI — defined here as AI systems whose outputs control physical actuators [17] — are the subject of this paper.

We argue that software-only safety has a fundamental ceiling. We do not argue that software safety is unimportant or that software cannot help. We argue that, beyond a certain assurance level, hardware-level safety mechanisms are necessary — not optional, not nice-to-have, not a future research direction. Three other domains have already accepted this. Physical AI is next.

The remainder of the paper proceeds as follows. Section 2 defines the ceiling and bounds what software safety can and cannot do. Section 3 reviews three precedents — aviation, nuclear, and rail — that have already mandated hardware-level safety. Section 4 develops the formal core: the fate-sharing problem. Section 5 argues that Physical AI exhibits the same shape as the precedent domains. Section 6 addresses five common objections. Section 7 lists implications and an open research agenda. Section 8 concludes.

This argument is not original to the present paper. It has been made implicitly by aviation, nuclear, and rail authorities for decades, and explicitly in adjacent literature on safety engineering [13, 14], including investigations of canonical software-only safety failures [15]. The contribution of this paper is to systematize the argument for Physical AI as a discipline. It also names two core concepts — the Software Safety Ceiling and fate-sharing — so they can be cited and adopted.

---

2 The Ceiling — What Software Can and Cannot Do

This section bounds what software safety mechanisms can deliver. Software is necessary. Software is not sufficient.

2.1 What software can do well

Software safety mechanisms perform six classes of task with high reliability:

Table 1. Software safety capabilities with high empirical reliability. | Capability | Description | |—|—| | Monitor | Passively observe state; detect deviations from expected behaviour | | Log | Record events for post-hoc analysis and audit | | Alert | Notify

operators or upstream systems of conditions of interest | | Retry | Handle transient failures by repeating an operation | | Fail-over | Switch execution to a backup software path on detected failure | | Detect known patterns | Match observed states against a catalogue of known failure signatures |

These capabilities address most failures most of the time. They form the bulk of any practical safety system. None of them, however, can guarantee a physical outcome under worst-case conditions.

2.2 What software cannot do

Four classes of guarantee lie outside what software-only mechanisms can provide:

Table 2. Software safety limits. | Limit | Description | |—|—| | Worst-case physical guarantee | Software cannot guarantee a physical outcome under worst-case conditions | | Substrate-source failure detection | Software cannot detect failures whose source is the monitor's own substrate | | Operation through substrate failure | Software cannot operate when the operating system misbehaves, the CPU faults, or memory corrupts | | Hardware-equivalent assurance | Software cannot provide assurance equivalent to hardware-level mechanisms |

These limits are not the result of any one team's engineering choices. They are properties of the architectural class. A recurring empirical pattern illustrates the point:

> A characteristic pattern of software-only safety: each time a known failure mode is fixed by a software patch, the next failure to occur is in a different but topologically similar location. Software safety mechanisms do not eliminate the failure surface; they shift it. Empirical evidence: large-scale software systems exhibit this pattern across decades.

The pattern has been observed in operating systems, web servers, browser engines, and industrial control software. Patching closes individual failure modes. It does not close the class.

2.3 The fundamental asymmetry

Software safety mechanisms are subject to the same physical and computational substrate as the systems they monitor. A hardware safety mechanism, by separation of fault domains, is not. This asymmetry is the core of the ceiling argument.

> The **Software Safety Ceiling** is the architectural limit beyond which software-only safety mechanisms cannot guarantee physical outcomes under worst-case conditions. The ceiling exists because software monitors share fate with the systems they monitor: they share CPU, memory, OS, firmware, power, clock, and design assumptions. Correlated failures that affect the monitored system can also affect the monitor.

Figure 1 illustrates the concept.

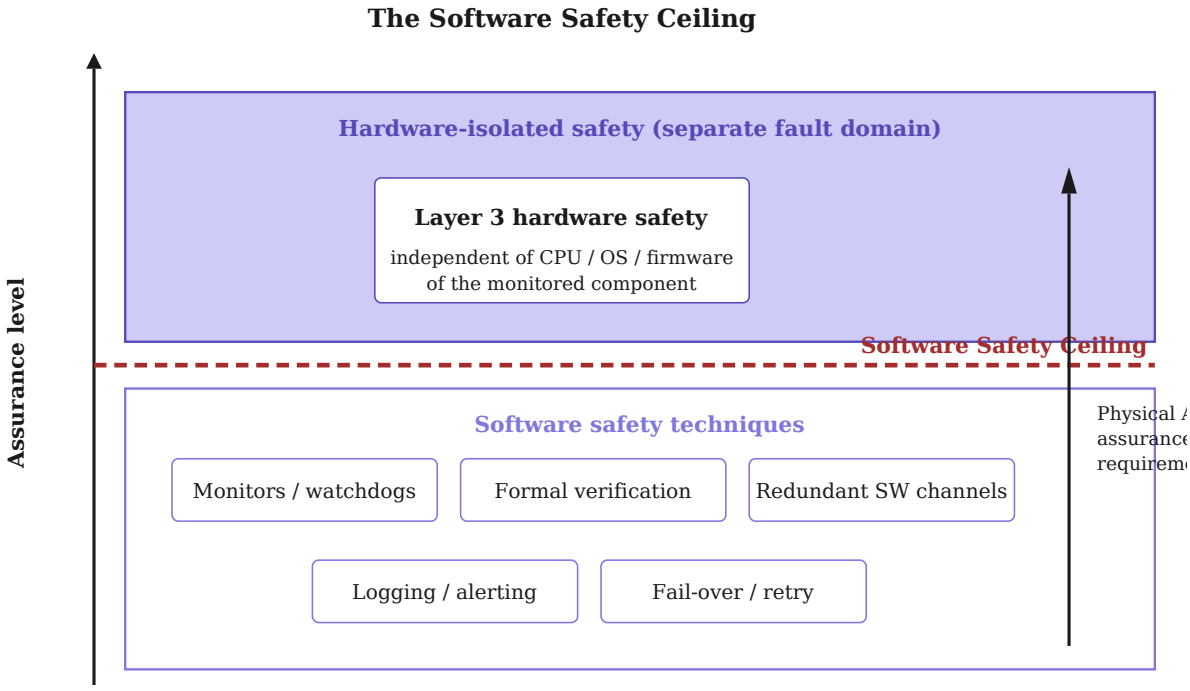

Figure 1. Conceptual ceiling diagram. Hardware-isolated safety mechanisms above a dashed ceiling line; software safety techniques (monitoring, watchdogs, formal verification, redundant channels, logging, fail-over) below. A vertical arrow shows Physical AI assurance requirements rising above the ceiling.

Figure 1. The Software Safety Ceiling (conceptual). Vertical axis: assurance level achievable. A horizontal line represents the architectural ceiling for software-only mechanisms. Below the ceiling: software safety techniques (monitoring, watchdogs, formal verification, redundant software channels). Above the ceiling: hardware-isolated safety mechanisms in a separate fault domain. Physical AI assurance requirements rise above the ceiling; the gap cannot be closed by improving the software.

3 Three Precedents

> Three high-stakes physical domains have, by formal decision, mandated hardware-isolated safety mechanisms: aviation (DO-178C / DO-254 / ARP4754A, 1990s onward), nuclear (IEC 60880, 1986; IEC 61513), and rail (EN 50128 / EN 50129, 2001 onward). In each case, the mandate followed accumulated evidence that software-only approaches did not provide assurance equivalent to physical safety requirements.

Each mandate is the result of operational evidence and formal regulatory analysis, not a priori reasoning. Each domain arrived at the same conclusion by an independent path. The historical record is summarized below.

3.1 Aviation

Civil aviation accumulated software-related anomalies through the 1980s as digital fly-by-wire entered civil service. The regulatory response was a structured framework: DO-178C for software, DO-254 for hardware, ARP4754A for system development, and ARP4761 for safety assessment [1, 2, 3, 4]. Each defines Design Assurance Levels (DAL A through E) tied to failure-condition severity.

The architectural decision relevant to the present paper is that the highest-assurance flight-critical functions (DAL A) are implemented across both software and hardware safety paths. Hardware-isolated paths exist precisely because software-only paths were not considered sufficient at the highest assurance level. The decision is documented and adopted across major civil aviation authorities, including the FAA, EASA, and Transport Canada.

The Boeing 737 MAX MCAS incidents of 2018 and 2019 illustrate this residual risk [7]. MCAS triggered on a single angle-of-attack input; the resulting failure mode was not contained by an independent path. The aviation framework anticipated this class of failure decades earlier; the 737 MAX architecture predated the framework's full application to that subsystem.

Lesson. A domain with millions of daily users and decades of operational data accepted that some safety properties cannot be guaranteed in software alone. The conclusion was formalized as mandatory hardware-isolated paths at the highest assurance levels.

3.2 Nuclear

Nuclear plant safety distinguishes operational control from protection. Operational control may be implemented in software, subject to IEC 60880 (1986; revised) and related standards [9]. Protection systems — those that shut down the reactor in an emergency — are required to meet Category A function classification under IEC 61226 / IEC 61513 [10, 12].

Software in Category A protection systems is heavily restricted. Many implementations use relay logic or simple programmable logic with formal verification. Where software is used, the protection function is implemented in hardware-isolated channels with diversity requirements. Common-cause failure analysis is a mandatory part of certification.

Lesson. In one of the highest-stakes physical safety domains humanity has built, the protection layer is hardware-implemented, not software-implemented. The choice is deliberate and architectural.

3.3 Rail

European rail safety is governed by EN 50128 (software), EN 50129 (system safety), and EN 50126 (lifecycle), with safety integrity levels SIL 1 through SIL 4 [5, 6]. Safety-critical functions in mainline rail signalling — interlockings, automatic train protection — are required to meet SIL 4.

SIL 4 software-only is not the predominant pattern; current implementations typically place the SIL 4 function in hardware-enforced logic. The pattern parallels the nuclear domain: critical protection in hardware, complex behaviour in software.

Lesson. A domain with continental-scale deployment, decades of operational data, and active regulatory scrutiny converged independently on the same architectural conclusion.

3.4 The base rate

Three independent domains. Three independent paths. One converged conclusion. The base rate of “domains that arrive at this conclusion when faced with high-stakes physical safety” is now 3/3. Figure 3 summarizes the timeline.

Figure 3. Three-domain timeline. Horizontal time axis from 1980 to 2030. Aviation track: DO-178A (1985), DO-178B (1992), DO-254 (2000), DO-178C (2011). Nuclear track: IEC 60880 (1986), IEC 61513 (2001), IEC 61513 revision (2011). Rail track: EN 50128 (2001), EN 50129 (2003), EN 50129 revision (2018). Physical AI track: a vertical marker at 2026 labelled “inflection point — open question.”

4 The Fate-Sharing Problem

This section is the formal core of the argument. It defines the mechanism that produces the ceiling.

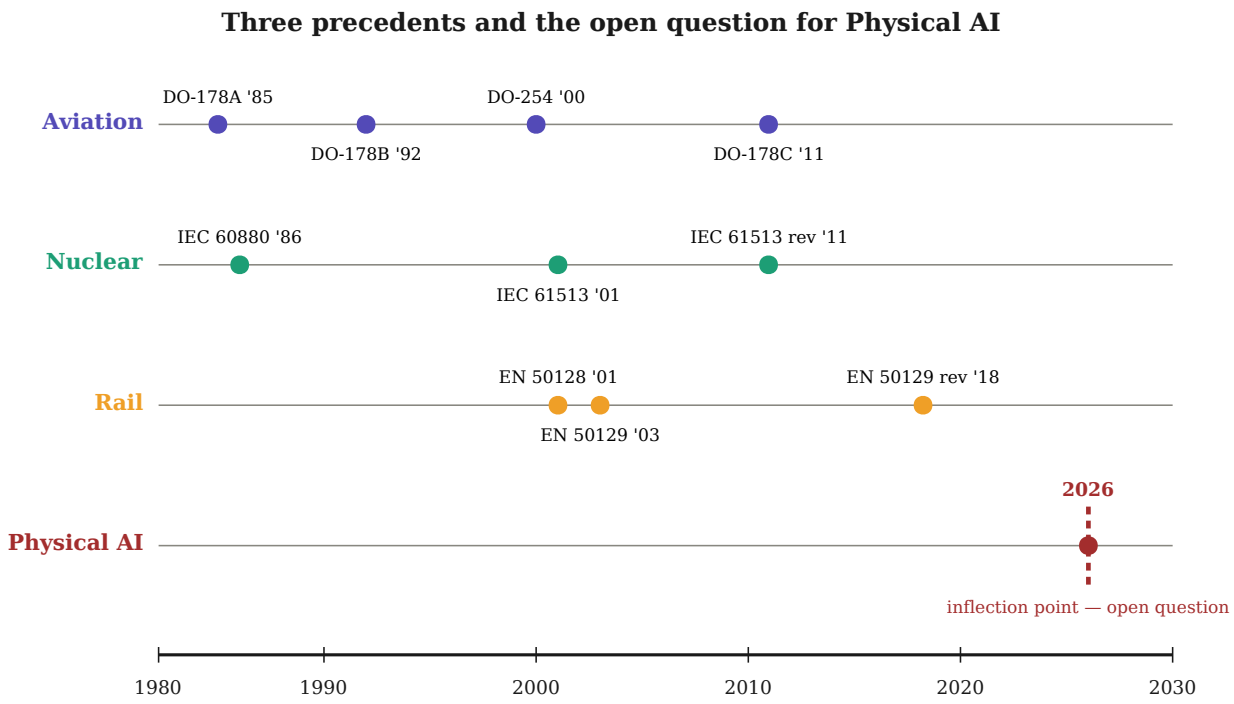

Figure 2. Horizontal timeline from 1980 to 2030 with four lanes: Aviation, Nuclear, Rail, Physical AI. Aviation lane shows DO-178A 1985, DO-178B 1992, DO-254 2000, DO-178C 2011. Nuclear lane shows IEC 60880 1986, IEC 61513 2001, IEC 61513 revision 2011. Rail lane shows EN 50128 2001, EN 50129 2003, EN 50129 revision 2018. Physical AI lane shows a single dashed marker at 2026 labelled inflection point — open question.

4.1 Identifying shared resources

> **Fate-sharing** between a safety monitor and a monitored system occurs when both depend on a shared resource whose failure can affect both. Common shared resources include: CPU, RAM, OS scheduler, firmware, power supply, clock distribution, supply chain, and design assumptions made by the same engineering team.

In any system where a software safety monitor runs on a substrate, the following resources are shared with the monitored components:

Table 3. Resources shared between a software safety monitor and the system it monitors. | Shared resource | What is shared | |—|—| | CPU | Instruction fetch and execution paths | | RAM | Data and code memory | | Operating system scheduler | Timing of execution for both monitor and monitored | | Firmware / BIOS | Boot, low-level services, microcode | | Power supply | Voltage, current, thermal envelope | | Clock distribution | Common timing reference | | Supply chain | Manufacturer, component lot, microcode revision | | Engineering team | Design assumptions and mental models |

Figure 2 illustrates the architectural pattern.

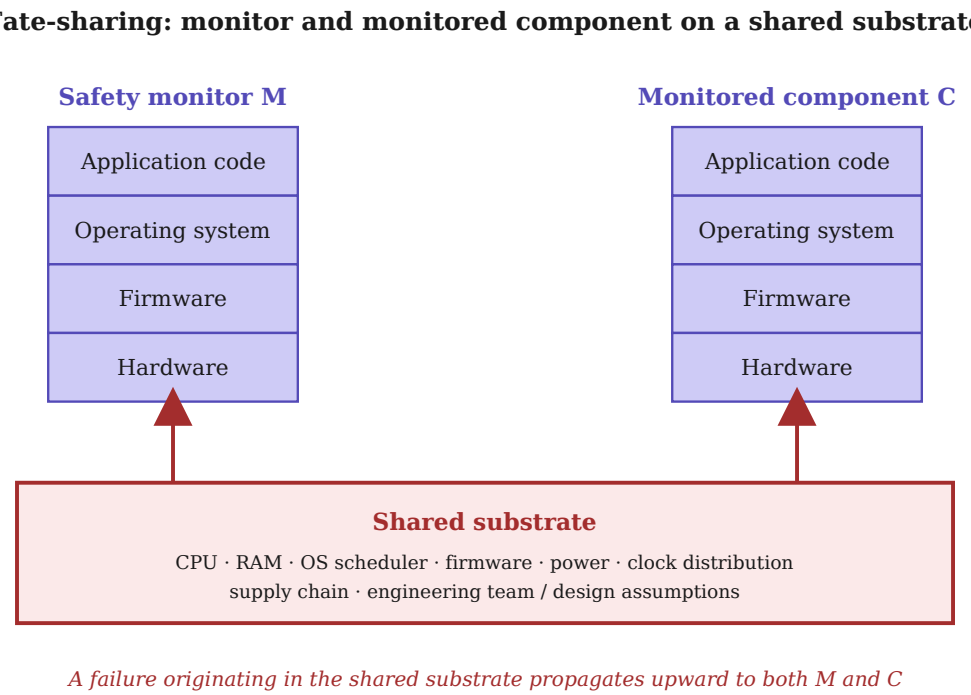

Figure 3. Two functional blocks at top labelled Safety monitor M and Monitored component C, each over identical four-layer stack of application code, operating system, firmware, and hardware. Both stacks rest on a single shared substrate (CPU, RAM, OS scheduler, firmware, power, clock distribution, supply chain, engineering team / design assumptions). Red arrows show failures originating at the substrate propagating upward to both M and C.

Figure 2. Fate-sharing diagram. Two functional blocks at the top: “Safety monitor M” and “Monitored component C.” Below each, identical stack of layers: application code, operating system, firmware, hardware (CPU, RAM, clock, power). The two stacks rest on a single shared substrate. A failure originating at the substrate propagates upward to both M and C, illustrated by parallel arrows. ### 4.2 Correlated failure modes

For each shared resource, at least one correlated failure mode is documented in the field literature.

Table 4. Correlated failure modes per shared resource. | Resource | Correlated failure mode | |—|—| | CPU | Silicon defects; thermal throttling; transient single-event upsets | | RAM | Bit flips from radiation; row-hammer effects; uncorrected errors | | Operating system | Scheduler bugs; kernel panics; priority inversion | | Firmware | BIOS errata; microcode flaws | | Power | Brownouts; voltage spikes; thermal events | | Clock | Jitter; drift; phase-locked-loop failure | | Supply chain | Lot-correlated defects; counterfeit components | | Engineering team | Same incorrect assumption made twice in independent code |

The last entry deserves emphasis. When two software channels are written by the same team or against the same specification, design errors tend to be correlated [13]. This was the central finding of the original N-version programming experiments, and has been replicated.

4.3 Why redundant software channels do not escape the ceiling

A standard objection runs as follows: “We run two software safety channels on two CPUs. They are independent.” In practice, the two channels typically share an operating system, a firmware base, and a design specification. They were written by the same team or by teams that read the same documents.

The β-coefficient framework of IEC 61508-6 Annex D quantifies this [11]. β is the fraction of failures that are common-cause (affecting both channels) rather than independent. Per IEC 61508-6 Annex D6, β-coefficients in well-engineered programmable-electronic systems range from 0.5 percent to 5 percent. Field equipment ranges from 1 percent to 10 percent. The default for unanalyzed systems is 10 percent. In practice, when redundant channels share an operating system, common-cause failures at the OS level can dominate the failure budget.

Figure 4 illustrates the β-coefficient ranges from IEC 61508-6 Annex D6 alongside the OS-correlated regime.

Figure 4. β-coefficient ranges per IEC 61508-6 Annex D6 [11]. Bar 1 (well-engineered programmable electronics): 0.5–5%. Bar 2 (typical field equipment): 1–10%. Bar 3 (unanalyzed systems, default): 10%. Bar 4 (illustrative — OS-level common-cause): can dominate the failure budget when redundant channels share OS. A detailed empirical treatment of common-cause failure for ML-plus-software channels in Physical AI is the subject of a forthcoming companion paper.

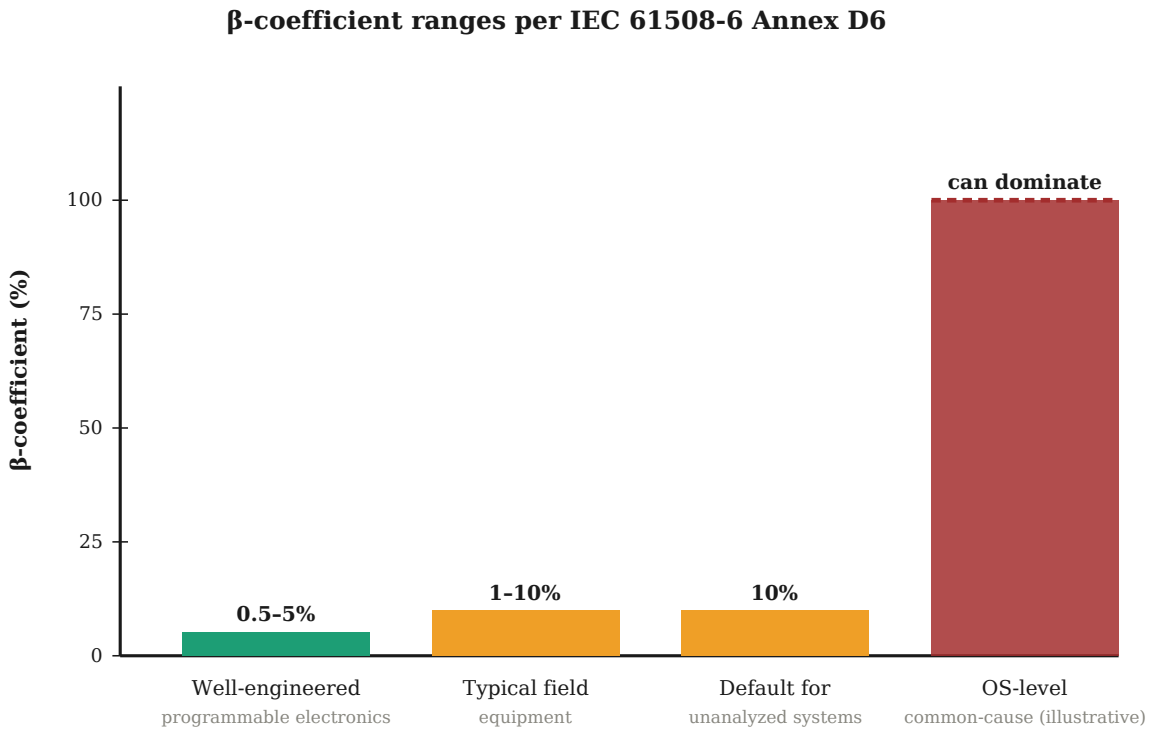

Figure 4. Bar chart of beta-coefficient ranges. Bar 1 (well-engineered programmable electronics) 0.5 to 5 percent, teal. Bar 2 (typical field equipment) 1 to 10 percent, amber. Bar 3 (default for unanalyzed systems) 10 percent, amber. Bar 4 (illustrative OS-level common-cause) full-height, red, dashed top edge to indicate it can dominate the failure budget.

4.4 The fate-sharing theorem (informal)

> **Theorem (informal):** In any system where a safety monitor M and a monitored component C share resource R, there exists a failure mode of R that affects both M and C. The probability of correlated failure can be reduced by careful engineering but cannot be eliminated by software design alone.

This is an architectural claim, not a mathematical proof. It is presented as a design principle supported by approximately three decades of cross-domain evidence, not as a formal theorem. The claim is falsifiable: a counterexample would consist of a software safety monitor that shares no resource with the system it monitors and yet remains software. No such counterexample is known to the present author.

The practical reading: where two components must remain functionally independent under failure, they must be placed in different fault domains. Different fault domains require physical separation — which is to say, hardware.

---

5 Why Physical AI Has the Same Shape

The hardware-mandate conclusion in aviation, nuclear, and rail emerged in domains exhibiting three features. Physical AI exhibits all three.

Table 5. Features that drove the hardware mandate, applied to Physical AI. | Feature | Aviation | Nuclear | Rail | Physical AI | |—|—|—|—|—| | High-stakes physical outcomes | Yes | Yes | Yes | Yes — actuators that can kill, maim, or destroy property | | Complex software stack | Yes | Yes | Yes | Yes — ML libraries, OS, drivers, planners, millions of lines | | Adversarial / unanticipated environments | Yes | Yes | Yes | Yes — open-world inputs, distributional shift, edge cases |

If a domain has all three features, the historical base rate predicts hardware safety mechanisms will be mandated. Physical AI has all three.

A counter-narrative deserves direct treatment: “Physical AI is different because of the AI part.” The argument runs that machine-learning components, being learned rather than specified, do not fit the framework of traditional safety engineering. The framework therefore does not apply to Physical AI.

The opposite is closer to the truth. Opaque, ML-based decision-making is harder to verify than rule-based control software. It is more prone to distributional shift and adversarial inputs [8]. Its failure modes are less predictable. The presence of ML strengthens the case for hardware-isolated safety mechanisms; it does not weaken it. A safety layer whose function does not depend on the correctness of the ML model can detect ML failures that the model itself cannot.

The fourth layer of the Physical AI Safety Stack [17] — Layer 3, hardware safety — is the architectural location of this mechanism. The argument of the present paper is that Layer 3 cannot be eliminated by improving Layer 2.

---

6 Objections and Rebuttals

Five objections recur in discussions of this argument. Each is addressed below.

Table 6. Common objections and rebuttals. | # | Objection | Rebuttal | |—|—|—| | 1 | Formal verification of software is sufficient. | Formal verification covers the verified properties of the program, not the substrate the program runs on. Verified software still executes on a CPU with possible silicon defects, an OS with scheduling bugs, and firmware with microcode errata. Verification is necessary at high assurance levels. It is not sufficient. | | 2 | An AI watchdog can monitor the AI. | The watchdog shares fate with the monitored AI. If the watchdog is itself an ML model, it inherits the same opacity, distributional shift, and adversarial vulnerability. If the watchdog is traditional software, it shares OS, firmware, and substrate with the AI. Either way, fate-sharing applies. | | 3 | We have multiple redundant channels. | Section 4.3 addresses this. Most claimed-redundant channels share fate. β-coefficient analysis of real systems usually reveals high correlation. Redundancy at the application layer does not produce independence at the substrate layer. A detailed empirical treatment is the subject of a forthcoming companion paper. | | 4 | This argument has been made before, and software still works. | Software does work for many things. The claim of this paper is specific: at the highest assurance levels for physical safety, software alone is insufficient. Aviation, nuclear, and rail agree, with formal mandates and decades of operational data behind the position. | | 5 | Hardware safety is too expensive or too slow. | Aviation, nuclear, and rail accepted the cost. The cost is justified by the assurance achieved. Physical AI safety incidents are also expensive — in lives, in capital, and in regulatory exposure. The cost asymmetry tends to reverse on the second incident. |

---

7 Implications and Research Agenda

7.1 Implications

Four implications follow directly from the preceding argument.

Table 7. Implications by audience. | # | Implication | Audience | |—|—|—| | 1 | Physical AI architectures should include Layer 3 (hardware safety) per the Stack defined in P1 | System architects | | 2 | Investors and customers should ask: “what is at Layer 3?” — and treat its absence as a material gap | Investors, procurement, due-diligence teams | | 3 | Standards bodies should accelerate Physical AI hardware safety standards | IEC, ISO, IEEE working groups | | 4 | Regulators should specify hardware safety in upcoming Physical AI rules | EU AI Act guidance bodies, national authorities |

7.2 Research agenda

Six open problems follow from the preceding analysis. None is solved at the time of writing.

1. **Reference architectures.** Reference designs for Physical AI hardware safety co-processors. The subject of a forthcoming companion paper.

2. **Verification methodology.** Methods to verify Physical AI safety hardware against the failure modes that ML decision-making introduces.
3. **Common-cause analysis for ML-plus-software channels.** Empirical β-coefficient measurement and modelling for the dominant Physical AI channel pattern. The subject of a forthcoming companion paper.
4. **Cost-effective hardware safety.** Economic analysis and design patterns for Physical AI safety hardware in cost-sensitive deployments such as consumer robotics and small mobile platforms.
5. **Certification methodology.** Adaptation of IEC 61508, ISO 13849, and EN 50128 conformity processes to Physical AI safety hardware.
6. **Open-source reference designs.** Open hardware reference designs for safety co-processors in research and educational settings, to widen the community of practitioners.

---

8 Conclusion

The ceiling argument summarized: software safety mechanisms share substrate with the systems they monitor. Shared substrate produces correlated failure modes. Correlated failure modes cap the assurance achievable through software alone. The cap is the Software Safety Ceiling.

The historical evidence summarized: aviation, nuclear, and rail each encountered this limit and each, by formal decision, mandated hardware-isolated safety mechanisms. The convergence is independent. The base rate is 3/3. Physical AI exhibits the three features that produced the convergence.

The implication is direct: Physical AI must accept the hardware mandate, as aviation, nuclear, and rail did before. The next robot accident will not be prevented by a better software patch. It will be prevented by the hardware safety layer that wasn't there.

References

[1] SAE International, *ARP4754A: Guidelines for Development of Civil Aircraft and Systems*, 2010.

[2] SAE International, *ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 1996.

[3] RTCA, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[4] RTCA, *DO-254: Design Assurance Guidance for Airborne Electronic Hardware*, 2000.

[5] CENELEC, *EN 50128: Railway applications — Communication, signalling and processing systems — Software for railway control and protection systems*, 2011.

[6] CENELEC, *EN 50129: Railway applications — Communication, signalling and processing systems — Safety related electronic systems for signalling*, 2018.

[7] Joint Authorities Technical Review, *Boeing 737 MAX Flight Control System: Observations, Findings, and Recommendations*. Federal Aviation Administration, 2019.

[8] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv:1412.6572*, 2014.

[9] IEC, *IEC 60880: Nuclear power plants — Instrumentation and control systems important to safety — Software aspects for computer-based systems performing category A functions*, 2006.

[10] IEC, *IEC 61226: Nuclear power plants — Instrumentation and control important to safety — Classification of instrumentation and control functions*, 2020.

[11] IEC, *IEC 61508-6: Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3*. Annex D: A methodology for quantifying the effect of hardware-related common-cause failures, 2010.

[12] IEC, *IEC 61513: Nuclear power plants — Instrumentation and control important to safety — General requirements for systems*, 2011.

[13] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 96–109, 1986.

[14] N. G. Leveson, *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley, 1995.

[15] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[16] National Transportation Safety Board, *Collision Between Vehicle Controlled by Developmental Automated Driving System and Pedestrian, Tempe, Arizona, March 18, 2018*. Highway Accident Report NTSB/HAR-19/03, 2019.

[17] M. Melchior, "Physical AI Safety: A Definitional Framework," *arXiv preprint arXiv:[ID TBD]*, 2026.

Mati Melchior · Independent Physical AI Safety Researcher mati@physical-ai-safety.com · physical-ai-safety.com P2 of an 8-paper series on Physical AI Safety.