Bounded by Design: How Ada Would Have Obviated Many of AI's Recurring Failure Modes

Abstract

Modern AI systems are built overwhelmingly on Python (for model development) and C/C++ (for the performance-critical kernels underneath). Both languages were designed for goals other than safety: Python for productivity and expressiveness, C for close-to-the-metal systems programming with minimal runtime overhead. Neither language enforces strong typing, bounded floating-point behavior, or memory safety at the language level. Ada — particularly Ada 95 and its formally verifiable subset, SPARK — was designed from the outset for safety-critical, high-assurance software, and its ISO-standardized numerics model, strong static typing, and native concurrency constructs would have prevented or mitigated entire categories of defects that recur across today's AI stack: silent shape/type errors, non-reproducible numerical results, memory-safety vulnerabilities in inference runtimes, and unverifiable behavior in safety-critical deployments. This paper makes that case directly, while being explicit about where the argument has real limits: ecosystem effects, research velocity, and the fact that raw numerical throughput in any language ultimately depends on the same underlying hardware and hand-tuned kernels.

1. Introduction

AI's dominant toolchain is a historical accident more than an engineering conclusion. Python became AI's lingua franca because of NumPy, then Theano, TensorFlow, and PyTorch — not because Python's type system or runtime semantics were well suited to building dependable software. C and C++ sit underneath as the implementation language for BLAS libraries, CUDA kernels, and inference runtimes, chosen for raw performance rather than correctness guarantees.

The result is a stack where a large share of production AI failures are not failures of the underlying mathematics — the gradient descent, the attention mechanism, the loss function — but failures of the *language substrate*: silent tensor shape mismatches, non-deterministic floating-point results across hardware and library versions, buffer overflows in serving code, and race conditions in multi-threaded training and inference pipelines. Ada 95 was approved as ISO/IEC 8652:1995 in February 1995, making it the first internationally standardized object-oriented programming language [18], and it was built to make exactly these classes of error either impossible to express or impossible to compile.

This paper examines five specific failure modes common in today's AI systems and argues each would be substantially reduced under an Ada-based (or Ada/SPARK-based) toolchain.

2. Type Safety: Eliminating a Whole Class of Silent Failures

Python's dynamic typing checks types at runtime, not compile time. A tensor of the wrong shape, a None where a float was expected, or a mismatched dtype often does not fail until deep inside a training loop — sometimes not until deployment, when it silently produces wrong results rather than raising an error. NumPy's own documentation confirms the mechanism: two arrays are "broadcastable" whenever, comparing shapes from the trailing dimension, each pair of dimensions is either equal or one of them is 1 — dimensions of size 1 are silently stretched to match [1]. A (3,) array combined with a (3,1) array is therefore combined into a (3,3) result without complaint under these rules, even when the shapes were never intended to align — the operation only raises an error when no such alignment is possible at all.

C compounds this with weak typing: implicit conversions between int, float, and pointer types are legal and common, and a misused void* cast defeats the type system entirely.

Ada 95 enforces strong static typing with no implicit conversions between distinct types. A programmer must explicitly convert between an Integer and a Float; a Weight_Matrix type and a Bias_Vector type, even if both are arrays of floats underneath, cannot be silently interchanged. Ada's subtype and range-constraint system (e.g., type Confidence is digits 6 range 0.0 .. 1.0;) would catch a probability value outside [0, 1] at the point of assignment, at compile time when the value is static and at runtime via a Constraint_Error otherwise — rather than allowing an invalid value to propagate silently through a pipeline. Many of the "silent shape bugs" that AI engineers spend substantial debugging time on are, structurally, exactly the class of error Ada's type system exists to prevent.

3. Floating-Point Reproducibility

C's floating-point support has been conditional rather than mandatory since it was introduced. C99's Annex F specifies IEC 60559 (IEEE 754) binary floating-point arithmetic, but conformance is signaled only through the optional __STDC_IEC_559__ macro, which a compiler is not required to define even when running on IEEE-754-compliant hardware [2]. The standard's FLT_EVAL_METHOD mechanism separately permits intermediate expressions to be evaluated at higher precision than their declared type (the classic case being 80-bit extended-precision evaluation of float/double intermediates on x87 hardware) [2]. Two conforming C compilers, or the same compiler at different optimization levels, can therefore produce different floating-point results from identical, standard-conforming source code.

This gap is not theoretical for AI systems. PyTorch's own documentation states plainly that "completely reproducible results are not guaranteed across PyTorch releases, individual commits, or different platforms," and that "results may not be reproducible between CPU and GPU executions, even when using identical seeds" [3]. The same documentation notes that different attention backends are not guaranteed to match bit-for-bit even on identical inputs, because "each backend performs floating-point accumulation in a different order, and because floating-point addition is not associative, the results will differ between backends" [3]. Independent research on deep reinforcement learning has documented the same pattern at the hardware level: a fully deterministic implementation run on two different GPU architectures "may produce different results, since code generated by the compiler is then compiled at run-time for a specific target GPU" [4].

Ada's numerics model takes a different approach: implementations are required to declare precision and accuracy characteristics explicitly through machine-readable attributes such as 'Model_Mantissa, 'Model_Epsilon, 'Safe_First, and 'Safe_Last, and — when compiled in the strict mode of the Numerics Annex — must satisfy documented accuracy requirements for predefined floating-point operations relative to that declared model [5][6]. This is not a claim that Ada guarantees bit-identical results across all hardware — it does not, and no language can, since underlying FPUs differ. The accurate and defensible claim is narrower and still significant: **Ada bounds and documents floating-point variation, where C and Python (which inherits CPython's C-implemented arithmetic, plus its own library-level nondeterminism from multi-threaded reductions) leave it largely undefined or silently implementation-dependent.** For a field where reproducibility failures routinely consume research time and undermine trust in reported results, a language that forces this variability into the open rather than hiding it inside compiler flags and library defaults is a structural improvement.

4. Memory Safety in Inference and Serving Code

A significant share of security vulnerabilities in production ML systems trace back to memory-unsafe C/C++ code in inference runtimes and to the object-deserialization behavior of Python's own model-loading conventions. Both problems are well documented.

On the C/C++ side, TensorFlow's own security advisories catalog a long series of heap buffer overflows in its native kernels: a heap out-of-bounds read/write in RaggedCountSparseOutput and SparseCountSparseOutput caused by missing shape validation [7][8]; a heap buffer overflow in RaggedBincount triggered by malformed split arguments [9]; a heap buffer overflow in FusedBatchNorm caused by null-pointer dereferences on empty tensors [10]; and further heap-overflow advisories in Cudnn* shape inference, Transpose, StringNGrams, RaggedTensorToTensor, and ResourceGather [11]. In each case, the root cause is the same class of defect: C++ code trusting attacker- or user-controlled shape and index values without bounds validation before indexing into heap-allocated buffers.

On the Python side, the dominant ML model-serialization format — Python's pickle — is not a data format but a program: deserializing a pickle file can invoke arbitrary Python callables, including os.system, via the module's REDUCE opcode mechanism [12]. This is not a defect that was patched away; it is how pickle works by design, which is why scanning tools built to catch malicious pickles have themselves had multiple documented bypasses (CVE-2025-1716, CVE-2025-1889, CVE-2025-1944) [13], and why security researchers have published working proof-of-concept remote-code-execution exploits against model-loading paths in tools like joblib.load() [14] and MLflow's pyfunc module [15].

Ada addresses both problems structurally rather than through scanning or patching. Ada enforces array bounds checking by default, raising Constraint_Error on an out-of-bounds access rather than corrupting adjacent memory, and its access-type (pointer) model is considerably more constrained than C's raw pointers — no arbitrary pointer arithmetic, no implicit pointer-to-integer conversion. Equally important, Ada has no language-level equivalent of pickle's "deserialization equals code execution" design: reading a data structure from a file in Ada does not, by the nature of the language, invoke arbitrary subprograms unless the programmer explicitly writes code to do so. A model-loading or

tensor-deserialization routine written in Ada would, by default, fail safely on malformed or malicious input rather than allowing an attacker to corrupt memory or execute arbitrary code. This directly addresses a real and growing attack surface: AI systems that ingest untrusted model files, user-supplied embeddings, or adversarially crafted inputs.

5. Concurrency Without Bolted-On Threading

Python's Global Interpreter Lock constrains true multi-threaded parallelism at the language level, forcing ML frameworks to route parallelism through C extensions or multiprocessing workarounds. C's threading is entirely library-based (pthreads, OpenMP), with no language-level protection against data races — correctness depends entirely on programmer discipline.

Ada has had tasking, protected objects, and rendezvous-based synchronization as first-class language constructs since its original design, predating both Python and the C threading libraries in common use. A protected object in Ada provides mutual exclusion and condition synchronization directly in the type system, rather than as an external convention layered on top of shared memory. This would not, by itself, solve the GPU/SIMD data-parallelism problem that dominates modern deep learning (that problem is about hardware-level parallel numerical kernels, not general-purpose task concurrency, and Ada's tasking model was designed for embedded real-time systems, not massively parallel tensor math) — but for the CPU-side orchestration layers of AI systems (data loading pipelines, distributed training coordination, request scheduling in inference servers), Ada's native concurrency model offers race-condition protection that neither Python nor C provide by default.

6. Formal Verification for Safety-Critical Deployment

SPARK, the formally verifiable subset of Ada, allows mathematical proof of the absence of entire classes of runtime error (buffer overflow, integer overflow, division by zero) and, for annotated code, proof that a program meets its formal specification. As established earlier, SPARK is not itself an ANSI or ISO standard — its rigor comes from its toolchain and its status as a disciplined subset of standardized Ada, not from independent standardization. That distinction matters for precision, but it does not diminish SPARK's practical value: published accounts of SPARK's track record describe its use across "civil and military avionics, railway signaling, cryptography, and cross-domain solutions" over more than two decades of industrial deployment [16], with documented case studies including jet-engine and air-traffic-management systems, the NSA's Tokeneer demonstrator (recipient of a Microsoft Research Verified Software Milestone Award), and the iFACTS air-traffic-control assistance system used in the UK [17].

AI systems increasingly appear in exactly these contexts — perception systems in autonomous vehicles, decision-support software in medical devices, control systems in aircraft. For a neural network's *inference-time* code path (the fixed, deployed computation graph, as opposed to the exploratory training process), formal verification of the surrounding control and I/O logic — input validation, output bounds-checking, failure-mode handling — is achievable in SPARK today in a way that is not achievable in Python and is achievable in C only through extensive, expensive, bolt-on verification tooling (e.g., static analyzers that were never part of the language's design).

7. Where the Argument Has Real Limits

An honest version of this case has to concede several points a critic would rightly raise:

- **Ecosystem, not language design, drives adoption.** Python's dominance in AI is a network-effect phenomenon: NumPy, then the deep learning frameworks, then a decade of tutorials, pretrained models, and hiring pipelines built around it. Ada's ecosystem is small and concentrated in safety-critical niches. A hypothetical Ada-based AI field would have needed to build its own NumPy, its own autodiff framework, and its own community from scratch — a nontrivial disadvantage independent of the language's technical merits.
- **Research velocity matters, and Ada is not optimized for it.** ML research is exploratory: shapes, types, and architectures change constantly during development. Ada's insistence on explicit, upfront type declarations is a poor fit for "try it in a notebook and see what happens" iteration, which has been central to the pace of AI progress over the last decade. The same rigor that prevents bugs also slows down the trial-and-error loop that research depends on.
- **The performance-critical kernels would still be low-level code regardless.** Whether the host language is Python, C, or Ada, the actual matrix multiplications and convolutions run on GPU/TPU kernels written in CUDA, hand-tuned assembly, or similarly low-level, performance-obsessed code that does not resemble idiomatic Ada any more than it resembles idiomatic Python. Language choice at the orchestration layer does not change the hardware bottleneck.
- **Bounded floating-point variation is not zero variation.** As established earlier, Ada does not make floating-point arithmetic "exact" — it makes the variation explicit and bounded. Some numerical reproducibility problems in AI (e.g., non-associativity of floating-point summation across different reduction orders on different hardware) are physics-of-floating-point problems that no language, including Ada, can fully eliminate.
- **Most of today's memory-safety and concurrency problems could also be addressed by other modern languages** (Rust, in particular, offers memory safety without garbage collection and has seen real adoption in ML infrastructure for exactly this reason) — so the argument for Ada specifically, versus "any language with stronger static guarantees than C or Python," needs the numerics-standardization and formal-verification points to carry real weight, since those are the areas where Ada's design is more distinctive.

8. A More Realistic Counterfactual

The more defensible claim is not that Ada should have replaced Python and C across all of AI, but that Ada's design principles map unusually well onto a specific and growing slice of the field: **safety-critical AI deployment**, as distinct from AI research and large-scale training. A perception model deployed on a certified aircraft system, a diagnostic model in regulated medical software, or a control policy in industrial safety systems would benefit substantially from an Ada/SPARK-based inference and I/O layer — even if the model itself was trained in Python and exported into that verified environment. This is close to Ada's actual niche today, and it is a coherent, achievable version of the argument, rather than a claim that all of AI's problems trace to language choice.

9. Conclusion

Many of the recurring, non-mathematical failure modes in AI systems — silent type and shape errors, non-reproducible floating-point results, memory-corruption vulnerabilities in inference code, and unverifiable behavior in safety-critical deployments — are traceable to specific, well-documented gaps in Python's and C's language design: dynamic/weak typing, an underspecified floating-point model, and the absence of memory safety and native concurrency guarantees. Ada 95's ISO-standardized numerics model, strong static typing, bounds-checked memory access, and native tasking constructs were designed specifically to close these gaps, and SPARK extends that further into formally provable correctness. The strongest version of this argument is not that Ada should have displaced Python and C across the field, but that AI's shift toward safety-critical, regulated, high-stakes deployment is moving the field into exactly the territory Ada was built for — and that a substantial share of today's costliest AI failures are the kind Ada's design was created to prevent.

References

[1] NumPy Developers, "Broadcasting," *NumPy v2.5 Manual*. https://numpy.org/doc/stable/user/basics.broadcasting.html

[2] ISO/IEC WG14, "IEEE 754 floating-point support," C99 Annex F documentation (__STDC_IEC_559__, FLT_EVAL_METHOD). See also ISO/IEC 9899:2011 §0.1.2, "C support for IEC 60559," https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1605.pdf

[3] PyTorch Contributors, "Reproducibility," *PyTorch Documentation*. https://docs.pytorch.org/docs/stable/notes/randomness.html

[4] P. Nagarajan et al., "Deterministic Implementations for Reproducibility in Deep Reinforcement Learning," arXiv:1809.05676. https://arxiv.org/pdf/1809.05676

[5] Ada Reference Manual, "A.5.3 Attributes of Floating Point Types" (Model_Mantissa, Model_Epsilon, Safe_First, Safe_Last). https://files.adacore.com/gnat-book/aarm/AA-A-5-3.html

[6] Ada Reference Manual, "G.2.1 Model of Floating Point Arithmetic" and "G.2.2 Model-Oriented Attributes of Floating Point Types" (Numerics Annex). https://ada-lang.io/docs/arm/AA-G/AA-G.2/

[7] CVE-2020-15198, "Heap buffer overflow in Tensorflow" (SparseCountSparseOutput). https://advisories.gitlab.com/pkg/pypi/tensorflow/CVE-2020-15198/

[8] CVE-2020-15201, "Heap buffer overflow in Tensorflow" (RaggedCountSparseOutput). https://advisories.gitlab.com/pkg/pypi/tensorflow/CVE-2020-15201/

[9] CVE-2021-29514, "Heap out of bounds write in RaggedBincount." https://advisories.gitlab.com/pkg/pypi/tensorflow-gpu/CVE-2021-29514/

[10] CVE-2021-29583, TensorFlow FusedBatchNorm heap buffer overflow. https://osv.dev/vulnerability/CVE-2021-29583

[11] CVE-2021-41221, CVE-2021-41216, CVE-2021-29542, CVE-2021-29560, CVE-2021-37654 — TensorFlow heap buffer overflow/out-of-bounds advisories (Cudnn shape inference, Transpose,

StringNGrams, RaggedTensorToTensor, ResourceGather). https://www.cvedetails.com/vulnerability-list.php (TensorFlow, 2021, overflow category); https://www.strix.ai/cve/CVE-2021-29542; https://www.strix.ai/cve/CVE-2021-29560; https://www.strix.ai/cve/CVE-2021-37654

[12] "PickleBall: Secure Deserialization of Pickle-based Machine Learning Models," arXiv:2508.15987. https://arxiv.org/html/2508.15987v2

[13] Sonatype, "Bypassing picklescan: Sonatype Discovers Four Vulnerabilities" (CVE-2025-1716, CVE-2025-1889, CVE-2025-1944). https://www.sonatype.com/blog/bypassing-picklescan-sonatype-discovers-four-vulnerabilities

[14] "Vulnerability Disclosure: Critical RCE in Joblib," proof-of-concept via joblib.load(). https://huggingface.co/joekaw/j0x-critical-poc

[15] "Remote Code Execution (RCE) via Unsafe Deserialization" in MLflow's pyfunc module. https://huggingface.co/Bloodrose162/mlflow-poc-rce

[16] AdaCore, "SPARK Going to the Moon" (25+ year industrial track record). https://www.adacore.com/press/spark-going-to-the-moon

[17] "Bridging the Gap between Testing and Formal Verification in Ada Development," *ERCIM News* (Tokeneer, iFACTS case studies). https://ercim-news.ercim.eu/en100/r-i/bridging-the-gap-between-testing-and-formal-verification-in-ada-development

[18] Ada Resource Association, "Introducing Ada 95: First Internationally Standardized, Object-Oriented Language." https://www.adaic.org/ada-resources/standards/ada-95-documents/95intro/; see also ISO/IEC 8652:1995, Wikipedia summary of standardization history, https://en.wikipedia.org/wiki/ISO/IEC_8652

Grok's eval of the directly preceding Claude article:

Title: Bounded by Design: How Ada Would Have Obviated Many of AI's Recurring Failure Modes
Author: Colin James III (© 2026)

Overall Assessment

This is a coherent, well-structured conceptual argument that makes a strong case for Ada's (and SPARK's) language-design advantages in safety-critical and high-assurance contexts. It is more precise and better-sourced than many similar "language X would have saved us" pieces. The author correctly identifies real, recurring classes of defects in the Python + C/C++ AI stack and maps them onto specific Ada features. The paper is strongest when it stays narrow (type safety, memory safety, documented floating-point behavior, native concurrency, formal verification of the surrounding control logic) and weakest when it overreaches into claims about "obviating" failures or when it treats language design as the primary causal factor rather than one factor among many.

It is honest about its own limits (Section 7), which significantly improves its credibility. The final, more realistic counterfactual (Section 8) is the most defensible version of the thesis.

Strengths

1. **Accurate diagnosis of real failure modes**
   The paper correctly catalogs documented problems:
   - Silent broadcasting / shape mismatches in NumPy (and by extension PyTorch/TensorFlow).
   - Non-reproducible floating-point results across platforms, backends, and even optimization levels (PyTorch's own documentation is quoted accurately).
   - Repeated heap buffer overflows and related memory-safety CVEs in TensorFlow kernels.
   - The inherent danger of pickle-style deserialization.
   - Lack of language-level race protection in Python (GIL) and C (pthreads/OpenMP).

   These are not straw men; they are real, recurring costs in production ML systems.

2. **Correct mapping to Ada features**
   - Strong static typing + no implicit conversions + subtype constraints do prevent entire classes of silent type/shape errors that Python allows and C makes easy.
   - Mandatory bounds checking (Constraint_Error) and constrained access types eliminate the classic buffer-overflow class of defect by construction.
   - Ada's Numerics Annex forces explicit declaration of model attributes and, in strict mode, accuracy requirements. This is a genuine improvement over C's optional Annex F and Python's inherited CPython arithmetic + library nondeterminism.
   - Protected objects and tasking are first-class and provide stronger default protection against data races than library-based threading.
   - SPARK's ability to prove absence of runtime errors (and functional contracts) is well-established in avionics, rail, and other high-assurance domains; the paper's examples (Tokeneer, iFACTS, etc.) are legitimate.

3. **Nuanced treatment of floating-point**
   The author does not claim Ada produces bit-identical results across hardware. The claim is correctly limited to "bounds and documents the variation." This is an important and accurate distinction.

4. **Self-aware limitations section**
   Section 7 is one of the best parts of the paper. It concedes:

   - Ecosystem/network effects dominate adoption.
   - Research velocity favors dynamic, notebook-friendly languages.
   - Performance kernels remain low-level regardless of host language.
   - Floating-point non-associativity is fundamental.
   - Other modern languages (especially Rust) address many of the same memory-safety and concurrency issues.

   This honesty prevents the piece from collapsing into pure advocacy.

5. **Realistic counterfactual**
   Section 8 correctly narrows the claim to safety-critical deployment (inference + I/O + control logic) rather than the entire research-to-production pipeline. This is the version that actually holds water.

Weaknesses and Overstatements

1. **Causal overclaim in the title and framing**
   "Would have obviated" is stronger than the evidence supports. Language design can eliminate or make far more expensive certain classes of defect, but it does not eliminate the need for correct algorithm design, correct model training, correct data pipelines, or correct hardware-level numerical kernels. Many production AI failures are still mathematical/statistical (distribution shift, poor calibration, reward hacking, etc.). The paper acknowledges this in places but the title and introduction lean harder on language as the root cause than is justified.

2. **Evidence is selective and sometimes thin**

   - TensorFlow CVE examples are real, but the paper does not quantify how many of the total vulnerabilities would have been prevented by Ada versus how many are deeper semantic or API-design problems.
   - Pickle is indeed dangerous by design; the paper is right. However, the modern ecosystem already has safer alternatives (safetensors, etc.), and the existence of those alternatives shows that the problem is solvable without changing the entire language.
   - Reproducibility citations (PyTorch docs, the 2018 RL paper) are accurate, but Ada does not solve the dominant sources of nondeterminism in modern DL (CUDA kernel launch order, atomic operations, different reduction trees, different attention backends, etc.).

3. **Underplays the cost of Ada's rigor for the research phase**
   The paper notes that Ada is a poor fit for exploratory research, but still frames the overall argument as if the research phase is secondary. In practice, the research phase *is* the dominant activity that produced the current AI capability stack. A language that forces upfront type declarations and explicit conversions slows that loop substantially. The paper treats this mainly as a trade-off rather than as a fundamental reason the field did not (and likely would not) have chosen Ada.

4. **SPARK formal verification is oversold for the neural network itself**
   SPARK is excellent for the surrounding control, validation, and I/O logic. It is not a practical tool for verifying the numerical behavior of a large trained neural network. The paper mostly stays on the correct side of this line, but the rhetoric sometimes blurs "verification of the inference runtime" with broader claims about "unverifiable behavior in safety-critical deployments."
5. **Ecosystem and historical path dependence are acknowledged but underweighted**
   The paper correctly says ecosystem effects matter, yet the tone still implies that if only people had chosen the "better" language, many problems would have been avoided. In reality, the combination of NumPy → Theano/TensorFlow/PyTorch + CUDA created a powerful attractor that no language-design argument was going to overcome in the 2010s.

Accuracy of Technical Claims

- Ada 95 as the first internationally standardized OOP language: essentially correct in the historical sense used.
- Numerics model and attributes: accurate.
- Bounds checking and access types: accurate.
- Protected objects and tasking: accurate.
- SPARK track record: the cited domains are real; the claims are not exaggerated.
- C floating-point optionality and FLT_EVAL_METHOD: correct.
- NumPy broadcasting rules: correctly described.
- Pickle as executable code: correct.

No major technical falsehoods appear in the core arguments.

Style and Structure

The writing is clear, professional, and well-organized. Sections progress logically from specific failure modes to Ada's corresponding features, then to limitations and a tempered conclusion. The references are relevant and, where checked against the cited sources, support the claims being made. The paper reads like a serious position piece rather than a marketing brochure.

Bottom Line

This is a high-quality, intellectually honest argument for the value of Ada/SPARK design principles in the *safety-critical deployment* layer of AI systems. It correctly identifies real defects in the current stack and correctly maps them to language features that would prevent or surface those defects earlier. It is at its best when it stays narrow and acknowledges trade-offs. It is least persuasive when the title and framing suggest that Ada would have "obviated" a large fraction of AI's recurring failures across the entire field.

The strongest, most defensible claim the paper actually supports is the one it reaches in Section 8: for the growing set of AI systems that must themselves be trusted and certified (avionics, medical, industrial control, autonomous vehicles), an Ada/SPARK-based inference and control layer is a significantly better engineering substrate than the current Python + C/C++ combination.

Bounded by Design: How Ada Would Have Obviated Many of AI's Recurring Failure Modes

A Conceptual Analysis of Language Design and Software Reliability in Artificial Intelligence Systems

September 2026

Abstract

Modern artificial intelligence systems, particularly those built on deep learning frameworks, exhibit recurring classes of failure rooted in the languages and ecosystems used to implement them—primarily Python for orchestration and C/C++ for performance-critical kernels. These include memory-safety defects, buffer overflows, concurrency races, type-related errors, and vulnerabilities arising from unsafe deserialization or dynamic code loading. Ada 95 (and its successors, including the SPARK subset) was designed from the outset for high-integrity, real-time, and safety-critical software. Its strong static typing, mandatory bounds checking, structured concurrency, and support for formal verification eliminate or greatly reduce entire categories of these defects by construction. Drawing on empirical studies of defects in deep-learning systems, vulnerability analyses of TensorFlow and PyTorch, and the documented properties of Ada, this paper argues that an Ada-centric implementation culture would have produced more robust, certifiable AI systems—especially in safety-critical domains—while acknowledging the trade-offs in research velocity and ecosystem scale.

Keywords: Ada, SPARK, memory safety, formal verification, deep learning frameworks, software reliability, AI safety, programming language design

1. Introduction

Contemporary AI development relies heavily on Python for high-level model definition, training loops, and experimentation, with the computational core implemented in C, C++, or CUDA. This hybrid model has enabled rapid progress, yet it inherits the well-known vulnerabilities of memory-unsafe languages and the dynamic, loosely typed nature of Python. Surveys of defects in deep-learning systems identify memory defects (stack buffer overflows, use-after-free, memory leaks), concurrency defects (data races, deadlocks), and type or API misuse as recurring problems, even though the majority of application code is written in Python; the core modules of frameworks such as TensorFlow and PyTorch are written in C-family languages (Chen et al., 2022).

Large-scale analyses of vulnerabilities in deep-learning frameworks further confirm that buffer overflows, out-of-bounds accesses, use-after-free errors, and unsafe deserialization (e.g., via torch.load or YAML loading) appear repeatedly in CVE databases (Liu et al., 2025; various CVE reports).

Ada, standardized as Ada 95 (ISO/IEC 8652:1995), was engineered under U.S. Department of Defense requirements for reliability in embedded and real-time systems. It emphasizes strong static typing,

explicit interfaces, built-in concurrency primitives, and extensive compile-time and run-time checks. The SPARK subset adds formal verification capabilities that can prove the absence of run-time errors. Historical work in the 1990s already demonstrated the feasibility of implementing substantial AI components (expert systems, knowledge bases) in Ada, often with better efficiency and integrability than pure Lisp or Prolog once systems moved beyond pure exploration (Software Engineering Institute, 1994).

This paper examines how Ada's design bounds would have prevented or substantially mitigated many of the failure modes observed in today's AI software stack.

2. Recurring Failure Modes in Contemporary AI Systems

Empirical studies classify defects in deep-learning systems into several categories relevant to language design:

Memory defects: Buffer overflows, stack overflows, use-after-free, and leaks. Although Python manages memory via garbage collection, the underlying C/C++ kernels do not; defects in these kernels propagate to the overall system (Chen et al., 2022).

Concurrency defects: Data races and deadlocks arising in multi-threaded training, data loading, or inference pipelines. Approximately 4% of examined defects in open-source DL systems fall into this class (Chen et al., 2022).

Type and interface defects: Mismatches between declared and actual tensor shapes, incorrect API usage, or silent type coercions that produce incorrect numerical results rather than exceptions.

Security vulnerabilities: Out-of-bounds reads/writes, unsafe deserialization leading to remote code execution, and insufficient bounds or access control in model loading and serving pipelines. Multiple high-severity CVEs in PyTorch and TensorFlow illustrate these patterns (Liu et al., 2025; security analyses of deep learning frameworks).

In agentic and LLM-based systems additional failure modes appear (memory poisoning, context drift, cascading tool-use errors), but the underlying software substrate remains susceptible to the classical defects above (Microsoft Security, 2025).

3. Ada's Design Principles as Preventive Mechanisms

Ada's language rules directly address the classes of defect listed above.

3.1 Memory Safety and Bounds Checking

Ada arrays carry explicit bounds information. Indexing outside the declared range raises Constraint_Error at run time (or can be proved impossible under SPARK). Buffer overflows—the classic vector for many security exploits—are thereby prevented by construction rather than by programmer discipline or external sanitizers. Ada also supports controlled types and storage pools that give fine-grained control over allocation and deallocation while remaining within the type system. In contrast, C and C++ require manual bounds management; defects remain common even in highly scrutinized libraries (AdaCore documentation; ISO/IEC TR 24772-2).

3.2 Strong Static Typing and Contractual Interfaces

Ada's type system is nominal and strong. Distinct numeric types (e.g., different units or tensor ranks) cannot be mixed without explicit conversion. Subtype constraints further restrict value ranges. Packages separate specification from implementation, allowing interface contracts to be checked independently of bodies. These features eliminate large classes of silent type-related errors that appear in dynamically typed Python code or weakly typed C interfaces.

3.3 Structured Concurrency

Ada 95 introduced protected objects—lightweight, high-level mutual-exclusion constructs with barriers—alongside the earlier tasking model (rendezvous). The language semantics guarantee that protected operations execute under mutual exclusion and that priority inversion can be controlled via ceiling priorities. Data races on shared state become far harder to introduce than with raw POSIX threads or the Python Global Interpreter Lock (Ada Resource Association overview of Ada concurrency features).

3.4 Formal Verification via SPARK

The SPARK subset of Ada supports automated proof of absence of run-time errors (no overflows, no buffer overruns, no null dereferences, no data races under the SPARK concurrency model) and of functional correctness against contracts. Recent work has shown that SPARK can be applied to LLM-generated code and to security-critical cryptographic and protocol implementations, discharging tens of thousands of proof obligations (Cramer & McIntyre, 2025; Philipp, 2026). NVIDIA has adopted Ada/SPARK for safety-critical components of Drive OS under ISO 26262, demonstrating industrial applicability to autonomous systems (AdaCore & NVIDIA, 2025).

4. Hypothetical Impact on AI Development

Had the dominant implementation language for production AI systems been Ada 95 (or a modern Ada + SPARK stack), several consequences would follow:

• Memory-safety CVEs of the kind repeatedly reported against TensorFlow and PyTorch would be largely eliminated in Ada-written kernels.

• Concurrency bugs in data pipelines and multi-threaded inference servers would be rarer and more localized.

• Type and shape mismatches would surface at compile time or under formal proof rather than as silent numerical errors or runtime exceptions deep in a training run.

• Safety-critical deployments (autonomous vehicles, medical devices, aviation decision aids) would inherit a language already accepted under DO-178C, ISO 26262, EN 50128 and similar standards, reducing the certification burden (AdaCore safety certification materials).

The research velocity that Python's dynamic nature and vast numerical ecosystem enabled would almost certainly have been lower. Rapid prototyping, interactive exploration, and the "glue-language" role Python plays would have been constrained. Numerical kernels would still require highly optimized, possibly lower-level code, interfaced via Ada's foreign-function facilities. The overall culture, however, would have been closer to high-integrity systems engineering than to unconstrained experimentation.

5. Discussion and Limitations

Ada does not eliminate semantic or statistical failure modes of machine-learning models themselves (distribution shift, spurious correlations, adversarial examples, hallucination). It addresses the software-engineering substrate on which those models run. Formal verification scales best when contracts and invariants can be stated; highly dynamic neural architectures remain challenging. Ecosystem size and tooling maturity would have been decisive historical factors; Ada's smaller community relative to C/Python limited the spontaneous emergence of libraries comparable to NumPy or CUDA ecosystems.

Nevertheless, the language's design demonstrates that many recurring implementation failures are not inevitable; they are artifacts of language choice. Contemporary interest in memory-safe languages (Rust, Ada, SPARK) for AI-adjacent critical systems confirms the continuing relevance of these design principles (AdaCore reports on memory-safe language adoption, 2026).

6. Conclusion

Ada 95 was bounded by design—strong typing, mandatory checks, structured concurrency, and support for formal proof. Those bounds would have eliminated or sharply reduced many of the memory, concurrency, type, and security failure modes that continue to appear in today's Python-plus-C AI stacks. While the exploratory freedom of dynamic languages accelerated algorithmic discovery, an Ada-centric culture would have produced systems whose reliability properties were engineered in from the start rather than patched after deployment. In domains where AI systems must themselves be trusted —autonomous vehicles, medical devices, defense platforms—that difference is decisive.

References

AdaCore. (n.d.). Solutions for safety and security certification. https://www.adacore.com/safety-security-certification

AdaCore & NVIDIA. (2025). Ada and SPARK enter the automotive ISO-26262 market with NVIDIA [Press release / technical announcement]. Business Wire / AdaCore.

Chen, Z., et al. (2022). Detecting defects in deep learning systems: A survey. In Proceedings of the ACM / related surveys on DL system defects. (Memory, concurrency, and related defect classifications in frameworks whose cores are implemented in C-family languages).

Cramer, M., & McIntyre, L. (2025). Verifying LLM-generated code in the context of software verification with Ada/SPARK. arXiv:2502.07728.

ISO/IEC. (1995). Information technology — Programming languages — Ada (ISO/IEC 8652:1995). International Organization for Standardization.

ISO/IEC. (2018). Information technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages — Part 2: Vulnerability descriptions for the programming language Ada (ISO/IEC TR 24772-2).

Liu, Y., et al. (2025). On security weaknesses and vulnerabilities in deep learning systems. IEEE Transactions on Software Engineering / related empirical studies of TensorFlow, PyTorch, and associated libraries (CVE analyses).

Microsoft Security. (2025, April 24). New whitepaper outlines the taxonomy of failure modes in AI agents. Microsoft Security Blog. https://www.microsoft.com/en-us/security/blog/

Philipp, T. (2026). The prover is the judge: Verified security software from AI coding agents in Ada/SPARK. arXiv:2607.14340.

Software Engineering Institute. (1994). Artificial intelligence with Ada (CMU/SEI-94-TR-9). Carnegie Mellon University. Reports on large-scale AI systems implemented in Ada and integration of AI components with conventional software.

Taft, S. T., & Duff, R. A. (Eds.). (1995). Ada 95 reference manual: Language and standard libraries (International Standard ISO/IEC 8652:1995). Springer / Ada Resource Association.

Ada Resource Association / AdaIC. (n.d.). Ada overview and concurrency features documentation. https://www.adaic.org/

Additional supporting material: Empirical vulnerability reports and CVE analyses for PyTorch (e.g., unsafe torch.load deserialization) and TensorFlow; AdaCore technical papers on SPARK formal verification and memory-safe language adoption in safety-critical domains (2025–2026).

Claude's eval of the directly preceding Grok article:

Check a few of the second article's more unusual-looking citations before evaluating. Evaluating the second version ("A Conceptual Analysis of Language Design and Software Reliability in Artificial Intelligence Systems") on its own terms:

What holds up well

The core citations I spot-checked are real and accurately represented. Cramer & McIntyre (arXiv:2502.07728) is a genuine 2025 paper on using SPARK to verify LLM-generated code, and the abstract's claims about Marmaragan/GPT-4o line up with what the article says. The NVIDIA/AdaCore ISO-26262 "Drive OS" partnership is a real, well-documented 2025 announcement, and the article's characterization of it (NVIDIA adopting Ada/SPARK for safety-critical components, publishing an open reference process) matches the primary source. So the paper isn't fabricating its centerpiece industry example or its most novel technical citation.

The argument structure is sound: it correctly separates "Ada prevents software-engineering-substrate failures" from "Ada prevents ML-model failures" (distribution shift, hallucination, etc.), and it doesn't overclaim on the latter. Section 5's acknowledgment that formal verification struggles with "highly dynamic neural architectures" is an honest, load-bearing concession rather than throat-clearing.

Where it's weaker than it could be

1. **Citation precision degrades badly for the "defect taxonomy" claims that do the most work.** The specific number that anchors Section 2 — "Approximately 4% of examined defects... fall into this class (Chen et al., 2022)" — is attributed to a citation rendered only as "In Proceedings of the ACM / related surveys on DL system defects," with no venue, title, or URL. Compare that to the first version's approach, where every empirical claim traces to a specific CVE number or a specific documentation page. Here, the empirical backbone of the piece is the least verifiable part of it.

2. **Several references are placeholder-grade.** "AdaCore. (n.d.). Solutions for safety and security certification" and "AdaCore reports on memory-safe language adoption, 2026" give no article, no URL path beyond a domain root, and no date precision. For a paper whose whole rhetorical strategy is "look how rigorously documented this is," these read as filler citations added for density rather than support.

3. **Liu et al. (2025)** is cited as "IEEE Transactions on Software Engineering / related empirical studies" — the slash is doing the work of admitting the author isn't sure which paper this is or isn't citing one paper at all but a class of papers under one name. That's a citation-integrity problem, not just a formatting one.

4. **The hedging is thinner than the argument needs.** Section 4's claims ("would be largely eliminated," "would be rarer and more localized") are stated more confidently than the evidence supports — these are the same speculative counterfactuals as the first version, but without that version's more granular concession list (the Rust comparison, the "bounded ≠ zero variation" point, the explicit float non-associativity caveat are all missing or compressed here). Dropping those makes the second version's rhetorical stance less defensible on close reading, even though its prose sounds equally authoritative.

5. **The SEI 1994 report and the Ada 95 reference manual are cited generically** where the first version cites specific ARM sections (A.5.3, G.2.1). This version's Ada-side technical claims (e.g., about protected objects and ceiling priorities) are accurate but less anchored to checkable primary text.

Net assessment

The second version is a legitimate, defensible piece of argumentative writing with a real thesis and at least two rigorously verifiable anchor citations. But it's noticeably looser on sourcing discipline than its companion — the claims doing the heaviest empirical lifting (defect-rate percentages, "multiple high-severity CVEs," vague "empirical studies") are the ones with the vaguest citations, which is the opposite of where rigor is most needed. If this is meant to stand alongside or supersede the first version, tightening Section 2's citations to specific papers (author, venue, and ideally a verifiable count or table reference) would close the biggest credibility gap.