



A Faithful MPI-5.0 Reference for AI-Assisted Implementation Coverage Comparison

— extended to every release of the standard, MPI-1.3 through MPI-5.x

Anthony Skjellum C. Nicole Avans

Department of Computer Science, ASCEND Center — Tennessee Technological University, Cookeville, TN, USA

EuroMPI 2026 · Vienna, Austria · 7–9 October 2026



Vault Access

Abstract

Three contributions of this work:

- ▶ a **reference vault** — a machine-readable reference representing the entire standard in markdown (.md), ready for synthesis and analysis;
- ▶ a **workflow** — an AI-co-designed but wholly deterministic, reusable pipeline that turns the standard’s own $\LaTeX$ source into a faithful reference vault, now applied to **every release with $\LaTeX$ sources**: ten vaults, MPI-1.3 through MPI-5.x, each validated against that release’s own binding appendix;
- ▶ a **prototypical study** — a static conformance and extension comparison of three common implementations, anchored to the reference and cited sources, using the vault.

The study: MPI-5.0 was ratified in June 2025; MPICH 5.0.1, Open MPI 5.0.10 and MVAPICH-Plus 5.0.0 shipped contemporaneously. A current comparison of them is useful, tedious, and was able to be automated for repeat comparisons.

Our experience: A comparison drawn from a language model risks hallucination and uneven coverage of the standard. We therefore separate the two concerns: a **deterministic reference vault**, built with no model in the trusted path, and AI assistance (for implementation comparison) confined to aspects that a researcher double checks.

Reference Vault Contents

The input is the Forum’s $\LaTeX$ sources, not the published PDF, transformed into a structured Markdown vault: one note per **chapter**, one per **API function**, and a machine-readable index of **C prototypes**. The transformation is rule-based, so it cannot invent content: a mishandled source yields a QA error, not a convincing fabrication. The same pipeline reads every release; each source tree explains itself through its own macro files.

release	routines	C	C++	F08	mpif.h	typed args	examples
MPI-1.3	129	129	—	—	129	506	77
MPI-2.0	194	187	207	—	158	642	60
MPI-2.1	306	330	329	—	298	1,041	134
MPI-2.2	313	337	340	—	305	1,084	145
MPI-3.0	404	427	—	356	364	1,534	174
MPI-3.1	414	437	—	363	371	1,567	155
MPI-4.0	483	667	—	564	415	1,938	177
MPI-4.1	507	696	—	593	439	1,990	210
MPI-5.0	535	724	—	599	445	2,002	214
MPI-5.x (draft)	539	728	—	603	449	2,010	215

C++ enters at MPI-2.0 and leaves at 3.0; Fortran 2008 enters at 3.0; the large-count `_c` variants double the C column at 4.0. MPI-1.0/1.1 exist only as PostScript.

All **196** chapter notes across the ten vaults report clean (0 errors, 0 suspect, 0 needing a macro map), and every release’s chapter-body extraction is cross-checked against its own C binding appendix: **identical prototype text for every shared routine in nine releases**; MPI-1.3 differs in three places, all in the 1993 document itself.

Findings per Release

- ▶ **The binding macros changed call form between 4.1 and 5.0:** `\mpibind{MPI_Send(args)}` is one brace group through 4.1 and two from 5.0. A tool that assumes either reads the alternate incorrectly; ours detects the form per call and takes every arity from that release’s own macro files.
- ▶ **MPI-3.0 carries 2,812 inline change marks**, some *inside* binding arguments; MPI-2.2 hides them in code listings. Both are resolved from what the macro files say each mark emits.
- ▶ **C++ came and went:** 207 C++ bindings in 2.0, 340 in 2.2, none from 3.0, yet 3.1’s sources carry 354 dead C++ lines, defined empty.
- ▶ **MPI-1.3 disagrees with itself** in three places: its annex gives `int` for `MPI_Wtime/MPI_Wtick` where the chapter says `double`, and the collective chapter spells `MPI_op_free`.
- ▶ **MPI-4.1 → 5.0, mechanically:** 28 routines added (six `MPI_Abi_*`, 22 `toint/fromint`), none removed; **3 breaking** changes, all one edit (status accessors take `const MPI_Status *`); **221** argument descriptions reworded in 138 routines. Reconciled against the 5.0 change log: 119 changed without being named (nearly all rewordings), 155 named but the interface is unchanged.

Correctness Insights (Reference Vault)

Language models are treated as the principal correctness hazard. We found that across twelve build iterations, **every consequential defect was silent content loss by a deterministic tool**. Content disappeared without any mechanical indication of an error.

- ▶ an unresolved `\input` became an empty string, erasing a **700-prototype appendix**;
- ▶ commented-out $\LaTeX$ was extracted as **six phantom routines** into the authoritative database;
- ▶ the example corpus reported a stable, healthy **“214 examples”** for the entire project while containing *no code at all*, because an optional-argument regular expression could not survive a nested bracket;
- ▶ the 5.0-tuned tool, run on MPI-4.1, left **646 bindings** in the prose and lost all **20 callback typedefs** — and the cross-check *passed*, because it compared *names*. It now compares prototype *text*.

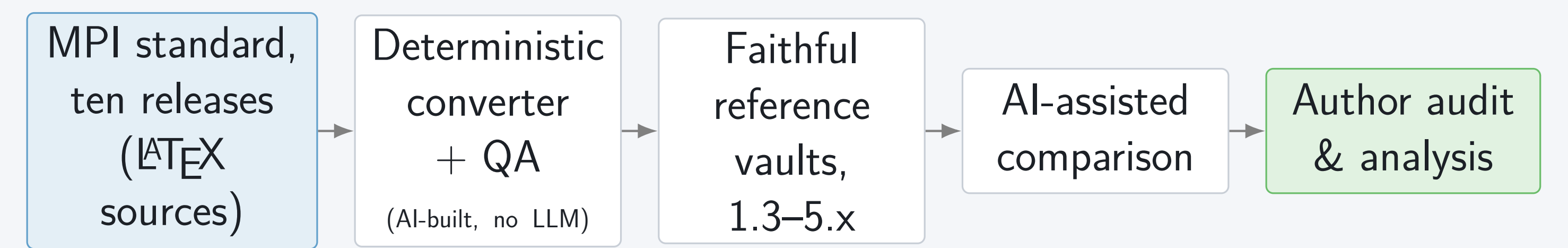
This demonstrates risks beyond hallucination. None raised an error. The mechanism that caught these was **differential cross-validation**: two independent derivations of the same fact, reconciled. It also found that MPI-5.0’s own C binding appendix omits `MPI_Wtime`.

References & Acknowledgment

References MPI Forum, *MPI: A Message-Passing Interface Standard Version 5.0* (2025) · Hammond et al., *MPI Application Binary Interface Standardization*, EuroMPI ’23 · MPI Forum, *pypistandard* · Zhou, Raffanetti, Bland, Guo, *Generating Bindings in MPICH*, arXiv:2401.16547 (2024) · Piñeiro, Vázquez, Pichel, *Towards Universal MPI Bindings for Enhanced New Language Support* (MPI4AI), J. Comput. Sci. (2025) · Li et al., *Extracting Formal Specifications from Documents Using LLMs for Test Automation*, ICPC 2025 · Nikbakht, Benzaghta, Geraci, *TSpec-LLM*, GLOBECOM Wkshps 2024 · Nichols, Davis, Xie, Rajaram, Bhatel, *Can Large Language Models Write Parallel Code?* (ParEval), HPDC ’24 · Chaturvedi, Nichols, Singh, Bhatel, *HPC-Coder-v2*, ISC 2025 · Schneider et al., *MPIrigen*, AI4Sys ’24 · Valero-Lara et al., *ChatMPI*, SCA/HPCAAsia 2026

Acknowledgment Support from the National Science Foundation (NSF) Under Grants 2450093 and 2514054 is gratefully acknowledged. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

End-to-end Tasks and Methodology



Methodology (Implementation Compliance)

LLM web-scraping proved insufficient at the function-point level, so a fully LLM-free toolchain cross-checked each library against the **535-symbol** normative set from two independent evidence sources:

- ▶ **binary / ABI side** — exported symbols via `nm -D --defined-only`
- ▶ **source / header side** — declared prototypes obtained by driving `mpi.h` through the C preprocessor, so macros, `C11_Generic` dispatch and `PMPI_` aliases are expanded by the real front end rather than guessed by regular expression.

Coverage is a *typed diff* against the normative set; disagreements between the two evidence sources are reported, not hidden. Enrolling an implementation is a one-time AI-agent step; every subsequent evaluation is deterministic.

Key Findings (Implementation Compliance)

With a faithful baseline in place, an AI assistant gathered implementation facts from primary sources and drafted a comparison of conformance and five extension families. The primary finding concerns three *non-core* capabilities: the standard API/ABI, ULFM fault tolerance, and a bundled OpenSHMEM.

	MPICH 5.0.1	Open MPI 5.0.10	MVAPICH-Plus 5.0.0
Standard MPI-5 ABI	yes (own ref.)	—	unverified
ULFM fault tolerance	experimental	yes (production)	DMTCP/SCR
Bundled OpenSHMEM	separate pkg.	yes	yes
GPU-aware	yes	yes	yes
Intel Level Zero	yes	no	yes

They differ mainly in **provenance** and **extensions**: MPICH adds the richest `MPIX_` set (stream/enqueue), MVAPICH-Plus adds ZFP compression. Fault tolerance coverage is diversified across implementations: production ULFM, experimental primitives and checkpoint/restart answer three different questions.

Related Work

Standardized, machine-readable MPI already exists, but only for bindings. The Forum’s document build produces a binding dataset, `apis.json`. While private, it is queryable through the `pypistandard` package. **Zhou et al.** generated language binding layers in MPICH on derived API data from their own $\LaTeX$ parse. **MPI4AI** analyzes MPI implementation compilers to generate bindings for additional programming languages. Extant references fail to provide significant detail or a mechanism for comparison with implementations, this contribution closes the gap.

Data extraction with LLMs fails to generate faithful references. **Li et al.** extracted formal specifications from documents with LLMs (achieving best-case 71.6% accuracy), naming two failure modes: specification *oversimplification* and specification *fabrication*. **TSpec-LLM** proposes a retrieval-augmented generation framework to identify relevant input information. Error mitigation relies on human review; in this work, a mishandled source raises a QA error instead of producing plausible but incorrect text.

LLMs for HPC code generation require validation. **ParEval** reports that models struggle *most* with MPI and provide benchmarks for evaluating the effectiveness of generated code. Various efforts have targeted parallel and MPI-specific code generation using different training strategies: **HPC-Coder-V2** trained on a synthetic dataset, **MPIrigen** trained on an HPC-specific dataset, and **ChatMPI** uses Code Llama as a base model, and used a self-instruction dataset. None have a validated, versioned rendering of the standard to check its output against; this work supplies that reference.

Future Work & Opportunities

An LLM-friendly data source representing the entire standard enables deep MPI analysis and synthesis without need to release the $\LaTeX$ source. Follow-on studies include:

- ▶ **Mechanical completeness.** Based on six checks over MPI-5. Issues discovered include `MPI_Sendrecv` having nonblocking forms but no persistent form, and 24 of the 63 error classes are never documented beyond a routine that raises them.
- ▶ **A category of types (in progress).** 189 argument kinds collapse to 33 objects, 17 of them opaque handles, over 535 routines read as typed arrows. All eleven first-class handle types carry the complete `c2f/f2c/toint/fromint` quartet; the six handles carrying none are the tool-interface handles, which never cross the language boundary. The one type carrying `c2f/f2c` but no integer conversion is `MPI_Status` (a struct, not a handle).
- ▶ **A vault along the time axis (built).** The ten vaults feed a longitudinal vault, regenerated by one command: a biography per routine (introduced, deprecated, renamed, removed; prototypes and wording by release), a history per section with word-level diffs between editions, and a routines-by-releases matrix, every entry linking to the per-release note it came from. **Next:** forward and backward compliance of implementations against every release.
- ▶ **Application of workflow to CCLs.** NCCL-like libraries may be analyzed with similar methods, yielding a dataset for deep comparison with each other and MPI.
- ▶ **Implementation or function synthesis.** Provide the reference vault as a basis for synthesizing MPI functions or an entire MPI-5.x implementation.

Disclosure of AI use

Under author direction, Claude (Opus-4.8) built the LLM-free extraction pipeline, retrieved primary sources, and created the evaluation toolchain — across 19 rounds of prompting by the first author; the extension to ten releases took one further directed session (Claude Fable 5.1). Repeat analysis of the standard is performed by the tool, without the model. No AI system is an author. The authors verified all claims.