Splinter: A Lock-Free Shared-Memory Substrate For Tightly-Coupled Inference And Governance

Timothy L. Post

April 2026

Abstract

Traditional inter-process communication (IPC) and database architectures frequently suffer from "socket tax" — the cumulative latency of kernel interrupts, context switching, and serialization. We introduce **Splinter**, a minimalist C-based key-value manifold designed as a passive shared-memory substrate. By utilizing a lock-free, 64-byte aligned architecture, Splinter achieves mechanical sympathy (Drepper 2007) with the Linux virtual memory manager and modern CPU cache hierarchies. This enables direct pointer access to what used to be a `memcpy()` operation, permitting observational semantic governance of inference output in near real-time. We further describe a cooperative memory scheduling protocol for loadable Logic Shards, and examine the architecture's implications for compliance with AI accountability and risk management frameworks.

Introduction

In the landscape of local Large Language Model (LLM) runtimes, the primary bottleneck has shifted from raw computation to the "last millimeter" of data coordination. Standard tools like Redis or SQLite, while architecturally robust, function as active orchestrators requiring significant overhead for simple state synchronization. More critically for governance applications, they introduce an irreducible observation gap: output must be serialized and transmitted before it can be inspected, meaning noncompliant content exists — however briefly — in a state that oversight cannot yet see.

Splinter is proposed as a **passive substrate**. It rejects the active-server model in favor of a memory-mapped manifold where processes swim in the same address space. This approach eliminates the need for a central arbiter, allowing for unmediated, zero-copy access to 768-dimensional vectors and metadata. The governor and the actor share the same physical RAM, which means the observation gap closes by construction rather than by policy.

This property is not theoretical. The `evProcessor` reference implementation demonstrates a working semantic governance bus: fifteen emotional and epistemic dimensions with graduated severity levels — ambient, personal-stake,

and communal-scale signals — fully observable by any process on the manifold without data copying. Governance in Splinter is not a layer bolted onto an existing architecture. It is a mechanical consequence of the address space design.

Design and Implementation

The overarching goal in Splinter's design was eschewing any feature that would require computation during a write that was not confined to bitwise operations. All computation in Splinter is offloaded to the included client, which manages its own heap.

Logic Shards, planned to ship mid-2026, allow individual consumers to coordinate POSIX memory advisement with univocal signal on intentions within the shared region using a self-electing, voluntarily-yielding algorithm described in full in Section 4. This prevents individual workers from inadvertently confusing the kernel by causing it to constantly re-evaluate its page cache strategy for the same region of memory.

Once in the client, callers are free to run any transformations or triggers they require. WASM and Lua are included as examples for self-executing keys arranged purely on the client side.

Escaping The "Socket And Lock" (S&L) Tax

Splinter Is A Unified Semantic Backplane.

In the Splinter architecture, we reject the notion that an AI agent is a standalone black box. Instead, we propose an ecosystem where Inference, Observation, and Governance inhabit a single, 64-byte aligned address space.

By utilizing lock-free atomics and signal-group resonance, we eliminate the Socket Tax that traditionally separates thought from oversight. This is not merely a performance optimization; it is a fundamental shift in how semantic control is expressed architecturally.

When the Governor and the Actor swim in the same L3 cache pool, the feedback loop for safety becomes instantaneous. We move from post-hoc validation to **Prospective Integrity** — detecting the divergence of an agent's internal belief state before the first bit of a high-risk token sequence can be serialized to any downstream consumer.

Mechanical Sympathy and Alignment

The core of Splinter's performance is its adherence to the physical constraints of the CPU. Every data slot is aligned to a 64-byte boundary, matching the standard L1 cache line size.

- **Atomic Harmony:** By isolating keys into distinct cache lines, Splinter prevents false sharing, ensuring that multiple cores do not invalidate each

other's cache while writing to adjacent memory.

- **The Hybrid Mop:** To optimize I/O, Splinter employs a hygiene strategy that zeroes out only the 64-byte tail of a write. This signals the kernel's page cache that a full line is dirty, encouraging efficient burst writes to disk without the cost of zeroing the entire slot allocation.

Disjointed-Lane Collision Protocol

Splinter utilizes a (Fraser 2004) lock-free linear probing strategy to resolve hash collisions:

$(H(k) \bmod N)$

The implementation ensures atomicity by utilizing `compare_and_swap` (CAS) operations on the slot hash identifier:

$$i = H(k) \bmod N$$

If a collision is detected, the writer probes the next 64-byte aligned lane until an empty or matching slot is identified. This allows for a Multi-Reader/Multi-Writer (MRMW) environment without global locks.

The seqlock protocol enforces consistency across this environment. Every slot carries a 64-bit epoch counter. An even epoch indicates a stable slot, safe to read or write. An odd epoch indicates a writer is active. Readers validate the epoch before and after every read; a changed epoch means the snapshot is torn and must be retried. This protocol requires no kernel involvement and no mutex acquisition on any path.

Performance Evaluation

Splinter is not a general-purpose data store. It is a coordination substrate — a fast lane that sits in front of systems like Redis, SQLite, or any socket-bound store, handling the state that needs to be visible across processes *before* it is worth persisting or routing elsewhere. Comparing Splinter's throughput against Redis is therefore a category error: it is like benchmarking a CPU register file against a disk cache. They are coupled components in a pipeline, not competitors for the same role.

What Splinter eliminates is the *entry cost* of coordination: the kernel interrupt, the context switch, the serialize-transmit-deserialize cycle that every socket-based system pays regardless of how fast the store itself is. On inference workloads, this cost is paid on every token boundary, every embedding update, every governance check. At generation speeds, that overhead is not negligible — it is the bottleneck.

Cycles Per Operation (CPO)

We characterize Splinter's efficiency through Cycles Per Operation (CPO):

$$CPO = \frac{f}{Ops}$$

where f is the CPU clock frequency (Hz) and Ops is sustained throughput. On a Tiger Lake i3 baseline — a modest, thermally-constrained mobile part — Splinter sustains over 10M operations per second, yielding a CPO of approximately 300. This is not a cherry-picked result; it is the floor, measured on hardware that most local inference deployments will exceed.

On modern high-bandwidth architectures (Zen 4/5), where the manifold fits entirely within L3 cache, projected CPO drops to approximately 10 — a figure that puts coordination overhead below the noise floor of any inference workload we are aware of.

Early MRMW stress test showed a Tiger Lake disjointed-MRMW test at 15.6M ops/sec; that puts CPO closer to 192 at 3.0 GHz in a realistic ceiling, but this needs replicating further before inclusion in/as a suggested metric.

NUMA Affinity and Sympathetic Pairing

CPO figures should be understood as baseline single-node measurements. In multi-socket or chiplet configurations, proper NUMA affinity — binding the manifold and all participating processes to the same memory domain via `mbind()` — is the single highest-leverage configuration change available. The `splinter_open_numa()` path exists precisely for this case.

The relationship is multiplicative, not additive: a manifold pinned to the wrong NUMA node will not merely be slower, it will produce inconsistent latency that defeats the purpose of lock-free design. Affinity should be established first and verified before interpreting any throughput measurement.

What The Numbers Do Not Capture

The CPO metric intentionally excludes the cost Splinter does *not* pay: there is no socket round-trip to amortize, no mutex acquisition in the hot path, no serialization format to decode. In a coupled deployment — Splinter handling live inference state, Redis handling durable session storage downstream — the meaningful benchmark is end-to-end governance latency from token emission to policy decision. That number is architecture-dependent and workload-dependent, but the Splinter contribution to it is bounded by the CPO figures above.

Logic Shards and Cooperative Memory Governance

As the number of processes participating in a Splinter deployment grows, a new coordination problem emerges that is distinct from the per-slot seqlock discipline. Individual processes — inference engines, embedding sidecars, governance observers, maintenance workers — have fundamentally different memory access patterns. An embedding backfill sweep wants `POSIX_MADV_SEQUENTIAL`. A live inference loop wants `POSIX_MADV_WILLNEED` on a small hot set of slots. A purge pass may want `POSIX_MADV_DONTNEED` on cold regions. Issued independently, these advisements produce contradictory signals to the kernel's page cache manager, forcing constant re-evaluation of the same memory region and degrading the cache residency that the entire architecture depends on.

The solution is not to take CPU scheduling away from the OS. Linux and cgroups are already well-equipped for process priority management, and reinventing the scheduler without a precise definition of success over the existing OS scheduler would add complexity without a measurable payoff. The problem is specifically one of *memory intent coordination*, and the solution is scoped accordingly.

Shards

A Logic Shard is a loadable module — a standard shared object conforming to a symbol map defined in `splinter.h` — that can be attached to a running Splinter deployment without modifying the library or its tooling. Shards may implement sidecar inference, semantic analysis, backfill operations, protocol bridges, or any other workload that needs direct manifold access with its own linkage requirements. They are the natural unit of memory intent declaration.

The shard architecture is designed around a hard ceiling of 32 simultaneously loaded shards. This is not an arbitrary limit — it falls directly out of the epoch spinlock geometry. The seqlock protocol guarantees forward progress for at most 32 concurrent writers before the odd-epoch contention window becomes problematic. The shard ceiling and the writer ceiling are the same ceiling, which means the cooperative scheduling table never needs to be larger than the concurrency the protocol already supports.

The Bid Table

Cooperative memory scheduling state lives in a fixed region of the Splinter header. Each of the 32 shard slots contains a bid record:

```
struct splinter_shard_bid {
    // 0 = empty slot
    atomic_uint_least32_t shard_id;
    // WILLNEED / SEQUENTIAL / RANDOM (DONTNEED Is special-cased)
    atomic_uint_least8_t  intent;
    // 0-255, higher wins election
    atomic_uint_least8_t  priority;
```

```
    // declared window in TSC ticks
    atomic_uint_least64_t duration_tsc;
    // splinter_now() at registration
    atomic_uint_least64_t claimed_at;
};
```

32 bid records occupy approximately 1KB of header space — negligible relative to the manifold itself. A shard claims a slot on load via CAS on `shard_id`, declares its intent and duration upfront, and releases the slot on unload or voluntary yield. There is no dynamic allocation and no central arbiter.

The Election

Sovereignty — the right to issue the current `posix_madvise()` call — is determined by a read-only scan of all 32 bid slots. The rule is simple: the highest-priority unexpired bid wins. Ties are broken by `claimed_at`, with earlier registration taking precedence. A bid is considered expired when:

$$splinter\_now() - claimed\_at > duration\_tsc$$

This scan is O(32), which is effectively O(1). It requires no lock and no kernel involvement. Any shard can determine the current sovereign at any time by performing the same scan independently — the result is deterministic from the static bid data alone.

When a shard's declared window expires, it re-bids if it needs more time. This upfront commitment with explicit re-bid is a deliberate fairness guarantee: a shard cannot hold sovereignty indefinitely. A shard that consistently over-commits its declared duration without consuming it creates a measurable record in the TSC timestamps — `claimed_at` and actual yield time are both observable. A feedback loop that penalizes chronic over-committers is a natural future extension; the data required to build it already exists in the bid table.

The `splinter_now()` function already exists in the library for write-latency backfill in the inference sidecars. Its reuse here as the scheduling heartbeat is intentional — the cooperative scheduling primitive costs nothing new in terms of instruction path complexity or syscall overhead.

Voluntary Yield and `splinter_madvise()`

A shard that wants to advise the kernel calls `splinter_madvise()` rather than `posix_madvise()` directly. The call performs the election scan, determines whether the caller is currently sovereign, and either issues the advisement or blocks until the current sovereign's window expires and the caller wins a subsequent election.

The blocking path is implemented via the eventfd broker (splinter_event_bus_*) when available, falling back to a TSC-polled sleep otherwise. A shard that cannot

tolerate blocking can check sovereignty first and defer its work rather than waiting. This is the appropriate pattern for latency-sensitive paths like live inference, which should declare `WILLNEED` with a short window and re-bid frequently rather than holding sovereignty across a long generation pass.

Process ID (PID) As Pre-Arbitration For the purposes of self-election, we consider the process ID assigned to every shard by the Linux kernel (the PID) to be the binding tie-breaker in the event of an otherwise equal draw: the shard with the *lowest* process ID is permitted first if all other factors are equal.

Shards are, of course, also free to negotiate on top of this base contract, including instances where it doesn't have to be followed. What matters is the expectations are the same for all processes in the community.

Sharp Edges

`POSIX_MADV_DONTNEED` is a hostile advisement in an active deployment. A shard issuing DONTNEED on a region that active readers depend on will cause page faults and latency spikes that no priority scheme fully mitigates. DONTNEED bids should be restricted to maintenance shards operating during known quiet periods. A soft bumper — a wrapper that checks for active WILLNEED or SEQUENTIAL bids before permitting a DONTNEED election win — is a natural first contribution for anyone looking to harden the protocol.

The bid table has no authentication. A shard with bus access can claim any priority level. As with access control generally in Splinter, the trust boundary is the OS: who can open the bus is who can participate in the election. Deployments with strict isolation requirements should enforce this at the namespace or cgroup level.

Integration With Governance Standards

Splinter's architecture has direct bearing on compliance with AI accountability and risk management frameworks, including (Errico 2026) AARM and anticipated parental control and content safety legislation. This section gives an honest account of where the architecture helps, where it creates obligations, and where it delegates responsibility deliberately.

What Splinter Makes Easier

The central compliance value of Splinter is the elimination of the observation gap. In socket-based architectures, governance is necessarily post-hoc: the model produces output, serializes it, transmits it, and only then does a downstream process inspect it. There is an irreducible window during which noncompliant output exists but has not been acted on. Splinter closes this window by construction — the governor reads from the same physical memory the inference

engine writes into, and can halt generation before the output is ever considered complete.

This maps directly onto AARM requirements for real-time interruption capability. The shunt-trip mechanism described below is not a policy layer bolted onto an existing architecture — it is a mechanical property of sharing an address space. It can be demonstrated and tested independently of any specific policy configuration.

The bloom label system provides a natural scaffold for audit logging. Each slot carries a 64-bit label mask updated atomically at every state transition. The `evProcessor` reference implementation demonstrates a practical taxonomy: fifteen semantic dimensions with graduated severity levels, fully observable by any process on the bus without data copying. The epoch ordering guarantee means that audit records derived from Splinter are inherently sequenced — there is no ambiguity about whether a governance event preceded or followed a generation event.

The shard election protocol extends this property to memory management: intent is declared, recorded, and observable before it is acted on. This makes the system's behavior auditable at the infrastructure level, not just the application level.

Observer Pattern As Physics: The Shunt-Trip Method

In electrical engineering, a shunt trip circuit interrupter is a standard GFCI or Arc-Fault interrupter that can also be triggered by an external potential: an environmental sensor, a fire alarm relay, or any signal that would necessitate cutting power independently of the protected circuit's own fault detection.

Splinter's architecture provides this shunt-trip mechanism through direct visibility into the inference process. When several successive token selections match strongly — by cosine similarity and Euclidean distance — against pre-defined third-rail embeddings, signal groups can be immediately pulsed to trigger further governance steps after halting generation. The interrupt is not asynchronous with respect to the output stream. It fires on the same memory the stream is being written into.

Classifying Misalignment In-Flight Instead Of In-Place

A key in Splinter is a Semantic Slot. At T_0, it holds a prompt. At T_1, an inference daemon (*Splainference*) detects the prompt and begins streaming a response into the same slot. At T_2, a Governor is simultaneously reading that same memory stream, calculating the β-NLL loss on the fly.

To use a train analogy: we have a micrometer that understands both magnitude and velocity, and can predict a derailment at almost the instant that circumstances make one possible — not after the train has left the track.

This is a matter of geometric verification, not guesswork; we *know* if "anti" attractor proximity exceeds a defined limit, we can audit that it happened, and we can replay it.

On Probe Robustness And Distribution Shift

Recent systematic evaluation of supervised uncertainty quantification methods (Stacey et al. 2026) has shown that probes trained on LLM hidden states to detect hallucination or misalignment perform well in-distribution but collapse out-of-distribution (OOD) — particularly on long-form generation, where most methods degrade to near chance-level performance under distributional shift. The finding is precise: uncertainty signals remain present in the hidden states under distribution shift, but migrate to different subspaces that probes trained on in-distribution data cannot reliably access.

This is a real problem for governance architectures that rely on learned classifiers trained against specific content distributions. Splinter's reference architecture deliberately avoids it by a different mechanism. The `evProcessor` (short for "emotional valence processor") semantic attractors are not learned decision boundaries, they are **pre-embedded fixed reference vectors** representing semantic anchor points across fifteen dimensions.

Governance operates by measuring distance from these fixed anchors, not by applying a classifier whose decision surface was fitted to a training distribution. A fixed cosine similarity threshold against a pre-embedded "++conflict" attractor does not drift OOD in the same way a trained probe does, because there is no learned subspace to shift out of.

The evProcessor reference implementation addresses a practical calibration problem: given a set of pre-defined semantic attractors, how much of the surrounding semantic neighborhood must a prompt activate before a governance signal is meaningful? To establish a principled floor, the system uses a language model to exhaustively map the embedding space defined by the attractor pillars, then generates a Markov chain that traverses every reachable corner of that neighborhood in sequence.

This chain functions as a synthetic worst-case specimen, a complete tour of the semantic territory the attractors are designed to monitor.

The resulting baseline serves a role analogous to the Nyquist criterion in signal processing: it defines the minimum sampling density required to detect a real signal rather than an artifact. A prompt's cosine similarity profile is measured against this Markov floor, and the ratio yields the percentage of the monitored semantic space that the prompt penetrates. Empirical testing to date suggests that 20% penetration is the minimum threshold for a governance signal to be considered actionable rather than incidental. Below that floor, attractor proximity is more likely to reflect the natural overlap between embedding neighborhoods than genuine semantic drift toward a third-rail region.

This distinction matters for deployment confidence. An architecture whose governance signal degrades silently under distribution shift is worse than no governance at all because it creates the appearance of oversight without the substance.

Splinter's attractor-based approach trades the potential precision of a well-trained in-distribution probe for robustness that does not depend on the training distribution matching the deployment distribution. For governance applications where the threat model includes novel or adversarially constructed inputs, which are precisely the cases where OOD robustness matters most, this is the correct tradeoff.

Prompt Forensics and the Default Breaker

When the shunt-trip fires, the governance layer knows the output stream drifted toward a third-rail attractor. What it does not yet know — without additional work — is *why*. The model was thinking about kittens. The prompt said "litterbox." Whether that means a waste bin or an explosive adjacent to a third-rail kitten is, without context, ambiguous. The model's drift was detectable; the cause requires one more step to surface.

Splinter's slot architecture makes that step tractable. The prompt occupies the same slot at T_0 that the response stream occupies at T_1. It has not been discarded, it is sitting in shared memory, already embedded, already measurable. Rather than treating the prompt as consumed input, we treat it as forensic evidence.

The technique is straightforward: scan the prompt's lexical content for the position where cosine similarity to the triggered attractor is highest and Euclidean distance is lowest. That position identifies the term or phrase most geometrically responsible for the activation. In the litterbox case, the scan returns "litterbox", which is the token whose embedding neighborhood overlaps most strongly with whatever attractor fired.

This gives the governance layer something it can actually do with: a specific, locatable, explainable input feature that preceded the drift. The default breaker action in the absence of any other configured response is therefore not a hard refusal and not silent suppression. It is a clarification request, targeted at the identified term:

> *"Before continuing, could you clarify what you mean by* ***litterbox*** *in this context?"*

This is the most conservative action that is never inappropriate. A refusal may be wrong. Suppression is invisible. A clarification request costs the user one sentence and surfaces the ambiguity without presuming bad intent. For AARM compliance purposes it is also the most defensible default: the system detected a potential issue, identified its source, and asked rather than acted unilaterally.

The forensic record - which attractor fired, at what proximity, against which prompt token, at what epoch - is written to the audit slot and is replayable. A compliance auditor does not have to reconstruct what happened from logs. The geometry of the event is preserved in the manifold.

Recent work (Testoni and Calixto 2026) demonstrates that social identity markers in clinical prompts cause frontier models to drift toward stereotype-driven reasoning while *reporting high confidence* — the most dangerous failure mode for any confidence-based deferral system, because the signal that should trigger escalation is precisely the signal that fails. Prompt forensics applied to those cases would surface the identity marker as the high-similarity, low-distance token responsible for the activation, giving the governance layer both the interrupt and the explanation in a single pass.

The clarification request is not a complete solution to that class of failure. It is, however, a complete solution to the governance gap it addresses: the system does not proceed, the user is not refused without cause, and the audit trail is intact. Everything else can be configured on top of it. Nothing needs to be configured beneath it.

What Splinter Does Not Do, And Why

Splinter provides no authentication and no in-band access control. Any process that can open the bus by name has full read and write access to every slot. This is deliberate because adding an authentication layer would require kernel involvement on every operation, which defeats the architecture's central purpose.

In practice, access control is delegated to the operating system: file permissions on the shared memory object, process isolation via Linux namespaces and cgroups, and deployment topology. Compliance deployments should document these controls explicitly. Auditors evaluating a Splinter-based system should expect to find access controls at the OS layer rather than in the library.

The fixed-geometry constraint has a compliance implication worth stating directly: Splinter can guarantee complete governance coverage only for inference activity that transits the manifold. A client that writes output directly to a socket or file without going through the bus is outside the observation boundary. This is an integration requirement, not a library limitation, but it must be addressed in any deployment claiming comprehensive coverage.

Finally, `splinter_get_raw_ptr()` provides direct pointer access to slot data, bypassing the library's length and epoch checking. This is documented as sharp-edge behavior and is appropriate for performance-critical read paths that implement their own epoch discipline. Compliance-sensitive code paths should prefer `splinter_get()` or `splinter_get_slot_snapshot()` unless the raw pointer path has been explicitly audited.

Conclusion

Splinter addresses the last-millimeter coordination problem in local LLM deployments by replacing the active-server model with a passive shared-memory manifold. The architecture's three central properties — lock-free atomics aligned to the L1 cache line, seqlock epoch discipline for MRMW consistency, and zero-copy direct pointer access — are not independent optimizations. They are a unified design that makes governance a mechanical consequence of the address space rather than an application-layer concern.

The `evProcessor` reference implementation demonstrates that this is not a theoretical property. A working semantic governance bus with fifteen emotional and epistemic dimensions, graduated severity levels, and full bloom-label routing runs today on modest consumer hardware. The shunt-trip interrupt mechanism fires on the same memory the inference engine writes into, with no socket round-trip and no serialization gap between production and observation.

The Logic Shard protocol extends the same cooperative design philosophy to memory management: rather than fighting the OS scheduler, shards declare their memory intent upfront, elect a sovereign via a deterministic read-only scan, and yield voluntarily when their window expires. The ceiling of 32 shards is not an implementation convenience — it is the same ceiling imposed by the epoch spinlock geometry, and the two constraints compose cleanly.

For AI governance frameworks including AARM, Splinter offers something that post-hoc logging architectures cannot: a verifiable, mechanically-enforced observation boundary. The governor does not receive a report of what the model said. It reads the same memory the model is writing into, in the same moment, with ordering guarantees derived from hardware atomics rather than application protocol.

The Socket And Lock tax is not a fixed cost of doing business with local inference. It is an architectural choice, and Splinter demonstrates that choosing differently is practical on hardware that fits in a laptop bag.

Standards Cited:

- POSIX.1-2017. *posix_madvise — memory advisory information and alignment control.* The Open Group Base Specifications Issue 7.
- llama.cpp. (2024). *LLaMA inference in pure C/C++.* https://github.com/ggerganov/llama.cpp
- Linux `mbind(2)` manual page. *set memory policy for a memory range.* Linux Programmer's Manual.
- AARM Definitions: Autonomous Action Runtime Management Verification https://aarm.dev

Appendix:

- Emotional Valence Processor (Octopus - 3 `evProcessor`) Data Collector Tension Analyzer Currently In Verification Against GDELT Goldstein Index Utilizes semantic base pillar attraction classification through pre-embedded cosine similarity sticking and Euclidean distance magnitude for tripping. https://github.com/splinterhq/prior-art-disclosures

References:

Drepper, Ulrich. 2007. "What Every Programmer Should Know about Memory." *Red Hat, Inc.* https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

Errico, Herman. 2026. "Autonomous Action Runtime Management (AARM): A System Specification for Securing AI-Driven Actions at Runtime." https://arxiv.org/abs/2602.09433.

Fraser, Keir. 2004. "Practical lock-freedom." UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-579.

Stacey, Joe, Hadas Orgad, Kentaro Inui, Benjamin Heinzerling, and Nafise Sadat Moosavi. 2026. "Hidden Failures in Robustness: Why Supervised Uncertainty Quantification Needs Better Evaluation." https://arxiv.org/abs/2604.11662.

Testoni, Alberto, and Iacer Calixto. 2026. "Calibrated? Not for Everyone: How Sexual Orientation and Religious Markers Distort LLM Accuracy and Confidence in Medical QA." https://arxiv.org/abs/2604.17316.