Implementation Examples and Parameter Guidance

Reference Implementations in C and Python

Alexandre Sah*

January 30, 2026

Abstract

This online resource provides reference implementations of Algorithmic Hysteresis Primacy (AHP), demonstrating that mathematical guarantees (particularly $\Delta T_{\min} > 0$) are architecturally enforceable with $O(1)$ complexity. We present: (1) a minimal C implementation for embedded/real-time systems; (2) Python simulation scaffolding; (3) domain-specific calibration guidance; (4) formal verification checklists; and (5) an empirical validation roadmap with publicly accessible datasets. These implementations provide runtime foundations for protocol headers, distributed governance mechanisms, and radiation-aware computing validation.

Keywords: Reference implementation, Embedded systems, Real-time control, Parameter calibration, Formal verification, Empirical validation

Contents

1 Introduction: Executable Specifications **2**

2 Minimal C Implementation (Reference Architecture) **2**
- 2.1 Core Logic . . . 2
- 2.2 Complexity Analysis . . . 3
- 2.3 Determinism Guarantees . . . 4

3 Parameter Calibration **4**
- 3.1 Design Procedure . . . 4
- 3.2 Domain-Specific Parameters . . . 4

4 Python Simulation Framework **4**

5 Verification Checklist **5**

6 Empirical Validation Roadmap **5**

7 Connection to Protocol Layer and Governance **6**

8 Conclusion **6**

*This document provides executable implementations validating the mathematical guarantees established in supplementary materials and enabling the protocol specifications detailed in accompanying technical documents. The main article *Algorithmic Hysteresis Primacy (AHP): Temporal Sovereignty in AI Governance* presents the complete conceptual framework and is available at zmem.org and SSRN.

1 Introduction: Executable Specifications

While mathematical formalizations prove that AHP provides guarantees (minimum hesitation $\Delta T_{\min}$, stability, noise rejection), this document demonstrates that such guarantees are *implementable* with minimal computational overhead. The code presented herein operationalizes the Non-Zeno Guarantee, enforcing architectural constraints at the implementation level.

Throughout this document, we use the term "reference implementation" to denote executable specifications whose primary purpose is architectural validation rather than performance optimization.

These implementations serve four purposes:

1. **Reference**: Canonical implementations for porting to specific platforms
2. **Verification**: Testbeds for validating parameter choices against historical failure scenarios
3. **Protocol Foundation**: Runtime support for HTTP headers specified in supplementary materials
4. **Distributed Governance**: Deterministic execution guarantees ($O(1)$ complexity, data-independent WCET) required for Byzantine consensus protocols analyzing multi-jurisdictional coordination

Remark 1.1 (Implementation as Governance Infrastructure)**.** The determinism guarantees (Section 2.3) are not merely performance optimizations but *architectural necessities* for distributed governance. In Byzantine consensus scenarios, heterogeneous nodes (representing different jurisdictions) must execute the AHP logic with *identical temporal guarantees*. The reference implementation provided herein ensures that $\Delta T_{\min}$ is enforced identically across ARM Cortex-M, x86-64, and RISC-V platforms, preventing consensus failures due to timing divergence.

2 Minimal C Implementation (Reference Architecture)

2.1 Core Logic

The following 42-line C implementation proves the Architectural Non-Zeno Property is enforceable. It maintains the minimal state required for AHP compliance.

```c
/* ============================================
 * ALGORITHMIC HYSTERESIS PRIMACY - CORE LOGIC
 * Implements Non-Zeno Guarantee
 * ============================================ */

#include <stdint.h>
#include <stdlib.h>

typedef struct {
    uint16_t I;          /* Accumulator [0, I_max] */
    uint8_t D;           /* Decision: 0=WAIT, 1=ACT */
    uint16_t I_max;      /* Saturation ceiling */
    uint16_t gamma_min;  /* Threshold: release (Wait) */
    uint16_t gamma_max;  /* Threshold: commit (Act) */
    uint16_t phi_max;    /* Governance cap Phi_max */
} AHP_State;

/**
 * @brief AHP Update Function - O(1) complexity
 * @param state Persistent accumulator state
 * @param epsilon Evidence input (error signal)
```

```c
 * @return New decision state D(t)
 *
 * Architectural Guarantees:
 * 1. Enforces dI/dt <= Phi_max (Governance Cap)
 * 2. Hysteresis band traversal >= Delta_T_min
 * 3. Clamp saturation prevents overflow
 */
uint8_t ahp_update(AHP_State* state, int16_t epsilon) {
    /* 1. BOUND ACTIVATION (Governance Cap) */
    uint16_t phi = (uint16_t)abs(epsilon);
    if (phi > state->phi_max) phi = state->phi_max;

    /* 2. UPDATE ACCUMULATOR */
    uint32_t temp = (uint32_t)state->I + phi;
    if (temp > state->I_max) {
        state->I = state->I_max;  /* Upper saturation */
    } else {
        state->I = (uint16_t)temp;
    }

    /* 3. HYSTERETIC DECISION */
    if (state->I >= state->gamma_max) {
        state->I = state->gamma_max;  /* Clamp to threshold */
        state->D = 1;                 /* COMMIT */
    }
    else if (state->I <= state->gamma_min) {
        state->I = state->gamma_min;  /* Clamp to threshold */
        state->D = 0;                 /* WAIT */
    }
    /* Else: Maintain D(t^-) -- Hold state (hysteresis) */

    return state->D;
}

/*
 * Non-Zeno Verification:
 * DT_min = (gamma_max - gamma_min) / phi_max > 0
 * Enforced by: (a) clamping logic above,
 * (b) strict gamma_max > gamma_min requirement,
 * (c) phi_max > 0 architectural constant.
 */
```

Listing 1: AHP Core Logic (C99) - Implementing Non-Zeno Guarantee

2.2 Complexity Analysis

Table 1: Computational Complexity of Listing 1

Operation	**Count**	**WCET (typical)**
Absolute value	1	1 cycle
Comparison/branch	3	1–3 cycles each
Addition (32-bit)	1	1 cycle
Memory access	4	2–4 cycles each
Total	$\sim$10 operations	$<$50 cycles

On a 100MHz embedded processor, one update requires $<0.5\mu$s—orders of magnitude faster than the $\Delta T_{\min}$ time constants (20ms–2000ms) specified for application domains.

2.3 Determinism Guarantees

The implementation in Listing 1 satisfies hard real-time requirements essential for distributed consensus:

- **No loops**: All paths bounded (WCET analysis feasible)
- **No recursion**: Flat call structure (stack depth constant)
- **No dynamic allocation**: Static memory only (heap fragmentation impossible)
- **Data-independent paths**: All branches execute in similar cycle counts (no timing side channels, critical for Byzantine fault tolerance)

3 Parameter Calibration

3.1 Design Procedure

Implementing AHP requires calibrating three parameters: $\Gamma_{\min}$, $\Gamma_{\max}$ (thresholds), and $\Phi_{\max}$ (governance cap). The following procedure connects domain requirements to register values:

1. **Specify** $\Delta T_{\min}$: Choose from domain requirements (Table **??**)
2. **Set hysteresis ratio**: Typically $\Gamma_{\max}/\Gamma_{\min} \in [2, 5]$
3. **Compute** $\Phi_{\max}$ (Governance Cap):

$$\Phi_{\max} = \frac{\Gamma_{\max} - \Gamma_{\min}}{\Delta T_{\min}} \quad \text{[evidence units per ms]} \tag{1}$$

4. **Verify noise**: Ensure $\Delta T_{\min} \gg \tau_{\text{noise}}$ (noise correlation time)

3.2 Domain-Specific Parameters

Domain-specific parameterizations are summarized in Table 3, which operationalizes the empirical validation roadmap described in the main article.

4 Python Simulation Framework

For rapid prototyping and parameter validation before C deployment:

```python
import time
from dataclasses import dataclass
from typing import Literal

@dataclass
class AHPSimulator:
    """
    Executable specification of AHP dynamics.
    Validates Non-Zeno property via simulation.
    """
    gamma_min: float     # Threshold Wait->Accumulating
    gamma_max: float     # Threshold Accumulating->Act
    phi_max: float       # Governance cap (evidence/sec)
    dt: float = 0.001    # Simulation timestep (1ms)

    # State variables
    I: float = 0.0       # Accumulator
    D: Literal[0, 1] = 0  # Decision state
```

```
19
20      def update(self, evidence: float) -> int:
21          """
22          Single update step.
23          Evidence: instantaneous error signal (unbounded input)
24          Returns: current decision state
25          """
26          # 1. Apply governance cap (Phi_max bound)
27          phi = min(abs(evidence), self.phi_max) * self.dt
28
29          # 2. Accumulate with saturation
30          self.I = min(self.I + phi, 1.0)  # Normalized I_max=1.0
31
32          # 3. Hysteretic switching
33          if self.I >= self.gamma_max:
34              self.I = self.gamma_max
35              self.D = 1
36          elif self.I <= self.gamma_min:
37              self.I = self.gamma_min
38              self.D = 0
39          # Else: hold previous state (hysteresis)
40
41          return self.D
42
43      @property
44      def gamma(self) -> float:
45          """Normalized conviction (for protocol headers)"""
46          return self.I / self.gamma_max if self.gamma_max > 0 else 0
47
48  # Example: Flash crash prevention simulation
49  def simulate_flash_crash():
50      # DT_min = (0.8 - 0.2) / 12.0 = 0.05s = 50ms
51      ahp = AHPSimulator(gamma_min=0.2, gamma_max=0.8, phi_max=12.0)
52
53      # Simulate high-frequency noise (microsecond spikes)
54      import random
55      for t in range(1000):  # 1 second simulation
56          noise = random.gauss(0, 100) if t % 10 == 0 else 0  # 10% noise
57          decision = ahp.update(evidence=noise)
58          if decision == 1:
59              print(f"Commit at t={t}ms, conviction={ahp.gamma:.2f}")
60              break
61
62  if __name__ == "__main__":
63      simulate_flash_crash()
```

Listing 2: AHP Simulator (Python 3) - Validating Non-Zeno Guarantee

5 Verification Checklist

To ensure an implementation respects architectural guarantees:

6 Empirical Validation Roadmap

While mathematical formalizations establish formal guarantees and the accompanying article outlines the strategic vision, this section provides concrete technical specifications for domain-specific implementation. Table 3 operationalizes the validation strategy with explicit data sources, temporal resolutions, AHP parameters, implementation tools, and success metrics.

Table 2: Formal Verification Checklist

Invariant	**Verification Method**	☐
$\Phi(\varepsilon) \leq \Phi_{\max}$	Static analysis: No paths bypass input capping before accumulation	☐
$\Gamma_{\max} > \Gamma_{\min}$	Unit test: Assert(gamma_max > gamma_min) on initialization	☐
$\Delta T_{\min} > 0$ enforced	Timing analysis: Verify no transition path completes in <50 cycles (WCET)	☐
Clamping to thresholds	Boundary test: Verify I clamps exactly to $\Gamma_{\max\,/\,\min}$ on transition	☐
Monotonicity in $\mathcal{H}$	Formal proof: Show $dI/dt \geq 0$ when $\Gamma_{\min} < I < \Gamma_{\max}$ and evidence > 0	☐
Deterministic WCET	Timing analysis: All branches execute in bounded cycles (no loops, no dynamic allocation)	☐
State serialization	Verify `AHP_State` can be atomically serialized for distributed consensus messages	☐

These specifications transform the conceptual framework into *falsifiable infrastructure*. Researchers in each domain may proceed independently, using the reference implementations above, with validation results feeding back into parameter refinement for future iterations of the AHP specification.

7 Connection to Protocol Layer and Governance

This section makes explicit how the executable logic presented here serves as a concrete instantiation of abstract protocol specifications.

The C implementation in Listing 1 serves as the runtime engine for PHA-Hysteresis HTTP headers and Byzantine consensus nodes. The mapping is direct:

- `state->D` maps to `state` parameter in HTTP headers (`accumulating` when $0 < I < \Gamma_{\max}$, `stabilized` when $D = 1$)
- `ahp_update()` computes `gamma` parameter ($I/\Gamma_{\max}$) exposed in headers
- `dt` (timestep) enforces `delay` parameter ($\Delta T_{\min}$)
- **Byzantine Consensus**: The function `ahp_update()` executes identically on all n nodes in the governance consortium, ensuring that the `Accumulating` phase provides the temporal window for PBFT `Prepare` and `Commit` phases

When embedded in network middleware, the function `ahp_update()` executes on each evidence packet arrival, translating physical sensor data or inter-node messages into protocol-compliant hesitation semantics that prevent pathological synchronization across jurisdictions.

8 Conclusion

This supplement has demonstrated that AHP is not merely a theoretical framework but an *implementable architecture* with:

- **Minimal footprint**: <50 CPU cycles per update ($O(1)$ complexity)

Table 3: Complete Technical Specifications for AHP Empirical Validation. This operationalization corresponds to the validation strategy outlined in the main article (Section 6.4). All implementations use reference code from Listings 1–2 with domain-specific parameter calibration.

Domain	**Source**	**Data Type**	**Temp. Res.**	**AHP Par.**	**Impl.**	**Metric**
Finances (HFT)	[3], [4]	Order book L2/L3	1–100 μs	$\Delta T_{\min}$ = 20–50 ms	Python (pandas)	Cross-corr. $\rho < 0.3$
Neurotech	[5]	ECoG imagery	<10 ms, 256ch	$\Delta T_{\min}$ = 100–150 ms	Python (MNE)	AUC > +15%
Smart Grid	[6]	Power flow	1–10 ms	$\Delta T_{\min}$ = 0.5–2 s	MATLAB/ PowerWorld	FPR <5%, TPR >95%
Autonomous Veh.	[7], [8]	Synth. + LiDAR	10–100 ms	$\Delta T_{\min}$ = 100–300 ms	Py (CARLA)	$\text{FPR}_{\text{brake}}$ ↓15%→3%

- **Provable guarantees**: Implements Non-Zeno Guarantee
- **Domain flexibility**: Parameters calibratable across 4 orders of magnitude (μs to seconds)
- **Protocol ready**: Runtime support for HTTP headers with deterministic execution
- **Governance enabled**: Deterministic temporal guarantees required for Byzantine consensus coordination across jurisdictions

Taken together, these implementations demonstrate that temporal governance can be specified, enforced, and audited at runtime, providing a concrete foundation for empirical evaluation and cross-jurisdictional deployment of AHP-based systems.

References

[1] Sah, A. (2026). *Algorithmic Hysteresis Primacy (AHP): Temporal Sovereignty in AI Governance*. Working paper available at `https://zmem.org` and SSRN.

[2] Sah, A. (2026). *Supplementary Materials: Protocol Specifications, Implementation Examples, Governance Frameworks, and Validation Protocols*. Working paper available at `https://zmem.org` and SSRN.

[3] LOBSTER Data. (2024). Limit Order Book System—The Efficient Reconstructor. `https://lobsterdata.com`. Free tier provides Level-2 order book data for academic research.

[4] QuantQuote. (2024). Historical Stock Data. `https://quantquote.com`. Historical tick data for equity markets.

[5] OpenNeuro. (2018). Dataset ds001787: Motor Imagery from ECoG recordings. `https://openneuro.org/datasets/ds001787`. Electrocorticographic (ECoG) motor imagery dataset; 256+ channels, <10ms temporal resolution.

[6] IEEE Power and Energy Society. (2019). IEEE 39-Bus Test System. `https://site.ieee.org/pes-testfeeders/`. Standard synthetic power system for transient and dynamic stability studies.

[7] Dosovitskiy, A., Ros, G., Codevilla, F., López, A., & Koltun, V. (2017). CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning* (pp. 1–16). `https://carla.org`

[8] Motional. (2020). nuScenes: A Multimodal Dataset for Autonomous Driving. `https://www.nuscenes.org`. Limited free tier for academic research; 1000 scenes with LiDAR, radar, camera.