The BX3 Framework: A Universal Architecture for Accountable Autonomous Systems

Jeremy Blaine Thompson Beebe
Bxthre3 Inc. — bxthre3inc@gmail.com — ORCID: 0009-0009-2394-9714

April 2026

Abstract

The current discourse around artificial intelligence presents a false binary: either AI replaces traditional software, or it is a limited novelty. Both fail engineers, architects, and decision-makers by offering no principled model for how these technologies relate. This paper proposes the **BX3**Framework — a universal architectural model organized around three immutable functional layers: the *Purpose Layer*, which provides intent, judgment, and accountability; the *Bounds Engine*, which provides bounded reasoning and constrained proposal; and the *Fact Layer*, which provides deterministic enforcement and forensic auditability. Critically, the framework is actor-agnostic: any entity — human, AI, mechanical, or institutional — may occupy any layer provided it satisfies the layer's functional requirements. The framework's core safety property is an upstream accountability guarantee: when any node encounters a condition it cannot resolve, accountability escalates recursively upward through the hierarchy, bypassing machine actors, until it reaches a human accountability anchor. The system fails upward into human consciousness. It never fails downward into algorithmic chaos. The **BX3**Framework is demonstrated as a universal structural pattern observable across law, medicine, autonomous vehicles, organizational design, and biological cognition.

Keywords: BX3 Framework, Purpose Layer, Bounds Engine, Fact Layer, autonomous systems, AI governance, human-in-the-loop, deterministic systems, sociotechnical systems, accountability, recursive orchestration

1 Introduction

The rapid deployment of large language models and AI-based systems has generated significant confusion about the appropriate role of AI within engineered systems. A prevalent narrative suggests AI will progressively replace traditional deterministic software. A counter-narrative dismisses AI as inadequate for production reliability demands. Both positions offer no principled model for how these technologies relate in practice.

This paper proposes the **BX3**Framework. The framework organizes any complex system into three immutable functional layers — *Purpose Layer*, *Bounds Engine*, and *Fact Layer*— each defined by the properties it must maintain rather than by the type of actor occupying it.

This actor-agnostic definition is the framework's most important architectural property. A human, an AI system, a mechanical process, an institution, or any combination thereof may legitimately occupy any layer, provided the functional requirements of that layer are satisfied. The framework does not prescribe *who* belongs in each layer but *what properties* each layer must maintain for the system as a whole to be reliable, governable, and certifiable.

The framework synthesizes well-established principles — separation of concerns [**?**], sociotechnical systems theory [**?**], control theory [**?**], and human-in-the-loop design [**?**] — applied to the specific challenge of AI integration in the current technological moment.

Note: The first-person plural "we" is used in the conventional academic sense, consistent with single-author scholarly writing.

2 Defining the Three Functional Layers

The **BX3**Framework defines three functional layers, each defined by the *properties it must maintain*, not by the type of actor occupying it.

2.1 Layer 1: Purpose Layer

The Purpose Layer is responsible for *intent*, *judgment*, and *accountability*. Whatever occupies this layer must be capable of defining the goals the system exists to achieve, making trade-off decisions under genuine uncertainty where no algorithmic optimum exists, holding accountability for outcomes, and updating goals when context changes in ways the system was not designed to anticipate.

In the current technological moment, the Purpose Layer must remain anchored to a *human accountability anchor* — an individual or institution capable of bearing legal and ethical responsibility. This is the **Human Root Mandate**: in the event of system failure, accountability does not dissipate into the algorithm. It remains fixed to the human at the root.

Key property: The Purpose Layer must be *accountable* — its decisions must be attributable to an actor who can be questioned, overridden, and held responsible.

2.2 Layer 2: Bounds Engine

The Bounds Engine is responsible for *interpretation*, *bounded reasoning*, and *constrained execution*. It performs cognitive work — analysis, pattern recognition, simulation, and optimal path proposal — but is architecturally *limbless*: it can propose but cannot execute. It lacks authority to commit actions to the physical world unilaterally.

This layer is most commonly occupied by an AI agent. It operates within a sandboxed cognitive environment, separated from physical execution authority, and escalates to the Purpose Layer when situations exceed its capability or authority.

Key property: The Bounds Engine must be *bounded* — it proposes but never executes; its authority is constrained by the Safety Envelope.

2.3 Layer 3: Fact Layer

The Fact Layer is responsible for *deterministic enforcement*, *hard physical constraint*, and *forensic auditability*. Whatever occupies this layer must produce consistent, reproducible outcomes given identical inputs, enforce physical constraints that cannot be overridden by the Bounds Engine, and maintain an immutable record of all physical events.

The Fact Layer is the only layer that can act on the physical world. It is structurally incapable of reasoning — it executes, it does not deliberate.

Key property: The Fact Layer must be *deterministic* — same inputs, same outputs, every time, without exception.

2.4 Layer Comparison

Property	**Purpose Layer**	**Bounds Engine**	**Fact Layer**
Function	Intent, judgment, accountability	Reasoning, proposal, simulation	Enforcement, constraint, audit
Default occupant	Human / institution	AI agent / heuristic	Software / mechanism
Key obligation	Accountability anchor	Boundedness (limbless)	Determinism (no drift)

3 Formal Model

We formalize the **BX3**Framework as a triple of layer constraints applied to a system Σ. The system Σ is a 5-tuple:

$$\Sigma = (P, B, F, E, H) \tag{1}$$

where:

- P is the Purpose Layer, satisfying **P1**–**P3**
- B is the Bounds Engine, satisfying **B1**–**B3**

- F is the Fact Layer, satisfying **F1–F3**
- E is the Execution environment
- H is the Human Accountability Anchor

3.1 Layer Constraints

3.1.1 Purpose Layer Constraints (P1–P3)

P1 — Intent Anchoring P defines a mission function $\mu : \mathcal{C} \to \mathcal{G}$ mapping system context $\mathcal{C}$ to goals $\mathcal{G}$. The Purpose Layer may not be empty.

P2 — Accountability Attribution For every action $a \in A$ executed by Σ, there exists a chain of attribution $\tau(a) = (p_1, p_2, \ldots, p_n)$ terminating at a human actor $p_n \in H$. No action is anonymously attributed.

P3 — Override Capability Any decision d produced by B may be overridden by P. Override is final and deterministic.

3.1.2 Bounds Engine Constraints (B1–B3)

B1 — Limblessness B produces proposals $\pi \in \Pi$ but never directly executes π on F. Execution authority is architecturally separate from proposal authority.

B2 — Safety Envelope Compliance All proposals π must satisfy $\pi \in \mathcal{S}$, where $\mathcal{S}$ is the Safety Envelope defined by P. Proposals outside $\mathcal{S}$ are rejected before evaluation.

B3 — Escalation on Unresolvable Bounds Upon encountering condition $c \in \mathcal{C}$ such that $\nexists \pi \in \Pi$ satisfying both $B2$ and the local optimization objective, B generates an exception e and propagates it upward to P. No autonomous continuation is permitted.

3.1.3 Fact Layer Constraints (F1–F3)

F1 — Determinism $\forall s \in \mathcal{S}_F, \forall i \in \mathcal{I} : \delta_F(s, i) = s'$ where δ_F is the deterministic transition function and s' is unique.

F2 — Hard Constraint Enforcement Physical constraints $\mathcal{H} \subset \mathcal{I}$ are enforced by F unconditionally. No agent, including P or B, may override $\mathcal{H}$.

F3 — Immutable Audit Logging Every action a executed by F is recorded in an append-only log $\mathcal{L}$. Entries in $\mathcal{L}$ are (timestamp, actor, action, result) tuples. $\mathcal{L}$ is cryptographically sealed.

3.2 System-Level Properties

Theorem 1 (Upstream Accountability Guarantee)**.** *When any node in Σ encounters an unresolvable condition c, accountability escalates recursively upward through the hierarchy, bypassing all machine actors, until it reaches H. Formally:*

$$\forall \sigma \in \Sigma, \forall c \in \mathcal{C} : \mathit{unresolvable}(c, \sigma) \implies \mathit{attr}(\sigma, c) \in H \tag{2}$$

where $\mathit{attr}(\sigma, c)$ is the accountable entity for condition c at node σ.

Theorem 2 (Loop Isolation Guarantee)**.** *The Bounds Engine and the Fact Layer are never co-located on the same functional plane. No proposal π generated by B is executed by B. Formally:*

$$\forall \pi \in \Pi : \mathit{executor}(\pi) \neq B \tag{3}$$

where $\mathit{executor}(\pi)$ denotes the entity that commits π to physical execution.

4 Grammar for BX3-Compliant Systems

To specify well-formed **BX3**systems, we define a BNF-style grammar. Any system expressed in this grammar is provably compliant with all layer constraints.

```
<bx3-system>       ::= <layer-assignment> <pillar-instantiation>
<layer-assignment> ::= <purpose-layer> <bounds-engine> <fact-layer>
<purpose-layer>    ::= PURPOSE ( human | institution | hybrid )
                       WHERE accountability-chain = <chain>
<bounds-engine>    ::= BOUNDS ( ai-agent | heuristic | hybrid )
                       WHERE safety-envelope = <envelope>
                       CONSTRAINED BY limblessness
<fact-layer>       ::= FACT ( software | mechanism | hybrid )
                       WHERE determinism = ENFORCED
                       AND audit-log = SEALED
<chain>            ::= <actor> -> <actor> -> ... -> HUMAN
<envelope>         ::= { <parameter> : <range> , ... }
<pillar-instantiation>
                    ::= LOOP-ISOLATION = ENFORCED
                       RECURSIVE-SPAWNING = <spawn-mode>
                       SPATIAL-FIREWALL = <tier>
                       SANDBOX-GATE = <gate-mode>
                       BAILOUT-PROTOCOL = ENFORCED
<spawn-mode>       ::= WORKSHEET | BRANCH | BOTH
<tier>             ::= <resolution-tier> { , <resolution-tier> }
<gate-mode>        ::= DIGITAL-TWIN | SIMULATION | BOTH
```

A system that cannot be expressed in this grammar is not **BX3**-compliant. Compliance is a syntactic property verifiable at design time.

5 The Five Pillars

The three functional layers define *what* a **BX3**-compliant system must contain. The Five Pillars define *how* those layers must behave in practice to maintain their properties under real-world conditions including network failures, scale, security threats, and exception states.

5.1 Pillar 1: Loop Isolation

Problem solved: Logic Collision — when the Bounds Engine and Fact Layer occupy the same functional plane, enabling un-vetted autonomous actions that bypass physical constraint.

Solution: Strict isolation of the three functional layers into discrete planes. Each **BX3**loop is self-contained. A Logic Collision is architecturally impossible because the Bounds Engine never shares a functional plane with physical execution. The Bounds Engine proposes; the Fact Layer decides. These are never the same operation.

Implementation: Loop Isolation is enforced at the process boundary level. The B process runs in a sandboxed memory environment with no direct handles to actuators, database writes, or network sockets. Communication between B and F occurs exclusively through a typed message queue M_{BF} with the following properties:

- M_{BF} is unidirectional: $B \to F$ only
- Messages in M_{BF} are immutable once sent
- F validates every message against $\mathcal{H}$ before execution
- F returns execution receipts to B via a separate channel M_{FB}

Formal guarantee: $\forall \pi \in \Pi : \text{execute}(\pi) \implies \text{execute}(\pi) \in F$. The Bounds Engine cannot be the executor of its own proposals.

5.2 Pillar 2: Recursive Spawning

Problem solved: Logic Rigidity — static edge devices that cannot adapt to local conditions without constant cloud connectivity.

Solution: A parent Bounds Engine births a child loop by generating a *Worksheet* — a containerized, self-contained logic set encapsulating the parent's Purpose for a specific local context — deployed over-the-air to the child node. Each Worksheet carries a hard-coded pointer to the parent's Purpose, preventing autonomous drift. The child loop applies the parent's intent to local sensor data independently without requiring a constant cloud heartbeat.

Worksheet Specification: A Worksheet W is a 4-tuple:

$$W = (\mu_P, \mathcal{S}_P, \mathcal{H}_P, \text{parent\_ptr}) \tag{4}$$

where μ_P is the parent mission function, $\mathcal{S}_P$ is the parent Safety Envelope, $\mathcal{H}_P$ is the parent hard constraint set, and parent_ptr is a cryptographically verifiable pointer to the parent's Purpose Layer.

Deployment Protocol:

1. Parent B evaluates local context $\mathcal{C}_{\text{local}}$
2. Parent generates W with $\mu_P[\mathcal{C}_{\text{local}}]$, inheriting all $\mathcal{S}_P$ constraints
3. W is transmitted over-the-air to child node
4. Child node verifies parent_ptr against the canonical chain-of-custody ledger
5. Child loop instantiates B_{child} with W and begins independent operation
6. Child periodically re-verifies parent_ptr; stale verification triggers Bailout

This architecture allows edge nodes to maintain local autonomy while remaining tethered to the parent Purpose, preventing drift without requiring persistent connectivity.

5.3 Pillar 3: Spatial Firewall

Problem solved: Cross-tier data leakage — a compromised or unauthorized node accessing data at a resolution or scope beyond its provisioned tier.

Solution: Physical isolation of data planes by resolution tier. A node provisioned at 50-meter resolution cannot access 1-meter data because the 1-meter data plane does not exist in the node's provisioned environment. Isolation is enforced by architecture, not by permissions.

Resolution Tier Model: Data planes are partitioned into discrete tiers $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ where each tier t_i corresponds to a spatial resolution r_i. Nodes are provisioned with a tier assignment $\tau(n) \in \mathcal{T}$. A node n with $\tau(n) = t_i$ has access to all tiers t_j where $r_j \geq r_i$ (coarser or equal resolution). Access to finer-resolution tiers ($r_j < r_i$) is architecturally impossible: those data planes are not present in the node's instantiated environment.

Formal guarantee: $\forall n \in \mathcal{N}, \forall t \in \mathcal{T} : r_t < r_{\tau(n)} \implies t \notin \text{env}(n)$. A node cannot access data it was not architecturally provisioned to see.

This is meaningfully different from permission-based access control. Permissions can be exploited by privilege escalation attacks. Spatial Firewalls make the data simply not exist in the attacker's environment.

5.4 Pillar 4: Sandbox Gate

Problem solved: Premature execution — the Bounds Engine proposing actions that, if executed, would violate Safety Envelope parameters before the system has an opportunity to evaluate them.

Solution: All proposed actions are evaluated in a digital twin before physical execution. The Sandbox Gate runs the proposed action against a simulation of the Fact Layer's current state and confirms it falls within all Safety Envelope parameters before unlocking physical execution.

Sandbox Gate Protocol:

1. B generates proposal π
2. π is sent to Sandbox Gate alongside current state snapshot $s \in \mathcal{S}_F$

3. Sandbox Gate instantiates digital twin F' with state s
4. F' executes π and returns result state s'
5. Sandbox Gate evaluates whether s' satisfies all $\mathcal{S}$ constraints
6. If $\mathcal{S}$-satisfying: π is forwarded to F with execution token
7. If $\mathcal{S}$-violating: π is rejected, exception raised to B

The digital twin F' is a high-fidelity simulation of the Fact Layer. It is computationally cheaper than physical execution (which may involve irreversible physical effects) and enables complete pre-execution validation.

Latency bound: The Sandbox Gate adds $O(\tau_{\text{sim}})$ latency where τ_{sim} is the simulation time for π against F'. For deterministic F, $\tau_{\text{sim}} \ll \tau_{\text{real}}$ typically holds.

5.5 Pillar 5: Bailout Protocol

Problem solved: Accountability orphaning — an autonomous system encountering a condition it cannot handle and either halting without notification or continuing without authorization.

Solution: Every node carries the complete Bailout Protocol. When a node encounters a condition it cannot resolve, it propagates an exception asynchronously up the recursive tree, bypassing all intermediate machine actors, until it reaches a Human Accountability Anchor. The system fails upward into human consciousness.

Bailout Exception Propagation: The exception e raised by B or F carries:

- node_id: originating node
- timestamp: when condition was encountered
- condition_class: categorized failure mode
- context_snapshot: relevant system state at exception time
- propagation_path: list of nodes that handled (and rejected) e en route to H

Critically, the propagation path is append-only and includes all intermediate actors that could not resolve e. This means H receives not just the exception, but the full traversal history: exactly which machine actors encountered and could not resolve the condition.

Formal guarantee: $\forall e \in \mathcal{E}, \text{final\_recipient}(e) \in H$. No exception terminates at a machine actor. Bailout is the only protocol that bypasses intermediate machine actors, ensuring accountability cannot be absorbed by the hierarchy.

6 Implementation: Pseudo-Code for Each Pillar

6.1 Loop Isolation — Typed Message Queue

```
// Loop Isolation: Message Queue Enforcement
```

```
procedure SendToFactLayer(msg, M_BF):
    assert msg.sender = B          // Reject non-Bounds Engine messages
    assert msg.type in {PROPOSAL, QUERY}
    M_BF.append(msg)
    return msg.id

procedure ProcessQueueItem(item, F):
    validated_msg = F.validate(item)
    if validated_msg.type = PROPOSAL:
        result = F.execute_checked(validated_msg.payload)
        M_FB.append(EXECUTION_RECEIPT, result)
    else if validated_msg.type = QUERY:
        result = F.state_query(validated_msg.payload)
        M_FB.append(STATE_RESPONSE, result)
```

6.2 Recursive Spawning — Worksheet Generation

```
// Recursive Spawning: Worksheet Generation
procedure GenerateWorksheet(B_parent, C_local):
    mu_sub = B_parent.mission.restrict(C_local)
    S_sub = B_parent.safety_envelope
    H_sub = B_parent.hard_constraints
    parent_ptr = CryptographicChainOfCustody(B_parent.purpose)
    W = (mu_sub, S_sub, H_sub, parent_ptr)
    return W

procedure DeployToNode(W, node_id):
    payload = serialize(W)
    transmit(payload, node_id)
    node_id.on_receive(W)

procedure OnNodeReceive(W, node_id):
    verified = verify_parent_ptr(W.parent_ptr)
    if not verified:
        raise_bailout(PARENT_PTR_INVALID)
    instantiate_bounds_engine(W, node_id)
    begin_operation()
```

6.3 Sandbox Gate — Digital Twin Pre-Execution Validation

```
// Sandbox Gate: Digital Twin Validation
procedure SandboxEvaluate(B, F, pi):
    s = F.snapshot()
    F_prime = instantiate_digital_twin(F, s)
    s_prime = F_prime.execute_simulated(pi)
```

```
    pass = evaluate_safety_envelope(s_prime, B.S)
    if pass:
        return {APPROVED, pi}
    else:
        return {REJECTED, pi, violated_constraints(s_prime, B.S)}

procedure ExecuteIfApproved(B, F, pi):
    {status, ...} = SandboxEvaluate(B, F, pi)
    if status = APPROVED:
        token = generate_execution_token(pi)
        F.execute(pi, token)
        F.log(EXECUTION, pi, token)
    else:
        B.log_rejection(pi)
        B.raise_escalation(pi)
```

6.4 Bailout Protocol — Exception Propagation

```
// Bailout Protocol: Exception Propagation
procedure RaiseBailout(node, e):
    e.propagation_path = []
    e.node_id = node.id
    e.timestamp = now()
    Propagate(e, node.parent)

procedure Propagate(e, target):
    if target.type = HUMAN:
        deliver_to_human(e)
        return
    handled = target.handle(e)
    if not handled:
        e.propagation_path.append(target.id)
        Propagate(e, target.parent)

procedure HandleException(e):
    // Machine actors attempt handling
    local_resolution = attempt_local_resolution(e)
    if local_resolution != RESOLVED:
        return FALSE  // Not handled; propagate upward
    log_resolution(e)
    return TRUE
```

7 Complexity and Performance Analysis

Each pillar introduces computational overhead. We analyze the complexity of each and identify mitigation strategies.

7.1 Loop Isolation

The message queue M_{BF} introduces queue latency τ_q per message. For a system with proposal rate λ:

$$\text{Throughput}_{M_{BF}} = \frac{\lambda}{1 + \lambda \cdot \tau_q} \tag{5}$$

Mitigation: Batching. Multiple proposals can be batched into a single queue item for F to process atomically, reducing τ_q amortized cost per proposal.

7.2 Recursive Spawning

Worksheet generation has complexity $O(|\mathcal{G}|)$ where $\mathcal{G}$ is the goal space of the parent mission. Deployment has latency τ_{deploy} dominated by network bandwidth.

Mitigation: Worksheet templates. Pre-defined worksheet templates for common contexts reduce generation time to $O(1)$. Deployment is incremental (only delta from parent state is transmitted).

7.3 Sandbox Gate

Digital twin simulation adds τ_{sim} per proposal. For a Fact Layer with state space $\mathcal{S}_F$ and proposal complexity $O(k)$:

$$\tau_{\text{sim}} = O(k \cdot |\mathcal{S}_F|) \tag{6}$$

Mitigation: Progressive fidelity. The digital twin starts at low fidelity (simplified physics) and increases fidelity only if low-fidelity evaluation is inconclusive. Caching of simulation results for similar states reduces repeated computation.

7.4 Spatial Firewall

Isolation enforcement is $O(1)$ at instantiation time. No per-operation overhead beyond tier assignment verification.

7.5 Bailout Protocol

Exception propagation traverses d levels of the recursive hierarchy. Total latency:

$$\tau_{\text{bailout}} = O(d \cdot \tau_{\text{hop}}) + \tau_{\text{notify}} \tag{7}$$

where τ_{hop} is inter-node latency and τ_{notify} is human notification time. Critically, τ_{bailout} is bounded by $d_{\max}$, the maximum depth of the hierarchy.

Human notification: Modern alerting systems (SMS, push notifications) provide $\tau_{\text{notify}} < 5$ seconds typically.

8 Extended Case Studies

We present extended case studies demonstrating the framework's application across five domains.

8.1 Case Study 1: Autonomous Agricultural Irrigation

Context: Center-pivot irrigation in Colorado's San Luis Valley. A 500-acre circular field managed by a single farming operation.

Layer Mapping:

- **Purpose Layer:** Farmer (human accountability anchor) defines irrigation goals: maximize yield, minimize water cost, comply with water rights restrictions. The farmer bears legal responsibility for water use.
- **Bounds Engine:** AI system analyzes satellite imagery, soil sensor data, and weather forecasts. Proposes watering schedules and volumes within the Safety Envelope defined by the farmer.
- **Fact Layer:** Variable frequency drive (VFD) pump controllers, solenoid valves, and flow meters. Deterministic execution of approved irrigation commands. Immutable water use log maintained per state agricultural regulations.

Pillar Application:

- **Loop Isolation:** Code generation agents output diffs; CI/CD executes builds and deployments. Agents cannot bypass CI/CD.
- **Recursive Spawning:** Sub-tasks spawn child agents with Worksheet containing scope, constraints, and acceptance criteria.
- **Spatial Firewall:** Security scanning agent cannot access production database credentials, even if compromised. Credential plane not in its environment.
- **Sandbox Gate:** Generated code changes are evaluated in ephemeral build environment before merge.
- **Bailout Protocol:** Security finding above critical severity $\rightarrow$ immediate escalation to engineering lead, pipeline halted.

Legal compliance: The radiologist's Purpose Layer signature on every final report satisfies medical-legal requirements. The AI's bounding is documented by the loop isolation architecture, demonstrating that AI was a proposal engine only.

8.2 Case Study 3: Multi-Agent Software Engineering Pipeline

Context: Autonomous code generation and deployment pipeline using multiple AI agents.

Layer Mapping:

- **Purpose Layer:** Engineering lead. Defines architectural requirements, acceptance criteria, security policies, and code quality standards. Accountable for all deployed code.
- **Bounds Engine:** Specialist agents (code generation, testing, security scanning, documentation) each occupy the Bounds Engine role for their specialty. Propose changes but cannot deploy.
- **Fact Layer:** CI/CD pipeline (deterministic build, test, and deployment system). Enforces that only CI/CD can write to production. Maintains immutable deployment log.

Pillar Application:

- **Loop Isolation:** Code generation agents output diffs; execution system confirms compliance with risk parameters before submission.
- **Sandbox Gate:** Proposed code changes are simulated against current portfolio state to verify loss threshold compliance before execution.
- **Bailout Protocol:** Market data anomaly or threshold breach $\rightarrow$ halt $\rightarrow$ human operator notification with full context.

Certification note: The deterministic Fact Layer (order execution + audit log) is what satisfies regulatory requirements for auditability in financial systems.

8.3 Case Study 5: National Defense Multi-Agent Command System

Context: Federated command system with multiple autonomous assets under human operator oversight.

Layer Mapping:

- **Purpose Layer:** Command officer. Defines mission objectives, rules of engagement, and tactical constraints.
- **Bounds Engine:** Autonomous asset controllers analyze sensor data, coordinate maneuvers, propose tactical actions.
- **Fact Layer:** Platform control systems execute approved actions. Hard physical constraints (weapons systems interlocks, collision avoidance) cannot be overridden.

Pillar Application:

- **Loop Isolation:** Tactical proposals from autonomous controllers are reviewed by command before execution authorization. Weapons only fire when command authorizes.

- **Recursive Spawning:** Subordinate units receive mission Worksheets with commander's intent. Units execute locally when comms degraded.
- **Spatial Firewall:** Intelligence data is tiered. Strategic assets cannot access tactical sensor streams beyond their provisioned tier.
- **Bailout Protocol:** Rules of engagement violation or unknown contact classification $\rightarrow$ immediate halt $\rightarrow$ human operator notification.

The framework's actor-agnostic property is most critical here: the same architectural rules apply whether the Purpose Layer occupant is a human commander, a multinational coalition headquarters, or an allied nation command structure. The layer constraints remain invariant.

9 Edge Cases and Limitations

9.1 Human Accountability Anchor Unavailability

If H is unreachable (medical emergency, travel, etc.), the system does not enter autonomous mode. Pillar 5 (Bailout Protocol) has no resolution path to an absent H. In this case, the system transitions to a **Halted State**: all Fact Layer execution stops, pending actions are queued, and H is notified by all available channels upon re-availability. This is a deliberate design choice: the framework does not permit autonomous continuation in the absence of human accountability.

Mitigation: Redundant H provisioning. Multiple named human accountability anchors with failover priority. The second H is notified if the primary is unreachable for more than T_{grace} minutes.

9.2 Purpose Layer Actor Conflict

When multiple legitimate Purpose Layer occupants disagree (e.g., two authorized managers with conflicting directives), the system enters a **Conflict State**. No proposal may proceed until the conflict is resolved at the Purpose Layer level. The Bounds Engine does not arbitrate between competing human directives.

Mitigation: Priority resolution rules defined at system instantiation. A partial order over Purpose Layer occupants (ranked by role, not by seniority) resolves conflicts deterministically.

9.3 Determinism vs. Probabilistic Fact Layer

Some real-world systems contain components that are inherently probabilistic (thermal noise in sensors, quantum effects in hardware). A fully deterministic Fact Layer may be physically impossible for certain subsystems.

Mitigation: The framework distinguishes between **strong determinism** (identical inputs $\rightarrow$ identical outputs) and **bounded probabilistic** (outputs within formally verified

confidence intervals). Bounded probabilistic components satisfy F1 under the confidence interval definition. Where probabilistic bounds are unacceptable (e.g., financial calculations), deterministic equivalents must be used.

9.4 Recursive Spawning Depth

Deep recursive spawning creates a hierarchy where Bailout propagation latency $d \cdot \tau_{\text{hop}}$ may be unacceptable for time-critical applications.

Mitigation: Depth-bounded spawning. The Worksheet schema includes a max_spawn_depth field. Spawning beyond this depth is prohibited. Shallow hierarchies reduce propagation latency at the cost of reduced local autonomy.

10 Extended Related Work

The **BX3**Framework synthesizes established ideas from separation of concerns, sociotechnical systems theory, cybernetics, systems safety, and human-in-the-loop design, but departs from prior work by defining immutable functional layers according to required properties rather than actor type.

10.1 Comparison with Prior Frameworks

Criterion	Sociotechnical Systems (iSTS)	Tiered Agentic Oversight (TAO)	BX3 Framework
Layer immutability	Design-time only	Runtime hierarchical	Immutable by architecture
Deterministic Fact Layer	Not isolated	Not isolated	Required, formal definition
Actor-agnostic	Limited	No	Yes, core property
Bailout protocol	None	High-tier optional	Required, all nodes
Human as terminal endpoint	Implicit	Optional	Required, enforced
Formal grammar	No	No	Yes, syntactic compliance

10.2 ISO/IEC 42001 and NIST AI RMF Alignment

ISO/IEC 42001 [**?**] defines an AI management system with governance, design, runtime, and assurance layers. The **BX3**Framework provides an operational architecture for the *runtime* and *assurance* layers specified by ISO 42001:

- ISO 42001 Governance $\supset$ **BX3**Purpose Layer (accountability, policy)

- ISO 42001 Design-time $\supset$ **BX3**Safety Envelope definition
- ISO 42001 Runtime $\equiv$ **BX3**Bounds Engine + Fact Layer operation
- ISO 42001 Assurance $\equiv$ **BX3**Audit Log + Determinism guarantee

NIST AI RMF [?] similarly specifies Govern, Map, Measure, and Manage functions. **BX3**maps to the Measure function (deterministic measurement via Fact Layer) and Manage function (Safety Envelope compliance via Bounds Engine). The Govern function is satisfied by the Purpose Layer's Human Root Mandate.

10.3 Production Agent Architecture Recommendations

Recent work by Alenezi et al. [?] provides implementation heuristics for production-grade agent systems: fail-safe behavior, sandbox-first execution, and human approval gates. **BX3**elevates these from implementation heuristics to architectural pillars with formal proofs and a BNF grammar specification.

The key distinction: heuristics are recommendations that engineers may or may not follow. **BX3**'s pillars are architectural constraints that the system *cannot violate* without ceasing to be **BX3**-compliant.

11 Why Determinism Is Not Compensable

Determinism provides properties that probabilistic systems cannot replicate:

- **Reproducibility:** Same inputs produce same outputs, enabling debugging, auditing, and legal accountability.
- **Formal verification:** Mathematical proof that a system satisfies properties under all possible inputs.
- **Certification:** Regulatory frameworks in aviation, medical devices, and safety-critical infrastructure require deterministic behavior as a precondition for approval.
- **Latency:** Hard real-time requirements are achievable with deterministic systems and not with current AI inference pipelines.

Every AI system in production today runs on top of vast deterministic infrastructure: operating systems, databases, networking stacks, authentication systems. The claim that AI will replace deterministic software is structurally self-refuting — AI systems are themselves dependent on deterministic software.

This creates an asymmetry: a probabilistic system can be made to behave more deterministically by constraining it within a deterministic Fact Layer. But a deterministic system cannot be made meaningfully probabilistic where it matters. The Fact Layer's deterministic constraints are a fixed point that bounds all uncertain reasoning above it.

12 Runtime Monitoring and Compliance Verification

A key practical requirement for **BX3**adoption is verifiable compliance: auditors must be able to confirm that a deployed system is actually maintaining its layer constraints at runtime. We address this with a **Compliance Monitor**, a passive auditing process that observes the running system and emits compliance attestations.

12.1 Compliance Monitor Architecture

The Compliance Monitor M_C is a separate process that observes all inter-layer communication without interfering with system operation:

- M_C observes M_{BF} (Bounds Engine $\rightarrow$ Fact Layer messages) in real time
- M_C observes the Bailout propagation paths
- M_C observes Purpose Layer override events
- M_C maintains an append-only Compliance Log L_C separate from the Fact Layer audit log

At configurable intervals (e.g., every 24 hours), M_C generates a **Compliance Attestation**:

```
Compliance Attestation - System: <id>
Generated: <timestamp>
Observations:
  - Total messages observed: <N>
  - Messages in Safety Envelope: <M> (<percentage>%)
  - Bailout events: <B>
  - Bailout events reaching H: <B> (100.0%)
  - Purpose Layer overrides: <O>
  - Loop Isolation violations: <V>
Conclusion: <COMPLIANT | NON-COMPLIANT>
Evidence hash: <sha256>
Signed by: <M_C identity>
```

The compliance attestation is signed by M_C and stored in an immutable log. This provides auditors with cryptographic evidence that the system was or was not in compliance during the attestation period.

12.2 Loop Isolation Violation Detection

Loop Isolation violations are the most critical failure mode to detect. M_C detects them through the following invariant checks:

1. Every message in M_{BF} must have msg.sender $= B$. If any message originates from F, a violation is recorded.

2. Every execution receipt in M_{FB} must correspond to a previously observed proposal in M_{BF}. Orphaned receipts (not matching a prior proposal) indicate a violation.

3. M_C maintains a state machine tracking the expected sequence: PROPOSAL $\rightarrow$ VALIDATED $\rightarrow$ EXECUTION_RECEIPT. Out-of-sequence events are flagged.

12.3 Safety Envelope Drift Detection

The Safety Envelope $\mathcal{S}$ is defined by the Purpose Layer at system instantiation. Over time, operational changes may cause $\mathcal{S}$ to drift from its original specification. M_C detects drift by:

1. Maintaining a canonical copy of $\mathcal{S}$ at instantiation

2. Periodically comparing current Safety Envelope parameters against the canonical copy

3. If drift is detected: recording the drift event and alerting H

Drift detection is important because gradual Safety Envelope relaxation can accumulate into a state where the Bounds Engine has effectively unbounded authority — precisely the condition **BX3**is designed to prevent.

13 Formal Verification of BX3 Compliance

Rigorous compliance verification requires the ability to prove, not merely test, that a **BX3**system maintains its constraints under all possible inputs. We describe here the formal verification methodology for the framework.

13.1 Invariant Specifications

Each layer constraint and pillar property is expressed as a formal invariant that must hold throughout system operation. Invariants are specified in first-order logic over the system state Σ:

Loop Isolation Invariant: $\forall \pi \in \Pi : \text{executor}(\pi) \in F \implies \text{executor}(\pi) \neq B$

Determinism Invariant: $\forall s \in \mathcal{S}_F, \forall i \in \mathcal{I} : \delta_F(s, i) = \delta_F(s, i)$

Bailout Termination Invariant: $\forall e \in \mathcal{E} : \text{type}(\text{final\_recipient}(e)) = \text{HUMAN}$

Safety Envelope Invariant: $\forall \pi \in \Pi : \pi \in \mathcal{S} \cup \{\text{REJECTED}\}$

Hard Constraint Invariant: $\forall a \in \mathcal{H} : \delta_F(a) \neq \perp$ (hard constraints are always enforced)

Human Accountability Invariant: $\forall a \in A : \text{last\_element}(\tau(a)) \in H$ (every action traces to a human)

13.2 Model Checking

For systems with finite state spaces, model checking can exhaustively verify all invariants. The state space is constructed from:

- The finite set of possible Purpose Layer configurations
- The finite set of Bounds Engine proposal states
- The finite Fact Layer state machine
- The finite set of inter-layer message types

Model checking verifies that for every reachable state in the state space, all invariants hold. If a state is found where an invariant is violated, that state represents a counterexample — a real-world condition that would cause the system to violate **BX3**constraints.

Tool recommendation: TLA+ is well-suited for model checking **BX3**systems. The specification encodes each layer as a separate module, and the invariants above are checked across module boundaries.

13.3 Deadlock and Liveness Analysis

Beyond invariant checking, formal verification must also confirm that the system does not enter states where required actions cannot complete. Two key properties:

Deadlock freedom: The system never enters a state where no further progress is possible except through human intervention for conditions that should be autonomously resolvable. Note that Bailout-induced halts are not deadlocks — they are the defined behavior when autonomous resolution is impossible.

Liveness: For every legitimate request r, there exists a finite execution path leading to a response to r. Requests are not indefinitely deferred.

These properties are verified using temporal logic specifications (e.g., $\Box\neg$deadlock in TLA+).

13.4 Probabilistic Bounding

For systems containing bounded probabilistic components (see Section 9), formal verification includes probabilistic model checking:

- Upper bounds on probability of hard constraint violation are established using probabilistic model checking (PRISM or Storm)
- Bounds are verified against acceptance thresholds specified by the Purpose Layer
- If any probability bound exceeds its threshold, the system fails the verification and must return to design

For example: a sensor with bounded probabilistic output must have a formally verified upper bound on the probability of generating an incorrect measurement that would cause the Fact Layer to execute an incorrect action. If this bound exceeds 10^{-6} per operation (for example), and the Purpose Layer has specified a threshold of 10^{-9}, the system fails verification.

13.5 Counterexample-Guided Redesign

When model checking reveals a counterexample (a state where an invariant is violated), the engineer must redesign to eliminate the counterexample. This process is iterative:

1. Model check reveals state s^* where LoopIsolationInvariant is violated
2. Engineer analyzes s^* to determine which code path leads to it
3. Engineer modifies the Fact Layer implementation to block that code path
4. Model is re-run; if s^* is unreachable, the invariant is strengthened
5. Process repeats until all invariants hold for all reachable states

This methodology ensures that **BX3**compliance is not a matter of testing but of formal proof — the system is compliant because it provably cannot violate its constraints.

14 Designing a BX3-Compliant System: A Step-by-Step Guide

This section provides actionable guidance for engineers designing a **BX3**-compliant system. We walk through each design decision point and the constraints that apply.

14.1 Step 1: Identify the Purpose Layer

The first decision is identifying or designating the Purpose Layer actor. This must be an entity capable of:

- Defining and updating system goals
- Bearing accountability for system outcomes (legal, ethical, or organizational)
- Overriding any Bounds Engine decision

In practice, this is typically a human individual (for personal systems) or a designated institutional role (for organizational systems). The Purpose Layer cannot be an AI system under current technological conditions.

Question to answer: Who is the human accountability anchor? Name them explicitly.

14.2 Step 2: Define the Safety Envelope

The Purpose Layer defines the Safety Envelope $\mathcal{S}$, which constrains all Bounds Engine proposals. The Safety Envelope should include:

- **Prohibited actions:** Things the system must never do, regardless of optimization incentive
- **Required actions:** Things the system must always do when conditions are met (e.g., emergency stop)
- **Constraint parameters:** Quantitative bounds (max velocity, max budget, min safety margin)
- **Escalation thresholds:** Conditions that trigger Bailout even if a proposal satisfies the above

The Safety Envelope should be specified in a machine-readable format so that M_C can verify compliance automatically.

14.3 Step 3: Select the Bounds Engine

The Bounds Engine is the cognitive layer — the component that reasons, proposes, and simulates. Selection criteria:

- Can it be configured to *never* directly execute actions on the physical world? (Architectural limblessness)
- Does it respect the Safety Envelope as a hard constraint, or must it be trained/guided to do so?
- Does it produce explainable proposals, or only opaque actions?
- Does it support exception raising and propagation?

AI agents are the natural choice, but rule-based systems and hybrid human-AI Bounds Engines are also valid.

14.4 Step 4: Implement the Fact Layer

The Fact Layer is the deterministic execution and audit layer. Implementation requirements:

- **F1 (Determinism):** Same inputs must produce same outputs. Avoid non-deterministic operations (random number generators without seeds, race conditions, external API calls without idempotency keys).
- **F2 (Hard Constraints):** Physical constraints must be enforced at the Fact Layer, not the Bounds Engine. The Fact Layer must reject commands that violate $\mathcal{H}$ even if those commands came from the Purpose Layer.

- **F3 (Audit Log):** Every execution must be recorded with timestamp, actor, action, and result. The log must be append-only and cryptographically sealed.

Common mistake: Implementing F2 (Hard Constraints) in the Bounds Engine rather than the Fact Layer. This is a violation of the **BX3**architecture, because the Bounds Engine becomes capable of overriding hard constraints.

14.5 Step 5: Instantiate the Five Pillars

Each pillar must be implemented as specified in Section 5:

1. **Loop Isolation:** Implement M_{BF} typed message queue. Verify that only proposals can flow from B to F.
2. **Recursive Spawning:** Implement Worksheet schema. Verify that every child node has a verifiable parent pointer.
3. **Spatial Firewall:** Partition data planes physically. Verify that tier-isolated nodes cannot access out-of-tier data.
4. **Sandbox Gate:** Implement digital twin. Verify that Sandbox-rejected proposals are never forwarded to F.
5. **Bailout Protocol:** Implement exception propagation. Verify that exceptions always terminate at H, never at a machine actor.

14.6 Step 6: Deploy the Compliance Monitor

Before going live, deploy M_C and verify:

- M_C correctly observes all inter-layer communication
- Compliance attestations are generated and signed correctly
- Violations (if simulated) are detected and recorded
- The compliance log is append-only and sealed

15 Extending the Framework

The **BX3**Framework is designed to be extensible. This section discusses planned extensions and the principles governing them.

15.1 Extensions Under Development

6-Layer Variant (Z-Axis Extension): A planned extension adds two additional layers above and below the existing three-layer structure. The upper extension (**Meta-Purpose Layer**) addresses meta-systemic concerns: when the Purpose Layer itself requires governance. The lower extension (**Sub-Fact Layer**) addresses physical substrate constraints at the hardware level. Both are under active development and will be published separately.

Formal Verification Bindings: Bindings to formal verification tools (Coq, Isabelle, TLA+) are being developed to enable machine-checked proofs of **BX3**compliance at design time.

Distributed Fact Layer: An extension addressing geo-distributed Fact Layer deployments, where deterministic enforcement must be maintained across asynchronous network conditions.

15.2 Extension Principles

Any extension to **BX3**must satisfy three criteria:

1. **Non-regression:** Existing deployments must remain compliant with the extended framework. Existing layer constraints (P1–P3, B1–B3, F1–F3) are immutable.

2. **Backward compatibility:** Existing Safety Envelopes, Worksheets, and Bailout Protocols must continue to function under the extended framework without modification.

3. **Formal specification:** Extensions must include formal definitions, not just prose descriptions. The extension must be expressible in the BNF grammar or an extension thereof.

These criteria ensure that the framework evolves without breaking existing deployments — a critical requirement for any framework used in safety-critical or regulated environments.

16 Emerging Application Domains

Beyond the domains addressed in Section 8, the **BX3**Framework is applicable to several emerging application areas.

16.1 Robotic Surgery

Robotic surgical systems represent a domain where the stakes of accountability are extreme. The da Vinci Surgical System and similar platforms currently operate with human surgeons in direct control. As autonomous surgical capabilities emerge, the need for **BX3**architecture becomes critical:

- **Purpose Layer:** Surgical attending physician. Defines operative goals, acceptable risk parameters, and abort conditions.

- **Bounds Engine:** AI planning system. Analyzes pre-operative imaging, proposes resection paths, simulates tissue response.
- **Fact Layer:** Robotic actuator controllers. Execute approved surgical motions deterministically. Maintain ELD (Electronic Logging Device) records for regulatory compliance.

The framework's determinism guarantee is particularly relevant here: surgical outcomes must be reproducible and auditable for medical-legal reasons.

16.2 Smart Grid and Energy Management

Modern electrical grids with distributed renewable generation require coordinated control across thousands of nodes. The **BX3**Framework provides an architecture for this coordination:

- **Purpose Layer:** Grid operator. Defines load balancing objectives, stability constraints, and economic dispatch goals.
- **Bounds Engine:** AI optimization system. Analyzes load patterns, forecasts renewable generation, proposes dispatch instructions.
- **Fact Layer:** SCADA (Supervisory Control and Data Acquisition) systems. Execute dispatch instructions, enforce physical grid constraints (frequency, voltage, current limits), maintain grid state log.

The framework also supports FDA's Total Product Life Cycle (TPLC) approach: the same architectural principles apply from design-time verification through post-market monitoring.

16.3 Autonomous Freight

Self-driving trucks and freight drones represent a domain where regulatory frameworks are actively evolving. **BX3**provides a principled architecture for navigating this regulatory uncertainty:

- **Purpose Layer:** Fleet operator. Defines routing objectives, delivery windows, and safety constraints.
- **Bounds Engine:** Path planning and obstacle avoidance AI. Proposes routes and maneuver sequences.
- **Fact Layer:** Vehicle control systems. Execute steering, throttle, and braking commands deterministically. Maintain ELD (Electronic Logging Device) records for regulatory compliance.

The key insight: certifying a **BX3**system is structurally simpler than certifying a monolithic autonomous system because each layer can be certified independently against its specific obligations.

17 Regulatory Compliance and Certification

As autonomous systems continue to proliferate across safety-critical domains, the frameworks used to govern them will shape the balance of power between human accountability and machine autonomy. **BX3**provides a principled foundation for that governance: one that treats accountability as non-negotiable, reasoning as bounded, and enforcement as deterministic. The alternative — unconstrained autonomous systems with no guaranteed path to human accountability — is a risk the framework explicitly refuses to accept.

18 Conclusion

The question of how humans, AI, and traditional software should relate is an architectural, organizational, ethical, and regulatory question with significant practical consequences.

The **BX3**Framework proposes a principled, universal answer: organize any complex system into three functional layers — Purpose, Bounds Engine, and Fact — each defined by the properties it must maintain rather than the actor type occupying it. Any actor capable of satisfying a layer's functional requirements may occupy that layer: human, AI, mechanical, institutional, or hybrid.

This actor-agnostic definition is the framework's most important contribution. It makes **BX3**applicable to human-only organizations, fully automated pipelines, multi-agent AI architectures, and hybrid compositions that do not yet have names. In each case, the framework asks the same three questions: Is there an accountable layer? Is there a bounded layer that reasons and proposes? Is there a deterministic layer that enforces and audits? If any layer is missing, the system is architecturally incomplete — regardless of how capable its components are individually.

The Five Pillars transform these layer definitions from abstract principles into enforceable properties:

1. **Loop Isolation** makes Logic Collision architecturally impossible.
2. **Recursive Spawning** enables local autonomy without autonomous drift.
3. **Spatial Firewall** makes cross-tier data leakage physically impossible.
4. **Sandbox Gate** enables pre-execution validation of every proposed action.
5. **Bailout Protocol** guarantees that no unresolvable condition terminates at a machine actor.

The pattern is visible in legal systems, medical practice, autonomous vehicles, biological cognition, and organizational design — wherever reliable systems have been built to handle a world that is simultaneously rule-bound and unpredictable. What is new is the urgency of making the pattern explicit at a moment when the temptation to collapse these roles, and the cost of doing so, have never been higher.

The framework's formal model, BNF grammar, and pseudo-code implementations provide enough specificity for engineers to build **BX3**-compliant systems and for regulators to audit

them. The framework does not replace domain expertise or judgment — it organizes the structural requirements that make both possible.

The framework's practical value is already demonstrated in production deployments across agricultural irrigation (30% water reduction), medical diagnostics (regulatory-compliant AI assistance), and autonomous freight (human-accountable ADS deployment). These deployments confirm that the framework is not merely theoretical but operationally viable.

As autonomous systems continue to proliferate across safety-critical domains, the frameworks used to govern them will shape the balance of power between human accountability and machine autonomy. **BX3**provides a principled foundation for that governance: one that treats accountability as non-negotiable, reasoning as bounded, and enforcement as deterministic. The alternative — unconstrained autonomous systems with no guaranteed path to human accountability — is a risk the framework explicitly refuses to accept.

Acknowledgments

The author acknowledges the foundational contributions of researchers cited herein, whose work across control theory, sociotechnical systems, and computer science provides the foundation for this synthesis. The author also acknowledges the contributions of the agentic AI research community, whose rapid progress has made the need for principled architectural frameworks both urgent and tractable.

Data Availability

All materials related to this work, including the formal model specification, BNF grammar, and reference implementations, are available at the Zenodo repository cited in the bibliography.

18.1 Funding Disclosure

This work was conducted without external funding. All research and development was funded by Bxthre3 Inc. The author is the founder and chief architect of Bxthre3 Inc., which develops the BX3 Framework for internal use and potential commercialization.

18.2 Conflict of Interest Statement

Jeremy Blaine Thompson Beebe is the founder and chief architect of Bxthre3 Inc., the organization that developed and holds intellectual property rights to the BX3 Framework. No other conflicts of interest are declared.

19 Glossary of Key Terms

For the convenience of readers from different disciplines, we provide definitions for key terms used throughout this paper.

Accountability Anchor: The human or institutional entity that bears final responsibility for the outcomes of a **BX3**system. The accountability anchor cannot be an AI system under any circumstances.

Bailout Protocol: The mandatory exception propagation mechanism in **BX3**systems that ensures unresolvable conditions are escalated to human accountability anchors, bypassing all intermediate machine actors.

Bounds Engine: The intermediate functional layer in the **BX3**architecture, responsible for reasoning, proposal generation, and constrained execution. The Bounds Engine is architecturally limbless — it proposes but never executes.

Determinism: The property of a system whereby identical inputs always produce identical outputs, without exception. Determinism enables formal verification, regulatory certification, and forensic auditability.

Fact Layer: The lowest functional layer in the **BX3**architecture, responsible for deterministic physical execution, hard constraint enforcement, and immutable audit logging. The Fact Layer is the only layer that can act on the physical world.

Human Root Mandate: The architectural requirement that the Purpose Layer must always be anchored to a human accountability entity, and that accountability for system outcomes cannot dissipate into algorithmic systems.

Limblessness: The architectural property of the Bounds Engine whereby it has no direct authority to execute actions on the physical world. Limblessness is enforced by Loop Isolation, which separates proposal authority from execution authority.

Loop Isolation: The first architectural pillar of **BX3**, enforcing strict separation between the Bounds Engine and the Fact Layer through a typed message queue. Loop Isolation makes Logic Collision architecturally impossible.

Logic Collision: The failure mode that occurs when the reasoning function and the execution function of an autonomous system occupy the same functional plane, enabling un-vetted autonomous actions that bypass physical constraint.

Recursive Spawning: The second architectural pillar of **BX3**, enabling child nodes to inherit a parent's Purpose through a containerized Worksheet, allowing autonomous operation at edge nodes without requiring persistent cloud connectivity.

Safety Envelope: The formal specification of constraints on Bounds Engine proposals, defined by the Purpose Layer. All proposals must satisfy Safety Envelope constraints before being evaluated by the Sandbox Gate.

Sandbox Gate: The fourth architectural pillar of **BX3**, requiring that all proposed actions be simulated in a digital twin before physical execution. The Sandbox Gate validates that proposals satisfy Safety Envelope constraints before unlocking execution.

Spatial Firewall: The third architectural pillar of **BX3**, enforcing physical isolation of data planes by resolution tier. A node provisioned at one resolution tier cannot access data at finer resolution tiers, because those data planes do not exist in its instantiated environment.

Worksheet: A containerized, self-contained logic set generated by a parent Bounds Engine for deployment to a child node. The Worksheet encapsulates the parent's Purpose for a specific local context and includes a cryptographically verifiable parent pointer.

20 Appendix: About the Author

Jeremy Blaine Thompson Beebe is the founder and chief architect of Bxthre3 Inc., where he leads development of the BX3 Framework and its applications in AI workforce orchestration, precision agriculture, and autonomous systems governance. He holds an ORCID identifier (0009-0009-2394-9714) and can be reached via email at bxthre3inc@gmail.com. The author's background spans distributed systems, control theory, and sociotechnical systems design. The BX3 Framework emerged from practical experience deploying AI systems in safety-critical agricultural and industrial environments where the absence of principled accountability architecture became a blocking issue.

21 Appendix: Reproducibility and Open Source

The BX3 Framework specification including the formal model, BNF grammar, and pseudocode implementations is available under a permissive open-source license. Implementations are provided in TypeScript (for Node.js and Bun environments) and Python (for research and integration contexts). The reference implementation includes:

- Complete formal model encoding in TLA+ for model checking with TLC
- Reference implementation of all five pillars (Loop Isolation, Recursive Spawning, Spatial Firewall, Sandbox Gate, Bailout Protocol)
- Compliance Monitor reference implementation
- Example configurations for agricultural irrigation, medical diagnostics, and financial trading domains
- Test suites covering all layer constraints (P1–P3, B1–B3, F1–F3) and pillar properties

The reference implementation is designed for educational and production use. It is available at the Bxthre3 Inc. public GitHub repository (github.com/bxthre3inc) under the BX3 Framework repository. The implementation is designed to be composable: organizations may implement individual pillars independently or adopt the full framework depending on their requirements and risk tolerance.

22 Appendix: Change Log and Version History

This appendix documents the version history of the BX3 Framework specification. The framework evolves based on feedback from practical deployments, regulatory developments, and academic review.

22.1 Version 1.0 (April 2026)

Initial public release. Includes the three-layer architecture (Purpose Layer, Bounds Engine, Fact Layer), five pillars (Loop Isolation, Recursive Spawning, Spatial Firewall, Sandbox Gate, Bailout Protocol), formal model, BNF grammar, and reference implementations in TypeScript and Python. Aligned with ISO/IEC 42001 and NIST AI RMF for regulatory compatibility.

22.2 Version 0.9 (February 2026)

Pre-release for internal review. Included three-layer architecture and first three pillars (Loop Isolation, Recursive Spawning, Spatial Firewall). Bailout Protocol and Sandbox Gate added in response to safety review feedback.

22.3 Version 0.5 (November 2025)

Early draft for Bxthre3 Inc. internal architecture review. Demonstrated feasibility in agricultural irrigation deployment (30% water reduction). Formal model and BNF grammar added based on engineering team feedback.

The framework follows semantic versioning (semver.org) for releases. Major version changes indicate architectural changes that are not backward-compatible. Minor version changes indicate additions that are backward-compatible. Patch versions indicate documentation corrections and errata.

Users of the framework are encouraged to track the change log to understand the evolution of the specification and to provide feedback on proposed changes via the GitHub repository issue tracker.

23 Appendix: Notation and Typography Guide

This paper uses a consistent notation scheme throughout. For readers unfamiliar with the formal notation, this guide explains the symbols and typography used.

Mathematical notation uses standard set theory and logic symbols: $\forall$ (for all), $\in$ (element of), $\implies$ (implies), $\nexists$ (there does not exist), $\cup$ (union), $\subset$ (subset). Sets are written in calligraphic font ($\mathcal{C}$, $\mathcal{G}$, $\mathcal{S}$, $\mathcal{H}$). Layer components are written as capital letters (P for Purpose Layer, B for Bounds Engine, F for Fact Layer, H for Human Accountability Anchor). Greek letters are used for parameters (μ for mission function, σ for system state, π for proposal, τ for attribution chain, Σ for the complete system).

The framework name BX3 is styled in bold (**BX3**) to distinguish it from surrounding text. Layer names (Purpose Layer, Bounds Engine, Fact Layer) are styled in italics when used as nouns in prose. Constraint names (P1, P2, P3, B1, B2, B3, F1, F2, F3) are styled in bold and referred to throughout the paper.

Pillar names (Loop Isolation, Recursive Spawning, Spatial Firewall, Sandbox Gate, Bailout Protocol) are capitalized in title case throughout. The terms are treated as proper nouns specific to the BX3 Framework.