CPUX: Cognitive Execution Paths Without Hidden Logic—Logic Through Perception

Pronab Pal[1]

[1]Affiliation not available

December 05, 2025

Keybyte Systems, Intentix Lab , Melbourne, Australia

pronab@keybytesystems.com.au Supported by AusIndustry Grant IR2405165

Abstract

Modern cloud-native applications distribute business logic across multiple layers: application code, orchestration frameworks, service meshes, and infrastructure configurations. This distribution creates "hidden logic"—execution rules embedded in infrastructure that are invisible during design and difficult to trace at runtime. We present **Intention Space** , a computing model built on the **CPUX (Common Path of Understanding and Execution)** paradigm that consolidates all business logic into explicit, design-time declarations using plain-language state pulses. In our model, Design Nodes (DNs) contain computation while Gatekeepers declare execution conditions as named pulses (e.g., "payment validated": Y). The infrastructure provides only mechanical enforcement through an Intention Loop that matches runtime state to Gatekeepers without adding decision logic. We demonstrate that complex workflows—traditionally requiring nested if-then branching and explicit loops—can be expressed as linear CPUX sequences where execution paths emerge from data state rather than code branching. Our Golang implementation shows complete elimination of orchestration code while maintaining full cognitive traceability. Beyond technical innovation, CPUX addresses a critical social computing crisis: the lack of accountability in distributed social platforms. By creating unique, device-level CPUX footprints for every interaction, our model enables verifiable traceability from device identity through user intention to executed action—restoring accountability to social computing while preserving privacy. We argue this separation of intent (CPUX) from enforcement (infrastructure) is essential for building LLM-integrated, auditable, and socially responsible distributed systems.

Keywords: CPUX, Intention Space, Design Nodes, Cognitive Computing, Data-Driven Execution, Microservices Architecture, Cloud Computing, LLM Integration, Social Computing Accountability

---

1. Introduction

1.1 The Hidden Logic Problem

Consider a typical e-commerce order processing system deployed on Kubernetes with Istio service mesh:

```
// order-service/main.go (Business Logic Layer) func ProcessOrder(order Order) error {
if order.Amount > 1000 { if err := premiumValidator.Validate(order); err != nil { return
retry(premiumValidator.Validate, 3, order) } } else { standardValidator.Validate(order) }
// ... more branching logic }

# k8s/hpa.yaml (Infrastructure Layer) spec: metrics: - type: Resource resource: name: cpu
target: type: Utilization averageUtilization: 80 # Hidden rule: Scale when CPU > 80%

# istio/retry-policy.yaml (Service Mesh Layer) spec: http: - retries: attempts: 3
perTryTimeout: 2s # Hidden rule: Retry 3 times on failure
```

Question: What is the complete execution flow for a $1500 order that fails validation on first attempt?

Answer: One must read and correlate:

1. Application code (branching logic)
2. Kubernetes manifests (scaling rules)
3. Istio configurations (retry policies)
4. Service mesh observability logs (runtime behavior)

This **hidden logic distribution** creates fundamental problems:

- **Traceability** : No single artifact shows complete flow
- **Testability** : Must test infrastructure + code interactions
- **Auditability** : Business stakeholders cannot validate logic
- **Maintainability** : Changes require coordinating multiple layers
- **LLM Integration** : No structured representation for AI reasoning
- **Social Accountability** : Cannot trace interactions to source devices/users

1.2 The Core Insight

We observe that traditional computing conflates two distinct concerns:

```
What should happen(business intent)

How to make it happen(mechanical execution)
```

Current architectures intertwine these concerns across code, configuration, and infrastructure, making systems cognitively opaque.

Our Contribution: We introduce **CPUX (Common Path of Understanding and Execution)** , a paradigm that separates business intent from infrastructure enforcement:

- **CPUX Structure** : Declares all possible execution paths as sequences of Design Nodes (DNs) with plain-language Gatekeeper conditions
- **Infrastructure** : Provides mechanical execution (Intention Loop) that enforces CPUX declarations without adding decision logic
- **Device-Level Identity** : Each CPUX execution tied to unique device fingerprint + user intention, enabling social computing accountability
- **Result** : Complete business logic is visible in CPUX; infrastructure remains purely mechanical; every social interaction is traceable

1.3 Key Contributions

1. **Formal Model** : CPUX as cognitive execution contract with Design Nodes, Intentions, Objects, and Pulses as primitive components
2. **Elimination of Hidden Logic** : All business decisions visible in design-time CPUX declarations; infrastructure adds zero decision logic

3. **Plain-Language State Declarations** : Execution conditions expressed as named pulses (e.g., "inventory confirmed": Y) enabling business stakeholder review and LLM integration
4. **Data-Driven Execution** : Runtime branching eliminated from code; execution paths emerge from pulse state matching via SyncTest
5. **Social Computing Accountability** : Device-level CPUX fingerprints create unique, traceable identity for every social interaction, addressing the accountability crisis in platforms like Facebook, Twitter, TikTok
6. **Implementation & Evaluation** : Golang framework code sample with concrete use case demonstrating zero orchestration code while maintaining full traceability

1.4 Paper Organization

Section 2 examines related work. Section 3 presents the PnR computing model and CPUX formalism. Section 4 details the architecture and implementation. Section 5 evaluates our approach through metrics and case studies. Section 6 discusses LLM integration. Section 7 introduces CPUX for social computing accountability—the urgent global need. Section 8 concludes with future directions.

---

2. Related Work

2.1 Workflow Orchestration Systems

AWS Step Functions [1] and **Azure Logic Apps** [2] provide visual workflow definition with explicit state machines. However, they:

- Use proprietary JSON/XML DSLs (not plain language)
- Embed conditional logic in workflow definitions (still branching)
- Remain platform-specific (vendor lock-in)
- Require reading workflow definitions to understand flow
- Cannot trace to device/user identity

Apache Airflow [3] and **Temporal** [4] define workflows as code with DAG structures. They improve on step functions ,recoverability but:

- Business logic still in code (if-then branches)
- Workflow orchestration separate from execution
- No plain-language condition declarations
- No device-level traceability

CPUX Advantage : All logic in plain-language pulses, platform-agnostic, no explicit branching in declarations, device-level identity for every execution, recoverability built into platform.

2.2 Service Mesh & Orchestration

Istio [5] and **Linkerd** [6] provide traffic management, retries, circuit breaking. **Kubernetes Operators** [7] encode reconciliation logic. These systems:

- Hide business rules in YAML configurations
- Distribute logic across mesh config + operator code
- Focus on infrastructure concerns (not business flow)
- Lack unified view of complete execution path
- No user/device attribution

CPUX Advantage : Consolidates all execution logic in CPUX; infrastructure config aligned with business intent; device identity integral.

2.3 Event-Driven Architectures

Apache Kafka [8], **AWS EventBridge** [9] enable event-driven systems with loose coupling. **Reactive systems** [10] promote message-passing. However:

- Event flows implicit (must trace message paths)
- Conditional logic in event handlers (code-level branching)
- No design-time declaration of all possible flows
- No provenance tracking to source device

CPUX Advantage : Explicit declaration of all event-driven paths as DN sequences with visible Gatekeepers; device identity in event provenance.

2.4 Intent-Based Systems

Intent-Based Networking [11] translates high-level intents to network configurations. **Policy-based management** [12] separates policy from mechanism. Closest to our work, but:

- Focus on infrastructure (not application logic)
- Policies often domain-specific (not general computing)
- Limited plain-language expressiveness
- No user accountability

CPUX Advantage : General-purpose computing model with full plain-language pulse declarations applicable to any domain; device-level user accountability.

2.5 Formal Methods & Model Checking

TLA+ [13], **Alloy** [14], and **Petri Nets** [15] enable formal specification and verification. These are powerful but:

- Require specialized formal notation (high learning curve)
- Specification separate from implementation (sync problems)
- Not designed for runtime execution
- No social computing traceability

CPUX Advantage : Declarations are executable; CPUX structure IS the implementation contract; device identity embedded.

2.6 Social Computing & Accountability

Blockchain-based identity [16] and **zero-knowledge proofs** [17] address digital identity but:

- Focus on cryptographic primitives (not execution tracing)
- Don't integrate with application logic
- No cognitive representation of intent

Federated social networks [18] (Mastodon, ActivityPub) improve decentralization but:

- Still lack device-level traceability
- No structured intent representation
- Cannot prove user intended specific action

CPUX Advantage : First system to integrate device identity, user intention, and execution trace in single cognitive framework.

2.7 Positioning

CPUX is the first system to combine:

1. Plain-language execution conditions (like Intent-Based Networking)
2. Executable specifications (unlike formal methods)
3. Complete flow visibility (unlike distributed orchestration)
4. Zero hidden infrastructure logic (unique contribution)
5. Device-level social accountability (unique contribution)

---

3. The PnR Computing Model

3.1 Core Abstractions

3.1.1 Pulse: Atomic State Unit

A **Pulse** is the fundamental data unit representing a named state with optional response and trivalence:

Pulse = (Name: String, Response: Value, Trivalence: {Y, N, U})

- **Name** : Plain-language identifier (e.g., "payment validated")
- **Response** : Optional value (e.g., transaction ID)
- **Trivalence** : Y (yes/true), N (no/false), U (undecided)

Example:

{ "name": "payment validated", "response": "txn_1234567", "trivalence": "Y" }

Design Principle: Pulses are declared at design time, instantiated at runtime. All possible system states are explicit in pulse declarations.

3.1.2 Signal: Immutable Pulse Collection

A **Signal** is an immutable collection of Pulses:

Signal = {Pulse1, Pulse2, . . . , Pulse[?]}

Immutability Constraint: Outside Design Nodes, Signals cannot be modified—only copied with transformations declared via mapping operations.

3.1.3 Intention: Signal Carrier

An **Intention** carries a Signal between components:

Intention = (Name: String, Signal: Signal, Timestamp: Time)

Intentions are like light rays in physical space—they carry information (Signal) but don't modify it in transit.

3.1.4 Design Node (DN): Computation Container

A **Design Node** is a black-box directional computational unit:

DN = (Name: String, Gatekeeper: Signal, // Pre-condition FlowinNames: [String], // Input pulse names FlowoutNames: [String], // Output pulse names Process: Signal - Signal // Computation)

Key Properties:

- **Black Box** : Internal computation hidden; only interface matters

- **Gatekeeper** : Declares when DN should execute (plain language)
- **Process** : THE ONLY PLACE COMPUTATION HAPPENS
- **Pure** : Same input Signal - same output Signal (functional)
- **Directional:** forward states in the Object ahead in the CPUX sequence

Example:

```
DN_ValidatePayment := DesignNode{ Name: "Validate Payment", Gatekeeper: Signal{ {"order received", true, "Y"}, {"payment validated", false, "U"}, }, FlowinNames: ["order received", "payment method", "amount"], FlowoutNames: ["payment validated", "transaction id"], Process: func(input Signal) Signal { // Actual validation logic here (black box) return Signal{ {"payment validated", true, "Y"}, {"transaction id", "txn_123", "Y"}, } }, }
```

3.1.5 Object (O): State Holder & Reflector

An **Object** persists state and performs declarative Signal mapping:

O = (Name: String, State: Signal, // Persisted state Mapping: MappingConfig // Declarative transformation)

Mapping Operations (declarative, no procedural code):

- **Union** : S1 [?] S2
- **Intersection** : S1 [?] S2
- **Difference** : S1 \ S2
- **Rename** : {pulse_name - new_pulse_name}

Objects are like mirrors to light rays—they reflect Intentions (change direction) and can map Signals declaratively. Objects can start a new CPUX if the reflected interface is configured as a CPUX starter.

3.2 Execution Unit: I1-DN-I2-O-I3

The fundamental computation unit is:

I1 - DN - I2 - O - I3

Flow:

1. **I1** carries input Signal to DN
2. **DN** performs computation (only place logic executes)
3. **I2** carries DN's output to Object (now immutable)
4. **O** reflects I2 - I3 via declarative mapping (no computation)
5. **I3** carries transformed Signal to next DN

Critical Property: Signals are immutable outside DNs. Objects only rearrange/rename Pulses declaratively.

3.3 CPUX: Common Path of Understanding and Execution

A **CPUX** is a ordered sequence of execution units:

CPUX = [DN1, O1, DN2, O2, DN3, O3, . . . , DN[?]]

Expanded:

I0 - DN1 - I1 - O1 - I2 - DN2 - I3 - O2 - I4 - DN3 - . . .

Key Properties:

1. **Linear Structure** : DNs arranged in ordered sequence (branching only through another CPUX)
2. **Explicit Dependencies** : Each DN's Gatekeeper shows what it needs
3. **Cognitive Meaning** : Sequence represents business process flow
4. **Visible Paths** : All possible execution paths visible as parallel DNs
5. **Unique Identity** : Each CPUX execution instance has unique address

Example:

```
OrderProcessing_CPUX = [ DN_ReceiveOrder, O_OrderState, DN_ValidatePayment_Credit, //
Parallel path 1 DN_ValidatePayment_PayPal, // Parallel path 2 O_PaymentState, DN_-
CheckInventory, O_InventoryState, DN_ShipOrder, ]
```

At design time , this CPUX shows:

- Four business steps (Receive, Validate, Check, Ship)
- Two payment validation options (Credit/PayPal)
- Dependencies via Gatekeepers (visible in DN declarations)

At runtime , Intention Loop determines which path executes based on current pulse state—NO code-level if-then needed.

3.4 SyncTest: Data-Driven Gatekeeper Matching

SyncTest is the ONLY decision point in the infrastructure:

SyncTest(Gatekeeper: Signal, Visitor: Signal) - Boolean

Algorithm:

For each pulse P in Gatekeeper: Find matching pulse P' in Visitor (by name) If not found: return False If P.Trivalence [?] P'.Trivalence: return False Return True

Key Properties:

- **Data-Driven** : Decision based purely on state matching
- **No Logic** : No if-then business rules in SyncTest
- **Declarative** : Gatekeeper declares conditions; SyncTest just matches
- **Traceable** : Match result explainable (which pulses matched/failed)

Example:

```
Gatekeeper = Signal{ {"payment validated", true, "Y"}, {"inventory confirmed", true,
"Y"}, }   Visitor = Signal{ {"order received", true, "Y"}, {"payment validated", true,
"Y"}, {"inventory confirmed", true, "Y"}, }   SyncTest(Gatekeeper, Visitor) - True // All
pulses match
```

3.5 Intention Loop: Mechanical Executor

The **Intention Loop** is the infrastructure execution engine:

```
Algorithm IntentionLoop(CPUX, RuntimeSignal): repeat: anyDNExecuted = False for
each DN in CPUX.DesignNodes: if SyncTest(DN.Gatekeeper, RuntimeSignal): // Execute
DN I1 = Intention("i1", RuntimeSignal) I2 = DN.Execute(I1)   // Object reflects O
= CPUX.Objects[DN.index] I3 = O.Reflect(I2, "i3")   // Merge into runtime state
RuntimeSignal = Merge(RuntimeSignal, I3.Signal)   anyDNExecuted = True   if not
anyDNExecuted: break // Self-termination
```

Critical Properties:

1. **Mechanical** : Only matches state; adds no business decisions
2. **Exhaustive** : Tries all DNs every pass
3. **Self-Terminating** : Stops when no DN qualifies
4. **Deterministic** : Same initial state - same execution path
5. **Traceable** : Every DN execution logged with state snapshot

The ONLY loop in the entire system is this mechanical Intention Loop. No loops in business logic.

3.6 Formal Model

3.6.1 Components

Define sets:

- **P** : Set of all Pulses (declared at design time)
- **DN** : Set of all Design Nodes
- **O** : Set of all Objects
- **I** : Set of all Intentions
- **CPUX** : Set of all execution paths

3.6.2 Execution Semantics

A CPUX execution is a state transition function:

Execute: (CPUX, Signal_initial) - Signal_final Where: - Signal_initial: Initial runtime state - Signal_final: Final state after CPUX completion - Transition = sequence of DN executions based on SyncTest

3.6.3 Theorem: Bounded Complexity

Theorem 1 (Design-Time Complexity Bound):

The maximum number of execution paths in a CPUX is bounded by |DN|, the number of Design Nodes declared at design time.

`Proof:`

`Each DN represents one possible execution step. The Intention Loop tries all DNs in sequence. At most |DN| steps can execute (assuming each DN executes once). Therefore, execution complexity is O(|DN|), determined entirely at design time. [?]`

Theorem 2 (No Hidden Branching):

If a DN D executes, there exists a visible Gatekeeper G in the CPUX declaration such that SyncTest(G, RuntimeState) = True.

`Proof:`

`By construction, DN.Execute() is only called after SyncTest passes (see Intention Loop algorithm). Every Gatekeeper is declared in DN definition. Therefore, all execution triggers are visible in CPUX declarations. [?]`

Corollary : No execution path exists that is not visible in CPUX structure.

3.7 Design Principle: Cognitive Contract

CPUX is a Cognitive Contract : An explicit, design-time agreement between:

- **Business Stakeholders** : Can read plain-language pulses/gatekeepers
- **Developers** : Implement DN.Process functions
- **Infrastructure** : Mechanically enforces via Intention Loop

- **Auditors** : Verify completeness by reviewing CPUX
- **End Users** : Their device + intention creates unique CPUX fingerprint

Contract Properties:

1. **Completeness** : All execution paths in CPUX
2. **Visibility** : All conditions in Gatekeepers
3. **Traceability** : Every execution maps to DN in CPUX
4. **Accountability** : Changes tracked via CPUX version control
5. **Device Attribution** : Each execution tied to device identity

Principle : *"If it's not in the CPUX, it doesn't execute."*

3.8 Philosophical Foundation: Logic as Data

The fundamental inversion in PnR computing is treating logic as data, not code. In traditional computing, logic is embedded in control flow constructs—if-then-else branches, loops, and function call hierarchies. In CPUX, logic is externalised as Pulse states and Gatekeeper declarations: literal data structures that the Intention Loop reads and matches mechanically.

This inversion has profound implications for how we conceptualise computation:

Intent-Driven Directional Elements: Traditional functions and code modules become intent-driven directional elements within the CPUX structure. Design Nodes do not "decide" what to do next; they simply transform input Signals to output Signals when their Gatekeeper conditions are satisfied. The direction of computation emerges from the Intention that carries Signals forward—the DN remains passive until activated by state matching.

Forward State Creation: Design Nodes do not modify existing state; they produce new Pulses that flow forward via Intentions and Objects. State progression is always additive and directional, never mutating-in-place. Each computational step creates forward states that become the preconditions for subsequent steps.

Pre-Configured Declarative Logic: The entire execution topology is declared at design time within the CPUX structure. Runtime does not invent new paths; it merely activates pre-declared paths based on state matching via SyncTest. All possible execution flows are visible before a single line of code executes.

Immutability as Architectural Constraint: Outside the Design Node's Process function (the black box where actual computation occurs), everything remains immutable. Signals flow, Objects reflect, but nothing mutates. Even the Object's mapping facility performs declarative transformation, not procedural mutation.

This formulation—logic as data, functions as directional elements, computation as forward state creation through declarative configuration—represents a return to the algebraic roots of computing, where transformation is explicit, traceable, and cognitively transparent.

3.7 The Design Node and the Freedom of Logic from code syntax

Syntax-Free Logic Activation

Traditional logic is inseparable from syntax. The expression `if (x && y) { doThis() }` binds meaning to grammatical structure—parentheses, operators, keywords, nesting. Change the syntax, break the logic. This coupling creates cognitive load and limits who can read, validate, or reason about system behaviour.

In PnR computing, logic reduces to perception: the presence or absence of Pulses. There are no operators—AND is implicit, as all Gatekeeper Pulses must match. There are no conditional keywords, no nested expressions, no precedence rules. The SyncTest mechanism does not parse a grammar; it pattern-matches state. Activation depends solely on whether named Pulses exist with the required Trivalence.

This syntax-free activation has profound implications:

- **Human readability**: Business stakeholders perceive preconditions as plain-language state requirements, not code constructs
- **LLM compatibility**: Language models reason naturally about "what Pulses exist?" rather than parsing control flow trees
- **Language independence**: The same Gatekeeper declaration works whether the DN is implemented in Go, Rust, Python, or any other language

This final point enables a powerful architectural capability: Design Nodes written in different languages can cooperate through realised state orchestration. A DN implemented in Go can produce Pulses that trigger a DN implemented in Rust, which in turn activates a DN written in Python. No language bindings, no foreign function interfaces, no serialisation protocols beyond the Pulse structure itself. The Intention Loop perceives only state—it is agnostic to the language that produced it.

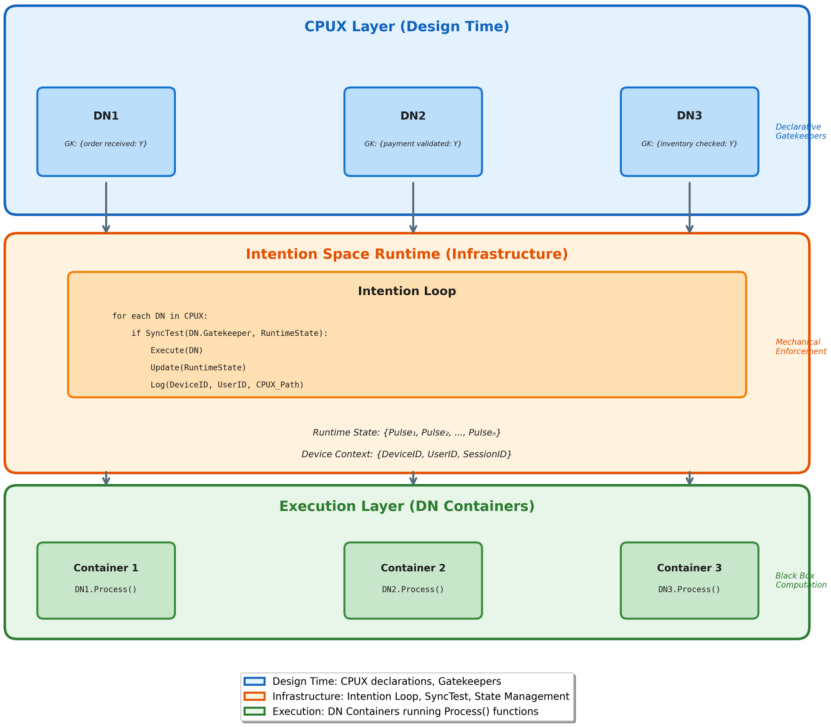
Logic, in this model, is not something you write in code. It is something you declare as required state, and the system perceives its realisation.

---

4. Architecture & Implementation

4.1 System Architecture

Figure 1: CPUX System Architecture



4.2 Golang Implementation

The full implementation of Intention Space as a platform is in progress at Keybyte Systems. Key components include:

- **Pulse & Signal** : Immutable data structures with deep copy enforcement
- **Intention** : Carrier with device/user attribution
- **Design Node** : With SyncTest and execution logging
- **Object** : Declarative reflection with state persistence
- **CPUX** : Execution path with device fingerprinting and audit trail

4.3 Design Node Template

A Design Node functions as a waiting worker that only acts when the right conditions appear. Unlike traditional functions that are called directly, a Design Node declares what it needs, what it produces, and what it does—then waits for the Intention Loop to activate it.

One DN, Multiple Intent Pairs

A Design Node is unique within Intention Space, but it is not limited to a single purpose. A DN can receive multiple distinct Intentions and emit corresponding outgoing Intentions—including exception cases. However, when a DN participates in a CPUX, the cognitive design pairs one specific incoming intent with one specific outgoing intent. This pairing is what makes CPUX flows readable and traceable.

```
Design Node (unique in Intention Space):


Intent A  -->  DN  -->  Intent A'

Intent B  -->  DN  -->  Intent B'

Intent C  -->  DN  -->  Exception Intent


CPUX (cognitive pairing):


... --> Intent A --> DN --> Intent A' --> ...
```

The Object following a DN in a CPUX sequence is able to start a new CPUX if it reflects an Intention that starts a new CPUX.

The infrastructure supports treating each execution unit and each CPUX independently, but those details remain outside the scope of this paper.

The Three Declarations

Every Design Node template comprises three declarations

```
1. GATEKEEPER: "I wake up when I see these Pulses..."

2. FLOWIN:     "When I wake, give me these values..."

3. FLOWOUT:    "When I finish, I produce these new Pulses..."
```

Template Structure in Go

```
var MyWorker = DesignNode{

 // WHAT I NEED TO SEE (my trigger conditions)
Gatekeeper: Signal{
{"order received", true, "Y"},
{"payment validated", false, "U"},  // Not yet done
},

// WHAT I TAKE IN (my inputs)
FlowinNames: []string{"order received", "amount"},

// WHAT I PRODUCE (my outputs)
FlowoutNames: []string{"payment validated", "transaction id"},

// WHAT I DO (pure transformation, no side effects)
Process: func(in Signal) Signal {
// Your logic here - but remember:
// - Read ONLY from 'in'
// - Return NEW pulses, don't modify anything
// - No calling other services, no database writes

return Signal{
{"payment validated", true, "Y"},
{"transaction id", "txn_12345", "Y"},
}
},
}
```

Three rules maintained by DN

1. **You do not call it** — The Intention Loop calls it when Gatekeeper matches
2. **You do not branch inside it** — One input shape, one output shape; exception cases are separate intent pairings, not hidden branches

3. **You do not mutate** — Take Pulses in, produce new Pulses out, touch nothing else

Benefits of the Template

When every Design Node follows this template:

- The system knows all possible paths at design time
- No hidden logic—Gatekeepers are readable English phrases
- Exception flows are explicit intent pairings, not buried catch blocks
- Testing becomes trivial: given these Pulses, expect those Pulses
- Tracing is automatic: every step logged with its trigger reason

---

5. Overall Evaluation at December 2025

We have used the CPUX methodology through several languages like JS ,Go and Rust and have been encouraged by development of Grid design lookout as a common user interaction interface with Intention Space across different platforms. We are aiming to bring the platform for general use in the early part of 2026.

---

6. LLM Integration

CPUX's plain-language pulses provide natural LLM interface ,we shall bring as part of the platform:

- LLMs generate Gatekeepers from business rules
- Design Nodes can wrap LLM calls
- Hybrid workflows mix traditional code + LLM
- Future: LLM-generated complete CPUX from natural language

---

7. CPUX for Social Computing Accountability

7.1 The Crisis we face:

online harassment is on the rise,

majority of misinformation from <0.1% of accounts

enormous economic harm from social media abuse

Root Problem : No way to verify who did what, from where, with what intent

7.2 Solution: Device-Level CPUX Fingerprints

Through Intention Space ,every social interaction gets unique, traceable CPUX:

```
{ "cpux_id": "SOCIAL_POST_001", "device_id_hash": "SHA256(MAC...)", "user_id": "user_-
12345", "intention_chain": ["compose_post", "send_post"], "pnr_state": { "user
clicked post": "Y", "moderation passed": "Y" }, "timestamp": "2024-12-05T10:30:00Z",
"cryptographic_signature": "..." }
```

Benefits:

- **Irrefutable identity** : Device + User + Intention
- **Intent proof** : User explicitly clicked "Post"
- **Platform accountability** : Moderation decisions visible
- **Court-admissible** : Forensic evidence
- **Bot detection** : Patterns visible in CPUX

7.3 Privacy vs. Accountability

Intention Space brings in Tiered Disclosure:

- **Public** : Hashed IDs, intention sequences
- **Platform** : Real device/user IDs (for moderation)
- **Legal** : Full details (court subpoena only)

7.4 Implementation Roadmap

We are building the platform with the overall goal

- **2024-2025** : Standards development
- **2025-2026** : Pilot deployments on mid-size platforms
- **2026-2027** : Regulatory mandates
- **2027-2030** : Major platform adoption, Global standard, bot networks eliminated

---

8. Discussion & Future Work

Current Limitations we are addressing:

- SyncTest overhead (mitigated by parallel evaluation)
- Learning curve (mitigated by visual editors)
- Storage costs (mitigated by time-limited retention)

Immediate Future Work:

- Visual CPUX editors with device preview
- Formal verification with device spoofing detection
- LLM-native CPUX generation
- Cross-platform device tracking standards
- Privacy-enhancing technologies (zero-knowledge proofs)

---

9. Conclusion

CPUX eliminates hidden infrastructure logic through:

1. **Design-time declarations** in plain language
2. **Mechanical enforcement** by infrastructure
3. **Device-level accountability** for social computing

Technical Impact : Zero orchestration code, full traceability, LLM integration

Societal Impact : Restores accountability to social computing while preserving privacy

The choice : Implement CPUX standards now, or wait for the next crisis?

The technology exists. The need is urgent. The time to act is now.

References

[1] AWS Step Functions. https://aws.amazon.com/step-functions/

[2] Azure Logic Apps. https://azure.microsoft.com/services/logic-apps/

[3] Apache Airflow. https://airflow.apache.org/

[4] Temporal. https://temporal.io/

[5] Istio Service Mesh. https://istio.io/

[6] Linkerd. https://linkerd.io/

[7] Kubernetes Operators. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[8] Apache Kafka. https://kafka.apache.org/

[9] AWS EventBridge. https://aws.amazon.com/eventbridge/

[10] Reactive Manifesto. https://www.reactivemanifesto.org/

[11] Clemm, A., et al. "Intent-Based Networking." RFC 9315, 2022.

[12] Verma, D. "Policy-Based Networking." New Riders, 2000.

[13] Lamport, L. "Specifying Systems: The TLA+ Language and Tools." Addison-Wesley, 2002.

[14] Jackson, D. "Software Abstractions: Logic, Language, and Analysis." MIT Press, 2012.

[15] Murata, T. "Petri Nets: Properties, Analysis and Applications." Proceedings of the IEEE, 1989.

[16] Pal, P. (2024). "Human Intention Space: Natural Language Phrase Driven Approach to Place Social Computing Interaction in a Designed Space." IJNLC, Vol.13, No.3.

[17] Nakamoto, S. (2008). "Bitcoin: A Peer-to-Peer Electronic Cash System."

[18] Lemmer-Webber, C., et al. (2018). "ActivityPub." W3C Recommendation.

[19] Pew Research Center. (2021). "The State of Online Harassment."

[20] Vosoughi, S., Roy, D., & Aral, S. (2018). "The spread of true and false news online." Science, 359(6380), 1146-1151.

[21] World Economic Forum. (2023). "The Global Risks Report."

[22] Zuboff, S. (2019). "The Age of Surveillance Capitalism." PublicAffairs.

[23] EU Digital Services Act. (2022). https://digital-strategy.ec.europa.eu/

- Appendix A: Runnable Golang code to run CPUX

```
// CPUX Design Node Demo - Simple Version
// Copy this to https://go.dev/play/to run
//
// This demonstrates the core PnR concept:
```

```
// - Pulses as named states with trivalence (Y/N/U)
// - Signals as collections of Pulses
// - SyncTest matching Gatekeeper against runtime state
// - Design Nodes triggered only when conditions match

package main

import "fmt"

// ============== CORE TYPES ==============

// Trivalence represents Y (yes/true), N (no/false), U (undecided)
type Trivalence string

const (
Y Trivalence = "Y"
N Trivalence = "N"
U Trivalence = "U"
)

// Pulse is the atomic state unit
type Pulse struct {
Name       string
Response   interface{}
Trivalence Trivalence
}

// Signal is an immutable collection of Pulses
type Signal []Pulse

// DesignNode is a waiting worker with Gatekeeper conditions
type DesignNode struct {
Name        string
Gatekeeper  Signal
FlowinNames []string
FlowoutNames []string
Process     func(Signal) Signal
```

```
}

// ============== SYNCTEST: THE ONLY DECISION POINT ==============

// SyncTest checks if all Gatekeeper pulses match the Visitor signal
func SyncTest(gatekeeper Signal, visitor Signal) bool {
for _, gkPulse := range gatekeeper {
found := false
for _, vPulse := range visitor {
if gkPulse.Name == vPulse.Name {
found = true
if gkPulse.Trivalence != vPulse.Trivalence {
return false // Trivalence mismatch
}
break
}
}
if !found {
return false // Required pulse not present
}
}
return true
}

// ============== HELPER FUNCTIONS ==============

// FindPulse retrieves a pulse by name from a signal
func FindPulse(sig Signal, name string) (Pulse, bool) {
for _, p := range sig {
if p.Name == name {
return p, true
}
}
return Pulse{}, false
}
```

```
// MergeSignals combines two signals (flowout overwrites existing)
func MergeSignals(base Signal, addition Signal) Signal {
result := make(Signal, len(base))
copy(result, base)

for _, newPulse := range addition {
found := false
for i, existing := range result {
if existing.Name == newPulse.Name {
result[i] = newPulse
found = true
break
}
}
if !found {
result = append(result, newPulse)
}
}
return result
}

// PrintSignal displays current state
func PrintSignal(label string, sig Signal) {
fmt.Printf("\n%s:\n", label)
for _, p := range sig {
fmt.Printf("  %-25s = %v [%s]\n", p.Name, p.Response, p.Trivalence)
}
}

// ============== DESIGN NODES ==============

var DN_ValidatePayment = DesignNode{
Name: "Validate Payment",
Gatekeeper: Signal{
{"order received", true, Y},
```

```
{"payment validated", false, U}, // Must be undecided
},
FlowinNames:  []string{"order received", "amount"},
FlowoutNames: []string{"payment validated", "transaction id"},
Process: func(in Signal) Signal {
// Simulate payment validation
return Signal{
{"payment validated", true, Y},
{"transaction id", "txn_98765", Y},
}
},
}

var DN_CheckInventory = DesignNode{
Name: "Check Inventory",
Gatekeeper: Signal{
{"payment validated", true, Y}, // Payment must be done
{"inventory checked", false, U}, // Inventory not yet checked
},
FlowinNames:  []string{"order received"},
FlowoutNames: []string{"inventory checked", "stock available"},
Process: func(in Signal) Signal {
return Signal{
{"inventory checked", true, Y},
{"stock available", true, Y},
}
},
}

var DN_ShipOrder = DesignNode{
Name: "Ship Order",
Gatekeeper: Signal{
{"payment validated", true, Y},
{"inventory checked", true, Y},
{"stock available", true, Y},
```

```
{"order shipped", false, U},
},
FlowinNames:  []string{"order received", "transaction id"},
FlowoutNames: []string{"order shipped", "tracking number"},
Process: func(in Signal) Signal {
return Signal{
{"order shipped", true, Y},
{"tracking number", "TRK-12345", Y},
}
},
}

// ============== INTENTION LOOP ==============

func IntentionLoop(nodes []DesignNode, runtime Signal) Signal {
pass := 0
for {
pass++
fmt.Printf("\n========== INTENTION LOOP PASS %d ==========\n", pass)

anyExecuted := false

for _, dn := range nodes {
matches := SyncTest(dn.Gatekeeper, runtime)

if matches {
fmt.Printf("\n [%s] TRIGGERED - Gatekeeper matched\n", dn.Name)

// Execute the Design Node
flowout := dn.Process(runtime)

// Merge results into runtime state
runtime = MergeSignals(runtime, flowout)

PrintSignal("  Flowout", flowout)
```

```
anyExecuted = true
} else {
fmt.Printf("\n [%s] WAITING - Gatekeeper not matched\n", dn.Name)
}
}

PrintSignal("Runtime State after pass", runtime)

if !anyExecuted {
fmt.Printf("\n========== LOOP TERMINATED (no DN triggered) ==========\n")
break
}

// Safety limit for demo
if pass > 10 {
fmt.Println("\n[Safety limit reached]")
break
}
}

return runtime
}

// ============== DEMO SCENARIOS ==============

func main() {
fmt.Println("+-------------------------------------------------------------+")
fmt.Println("|     CPUX DESIGN NODE DEMO - PULSE PRESENCE/ABSENCE          |")
fmt.Println("+-------------------------------------------------------------+")

nodes := []DesignNode{DN_ValidatePayment, DN_CheckInventory, DN_ShipOrder}

// ============== SCENARIO 1: Complete Flow ==============
fmt.Println("\n\n>>> SCENARIO 1: Order received - full flow should execute")
fmt.Println("-------------------------------------------------------------")
```

```
runtime1 := Signal{
{"order received", true, Y},
{"payment validated", false, U},
{"inventory checked", false, U},
{"order shipped", false, U},
}

PrintSignal("Initial State", runtime1)
final1 := IntentionLoop(nodes, runtime1)
PrintSignal("FINAL STATE", final1)

// ============== SCENARIO 2: Missing Order ==============
fmt.Println("\n\n>>> SCENARIO 2: No order received - nothing should trigger")
fmt.Println("------------------------------------------------------------")

runtime2 := Signal{
// "order received" is MISSING
{"payment validated", false, U},
{"inventory checked", false, U},
{"order shipped", false, U},
}

PrintSignal("Initial State", runtime2)
final2 := IntentionLoop(nodes, runtime2)
PrintSignal("FINAL STATE", final2)

// ============== SCENARIO 3: Payment Already Done ==============
fmt.Println("\n\n>>> SCENARIO 3: Payment already validated - skip to
inventory")
fmt.Println("------------------------------------------------------------")

runtime3 := Signal{
{"order received", true, Y},
{"payment validated", true, Y},      // Already done!
```

```
{"transaction id", "txn_pre", Y},
{"inventory checked", false, U},
{"order shipped", false, U},
}

PrintSignal("Initial State", runtime3)
final3 := IntentionLoop(nodes, runtime3)
PrintSignal("FINAL STATE", final3)

// ============== SCENARIO 4: Wrong Trivalence ==============
fmt.Println("\n\n>>> SCENARIO 4: Payment validated = N (failed) - flow
blocked")
fmt.Println("------------------------------------------------------------")

runtime4 := Signal{
{"order received", true, Y},
{"payment validated", false, N},     // Failed, not Undecided!
{"inventory checked", false, U},
{"order shipped", false, U},
}

PrintSignal("Initial State", runtime4)
final4 := IntentionLoop(nodes, runtime4)
PrintSignal("FINAL STATE", final4)

fmt.Println("\n\n------------------------------------------------------------")
fmt.Println("KEY OBSERVATIONS:")
fmt.Println("  * Design Nodes only trigger when ALL Gatekeeper pulses match")
fmt.Println("  * Missing pulse = no match")
fmt.Println("  * Wrong trivalence (Y/N/U) = no match")
fmt.Println("  * Execution order emerges from data state, not code sequence")
fmt.Println("  * No if-then branching in the flow logic")
fmt.Println("------------------------------------------------------------")
}
```
