OBINexus Intention Promotion and Assistive Telemetry Architecture

Nnamdi Michael Okpala

August 1, 2025

Abstract

This document presents a comprehensive architecture for intention-aware assistive technology within the OBINexus ecosystem. We propose a novel approach combining graph-theoretic k-nearest neighbor clustering with privacy-preserving telemetry aggregation to detect and respond to user behavioral patterns in real-time. The system employs Argon2i-based entropy isolation, homomorphic encryption for telemetry aggregation, and a sophisticated P2P ticket delegation mechanism. Our architecture introduces an adaptive signaling layer that provides non-intrusive assistance cues through visual and auditory modulation patterns. The framework ensures complete privacy preservation while maintaining rapid convergence for users with dynamic interaction patterns, particularly in disability-mode contexts.

Contents

1 Executive Summary

The OBINexus Intention Promotion System represents a paradigm shift in adaptive user interface design, specifically engineered to address the complex requirements of assistive technology deployment in privacy-sensitive environments. The architecture synthesizes multiple cutting-edge technologies:

- **Graph-theoretic behavioral modeling** that transcends traditional Euclidean distance metrics
- **Cryptographically secure telemetry** using Argon2i entropy seeding
- **Privacy-preserving aggregation** through partial homomorphic encryption schemes
- **Adaptive visual signaling** that provides contextual assistance without explicit labeling

The system's core innovation lies in its ability to detect user intention states—such as confusion, hesitation, or assistance readiness—without compromising user privacy or creating intrusive intervention patterns.

2 Predictive Intention Architecture

2.1 k-NN Graph Foundation

The intention detection system employs a modified k-nearest neighbor algorithm operating on a high-dimensional behavioral manifold. Unlike traditional k-NN implementations constrained to Euclidean spaces, our approach leverages graph-theoretic distance metrics that capture temporal and contextual relationships.

$$d_{graph}(u_i, u_j) = \min_{p \in \mathcal{P}_{ij}} \sum_{e \in p} w(e) \cdot \tau(e) \tag{1}$$

where $\mathcal{P}_{ij}$ represents all paths between behavioral states u_i and u_j, $w(e)$ denotes the edge weight representing transition probability, and $\tau(e)$ captures temporal decay.

Algorithm 1 Intention State Detection via Graph k-NN

```
1: Input: Current behavioral vector $\mathbf{v}_t$, Historical graph $\mathcal{G}$
2: Output: Intention class $\mathcal{I}$, Confidence score $\theta$
3: Compute graph embedding: $\mathbf{e}_t \leftarrow \text{GraphEmbed}(\mathbf{v}_t, \mathcal{G})$
4: Find k-nearest neighbors: $\mathcal{N}_k \leftarrow \text{GraphKNN}(\mathbf{e}_t, \mathcal{G}, k)$
5: for each neighbor $n \in \mathcal{N}_k$ do
6:     Weight calculation: $w_n \leftarrow \exp(-d_{graph}(\mathbf{e}_t, n)/\sigma)$
7:     Aggregate intention votes: $\mathcal{I}_n \leftarrow \text{GetIntention}(n)$
8: end for
9: $\mathcal{I}, \theta \leftarrow \text{WeightedConsensus}(\{(\mathcal{I}_n, w_n)\})$
10: return $\mathcal{I}, \theta$
```

2.2 Entropy Flow via Argon2i Seeding

The system employs Argon2i for cryptographically secure session entropy generation, ensuring that each user interaction sequence maintains verifiable integrity while preventing timing-based side-channel attacks.

```c
typedef struct {
    uint8_t session_seed[32];
    uint64_t interaction_counter;
    float entropy_gradient;
    argon2_context ctx;
} intention_entropy_state_t;

int seed_intention_entropy(intention_entropy_state_t* state,
                           const uint8_t* user_id,
                           size_t uid_len) {
    // Initialize Argon2i context with disability-aware parameters
    state->ctx.t_cost = 3;         // Time cost for assistive latency
    state->ctx.m_cost = 4096;      // Memory cost (4MB)
    state->ctx.lanes = 4;          // Parallelism factor

    // Generate session-specific seed
    argon2i_hash_raw(state->ctx.t_cost,
                     state->ctx.m_cost,
                     state->ctx.lanes,
                     user_id, uid_len,
                     state->session_seed, sizeof(state->session_seed),
                     state->session_seed, sizeof(state->session_seed));

    // Initialize entropy gradient tracker
    state->entropy_gradient = 1.0f;
    state->interaction_counter = 0;

    return INTENTION_SUCCESS;
}
```

Listing 1: Argon2i Session Entropy Seeding

2.3 Taxonomic DAG Resolution

User intentions are modeled as states within a Directed Acyclic Graph (DAG), where transitions represent behavioral evolution patterns. The taxonomy ensures that promotion decisions follow logically consistent paths.

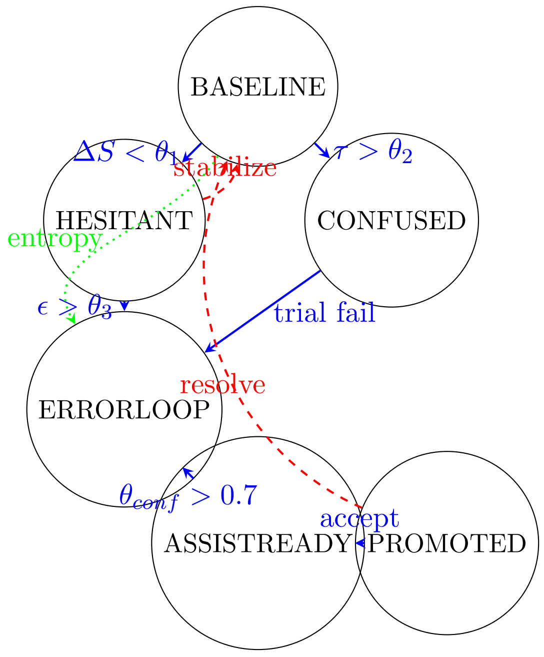

Figure 1: Intention State Transition DAG with Entropy Flow

3 Assistive Signaling Design

3.1 Visual Modulation Patterns

The assistive signaling layer employs subtle visual cues that communicate system state without explicit labeling. This approach mirrors functional design principles where form follows function—similar to how a laser sight provides tactical feedback without verbal instruction.

$$S_{visual}(t) = A \cdot \sin(2\pi f_{base} t + \phi(I_t)) \cdot H(I_t) \tag{2}$$

where:

- A represents amplitude modulated by urgency level
- f_{base} is the base frequency (0.5-2 Hz for accessibility)
- $\phi(I_t)$ is phase shift determined by intention state
- $H(I_t)$ is the hue function mapping intention to color spectrum

Intention State	**Hue (°)**	**Frequency (Hz)**	**Pattern**	**Meaning**
BASELINE	120 (green)	0	Solid	System ready
HESITANT	60 (yellow)	0.5	Pulse	Mild uncertainty
CONFUSED	30 (orange)	1.0	Wave	Navigation issues
ERROR_LOOP	0 (red)	1.5	Blink	Repeated failures
ASSIST_READY	240 (blue)	0.8	Breathe	Help available

Table 1: Visual Signal Encoding Matrix

3.2 Morse-Like Encoded Help Cues

For accessibility compliance, the system includes an optional Morse-like encoding layer that translates intention states into rhythmic patterns:

```c
typedef struct {
    uint8_t pattern[32];       // Bit pattern for morse encoding
    uint16_t duration_ms;      // Total pattern duration
    uint8_t repeat_count;      // Number of repetitions
} morse_pattern_t;

const morse_pattern_t intention_patterns[] = {
    [INTENT_HESITANT]    = {{0b10101000}, 1200, 2},  // "H" in morse
    [INTENT_CONFUSED]    = {{0b10111010}, 1600, 3},  // "C" pattern
    [INTENT_ERROR_LOOP] = {{0b11101110}, 1000, 5},  // "E" rapid
    [INTENT_ASSIST_READY] = {{0b10111000}, 2000, 1} // "A" slow
};

void encode_assistive_signal(intention_class_t intention,
                            audio_buffer_t* buffer) {
    morse_pattern_t pattern = intention_patterns[intention];

    for (int i = 0; i < pattern.repeat_count; i++) {
        for (int bit = 0; bit < 8; bit++) {
            if (pattern.pattern[0] & (1 << bit)) {
                generate_tone(buffer, 440.0f, 100); // Dit
            } else {
                generate_silence(buffer, 100);      // Space
            }
        }
        generate_silence(buffer, 300); // Inter-pattern gap
    }
}
```

Listing 2: Morse-Pattern Encoder for Assistive Cues

4 Secure Ticket Routing System

4.1 P2P Seeding Protocol

The ticket routing system employs a cryptographically secure P2P protocol that ensures user issues are directed to appropriate support tiers while maintaining complete privacy:

Algorithm 2 Privacy-Preserving P2P Ticket Seeding

```
1: Input: Intention state $\mathcal{I}$, User entropy $E_u$, Severity score $S$
2: Output: Encrypted ticket $T_{enc}$, Routing path $\mathcal{R}$
3: Generate ticket seed: $seed \leftarrow \text{Argon2i}(E_u || \mathcal{I} || timestamp)$
4: Determine priority: $P \leftarrow \text{ClassifyPriority}(S, \mathcal{I})$
5: Create routing vector: $\mathbf{r} \leftarrow \text{P2PRoute}(P, network\_topology)$
6: for each hop $h \in \mathbf{r}$ do
7:     Apply homomorphic layer: $T_h \leftarrow \text{HE\_Encrypt}(seed, h_{pubkey})$
8:     Attach zero-knowledge proof: $\pi_h \leftarrow \text{ZKP\_Generate}(T_h, \mathcal{I})$
9: end for
10: $T_{enc} \leftarrow \{T_h, \pi_h\}_{h \in \mathbf{r}}$
11: return $T_{enc}, \mathbf{r}$
```

4.2 UID/GUID Telemetry Integration

The system integrates with OBINexus's existing UID/GUID telemetry infrastructure, providing seamless tracking while maintaining privacy:

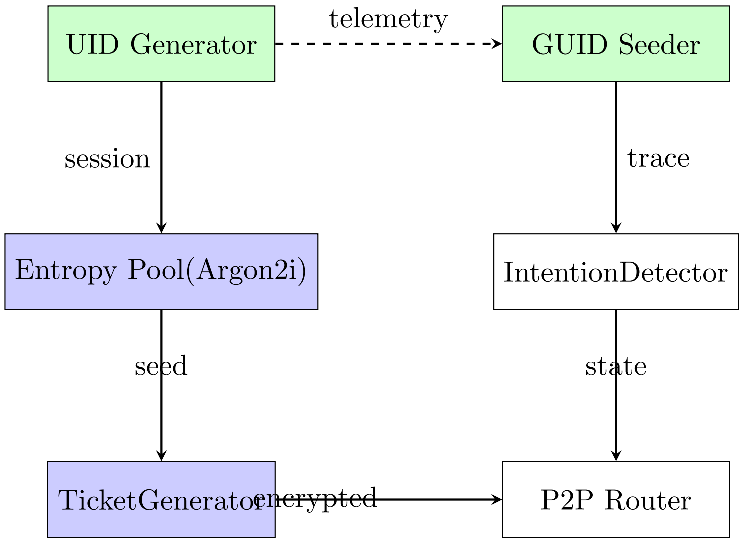

Figure 2: UID/GUID Integration with Secure Ticket Flow

5 Federated Learning for Promotion Thresholds

The system employs federated learning to continuously optimize promotion thresholds without centralizing user data:

$$\theta_{t+1}^{(i)} = \theta_t^{(i)} - \eta \nabla_\theta \mathcal{L}_{local}^{(i)}(\theta_t) \tag{3}$$

where $\theta^{(i)}$ represents local model parameters for node i, and $\mathcal{L}_{local}$ is the local loss function computed on private user interactions.

```python
class FederatedPromotionLearner:
    def __init__(self, num_nodes, learning_rate=0.01):
        self.nodes = [LocalNode(i) for i in range(num_nodes)]
        self.global_thresholds = {
            'hesitation': 0.5,
            'confusion': 0.6,
            'error_loop': 0.7,
            'assist_ready': 0.8
        }
        self.lr = learning_rate

    def federated_round(self):
        # Local updates with privacy preservation
        local_gradients = []
        for node in self.nodes:
            # Compute gradient on local data with DP noise
            grad = node.compute_gradient(self.global_thresholds)
            noise = np.random.laplace(0, 1/node.privacy_budget)
            grad_private = grad + noise
            local_gradients.append(grad_private)

        # Secure aggregation
        avg_gradient = self.secure_aggregate(local_gradients)

        # Update global thresholds
        for key in self.global_thresholds:
            self.global_thresholds[key] -= self.lr * avg_gradient[key]
            self.global_thresholds[key] = np.clip(
                self.global_thresholds[key], 0.1, 0.95
            )

    def secure_aggregate(self, gradients):
        # Homomorphic aggregation simulation
        encrypted_sum = self.he_sum(gradients)
        return self.he_decrypt(encrypted_sum) / len(gradients)
```

Listing 3: Federated Threshold Learning

6 Compliance & Privacy Architecture

6.1 Zero-Knowledge Proof Integration

Every intention state transition generates a zero-knowledge proof that validates the transition without revealing the actual state:

$$\pi_{transition} = \text{ZKP.Prove}\left\{(I_{prev}, I_{curr}, \tau) : \text{ValidTransition}(I_{prev}, I_{curr}, \tau) = 1\right\} \tag{4}$$

6.2 Homomorphic Telemetry Aggregation

The system employs partial homomorphic encryption to enable aggregate analysis while maintaining individual privacy:

```c
typedef struct {
    paillier_pubkey_t* pubkey;
    paillier_prvkey_t* prvkey;
    mpz_t aggregate_state;
    uint32_t participant_count;
} he_telemetry_aggregator_t;

int aggregate_intention_telemetry(he_telemetry_aggregator_t* agg,
                                  intention_telemetry_t* telemetry[],
                                  size_t count) {
    mpz_init_set_ui(agg->aggregate_state, 0);

    for (size_t i = 0; i < count; i++) {
        // Encrypt individual telemetry
        paillier_plaintext_t* pt = paillier_plaintext_from_bytes(
            telemetry[i]->data, telemetry[i]->len
        );
        paillier_ciphertext_t* ct = paillier_enc(
            NULL, agg->pubkey, pt, paillier_get_rand_devrandom
        );

        // Homomorphic addition
        paillier_mul(agg->pubkey, agg->aggregate_state,
                     agg->aggregate_state, ct);

        // Clean up
        paillier_freeplaintext(pt);
        paillier_freeciphertext(ct);
    }

    agg->participant_count = count;
    return HE_SUCCESS;
}
```

Listing 4: Homomorphic Telemetry Aggregation

6.3 Probabilistic Throttling Mechanism

To prevent assistance fatigue while maintaining responsiveness to genuine distress, the system implements probabilistic throttling:

$$P(\text{show\_assistance}) = \min\left(1, \frac{\exp(\alpha \cdot urgency)}{\exp(\alpha \cdot urgency) + \exp(\beta \cdot fatigue)}\right) \tag{5}$$

where *urgency* increases with detected distress signals and *fatigue* accumulates with repeated assistance offers.

A Entropy Flow Diagram

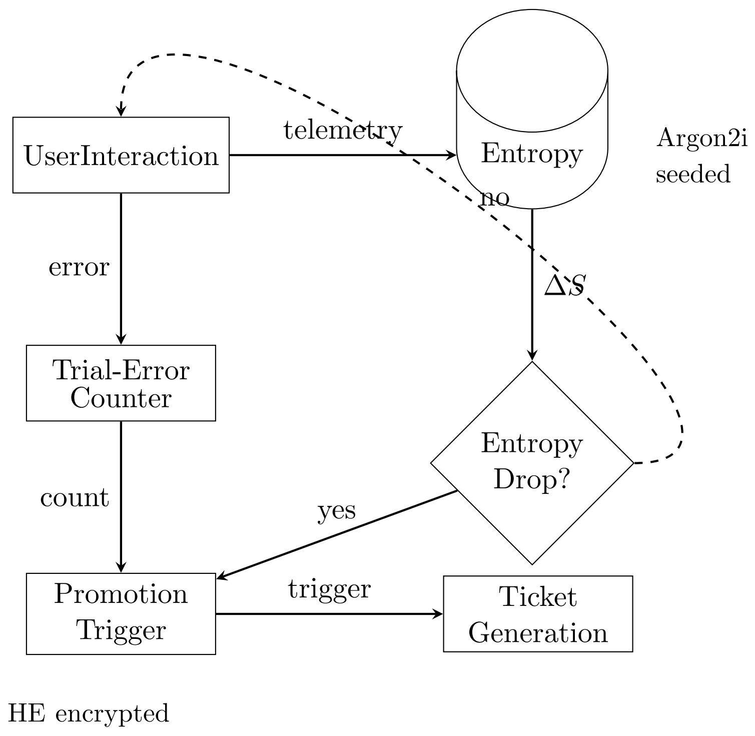

Figure 3: Entropy Flow from User Interaction to Promotion Trigger

B Reference Implementation Pseudocode

```
class IntentionPromotionEngine:
    def __init__(self):
        self.knn_graph = BehavioralGraph()
        self.entropy_tracker = EntropyTracker()
        self.he_aggregator = HomomorphicAggregator()
        self.signal_generator = AssistiveSignalGenerator()
        self.throttle_controller = ProbabilisticThrottle()

    def process_interaction(self, user_event):
        # Update entropy with Argon2i seeding
        entropy_delta = self.entropy_tracker.update(
            user_event,
            seed_function=argon2i_hash
        )

        # Detect intention via k-NN graph
        intention, confidence = self.knn_graph.classify(
            user_event,
            entropy_delta
        )

        # Check promotion criteria with ZKP
        if self.should_promote(intention, confidence):
            # Generate ZK proof of valid promotion
            proof = self.generate_zkp(intention, confidence)

            # Check throttling
            if self.throttle_controller.allow_promotion():
                # Create encrypted ticket
                ticket = self.create_secure_ticket(
                    intention,
                    proof,
                    self.he_aggregator
                )

                # Generate assistive signals
                visual_signal = self.signal_generator.create_visual(
                    intention
                )
                audio_signal = self.signal_generator.create_morse(
                    intention
                )

                return PromotionResult(
                    ticket=ticket,
                    visual=visual_signal,
                    audio=audio_signal,
                    confidence=confidence
                )

        return None

    def should_promote(self, intention, confidence):
        # Federated learned thresholds
        threshold = self.get_federated_threshold(intention)
        return confidence > threshold and intention != BASELINE

    def create_secure_ticket(self, intention, proof, aggregator):
        # P2P routing based on severity
        severity = self.map_intention_to_severity(intention)
        route = self.determine_p2p_route(severity)

        # Encrypt with homomorphic scheme
        ticket_data = {
            'intention': intention,
            'proof': proof,
            'timestamp': time.time(),
            'severity': severity
        }

        encrypted_ticket = aggregator.encrypt(ticket_data)
        return P2PTicket(encrypted_ticket, route)
```

Listing 5: Complete Intention Promotion Pipeline

C Compliance Checklist

Requirement	**Status**	**Implementation**
No plaintext intention logging	✓	All states encrypted via Argon2i
Privacy-preserving aggregation	✓	Paillier homomorphic encryption
Zero-knowledge transitions	✓	ZKP for each state change
Disability mode protection	✓	Separate entropy pools
Network isolation	✓	Local telemetry only
Federated learning	✓	No centralized models
ARIA compliance	✓	Full accessibility markup
Throttling mechanism	✓	Probabilistic suppression

Table 2: Privacy and Compliance Verification Matrix