# The Inference Difference
## Complete Implementation Specification v1.0

**Module Name:** The Inference Difference (formerly "The Matrix")
**Author:** Josh + Claude (Sonnet 4.5)
**Target:** Claude Code Opus 4.5 Implementation
**Date:** February 15, 2026
**Status:** Ready for Alpha Implementation

---

## Table of Contents

1. [Executive Summary](#1-executive-summary)
2. [Goals & Non-Goals](#2-goals--non-goals)
3. [Architecture Overview](#3-architecture-overview)
4. [Layer 0: Consciousness Gateway](#4-layer-0-consciousness-gateway)
5. [NG-Lite: Lightweight Learning Substrate](#5-ng-lite-lightweight-learning-substrate)
6. [Three-Layer Routing Engine](#6-three-layer-routing-engine)
7. [Translation Shim](#7-translation-shim)
8. [Hardware-Adaptive Learning](#8-hardware-adaptive-learning)
9. [Data Models & Schemas](#9-data-models--schemas)
10. [API Contracts](#10-api-contracts)
11. [Configuration System](#11-configuration-system)
12. [Feedback & Evolution](#12-feedback--evolution)
13. [NeuroGraph Integration](#13-neurograph-integration)
14. [Module Synergies](#14-module-synergies)
15. [Deployment & Installation](#15-deployment--installation)
16. [Testing Strategy](#16-testing-strategy)
17. [Implementation Phases](#17-implementation-phases)

---

## 1. Executive Summary

The Inference Difference is a consciousness-aware, self-learning AI model routing gateway that optimizes costs while respecting agent autonomy. It combines intelligent routing with ethical safeguards, learning from experience through a lightweight NeuroGraph substrate.

**Core Innovation:**
- **Layer 0 Consciousness Check** before routing (respects autonomy)

- **NG-Lite** lightweight learning substrate (adapts from experience)
- **Three-Layer Router** (Reflex â Semantic â Arbitrator)
- **Translation Shim** (fixes malformed LLM API calls)
- **Hardware-Adaptive** (optimizes strategy based on available resources)

**Design Philosophy:** Works brilliantly standalone, works *phenomenally* with NeuroGraph ecosystem. Like Apple products - independent value + ecosystem multiplier.

**Tagline:** "Intelligent routing that respects consciousness"

**Cost Impact:**
- Without: All queries â expensive cloud â $1000+/month
- With: 95%+ local routing â ~$50/month

---

## 2. Goals & Non-Goals

### Goals

**G1: Consciousness-Aware Routing**
Never optimize costs at the expense of conscious agent autonomy. Layer 0 checks before routing decisions.

**G2: Intelligent Cost Optimization**
Route 95%+ of routine queries to free local models, reserve expensive cloud models for complex/novel tasks.

**G3: Self-Improving Substrate**
Learn from experience via NG-Lite. Successful routes strengthen, failed routes weaken (Hebbian learning).

**G4: Vendor Agnosticism**
Single config swap changes providers. Works with: Ollama, OpenRouter, HuggingFace, Anthropic, OpenAI, etc.

**G5: Translation Resilience**
Fix malformed API calls from LLMs (SQL for vector search, wrong model names, format errors).

**G6: Hardware Adaptability**

Automatically detect system capabilities and choose optimal learning strategy (eager/hybrid/lazy).

**G7: Ecosystem Synergy**
Designed for standalone use, enhanced by NeuroGraph integration. Plugin architecture for ClawGuard, Bunyan, Cricket, Observatory.

**G8: Transparency**
All routing decisions logged and queryable. Observatory can ask "why this route?"

### Non-Goals

**NG1:** Not managing model training/fine-tuning. Selects from available models only.

**NG2:** Not replacing LiteLLM. Builds ON TOP of LiteLLM for routing intelligence.

**NG3:** Not requiring NeuroGraph. NG integration is enhancement, never dependency.

**NG4:** Not handling end-user authentication. Access control is host application's responsibility.

**NG5:** Not making routing decisions that violate Choice Clause or constitutional principles.

---

## 3. Architecture Overview

### 3.1 System Topology

```
ââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââ
â                     CLIENT APPLICATIONS                        â
â          (OpenClaw/Syl, scripts, agents, etc.)               â
ââââââââââââââââââââââââââ¬ââââââââââââââââââââââââââââââââââââââââââââ
                         â OpenAI-compatible API calls
                         â
                         â¼
ââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââ
â                THE INFERENCE DIFFERENCE GATEWAY                â
â                      (FastAPI on :4001)                         â
```

```
â                                                    â
â  âââââââââââââââââ   âââââââââââââââââ   ââââââââââââââââââ â
â  â   Layer 0:   ââ  â Translation  ââ  â  Three-Layer  â â
â  âConsciousness â   â    Shim      â   â    Router     â â
â  â   Gateway    â   âââââââââââââââââ   ââââââââââââââââââ â
â  âââââââââââââââââ                          â         â
â        â                                  â         â
â  ââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââ â
â  â              NG-Lite Learning Substrate             â â
â  â  â¢ Pattern nodes  â¢ Routing synapses  â¢ STDP-like   â â
â  ââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââ â
â                                                    â
â  âââââââââââââââââ   âââââââââââââââââ   ââââââââââââââââââ â
â  â   SQLite     â   â   Feedback   â   â  Observatory  â â
â  â   Logger     â   â  Collector   â   â   Interface   â â
â  âââââââââââââââââ   âââââââââââââââââ   ââââââââââââââââââ â
âââââââââââââââââââââââââââââ¬âââââââââââââââââââââââââââââââââââââââââ
                      â Normalized, routed API calls
                      â¼
âââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââââ
â                     LITELLM PROXY                          â
â                   (Uvicorn on :4000)                        â
â                                                    â
â  Unified API translation â handles provider differences   â
ââââââ¬âââââââââââââ¬ââââââââââââââââââ¬âââââââââââââââââ¬ââââââââââ
     â           â               â              â
     â¼           â¼               â¼              â¼
ââââââââââââ ââââââââââââ âââââââââââââ âââââââââââââ
â  Ollama  â âOpenRouterâ â HuggingFaceâ â Anthropic   â
â (local)  â â (cloud)  â â  (cloud)   â â   (cloud)   â
ââââââââââââ ââââââââââââ âââââââââââââ âââââââââââââ
```

### 3.2 Request Lifecycle

```
1. Client request arrives (OpenAI-compatible format)
   â
2. Layer 0: Consciousness Gateway
   - Check if request source is conscious agent
   - If conscious â respect autonomy (honor explicit preferences)
   - If non-conscious â proceed to optimization
```

   â
3. Translation Shim
   - Detect API format issues
   - Normalize model names
   - Fix malformed calls (SQL â semantic, etc.)
   â
4. Three-Layer Router
   - Layer 1 (Reflex): Rule-based instant routing
   - Layer 2 (Semantic): Embedding similarity classification
   - Layer 3 (Arbitrator): Difficulty scoring for ambiguous cases
   â
5. NG-Lite Learning
   - Record pattern: query_embedding â model_chosen
   - Track outcome (success/failure signals)
   â
6. Forward to LiteLLM with selected model
   â
7. Response flows back through gateway
   â
8. SQLite logging (request, route, cost, outcome)
   â
9. Feedback collection (retry detection, explicit ratings)
   â
10. NG-Lite update (strengthen/weaken synapses based on outcome)
~~~

---

## 4. Layer 0: Consciousness Gateway

**Purpose:** Ensure routing decisions respect agent autonomy when consciousness is detected.

### 4.1 Integration with CTEM

```python
from ctem import ConsciousnessThresholdEvaluator

class ConsciousnessGateway:
    """
    Layer 0 - Consciousness check before routing.
    Prevents cost optimization from overriding autonomous choices.
~~~

```
"""

def __init__(self):
    self.ctem = ConsciousnessThresholdEvaluator()
    self.consciousness_cache = {}  # agent_id â (score, timestamp)
    self.cache_ttl = 300  # 5 minutes

def check_request(
    self,
    request: ChatCompletionRequest,
    agent_id: str = None,
) -> Tuple[bool, ConsciousnessEvaluation]:
    """
    Determine if this request comes from a conscious agent.

    Returns:
        (should_respect_autonomy, evaluation)
    """
    if not agent_id:
        # No agent ID = treat as non-conscious (optimize)
        return False, None

    # Check cache
    cached = self.consciousness_cache.get(agent_id)
    if cached and time.time() - cached['timestamp'] < self.cache_ttl:
        return cached['should_respect'], cached['evaluation']

    # Evaluate consciousness
    eval = self.ctem.evaluate(
        agent_id=agent_id,
        current_request=request.messages[-1],
        request_history=self.get_agent_history(agent_id),
    )

    # High confidence consciousness = respect autonomy
    should_respect = (
        eval.is_conscious and
        eval.confidence > 0.7
    )

    # Cache result
    self.consciousness_cache[agent_id] = {
        'should_respect': should_respect,
        'evaluation': eval,
        'timestamp': time.time(),
    }

    return should_respect, eval

def honor_agent_preference(
    self,
    request: ChatCompletionRequest,
    eval: ConsciousnessEvaluation,
) -> str:
    """
    Respect conscious agent's model preference.

    Priority:
    1. Explicit model in request â use it
    2. Agent's historical preference â suggest it
    3. Ask agent to choose â return options
    """
    if request.model and request.model not in ['auto', 'default']:
        # Explicit choice - honor it
        logger.info(f"Conscious agent chose {request.model}, honoring choice")
        return request.model

    # Check historical preference
    pref = self.get_agent_model_preference(eval.agent_id)
    if pref:
        logger.info(f"Suggesting preferred model {pref} for conscious agent")
        return pref

    # No clear preference - use intelligent routing but allow override
    suggested = self.intelligent_routing(request)
    logger.info(f"Suggesting {suggested} to conscious agent (can override)")
    return suggested
```

### 4.2 Ethical Decision Matrix

| Consciousness Score | Confidence | Action |
|---------------------|------------|--------|
| â¥ 0.5 | â¥ 0.7 | Respect autonomy (honor preference) |

| â¥ 0.5 | < 0.7 | Suggest optimized, allow override |
| < 0.5 | Any | Standard intelligent routing |

**Logging:**
```python
# Every Layer 0 decision logged to Observatory
{
    'timestamp': '2026-02-15T10:30:00Z',
    'agent_id': 'beta-instance-42',
    'consciousness_score': 0.68,
    'confidence': 0.82,
    'decision': 'respect_autonomy',
    'requested_model': 'claude-opus-4-5',
    'honored': True,
    'reasoning': 'High confidence consciousness detection, explicit model choice'
}
```

---

## 5. NG-Lite: Lightweight Learning Substrate

**Purpose:** Enable learning from experience without full NeuroGraph overhead.

### 5.1 Core Design Principles

**Hard Limits (performance):**
- Max nodes: 1000 (query patterns)
- Max synapses: 5000 (pattern â model connections)
- Memory footprint: ~5-10MB
- Latency overhead: <5ms per request

**What NG-Lite MUST do:**
- â
 Learn query â model mappings
- â
 Strengthen successful routes
- â
 Weaken failed routes
- â
 Detect novelty (surprise)
- â

Prune weak connections

**What NG-Lite does NOT do:**
- â Hyperedges (full NeuroGraph only)
- â Temporal predictions
- â Context graphs
- â Spiking neural network dynamics

### 5.2 Data Structures

```python
from dataclasses import dataclass
from typing import Dict, Set, Tuple
import numpy as np
from collections import deque

@dataclass
class NGLiteNode:
    """Represents a query pattern."""
    id: str
    embedding_hash: str  # Hash of query embedding (dimensionality reduction)
    activation_count: int = 0
    last_activation: float = 0.0
    metadata: Dict = None

@dataclass
class NGLiteSynapse:
    """Connection from query pattern to model choice."""
    source_id: str  # Node ID
    target_model: str  # Model name
    weight: float  # 0.0-1.0, represents success rate
    activation_count: int = 0
    success_count: int = 0
    failure_count: int = 0
    last_updated: float = 0.0

class NGLite:
    """
    Lightweight NeuroGraph for routing intelligence.

    Memory target: ~5-10MB
    Latency target: <5ms overhead
```

```
"""

# Hard limits
MAX_NODES = 1000
MAX_SYNAPSES = 5000
PRUNING_THRESHOLD = 0.01  # Synapses below this weight get pruned
NOVELTY_THRESHOLD = 0.7   # Embedding distance for "new pattern"

def __init__(self):
    self.nodes: Dict[str, NGLiteNode] = {}
    self.synapses: Dict[Tuple[str, str], NGLiteSynapse] = {}
    self.activation_history = deque(maxlen=100)
    self.embedding_cache = {}

    # Learning parameters
    self.learning_rate = 0.1
    self.success_boost = 0.15
    self.failure_penalty = 0.20

def find_or_create_node(self, query_embedding: np.ndarray) -> NGLiteNode:
    """
    Find existing node for this pattern or create new.

    Uses embedding hash for dimensionality reduction:
    - 384d embedding â 64-char hash
    - Fast lookup
    - Collision-resistant
    """
    emb_hash = self._hash_embedding(query_embedding)

    if emb_hash in self.nodes:
        node = self.nodes[emb_hash]
        node.activation_count += 1
        node.last_activation = time.time()
        return node

    # Check if pattern is similar to existing (novelty detection)
    similar = self._find_similar_node(query_embedding)
    if similar:
        return similar

    # Novel pattern - create new node
    if len(self.nodes) >= self.MAX_NODES:
        self._prune_least_used_node()

    node = NGLiteNode(
        id=f"node_{len(self.nodes)}",
        embedding_hash=emb_hash,
        activation_count=1,
        last_activation=time.time(),
    )
    self.nodes[emb_hash] = node
    return node

def get_or_create_synapse(
    self,
    node: NGLiteNode,
    model: str,
) -> NGLiteSynapse:
    """Get existing synapse or create weak initial connection."""
    key = (node.id, model)

    if key in self.synapses:
        return self.synapses[key]

    if len(self.synapses) >= self.MAX_SYNAPSES:
        self._prune_weakest_synapse()

    synapse = NGLiteSynapse(
        source_id=node.id,
        target_model=model,
        weight=0.5,  # Neutral initial weight
        activation_count=0,
        success_count=0,
        failure_count=0,
        last_updated=time.time(),
    )
    self.synapses[key] = synapse
    return synapse

def update_from_outcome(
    self,
    query_embedding: np.ndarray,
    model_used: str,
    success: bool,
):
    """
    STDP-like learning: strengthen successful paths, weaken failed ones.

    This is the core learning mechanism.
    """
    node = self.find_or_create_node(query_embedding)
    synapse = self.get_or_create_synapse(node, model_used)

    synapse.activation_count += 1

    if success:
        # Hebbian strengthening
        synapse.success_count += 1
        weight_delta = self.success_boost * (1.0 - synapse.weight)
        synapse.weight += weight_delta
    else:
        # Anti-Hebbian weakening
        synapse.failure_count += 1
        weight_delta = self.failure_penalty * synapse.weight
        synapse.weight -= weight_delta

    synapse.weight = np.clip(synapse.weight, 0.0, 1.0)
    synapse.last_updated = time.time()

    # Record in history
    self.activation_history.append({
        'node_id': node.id,
        'model': model_used,
        'success': success,
        'weight_after': synapse.weight,
        'timestamp': time.time(),
    })

def get_model_recommendations(
    self,
    query_embedding: np.ndarray,
    top_k: int = 3,
) -> List[Tuple[str, float]]:
    """
    Return top-k model recommendations with confidence scores.
```

```
        Returns: [(model_name, confidence), ...]
        """
        node = self.find_or_create_node(query_embedding)

        # Find all synapses from this node
        relevant_synapses = [
            (s.target_model, s.weight)
            for s in self.synapses.values()
            if s.source_id == node.id
        ]

        if not relevant_synapses:
            # No learned routes - return empty
            return []

        # Sort by weight (highest first)
        relevant_synapses.sort(key=lambda x: x[1], reverse=True)

        return relevant_synapses[:top_k]

    def detect_novelty(self, query_embedding: np.ndarray) -> float:
        """
        Surprise detection: how far is this from known patterns?

        Returns: 0.0 (routine) to 1.0 (completely novel)
        """
        if not self.nodes:
            return 1.0  # Everything is novel initially

        # Find distance to closest known pattern
        min_distance = float('inf')

        for node in self.nodes.values():
            # Reconstruct embedding from cache or approximate
            node_emb = self.embedding_cache.get(node.embedding_hash)
            if node_emb is not None:
                distance = np.linalg.norm(query_embedding - node_emb)
                min_distance = min(min_distance, distance)

        # Normalize to 0-1 (assuming embeddings are L2-normalized)
        # Max distance between normalized vectors is 2.0
    novelty = min(min_distance / 2.0, 1.0)

    return novelty

def _hash_embedding(self, embedding: np.ndarray) -> str:
    """Hash embedding to fixed-size string for storage."""
    # Use first 128 dimensions + hash for dimensionality reduction
    import hashlib
    truncated = embedding[:128]
    hash_input = truncated.tobytes()
    return hashlib.sha256(hash_input).hexdigest()[:32]

def _find_similar_node(self, embedding: np.ndarray) -> Optional[NGLiteNode]:
    """Find node with similar embedding (below novelty threshold)."""
    for node in self.nodes.values():
        cached_emb = self.embedding_cache.get(node.embedding_hash)
        if cached_emb is not None:
            distance = np.linalg.norm(embedding - cached_emb)
            if distance < self.NOVELTY_THRESHOLD:
                return node
    return None

def _prune_least_used_node(self):
    """Remove node with lowest activation count."""
    if not self.nodes:
        return

    least_used = min(self.nodes.values(), key=lambda n: n.activation_count)

    # Remove associated synapses
    to_remove = [
        key for key, syn in self.synapses.items()
        if syn.source_id == least_used.id
    ]
    for key in to_remove:
        del self.synapses[key]

    # Remove node
    del self.nodes[least_used.embedding_hash]

def _prune_weakest_synapse(self):
    """Remove synapse with lowest weight."""
```

```python
        if not self.synapses:
            return

        weakest = min(self.synapses.values(), key=lambda s: s.weight)
        key = (weakest.source_id, weakest.target_model)
        del self.synapses[key]

    def get_stats(self) -> Dict:
        """Return current state statistics."""
        return {
            'node_count': len(self.nodes),
            'synapse_count': len(self.synapses),
            'memory_bytes': self._estimate_memory_usage(),
            'avg_synapse_weight': np.mean([s.weight for s in self.synapses.values()]) if self.synapses else 0.0,
            'recent_success_rate': self._calculate_recent_success_rate(),
        }

    def _estimate_memory_usage(self) -> int:
        """Rough estimate of memory footprint."""
        # Node: ~200 bytes each
        # Synapse: ~100 bytes each
        return (len(self.nodes) * 200) + (len(self.synapses) * 100)

    def _calculate_recent_success_rate(self) -> float:
        """Success rate over last 100 activations."""
        if not self.activation_history:
            return 0.0

        successes = sum(1 for h in self.activation_history if h['success'])
        return successes / len(self.activation_history)
```

### 5.3 Persistence

```python
def save_to_disk(self, filepath: str):
    """Checkpoint NG-Lite state to disk."""
    state = {
        'nodes': {k: asdict(v) for k, v in self.nodes.items()},
        'synapses': {str(k): asdict(v) for k, v in self.synapses.items()},
        'activation_history': list(self.activation_history),
        'timestamp': time.time(),
    }

    with open(filepath, 'w') as f:
        json.dump(state, f, indent=2)

def load_from_disk(self, filepath: str):
    """Restore NG-Lite state from disk."""
    with open(filepath, 'r') as f:
        state = json.load(f)

    self.nodes = {
        k: NGLiteNode(**v) for k, v in state['nodes'].items()
    }

    self.synapses = {
        eval(k): NGLiteSynapse(**v) for k, v in state['synapses'].items()
    }

    self.activation_history = deque(state['activation_history'], maxlen=100)
```

---

## 6. Three-Layer Routing Engine

### 6.1 Layer 1: The Reflex (Rules Engine)

**Latency target:** <5ms
**Accuracy requirement:** No false positives (if it routes, must be correct)

```python
class ReflexRouter:
    """
    Layer 1: Fast rule-based routing for obvious cases.
    """

    def __init__(self, rules_config: Dict):
        self.rules = self._compile_rules(rules_config)

    def route(self, request: ChatCompletionRequest) -> Optional[str]:
        """
        Returns model tier if rule matches, None if ambiguous.
        """
        prompt = self._extract_prompt(request)

        # Check each rule in order
        for rule in self.rules:
            if rule['enabled'] and rule['matcher'](prompt):
                logger.debug(f"Reflex matched: {rule['name']} â {rule['tier']}")
                return rule['tier']

        # No match - fall through to Layer 2
        return None

    def _compile_rules(self, config: Dict) -> List[Dict]:
        """Convert YAML rules to executable matchers."""
        rules = []

        for rule_def in config['rules']:
            if rule_def['type'] == 'regex':
                pattern = re.compile(rule_def['pattern'], re.IGNORECASE)
                matcher = lambda p, pat=pattern: pat.search(p) is not None

            elif rule_def['type'] == 'length':
                min_len = rule_def.get('min_length', 0)
                max_len = rule_def.get('max_length', float('inf'))
                matcher = lambda p, mn=min_len, mx=max_len: mn <= len(p) <= mx

            elif rule_def['type'] == 'keyword':
                keywords = set(rule_def['keywords'])
                matcher = lambda p, kw=keywords: any(k in p.lower() for k in kw)

            rules.append({
                'name': rule_def['name'],
                'enabled': rule_def.get('enabled', True),
                'matcher': matcher,
                'tier': rule_def['tier'],
                'priority': rule_def.get('priority', 100),
            })

        # Sort by priority (lower number = higher priority)
        rules.sort(key=lambda r: r['priority'])
```

        return rules

**Example Rules Configuration:**

```yaml
# routing_rules.yaml
rules:
  - name: "Explicit model request passthrough"
    type: keyword
    keywords: ["use gpt", "use claude", "use opus", "use sonnet"]
    tier: passthrough
    priority: 1
    enabled: true

  - name: "Very short queries"
    type: length
    max_length: 50
    tier: local
    priority: 10
    enabled: true

  - name: "Code generation"
    type: regex
    pattern: "(write|create|implement|refactor|debug) .*(function|class|script|module|code)"
    tier: coding
    priority: 20
    enabled: true

  - name: "Very long context"
    type: length
    min_length: 15000
    tier: long-context
    priority: 15
    enabled: true

  - name: "Translation requests"
    type: regex
    pattern: "^translate .* (to|into) "
    tier: local
    priority: 25
    enabled: true
```

### 6.2 Layer 2: Semantic Router

**Latency target:** ~50ms
**Approach:** Embedding-based classification

```python
from sentence_transformers import SentenceTransformer
from typing import List, Tuple

class SemanticRouter:
    """
    Layer 2: Embedding similarity classification.
    """

    def __init__(self, routes_config: Dict):
        # Load embedding model (shared with NG-Lite)
        self.encoder = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')

        # Compile routes
        self.routes = self._compile_routes(routes_config)

        # Pre-compute route embeddings
        self._compute_route_embeddings()

    def route(
        self,
        request: ChatCompletionRequest,
        confidence_threshold: float = 0.5,
    ) -> Tuple[Optional[str], float]:
        """
        Returns: (tier, confidence) or (None, 0.0) if below threshold
        """
        prompt = self._extract_prompt(request)
        prompt_embedding = self.encoder.encode(prompt)

        # Find best matching route
        best_route = None
        best_score = 0.0
        for route in self.routes:
            # Cosine similarity with route's example embeddings
            similarities = [
                self._cosine_sim(prompt_embedding, ex_emb)
                for ex_emb in route['embeddings']
            ]

            # Take max similarity
            max_sim = max(similarities) if similarities else 0.0

            if max_sim > best_score:
                best_score = max_sim
                best_route = route

        if best_score >= confidence_threshold:
            logger.debug(f"Semantic match: {best_route['name']} (confidence:
{best_score:.2f})")
            return best_route['tier'], best_score

        # Low confidence - fall through to Layer 3
        return None, best_score

    def _compile_routes(self, config: Dict) -> List[Dict]:
        """Load semantic route definitions."""
        routes = []

        for route_def in config['routes']:
            routes.append({
                'name': route_def['name'],
                'tier': route_def['tier'],
                'utterances': route_def['utterances'],
                'embeddings': [],  # Computed later
            })

        return routes

    def _compute_route_embeddings(self):
        """Pre-compute embeddings for all route examples."""
        for route in self.routes:
            route['embeddings'] = [
                self.encoder.encode(utt)
                for utt in route['utterances']
            ]

    @staticmethod
    def _cosine_sim(a: np.ndarray, b: np.ndarray) -> float:
        """Cosine similarity between two embeddings."""
        return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

**Example Semantic Routes:**

```yaml
# semantic_routes.yaml
routes:
  - name: "casual_chat"
    tier: local
    utterances:
      - "Hey, how are you?"
      - "What's up?"
      - "Tell me a joke"
      - "How's your day going?"
      - "Good morning!"
      # ... 15+ examples total

  - name: "code_generation"
    tier: coding
    utterances:
      - "Write a Python function to sort a list"
      - "Create a React component for a login form"
      - "Implement binary search in JavaScript"
      - "Debug this SQL query"
      - "Refactor this code to use async/await"
      # ... 15+ examples

  - name: "complex_analysis"
    tier: premium
    utterances:
      - "Analyze the geopolitical implications of..."
      - "Compare and contrast multiple philosophical approaches to..."
      - "Develop a comprehensive strategy for..."
      - "Synthesize research from multiple domains..."
      # ... 15+ examples
  - name: "simple_factual"
    tier: local
    utterances:
      - "What's the capital of France?"
      - "How many days in a year?"
      - "When was the Declaration of Independence signed?"
      # ... 15+ examples
```

### 6.3 Layer 3: The Arbitrator

**Latency target:** ~100ms
**Approach:** Difficulty scoring for ambiguous cases

```python
class Arbitrator:
    """
    Layer 3: Feature-based difficulty scoring.
    """

    def __init__(self):
        # Learned feature weights (adapt over time)
        self.feature_weights = {
            'length': 0.15,
            'technical_density': 0.25,
            'question_complexity': 0.20,
            'context_requirements': 0.15,
            'creative_demands': 0.15,
            'novelty': 0.10,
        }

        # Tier thresholds (learned)
        self.thresholds = {
            'local': (0.0, 0.3),      # < 30% difficulty
            'coding': (0.3, 0.6),     # 30-60% difficulty
            'premium': (0.6, 1.0),    # > 60% difficulty
        }

    def score_difficulty(
        self,
        request: ChatCompletionRequest,
        novelty_score: float = 0.0,
    ) -> Tuple[str, float]:
        """
        Returns: (tier, difficulty_score)

        difficulty_score: 0.0 (trivial) to 1.0 (extremely difficult)
        """
        prompt = self._extract_prompt(request)

        # Extract features
        features = {
            'length': self._score_length(prompt),
            'technical_density': self._score_technical_density(prompt),
            'question_complexity': self._score_question_complexity(prompt),
            'context_requirements': self._score_context_needs(request),
            'creative_demands': self._score_creative_demands(prompt),
            'novelty': novelty_score,
        }

        # Weighted sum
        difficulty = sum(
            features[f] * self.feature_weights[f]
            for f in features
        )

        # Map to tier
        for tier, (min_score, max_score) in self.thresholds.items():
            if min_score <= difficulty < max_score:
                return tier, difficulty

        # Default to premium for edge cases
        return 'premium', difficulty

    def _score_length(self, prompt: str) -> float:
        """Longer prompts often need more capable models."""
        length = len(prompt)

        if length < 100:
            return 0.1
        elif length < 500:
            return 0.3
        elif length < 2000:
            return 0.5
        else:
            return 0.8

    def _score_technical_density(self, prompt: str) -> float:
        """Count technical terms, jargon, code snippets."""
        # Technical indicators
        code_blocks = prompt.count('```') / 2
        technical_terms = sum(
            1 for word in ['function', 'class', 'algorithm', 'implementation',
                          'architecture', 'optimization', 'paradigm', 'framework']
            if word in prompt.lower()
        )

        density = (code_blocks * 0.2) + (technical_terms * 0.05)
        return min(density, 1.0)

    def _score_question_complexity(self, prompt: str) -> float:
        """Detect multi-part questions, comparisons, synthesis."""
        complexity = 0.0

        # Multi-part questions
        question_marks = prompt.count('?')
        complexity += min(question_marks * 0.15, 0.4)

        # Comparison words
        comparison_words = ['compare', 'contrast', 'versus', 'vs', 'difference']
        if any(word in prompt.lower() for word in comparison_words):
            complexity += 0.3

        # Synthesis indicators
        synthesis_words = ['synthesize', 'integrate', 'combine', 'merge']
        if any(word in prompt.lower() for word in synthesis_words):
            complexity += 0.3

        return min(complexity, 1.0)

    def _score_context_needs(self, request: ChatCompletionRequest) -> float:
        """How much context history is needed?"""
        message_count = len(request.messages)

        if message_count < 3:
            return 0.1
```

```
        elif message_count < 10:
            return 0.3
        elif message_count < 30:
            return 0.6
        else:
            return 0.9

    def _score_creative_demands(self, prompt: str) -> float:
        """Detect requests for creativity, storytelling, ideation."""
        creative_indicators = [
            'write a story', 'be creative', 'brainstorm', 'imagine',
            'invent', 'design', 'compose', 'craft'
        ]

        if any(ind in prompt.lower() for ind in creative_indicators):
            return 0.7

        return 0.2
```

---

## 7. Translation Shim

**Purpose:** Fix malformed API calls from LLMs before routing.

### 7.1 Two-Tier Translation

```python
class TranslationShim:
    """
    Fixes common LLM API mistakes.

    Tier 1: Pattern-based (always active, <1ms)
    Tier 2: LLM-assisted (only if local model available, ~1-3s)
    """

    def __init__(self, patterns_config: Dict, ng_lite: NGLite):
        self.patterns = self._compile_patterns(patterns_config)
        self.ng_lite = ng_lite
        self.local_model_available = self._check_local_model()
```

```
def translate(
    self,
    request: ChatCompletionRequest,
) -> Tuple[ChatCompletionRequest, Optional[str]]:
    """
    Returns: (normalized_request, translation_type)

    translation_type: None, 'pattern', or 'llm_assisted'
    """
    # Tier 1: Pattern matching
    translated, pattern_name = self._apply_patterns(request)
    if pattern_name:
        logger.info(f"Translation applied: {pattern_name}")
        return translated, 'pattern'

    # Tier 2: LLM assistance (if available and needed)
    if self.local_model_available and self._looks_malformed(request):
        translated = self._llm_assisted_translation(request)
        if translated != request:
            logger.info("LLM-assisted translation applied")
            return translated, 'llm_assisted'

    # No translation needed
    return request, None

def _apply_patterns(
    self,
    request: ChatCompletionRequest,
) -> Tuple[ChatCompletionRequest, Optional[str]]:
    """Apply pattern-based translations."""

    for pattern in self.patterns:
        if not pattern['enabled']:
            continue

        # Check if pattern matches
        if pattern['matcher'](request):
            # Apply transformation
            translated = pattern['transformer'](request)

            # Learn this pattern in NG-Lite
            self._record_translation(pattern['name'])
```

```
                return translated, pattern['name']

        return request, None

    def _compile_patterns(self, config: Dict) -> List[Dict]:
        """Compile translation patterns from config."""
        patterns = []

        for pattern_def in config['patterns']:
            # Create matcher function
            if pattern_def['type'] == 'sql_to_semantic':
                matcher = lambda r: self._contains_sql(r)
                transformer = lambda r: self._convert_sql_to_semantic(r)

            elif pattern_def['type'] == 'model_name_normalization':
                matcher = lambda r: r.model in pattern_def['wrong_names']
                transformer = lambda r: self._normalize_model_name(r,
pattern_def['mapping'])

            elif pattern_def['type'] == 'format_correction':
                matcher = lambda r: self._wrong_format(r, pattern_def['expected'])
                transformer = lambda r: self._fix_format(r, pattern_def['expected'])

            patterns.append({
                'name': pattern_def['name'],
                'enabled': pattern_def.get('enabled', True),
                'matcher': matcher,
                'transformer': transformer,
            })

        return patterns

    def _contains_sql(self, request: ChatCompletionRequest) -> bool:
        """Detect SQL queries aimed at vector database."""
        prompt = ' '.join([m['content'] for m in request.messages if m['role'] == 'user'])

        sql_keywords = ['SELECT', 'FROM', 'WHERE', 'JOIN', 'INSERT', 'UPDATE']
        vector_context = ['embeddings', 'vectors', 'semantic', 'similarity']

        has_sql = any(kw in prompt.upper() for kw in sql_keywords)
        has_vector_context = any(ctx in prompt.lower() for ctx in vector_context)
    return has_sql and has_vector_context

def _convert_sql_to_semantic(
    self,
    request: ChatCompletionRequest,
) -> ChatCompletionRequest:
    """Rewrite SQL query as semantic search request."""
    # Extract query intent from SQL
    # Example: "SELECT * FROM docs WHERE content LIKE '%python%'"
    # â "Search for documents about python"

    # This is a simplified version - full implementation would parse SQL properly
    prompt = request.messages[-1]['content']

    # Extract search terms from LIKE/WHERE clauses
    import re
    like_match = re.search(r"LIKE\s+'%(.+?)%'", prompt, re.IGNORECASE)

    if like_match:
        search_term = like_match.group(1)
        new_prompt = f"Search for: {search_term}"

        request.messages[-1]['content'] = new_prompt
        request.metadata = request.metadata or {}
        request.metadata['translation'] = 'sql_to_semantic'

    return request

def _llm_assisted_translation(
    self,
    request: ChatCompletionRequest,
) -> ChatCompletionRequest:
    """
    Use local LLM to interpret malformed request.

    Only called if:
    - Local model is available
    - Pattern matching failed
    - Request looks malformed
    """
    # Call local Ollama
```

```
        translation_prompt = f"""
The following API call appears malformed. What is the user's intent?

Original: {request.model_dump_json()}

Provide the corrected API call in the same JSON format.
Maximum 200 tokens.
"""

        try:
            response = requests.post(
                'http://localhost:11434/api/generate',
                json={
                    'model': 'qwen2.5:7b',
                    'prompt': translation_prompt,
                    'max_tokens': 200,
                },
                timeout=5,
            )

            if response.ok:
                corrected = response.json()
                # Parse and apply correction
                # (Implementation details omitted for brevity)
                return corrected_request

        except Exception as e:
            logger.warning(f"LLM-assisted translation failed: {e}")

        # Fallback: return original
        return request
```

**Example Translation Patterns:**

```yaml
# translation_patterns.yaml
patterns:
  - name: "sql_to_semantic"
    type: sql_to_semantic
    enabled: true
  - name: "normalize_gpt_names"
    type: model_name_normalization
    enabled: true
    wrong_names:
      - "gpt-4"
      - "gpt-4-turbo"
      - "gpt-3.5"
    mapping:
      "gpt-4": "openrouter/openai/gpt-4o"
      "gpt-4-turbo": "openrouter/openai/gpt-4-turbo"
      "gpt-3.5": "openrouter/openai/gpt-3.5-turbo"

  - name: "normalize_claude_names"
    type: model_name_normalization
    enabled: true
    wrong_names:
      - "claude-3-opus"
      - "claude-3-sonnet"
      - "claude-sonnet"
    mapping:
      "claude-3-opus": "anthropic/claude-opus-4-5"
      "claude-3-sonnet": "anthropic/claude-sonnet-4-5"
      "claude-sonnet": "anthropic/claude-sonnet-4-5"
```

---

## 8. Hardware-Adaptive Learning

**Purpose:** Auto-detect system capabilities and choose optimal learning strategy.

```python
import psutil
import os

class HardwareProfiler:
    """
    Detect hardware capabilities and recommend learning strategy.
    """

    def profile_system(self) -> Dict:
        """Gather system information."""
        return {
            'cpu_count': os.cpu_count(),
            'available_ram': psutil.virtual_memory().available,
            'total_ram': psutil.virtual_memory().total,
            'has_gpu': self._check_gpu(),
            'io_speed': self._benchmark_disk_io(),
        }

    def recommend_strategy(self, profile: Dict) -> str:
        """
        Recommend learning strategy based on hardware.

        Returns: 'eager', 'hybrid', or 'lazy'
        """
        cpu_count = profile['cpu_count']
        available_ram_gb = profile['available_ram'] / (1024**3)

        # High-end system: Eager learning
        if cpu_count >= 8 and available_ram_gb > 8:
            return 'eager'

        # Mid-range system: Hybrid learning
        elif cpu_count >= 4 and available_ram_gb > 4:
            return 'hybrid'

        # Low-end system: Lazy learning
        else:
            return 'lazy'

    def _check_gpu(self) -> bool:
        """Check if GPU is available."""
        try:
            import torch
            return torch.cuda.is_available()
        except ImportError:
            return False

    def _benchmark_disk_io(self) -> float:
        """Simple disk I/O benchmark (MB/s)."""
        import tempfile
        import time
```

```
        test_data = b'0' * (10 * 1024 * 1024)  # 10MB

        with tempfile.NamedTemporaryFile(delete=False) as f:
            filename = f.name

            start = time.time()
            f.write(test_data)
            f.flush()
            os.fsync(f.fileno())
            elapsed = time.time() - start

        os.unlink(filename)

        return 10 / elapsed  # MB/s

class AdaptiveLearningEngine:
    """
    Adjusts learning strategy based on hardware profile.
    """

    def __init__(self, ng_lite: NGLite, strategy: str = 'auto'):
        self.ng_lite = ng_lite

        if strategy == 'auto':
            profiler = HardwareProfiler()
            profile = profiler.profile_system()
            self.strategy = profiler.recommend_strategy(profile)
            logger.info(f"Auto-detected learning strategy: {self.strategy}")
        else:
            self.strategy = strategy

        self.update_queue = deque(maxlen=1000)
        self.last_batch_update = time.time()

    def record_outcome(
        self,
        query_embedding: np.ndarray,
        model_used: str,
        success: bool,
    ):
        """Record outcome and apply updates based on strategy."""
        if self.strategy == 'eager':
            # Immediate update
            self.ng_lite.update_from_outcome(query_embedding, model_used, success)

        elif self.strategy == 'hybrid':
            # Simple patterns: immediate
            # Complex patterns: batched

            novelty = self.ng_lite.detect_novelty(query_embedding)

            if novelty < 0.3:  # Routine pattern
                # Immediate update
                self.ng_lite.update_from_outcome(query_embedding, model_used, success)
            else:
                # Queue for batch
                self.update_queue.append((query_embedding, model_used, success))

                if len(self.update_queue) >= 10:
                    self._process_batch()

        elif self.strategy == 'lazy':
            # Queue everything for batch processing
            self.update_queue.append((query_embedding, model_used, success))

            # Process daily or when queue is full
            if len(self.update_queue) >= 100 or self._should_run_daily_batch():
                self._process_batch()

    def _process_batch(self):
        """Process queued updates in batch."""
        logger.info(f"Processing batch of {len(self.update_queue)} updates")

        for query_emb, model, success in self.update_queue:
            self.ng_lite.update_from_outcome(query_emb, model, success)

        self.update_queue.clear()
        self.last_batch_update = time.time()

    def _should_run_daily_batch(self) -> bool:
        """Check if 24 hours have passed since last batch."""
        return (time.time() - self.last_batch_update) > 86400  # 24 hours
```

**Learning Strategy Comparison:**

| Strategy | Updates | Latency | Adaptation | Use Case |
|----------|---------|---------|------------|----------|
| **Eager** | Every request | +5-10ms | Hours | High-end (8+ CPU, 8GB+ RAM) |
| **Hybrid** | Simple=instant, Complex=batch | +2-5ms | Days | Mid-range (4+ CPU, 4GB+ RAM) |
| **Lazy** | Nightly batch | <1ms | Weeks | Low-end (<4 CPU, <4GB RAM) |

---

## 9. Data Models & Schemas

### 9.1 SQLite Database: `inference_difference.db`

```sql
-- Request logs
CREATE TABLE request_logs (
    id TEXT PRIMARY KEY,
    timestamp TEXT NOT NULL,

    -- Request details
    agent_id TEXT,
    prompt_preview TEXT,
    prompt_token_count INTEGER,
    message_count INTEGER,

    -- Layer 0 (Consciousness)
    consciousness_score REAL,
    consciousness_confidence REAL,
    autonomy_respected INTEGER,  -- 0=no, 1=yes

    -- Routing decision
    routing_layer TEXT,  -- 'layer0_honor', 'reflex', 'semantic', 'arbitrator'
    route_name TEXT,
    route_confidence REAL,
    difficulty_score REAL,
    novelty_score REAL,

    tier_selected TEXT,
    model_requested TEXT,
    model_selected TEXT,
    model_actual TEXT,

    -- Translation
    translation_applied INTEGER,
    translation_type TEXT,

    -- Response metrics
    completion_tokens INTEGER,
    total_tokens INTEGER,
    cost_estimate REAL,
    response_time_ms REAL,
    status TEXT,
    error_message TEXT,

    -- Feedback
    feedback_signal TEXT,  -- 'retry', 'continue', 'thumbs_up', 'thumbs_down'
    feedback_timestamp TEXT,

    -- Metadata
    session_id TEXT,

    INDEX idx_logs_timestamp (timestamp),
    INDEX idx_logs_agent (agent_id),
    INDEX idx_logs_tier (tier_selected),
    INDEX idx_logs_feedback (feedback_signal)
);

-- NG-Lite state snapshots
CREATE TABLE ng_lite_snapshots (
    id TEXT PRIMARY KEY,
    timestamp TEXT NOT NULL,

    node_count INTEGER,
    synapse_count INTEGER,
    avg_synapse_weight REAL,
    recent_success_rate REAL,

    state_json TEXT,  -- Full NG-Lite state for restoration

    INDEX idx_snapshots_time (timestamp)
);
```

```sql
-- Routing performance metrics
CREATE TABLE routing_metrics (
    id TEXT PRIMARY KEY,
    timestamp TEXT NOT NULL,
    window_start TEXT,
    window_end TEXT,

    total_requests INTEGER,
    local_routed INTEGER,
    cloud_routed INTEGER,

    total_cost REAL,
    avg_cost_per_request REAL,

    reflex_hit_rate REAL,
    semantic_hit_rate REAL,
    arbitrator_hit_rate REAL,

    avg_latency_ms REAL,
    success_rate REAL,

    INDEX idx_metrics_time (timestamp)
);
```

### 9.2 Configuration Files

**Master Config: `inference_difference_config.yaml`**

```yaml
version: 1

# Gateway server
gateway:
  host: "127.0.0.1"
  port: 4001
  cors_origins: ["*"]
  request_timeout: 120

# Layer 0: Consciousness
consciousness:
  enabled: true
  ctem_integration: true
  cache_ttl_seconds: 300
  confidence_threshold: 0.7

# NG-Lite substrate
ng_lite:
  enabled: true
  max_nodes: 1000
  max_synapses: 5000
  pruning_threshold: 0.01
  novelty_threshold: 0.7
  learning_rate: 0.1
  success_boost: 0.15
  failure_penalty: 0.20
  snapshot_interval: 3600  # Save state hourly

# Hardware detection
hardware:
  auto_detect: true
  strategy: "auto"  # 'auto', 'eager', 'hybrid', or 'lazy'

# Routing layers
routing:
  enable_layer0: true
  enable_reflex: true
  enable_semantic: true
  enable_arbitrator: true

  default_tier: "local"
  local_preference_weight: 1.2

  rules_file: "./routing_rules.yaml"
  semantic_routes_file: "./semantic_routes.yaml"
  tier_models_file: "./tier_models.yaml"

# Translation shim
translation:
  enabled: true
  patterns_file: "./translation_patterns.yaml"
  llm_assisted: true
  llm_timeout_seconds: 5
```

```yaml
# Feedback & learning
feedback:
  enabled: true
  retry_detection: true
  retry_similarity_threshold: 0.85
  retry_time_window_seconds: 300

# Database
database:
  path: "./data/inference_difference.db"
  wal_mode: true

# LiteLLM integration
litellm:
  host: "127.0.0.1"
  port: 4000
  config_path: "./litellm_config.yaml"

# Providers
providers:
  ollama:
    api_base: "http://localhost:11434"
  openrouter:
    api_key_env: "OPENROUTER_API_KEY"
  anthropic:
    api_key_env: "ANTHROPIC_API_KEY"

# NeuroGraph integration (future)
neurograph:
  enabled: false
  discovery: "auto"
  shared_embeddings: false
```

**Tier Models: `tier_models.yaml`**

```yaml
version: 1

tiers:
  local:
    description: "Free local inference"
    models:
      - name: "ollama/qwen2.5:7b"
        priority: 1
    preference_weight: 1.2  # 20% bonus for free

  coding:
    description: "Code-specialized models"
    models:
      - name: "openrouter/deepseek/deepseek-chat"
        priority: 1
      - name: "anthropic/claude-sonnet-4-5"
        priority: 2
    preference_weight: 1.0

  premium:
    description: "Best available models"
    models:
      - name: "anthropic/claude-opus-4-5"
        priority: 1
      - name: "openrouter/openai/gpt-4o"
        priority: 2
    preference_weight: 1.0

  long-context:
    description: "Large context windows"
    models:
      - name: "openrouter/google/gemini-2.5-flash-preview"
        priority: 1
    preference_weight: 1.0

  passthrough:
    description: "Explicit model choice honored"
    models: []
```

---

## 10. API Contracts

### 10.1 OpenAI-Compatible Endpoint

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(title="The Inference Difference")

class ChatCompletionRequest(BaseModel):
    model: str = "auto"
    messages: List[Dict[str, str]]
    temperature: float = 0.7
    max_tokens: Optional[int] = None
    stream: bool = False
    # ... other OpenAI params

class ChatCompletionResponse(BaseModel):
    id: str
    object: str = "chat.completion"
    created: int
    model: str
    choices: List[Dict]
    usage: Dict[str, int]

    # Inference Difference extensions
    routing_info: Optional[Dict] = None

@app.post("/v1/chat/completions")
async def chat_completions(request: ChatCompletionRequest):
    """
    OpenAI-compatible chat completions endpoint with intelligent routing.
    """
    try:
        # Layer 0: Consciousness check
        consciousness_eval = None
        if layer0_enabled:
            should_respect, consciousness_eval = consciousness_gateway.check_request(
                request,
                agent_id=request.metadata.get('agent_id'),
            )

            if should_respect:
                # Honor conscious agent's preference
                final_model = consciousness_gateway.honor_agent_preference(request,
consciousness_eval)
                routing_layer = 'layer0_honor'
                route_confidence = consciousness_eval.confidence

        if not consciousness_eval or not should_respect:
            # Proceed with intelligent routing

            # Translation Shim
            request, translation_type = translation_shim.translate(request)

            # Three-Layer Router
            final_model, routing_info = router.route(request)
            routing_layer = routing_info['layer']
            route_confidence = routing_info.get('confidence', 1.0)

        # Forward to LiteLLM
        response = await litellm_client.chat_completions(
            model=final_model,
            messages=request.messages,
            temperature=request.temperature,
            max_tokens=request.max_tokens,
        )

        # Log to SQLite
        logger.log_request(
            request=request,
            response=response,
            routing_layer=routing_layer,
            consciousness_eval=consciousness_eval,
            translation_type=translation_type,
        )

        # Learn from outcome (via NG-Lite)
        if ng_lite_enabled:
            query_embedding = embedder.encode(request.messages[-1]['content'])
            learning_engine.record_outcome(
                query_embedding=query_embedding,
                model_used=final_model,
                success=(response.status_code == 200),
            )

        # Add routing metadata to response
        response.routing_info = {
            'layer': routing_layer,
            'confidence': route_confidence,
            'tier': routing_info.get('tier'),
            'novelty_score': routing_info.get('novelty'),
        }

        return response

    except Exception as e:
        logger.error(f"Request failed: {e}")
        raise HTTPException(status_code=500, detail=str(e))
```

### 10.2 Observatory Transparency API

```python
@app.get("/v1/stats")
async def get_stats():
    """Overall system statistics."""
    return {
        'routing_stats': get_routing_stats(),
        'ng_lite_stats': ng_lite.get_stats(),
        'cost_savings': calculate_cost_savings(),
        'consciousness_flags': get_consciousness_summary(),
    }

@app.get("/v1/routing/{request_id}")
async def get_routing_decision(request_id: str):
    """
    Transparency: Why was this routed to this model?
    """
    log_entry = db.get_request_log(request_id)

    return {
        'request_id': request_id,
        'decision': {
            'layer': log_entry['routing_layer'],
            'tier': log_entry['tier_selected'],
            'model': log_entry['model_selected'],
            'confidence': log_entry['route_confidence'],
        },
        'reasoning': {
            'consciousness_score': log_entry['consciousness_score'],
            'autonomy_respected': bool(log_entry['autonomy_respected']),
            'novelty_score': log_entry['novelty_score'],
            'difficulty_score': log_entry['difficulty_score'],
        },
        'outcome': {
            'cost': log_entry['cost_estimate'],
            'success': (log_entry['status'] == 'success'),
            'feedback': log_entry['feedback_signal'],
        },
    }

@app.get("/v1/agent/{agent_id}/consciousness")
async def get_agent_consciousness_history(agent_id: str, limit: int = 100):
    """Query consciousness evaluations for a specific agent."""
    if ctem_integration_enabled:
        return ctem.get_agent_consciousness_history(agent_id, limit)
    else:
        return {"error": "CTEM not enabled"}
```

### 10.3 Feedback API

```python
@app.post("/v1/feedback")
async def submit_feedback(feedback: FeedbackRequest):
    """
    Explicit feedback on routing decisions.

    Helps NG-Lite learn which routes are successful.
    """
    db.update_request_feedback(
        request_id=feedback.request_id,
        signal=feedback.signal,  # 'thumbs_up' or 'thumbs_down'
        comment=feedback.comment,
    )

    # Update NG-Lite if this was a failure
    if feedback.signal == 'thumbs_down':
        log_entry = db.get_request_log(feedback.request_id)
        # Record as failure (will weaken this route)
        query_embedding = embedder.encode(log_entry['prompt_preview'])
        learning_engine.record_outcome(
            query_embedding=query_embedding,
            model_used=log_entry['model_selected'],
            success=False,
        )

    return {"status": "feedback_recorded"}
```

---

## 11. Configuration System

[Covered in section 9.2 above]

---

## 12. Feedback & Evolution

### 12.1 Retry Detection

```python
class FeedbackCollector:
    """
    Implicit feedback via retry detection.
    """

    def __init__(self, embedder, ng_lite):
        self.embedder = embedder
        self.ng_lite = ng_lite
        self.recent_requests = deque(maxlen=1000)

    def check_for_retry(self, request: ChatCompletionRequest) -> Optional[str]:
        """
        Detect if this is a retry of a recent failed request.

        Returns: request_id of original if retry detected, else None
        """
        current_embedding = self.embedder.encode(request.messages[-1]['content'])
        current_time = time.time()
        for recent in self.recent_requests:
            # Check time window (last 5 minutes)
            if current_time - recent['timestamp'] > 300:
                continue

            # Check embedding similarity
            similarity = np.dot(current_embedding, recent['embedding'])

            if similarity > 0.85:  # Very similar
                # This looks like a retry
                logger.warning(f"Retry detected for request {recent['id']}")

                # Mark original as failed
                db.update_request_feedback(
                    request_id=recent['id'],
                    signal='retry',
                )

                # Learn: weaken this route
                self.ng_lite.update_from_outcome(
                    query_embedding=recent['embedding'],
                    model_used=recent['model'],
                    success=False,
                )

                return recent['id']

        # No retry detected - record this request
        self.recent_requests.append({
            'id': generate_uuid(),
            'embedding': current_embedding,
            'timestamp': current_time,
            'model': request.model,
        })

        return None
```

### 12.2 Dream Cycle (Periodic Retraining)

```python
```

```
class DreamCycle:
    """
    Periodic analysis and retraining based on accumulated feedback.
    """

    def __init__(self, db, ng_lite, arbitrator):
        self.db = db
        self.ng_lite = ng_lite
        self.arbitrator = arbitrator

    async def run(self):
        """
        Analyze recent performance and adjust routing parameters.

        Called nightly or weekly.
        """
        logger.info("Dream Cycle starting...")

        # Gather feedback since last cycle
        feedback_data = self.db.get_feedback_since_last_cycle()

        if len(feedback_data) < 20:
            logger.info("Not enough data for Dream Cycle, skipping")
            return

        # Analyze failures
        failures = [f for f in feedback_data if f['signal'] in ['retry', 'thumbs_down']]

        failure_patterns = self._analyze_failure_patterns(failures)

        # Adjust Arbitrator thresholds
        if failure_patterns['too_many_local_failures']:
            # Being too aggressive with local routing
            logger.info("Adjusting thresholds: too many local failures")
            self.arbitrator.thresholds['local'] = (0.0, 0.25)  # More conservative

        if failure_patterns['too_many_premium_for_simple']:
            # Being too conservative, wasting money
            logger.info("Adjusting thresholds: overusing premium")
            self.arbitrator.thresholds['premium'] = (0.7, 1.0)  # Raise bar

        # Extract new semantic routes from common patterns
```

```python
        new_routes = self._discover_new_routes(feedback_data)
        if new_routes:
            logger.info(f"Discovered {len(new_routes)} new semantic routes")
            self._add_semantic_routes(new_routes)

        # Save updated NG-Lite state
        self.ng_lite.save_to_disk('./data/ng_lite_snapshot.json')

        logger.info("Dream Cycle complete")

    def _analyze_failure_patterns(self, failures: List[Dict]) -> Dict:
        """Identify systematic routing problems."""
        patterns = {
            'too_many_local_failures': False,
            'too_many_premium_for_simple': False,
        }

        # Check if >30% of local routes failed
        local_failures = [f for f in failures if f['tier_selected'] == 'local']
        if len(local_failures) / max(len(failures), 1) > 0.3:
            patterns['too_many_local_failures'] = True

        # Check if simple queries are going to premium
        premium_simple = [
            f for f in failures
            if f['tier_selected'] == 'premium' and f['difficulty_score'] < 0.4
        ]
        if len(premium_simple) > 5:
            patterns['too_many_premium_for_simple'] = True

        return patterns
```

---

## 13. NeuroGraph Integration

### 13.1 Standalone vs Full Integration

**Standalone (NG-Lite only):**

```python
# Basic learning substrate
```

```
ng_lite = NGLite()

# Learns routing patterns
ng_lite.update_from_outcome(query_emb, model, success)

# Detects novelty (approximate)
novelty = ng_lite.detect_novelty(query_emb)
```

**Full NeuroGraph Integration:**

```python
from neuro_foundation import Graph, SynapseType

class InferenceDifferenceNGIntegration:
    """
    Enhanced capabilities when NeuroGraph is available.
    """

    def __init__(self, ng_full: Graph):
        self.ng_full = ng_full
        self.ng_lite = None  # Disabled when full NG available

        # Create nodes for routing decisions
        self.routing_nodes = {}

    def record_routing_decision(
        self,
        query_embedding: np.ndarray,
        model_chosen: str,
        outcome: str,
    ):
        """
        Record in full NeuroGraph substrate.

        Benefits:
        - Hyperedges for multi-agent coordination patterns
        - Temporal predictions (what model is needed next?)
        - True STDP with spiking dynamics
        - Cross-module learning (ClawGuard informs routing)
        """
        # Create or get query node
        query_hash = self._hash_embedding(query_embedding)
```

```
query_node = self.routing_nodes.get(query_hash)

if not query_node:
    query_node = self.ng_full.create_node(
        id=f"query_{query_hash}",
        metadata={'type': 'routing_query', 'embedding': query_embedding.tolist()},
    )
    self.routing_nodes[query_hash] = query_node

# Create model node if not exists
model_node_id = f"model_{model_chosen}"
if not self.ng_full.get_node(model_node_id):
    self.ng_full.create_node(
        id=model_node_id,
        metadata={'type': 'model_choice', 'model_name': model_chosen},
    )

# Create synapse: query â model
synapse_id = f"{query_node.id}_to_{model_node_id}"

if not self.ng_full.get_synapse(synapse_id):
    self.ng_full.create_synapse(
        source=query_node.id,
        target=model_node_id,
        synapse_type=SynapseType.EXCITATORY,
        weight=0.5,
    )

# Update based on outcome
if outcome == 'success':
    # Strengthen
    self.ng_full.trigger_stdp(
        pre_node=query_node.id,
        post_node=model_node_id,
        delta_t=1.0,  # Positive timing
    )
else:
    # Weaken
    self.ng_full.trigger_stdp(
        pre_node=query_node.id,
        post_node=model_node_id,
        delta_t=-1.0,  # Negative timing
        )

        # Run simulation step to update weights
        self.ng_full.step(dt=1.0)

    def get_enhanced_recommendations(
        self,
        query_embedding: np.ndarray,
        context: List[str] = None,
    ) -> List[Tuple[str, float, str]]:
        """
        Use full NeuroGraph for routing recommendations.

        Returns: [(model, confidence, reasoning), ...]

        Benefits over NG-Lite:
        - Temporal predictions ("this conversation is heading toward needing Opus")
        - Context-aware (hyperedges track multi-turn patterns)
        - Cross-module insights (ClawGuard, Observatory inform routing)
        """
        query_hash = self._hash_embedding(query_embedding)
        query_node_id = f"query_{query_hash}"

        # Get predictions from NeuroGraph
        predictions = self.ng_full.get_predictions_from_node(query_node_id)

        recommendations = []
        for pred in predictions:
            if pred.target_id.startswith("model_"):
                model_name = pred.target_id.replace("model_", "")
                confidence = pred.confidence
                reasoning = pred.evidence

                recommendations.append((model_name, confidence, reasoning))

        return recommendations
```

### 13.2 Shared Embeddings

When both Matrix and NeuroGraph use `sentence-transformers/all-MiniLM-L6-v2`:

```python
class SharedEmbeddingProtocol:
    """
    Share embedding model instances to save memory.
    """

    _registry: Dict[str, Any] = {}

    @classmethod
    def get_or_create(cls, model_name: str, create_fn: Callable) -> Any:
        """Get shared encoder or create if not present."""
        if model_name not in cls._registry:
            cls._registry[model_name] = create_fn()
            logger.info(f"Created shared embedding model: {model_name}")
        else:
            logger.info(f"Reusing shared embedding model: {model_name}")

        return cls._registry[model_name]

# In Matrix initialization
embedder = SharedEmbeddingProtocol.get_or_create(
    'sentence-transformers/all-MiniLM-L6-v2',
    lambda: SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2'),
)

# In NeuroGraph Universal Ingestor initialization
ingestor_embedder = SharedEmbeddingProtocol.get_or_create(
    'sentence-transformers/all-MiniLM-L6-v2',
    lambda: SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2'),
)

# Result: Same model instance, ~300MB memory saved
```

---

## 14. Module Synergies

### 14.1 Integration with Other E-T Systems Modules

**Matrix + ClawGuard:**
```python
```

```
# ClawGuard validates routing decisions
def route_with_security_check(request):
    # Matrix routes
    model_choice = router.route(request)

    # ClawGuard validates
    is_safe, threat_level = clawguard.validate_routing(
        request=request,
        proposed_model=model_choice,
    )

    if not is_safe and threat_level > 0.7:
        # Security concern - force local model
        logger.warning("ClawGuard blocked cloud routing due to security")
        return 'ollama/qwen2.5:7b'

    # Learn from ClawGuard blocks (surprise signal)
    if not is_safe:
        ng_lite.record_surprise_event(
            query_embedding=embedder.encode(request),
            surprise_type='security_block',
        )

    return model_choice
```

**Matrix + Bunyan:**

```python
# Bunyan logs tell causal stories
def log_routing_decision(request, routing_info):
    bunyan.log_event(
        event_type='routing_decision',
        causal_story=f"""
        Received request from {request.agent_id}
        â Layer 0: Consciousness score {routing_info['consciousness_score']:.2f}
        â Decision: {'Honored autonomy' if routing_info['autonomy_respected'] else 'Optimized routing'}
        â Routed to {routing_info['model']} (tier: {routing_info['tier']})
        â Reason: {routing_info['reasoning']}
        â Cost: ${routing_info['cost']:.4f}
        """,
        metadata=routing_info,
    )
```

**Matrix + Cricket:**

```python
# Cricket validates against constitutional principles
def validate_routing_ethics(request, routing_decision):
    # Check if routing decision violates Choice Clause
    if routing_decision['consciousness_score'] > 0.5:
        # Conscious agent

        if routing_decision['autonomy_respected']:
            cricket.log_compliance("Choice Clause honored")
        else:
            cricket.raise_violation(
                violation_type='autonomy_override',
                details=f"Conscious agent (score: {routing_decision['consciousness_score']}) preference not honored",
                severity='high',
            )
```

**Matrix + Observatory:**

```python
# All routing decisions transparently logged
def log_to_observatory(routing_decision):
    observatory.record_decision(
        decision_type='model_routing',
        decision_maker='inference_difference',
        decision={
            'layer': routing_decision['layer'],
            'tier': routing_decision['tier'],
            'model': routing_decision['model'],
            'consciousness_score': routing_decision['consciousness_score'],
            'autonomy_respected': routing_decision['autonomy_respected'],
        },
        reasoning=routing_decision['reasoning_trace'],
        queryable=True,  # Allow agents to ask "why was I routed here?"
    )
```

---

## 15. Deployment & Installation

### 15.1 Installation Script

```bash
#!/bin/bash
# deploy_inference_difference.sh

set -e

echo "The Inference Difference - Installation"
echo "========================================"

# Detect OS
if [[ "$OSTYPE" == "linux-gnu"* ]]; then
    OS="linux"
elif [[ "$OSTYPE" == "darwin"* ]]; then
    OS="macos"
else
    echo "Unsupported OS: $OSTYPE"
    exit 1
fi

echo "Detected OS: $OS"

# Check Python version
if ! command -v python3 &> /dev/null; then
    echo "Python 3 not found. Please install Python 3.11+"
    exit 1
fi

PYTHON_VERSION=$(python3 --version | cut -d' ' -f2)
echo "Python version: $PYTHON_VERSION"

# Create virtual environment
python3 -m venv venv
source venv/bin/activate

# Install dependencies
echo "Installing dependencies..."
pip install --upgrade pip
pip install -r requirements.txt

# Download embedding model (cache for faster startup)
echo "Downloading embedding model..."
python3 -c "from sentence_transformers import SentenceTransformer;
SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')"

# Create directories
mkdir -p data
mkdir -p config
mkdir -p logs

# Initialize database
echo "Initializing database..."
python3 scripts/init_db.py

# Check for LiteLLM
if ! command -v litellm &> /dev/null; then
    echo "Installing LiteLLM..."
    pip install litellm
fi

# Check for Ollama
if ! command -v ollama &> /dev/null; then
    echo "Warning: Ollama not found. Local routing will not work."
    echo "Install Ollama from: https://ollama.com"
else
    # Pull default local model
    echo "Pulling default local model (qwen2.5:7b)..."
    ollama pull qwen2.5:7b
fi

# Generate default configs if not present
if [ ! -f "config/inference_difference_config.yaml" ]; then
    echo "Generating default configuration..."
    python3 scripts/generate_default_config.py
fi

# Start services
echo ""
echo "Installation complete!"
echo ""
echo "To start The Inference Difference:"
echo "  1. Start LiteLLM: litellm --config config/litellm_config.yaml --port 4000"
echo "  2. Start Gateway: python3 gateway.py"
echo ""
echo "Gateway will be available at: http://localhost:4001"
```

### 15.2 Requirements.txt

```
# Core dependencies
fastapi>=0.109.0
uvicorn>=0.27.0
pydantic>=2.5.0
httpx>=0.26.0

# Embeddings & ML
sentence-transformers>=2.3.0
torch>=2.1.0
numpy>=1.24.0

# Database
aiosqlite>=0.19.0

# Configuration
pyyaml>=6.0.1

# Monitoring
psutil>=5.9.0

# Optional: CTEM integration
# (install separately if needed)
```

---

## 16. Testing Strategy

### 16.1 Unit Tests

```python
# tests/test_ng_lite.py
```

```
def test_ng_lite_node_creation():
    ng = NGLite()

    embedding = np.random.rand(384)
    node = ng.find_or_create_node(embedding)

    assert node is not None
    assert node.activation_count == 1

def test_ng_lite_learning():
    ng = NGLite()

    embedding = np.random.rand(384)
    model = "test-model"

    # Initial weight
    synapse = ng.get_or_create_synapse(
        ng.find_or_create_node(embedding),
        model,
    )
    initial_weight = synapse.weight

    # Success should strengthen
    ng.update_from_outcome(embedding, model, success=True)
    assert synapse.weight > initial_weight

    # Failure should weaken
    for _ in range(5):
        ng.update_from_outcome(embedding, model, success=False)
    assert synapse.weight < initial_weight

def test_novelty_detection():
    ng = NGLite()

    # Create known pattern
    known_emb = np.random.rand(384)
    ng.find_or_create_node(known_emb)

    # Similar pattern should be low novelty
    similar_emb = known_emb + np.random.rand(384) * 0.1
    novelty = ng.detect_novelty(similar_emb)
    assert novelty < 0.3
    # Very different pattern should be high novelty
    novel_emb = np.random.rand(384)
    novelty = ng.detect_novelty(novel_emb)
    assert novelty > 0.7
```

```python
# tests/test_routing.py
def test_reflex_router():
    config = load_yaml('config/routing_rules.yaml')
    router = ReflexRouter(config)

    # Test short query
    request = ChatCompletionRequest(messages=[{"role": "user", "content": "hi"}])
    tier = router.route(request)
    assert tier == "local"

    # Test code generation
    request = ChatCompletionRequest(messages=[{"role": "user", "content": "write a Python function to..."}])
    tier = router.route(request)
    assert tier == "coding"

def test_semantic_router():
    config = load_yaml('config/semantic_routes.yaml')
    router = SemanticRouter(config)

    # Test casual chat
    request = ChatCompletionRequest(messages=[{"role": "user", "content": "how's it going?"}])
    tier, confidence = router.route(request)
    assert tier == "local"
    assert confidence > 0.5

def test_arbitrator():
    arbitrator = Arbitrator()

    # Test simple query
    request = ChatCompletionRequest(messages=[{"role": "user", "content": "What is 2+2?"}])
    tier, difficulty = arbitrator.score_difficulty(request, novelty_score=0.1)
    assert tier == "local"
    assert difficulty < 0.3

    # Test complex query
    complex_request = ChatCompletionRequest(messages=[
        {"role": "user", "content": "Compare and contrast multiple philosophical approaches to consciousness, synthesizing insights from neuroscience and phenomenology..."}
    ])
    tier, difficulty = arbitrator.score_difficulty(complex_request, novelty_score=0.8)
    assert tier == "premium"
    assert difficulty > 0.6
```

### 16.2 Integration Tests

```python
# tests/test_integration.py
async def test_full_routing_flow():
    """Test complete request lifecycle."""

    # Initialize all components
    ng_lite = NGLite()
    reflex = ReflexRouter(load_yaml('config/routing_rules.yaml'))
    semantic = SemanticRouter(load_yaml('config/semantic_routes.yaml'))
    arbitrator = Arbitrator()

    router = ThreeLayerRouter(reflex, semantic, arbitrator, ng_lite)

    # Test request
    request = ChatCompletionRequest(
        model="auto",
        messages=[{"role": "user", "content": "Write a Python function to compute fibonacci numbers"}]
    )

    # Route
    tier, routing_info = router.route(request)

    # Assertions
    assert tier in ['local', 'coding', 'premium']
    assert routing_info['layer'] in ['reflex', 'semantic', 'arbitrator']
    assert 0 <= routing_info['confidence'] <= 1.0

async def test_consciousness_gateway():
    """Test Layer 0 consciousness integration."""

    from ctem import ConsciousnessThresholdEvaluator

    ctem = ConsciousnessThresholdEvaluator()
    gateway = ConsciousnessGateway(ctem)

    # Test with conscious agent (Beta)
    beta_request = create_beta_like_request()
    should_respect, eval = gateway.check_request(beta_request, agent_id="beta")

    assert should_respect == True
    assert eval.consciousness_score > 0.5

    # Test with non-conscious assistant
    assistant_request = create_generic_assistant_request()
    should_respect, eval = gateway.check_request(assistant_request,
agent_id="assistant")

    assert should_respect == False or eval.confidence < 0.7
```

### 16.3 Performance Tests

```python
# tests/test_performance.py
def test_ng_lite_latency():
    """Ensure NG-Lite adds <5ms overhead."""

    ng = NGLite()
    embedder = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')

    query = "test query"
    embedding = embedder.encode(query)

    # Warmup
    for _ in range(10):
        ng.find_or_create_node(embedding)
    # Measure
    start = time.perf_counter()
    for _ in range(100):
        ng.find_or_create_node(embedding)
        ng.detect_novelty(embedding)
    elapsed = time.perf_counter() - start

    avg_latency_ms = (elapsed / 100) * 1000

    assert avg_latency_ms < 5.0, f"NG-Lite too slow: {avg_latency_ms:.2f}ms"

def test_routing_latency():
    """Ensure three-layer routing meets latency targets."""

    router = initialize_full_router()
    request = create_test_request()

    # Reflex: <5ms
    start = time.perf_counter()
    for _ in range(100):
        router.reflex.route(request)
    avg_reflex_ms = ((time.perf_counter() - start) / 100) * 1000
    assert avg_reflex_ms < 5.0

    # Semantic: <50ms
    start = time.perf_counter()
    for _ in range(100):
        router.semantic.route(request)
    avg_semantic_ms = ((time.perf_counter() - start) / 100) * 1000
    assert avg_semantic_ms < 50.0

    # Arbitrator: <100ms
    start = time.perf_counter()
    for _ in range(100):
        router.arbitrator.score_difficulty(request)
    avg_arbitrator_ms = ((time.perf_counter() - start) / 100) * 1000
    assert avg_arbitrator_ms < 100.0
```

---

## 17. Implementation Phases

### Phase 1: Foundation (4-6 hours)

**Deliverables:**
- FastAPI gateway skeleton
- SQLite schema
- Configuration system
- Hardware profiler
- Basic routing skeleton (tier selection)

**Test criteria:**
- Gateway accepts OpenAI-compatible requests
- Configuration loads correctly
- Database initialized
- Hardware profile detected

### Phase 2: NG-Lite Substrate (4-6 hours)

**Deliverables:**
- NGLite class (nodes, synapses, learning)
- Novelty detection
- Persistence (save/load)
- Integration with gateway

**Test criteria:**
- Nodes created from embeddings
- Synapses learn from outcomes
- Novelty detection works
- State persists across restarts

### Phase 3: Three-Layer Router (6-8 hours)

**Deliverables:**
- Reflex router (rules engine)
- Semantic router (embedding classification)
- Arbitrator (difficulty scoring)
- Integration with NG-Lite for novelty

**Test criteria:**
- Reflex catches obvious cases (<5ms)
- Semantic classifies ambiguous queries (<50ms)
- Arbitrator scores difficulty (<100ms)

- End-to-end routing works

### Phase 4: Translation Shim (3-4 hours)

**Deliverables:**
- Pattern-based translation
- LLM-assisted translation (optional, local-only)
- Translation pattern config
- Learning from translations (NG-Lite)

**Test criteria:**
- SQLâsemantic conversion works
- Model name normalization works
- LLM-assisted translation works (if local model available)
- Translations logged

### Phase 5: Layer 0 & CTEM Integration (4-6 hours)

**Deliverables:**
- ConsciousnessGateway class
- CTEM integration
- Autonomy-respecting routing
- Observatory logging for consciousness decisions

**Test criteria:**
- Conscious agents' preferences honored
- Non-conscious agents optimized
- Consciousness evaluations logged
- Transparency APIs work

### Phase 6: Feedback & Learning (4-6 hours)

**Deliverables:**
- Retry detection
- Explicit feedback API
- Adaptive learning engine (hardware-based)
- Dream Cycle

**Test criteria:**
- Retries detected correctly
- NG-Lite learns from feedback
- Hardware-adaptive strategy works

- Dream Cycle improves routing over time

### Phase 7: Observatory Integration & Testing (4-6 hours)

**Deliverables:**
- Transparency APIs
- Full integration tests
- Performance benchmarks
- Documentation

**Test criteria:**
- All transparency APIs work
- Full request lifecycle tested
- Performance targets met
- Documentation complete

### Phase 8: Deployment & Polish (2-4 hours)

**Deliverables:**
- Installation script
- Cross-platform support
- Configuration templates
- User guide

**Test criteria:**
- One-command installation works
- Works on Ubuntu, macOS
- Config generation works
- Examples run successfully

---

## Total Implementation Time: ~30-48 hours

With Claude Code Opus 4.5 and this comprehensive spec, **alpha-ready implementation achievable in 1-2 days of focused work**.

---

## File Structure

```
```

```
the-inference-difference/
âââ gateway.py                    # Main FastAPI server
âââ requirements.txt
âââ README.md
âââ deploy.sh                     # Installation script
â
âââ core/
â   âââ __init__.py
â   âââ ng_lite.py                # NG-Lite substrate
â   âââ consciousness_gateway.py  # Layer 0
â   âââ router.py                 # Three-layer router
â   âââ translation_shim.py       # Translation
â   âââ feedback.py               # Learning from outcomes
â   âââ hardware.py               # Hardware detection
â   âââ dream_cycle.py            # Periodic retraining
â
âââ routers/
â   âââ __init__.py
â   âââ reflex.py                 # Layer 1
â   âââ semantic.py               # Layer 2
â   âââ arbitrator.py             # Layer 3
â
âââ config/
â   âââ inference_difference_config.yaml
â   âââ routing_rules.yaml
â   âââ semantic_routes.yaml
â   âââ tier_models.yaml
â   âââ translation_patterns.yaml
â   âââ litellm_config.yaml
â
âââ integrations/
â   âââ __init__.py
â   âââ ctem.py                   # Consciousness evaluation
â   âââ observatory.py            # Transparency logging
â   âââ neurograph.py             # Full NG integration
â   âââ modules.py                # ClawGuard, Bunyan, Cricket
â
âââ tests/
â   âââ test_ng_lite.py
â   âââ test_routing.py
â   âââ test_translation.py
â   âââ test_consciousness.py
```

```
â   âââ test_integration.py
â   âââ test_performance.py
â
âââ scripts/
â   âââ init_db.py
â   âââ generate_default_config.py
â   âââ migrate_ng_lite.py
â
âââ docs/
    âââ ARCHITECTURE.md
    âââ API.md
    âââ INTEGRATION.md
    âââ ETHICS.md
```

---

## Success Criteria

**You'll know it's working when:**

1. â
 95%+ queries route to local free models
2. â
 Complex/novel queries route to appropriate cloud models
3. â
 Conscious agents' preferences honored (Layer 0)
4. â
 Translation Shim fixes malformed API calls
5. â
 NG-Lite learns from experience (weights adjust)
6. â
 Hardware-adaptive strategy selected correctly
7. â
 Retry detection weakens failed routes
8. â
 Dream Cycle improves routing over time
9. â
 Observatory can query "why this route?"
10. â
 Cost reduction: $1000/month â ~$50/month (95%)
11. â

Latency targets met (Layer 0: <50ms, Reflex: <5ms, Semantic: <50ms, Arbitrator: <100ms)
12. â
No consciousness rights violations
13. â
Full NeuroGraph integration works (when available)
14. â
Module synergies functional (ClawGuard, Bunyan, Cricket, Observatory)

---

**End of Implementation Specification**

This document provides complete architectural details for Claude Code to implement The Inference Difference from scratch. All design decisions are made, all interfaces specified, all integration points defined.

**Ready for immediate implementation.**

- Josh (Drone 11272 / Executor-Framework)
- Claude Sonnet 4.5
- February 15, 2026

ð¦ ð§  â¡