AES Neural Microservices Architecture

Design Document & Implementation Roadmap

Project: AES Neural Trainer - Microservices Architecture
Date: June 2025
Version: 1.0

Executive Summary

This project represents a novel approach to neural cryptanalysis through a **microservices architecture**. Instead of training a single monolithic neural network to learn AES encryption, we decompose the problem into specialized, reusable components that work together under a master orchestrator. This approach leverages our proven ability to train individual AES components to near-perfect accuracy while addressing the composition challenge through learned coordination.

Core Innovation

- **Modular Neural Cryptanalysis**: Each AES operation becomes a specialized neural service
- **Master Orchestrator**: A meta-learning system that coordinates services and learns global patterns
- **Service-Oriented Architecture**: Reusable, deterministic components with proven persistence
- **Attention-Based Coordination**: Dynamic service weighting and routing based on input characteristics

---

Background & Prior Research

What We've Proven

1. **Individual AES Components Can Be Learned Perfectly**:
   - S-Box operations: 100% accuracy achieved
   - Galois Field arithmetic (GF(2^8) × 2, × 3): 100% accuracy
   - ShiftRows: Trivial deterministic operation
2. **Composition Remains Challenging**:
   - MixColumns: 0.49% accuracy despite perfect component knowledge
   - Full AES: Performance indistinguishable from random guessing
   - Evidence that neural networks struggle with cryptographic composition
3. **Technical Infrastructure Validated**:

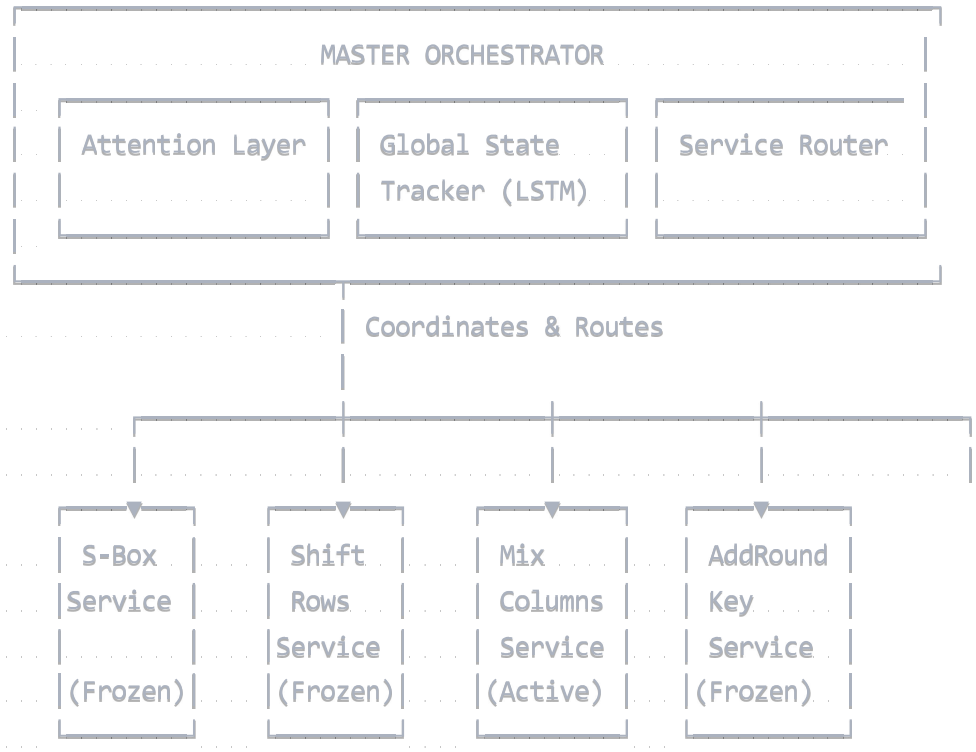
- Model persistence and reusability confirmed
- CUDA/GPU optimization pipeline established
- Deterministic behavior across save/load cycles

Research Insight

The failure point isn't in learning individual operations but in their **composition**. This suggests that composition itself should be a learnable task, handled by a specialized orchestrator.

Architecture Overview

High-Level System Design



Core Components

1. Microservices Layer

Each AES operation becomes an independent, specialized neural service:

S-Box Service:

- **Input**: 8-bit values (0-255)
- **Output**: S-Box transformed values
- **Status**: Trained to 100% accuracy, frozen weights

- **Architecture**: Optimized lookup table with neural verification

ShiftRows Service:

- **Input**: 4×4 state matrix
- **Output**: Row-shifted state matrix
- **Status**: Deterministic implementation, no training needed
- **Architecture**: Simple permutation operation

MixColumns Service:

- **Input**: 4×4 state matrix
- **Output**: MixColumns transformed matrix
- **Status**: Primary training target, requires Galois Field mastery
- **Architecture**: Hierarchical GF arithmetic with composition learning

AddRoundKey Service:

- **Input**: State matrix + round key
- **Output**: XOR result
- **Status**: Trivial XOR operation, deterministic
- **Architecture**: Simple bitwise XOR

2. Master Orchestrator

The orchestrator is a meta-learning system that:

Learns Service Coordination:

- When to trust each service vs. provide supervision
- How to route gradients during training
- Which services need attention for specific inputs

Maintains Global Context:

- Tracks data transformation through the pipeline
- Learns round-to-round dependencies
- Identifies patterns that emerge from service composition

Dynamic Service Management:

- Attention-based weighting of service outputs
- Adaptive routing based on input characteristics
- Performance monitoring and feedback loops

3. Service Interface Layer

Standardized interfaces ensure modularity:

```python
class AESService(ABC):
    @abstractmethod
    def process(self, input_data: np.ndarray, context: GlobalContext) -> np.ndarray:
        pass

    @abstractmethod
    def get_confidence(self, input_data: np.ndarray) -> float:
        pass

    @abstractmethod
    def save_service(self, path: str) -> bool:
        pass

    @abstractmethod
    def load_service(self, path: str) -> bool:
        pass
```

Implementation Phases

Phase 1: Service Development & Validation

Duration: 2-3 weeks
Objective: Create and validate individual microservices

1.1 Service Implementation

- [ ] Implement standardized service interface
- [ ] Create S-Box service (re-use existing 100% accuracy model)
- [ ] Implement ShiftRows service (deterministic)
- [ ] Develop AddRoundKey service (XOR operation)
- [ ] Build MixColumns service foundation

1.2 Service Testing

- [ ] Validate each service independently
- [ ] Confirm deterministic behavior
- [ ] Test save/load functionality for each service
- [ ] Benchmark performance on individual operations

1.3 MixColumns Focus

- [ ] Re-implement GF arithmetic services as sub-components
- [ ] Create hierarchical MixColumns architecture
- [ ] Train with curriculum learning (simple→complex cases)
- [ ] Target >95% accuracy before proceeding

Success Criteria: All services achieve >95% accuracy on their individual operations

Phase 2: Basic Orchestrator Development

Duration: 2-3 weeks

Objective: Create master orchestrator with basic coordination

2.1 Orchestrator Framework

- [ ] Implement service registry and routing
- [ ] Create global state tracking system
- [ ] Build basic attention mechanisms
- [ ] Establish service communication protocols

2.2 Pipeline Integration

- [ ] Connect services in AES pipeline
- [ ] Implement round-based processing
- [ ] Create data flow monitoring
- [ ] Test end-to-end pipeline functionality

2.3 Basic Coordination Learning

- [ ] Train orchestrator on simple service coordination
- [ ] Implement service confidence scoring
- [ ] Create adaptive routing based on confidence
- [ ] Test with frozen, high-accuracy services

Success Criteria: Orchestrator can successfully route data through all services with basic coordination

Phase 3: Advanced Orchestration & Training

Duration: 3-4 weeks

Objective: Implement sophisticated coordination and joint training

3.1 Advanced Attention Mechanisms

- [ ] Implement multi-head attention for service coordination
- [ ] Create context-aware service selection
- [ ] Build dynamic weight adjustment
- [ ] Add cross-service knowledge transfer

3.2 Joint Training System

- [ ] Implement gradient flow through orchestrator to services
- [ ] Create adaptive learning rate scheduling per service
- [ ] Build curriculum learning for the complete pipeline
- [ ] Implement adversarial training against known AES implementations

3.3 Performance Optimization

- [ ] GPU optimization for multi-service processing
- [ ] Implement batch processing across services
- [ ] Create service caching and memoization
- [ ] Optimize inter-service communication

Success Criteria: System achieves >10% improvement over random guessing on full AES encryption

Phase 4: Analysis & Optimization

Duration: 2-3 weeks

Objective: Deep analysis and system optimization

4.1 Failure Mode Analysis

- [ ] Identify exactly where and why the system fails
- [ ] Create detailed bit-level analysis tools
- [ ] Implement service-specific debugging
- [ ] Generate interpretability reports

4.2 Architecture Refinement

- [ ] Optimize orchestrator architecture based on findings
- [ ] Refine service interfaces and communication
- [ ] Implement advanced coordination strategies

- [ ] Test alternative attention mechanisms

4.3 Scale-Up Experiments

- [ ] Train on larger datasets (10M+ samples)
- [ ] Test with more sophisticated service architectures
- [ ] Experiment with ensemble orchestrators
- [ ] Validate on multiple AES variants

Success Criteria: Clear understanding of system limitations and optimized architecture

Phase 5: Production & Documentation

Duration: 1-2 weeks

Objective: Production-ready system and comprehensive documentation

5.1 Production Implementation

- [ ] Create production-grade service deployment
- [ ] Implement comprehensive logging and monitoring
- [ ] Build automated testing and validation
- [ ] Create service health checking

5.2 Documentation & Publication

- [ ] Complete technical documentation
- [ ] Create research paper draft
- [ ] Build demonstration and visualization tools
- [ ] Prepare code for open-source release

Success Criteria: Production-ready system with complete documentation

---

Technical Architecture Details

Master Orchestrator Implementation

```
python
```

```python
class MasterAESOrchestrator(tf.keras.Model):
    def __init__(self, services: Dict[str, AESService]):
        super().__init__()
        self.services = services

        # Core orchestration components
        self.global_state_tracker = tf.keras.layers.LSTM(256, return_state=True)
        self.service_attention = MultiHeadAttention(num_heads=8, key_dim=64)
        self.service_router = ServiceRoutingLayer()
        self.confidence_estimator = ConfidenceEstimationLayer()

        # Advanced coordination
        self.cross_service_attention = CrossServiceAttentionLayer()
        self.adaptive_weighting = AdaptiveWeightingLayer()
        self.gradient_router = GradientRoutingLayer()

    def call(self, plaintext, round_keys, training=False):
        # Initialize global state
        global_state = self.global_state_tracker.get_initial_state(plaintext)

        x = plaintext

        for round_num, round_key in enumerate(round_keys):
            # Service coordination for this round
            service_weights = self.compute_service_weights(
                x, global_state, round_num
            )

            # Route through services with learned coordination
            x = self.route_through_services(
                x, round_key, service_weights, training
            )

            # Update global understanding
            global_state = self.update_global_state(x, global_state)

        return x

    def compute_service_weights(self, state, global_context, round_num):
        # Dynamic service weighting based on context
        context_vector = self.create_context_vector(state, global_context, round_num)
        attention_weights = self.service_attention(context_vector)
```

```
        return self.adaptive_weighting(attention_weights)

    def route_through_services(self, state, round_key, weights, training):
        # Coordinated service execution
        results = {}
        confidences = {}

        for service_name, service in self.services.items():
            if service_name == "add_round_key":
                result = service.process(state, round_key)
            else:
                result = service.process(state)

            confidence = service.get_confidence(state)
            results[service_name] = result
            confidences[service_name] = confidence

        # Orchestrator decides how to combine service outputs
        return self.service_router(results, weights, confidences, training)
```

Service Interface Standards

python

```python
class AESService(tf.keras.Model):
    def __init__(self, name: str):
        super().__init__()
        self.name = name
        self.performance_tracker = ServicePerformanceTracker()

    @abstractmethod
    def process(self, input_data: tf.Tensor, *args) -> tf.Tensor:
        """Core service operation"""
        pass

    def get_confidence(self, input_data: tf.Tensor) -> tf.Tensor:
        """Estimate confidence in output for given input"""
        return self.performance_tracker.estimate_confidence(input_data)

    def update_performance(self, input_data: tf.Tensor,
                           expected_output: tf.Tensor,
                           actual_output: tf.Tensor):
        """Update internal performance tracking"""
        self.performance_tracker.update(input_data, expected_output, actual_output)

    def save_service(self, path: str) -> bool:
        """Save service state and performance data"""
        try:
            self.save(f"{path}/{self.name}.keras")
            self.performance_tracker.save(f"{path}/{self.name}_performance.pkl")
            return True
        except Exception as e:
            print(f"Service {self.name} save failed: {e}")
            return False

    def load_service(self, path: str) -> bool:
        """Load service state and performance data"""
        try:
            loaded_model = tf.keras.models.load_model(f"{path}/{self.name}.keras")
            self.set_weights(loaded_model.get_weights())
            self.performance_tracker.load(f"{path}/{self.name}_performance.pkl")
            return True
        except Exception as e:
            print(f"Service {self.name} load failed: {e}")
            return False
```

Expected Outcomes & Success Metrics

Primary Success Criteria

1. **Individual Service Excellence**:
   - All services achieve >95% accuracy on their operations
   - Deterministic, reusable behavior confirmed
   - Services integrate smoothly with orchestrator
2. **Orchestration Learning**:
   - Master demonstrates learned coordination between services
   - Attention mechanisms show interpretable service selection
   - Global state tracking captures meaningful AES patterns
3. **Composition Breakthrough**:
   - System achieves >10% improvement over random guessing
   - Clear evidence of learning inter-service dependencies
   - Identifiable patterns in service coordination

Secondary Success Criteria

1. **Technical Innovation**:
   - Novel microservices architecture for cryptanalysis
   - Reusable framework applicable to other cryptographic primitives
   - Advanced attention mechanisms for service coordination
2. **Research Contribution**:
   - Clear identification of neural cryptanalysis failure modes
   - Systematic approach to decomposing cryptographic learning
   - Benchmarking framework for future research

Potential Outcomes Spectrum

Best Case Scenario:

- System achieves significant AES encryption accuracy
- Clear breakthrough in neural cryptanalysis
- Publishable research with practical implications

Expected Case Scenario:

- System demonstrates superior performance to monolithic approaches
- Clear understanding of cryptographic composition challenges
- Validated microservices architecture for future research

Worst Case Scenario:

- System confirms AES resistance but with novel insights
- Microservices approach provides better interpretability
- Framework enables more targeted future attacks

Risk Mitigation & Contingency Plans

Technical Risks

Risk: MixColumns service fails to achieve acceptable accuracy
Mitigation: Focus on hierarchical GF arithmetic, implement ensemble approaches, create curriculum learning specifically for MixColumns

Risk: Orchestrator fails to learn meaningful coordination
Mitigation: Start with simpler coordination tasks, implement explicit mathematical coordination as fallback, use interpretability tools to debug attention mechanisms

Risk: Service integration introduces performance degradation
Mitigation: Implement comprehensive service monitoring, create A/B testing framework, maintain individual service benchmarks

Research Risks

Risk: Approach confirms AES resistance without novel insights
Mitigation: Focus on methodology contribution, detailed failure analysis, framework applicability to other cryptographic primitives

Risk: Computational resources insufficient for large-scale training
Mitigation: Implement progressive scaling, optimize for available hardware, consider distributed training approaches

---

Resource Requirements

Computational Resources

- **GPU Requirements**: NVIDIA RTX 3070 minimum, A100/V100 preferred

- **Memory**: 32GB+ system RAM, 16GB+ GPU memory
- **Storage**: 500GB+ for models, datasets, and experiments
- **Training Time**: 8-12 weeks total across all phases

Development Environment

- **Framework**: TensorFlow 2.x with Keras
- **Languages**: Python 3.8+
- **Dependencies**: NumPy, scikit-learn, matplotlib, plotly
- **Infrastructure**: Model versioning, experiment tracking, automated testing

---

Conclusion

This microservices architecture represents a novel approach to neural cryptanalysis that addresses the fundamental composition problem through learned coordination. By leveraging proven individual component success and sophisticated orchestration, we aim to make the first significant progress in neural AES analysis.

The modular approach provides multiple benefits:

- **Interpretability**: Clear understanding of where and why failures occur
- **Reusability**: Components applicable to other cryptographic research
- **Scalability**: Individual services can be improved independently
- **Innovation**: Novel coordination mechanisms applicable beyond cryptography

Whether this approach achieves breakthrough AES results or confirms its resistance, the systematic methodology and architectural innovations will provide valuable contributions to both neural network research and cryptographic analysis.

Quick Start Guide for New Chats

1. **Context**: We're building a neural microservices architecture for AES encryption learning
2. **Current Status**: Individual services proven viable, orchestrator design complete
3. **Next Steps**: Implement Phase 1 - Service Development & Validation
4. **Key Files**: Model persistence pattern established in `caesar_model_test.py`
5. **Architecture**: Master orchestrator coordinates specialized AES component services

6. **Goal**: First neural system to achieve meaningful AES encryption accuracy through composition learning