Enhanced SICA AI & ML Documentation

Version: 1.0
Date: December 2024
Document Type: AI & ML Documentation
Classification: Technical

Table of Contents

AI & ML Architecture Overview

System Architecture

Enhanced SICA implements a sophisticated AI/ML architecture that combines multiple machine learning paradigms to provide comprehensive cybersecurity intelligence. The system is built on a modular architecture that supports:

- **Ensemble Learning**: Multiple models working together for improved accuracy
- **Real-time Processing**: Stream processing for immediate threat detection
- **Adaptive Learning**: Continuous model improvement based on new data
- **Explainable AI**: Transparent decision-making processes

- **Distributed Computing**: Scalable processing across multiple nodes

Core Components

```python
class EnhancedAIMLSystem:
    def __init__(self):
        self.predictive_intelligence = PredictiveIntelligenceEngine()
        self.explainable_ai = ExplainableAIEngine()
        self.confidence_scorer = ConfidenceScorer()
        self.time_series_forecaster = TimeSeriesForecaster()
        self.geopolitical_analyzer = GeopoliticalAnalyzer()
        self.model_manager = ModelManager()

    async def process_intelligence_request(self, request: IntelligenceRequest)
-> IntelligenceResponse:
        """Process intelligence request using AI/ML capabilities"""

        # Route request to appropriate engine
        if request.type == "prediction":
            result = await self.predictive_intelligence.predict(request)
        elif request.type == "explanation":
            result = await self.explainable_ai.explain(request)
        elif request.type == "forecast":
            result = await self.time_series_forecaster.forecast(request)
        elif request.type == "geopolitical":
            result = await self.geopolitical_analyzer.analyze(request)
        else:
            raise ValueError(f"Unknown request type: {request.type}")

        # Add confidence scoring
        confidence = await self.confidence_scorer.score(result)

        return IntelligenceResponse(
            result=result,
            confidence=confidence,
            explanation=await self.explainable_ai.explain_result(result)
        )
```

Data Pipeline Architecture

```python
class AIMLDataPipeline:
    def __init__(self):
        self.data_ingestion = DataIngestionLayer()
        self.preprocessing = PreprocessingLayer()
        self.feature_engineering = FeatureEngineeringLayer()
        self.model_inference = ModelInferenceLayer()
        self.post_processing = PostProcessingLayer()

    async def process_data_stream(self, data_stream: DataStream) ->
ProcessedResults:
        """Process continuous data stream through AI/ML pipeline"""

        processed_results = []

        async for data_batch in data_stream:
            # Ingest and validate data
            ingested_data = await self.data_ingestion.ingest(data_batch)

            # Preprocess data
            preprocessed_data = await self.preprocessing.process(ingested_data)

            # Engineer features
            features = await
self.feature_engineering.extract_features(preprocessed_data)

            # Run model inference
            predictions = await self.model_inference.predict(features)

            # Post-process results
            final_results = await self.post_processing.process(predictions)

            processed_results.extend(final_results)

        return ProcessedResults(processed_results)
```

Predictive Intelligence

Predictive Intelligence Engine

The Predictive Intelligence Engine is the core AI component that anticipates future threats and security events. It combines multiple machine learning techniques to provide accurate predictions with varying time horizons.

Architecture Overview

```
class PredictiveIntelligenceEngine:
    def __init__(self):
        self.ensemble_models = EnsembleModelManager()
        self.feature_extractor = ThreatFeatureExtractor()
        self.prediction_aggregator = PredictionAggregator()
        self.temporal_analyzer = TemporalAnalyzer()

    async def predict_threats(self,
                            context: ThreatContext,
                            time_horizon: str = "24h") -> ThreatPrediction:
        """Predict future threats based on current context"""

        # Extract features from context
        features = await self.feature_extractor.extract(context)

        # Get predictions from ensemble models
        model_predictions = await self.ensemble_models.predict_all(features,
time_horizon)

        # Aggregate predictions
        aggregated_prediction = await
self.prediction_aggregator.aggregate(model_predictions)

        # Analyze temporal patterns
        temporal_analysis = await
self.temporal_analyzer.analyze(aggregated_prediction)

        return ThreatPrediction(
            predicted_threats=aggregated_prediction.threats,
            confidence=aggregated_prediction.confidence,
            time_horizon=time_horizon,
            temporal_patterns=temporal_analysis,
            contributing_factors=aggregated_prediction.factors
        )
```

Ensemble Model Architecture

```python
class EnsembleModelManager:
    def __init__(self):
        self.models = {
            "lstm_threat_predictor": LSTMThreatPredictor(),
            "transformer_analyzer": TransformerThreatAnalyzer(),
            "random_forest_classifier": RandomForestThreatClassifier(),
            "gradient_boosting_predictor": GradientBoostingPredictor(),
            "neural_network_ensemble": NeuralNetworkEnsemble()
        }
        self.model_weights = self.initialize_model_weights()

    async def predict_all(self,
                          features: ThreatFeatures,
                          time_horizon: str) -> List[ModelPrediction]:
        """Get predictions from all ensemble models"""

        predictions = []

        for model_name, model in self.models.items():
            try:
                prediction = await model.predict(features, time_horizon)
                weighted_prediction = self.apply_model_weight(
                    prediction, self.model_weights[model_name]
                )
                predictions.append(ModelPrediction(
                    model_name=model_name,
                    prediction=weighted_prediction,
                    confidence=prediction.confidence,
                    processing_time=prediction.processing_time
                ))
            except Exception as e:
                logger.warning(f"Model {model_name} prediction failed: {e}")

        return predictions

    def initialize_model_weights(self) -> Dict[str, float]:
        """Initialize model weights based on historical performance"""
        return {
            "lstm_threat_predictor": 0.25,
            "transformer_analyzer": 0.30,
            "random_forest_classifier": 0.20,
            "gradient_boosting_predictor": 0.15,
            "neural_network_ensemble": 0.10
        }
```

LSTM Threat Predictor

```
class LSTMThreatPredictor:
    def __init__(self):
        self.model = self.build_lstm_model()
        self.sequence_length = 100
        self.feature_scaler = StandardScaler()

    def build_lstm_model(self) -> tf.keras.Model:
        """Build LSTM model for threat prediction"""

        model = tf.keras.Sequential([
            tf.keras.layers.LSTM(128, return_sequences=True, input_shape=(100,
50)),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.LSTM(64, return_sequences=True),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.LSTM(32),
            tf.keras.layers.Dense(16, activation='relu'),
            tf.keras.layers.Dense(8, activation='relu'),
            tf.keras.layers.Dense(1, activation='sigmoid')
        ])

        model.compile(
            optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy', 'precision', 'recall']
        )

        return model

    async def predict(self,
                      features: ThreatFeatures,
                      time_horizon: str) -> ThreatPrediction:
        """Predict threats using LSTM model"""

        # Prepare sequence data
        sequence_data = await self.prepare_sequence_data(features)

        # Scale features
        scaled_data = self.feature_scaler.transform(sequence_data)

        # Make prediction
        prediction_prob = self.model.predict(scaled_data.reshape(1, -1,
scaled_data.shape[1]))

        # Convert to threat prediction
        threat_probability = float(prediction_prob[0][0])

        # Determine threat level
        if threat_probability > 0.8:
            threat_level = ThreatLevel.CRITICAL
        elif threat_probability > 0.6:
            threat_level = ThreatLevel.HIGH
        elif threat_probability > 0.4:
            threat_level = ThreatLevel.MEDIUM
        else:
            threat_level = ThreatLevel.LOW

        return ThreatPrediction(
            threat_probability=threat_probability,
            threat_level=threat_level,
            confidence=self.calculate_prediction_confidence(prediction_prob),
            model_type="LSTM",
            features_used=features.get_feature_names()
        )
```

Transformer Threat Analyzer

```
class TransformerThreatAnalyzer:
    def __init__(self):
        self.model = self.build_transformer_model()
        self.tokenizer = ThreatTokenizer()
        self.max_sequence_length = 512

    def build_transformer_model(self) -> tf.keras.Model:
        """Build transformer model for threat analysis"""

        # Input layers
        input_ids = tf.keras.layers.Input(shape=(self.max_sequence_length,),
dtype=tf.int32)
        attention_mask = tf.keras.layers.Input(shape=
(self.max_sequence_length,), dtype=tf.int32)

        # Embedding layer
        embeddings = tf.keras.layers.Embedding(
            input_dim=10000,
            output_dim=256,
            mask_zero=True
        )(input_ids)

        # Multi-head attention layers
        attention_output = tf.keras.layers.MultiHeadAttention(
            num_heads=8,
            key_dim=32
        )(embeddings, embeddings, attention_mask=attention_mask)

        # Add & Norm
        attention_output = tf.keras.layers.Add()([embeddings,
attention_output])
        attention_output = tf.keras.layers.LayerNormalization()
(attention_output)

        # Feed forward network
        ffn_output = tf.keras.layers.Dense(512, activation='relu')
(attention_output)
        ffn_output = tf.keras.layers.Dense(256)(ffn_output)

        # Add & Norm
        ffn_output = tf.keras.layers.Add()([attention_output, ffn_output])
        ffn_output = tf.keras.layers.LayerNormalization()(ffn_output)

        # Global average pooling
        pooled_output = tf.keras.layers.GlobalAveragePooling1D()(ffn_output)

        # Classification head
        dropout = tf.keras.layers.Dropout(0.1)(pooled_output)
        outputs = tf.keras.layers.Dense(1, activation='sigmoid')(dropout)

        model = tf.keras.Model(inputs=[input_ids, attention_mask],
outputs=outputs)

        model.compile(
            optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
            loss='binary_crossentropy',
            metrics=['accuracy']
        )
```

```python
        return model

    async def predict(self,
                     features: ThreatFeatures,
                     time_horizon: str) -> ThreatPrediction:
        """Predict threats using transformer model"""

        # Tokenize threat features
        tokenized_data = await self.tokenizer.tokenize(features)

        # Prepare model inputs
        input_ids = tokenized_data['input_ids']
        attention_mask = tokenized_data['attention_mask']

        # Make prediction
        prediction = self.model.predict([input_ids, attention_mask])

        # Process prediction
        threat_score = float(prediction[0][0])

        return ThreatPrediction(
            threat_probability=threat_score,
            threat_level=self.score_to_threat_level(threat_score),
            confidence=self.calculate_transformer_confidence(prediction),
            model_type="Transformer",
            attention_weights=self.extract_attention_weights()
        )
```

Feature Engineering

```
 class ThreatFeatureExtractor:
    def __init__(self):
        self.network_analyzer = NetworkFeatureAnalyzer()
        self.behavioral_analyzer = BehavioralFeatureAnalyzer()
        self.temporal_analyzer = TemporalFeatureAnalyzer()
        self.contextual_analyzer = ContextualFeatureAnalyzer()

    async def extract(self, context: ThreatContext) -> ThreatFeatures:
        """Extract comprehensive features for threat prediction"""

        # Network-based features
        network_features = await
self.network_analyzer.extract(context.network_data)

        # Behavioral features
        behavioral_features = await
self.behavioral_analyzer.extract(context.behavioral_data)

        # Temporal features
        temporal_features = await
self.temporal_analyzer.extract(context.temporal_data)

        # Contextual features
        contextual_features = await
self.contextual_analyzer.extract(context.contextual_data)

        return ThreatFeatures(
            network=network_features,
            behavioral=behavioral_features,
            temporal=temporal_features,
            contextual=contextual_features
        )

class NetworkFeatureAnalyzer:
    def __init__(self):
        self.packet_analyzer = PacketAnalyzer()
        self.flow_analyzer = FlowAnalyzer()
        self.protocol_analyzer = ProtocolAnalyzer()

    async def extract(self, network_data: NetworkData) -> NetworkFeatures:
        """Extract network-based features"""

        features = {}

        # Packet-level features
        packet_features = await
self.packet_analyzer.analyze(network_data.packets)
        features.update({
            "packet_size_mean": packet_features.size_statistics.mean,
            "packet_size_std": packet_features.size_statistics.std,
            "packet_rate": packet_features.rate,
            "packet_size_distribution": packet_features.size_distribution
        })

        # Flow-level features
        flow_features = await self.flow_analyzer.analyze(network_data.flows)
        features.update({
            "flow_duration_mean": flow_features.duration_statistics.mean,
            "flow_bytes_mean": flow_features.bytes_statistics.mean,
            "flow_packets_mean": flow_features.packets_statistics.mean,
            "flow_rate": flow_features.rate
        })

        # Protocol-level features
        protocol_features = await
self.protocol_analyzer.analyze(network_data.protocols)
        features.update({
            "protocol_distribution": protocol_features.distribution,
            "unusual_protocols": protocol_features.unusual_protocols,
            "protocol_anomalies": protocol_features.anomalies
        })

        return NetworkFeatures(features)
```

Explainable AI

Explainable AI Engine

The Explainable AI (XAI) engine provides transparency into AI decision-making processes, enabling security analysts to understand why specific predictions or classifications were made.

Architecture Overview

```
class ExplainableAIEngine:
    def __init__(self):
        self.explanation_generators = {
            "lime": LIMEExplainer(),
            "shap": SHAPExplainer(),
            "attention": AttentionExplainer(),
            "gradient": GradientExplainer(),
            "counterfactual": CounterfactualExplainer()
        }
        self.explanation_aggregator = ExplanationAggregator()

    async def explain_prediction(self,
                               prediction: Prediction,
                               model: AIModel,
                               input_data: InputData) -> Explanation:
        """Generate comprehensive explanation for AI prediction"""

        explanations = {}

        # Generate explanations using different methods
        for method_name, explainer in self.explanation_generators.items():
            try:
                explanation = await explainer.explain(prediction, model,
input_data)
                explanations[method_name] = explanation
            except Exception as e:
                logger.warning(f"Explanation method {method_name} failed: {e}")

        # Aggregate explanations
        aggregated_explanation = await
self.explanation_aggregator.aggregate(explanations)

        return Explanation(
            prediction=prediction,
            method_explanations=explanations,
            aggregated_explanation=aggregated_explanation,
            confidence=self.calculate_explanation_confidence(explanations)
        )
```

LIME Explainer

```python
class LIMEExplainer:
    def __init__(self):
        self.lime_explainer = lime.lime_tabular.LimeTabularExplainer(
            training_data=None,  # Will be set during initialization
            mode='classification',
            feature_names=None,  # Will be set during initialization
            class_names=['benign', 'malicious'],
            discretize_continuous=True
        )

    async def explain(self,
                     prediction: Prediction,
                     model: AIModel,
                     input_data: InputData) -> LIMEExplanation:
        """Generate LIME explanation for prediction"""

        # Prepare data for LIME
        feature_vector = input_data.to_feature_vector()

        # Generate explanation
        explanation = self.lime_explainer.explain_instance(
            feature_vector,
            model.predict_proba,
            num_features=10,
            num_samples=1000
        )

        # Extract feature importance
        feature_importance = {}
        for feature_idx, importance in explanation.as_list():
            feature_name = input_data.get_feature_name(feature_idx)
            feature_importance[feature_name] = importance

        # Generate textual explanation
        textual_explanation = self.generate_textual_explanation(
            feature_importance, prediction
        )

        return LIMEExplanation(
            feature_importance=feature_importance,
            textual_explanation=textual_explanation,
            confidence=explanation.score,
            local_prediction=explanation.local_pred
        )

    def generate_textual_explanation(self,
                                   feature_importance: Dict[str, float],
                                   prediction: Prediction) -> str:
        """Generate human-readable explanation"""

        # Sort features by importance
        sorted_features = sorted(
            feature_importance.items(),
            key=lambda x: abs(x[1]),
            reverse=True
        )

        explanation_parts = []
        if prediction.is_malicious():
            explanation_parts.append("This prediction indicates a potential threat because:")
        else:
            explanation_parts.append("This prediction indicates benign activity because:")

        for feature_name, importance in sorted_features[:5]:
            if importance > 0:
                explanation_parts.append(
                    f"- {feature_name} strongly supports this classification (importance: {importance:.3f})"
                )
            else:
                explanation_parts.append(
                    f"- {feature_name} contradicts this classification (importance: {importance:.3f})"
                )

        return "\n".join(explanation_parts)
```

SHAP Explainer

```
class SHAPExplainer:
    def __init__(self):
        self.explainer = None  # Will be initialized based on model type

    async def explain(self,
                     prediction: Prediction,
                     model: AIModel,
                     input_data: InputData) -> SHAPExplanation:
        """Generate SHAP explanation for prediction"""

        # Initialize explainer based on model type
        if isinstance(model, TreeBasedModel):
            self.explainer = shap.TreeExplainer(model.model)
        elif isinstance(model, NeuralNetworkModel):
            self.explainer = shap.DeepExplainer(model.model,
input_data.background_data)
        else:
            self.explainer = shap.KernelExplainer(model.predict,
input_data.background_data)

        # Calculate SHAP values
        shap_values =
self.explainer.shap_values(input_data.to_feature_vector())

        # Process SHAP values
        if isinstance(shap_values, list):
            # Multi-class case
            shap_values = shap_values[1]  # Use positive class

        # Create feature importance mapping
        feature_importance = {}
        for i, shap_value in enumerate(shap_values):
            feature_name = input_data.get_feature_name(i)
            feature_importance[feature_name] = float(shap_value)

        # Generate explanation summary
        explanation_summary = self.generate_shap_summary(
            feature_importance, prediction
        )

        return SHAPExplanation(
            shap_values=shap_values.tolist(),
            feature_importance=feature_importance,
            explanation_summary=explanation_summary,
            expected_value=float(self.explainer.expected_value)
        )

    def generate_shap_summary(self,
                              feature_importance: Dict[str, float],
                              prediction: Prediction) -> str:
        """Generate SHAP-based explanation summary"""

        # Find most influential features
        positive_features = {k: v for k, v in feature_importance.items() if v >
0}
        negative_features = {k: v for k, v in feature_importance.items() if v <
0}

        # Sort by absolute importance
        top_positive = sorted(positive_features.items(), key=lambda x: x[1],
reverse=True)[:3]
        top_negative = sorted(negative_features.items(), key=lambda x:
abs(x[1]), reverse=True)[:3]

        summary_parts = []
        summary_parts.append(f"Prediction: {prediction.class_name} (confidence:
{prediction.confidence:.3f})")

        if top_positive:
            summary_parts.append("\nFeatures supporting this prediction:")
            for feature, value in top_positive:
                summary_parts.append(f"  • {feature}: +{value:.3f}")

        if top_negative:
            summary_parts.append("\nFeatures opposing this prediction:")
            for feature, value in top_negative:
                summary_parts.append(f"  • {feature}: {value:.3f}")

        return "\n".join(summary_parts)
```

Attention Explainer

```
class AttentionExplainer:
    def __init__(self):
        self.attention_extractor = AttentionExtractor()
        self.attention_visualizer = AttentionVisualizer()

    async def explain(self,
                     prediction: Prediction,
                     model: AIModel,
                     input_data: InputData) -> AttentionExplanation:
        """Generate attention-based explanation for transformer models"""

        if not isinstance(model, TransformerModel):
            raise ValueError("Attention explanation only available for
transformer models")

        # Extract attention weights
        attention_weights = await self.attention_extractor.extract(
            model, input_data
        )

        # Analyze attention patterns
        attention_analysis = await self.analyze_attention_patterns(
            attention_weights, input_data
        )

        # Generate attention visualization
        attention_visualization = await self.attention_visualizer.visualize(
            attention_weights, input_data
        )

        # Create textual explanation
        textual_explanation = self.generate_attention_explanation(
            attention_analysis, prediction
        )

        return AttentionExplanation(
            attention_weights=attention_weights,
            attention_analysis=attention_analysis,
            visualization=attention_visualization,
            textual_explanation=textual_explanation
        )

    async def analyze_attention_patterns(self,
                                       attention_weights: AttentionWeights,
                                       input_data: InputData) ->
AttentionAnalysis:
        """Analyze attention patterns to understand model focus"""

        # Find tokens with highest attention
        high_attention_tokens = []
        for layer_idx, layer_weights in enumerate(attention_weights.layers):
            for head_idx, head_weights in enumerate(layer_weights.heads):
                top_indices = np.argsort(head_weights)[-5:]  # Top 5 tokens
                for token_idx in top_indices:
                    token = input_data.get_token(token_idx)
                    attention_score = head_weights[token_idx]
                    high_attention_tokens.append(AttentionToken(
                        token=token,
                        layer=layer_idx,
                        head=head_idx,
                        attention_score=attention_score
                    ))

        # Analyze attention distribution
        attention_distribution =
self.calculate_attention_distribution(attention_weights)

        # Identify attention patterns
        attention_patterns =
self.identify_attention_patterns(attention_weights)

        return AttentionAnalysis(
            high_attention_tokens=high_attention_tokens,
            attention_distribution=attention_distribution,
            attention_patterns=attention_patterns
        )
```

Decision Tree Visualization

```python
class DecisionTreeExplainer:
    def __init__(self):
        self.tree_visualizer = TreeVisualizer()
        self.path_extractor = DecisionPathExtractor()

    async def explain(self,
                      prediction: Prediction,
                      model: AIModel,
                      input_data: InputData) -> DecisionTreeExplanation:
        """Generate decision tree explanation"""

        if not isinstance(model, TreeBasedModel):
            raise ValueError("Decision tree explanation only available for
tree-based models")

        # Extract decision path
        decision_path = await self.path_extractor.extract_path(
            model, input_data
        )

        # Generate tree visualization
        tree_visualization = await self.tree_visualizer.visualize(
            model.tree, decision_path
        )

        # Create rule-based explanation
        rule_explanation = self.generate_rule_explanation(decision_path)

        return DecisionTreeExplanation(
            decision_path=decision_path,
            tree_visualization=tree_visualization,
            rule_explanation=rule_explanation,
            feature_importance=model.feature_importances_
        )

    def generate_rule_explanation(self, decision_path: DecisionPath) -> str:
        """Generate rule-based explanation from decision path"""

        rules = []

        for node in decision_path.nodes:
            if node.is_decision_node():
                feature_name = node.feature_name
                threshold = node.threshold
                direction = "≤" if node.went_left else ">"

                rules.append(f"{feature_name} {direction} {threshold}")

        rule_text = " AND ".join(rules)
        final_prediction = decision_path.leaf_node.prediction

        return f"Prediction: {final_prediction}\nRule: {rule_text}"
```

Confidence Scoring

Confidence Scoring System

The confidence scoring system provides quantitative measures of prediction reliability, helping security analysts understand the trustworthiness of AI-generated insights.

Architecture Overview

```python
class ConfidenceScorer:
    def __init__(self):
        self.uncertainty_estimators = {
            "epistemic": EpistemicUncertaintyEstimator(),
            "aleatoric": AleatoricUncertaintyEstimator(),
            "ensemble": EnsembleUncertaintyEstimator(),
            "calibration": CalibrationBasedScorer()
        }
        self.confidence_aggregator = ConfidenceAggregator()

    async def score(self, prediction: Prediction) -> ConfidenceScore:
        """Calculate comprehensive confidence score for prediction"""

        uncertainty_scores = {}

        # Calculate different types of uncertainty
        for uncertainty_type, estimator in self.uncertainty_estimators.items():
            try:
                uncertainty = await estimator.estimate(prediction)
                uncertainty_scores[uncertainty_type] = uncertainty
            except Exception as e:
                logger.warning(f"Uncertainty estimation {uncertainty_type}
failed: {e}")

        # Aggregate uncertainties into confidence
        confidence = await
self.confidence_aggregator.aggregate(uncertainty_scores)

        return ConfidenceScore(
            overall_confidence=confidence.overall,
            uncertainty_breakdown=uncertainty_scores,
            confidence_level=self.categorize_confidence(confidence.overall),
            reliability_indicators=confidence.reliability_indicators
        )
```

Epistemic Uncertainty Estimation

```python
class EpistemicUncertaintyEstimator:
    """Estimates uncertainty due to model limitations (lack of knowledge)"""

    def __init__(self):
        self.monte_carlo_samples = 100
        self.dropout_rate = 0.1

    async def estimate(self, prediction: Prediction) -> EpistemicUncertainty:
        """Estimate epistemic uncertainty using Monte Carlo dropout"""

        model = prediction.model
        input_data = prediction.input_data

        # Enable dropout during inference
        model.enable_dropout()

        # Collect multiple predictions
        predictions = []
        for _ in range(self.monte_carlo_samples):
            sample_prediction = await model.predict(input_data)
            predictions.append(sample_prediction.probability)

        # Disable dropout
        model.disable_dropout()

        # Calculate uncertainty metrics
        mean_prediction = np.mean(predictions)
        variance = np.var(predictions)
        entropy = self.calculate_entropy(predictions)

        return EpistemicUncertainty(
            variance=variance,
            entropy=entropy,
            mean_prediction=mean_prediction,
            prediction_samples=predictions
        )

    def calculate_entropy(self, predictions: List[float]) -> float:
        """Calculate entropy of prediction distribution"""

        # Convert to probability distribution
        probs = np.array(predictions)
        probs = probs / np.sum(probs)  # Normalize

        # Calculate entropy
        entropy = -np.sum(probs * np.log(probs + 1e-10))

        return float(entropy)
```

Aleatoric Uncertainty Estimation

```python
class AleatoricUncertaintyEstimator:
    """Estimates uncertainty inherent in the data"""

    def __init__(self):
        self.noise_estimator = NoiseEstimator()
        self.data_quality_assessor = DataQualityAssessor()

    async def estimate(self, prediction: Prediction) -> AleatoricUncertainty:
        """Estimate aleatoric uncertainty"""

        input_data = prediction.input_data

        # Assess data quality
        data_quality = await self.data_quality_assessor.assess(input_data)

        # Estimate noise level
        noise_level = await self.noise_estimator.estimate(input_data)

        # Calculate uncertainty based on data characteristics
        feature_uncertainty = self.calculate_feature_uncertainty(input_data)

        # Combine uncertainty sources
        total_uncertainty = self.combine_uncertainties(
            data_quality.uncertainty,
            noise_level,
            feature_uncertainty
        )

        return AleatoricUncertainty(
            data_quality_uncertainty=data_quality.uncertainty,
            noise_uncertainty=noise_level,
            feature_uncertainty=feature_uncertainty,
            total_uncertainty=total_uncertainty
        )

    def calculate_feature_uncertainty(self, input_data: InputData) -> float:
        """Calculate uncertainty based on feature characteristics"""

        features = input_data.features
        uncertainties = []

        for feature_name, feature_value in features.items():
            # Check for missing values
            if feature_value is None or np.isnan(feature_value):
                uncertainties.append(1.0)  # Maximum uncertainty
                continue

            # Check for outliers
            if self.is_outlier(feature_value, feature_name):
                uncertainties.append(0.8)  # High uncertainty
                continue

            # Check feature stability
            stability = self.assess_feature_stability(feature_name)
            uncertainties.append(1.0 - stability)

        return np.mean(uncertainties) if uncertainties else 0.0
```

Ensemble Uncertainty Estimation

```
class EnsembleUncertaintyEstimator:
    """Estimates uncertainty based on ensemble model disagreement"""

    def __init__(self):
        self.disagreement_calculator = DisagreementCalculator()

    async def estimate(self, prediction: Prediction) -> EnsembleUncertainty:
        """Estimate uncertainty from ensemble model disagreement"""

        if not isinstance(prediction, EnsemblePrediction):
            raise ValueError("Ensemble uncertainty requires ensemble
prediction")

        model_predictions = prediction.model_predictions

        # Calculate disagreement metrics
        disagreement_metrics = await self.disagreement_calculator.calculate(
            model_predictions
        )

        # Calculate prediction variance
        prediction_variance =
self.calculate_prediction_variance(model_predictions)

        # Calculate consensus strength
        consensus_strength =
self.calculate_consensus_strength(model_predictions)

        return EnsembleUncertainty(
            disagreement_score=disagreement_metrics.overall_disagreement,
            prediction_variance=prediction_variance,
            consensus_strength=consensus_strength,
            model_disagreements=disagreement_metrics.pairwise_disagreements
        )

    def calculate_prediction_variance(self,
                                      model_predictions: List[ModelPrediction]) ->
float:
        """Calculate variance in model predictions"""

        probabilities = [pred.probability for pred in model_predictions]
        return float(np.var(probabilities))

    def calculate_consensus_strength(self,
                                     model_predictions: List[ModelPrediction]) ->
float:
        """Calculate strength of consensus among models"""

        # Count predictions for each class
        class_counts = {}
        for pred in model_predictions:
            class_name = pred.predicted_class
            class_counts[class_name] = class_counts.get(class_name, 0) + 1

        # Calculate consensus strength
        total_models = len(model_predictions)
        max_agreement = max(class_counts.values())
        return max_agreement / total_models
```

Calibration-Based Scoring

```python
 class CalibrationBasedScorer:
     """Scores confidence based on model calibration"""

     def __init__(self):
         self.calibration_curve = CalibrationCurve()
         self.reliability_diagram = ReliabilityDiagram()

     async def estimate(self, prediction: Prediction) -> CalibrationScore:
         """Estimate confidence based on model calibration"""

         model = prediction.model
         predicted_probability = prediction.probability

         # Get calibration curve for model
         calibration_data = await
self.calibration_curve.get_calibration_data(model)

         # Find calibrated probability
         calibrated_probability = self.interpolate_calibrated_probability(
             predicted_probability, calibration_data
         )

         # Calculate calibration error
         calibration_error = abs(predicted_probability - calibrated_probability)

         # Calculate reliability score
         reliability_score = self.calculate_reliability_score(
             predicted_probability, calibration_data
         )

         return CalibrationScore(
             original_probability=predicted_probability,
             calibrated_probability=calibrated_probability,
             calibration_error=calibration_error,
             reliability_score=reliability_score
         )

     def interpolate_calibrated_probability(self,
                                          predicted_prob: float,
                                          calibration_data: CalibrationData) ->
float:
         """Interpolate calibrated probability from calibration curve"""

         # Find nearest calibration points
         lower_idx = 0
         upper_idx = len(calibration_data.points) - 1

         for i, point in enumerate(calibration_data.points):
             if point.predicted_probability <= predicted_prob:
                 lower_idx = i
             else:
                 upper_idx = i
                 break

         # Interpolate between points
         if lower_idx == upper_idx:
             return calibration_data.points[lower_idx].actual_probability

         lower_point = calibration_data.points[lower_idx]
```

```
        upper_point = calibration_data.points[upper_idx]

        # Linear interpolation
        weight = ((predicted_prob - lower_point.predicted_probability) /
                 (upper_point.predicted_probability -
lower_point.predicted_probability))

        calibrated_prob = (lower_point.actual_probability * (1 - weight) +
                          upper_point.actual_probability * weight)

        return calibrated_prob
```

Confidence Level Categorization

```
class ConfidenceCategorizer:
    def __init__(self):
        self.confidence_thresholds = {
            "very_high": 0.9,
            "high": 0.75,
            "medium": 0.6,
            "low": 0.4,
            "very_low": 0.0
        }

    def categorize_confidence(self, confidence_score: float) ->
ConfidenceLevel:
        """Categorize numerical confidence score into levels"""

        if confidence_score >= self.confidence_thresholds["very_high"]:
            return ConfidenceLevel(
                level="very_high",
                description="Very High Confidence - Prediction is highly
reliable",
                color="green",
                action_recommendation="Act on prediction with high confidence"
            )
        elif confidence_score >= self.confidence_thresholds["high"]:
            return ConfidenceLevel(
                level="high",
                description="High Confidence - Prediction is reliable",
                color="light_green",
                action_recommendation="Act on prediction with reasonable
confidence"
            )
        elif confidence_score >= self.confidence_thresholds["medium"]:
            return ConfidenceLevel(
                level="medium",
                description="Medium Confidence - Prediction should be
verified",
                color="yellow",
                action_recommendation="Verify prediction before acting"
            )
        elif confidence_score >= self.confidence_thresholds["low"]:
            return ConfidenceLevel(
                level="low",
                description="Low Confidence - Prediction is uncertain",
                color="orange",
                action_recommendation="Seek additional evidence before acting"
            )
        else:
            return ConfidenceLevel(
                level="very_low",
                description="Very Low Confidence - Prediction is highly
uncertain",
                color="red",
                action_recommendation="Do not act on prediction without
verification"
            )
```

Time Series Forecasting

Time Series Forecasting Engine

The Time Series Forecasting Engine predicts future security events and trends based on historical data patterns, enabling proactive threat mitigation.

Architecture Overview

```python
class TimeSeriesForecaster:
    def __init__(self):
        self.forecasting_models = {
            "lstm": LSTMForecaster(),
            "arima": ARIMAForecaster(),
            "prophet": ProphetForecaster(),
            "transformer": TransformerForecaster(),
            "ensemble": EnsembleForecaster()
        }
        self.data_preprocessor = TimeSeriesPreprocessor()
        self.forecast_evaluator = ForecastEvaluator()

    async def forecast(self,
                       time_series_data: TimeSeriesData,
                       forecast_horizon: int,
                       model_type: str = "ensemble") -> ForecastResult:
        """Generate time series forecast"""

        # Preprocess data
        preprocessed_data = await
self.data_preprocessor.preprocess(time_series_data)

        # Select forecasting model
        forecaster = self.forecasting_models[model_type]

        # Generate forecast
        forecast = await forecaster.forecast(preprocessed_data,
forecast_horizon)

        # Evaluate forecast quality
        evaluation = await self.forecast_evaluator.evaluate(forecast)

        return ForecastResult(
            forecast=forecast,
            confidence_intervals=forecast.confidence_intervals,
            evaluation_metrics=evaluation,
            model_used=model_type
        )
```

LSTM Time Series Forecaster

```
class LSTMForecaster:
    def __init__(self):
        self.model = None
        self.sequence_length = 60
        self.features_count = 1

    def build_model(self, input_shape: Tuple[int, int]) -> tf.keras.Model:
        """Build LSTM model for time series forecasting"""

        model = tf.keras.Sequential([
            tf.keras.layers.LSTM(50, return_sequences=True,
input_shape=input_shape),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.LSTM(50, return_sequences=True),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.LSTM(50),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.Dense(25),
            tf.keras.layers.Dense(1)
        ])

        model.compile(optimizer='adam', loss='mean_squared_error')
        return model

    async def forecast(self,
                      data: TimeSeriesData,
                      horizon: int) -> TimeSeriesForecast:
        """Generate LSTM-based forecast"""

        # Prepare training data
        X_train, y_train = self.prepare_training_data(data)

        # Build and train model
        self.model = self.build_model((X_train.shape[1], X_train.shape[2]))
        self.model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0)

        # Generate forecast
        forecast_values = []
        current_sequence = data.values[-self.sequence_length:].reshape(1,
self.sequence_length, 1)

        for _ in range(horizon):
            # Predict next value
            next_value = self.model.predict(current_sequence, verbose=0)[0, 0]
            forecast_values.append(next_value)

            # Update sequence for next prediction
            current_sequence = np.roll(current_sequence, -1, axis=1)
            current_sequence[0, -1, 0] = next_value

        # Calculate confidence intervals
        confidence_intervals = self.calculate_confidence_intervals(
            forecast_values, data
        )

        return TimeSeriesForecast(
            values=forecast_values,
            timestamps=self.generate_future_timestamps(data.timestamps,
horizon),
            confidence_intervals=confidence_intervals,
            model_type="LSTM"
        )

    def prepare_training_data(self, data: TimeSeriesData) -> Tuple[np.ndarray,
np.ndarray]:
        """Prepare training data for LSTM"""

        values = data.values
        X, y = [], []

        for i in range(self.sequence_length, len(values)):
            X.append(values[i-self.sequence_length:i])
            y.append(values[i])

        return np.array(X), np.array(y)
```

ARIMA Forecaster

```
class ARIMAForecaster:
    def __init__(self):
        self.model = None
        self.auto_arima = True

    async def forecast(self,
                      data: TimeSeriesData,
                      horizon: int) -> TimeSeriesForecast:
        """Generate ARIMA-based forecast"""

        # Determine ARIMA parameters
        if self.auto_arima:
            arima_params = await self.auto_arima_selection(data)
        else:
            arima_params = (1, 1, 1)  # Default parameters

        # Fit ARIMA model
        self.model = ARIMA(data.values, order=arima_params)
        fitted_model = self.model.fit()

        # Generate forecast
        forecast_result = fitted_model.forecast(steps=horizon, alpha=0.05)

        # Extract forecast values and confidence intervals
        forecast_values = forecast_result[0].tolist()
        confidence_intervals = [
            (lower, upper) for lower, upper in zip(
                forecast_result[1][:, 0], forecast_result[1][:, 1]
            )
        ]

        return TimeSeriesForecast(
            values=forecast_values,
            timestamps=self.generate_future_timestamps(data.timestamps,
horizon),
            confidence_intervals=confidence_intervals,
            model_type="ARIMA",
            model_parameters=arima_params
        )

    async def auto_arima_selection(self, data: TimeSeriesData) -> Tuple[int,
int, int]:
        """Automatically select optimal ARIMA parameters"""

        best_aic = float('inf')
        best_params = (1, 1, 1)

        # Grid search for optimal parameters
        for p in range(0, 3):
            for d in range(0, 2):
                for q in range(0, 3):
                    try:
                        model = ARIMA(data.values, order=(p, d, q))
                        fitted_model = model.fit()

                        if fitted_model.aic < best_aic:
                            best_aic = fitted_model.aic
                            best_params = (p, d, q)
                    except Exception:
                        continue

        return best_params
```

Prophet Forecaster

```
class ProphetForecaster:
    def __init__(self):
        self.model = Prophet()

    async def forecast(self,
                       data: TimeSeriesData,
                       horizon: int) -> TimeSeriesForecast:
        """Generate Prophet-based forecast"""

        # Prepare data for Prophet
        prophet_data = pd.DataFrame({
            'ds': data.timestamps,
            'y': data.values
        })

        # Fit Prophet model
        self.model.fit(prophet_data)

        # Create future dataframe
        future = self.model.make_future_dataframe(periods=horizon, freq='H')

        # Generate forecast
        forecast = self.model.predict(future)

        # Extract forecast values for future periods
        forecast_values = forecast['yhat'][-horizon:].tolist()

        # Extract confidence intervals
        confidence_intervals = [
            (lower, upper) for lower, upper in zip(
                forecast['yhat_lower'][-horizon:],
                forecast['yhat_upper'][-horizon:]
            )
        ]

        return TimeSeriesForecast(
            values=forecast_values,
            timestamps=forecast['ds'][-horizon:].tolist(),
            confidence_intervals=confidence_intervals,
            model_type="Prophet",

seasonality_components=self.extract_seasonality_components(forecast)
        )

    def extract_seasonality_components(self, forecast: pd.DataFrame) ->
Dict[str, List[float]]:
        """Extract seasonality components from Prophet forecast"""

        components = {}

        if 'trend' in forecast.columns:
            components['trend'] = forecast['trend'].tolist()

        if 'weekly' in forecast.columns:
            components['weekly'] = forecast['weekly'].tolist()

        if 'daily' in forecast.columns:
            components['daily'] = forecast['daily'].tolist()
        return components
```

Ensemble Forecaster

```
class EnsembleForecaster:
    def __init__(self):
        self.base_forecasters = {
            "lstm": LSTMForecaster(),
            "arima": ARIMAForecaster(),
            "prophet": ProphetForecaster()
        }
        self.ensemble_weights = {
            "lstm": 0.4,
            "arima": 0.3,
            "prophet": 0.3
        }

    async def forecast(self,
                       data: TimeSeriesData,
                       horizon: int) -> TimeSeriesForecast:
        """Generate ensemble forecast"""

        # Get forecasts from all base models
        base_forecasts = {}
        for name, forecaster in self.base_forecasters.items():
            try:
                forecast = await forecaster.forecast(data, horizon)
                base_forecasts[name] = forecast
            except Exception as e:
                logger.warning(f"Forecaster {name} failed: {e}")

        # Combine forecasts using weighted average
        ensemble_values = self.combine_forecasts(base_forecasts)

        # Calculate ensemble confidence intervals
        ensemble_confidence = self.combine_confidence_intervals(base_forecasts)

        return TimeSeriesForecast(
            values=ensemble_values,
            timestamps=self.generate_future_timestamps(data.timestamps,
horizon),
            confidence_intervals=ensemble_confidence,
            model_type="Ensemble",
            base_forecasts=base_forecasts
        )

    def combine_forecasts(self, base_forecasts: Dict[str, TimeSeriesForecast])
-> List[float]:
        """Combine forecasts using weighted average"""

        if not base_forecasts:
            return []

        # Get forecast length
        forecast_length = len(list(base_forecasts.values())[0].values)

        # Initialize ensemble forecast
        ensemble_values = [0.0] * forecast_length

        # Weighted combination
        total_weight = 0.0
        for name, forecast in base_forecasts.items():
            weight = self.ensemble_weights.get(name, 1.0)
            total_weight += weight

            for i, value in enumerate(forecast.values):
                ensemble_values[i] += weight * value

        # Normalize by total weight
        ensemble_values = [value / total_weight for value in ensemble_values]

        return ensemble_values
```

Geopolitical Analysis

Geopolitical Analysis Engine

The Geopolitical Analysis Engine correlates cybersecurity threats with geopolitical events and trends, providing context for threat attribution and prediction.

Architecture Overview

```python
class GeopoliticalAnalyzer:
    def __init__(self):
        self.event_collector = GeopoliticalEventCollector()
        self.threat_correlator = ThreatCorrelator()
        self.attribution_engine = AttributionEngine()
        self.trend_analyzer = GeopoliticalTrendAnalyzer()

    async def analyze_geopolitical_context(self,
                                           threat_data: ThreatData) ->
GeopoliticalAnalysis:
        """Analyze geopolitical context of cybersecurity threats"""

        # Collect recent geopolitical events
        recent_events = await self.event_collector.collect_recent_events()

        # Correlate threats with geopolitical events
        correlations = await self.threat_correlator.correlate(threat_data,
recent_events)

        # Perform threat attribution
        attribution = await self.attribution_engine.attribute_threat(
            threat_data, correlations
        )

        # Analyze geopolitical trends
        trends = await self.trend_analyzer.analyze_trends(recent_events,
threat_data)

        return GeopoliticalAnalysis(
            threat_data=threat_data,
            correlated_events=correlations,
            attribution=attribution,
            geopolitical_trends=trends,
            risk_assessment=self.assess_geopolitical_risk(correlations, trends)
        )
```

Geopolitical Event Collection

```
class GeopoliticalEventCollector:
    def __init__(self):
        self.news_sources = [
            "reuters", "bbc", "ap", "cnn", "guardian"
        ]
        self.event_classifiers = {
            "conflict": ConflictEventClassifier(),
            "diplomatic": DiplomaticEventClassifier(),
            "economic": EconomicEventClassifier(),
            "political": PoliticalEventClassifier()
        }

    async def collect_recent_events(self,
                                     days_back: int = 30) ->
List[GeopoliticalEvent]:
        """Collect recent geopolitical events from multiple sources"""

        all_events = []

        # Collect from each news source
        for source in self.news_sources:
            try:
                source_events = await self.collect_from_source(source,
days_back)
                all_events.extend(source_events)
            except Exception as e:
                logger.warning(f"Failed to collect from {source}: {e}")

        # Deduplicate events
        unique_events = self.deduplicate_events(all_events)

        # Classify events
        classified_events = await self.classify_events(unique_events)

        return classified_events

    async def collect_from_source(self,
                                   source: str,
                                   days_back: int) -> List[GeopoliticalEvent]:
        """Collect events from specific news source"""

        # This would integrate with news APIs
        # For demonstration, we'll simulate event collection

        events = []

        # Simulate different types of events
        event_templates = [
            {
                "type": "conflict",
                "description": "Military tensions escalate between {country1}
and {country2}",
                "severity": "high"
            },
            {
                "type": "diplomatic",
                "description": "Trade negotiations between {country1} and
{country2}",
                "severity": "medium"
            },
            {
                "type": "economic",
                "description": "Economic sanctions imposed on {country}",
                "severity": "high"
            }
        ]

        # Generate sample events
        for template in event_templates:
            event = GeopoliticalEvent(
                source=source,
                timestamp=datetime.now() - timedelta(days=random.randint(1,
days_back)),
                event_type=template["type"],
                description=template["description"],
                severity=template["severity"],

countries_involved=self.extract_countries(template["description"]),
                confidence=0.8
            )
            events.append(event)

        return events

    async def classify_events(self,
                            events: List[GeopoliticalEvent]) ->
List[GeopoliticalEvent]:
        """Classify events using specialized classifiers"""

        classified_events = []

        for event in events:
            # Determine event type if not already classified
            if not event.event_type:
                event.event_type = await self.determine_event_type(event)

            # Use specialized classifier
            classifier = self.event_classifiers.get(event.event_type)
            if classifier:
                enhanced_event = await classifier.classify(event)
                classified_events.append(enhanced_event)
            else:
                classified_events.append(event)

        return classified_events
```

Threat Correlation

```
 class ThreatCorrelator:
    def __init__(self):
        self.correlation_algorithms = {
            "temporal": TemporalCorrelator(),
            "geographic": GeographicCorrelator(),
            "actor": ActorCorrelator(),
            "technique": TechniqueCorrelator()
        }

    async def correlate(self,
                       threat_data: ThreatData,
                       geopolitical_events: List[GeopoliticalEvent]) ->
List[ThreatCorrelation]:
        """Correlate threats with geopolitical events"""

        correlations = []

        for event in geopolitical_events:
            # Calculate correlations using different algorithms
            correlation_scores = {}

            for algorithm_name, correlator in
self.correlation_algorithms.items():
                try:
                    score = await correlator.correlate(threat_data, event)
                    correlation_scores[algorithm_name] = score
                except Exception as e:
                    logger.warning(f"Correlation algorithm {algorithm_name}
failed: {e}")

            # Calculate overall correlation score
            overall_score =
self.calculate_overall_correlation(correlation_scores)

            if overall_score > 0.3:  # Threshold for significant correlation
                correlation = ThreatCorrelation(
                    threat_data=threat_data,
                    geopolitical_event=event,
                    correlation_score=overall_score,
                    correlation_breakdown=correlation_scores,

correlation_explanation=self.generate_correlation_explanation(
                        threat_data, event, correlation_scores
                    )
                )
                correlations.append(correlation)

        # Sort by correlation strength
        correlations.sort(key=lambda c: c.correlation_score, reverse=True)

        return correlations

class TemporalCorrelator:
    def __init__(self):
        self.time_window = timedelta(days=7)  # Consider events within 7 days

    async def correlate(self,
                       threat_data: ThreatData,
                       event: GeopoliticalEvent) -> float:
```

```python
        """Calculate temporal correlation between threat and event"""

        threat_time = threat_data.timestamp
        event_time = event.timestamp

        # Calculate time difference
        time_diff = abs(threat_time - event_time)

        if time_diff > self.time_window:
            return 0.0

        # Calculate correlation based on temporal proximity
        # Closer in time = higher correlation
        correlation = 1.0 - (time_diff.total_seconds() /
self.time_window.total_seconds())

        return max(0.0, correlation)

class GeographicCorrelator:
    def __init__(self):
        self.country_ip_database = CountryIPDatabase()

    async def correlate(self,
                       threat_data: ThreatData,
                       event: GeopoliticalEvent) -> float:
        """Calculate geographic correlation between threat and event"""

        # Get threat origin country
        threat_country = await self.get_threat_origin_country(threat_data)

        # Get event countries
        event_countries = event.countries_involved

        if not threat_country or not event_countries:
            return 0.0

        # Check for direct country match
        if threat_country in event_countries:
            return 1.0

        # Check for regional proximity
        regional_correlation = await self.calculate_regional_correlation(
            threat_country, event_countries
        )

        return regional_correlation
```

Attribution Engine

```
class AttributionEngine:
    def __init__(self):
        self.actor_database = ThreatActorDatabase()
        self.technique_analyzer = TechniqueAnalyzer()
        self.infrastructure_analyzer = InfrastructureAnalyzer()

    async def attribute_threat(self,
                               threat_data: ThreatData,
                               correlations: List[ThreatCorrelation]) ->
ThreatAttribution:
        """Perform threat attribution based on various indicators"""

        # Analyze threat techniques
        technique_attribution = await
self.technique_analyzer.analyze(threat_data)

        # Analyze infrastructure
        infrastructure_attribution = await
self.infrastructure_analyzer.analyze(threat_data)

        # Analyze geopolitical context
        geopolitical_attribution =
self.analyze_geopolitical_context(correlations)

        # Combine attribution indicators
        combined_attribution = self.combine_attribution_indicators(
            technique_attribution,
            infrastructure_attribution,
            geopolitical_attribution
        )

        return ThreatAttribution(
            attributed_actors=combined_attribution.actors,
            confidence=combined_attribution.confidence,
            attribution_reasoning=combined_attribution.reasoning,
            supporting_evidence=combined_attribution.evidence
        )

    def analyze_geopolitical_context(self,
                                     correlations: List[ThreatCorrelation]) ->
GeopoliticalAttribution:
        """Analyze geopolitical context for attribution"""

        if not correlations:
            return GeopoliticalAttribution(confidence=0.0)

        # Find strongest correlation
        strongest_correlation = max(correlations, key=lambda c:
c.correlation_score)

        # Extract potential actors from geopolitical context
        potential_actors = self.extract_potential_actors(strongest_correlation)

        # Calculate attribution confidence
        confidence = self.calculate_geopolitical_confidence(correlations)

        return GeopoliticalAttribution(
            potential_actors=potential_actors,
            confidence=confidence,
            supporting_events=[c.geopolitical_event for c in correlations[:3]]
        )
```

Trend Analysis

```
 class GeopoliticalTrendAnalyzer:
     def __init__(self):
         self.trend_detectors = {
             "escalation": EscalationTrendDetector(),
             "alliance": AllianceTrendDetector(),
             "economic": EconomicTrendDetector(),
             "cyber": CyberTrendDetector()
         }

     async def analyze_trends(self,
                                    events: List[GeopoliticalEvent],
                                    threat_data: ThreatData) -> GeopoliticalTrends:
         """Analyze geopolitical trends relevant to cybersecurity"""

         detected_trends = {}

         # Detect different types of trends
         for trend_type, detector in self.trend_detectors.items():
             try:
                 trend = await detector.detect_trend(events, threat_data)
                 if trend.significance > 0.5:
                     detected_trends[trend_type] = trend
             except Exception as e:
                 logger.warning(f"Trend detector {trend_type} failed: {e}")

         # Analyze trend implications
         trend_implications = await
self.analyze_trend_implications(detected_trends)

         return GeopoliticalTrends(
             detected_trends=detected_trends,
             trend_implications=trend_implications,
             risk_level=self.calculate_overall_risk_level(detected_trends)
         )

class EscalationTrendDetector:
    def __init__(self):
        self.escalation_indicators = [
            "military", "conflict", "tension", "dispute", "sanctions"
        ]

    async def detect_trend(self,
                                  events: List[GeopoliticalEvent],
                                  threat_data: ThreatData) -> Trend:
        """Detect escalation trends in geopolitical events"""

        # Filter events related to escalation
        escalation_events = [
            event for event in events
            if any(indicator in event.description.lower()
                   for indicator in self.escalation_indicators)
        ]

        if len(escalation_events) < 2:
            return Trend(significance=0.0)

        # Analyze escalation pattern
        escalation_pattern = self.analyze_escalation_pattern(escalation_events)
        # Calculate trend significance
        significance = 
self.calculate_escalation_significance(escalation_pattern)

        return Trend(
            trend_type="escalation",
            significance=significance,
            description=f"Detected escalation trend with 
{len(escalation_events)} related events",
            supporting_events=escalation_events,

implications=self.generate_escalation_implications(escalation_pattern)
        )
```

Model Management

Model Lifecycle Management

```
class ModelManager:
    def __init__(self):
        self.model_registry = ModelRegistry()
        self.training_pipeline = TrainingPipeline()
        self.deployment_manager = ModelDeploymentManager()
        self.monitoring_system = ModelMonitoringSystem()

    async def manage_model_lifecycle(self, model_config: ModelConfig) -> 
ModelLifecycleResult:
        """Manage complete model lifecycle"""

        # Train model
        trained_model = await self.training_pipeline.train_model(model_config)

        # Register model
        model_version = await self.model_registry.register_model(trained_model)

        # Deploy model
        deployment_result = await 
self.deployment_manager.deploy_model(model_version)

        # Start monitoring
        await self.monitoring_system.start_monitoring(model_version)

        return ModelLifecycleResult(
            model_version=model_version,
            deployment_result=deployment_result,
            monitoring_started=True
        )
```

Document Information: - **Version**: 1.0 - **Last Updated**: December 2024 - **Document Owner**: Enhanced SICA Development Team - **Classification**: Technical - **Distribution**: Internal/Partner