Autonomous Mode Guide

Document Information

Title: Autonomous Mode Guide

Generated: 2025-06-24 16:37:46

Version: 1.0

Project: Pynomaly - State-of-the-Art Anomaly Detection Platform

Pynomaly Autonomous Mode Guide¶

Overview¶

Pynomaly's Autonomous Mode represents the pinnacle of automated anomaly detection, leveraging advanced AutoML techniques, intelligent algorithm selection, and adaptive learning to provide optimal anomaly detection with minimal human intervention. This guide covers the complete autonomous system, its capabilities, configuration, and best practices.

Table of Contents¶

Introduction to Autonomous Mode¶

What is Autonomous Mode?¶

Autonomous Mode is an intelligent system that automatically: - Analyzes your data characteristics - Selects optimal algorithms - Tunes hyperparameters - Handles data preprocessing - Monitors model performance - Adapts to changing patterns - Provides explanations and insights

Key Benefits¶

- **Zero Configuration**: Works out-of-the-box with sensible defaults
- **Optimal Performance**: Automatically finds best algorithms and parameters
- **Continuous Learning**: Adapts to new patterns and data drift
- **Expert Knowledge**: Incorporates domain expertise and best practices
- **Time Saving**: Reduces weeks of experimentation to minutes
- **Transparency**: Provides clear explanations for all decisions

When to Use Autonomous Mode¶

✅ **Recommended for:** - New anomaly detection projects - Time-constrained deployments - Non-expert users - Exploratory data analysis - Production systems requiring adaptability - Complex, multi-modal datasets

❌ **Consider alternatives for:** - Highly specialized domains with strict requirements - Systems requiring deterministic behavior - Regulatory environments with specific algorithm mandates - Resource-constrained environments

Core Capabilities¶

1. Intelligent Data Analysis¶

The system performs comprehensive data analysis to understand:

Data Characteristics Detection¶

```
# Automatic data profiling
data_profile = {
    "size": {"samples": 100000, "features": 25},
    "types": {
        "numerical": 20,
        "categorical": 3,
        "temporal": 2
    },
```

```
    "quality": {
        "missing_values": 0.02,
        "duplicates": 0.001,
        "outliers": 0.05
    },
    "distributions": {
        "gaussian_features": 12,
        "skewed_features": 5,
        "uniform_features": 3
    },
    "patterns": {
        "seasonality": True,
        "trend": "increasing",
        "cycles": ["weekly", "monthly"]
    },
    "complexity": "moderate"
}
```

Feature Analysis¶

- **Statistical Properties**: Mean, variance, skewness, kurtosis
- **Correlation Structure**: Feature interactions and dependencies
- **Information Content**: Feature importance and redundancy
- **Temporal Patterns**: Seasonality, trends, and cycles
- **Data Quality**: Missing values, outliers, and inconsistencies

2. Algorithm Selection Engine¶

The system uses a sophisticated algorithm selection engine:

Selection Criteria¶

```
selection_criteria = {
    "data_characteristics": {
        "size": "large",           # small, medium, large, huge
        "dimensionality": "high",  # low, medium, high
        "data_type": "mixed",      # numerical, categorical, mixed
        "distribution": "mixed"    # gaussian, skewed, mixed
    },
    "performance_requirements": {
```

```
        "accuracy": "high",        # low, medium, high
        "speed": "medium",         # low, medium, high
        "interpretability": "medium", # low, medium, high
        "memory": "medium"         # low, medium, high
    },
    "domain_context": {
        "domain": "finance",       # finance, healthcare, security, etc.
        "use_case": "fraud_detection",
        "criticality": "high"      # low, medium, high
    }
}
```

Algorithm Recommendation Matrix¶

Data Size	Dimensionality	Primary Algorithms	Ensemble Options
Small (<1K)	Low	LOF, One-Class SVM	Simple Voting
Small	High	PCA + LOF, Isolation Forest	Weighted Voting
Medium (1K-100K)	Low	Isolation Forest, LOF	Bagging
Medium	High	Isolation Forest, AutoEncoder	Stacking
Large (>100K)	Low	Isolation Forest, k-NN	Distributed Ensemble
Large	High	AutoEncoder, Deep Ensemble	Hierarchical

3. Hyperparameter Optimization¶

Multi-Objective Optimization¶

The system optimizes multiple objectives simultaneously:

```
optimization_objectives = {
    "primary": {
        "metric": "f1_score",
        "weight": 0.6
    },
    "secondary": [
        {"metric": "precision", "weight": 0.2},
        {"metric": "recall", "weight": 0.1},
        {"metric": "training_time", "weight": 0.05, "minimize": True},
        {"metric": "memory_usage", "weight": 0.05, "minimize": True}
    ]
}
```

Optimization Strategies¶

- **Bayesian Optimization**: For expensive evaluations
- **Random Search**: For quick exploration
- **Grid Search**: For final fine-tuning
- **Evolutionary Algorithms**: For complex search spaces
- **Multi-Fidelity**: Using early stopping and progressive training

4. Adaptive Learning System¶

Continuous Model Monitoring¶

```
monitoring_config = {
    "data_drift": {
        "method": "ks_test",
        "threshold": 0.05,
        "window_size": 1000
    },
    "performance_drift": {
        "metric": "f1_score",
        "threshold": 0.1,
        "baseline_window": 5000
    },
    "concept_drift": {
        "method": "adwin",
```

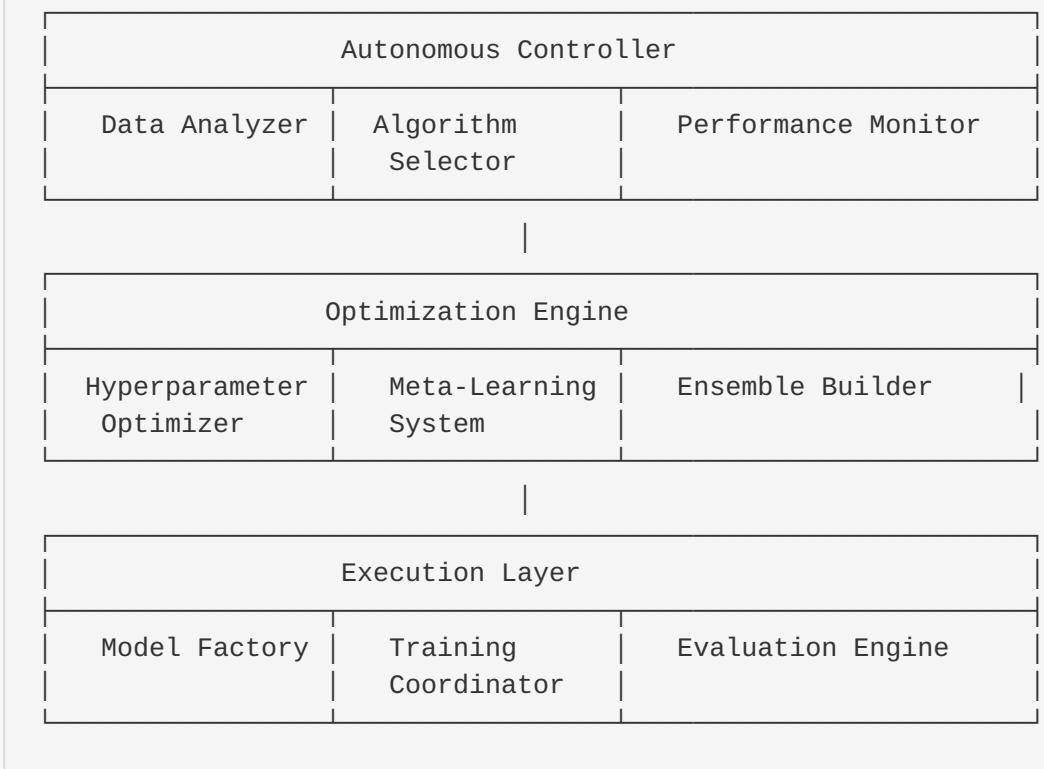
```
        "confidence": 0.95
    }
}
```

Adaptive Strategies¶

- **Incremental Learning**: Update models with new data
- **Model Replacement**: Switch to better-performing algorithms
- **Ensemble Adaptation**: Adjust ensemble weights
- **Parameter Updates**: Fine-tune existing models
- **Complete Retraining**: Full model refresh when needed

System Architecture¶

Autonomous Mode Components¶



Core Components¶

1. Autonomous Controller¶

```
class AutonomousController:
    """Main controller for autonomous anomaly detection."""

    def __init__(self):
        self.data_analyzer = DataAnalyzer()
        self.algorithm_selector = AlgorithmSelector()
        self.optimizer = HyperparameterOptimizer()
        self.monitor = PerformanceMonitor()
        self.explainer = ExplanationEngine()

    async def auto_detect(
        self,
        data: np.ndarray,
        target_metric: str = "f1_score",
        time_budget: int = 3600,  # seconds
        compute_budget: float = 1.0  # relative
    ) -> AutonomousResult:
        """Execute autonomous anomaly detection."""

        # 1. Analyze data
        profile = await self.data_analyzer.analyze(data)

        # 2. Select algorithms
        candidates = await self.algorithm_selector.select(
            profile, target_metric
        )

        # 3. Optimize and evaluate
        results = await self.optimizer.optimize_ensemble(
            candidates, data, time_budget, compute_budget
        )

        # 4. Build final model
        final_model = await self._build_final_model(results)

        # 5. Generate explanations
        explanations = await self.explainer.explain(
            final_model, data, profile
        )

        return AutonomousResult(
            model=final_model,
```

```
            performance=results.best_performance,
            explanations=explanations,
            recommendations=self._generate_recommendations(results)
        )
```

2. Data Analyzer¶

```
class DataAnalyzer:
    """Comprehensive data analysis for autonomous mode."""

    async def analyze(self, data: np.ndarray) -> DataProfile:
        """Analyze data characteristics."""

        profile = DataProfile()

        # Basic statistics
        profile.size = data.shape
        profile.dtypes = self._infer_types(data)

        # Quality assessment
        profile.quality = await self._assess_quality(data)

        # Distribution analysis
        profile.distributions = await self._analyze_distributions(data)

        # Pattern detection
        profile.patterns = await self._detect_patterns(data)

        # Complexity estimation
        profile.complexity = self._estimate_complexity(data)

        # Anomaly characteristics
        profile.anomaly_hints = await self._detect_anomaly_hints(data)

        return profile

    async def _assess_quality(self, data: np.ndarray) -> QualityMetrics:
        """Assess data quality."""
        return QualityMetrics(
            missing_ratio=np.isnan(data).mean(),
            duplicate_ratio=self._calculate_duplicates(data),
            outlier_ratio=self._estimate_outliers(data),
            noise_level=self._estimate_noise(data),
            consistency_score=self._check_consistency(data)
```

```
        )

    async def _detect_patterns(self, data: np.ndarray) -> PatternInfo:
        """Detect temporal and spatial patterns."""
        patterns = PatternInfo()

        # Temporal patterns
        if self._has_temporal_structure(data):
            patterns.seasonality = self._detect_seasonality(data)
            patterns.trends = self._detect_trends(data)
            patterns.cycles = self._detect_cycles(data)

        # Spatial patterns
        patterns.clusters = self._detect_clusters(data)
        patterns.correlations = self._analyze_correlations(data)

        return patterns
```

3. Algorithm Selector¶

```
class AlgorithmSelector:
    """Intelligent algorithm selection based on data characteristics."""

    def __init__(self):
        self.meta_learner = MetaLearner()
        self.knowledge_base = AlgorithmKnowledgeBase()

    async def select(
        self,
        profile: DataProfile,
        target_metric: str
    ) -> List[AlgorithmCandidate]:
        """Select optimal algorithms for given data profile."""

        # 1. Get meta-learned recommendations
        meta_predictions = await self.meta_learner.predict(
            profile, target_metric
        )

        # 2. Apply rule-based filters
        rule_based = self._apply_rules(profile)

        # 3. Combine recommendations
        candidates = self._combine_recommendations(
```

```
            meta_predictions, rule_based
        )

        # 4. Rank by expected performance
        ranked_candidates = await self._rank_candidates(
            candidates, profile, target_metric
        )

        return ranked_candidates[:5]  # Top 5 candidates

    def _apply_rules(self, profile: DataProfile) -> List[str]:
        """Apply rule-based algorithm selection."""

        rules = []

        # Size-based rules
        if profile.size[0] < 1000:
            rules.extend(["LOF", "OneClassSVM"])
        elif profile.size[0] > 100000:
            rules.extend(["IsolationForest", "MiniBatchKMeans"])

        # Dimensionality rules
        if profile.size[1] > 100:
            rules.extend(["AutoEncoder", "PCA"])

        # Pattern-based rules
        if profile.patterns.has_temporal:
            rules.extend(["LSTM", "Prophet"])

        # Quality-based rules
        if profile.quality.noise_level > 0.3:
            rules.extend(["RobustScaler", "IsolationForest"])

        return rules
```

Configuration and Setup¶

Basic Configuration¶

```
# autonomous_config.yml
autonomous_mode:
```

```
  # Core settings
  enabled: true
  auto_preprocessing: true
  auto_feature_selection: true
  auto_algorithm_selection: true
  auto_hyperparameter_tuning: true

  # Performance targets
  target_metrics:
    primary: "f1_score"
    minimum_threshold: 0.8
    optimization_direction: "maximize"

  # Resource constraints
  time_budget: 3600  # seconds
  compute_budget: 1.0  # relative to available resources
  memory_limit: "8GB"
  cpu_cores: -1  # use all available
  gpu_enabled: true

  # Algorithm preferences
  algorithm_constraints:
    excluded_algorithms: []
    preferred_families: ["tree_based", "ensemble"]
    interpretability_weight: 0.3
    speed_weight: 0.2

  # Adaptation settings
  adaptation:
    enabled: true
    monitoring_interval: 3600  # seconds
    drift_sensitivity: 0.1
    retraining_threshold: 0.15
    max_model_age: 604800  # 1 week in seconds
```

Advanced Configuration¶

```
# Python configuration
autonomous_config = AutonomousConfig(
    # Data analysis settings
    data_analysis=DataAnalysisConfig(
        sample_size=10000,
        correlation_threshold=0.8,
```

```
        outlier_methods=["iqr", "isolation_forest"],
        pattern_detection=True,
        statistical_tests=True
    ),

    # Algorithm selection
    algorithm_selection=AlgorithmSelectionConfig(
        max_candidates=5,
        diversity_weight=0.3,
        performance_weight=0.7,
        meta_learning_enabled=True,
        cold_start_algorithms=["IsolationForest", "LOF"]
    ),

    # Optimization settings
    optimization=OptimizationConfig(
        method="bayesian",
        n_trials=100,
        early_stopping=True,
        pruning_enabled=True,
        parallel_trials=4
    ),

    # Ensemble configuration
    ensemble=EnsembleConfig(
        enabled=True,
        max_models=5,
        selection_method="diversity",
        combination_method="weighted_voting",
        weight_optimization=True
    ),

    # Monitoring and adaptation
    monitoring=MonitoringConfig(
        metrics=["accuracy", "precision", "recall", "f1_score"],
        drift_detection=["data_drift", "concept_drift"],
        alert_thresholds={"f1_score": 0.1},
        adaptation_strategy="incremental"
    )
)
```

Usage Patterns¶

1. Quick Start (Zero Configuration)¶

```
# CLI - Simplest usage
pynomaly auto detect --dataset data.csv

# Automatic everything: preprocessing, algorithm selection, tuning
pynomaly auto detect \
    --dataset data.csv \
    --target-metric f1 \
    --output results/
```

```
# Python SDK - Minimal code
from pynomaly import AutonomousDetector

detector = AutonomousDetector()
results = detector.fit_predict(data)
```

2. Guided Configuration¶

```
# Interactive setup
pynomaly auto configure --interactive

# Guided questions:
# - What type of data? (tabular/time-series/text/images)
# - What's your priority? (accuracy/speed/interpretability)
# - What's your use case? (fraud/quality/security/monitoring)
# - Any resource constraints?
```

```
# Programmatic guided setup
detector = AutonomousDetector()
config = detector.guided_setup(
    data_preview=data.head(1000),
    priorities={"accuracy": 0.6, "speed": 0.4},
    constraints={"max_training_time": 1800}
)
results = detector.fit_predict(data, config=config)
```

3. Domain-Specific Presets¶

```
# Use domain presets
pynomaly auto detect \
    --dataset financial_transactions.csv \
    --preset fraud_detection

# Available presets: fraud_detection, quality_control,
# network_security, predictive_maintenance, etc.
```

```
# Domain-specific configuration
detector = AutonomousDetector.for_fraud_detection()
results = detector.fit_predict(transaction_data)

detector = AutonomousDetector.for_quality_control()
results = detector.fit_predict(manufacturing_data)
```

4. Continuous Learning Setup¶

```
# Set up continuous learning
detector = AutonomousDetector(
    adaptation_enabled=True,
    monitoring_interval=3600,  # 1 hour
    retraining_threshold=0.1
)

# Initial training
detector.fit(historical_data)

# Deploy for continuous operation
detector.start_continuous_learning(
    data_stream=stream,
    feedback_loop=feedback_handler
)
```

Intelligent Features¶

1. Automated Data Preprocessing¶

Feature Engineering¶

```
# Automatic feature engineering pipeline
preprocessing_pipeline = AutonomousPreprocessor(
    numeric_strategies=[
        "standard_scaling",
        "outlier_clipping",
        "polynomial_features",
        "interaction_terms"
    ],
    categorical_strategies=[
        "target_encoding",
        "frequency_encoding",
        "category_embedding"
    ],
```

```
    temporal_strategies=[
        "time_features",
        "lag_features",
        "rolling_statistics",
        "seasonality_features"
    ]
)

processed_data = preprocessing_pipeline.fit_transform(raw_data)
```

Automated Feature Selection¶

```
# Intelligent feature selection
feature_selector = AutonomousFeatureSelector(
    methods=[
        "variance_threshold",
        "correlation_filter",
        "mutual_info_selection",
        "recursive_elimination",
        "lasso_selection"
    ],
    target_features=None,  # Auto-determine optimal number
    redundancy_threshold=0.95
)

selected_features = feature_selector.select(data, target)
```

2. Meta-Learning System¶

Algorithm Performance Prediction¶

```
class MetaLearner:
    """Predicts algorithm performance based on dataset characteristics."""

    def __init__(self):
        self.meta_features_extractor = MetaFeaturesExtractor()
        self.performance_predictor = GradientBoostingRegressor()
```

```
        self.knowledge_base = self._load_knowledge_base()

    def predict_performance(
        self,
        dataset_profile: DataProfile,
        algorithm: str
    ) -> PerformancePrediction:
        """Predict expected performance of algorithm on dataset."""

        # Extract meta-features
        meta_features = self.meta_features_extractor.extract(dataset_profile)

        # Predict performance metrics
        predicted_metrics = {}
        for metric in ["accuracy", "precision", "recall", "f1_score"]:
            prediction = self.performance_predictor.predict([
                meta_features + [self._encode_algorithm(algorithm)]
            ])[0]
            predicted_metrics[metric] = prediction

        # Estimate confidence
        confidence = self._calculate_confidence(meta_features, algorithm)

        return PerformancePrediction(
            metrics=predicted_metrics,
            confidence=confidence,
            reasoning=self._generate_reasoning(meta_features, algorithm)
        )
```

Transfer Learning¶

```
class TransferLearner:
    """Applies knowledge from similar datasets."""

    def find_similar_datasets(
        self,
        target_profile: DataProfile
    ) -> List[SimilarDataset]:
        """Find datasets similar to target for knowledge transfer."""

        similarities = []
        for dataset in self.knowledge_base.datasets:
            similarity = self._calculate_similarity(
                target_profile, dataset.profile
```

```
            )
            if similarity > 0.7:
                similarities.append(SimilarDataset(
                    dataset=dataset,
                    similarity=similarity,
                    best_algorithms=dataset.performance.best_algorithms
                ))

        return sorted(similarities, key=lambda x: x.similarity, reverse=True)

    def transfer_hyperparameters(
        self,
        algorithm: str,
        similar_datasets: List[SimilarDataset]
    ) -> Dict[str, Any]:
        """Transfer hyperparameters from similar datasets."""

        # Weighted average of hyperparameters based on similarity
        transferred_params = {}
        total_weight = sum(ds.similarity for ds in similar_datasets)

        for param_name in self._get_algorithm_params(algorithm):
            weighted_sum = 0
            for dataset in similar_datasets:
                if algorithm in dataset.best_algorithms:
                    param_value = dataset.best_algorithms[algorithm].params.get(pa
                    if param_value is not None:
                        weighted_sum += param_value * dataset.similarity

            if total_weight > 0:
                transferred_params[param_name] = weighted_sum / total_weight

        return transferred_params
```

3. Intelligent Ensemble Construction¶

Dynamic Ensemble Building¶

```
class IntelligentEnsembleBuilder:
    """Builds optimal ensembles based on model diversity and performance."""

    def build_ensemble(
        self,
```

```
        candidate_models: List[Model],
        validation_data: Tuple[np.ndarray, np.ndarray]
    ) -> EnsembleModel:
        """Build optimal ensemble from candidate models."""

        X_val, y_val = validation_data

        # Evaluate individual models
        model_performances = {}
        model_predictions = {}

        for model in candidate_models:
            predictions = model.predict(X_val)
            performance = self._evaluate_model(predictions, y_val)

            model_performances[model.id] = performance
            model_predictions[model.id] = predictions

        # Calculate diversity matrix
        diversity_matrix = self._calculate_diversity_matrix(model_predictions)

        # Select models using multi-objective optimization
        selected_models = self._select_ensemble_models(
            candidate_models,
            model_performances,
            diversity_matrix,
            max_models=5
        )

        # Optimize ensemble weights
        weights = self._optimize_ensemble_weights(
            selected_models, model_predictions, y_val
        )

        return EnsembleModel(
            models=selected_models,
            weights=weights,
            combination_method="weighted_voting"
        )

    def _select_ensemble_models(
        self,
        candidates: List[Model],
        performances: Dict[str, float],
        diversity_matrix: np.ndarray,
        max_models: int
    ) -> List[Model]:
        """Select models for ensemble using Pareto optimization."""
```

```
        # Multi-objective optimization: maximize performance and diversity
        def objective(model_indices):
            selected_models = [candidates[i] for i in model_indices]

            # Average performance
            avg_performance = np.mean([
                performances[model.id] for model in selected_models
            ])

            # Average pairwise diversity
            if len(model_indices) > 1:
                diversity_pairs = []
                for i in range(len(model_indices)):
                    for j in range(i+1, len(model_indices)):
                        diversity_pairs.append(
                            diversity_matrix[model_indices[i], model_indices[j]]
                        )
                avg_diversity = np.mean(diversity_pairs)
            else:
                avg_diversity = 0

            # Combined objective (weighted sum)
            return 0.7 * avg_performance + 0.3 * avg_diversity

        # Use genetic algorithm for selection
        best_combination = self._genetic_algorithm_selection(
            objective, len(candidates), max_models
        )

        return [candidates[i] for i in best_combination]
```

4. Adaptive Learning and Drift Detection¶

Drift Detection System¶

```
class DriftDetector:
    """Detects various types of drift in data and model performance."""

    def __init__(self):
        self.data_drift_detectors = {
            "ks_test": KolmogorovSmirnovTest(),
            "wasserstein": WassersteinDistance(),
            "jensen_shannon": JensenShannonDivergence()
```

```
        }
        self.concept_drift_detectors = {
            "adwin": ADWIN(),
            "page_hinkley": PageHinkley(),
            "ddm": DDM()
        }

    def detect_drift(
        self,
        reference_data: np.ndarray,
        current_data: np.ndarray,
        performance_history: List[float]
    ) -> DriftReport:
        """Comprehensive drift detection."""

        report = DriftReport()

        # Data drift detection
        data_drift_results = {}
        for name, detector in self.data_drift_detectors.items():
            drift_score = detector.detect(reference_data, current_data)
            data_drift_results[name] = drift_score

        # Aggregate data drift signals
        report.data_drift = DriftSignal(
            detected=np.mean(list(data_drift_results.values())) > 0.1,
            confidence=np.std(list(data_drift_results.values())),
            details=data_drift_results
        )

        # Concept drift detection
        concept_drift_results = {}
        for name, detector in self.concept_drift_detectors.items():
            for performance in performance_history:
                detector.add_element(performance)

            concept_drift_results[name] = detector.detected_change()

        report.concept_drift = DriftSignal(
            detected=any(concept_drift_results.values()),
            confidence=sum(concept_drift_results.values()) / len(concept_drift_res
            details=concept_drift_results
        )

        # Performance drift
        if len(performance_history) > 10:
            recent_performance = np.mean(performance_history[-5:])
            baseline_performance = np.mean(performance_history[:5])
            performance_degradation = baseline_performance - recent_performance
```

```
            report.performance_drift = DriftSignal(
                detected=performance_degradation > 0.1,
                confidence=abs(performance_degradation),
                details={"degradation": performance_degradation}
            )

        return report
```

Adaptive Response System¶

```
class AdaptiveResponseSystem:
    """Responds to detected drift with appropriate adaptation strategies."""

    def __init__(self):
        self.adaptation_strategies = {
            "incremental_update": IncrementalLearning(),
            "model_retraining": ModelRetraining(),
            "ensemble_update": EnsembleAdaptation(),
            "parameter_tuning": ParameterAdaptation(),
            "algorithm_switch": AlgorithmSwitching()
        }

    def respond_to_drift(
        self,
        drift_report: DriftReport,
        current_model: Model,
        new_data: np.ndarray
    ) -> AdaptationResult:
        """Select and execute appropriate adaptation strategy."""

        # Determine adaptation strategy based on drift type and severity
        strategy = self._select_adaptation_strategy(drift_report)

        # Execute adaptation
        adaptation_result = strategy.adapt(current_model, new_data, drift_report)

        # Validate adaptation
        validation_result = self._validate_adaptation(
            original_model=current_model,
            adapted_model=adaptation_result.model,
            validation_data=new_data
        )
```

```
        return AdaptationResult(
            strategy=strategy.name,
            model=adaptation_result.model,
            performance_improvement=validation_result.improvement,
            adaptation_confidence=validation_result.confidence,
            rollback_available=True
        )

    def _select_adaptation_strategy(self, drift_report: DriftReport) -> str:
        """Select optimal adaptation strategy based on drift characteristics."""

        # Data drift -> incremental learning or retraining
        if drift_report.data_drift.detected:
            if drift_report.data_drift.confidence < 0.3:
                return "incremental_update"
            else:
                return "model_retraining"

        # Concept drift -> ensemble update or algorithm switch
        if drift_report.concept_drift.detected:
            if drift_report.concept_drift.confidence < 0.5:
                return "ensemble_update"
            else:
                return "algorithm_switch"

        # Performance drift -> parameter tuning
        if drift_report.performance_drift.detected:
            return "parameter_tuning"

        return "incremental_update"  # Default conservative approach
```

Performance Optimization¶

Computational Optimization¶

Intelligent Resource Management¶

```
class ResourceManager:
    """Manages computational resources for optimal performance."""

    def __init__(self):
```

```
        self.system_monitor = SystemMonitor()
        self.resource_predictor = ResourcePredictor()

    def optimize_resource_allocation(
        self,
        training_tasks: List[TrainingTask]
    ) -> ResourceAllocation:
        """Optimize resource allocation across training tasks."""

        # Monitor current system state
        system_state = self.system_monitor.get_current_state()

        # Predict resource requirements for each task
        resource_predictions = {}
        for task in training_tasks:
            prediction = self.resource_predictor.predict(
                algorithm=task.algorithm,
                data_size=task.data_size,
                hyperparameter_space=task.param_space
            )
            resource_predictions[task.id] = prediction

        # Optimize allocation using integer programming
        allocation = self._solve_resource_allocation(
            tasks=training_tasks,
            predictions=resource_predictions,
            constraints=system_state.constraints
        )

        return allocation
```

Parallel and Distributed Training¶

```
class DistributedTrainingCoordinator:
    """Coordinates distributed training across multiple nodes."""

    def __init__(self, cluster_config: ClusterConfig):
        self.cluster = cluster_config
        self.task_scheduler = TaskScheduler()
        self.result_aggregator = ResultAggregator()

    async def distribute_hyperparameter_search(
        self,
        algorithm: str,
```

```
        data: np.ndarray,
        param_space: Dict[str, Any],
        n_trials: int
    ) -> OptimizationResult:
        """Distribute hyperparameter search across cluster."""

        # Partition parameter space
        param_partitions = self._partition_parameter_space(
            param_space, self.cluster.n_nodes
        )

        # Create training tasks
        tasks = []
        for i, partition in enumerate(param_partitions):
            task = TrainingTask(
                node_id=self.cluster.nodes[i],
                algorithm=algorithm,
                data_partition=self._partition_data(data, i),
                param_space=partition,
                n_trials=n_trials // self.cluster.n_nodes
            )
            tasks.append(task)

        # Schedule and execute tasks
        task_futures = []
        for task in tasks:
            future = self.task_scheduler.schedule(task)
            task_futures.append(future)

        # Collect results
        node_results = await asyncio.gather(*task_futures)

        # Aggregate results
        final_result = self.result_aggregator.aggregate(node_results)

        return final_result
```

Memory Optimization¶

Memory-Efficient Training¶

```
class MemoryOptimizer:
    """Optimizes memory usage during training."""
```

```
    def __init__(self):
        self.memory_monitor = MemoryMonitor()
        self.data_sampler = AdaptiveDataSampler()

    def optimize_training_memory(
        self,
        algorithm: str,
        data: np.ndarray,
        memory_limit: int
    ) -> OptimizedTrainingPlan:
        """Create memory-optimized training plan."""

        # Estimate memory requirements
        memory_estimate = self._estimate_memory_usage(algorithm, data.shape)

        if memory_estimate <= memory_limit:
            # Sufficient memory - use full dataset
            return OptimizedTrainingPlan(
                strategy="full_batch",
                batch_size=len(data),
                data_sampling_ratio=1.0
            )

        # Insufficient memory - optimize
        if memory_estimate <= memory_limit * 2:
            # Use mini-batch training
            optimal_batch_size = self._calculate_optimal_batch_size(
                memory_limit, algorithm, data.shape
            )
            return OptimizedTrainingPlan(
                strategy="mini_batch",
                batch_size=optimal_batch_size,
                data_sampling_ratio=1.0
            )
        else:
            # Use data sampling + mini-batch
            sampling_ratio = memory_limit / memory_estimate
            optimal_batch_size = self._calculate_optimal_batch_size(
                memory_limit, algorithm, (int(len(data) * sampling_ratio), data.sh
            )
            return OptimizedTrainingPlan(
                strategy="sampled_mini_batch",
                batch_size=optimal_batch_size,
                data_sampling_ratio=sampling_ratio
            )
```

Monitoring and Control¶

Real-time Monitoring Dashboard¶

Performance Metrics¶

```
monitoring_metrics = {
    "model_performance": {
        "accuracy": 0.89,
        "precision": 0.87,
        "recall": 0.91,
        "f1_score": 0.89,
        "auc_roc": 0.94
    },
    "system_performance": {
        "prediction_latency_ms": 45,
        "throughput_per_second": 1000,
        "memory_usage_mb": 2048,
        "cpu_utilization": 0.65,
        "gpu_utilization": 0.82
    },
    "data_quality": {
        "missing_values_ratio": 0.02,
        "outlier_ratio": 0.05,
        "drift_score": 0.08,
        "schema_violations": 0
    },
    "adaptation_history": {
        "total_adaptations": 15,
        "successful_adaptations": 14,
        "last_adaptation": "2024-06-24T10:30:00Z",
        "adaptation_frequency": "every 4 hours"
    }
}
```

Alert System¶

```
class AlertSystem:
    """Intelligent alerting for autonomous mode."""

    def __init__(self):
        self.alert_rules = self._initialize_alert_rules()
        self.notification_channels = NotificationChannels()

    def _initialize_alert_rules(self) -> List[AlertRule]:
        return [
            AlertRule(
                name="performance_degradation",
                condition="f1_score < baseline_f1 - 0.1",
                severity="high",
                actions=["retrain_model", "notify_admin"]
            ),
            AlertRule(
                name="data_drift_detected",
                condition="drift_score > 0.2",
                severity="medium",
                actions=["schedule_adaptation", "notify_team"]
            ),
            AlertRule(
                name="system_overload",
                condition="cpu_utilization > 0.9 OR memory_usage > 0.95",
                severity="critical",
                actions=["scale_resources", "emergency_notification"]
            ),
            AlertRule(
                name="adaptation_failure",
                condition="adaptation_success_rate < 0.8",
                severity="high",
                actions=["fallback_to_baseline", "expert_review"]
            )
        ]

    async def check_alerts(self, metrics: Dict[str, Any]) -> List[Alert]:
        """Check all alert conditions against current metrics."""

        triggered_alerts = []

        for rule in self.alert_rules:
            if self._evaluate_condition(rule.condition, metrics):
                alert = Alert(
                    rule=rule.name,
```

```
                    severity=rule.severity,
                    message=self._generate_alert_message(rule, metrics),
                    timestamp=datetime.utcnow(),
                    suggested_actions=rule.actions
                )
                triggered_alerts.append(alert)

        # Execute alert actions
        for alert in triggered_alerts:
            await self._execute_alert_actions(alert)

        return triggered_alerts
```

Control Interface¶

Manual Override System¶

```
class ManualOverrideSystem:
    """Allows manual control over autonomous decisions."""

    def __init__(self):
        self.override_history = []
        self.current_overrides = {}

    def override_algorithm_selection(
        self,
        forced_algorithms: List[str],
        reason: str,
        duration: Optional[int] = None
    ) -> OverrideResult:
        """Force specific algorithms to be used."""

        override = Override(
            type="algorithm_selection",
            parameters={"forced_algorithms": forced_algorithms},
            reason=reason,
            duration=duration,
            created_by="manual",
            created_at=datetime.utcnow()
        )

        self.current_overrides["algorithm_selection"] = override
        self.override_history.append(override)
```

```
        return OverrideResult(
            success=True,
            override_id=override.id,
            message=f"Algorithm selection overridden with {forced_algorithms}"
        )

    def override_hyperparameters(
        self,
        algorithm: str,
        forced_params: Dict[str, Any],
        reason: str
    ) -> OverrideResult:
        """Force specific hyperparameters."""

        override = Override(
            type="hyperparameters",
            parameters={
                "algorithm": algorithm,
                "forced_params": forced_params
            },
            reason=reason,
            created_by="manual",
            created_at=datetime.utcnow()
        )

        self.current_overrides[f"hyperparameters_{algorithm}"] = override
        self.override_history.append(override)

        return OverrideResult(
            success=True,
            override_id=override.id,
            message=f"Hyperparameters overridden for {algorithm}"
        )

    def clear_override(self, override_id: str) -> bool:
        """Clear a specific override."""

        for key, override in self.current_overrides.items():
            if override.id == override_id:
                del self.current_overrides[key]
                override.cleared_at = datetime.utcnow()
                return True

        return False
```

Advanced Scenarios¶

1. Multi-Dataset Learning¶

```
class MultiDatasetLearner:
    """Learns from multiple related datasets simultaneously."""

    async def learn_from_multiple_datasets(
        self,
        datasets: List[Dataset],
        relationships: Dict[str, str]  # dataset relationships
    ) -> MultiDatasetModel:
        """Learn optimal models across multiple related datasets."""

        # Analyze dataset relationships
        relationship_graph = self._build_relationship_graph(datasets, relationship

        # Identify shared patterns
        shared_patterns = await self._identify_shared_patterns(datasets)

        # Train base models on individual datasets
        individual_models = {}
        for dataset in datasets:
            model = await self._train_individual_model(dataset)
            individual_models[dataset.id] = model

        # Train meta-model for cross-dataset knowledge
        meta_model = await self._train_meta_model(
            individual_models, shared_patterns, relationship_graph
        )

        return MultiDatasetModel(
            individual_models=individual_models,
            meta_model=meta_model,
            shared_patterns=shared_patterns
        )
```

2. Federated Autonomous Learning¶

```
class FederatedAutonomousLearner:
    """Autonomous learning across federated data sources."""

    def __init__(self, federation_config: FederationConfig):
        self.federation = federation_config
        self.consensus_engine = ConsensusEngine()
        self.privacy_engine = PrivacyEngine()

    async def federated_learning(
        self,
        local_data: np.ndarray,
        global_rounds: int = 10
    ) -> FederatedModel:
        """Perform federated autonomous learning."""

        # Initialize local autonomous detector
        local_detector = AutonomousDetector()

        # Global training loop
        global_model = None
        for round_num in range(global_rounds):

            # Local training
            local_model = await local_detector.fit(
                local_data,
                base_model=global_model
            )

            # Privacy-preserving model update
            private_update = self.privacy_engine.privatize_model(
                local_model, global_model
            )

            # Send update to federation
            await self.federation.send_update(private_update)

            # Receive global model update
            global_updates = await self.federation.receive_updates()

            # Consensus-based aggregation
            global_model = self.consensus_engine.aggregate_models(
                global_updates, consensus_threshold=0.7
            )
```

```
        return FederatedModel(
            global_model=global_model,
            local_contribution=local_model,
            privacy_level=self.privacy_engine.privacy_level
        )
```

3. Explainable Autonomous Decisions¶

```
class ExplainableAutonomousMode:
    """Provides explanations for all autonomous decisions."""

    def __init__(self):
        self.decision_logger = DecisionLogger()
        self.explanation_generator = ExplanationGenerator()

    async def explain_algorithm_selection(
        self,
        selected_algorithms: List[str],
        data_profile: DataProfile
    ) -> AlgorithmSelectionExplanation:
        """Explain why specific algorithms were selected."""

        explanation = AlgorithmSelectionExplanation()

        for algorithm in selected_algorithms:
            # Rule-based reasoning
            rule_reasons = self._get_rule_based_reasons(algorithm, data_profile)

            # Meta-learning reasoning
            meta_reasons = await self._get_meta_learning_reasons(
                algorithm, data_profile
            )

            # Performance prediction reasoning
            perf_reasons = self._get_performance_reasons(algorithm, data_profile)

            explanation.add_algorithm_explanation(
                algorithm=algorithm,
                rule_based_reasons=rule_reasons,
                meta_learning_reasons=meta_reasons,
                performance_reasons=perf_reasons
            )
```

```
        return explanation

    async def explain_hyperparameter_choices(
        self,
        algorithm: str,
        selected_params: Dict[str, Any],
        optimization_history: List[Trial]
    ) -> HyperparameterExplanation:
        """Explain hyperparameter selection process."""

        explanation = HyperparameterExplanation(algorithm=algorithm)

        for param_name, param_value in selected_params.items():

            # Optimization path explanation
            optimization_path = self._trace_optimization_path(
                param_name, optimization_history
            )

            # Sensitivity analysis
            sensitivity = self._analyze_parameter_sensitivity(
                param_name, optimization_history
            )

            # Literature-based reasoning
            literature_support = self._get_literature_support(
                algorithm, param_name, param_value
            )

            explanation.add_parameter_explanation(
                parameter=param_name,
                value=param_value,
                optimization_path=optimization_path,
                sensitivity=sensitivity,
                literature_support=literature_support
            )

        return explanation
```

Conclusion¶

Pynomaly's Autonomous Mode represents the state-of-the-art in automated anomaly detection, combining:

- **Intelligent automation** with minimal configuration required
- **Adaptive learning** that evolves with your data
- **Explainable decisions** for transparency and trust
- **Scalable architecture** from single machine to distributed clusters
- **Domain expertise** built into the system
- **Continuous optimization** for sustained performance

Key Benefits Summary¶

1. **Dramatically reduced time-to-value** - from weeks to minutes
2. **Optimal performance** without manual tuning
3. **Robust operations** with automatic adaptation
4. **Expert-level decisions** accessible to non-experts
5. **Scalable deployment** across any infrastructure
6. **Future-proof architecture** that evolves with new algorithms

Getting Started Recommendations¶

1. **Start simple**: Use zero-configuration mode first
2. **Monitor closely**: Watch the decision explanations to build trust
3. **Gradual customization**: Add constraints and preferences over time
4. **Leverage presets**: Use domain-specific configurations when available
5. **Enable adaptation**: Allow the system to evolve with your data
6. **Provide feedback**: Help improve the meta-learning system

The autonomous mode embodies the future of anomaly detection - intelligent, adaptive, and accessible to users of all skill levels while maintaining the sophistication required for production deployments.