Module 3: Adaptive Algorithms

Table of Contents

Introduction to Adaptive Algorithms

What are Adaptive Algorithms?

Adaptive algorithms are systems that can automatically adjust their behavior based on changing environment conditions and performance feedback. In MEV, these algorithms are crucial because:

1. **Market Dynamics**: Blockchain environments are constantly evolving
2. **Competition**: Other MEV searchers adapt their strategies
3. **Protocol Changes**: DeFi protocols update their mechanisms
4. **Network Conditions**: Gas prices and block times fluctuate
5. **Regulatory Changes**: New rules and restrictions emerge

Types of Adaptation

1. Parameter Adaptation

- Adjusting strategy parameters based on performance
- Learning optimal thresholds and limits
- Dynamic position sizing

2. Strategy Adaptation

- Switching between different MEV strategies
- Combining multiple approaches
- Hybrid systems

3. Market Adaptation

- Responding to changing volatility
- Adapting to new DeFi protocols
- Handling network congestion

4. Risk Adaptation

- Dynamic risk limits
- Adaptive position sizing
- Real-time risk assessment

Adaptive Algorithm Framework

```
Observation → Analysis → Decision → Action → Feedback → Update
```

Online Learning Methods

Stochastic Gradient Descent for MEV

```
import numpy as np
import pandas as pd
from typing import Dict, List, Tuple
from dataclasses import dataclass
import asyncio
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns

def setup_matplotlib_for_plotting():
    """Setup matplotlib for non-interactive plotting"""
    warnings.filterwarnings('default')
    plt.switch_backend("Agg")
    plt.style.use("seaborn-v0_8")
    sns.set_palette("husl")
    plt.rcParams["font.sans-serif"] = ["Noto Sans CJK SC", "WenQuanYi
Zen Hei", "PingFang SC", "Arial Unicode MS", "Hiragino Sans GB"]
    plt.rcParams["axes.unicode_minus"] = False

@dataclass
class MEVObservation:
    """Represents a MEV observation for online learning"""
    timestamp: datetime
    features: np.ndarray
    market_conditions: Dict
    action_taken: str
    reward: float
    profit_loss: float
    risk_metrics: Dict

class OnlineGradientOptimizer:
    """Online gradient-based optimization for MEV parameters"""

    def __init__(self, parameter_dim: int, learning_rate: float = 0.01,
                 momentum: float = 0.9, regularization: float = 0.001):
        self.parameter_dim = parameter_dim
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.regularization = regularization

        # Initialize parameters
        self.parameters = np.random.normal(0, 0.1, parameter_dim)
```

```
        # Momentum terms
        self.velocity = np.zeros(parameter_dim)

        # Performance tracking
        self.performance_history = []
        self.parameter_history = []
        self.gradients_history = []

        # Adaptation parameters
        self.base_learning_rate = learning_rate
        self.min_learning_rate = 0.0001
        self.max_learning_rate = 0.1

    def update_parameters(self, observation: MEVObservation) ->
np.ndarray:
        """Update parameters using SGD with observation"""

        # Calculate gradient of loss function
        gradient = self._calculate_gradient(observation)

        # Add momentum
        self.velocity = self.momentum * self.velocity -
self.learning_rate * gradient

        # Update parameters
        self.parameters += self.velocity

        # Apply regularization
        self.parameters *= (1 - self.learning_rate *
self.regularization)

        # Clip parameters to prevent extreme values
        self.parameters = np.clip(self.parameters, -10, 10)

        # Track history
        self.gradients_history.append(gradient.copy())
        self.parameter_history.append(self.parameters.copy())
        self.performance_history.append(observation.reward)

        # Adapt learning rate
        self._adapt_learning_rate(observation)
```

```
        return self.parameters.copy()

    def _calculate_gradient(self, observation: MEVObservation) ->
np.ndarray:
        """Calculate gradient of loss function w.r.t. parameters"""

        # Simple example: gradient based on profit/loss and prediction
error
        features = observation.features

        # Example loss function: -profit + regularization
        # dL/dθ = -∂profit/∂θ + regularization_term

        # For this example, we'll use a simple gradient
        # In practice, this would be computed using automatic
differentiation
        predicted_action = self._predict_action(features)
        actual_action = observation.action_taken

        # Action prediction gradient (simplified)
        action_error = self._calculate_action_error(predicted_action,
actual_action)

        # Risk adjustment gradient
        risk_adjustment = self._calculate_risk_adjustment(observation)

        # Total gradient
        gradient = features * action_error * risk_adjustment

        # Add regularization gradient
        gradient += self.regularization * self.parameters

        return gradient

    def _predict_action(self, features: np.ndarray) -> str:
        """Predict action based on current parameters and features"""

        # Simplified action prediction
        # In practice, this would be a more sophisticated model
        action_score = np.dot(self.parameters, features)
```

```
        if action_score > 0.5:
            return "arbitrage"
        elif action_score > 0:
            return "liquidation"
        else:
            return "hold"

    def _calculate_action_error(self, predicted_action: str,
actual_action: str) -> float:
        """Calculate error between predicted and actual action"""

        if predicted_action == actual_action:
            return 0.0
        else:
            # Different actions have different penalties
            action_penalties = {
                ("hold", "arbitrage"): 1.0,
                ("hold", "liquidation"): 1.2,
                ("arbitrage", "hold"): 0.8,
                ("liquidation", "hold"): 0.8,
                ("arbitrage", "liquidation"): 0.5,
                ("liquidation", "arbitrage"): 0.5
            }

            return action_penalties.get((predicted_action,
actual_action), 1.0)

    def _calculate_risk_adjustment(self, observation: MEVObservation) -
> float:
        """Calculate risk-based gradient adjustment"""

        # High-risk situations should have smaller gradient updates
        risk_score = observation.risk_metrics.get('overall_risk', 0.5)

        # Reduce gradient magnitude for high-risk situations
        risk_adjustment = 1.0 / (1.0 + 2.0 * risk_score)

        return risk_adjustment

    def _adapt_learning_rate(self, observation: MEVObservation):
        """Adapt learning rate based on performance"""
```

```
        recent_performance = self._get_recent_performance(window=10)

        if len(recent_performance) >= 5:
            # Calculate performance trend
            performance_trend = recent_performance[-1] -
recent_performance[-5]

            # Adapt learning rate based on trend
            if performance_trend > 0:
                # Improving performance - slightly increase learning
rate
                self.learning_rate = min(self.learning_rate * 1.01,
self.max_learning_rate)
            else:
                # Declining performance - decrease learning rate
                self.learning_rate = max(self.learning_rate * 0.99,
self.min_learning_rate)

    def _get_recent_performance(self, window: int = 10) -> List[float]:
        """Get recent performance values"""

        return self.performance_history[-window:] if
len(self.performance_history) >= window else
self.performance_history.copy()

    def get_adaptation_statistics(self) -> Dict:
        """Get statistics about the adaptation process"""

        if len(self.performance_history) < 2:
            return {'status': 'insufficient_data'}

        recent_performance = self._get_recent_performance(50)

        return {
            'total_updates': len(self.performance_history),
            'current_performance': recent_performance[-1] if
recent_performance else 0,
            'performance_improvement': recent_performance[-1] -
recent_performance[0] if len(recent_performance) > 1 else 0,
            'performance_volatility': np.std(recent_performance) if
len(recent_performance) > 1 else 0,
            'parameter_stability': np.std([np.linalg.norm(p) for p in
```

```
self.parameter_history[-10:]]) if len(self.parameter_history) >= 10
else 0,
            'current_learning_rate': self.learning_rate,
            'gradient_magnitude':
np.linalg.norm(self.gradients_history[-1]) if self.gradients_history
else 0
        }

    def plot_adaptation_history(self, save_path: str = None):
        """Plot adaptation history"""

        setup_matplotlib_for_plotting()

        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # Performance over time
        axes[0, 0].plot(self.performance_history)
        axes[0, 0].set_title('Performance Over Time')
        axes[0, 0].set_xlabel('Update Number')
        axes[0, 0].set_ylabel('Reward')

        # Parameter evolution
        if len(self.parameter_history) > 0:
            param_array = np.array(self.parameter_history)
            for i in range(min(param_array.shape[1], 5)):
# Plot first 5 parameters
                axes[0, 1].plot(param_array[:, i], label=f'Param {i}')
            axes[0, 1].set_title('Parameter Evolution')
            axes[0, 1].set_xlabel('Update Number')
            axes[0, 1].set_ylabel('Parameter Value')
            axes[0, 1].legend()

        # Learning rate adaptation
        learning_rates = [self.base_learning_rate] *
len(self.performance_history)
        axes[1, 0].plot(learning_rates, label='Base LR')
        # In a real implementation, you'd track learning rate changes
        axes[1, 0].set_title('Learning Rate Adaptation')
        axes[1, 0].set_xlabel('Update Number')
        axes[1, 0].set_ylabel('Learning Rate')

        # Gradient magnitude
```

```
        if self.gradients_history:
            gradient_magnitudes = [np.linalg.norm(g) for g in
self.gradients_history]
            axes[1, 1].plot(gradient_magnitudes)
            axes[1, 1].set_title('Gradient Magnitude')
            axes[1, 1].set_xlabel('Update Number')
            axes[1, 1].set_ylabel('||Gradient||')

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.close()

class AdaptiveThresholdOptimizer:
    """Online optimization of MEV thresholds and parameters"""

    def __init__(self, initial_thresholds: Dict):
        self.thresholds = initial_thresholds.copy()
        self.performance_history = {}
        self.adaptation_rules = {}

        # Initialize performance tracking for each threshold
        for threshold_name in self.thresholds.keys():
            self.performance_history[threshold_name] = []
            self.adaptation_rules[threshold_name] = {
                'learning_rate': 0.1,
                'min_value': 0.0,
                'max_value': 1.0,
                'adaptation_frequency': 10,
                'performance_window': 50
            }

    def evaluate_thresholds(self, observations: List[MEVObservation]) -
> Dict:
        """Evaluate current threshold performance"""

        evaluation_results = {}

        for threshold_name, threshold_value in self.thresholds.items():
            # Get observations relevant to this threshold
```

```
            relevant_obs =
self._get_relevant_observations(observations, threshold_name)

            if relevant_obs:
                # Calculate performance metrics
                metrics =
self._calculate_threshold_metrics(relevant_obs, threshold_value)
                evaluation_results[threshold_name] = metrics

                # Store in history
                self.performance_history[threshold_name].append({
                    'timestamp': datetime.now(),
                    'threshold_value': threshold_value,
                    'performance': metrics['overall_score']
                })

        return evaluation_results

    def adapt_thresholds(self, evaluation_results: Dict):
        """Adapt thresholds based on evaluation results"""

        adaptations_made = {}

        for threshold_name, metrics in evaluation_results.items():
            if threshold_name in self.adaptation_rules:
                adaptation_rule = self.adaptation_rules[threshold_name]

                # Check if we should adapt this threshold
                if self._should_adapt_threshold(threshold_name):
                    # Calculate adaptation
                    delta = self._calculate_adaptation_delta(metrics,
adaptation_rule)

                    # Apply adaptation
                    old_value = self.thresholds[threshold_name]
                    new_value = np.clip(
                        old_value + delta,
                        adaptation_rule['min_value'],
                        adaptation_rule['max_value']
                    )

                    self.thresholds[threshold_name] = new_value
```

```
                    adaptations_made[threshold_name] = {
                        'old_value': old_value,
                        'new_value': new_value,
                        'delta': delta,
                        'reason': self._get_adaptation_reason(metrics)
                    }

        return adaptations_made

    def _get_relevant_observations(self, observations:
List[MEVObservation],
                                   threshold_name: str) ->
List[MEVObservation]:
        """Get observations relevant to a specific threshold"""

        # Map threshold names to observation criteria
        threshold_criteria = {
            'arbitrage_threshold': lambda obs: obs.action_taken ==
'arbitrage',
            'liquidation_threshold': lambda obs: obs.action_taken ==
'liquidation',
            'risk_threshold': lambda obs:
obs.risk_metrics.get('overall_risk', 0) > 0.5,
            'gas_threshold': lambda obs:
obs.market_conditions.get('gas_price', 0) > 50
        }

        criterion = threshold_criteria.get(threshold_name, lambda obs:
True)
        return [obs for obs in observations if criterion(obs)]

    def _calculate_threshold_metrics(self, observations:
List[MEVObservation],
                                     threshold_value: float) -> Dict:
        """Calculate performance metrics for a threshold"""

        if not observations:
            return {'overall_score': 0, 'count': 0}

        # Calculate success rate
        successful_actions = sum(1 for obs in observations if
```

```
obs.reward > 0)
        success_rate = successful_actions / len(observations)

        # Calculate average profit/loss
        avg_profit = np.mean([obs.profit_loss for obs in observations])

        # Calculate risk-adjusted return
        avg_risk = np.mean([obs.risk_metrics.get('overall_risk', 0.5)
for obs in observations])
        risk_adjusted_return = avg_profit / (avg_risk + 0.01)  # Avoid
division by zero

        # Calculate threshold efficiency
        threshold_efficiency =
self._calculate_threshold_efficiency(observations, threshold_value)

        # Overall score (weighted combination)
        overall_score = (
            0.4 * success_rate +
            0.3 * risk_adjusted_return +
            0.2 * threshold_efficiency +
            0.1 * (len(observations) / 100)  # Volume bonus
        )

        return {
            'success_rate': success_rate,
            'avg_profit': avg_profit,
            'risk_adjusted_return': risk_adjusted_return,
            'threshold_efficiency': threshold_efficiency,
            'overall_score': overall_score,
            'count': len(observations)
        }

    def _calculate_threshold_efficiency(self, observations:
List[MEVObservation],
                                        threshold_value: float) -> float:
        """Calculate how efficiently the threshold captures
opportunities"""

        # Count opportunities that met the threshold
        opportunities_captured = sum(1 for obs in observations if
obs.reward > 0)
```

```
        # Count total potential opportunities
        total_opportunities = len(observations)

        # Calculate precision and recall
        precision = opportunities_captured / total_opportunities if
total_opportunities > 0 else 0

        # F1 score (harmonic mean of precision and recall)
        f1_score = 2 * precision * precision / (2 * precision) if
precision > 0 else 0

        return f1_score

    def _should_adapt_threshold(self, threshold_name: str) -> bool:
        """Determine if threshold should be adapted"""

        rule = self.adaptation_rules[threshold_name]

        # Check adaptation frequency
        if len(self.performance_history[threshold_name]) <
rule['adaptation_frequency']:
            return False

        # Check if we have sufficient recent data
        recent_performance = self.performance_history[threshold_name][-
rule['performance_window']:]

        return len(recent_performance) >= rule['performance_window']

    def _calculate_adaptation_delta(self, metrics: Dict, rule: Dict) ->
float:
        """Calculate adaptation delta for threshold"""

        # Adaptation based on performance score
        performance_score = metrics['overall_score']

        # Target performance (can be tuned)
        target_performance = 0.7

        # Calculate error
        error = target_performance - performance_score
```

```
        # Adaptation delta
        delta = rule['learning_rate'] * error

        return delta

    def _get_adaptation_reason(self, metrics: Dict) -> str:
        """Get reason for adaptation"""

        if metrics['overall_score'] > 0.7:
            return "performance_above_target"
        elif metrics['success_rate'] < 0.5:
            return "low_success_rate"
        elif metrics['risk_adjusted_return'] < 0:
            return "negative_risk_adjusted_return"
        else:
            return "general_optimization"

    def get_adaptation_summary(self) -> Dict:
        """Get summary of threshold adaptations"""

        summary = {
            'current_thresholds': self.thresholds.copy(),
            'adaptation_count': {},
            'recent_performance': {},
            'recommendations': []
        }

        for threshold_name in self.thresholds.keys():
            if threshold_name in self.performance_history:
                history = self.performance_history[threshold_name]

                # Count adaptations
                adaptations = 0
                if len(history) > 1:
                    for i in range(1, len(history)):
                        if history[i]['threshold_value'] !=
history[i-1]['threshold_value']:
                            adaptations += 1

                summary['adaptation_count'][threshold_name] =
adaptations
```

```
                # Recent performance
                recent_perf = [h['performance'] for h in history[-10:]]
if len(history) >= 10 else [h['performance'] for h in history]
                summary['recent_performance'][threshold_name] = {
                    'mean': np.mean(recent_perf),
                    'std': np.std(recent_perf),
                    'latest': recent_perf[-1] if recent_perf else 0
                }

        return summary
```

Multi-Armed Bandits

Thompson Sampling for MEV Strategy Selection

```
from scipy import stats
import random
from typing import Dict, List

class MEVThompsonSampler:
    """Thompson Sampling for MEV strategy selection"""

    def __init__(self, strategies: List[str], prior_alpha: float = 1.0,
prior_beta: float = 1.0):
        self.strategies = strategies
        self.prior_alpha = prior_alpha
        self.prior_beta = prior_beta

        # Initialize Beta distributions for each strategy
        self.beta_parameters = {
            strategy: {'alpha': prior_alpha, 'beta': prior_beta}
            for strategy in strategies
        }

        # Track performance
        self.arm_pulls = {strategy: 0 for strategy in strategies}
        self.arm_rewards = {strategy: [] for strategy in strategies}
        self.total_reward = 0
        self.total_pulls = 0

        # Adaptation parameters
        self.exploration_factor = 1.0
        self.decay_rate = 0.99
        self.min_exploration = 0.1

    def select_strategy(self) -> str:
        """Select strategy using Thompson Sampling"""

        # Sample from each strategy's Beta distribution
        samples = {}

        for strategy in self.strategies:
            alpha = self.beta_parameters[strategy]['alpha']
            beta = self.beta_parameters[strategy]['beta']

            # Sample from Beta distribution
            sample = np.random.beta(alpha, beta)
```

```
            samples[strategy] = sample

        # Select strategy with highest sample
        selected_strategy = max(samples, key=samples.get)

        return selected_strategy

    def update_strategy_reward(self, strategy: str, reward: float):
        """Update strategy reward using Bayesian inference"""

        # Increment pull count
        self.arm_pulls[strategy] += 1
        self.arm_rewards[strategy].append(reward)

        # Update Beta parameters (conjugate prior)
        success_reward = 1 if reward > 0 else 0

        self.beta_parameters[strategy]['alpha'] += success_reward
        self.beta_parameters[strategy]['beta'] += (1 - success_reward)

        # Update total statistics
        self.total_reward += reward
        self.total_pulls += 1

        # Adapt exploration factor
        self.exploration_factor = max(
            self.exploration_factor * self.decay_rate,
            self.min_exploration
        )

    def get_strategy_statistics(self) -> Dict:
        """Get current statistics for all strategies"""

        statistics = {}

        for strategy in self.strategies:
            pulls = self.arm_pulls[strategy]

            if pulls > 0:
                alpha = self.beta_parameters[strategy]['alpha']
                beta = self.beta_parameters[strategy]['beta']
```

```
                # Calculate posterior mean
                posterior_mean = alpha / (alpha + beta)

                # Calculate confidence interval
                ci_lower = stats.beta.ppf(0.025, alpha, beta)
                ci_upper = stats.beta.ppf(0.975, alpha, beta)

                # Calculate win rate
                win_rate = sum(1 for r in self.arm_rewards[strategy] if
r > 0) / pulls

                statistics[strategy] = {
                    'pulls': pulls,
                    'posterior_mean': posterior_mean,
                    'confidence_interval': (ci_lower, ci_upper),
                    'win_rate': win_rate,
                    'total_reward': sum(self.arm_rewards[strategy]),
                    'average_reward':
np.mean(self.arm_rewards[strategy]),
                    'exploration_score':
self._calculate_exploration_score(strategy)
                }
            else:
                statistics[strategy] = {
                    'pulls': 0,
                    'posterior_mean': 0.5,
                    'confidence_interval': (0.0, 1.0),
                    'win_rate': 0.0,
                    'total_reward': 0,
                    'average_reward': 0,
                    'exploration_score': 1.0
                }

        return statistics

    def _calculate_exploration_score(self, strategy: str) -> float:
        """Calculate exploration score for strategy"""

        # Based on number of pulls and uncertainty
        pulls = self.arm_pulls[strategy]

        if pulls == 0:
```

```
            return 1.0  # Maximum exploration

        # Uncertainty decreases with more pulls
        uncertainty = 1.0 / (1.0 + pulls * 0.1)

        return uncertainty * self.exploration_factor

    def plot_strategy_performance(self, save_path: str = None):
        """Plot strategy performance over time"""

        setup_matplotlib_for_plotting()

        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # Win rates over time
        for strategy in self.strategies:
            if self.arm_pulls[strategy] > 0:
                rewards = self.arm_rewards[strategy]
                win_rates = []

                for i in range(1, len(rewards) + 1):
                    recent_rewards = rewards[:i]
                    win_rate = sum(1 for r in recent_rewards if r >
0) / len(recent_rewards)
                    win_rates.append(win_rate)

                axes[0, 0].plot(win_rates, label=strategy)

        axes[0, 0].set_title('Win Rates Over Time')
        axes[0, 0].set_xlabel('Number of Trials')
        axes[0, 0].set_ylabel('Win Rate')
        axes[0, 0].legend()
        axes[0, 0].grid(True)

        # Cumulative rewards
        cumulative_rewards = {strategy: np.cumsum(rewards) for
strategy, rewards in self.arm_rewards.items()}

        for strategy, cum_reward in cumulative_rewards.items():
            axes[0, 1].plot(cum_reward, label=strategy)

        axes[0, 1].set_title('Cumulative Rewards')
```

```
        axes[0, 1].set_xlabel('Number of Trials')
        axes[0, 1].set_ylabel('Cumulative Reward')
        axes[0, 1].legend()
        axes[0, 1].grid(True)

        # Number of pulls per strategy
        strategies = list(self.strategies)
        pulls = [self.arm_pulls[s] for s in strategies]

        axes[1, 0].bar(strategies, pulls)
        axes[1, 0].set_title('Number of Strategy Selections')
        axes[1, 0].set_xlabel('Strategy')
        axes[1, 0].set_ylabel('Number of Pulls')
        axes[1, 0].tick_params(axis='x', rotation=45)

        # Posterior distributions
        x = np.linspace(0, 1, 1000)

        for strategy in self.strategies[:3]:  # Plot first 3 strategies
            alpha = self.beta_parameters[strategy]['alpha']
            beta = self.beta_parameters[strategy]['beta']

            y = stats.beta.pdf(x, alpha, beta)
            axes[1, 1].plot(x, y, label=strategy)

        axes[1, 1].set_title('Posterior Beta Distributions')
        axes[1, 1].set_xlabel('Success Probability')
        axes[1, 1].set_ylabel('Density')
        axes[1, 1].legend()

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.close()

    def reset(self):
        """Reset the Thompson Sampler"""

        for strategy in self.strategies:
            self.arm_pulls[strategy] = 0
```

```
            self.arm_rewards[strategy] = []
            self.beta_parameters[strategy] = {
                'alpha': self.prior_alpha,
                'beta': self.prior_beta
            }

        self.total_reward = 0
        self.total_pulls = 0
        self.exploration_factor = 1.0

class ContextualThompsonSampler:
    """Contextual Thompson Sampling for MEV with market conditions"""

    def __init__(self, strategies: List[str], context_dim: int):
        self.strategies = strategies
        self.context_dim = context_dim

        # Initialize parameters for each strategy-context combination
        self.strategy_params = {
            strategy: {
                'weights': np.random.normal(0, 0.1, context_dim),
                'precision': 1.0,
                'bias': 0.0
            }
            for strategy in strategies
        }

        # Hyperparameters for Bayesian linear regression
        self.prior_precision = 1.0
        self.likelihood_precision = 1.0

        # Track performance
        self.observations = []  # (context, strategy, reward)

    def select_strategy(self, context: np.ndarray) -> Tuple[str, Dict]:
        """Select strategy based on context using Thompson Sampling"""

        # Sample from posterior for each strategy
        samples = {}
        sample_info = {}

        for strategy in self.strategies:
```

```
            sample, info = self._sample_from_posterior(strategy,
context)
            samples[strategy] = sample
            sample_info[strategy] = info

        # Select strategy with highest sample
        selected_strategy = max(samples, key=samples.get)

        return selected_strategy, sample_info[selected_strategy]

    def _sample_from_posterior(self, strategy: str, context:
np.ndarray) -> Tuple[float, Dict]:
        """Sample from posterior distribution for strategy given
context"""

        params = self.strategy_params[strategy]

        # Sample from posterior distribution
        # For simplicity, using a normal approximation
        predicted_reward = np.dot(params['weights'], context) +
params['bias']

        # Add uncertainty
        uncertainty = 1.0 / params['precision']
        sampled_reward = np.random.normal(predicted_reward,
uncertainty)

        return sampled_reward, {
            'predicted_reward': predicted_reward,
            'uncertainty': uncertainty,
            'sampled_reward': sampled_reward,
            'context': context.copy(),
            'strategy_weights': params['weights'].copy()
        }

    def update_observation(self, context: np.ndarray, strategy: str,
reward: float):
        """Update model with new observation"""

        params = self.strategy_params[strategy]

        # Update weights using Bayesian linear regression
```

```
# precision_matrix = prior_precision * I + likelihood_precision *
outer(context, context)
        precision_matrix = self.prior_precision *
np.eye(self.context_dim) + \
                           self.likelihood_precision * np.outer(context,
context)

        # precision_vector = prior_precision * prior_mean +
likelihood_precision * reward * context
        prior_mean = np.zeros(self.context_dim)
        precision_vector = self.prior_precision * prior_mean + \
                           self.likelihood_precision * reward * context

        # Solve for posterior precision and mean
        posterior_precision = precision_matrix
        posterior_mean = np.linalg.solve(posterior_precision,
precision_vector)

        # Update parameters
        params['weights'] = posterior_mean
        params['precision'] = np.trace(posterior_precision)

        # Store observation
        self.observations.append({
            'context': context.copy(),
            'strategy': strategy,
            'reward': reward,
            'timestamp': datetime.now()
        })

    def get_strategy_context_performance(self, strategy: str,
context_range: Dict) -> Dict:
        """Get performance of strategy across different contexts"""

        if strategy not in self.strategy_params:
            return {}

        params = self.strategy_params[strategy]

        # Sample contexts in the specified range
        contexts = []
```

```
        performance_predictions = []

        for _ in range(100):  # Sample 100 contexts
            context = np.random.uniform(
                low=context_range.get('low', [0] * self.context_dim),
                high=context_range.get('high', [1] * self.context_dim)
            )
            contexts.append(context)

            predicted_performance = np.dot(params['weights'], context)
+ params['bias']
            performance_predictions.append(predicted_performance)

        return {
            'mean_performance': np.mean(performance_predictions),
            'std_performance': np.std(performance_predictions),
            'min_performance': np.min(performance_predictions),
            'max_performance': np.max(performance_predictions),
            'context_performance_pairs': list(zip(contexts,
performance_predictions))
        }
```

Reinforcement Learning for MEV

Q-Learning for MEV Decision Making

```
import random
from collections import defaultdict, deque
from typing import Dict, Tuple

class QLearningMEVAgent:
    """Q-Learning agent for MEV decision making"""

    def __init__(self, state_space_size: int, action_space_size: int,
                 learning_rate: float = 0.1, discount_factor: float =
0.95,
                 epsilon: float = 0.1, epsilon_decay: float = 0.995,
                 epsilon_min: float = 0.01):

        self.state_space_size = state_space_size
        self.action_space_size = action_space_size
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min

        # Q-table: state -> action -> q_value
        self.q_table = defaultdict(lambda: defaultdict(float))

        # Experience replay for stable learning
        self.experience_replay = deque(maxlen=10000)
        self.batch_size = 32

        # Tracking
        self.episode_rewards = []
        self.episode_lengths = []
        self.current_episode_reward = 0
        self.current_episode_length = 0

    def get_action(self, state: int, training: bool = True) -> int:
        """Get action using epsilon-greedy policy"""

        if training and random.random() < self.epsilon:
            # Explore: random action
            return random.randint(0, self.action_space_size - 1)
        else:
            # Exploit: best action according to Q-table
```

```
            q_values = [self.q_table[state][action] for action in
range(self.action_space_size)]
            return np.argmax(q_values)

    def update_q_value(self, state: int, action: int, reward: float,
next_state: int):
        """Update Q-value using Q-learning update rule"""

        # Calculate target Q-value
        max_next_q = max([self.q_table[next_state][a] for a in
range(self.action_space_size)])
        target_q = reward + self.discount_factor * max_next_q

        # Update Q-value
        current_q = self.q_table[state][action]
        new_q = current_q + self.learning_rate * (target_q - current_q)
        self.q_table[state][action] = new_q

        # Add to experience replay
        self.experience_replay.append((state, action, reward,
next_state))

    def experience_replay_training(self):
        """Train agent using experience replay"""

        if len(self.experience_replay) < self.batch_size:
            return

        # Sample random batch from experience replay
        batch = random.sample(self.experience_replay, self.batch_size)

        for state, action, reward, next_state in batch:
            self.update_q_value(state, action, reward, next_state)

    def decay_epsilon(self):
        """Decay epsilon for exploration"""

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def reset_episode(self):
        """Reset episode tracking"""
```

```
        self.episode_rewards.append(self.current_episode_reward)
        self.episode_lengths.append(self.current_episode_length)

        self.current_episode_reward = 0
        self.current_episode_length = 0

        self.decay_epsilon()

    def get_q_table_statistics(self) -> Dict:
        """Get statistics about the Q-table"""

        states = list(self.q_table.keys())

        if not states:
            return {'message': 'No states visited yet'}

        # Calculate Q-value statistics
        all_q_values = []
        for state in states:
            for action in range(self.action_space_size):
                all_q_values.append(self.q_table[state][action])

        # Calculate state statistics
        state_visit_counts = {state: sum(1 for _ in
range(self.action_space_size))
                              for state in states}

        return {
            'total_states_visited': len(states),
            'total_q_values': len(all_q_values),
            'max_q_value': max(all_q_values) if all_q_values else 0,
            'min_q_value': min(all_q_values) if all_q_values else 0,
            'mean_q_value': np.mean(all_q_values) if all_q_values else
0,
            'std_q_value': np.std(all_q_values) if all_q_values else 0,
            'most_visited_state': max(state_visit_counts,
key=state_visit_counts.get) if state_visit_counts else None,
            'epsilon': self.epsilon,
            'total_episodes': len(self.episode_rewards)
        }
```

```
    def plot_learning_progress(self, save_path: str = None):
        """Plot learning progress"""

        setup_matplotlib_for_plotting()

        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # Episode rewards
        axes[0, 0].plot(self.episode_rewards)
        axes[0, 0].set_title('Episode Rewards')
        axes[0, 0].set_xlabel('Episode')
        axes[0, 0].set_ylabel('Total Reward')
        axes[0, 0].grid(True)

        # Episode lengths
        axes[0, 1].plot(self.episode_lengths)
        axes[0, 1].set_title('Episode Lengths')
        axes[0, 1].set_xlabel('Episode')
        axes[0, 1].set_ylabel('Steps')
        axes[0, 1].grid(True)

        # Epsilon decay
        epsilons = [self.epsilon * (self.epsilon_decay ** i) for i in
range(len(self.episode_rewards))]
        axes[1, 0].plot(epsilons)
        axes[1, 0].set_title('Epsilon Decay')
        axes[1, 0].set_xlabel('Episode')
        axes[1, 0].set_ylabel('Epsilon')
        axes[1, 0].grid(True)

        # Q-value distribution
        all_q_values = []
        for state in self.q_table:
            for action in range(self.action_space_size):
                all_q_values.append(self.q_table[state][action])

        if all_q_values:
            axes[1, 1].hist(all_q_values, bins=30, alpha=0.7)
            axes[1, 1].set_title('Q-Value Distribution')
            axes[1, 1].set_xlabel('Q-Value')
            axes[1, 1].set_ylabel('Frequency')
```

```
        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.close()

class DeepQLearningAgent:
    """Deep Q-Learning agent using neural networks"""

    def __init__(self, state_dim: int, action_dim: int, hidden_dims:
List[int] = [128, 64]):

        self.state_dim = state_dim
        self.action_dim = action_dim
        self.hidden_dims = hidden_dims

        # Neural network for Q-function approximation
        self.q_network = self._build_network()
        self.target_network = self._build_network()

        # Copy weights to target network
        self.update_target_network()

        # Training parameters
        self.learning_rate = 0.001
        self.discount_factor = 0.99
        self.epsilon = 1.0
        self.epsilon_decay = 0.995
        self.epsilon_min = 0.01

        # Experience replay
        self.memory = deque(maxlen=100000)
        self.batch_size = 32

        # Training tracking
        self.training_losses = []

    def _build_network(self):
        """Build neural network for Q-function approximation"""

        import torch
```

```
        import torch.nn as nn
        import torch.optim as optim

        layers = []
        input_dim = self.state_dim

        # Hidden layers
        for hidden_dim in self.hidden_dims:
            layers.append(nn.Linear(input_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))
            input_dim = hidden_dim

        # Output layer
        layers.append(nn.Linear(input_dim, self.action_dim))

        network = nn.Sequential(*layers)

        # Move to device
        device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
        network = network.to(device)

        return network

    def get_action(self, state: np.ndarray, training: bool = True) ->
int:
        """Get action using epsilon-greedy policy"""

        import torch

        if training and random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 1)
        else:
            # Convert state to tensor
            state_tensor = torch.FloatTensor(state).unsqueeze(0)
            device = next(self.q_network.parameters()).device
            state_tensor = state_tensor.to(device)

            # Get Q-values
            with torch.no_grad():
                q_values = self.q_network(state_tensor)
```

```
            return q_values.argmax().item()

    def store_experience(self, state: np.ndarray, action: int, reward:
float,
                         next_state: np.ndarray, done: bool):
        """Store experience in replay buffer"""

        self.memory.append((state, action, reward, next_state, done))

    def train_step(self):
        """Train the network using a batch of experiences"""

        import torch
        import torch.nn as nn
        import torch.optim as optim

        if len(self.memory) < self.batch_size:
            return

        # Sample random batch
        batch = random.sample(self.memory, self.batch_size)

        # Prepare batch
        states = torch.FloatTensor([e[0] for e in batch])
        actions = torch.LongTensor([e[1] for e in batch])
        rewards = torch.FloatTensor([e[2] for e in batch])
        next_states = torch.FloatTensor([e[3] for e in batch])
        dones = torch.BoolTensor([e[4] for e in batch])

        device = next(self.q_network.parameters()).device
        states = states.to(device)
        actions = actions.to(device)
        rewards = rewards.to(device)
        next_states = next_states.to(device)
        dones = dones.to(device)

        # Current Q-values
        current_q_values = self.q_network(states).gather(1,
actions.unsqueeze(1))

        # Next Q-values from target network
```

```
        with torch.no_grad():
            next_q_values = self.target_network(next_states).max(1)[0]
            target_q_values = rewards + (self.discount_factor *
next_q_values * ~dones)

        # Loss
        loss = nn.MSELoss()(current_q_values.squeeze(),
target_q_values)

        # Optimize
        optimizer = optim.Adam(self.q_network.parameters(),
lr=self.learning_rate)
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(self.q_network.parameters(),
1.0)
        optimizer.step()

        self.training_losses.append(loss.item())

        # Decay epsilon
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def update_target_network(self):
        """Update target network weights"""


self.target_network.load_state_dict(self.q_network.state_dict())

    def save_model(self, filepath: str):
        """Save model weights"""

        torch.save({
            'q_network_state_dict': self.q_network.state_dict(),
            'target_network_state_dict':
self.target_network.state_dict(),
            'epsilon': self.epsilon
        }, filepath)

    def load_model(self, filepath: str):
        """Load model weights"""
```

```
        checkpoint = torch.load(filepath, map_location='cpu')

self.q_network.load_state_dict(checkpoint['q_network_state_dict'])

self.target_network.load_state_dict(checkpoint['target_network_state_dict'])
        self.epsilon = checkpoint.get('epsilon', 1.0)

class PolicyGradientAgent:
    """Policy Gradient agent for MEV strategy optimization"""

    def __init__(self, state_dim: int, action_dim: int, hidden_dims:
List[int] = [128, 64]):

        self.state_dim = state_dim
        self.action_dim = action_dim

        # Policy network (outputs action probabilities)
        self.policy_network = self._build_policy_network()

        # Value network (estimates state value)
        self.value_network = self._build_value_network()

        # Training parameters
        self.learning_rate = 0.0003
        self.gamma = 0.99

        # Tracking
        self.episode_log_probs = []
        self.episode_rewards = []
        self.episode_values = []

    def _build_policy_network(self):
        """Build policy network"""

        import torch
        import torch.nn as nn

        layers = []
        input_dim = self.state_dim

        # Hidden layers
```

```
        for hidden_dim in self.hidden_dims:
            layers.append(nn.Linear(input_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))
            input_dim = hidden_dim

        # Output layer (action probabilities)
        layers.append(nn.Linear(input_dim, self.action_dim))
        layers.append(nn.Softmax(dim=-1))

        return nn.Sequential(*layers)

    def _build_value_network(self):
        """Build value network"""

        import torch
        import torch.nn as nn

        layers = []
        input_dim = self.state_dim

        # Hidden layers
        for hidden_dim in self.hidden_dims:
            layers.append(nn.Linear(input_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))
            input_dim = hidden_dim

        # Output layer (state value)
        layers.append(nn.Linear(input_dim, 1))

        return nn.Sequential(*layers)

    def get_action(self, state: np.ndarray) -> Tuple[int, float,
float]:
        """Get action with log probability and value estimate"""

        import torch

        # Convert state to tensor
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        device = next(self.policy_network.parameters()).device
```

```
        state_tensor = state_tensor.to(device)

        # Get action probabilities
        with torch.no_grad():
            action_probs = self.policy_network(state_tensor)
            value_estimate = self.value_network(state_tensor)

        # Sample action
        action_dist = torch.distributions.Categorical(action_probs)
        action = action_dist.sample()
        log_prob = action_dist.log_prob(action)

        return action.item(), log_prob.item(), value_estimate.item()

    def update_policy(self, rewards: List[float], log_probs:
List[float],
                      values: List[float]):
        """Update policy using REINFORCE algorithm"""

        import torch
        import torch.nn as nn
        import torch.optim as optim

        # Calculate returns and advantages
        returns = self._calculate_returns(rewards)
        advantages = self._calculate_advantages(rewards, values)

        # Convert to tensors
        log_probs_tensor = torch.tensor(log_probs, requires_grad=True)
        advantages_tensor = torch.tensor(advantages,
requires_grad=True)

        # Policy loss (negative because we maximize)
        policy_loss = -torch.sum(log_probs_tensor * advantages_tensor)

        # Value loss
        returns_tensor = torch.tensor(returns, requires_grad=True)
        value_estimates = torch.tensor(values, requires_grad=True)
        value_loss = nn.MSELoss()(value_estimates, returns_tensor)

        # Total loss
        total_loss = policy_loss + 0.5 * value_loss
```

```
        # Optimize
        optimizer = optim.Adam(
            list(self.policy_network.parameters()) +
list(self.value_network.parameters()),
            lr=self.learning_rate
        )

        optimizer.zero_grad()
        total_loss.backward()

torch.nn.utils.clip_grad_norm_(self.policy_network.parameters(), 1.0)
        torch.nn.utils.clip_grad_norm_(self.value_network.parameters(),
1.0)
        optimizer.step()

        return policy_loss.item(), value_loss.item()

    def _calculate_returns(self, rewards: List[float], gamma: float =
None) -> List[float]:
        """Calculate discounted returns"""

        if gamma is None:
            gamma = self.gamma

        returns = []
        G = 0

        for reward in reversed(rewards):
            G = reward + gamma * G
            returns.insert(0, G)

        return returns

    def _calculate_advantages(self, rewards: List[float], values:
List[float],
                              gamma: float = None) -> List[float]:
        """Calculate advantage estimates"""

        if gamma is None:
            gamma = self.gamma
```

```
        returns = self._calculate_returns(rewards, gamma)
        advantages = [r - v for r, v in zip(returns, values)]

        return advantages
```

Evolutionary Algorithms

Genetic Algorithm for Parameter Optimization

```
from typing import Callable, List
import copy

class GeneticAlgorithmOptimizer:
    """Genetic Algorithm for MEV parameter optimization"""

    def __init__(self, parameter_bounds: Dict[str, Tuple[float,
float]],
                 population_size: int = 100, generations: int = 100,
                 mutation_rate: float = 0.1, crossover_rate: float =
0.8,
                 elitism_rate: float = 0.1):

        self.parameter_bounds = parameter_bounds
        self.parameter_names = list(parameter_bounds.keys())
        self.param_count = len(self.parameter_names)

        # GA parameters
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.elitism_rate = elitism_rate

        # Tracking
        self.population_history = []
        self.fitness_history = []
        self.best_individual = None
        self.best_fitness = float('-inf')

        # Initialize population
        self.population = self._initialize_population()

    def _initialize_population(self) -> List[np.ndarray]:
        """Initialize random population within bounds"""

        population = []

        for _ in range(self.population_size):
            individual = np.zeros(self.param_count)

            for i, param_name in enumerate(self.parameter_names):
```

```
                min_val, max_val = self.parameter_bounds[param_name]
                individual[i] = np.random.uniform(min_val, max_val)

            population.append(individual)

        return population

    def _evaluate_fitness(self, individual: np.ndarray,
fitness_function: Callable) -> float:
        """Evaluate fitness of an individual"""

        # Convert numpy array to parameter dictionary
        params = {self.parameter_names[i]: individual[i] for i in
range(self.param_count)}

        try:
            fitness = fitness_function(params)
            return fitness
        except Exception as e:
            print(f"Error evaluating fitness: {e}")
            return 0.0

    def _selection(self, fitnesses: List[float]) -> int:
        """Select individual using tournament selection"""

        tournament_size = 3
        tournament_indices = np.random.choice(len(self.population),
tournament_size, replace=False)
        tournament_fitnesses = [fitnesses[i] for i in
tournament_indices]

        winner_index =
tournament_indices[np.argmax(tournament_fitnesses)]
        return winner_index

    def _crossover(self, parent1: np.ndarray, parent2: np.ndarray) ->
Tuple[np.ndarray, np.ndarray]:
        """Perform crossover between two parents"""

        if np.random.random() > self.crossover_rate:
            return parent1.copy(), parent2.copy()
```

```
        # Single-point crossover
        crossover_point = np.random.randint(1, self.param_count)

        child1 = np.concatenate([parent1[:crossover_point],
parent2[crossover_point:]])
        child2 = np.concatenate([parent2[:crossover_point],
parent1[crossover_point:]])

        return child1, child2

    def _mutate(self, individual: np.ndarray) -> np.ndarray:
        """Mutate an individual"""

        mutated = individual.copy()

        for i in range(self.param_count):
            if np.random.random() < self.mutation_rate:
                # Gaussian mutation
                param_name = self.parameter_names[i]
                min_val, max_val = self.parameter_bounds[param_name]

                # Add Gaussian noise
                noise = np.random.normal(0, (max_val - min_val) * 0.1)
                mutated[i] += noise

                # Clip to bounds
                mutated[i] = np.clip(mutated[i], min_val, max_val)

        return mutated

    def evolve(self, fitness_function: Callable) -> Dict:
        """Evolve the population"""

        print(f"Starting evolution with population size
{self.population_size}")

        for generation in range(self.generations):
            # Evaluate fitness for all individuals
            fitnesses = []
            for individual in self.population:
                fitness = self._evaluate_fitness(individual,
fitness_function)
```

```
            fitnesses.append(fitness)

        # Track best individual
        max_fitness = max(fitnesses)
        if max_fitness > self.best_fitness:
            self.best_fitness = max_fitness
            best_idx = fitnesses.index(max_fitness)
            self.best_individual = self.population[best_idx].copy()

        # Store generation statistics
        self.fitness_history.append({
            'generation': generation,
            'best_fitness': max_fitness,
            'mean_fitness': np.mean(fitnesses),
            'std_fitness': np.std(fitnesses),
            'worst_fitness': min(fitnesses)
        })

        # Create new population
        new_population = []

        # Elitism: keep best individuals
        elite_count = int(self.population_size * self.elitism_rate)
        elite_indices = np.argsort(fitnesses)[-elite_count:]
        for idx in elite_indices:
            new_population.append(self.population[idx].copy())

        # Generate offspring
        while len(new_population) < self.population_size:
            # Selection
            parent1_idx = self._selection(fitnesses)
            parent2_idx = self._selection(fitnesses)

            parent1 = self.population[parent1_idx]
            parent2 = self.population[parent2_idx]

            # Crossover
            child1, child2 = self._crossover(parent1, parent2)

            # Mutation
            child1 = self._mutate(child1)
            child2 = self._mutate(child2)
```

```
                new_population.extend([child1, child2])

            # Ensure population size
            self.population = new_population[:self.population_size]

            # Print progress
            if generation % 10 == 0 or generation == self.generations -
1:
                print(f"Generation {generation}: Best fitness =
{max_fitness:.6f}, "
                      f"Mean fitness = {np.mean(fitnesses):.6f}")

        # Final evaluation
        final_fitnesses = []
        for individual in self.population:
            fitness = self._evaluate_fitness(individual,
fitness_function)
            final_fitnesses.append(fitness)

        final_best_idx = np.argmax(final_fitnesses)
        final_best_fitness = final_fitnesses[final_best_idx]
        final_best_individual = self.population[final_best_idx]

        if final_best_fitness > self.best_fitness:
            self.best_fitness = final_best_fitness
            self.best_individual = final_best_individual.copy()

        return self._get_optimization_results()

    def _get_optimization_results(self) -> Dict:
        """Get optimization results"""

        # Convert best individual to parameter dictionary
        best_params = {self.parameter_names[i]: self.best_individual[i]
                       for i in range(self.param_count)}

        # Calculate convergence metrics
        fitness_values = [gen['best_fitness'] for gen in
self.fitness_history]

        # Check if algorithm converged
```

```
        if len(fitness_values) > 10:
            recent_improvement = fitness_values[-1] -
fitness_values[-10]
            converged = recent_improvement < 1e-6
        else:
            converged = False

        return {
            'best_parameters': best_params,
            'best_fitness': self.best_fitness,
            'generations_run': len(self.fitness_history),
            'converged': converged,
            'final_population_fitness': {
                'mean': np.mean([gen['mean_fitness'] for gen in
self.fitness_history[-1:]]),
                'std': np.std([gen['std_fitness'] for gen in
self.fitness_history[-1:]])
            },
            'convergence_analysis': {
                'fitness_improvement': fitness_values[-1] -
fitness_values[0] if len(fitness_values) > 1 else 0,
                'stagnation_generations':
self._count_stagnation_generations()
            }
        }

    def _count_stagnation_generations(self) -> int:
        """Count consecutive generations without significant
improvement"""

        if len(self.fitness_history) < 10:
            return 0

        stagnation_count = 0
        threshold = 1e-6

        for i in range(len(self.fitness_history) - 1, 0, -1):
            current_fitness = self.fitness_history[i]['best_fitness']
            previous_fitness = self.fitness_history[i-1]
['best_fitness']

            if current_fitness - previous_fitness < threshold:
```

```
                stagnation_count += 1
            else:
                break

        return stagnation_count

    def plot_evolution_progress(self, save_path: str = None):
        """Plot evolution progress"""

        setup_matplotlib_for_plotting()

        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        generations = [gen['generation'] for gen in
self.fitness_history]
        best_fitnesses = [gen['best_fitness'] for gen in
self.fitness_history]
        mean_fitnesses = [gen['mean_fitness'] for gen in
self.fitness_history]
        std_fitnesses = [gen['std_fitness'] for gen in
self.fitness_history]

        # Best and mean fitness over generations
        axes[0, 0].plot(generations, best_fitnesses, 'b-', label='Best
Fitness', linewidth=2)
        axes[0, 0].plot(generations, mean_fitnesses, 'r--',
label='Mean Fitness')
        axes[0, 0].fill_between(generations,
                                [m - s for m, s in zip(mean_fitnesses,
std_fitnesses)],
                                [m + s for m, s in zip(mean_fitnesses,
std_fitnesses)],
                                alpha=0.3, color='red')
        axes[0, 0].set_title('Fitness Evolution')
        axes[0, 0].set_xlabel('Generation')
        axes[0, 0].set_ylabel('Fitness')
        axes[0, 0].legend()
        axes[0, 0].grid(True)

        # Parameter evolution (if we have the history)
        if hasattr(self, 'population_history') and
self.population_history:
```

```
            param_history = np.array(self.population_history)

            for i in range(min(param_history.shape[2], 5)):  # Plot
first 5 parameters
                param_means = np.mean(param_history[:, :, i], axis=1)
                axes[0, 1].plot(generations, param_means,
label=f'Param {i}')

            axes[0, 1].set_title('Parameter Evolution')
            axes[0, 1].set_xlabel('Generation')
            axes[0, 1].set_ylabel('Parameter Value')
            axes[0, 1].legend()

        # Fitness distribution in final generation
        final_fitnesses = []
        for individual in self.population:
            # Re-evaluate fitness for display
            fitness = self._evaluate_fitness(individual,
self.fitness_function if hasattr(self, 'fitness_function') else lambda
x: 0)
            final_fitnesses.append(fitness)

        axes[1, 0].hist(final_fitnesses, bins=20, alpha=0.7,
edgecolor='black')
        axes[1, 0].axvline(self.best_fitness, color='red',
linestyle='--',
                           label=f'Best: {self.best_fitness:.6f}')
        axes[1, 0].set_title('Final Population Fitness Distribution')
        axes[1, 0].set_xlabel('Fitness')
        axes[1, 0].set_ylabel('Count')
        axes[1, 0].legend()

        # Convergence analysis
        if len(fitness_values) > 1:
            fitness_improvements = [best_fitnesses[i] -
best_fitnesses[i-1]
                                    for i in range(1,
len(best_fitnesses))]
            axes[1, 1].plot(generations[1:], fitness_improvements)
            axes[1, 1].set_title('Fitness Improvement per Generation')
            axes[1, 1].set_xlabel('Generation')
            axes[1, 1].set_ylabel('Improvement')
```

```
            axes[1, 1].grid(True)

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.close()

    def set_fitness_function(self, fitness_function: Callable):
        """Set the fitness function for evaluation"""
        self.fitness_function = fitness_function

class DifferentialEvolution:
    """Differential Evolution optimizer for MEV parameters"""

    def __init__(self, parameter_bounds: Dict[str, Tuple[float,
float]],
                 population_size: int = 50, generations: int = 100,
                 mutation_factor: float = 0.8, crossover_probability:
float = 0.7):

        self.parameter_bounds = parameter_bounds
        self.parameter_names = list(parameter_bounds.keys())
        self.param_count = len(self.parameter_names)

        # DE parameters
        self.population_size = population_size
        self.generations = generations
        self.mutation_factor = mutation_factor
        self.crossover_probability = crossover_probability

        # Initialize population
        self.population = self._initialize_population()

        # Tracking
        self.best_solution = None
        self.best_fitness = float('-inf')
        self.fitness_history = []

    def _initialize_population(self) -> List[np.ndarray]:
        """Initialize population randomly within bounds"""
```

```
        population = []

        for _ in range(self.population_size):
            individual = np.zeros(self.param_count)

            for i, param_name in enumerate(self.parameter_names):
                min_val, max_val = self.parameter_bounds[param_name]
                individual[i] = np.random.uniform(min_val, max_val)

            population.append(individual)

        return population

    def _evaluate_fitness(self, individual: np.ndarray,
fitness_function: Callable) -> float:
        """Evaluate fitness of an individual"""

        params = {self.parameter_names[i]: individual[i] for i in
range(self.param_count)}

        try:
            return fitness_function(params)
        except Exception as e:
            print(f"Error evaluating fitness: {e}")
            return 0.0

    def _mutation(self, target_index: int) -> np.ndarray:
        """Create mutant vector using current-to-best strategy"""

        # Select three random individuals (different from target)
        candidates = list(range(self.population_size))
        candidates.remove(target_index)

        if len(candidates) < 3:
            candidates = np.random.choice(candidates, min(3,
len(candidates)), replace=False)
        else:
            candidates = np.random.choice(candidates, 3, replace=False)

        # Current-to-best mutation: V = X_target + F * (X_best -
X_current) + F * (X_r1 - X_r2)
```

```
        target = self.population[target_index]
        best = self.best_solution
        r1, r2 = candidates[0], candidates[1]

        mutant = target + self.mutation_factor * (best -
self.population[r1]) + \
                 self.mutation_factor * (self.population[r1] -
self.population[r2])

        return mutant

    def _crossover(self, target: np.ndarray, mutant: np.ndarray) ->
np.ndarray:
        """Perform binomial crossover"""

        trial = target.copy()

        for i in range(self.param_count):
            if np.random.random() <= self.crossover_probability or i ==
np.random.randint(self.param_count):
                trial[i] = mutant[i]

        # Clip to bounds
        for i, param_name in enumerate(self.parameter_names):
            min_val, max_val = self.parameter_bounds[param_name]
            trial[i] = np.clip(trial[i], min_val, max_val)

        return trial

    def evolve(self, fitness_function: Callable) -> Dict:
        """Run Differential Evolution optimization"""

        print(f"Starting Differential Evolution with population size
{self.population_size}")

        # Evaluate initial population
        fitnesses = []
        for individual in self.population:
            fitness = self._evaluate_fitness(individual,
fitness_function)
            fitnesses.append(fitness)
```

```
        # Find best initial solution
        best_idx = np.argmax(fitnesses)
        self.best_solution = self.population[best_idx].copy()
        self.best_fitness = fitnesses[best_idx]

        # Evolution loop
        for generation in range(self.generations):
            new_population = []
            new_fitnesses = []

            for i in range(self.population_size):
                # Mutation
                mutant = self._mutation(i)

                # Crossover
                trial = self._crossover(self.population[i], mutant)

                # Evaluate trial vector
                trial_fitness = self._evaluate_fitness(trial,
fitness_function)

                # Selection
                if trial_fitness > fitnesses[i]:
                    new_population.append(trial)
                    new_fitnesses.append(trial_fitness)

                    # Update best solution if needed
                    if trial_fitness > self.best_fitness:
                        self.best_solution = trial.copy()
                        self.best_fitness = trial_fitness
                else:
                    new_population.append(self.population[i])
                    new_fitnesses.append(fitnesses[i])

            # Update population
            self.population = new_population
            fitnesses = new_fitnesses

            # Record statistics
            mean_fitness = np.mean(fitnesses)
            std_fitness = np.std(fitnesses)
```

```
            self.fitness_history.append({
                'generation': generation,
                'best_fitness': self.best_fitness,
                'mean_fitness': mean_fitness,
                'std_fitness': std_fitness
            })

            # Print progress
            if generation % 10 == 0 or generation == self.generations -
1:
                print(f"Generation {generation}: Best fitness =
{self.best_fitness:.6f}, "
                      f"Mean fitness = {mean_fitness:.6f}")

        # Convert best solution to parameter dictionary
        best_params = {self.parameter_names[i]: self.best_solution[i]
                       for i in range(self.param_count)}

        return {
            'best_parameters': best_params,
            'best_fitness': self.best_fitness,
            'generations_run': len(self.fitness_history),
            'optimization_complete': True
        }
```

Parameter Optimization

Bayesian Optimization

```
from scipy.optimize import minimize
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, ConstantKernel

class BayesianOptimizer:
    """Bayesian Optimization for MEV parameter tuning"""

    def __init__(self, parameter_bounds: Dict[str, Tuple[float,
float]],
                 acquisition_function: str = 'expected_improvement',
                 n_initial_points: int = 10, random_state: int = 42):

        self.parameter_bounds = parameter_bounds
        self.parameter_names = list(parameter_bounds.keys())
        self.param_count = len(self.parameter_names)
        self.acquisition_function = acquisition_function

        # Initialize Gaussian Process
        kernel = ConstantKernel(1.0, (1e-3, 1e3)) *
Matern(length_scale=1.0, nu=2.5)
        self.gp = GaussianProcessRegressor(kernel=kernel,
random_state=random_state)

        # Data storage
        self.X_observed = []
        self.y_observed = []
        self.n_initial_points = n_initial_points

        # Optimization tracking
        self.optimization_history = []
        self.acquisition_history = []

    def _format_bounds(self) -> List[Tuple[float, float]]:
        """Format parameter bounds for optimizer"""
        return [self.parameter_bounds[name] for name in
self.parameter_names]

    def _sample_random_point(self) -> np.ndarray:
        """Sample random point within bounds"""
        point = np.zeros(self.param_count)

        for i, param_name in enumerate(self.parameter_names):
```

```
            min_val, max_val = self.parameter_bounds[param_name]
            point[i] = np.random.uniform(min_val, max_val)

        return point

    def optimize(self, objective_function: Callable, n_iterations: int
= 50) -> Dict:
        """Run Bayesian optimization"""

        print(f"Starting Bayesian Optimization for {n_iterations}
iterations")

        # Phase 1: Initial random sampling
        print(f"Phase 1: Initial random sampling
({self.n_initial_points} points)")

        for i in range(self.n_initial_points):
            # Sample random point
            x = self._sample_random_point()

            # Evaluate objective
            y = self._evaluate_objective(x, objective_function)

            # Store observation
            self.X_observed.append(x)
            self.y_observed.append(y)

            print(f"Initial point {i+1}/
{self.n_initial_points}: f(x) = {y:.6f}")

        # Phase 2: Bayesian optimization
        print(f"Phase 2: Bayesian optimization ({n_iterations}
iterations)")

        for iteration in range(n_iterations):
            # Fit Gaussian Process
            X_array = np.array(self.X_observed)
            y_array = np.array(self.y_observed)

            if len(X_array) > 1:  # Need at least 2 points to fit GP
                self.gp.fit(X_array, y_array)
```

```
            # Optimize acquisition function
            acquisition_result = self._optimize_acquisition()

            # Evaluate at suggested point
            x_new = acquisition_result['x']
            y_new = self._evaluate_objective(x_new, objective_function)

            # Store observation
            self.X_observed.append(x_new)
            self.y_observed.append(y_new)

            # Track optimization
            self.optimization_history.append({
                'iteration': iteration,
                'x': x_new.copy(),
                'y': y_new,
                'acquisition_value':
acquisition_result['acquisition_value']
            })

            # Print progress
            print(f"Iteration {iteration+1}/{n_iterations}: f(x) =
{y_new:.6f}, "
                  f"Best so far = {max(self.y_observed):.6f}")

        # Find best solution
        best_idx = np.argmax(self.y_observed)
        best_x = self.X_observed[best_idx]
        best_y = self.y_observed[best_idx]

        # Convert to parameter dictionary
        best_params = {self.parameter_names[i]: best_x[i] for i in
range(self.param_count)}

        return {
            'best_parameters': best_params,
            'best_fitness': best_y,
            'optimization_history': self.optimization_history,
            'total_evaluations': len(self.y_observed),
            'convergence_achieved': self._check_convergence()
        }
```

```
    def _evaluate_objective(self, x: np.ndarray, objective_function:
Callable) -> float:
        """Evaluate objective function at point x"""

        # Convert to parameter dictionary
        params = {self.parameter_names[i]: x[i] for i in
range(self.param_count)}

        try:
            return objective_function(params)
        except Exception as e:
            print(f"Error evaluating objective: {e}")
            return float('-inf')

    def _optimize_acquisition(self) -> Dict:
        """Optimize acquisition function"""

        def acquisition_function(x):
            """Acquisition function to minimize"""
            x = x.reshape(1, -1)

            # Get GP predictions
            mu, sigma = self.gp.predict(x, return_std=True)

            if self.acquisition_function == 'expected_improvement':
                return -self._expected_improvement(mu, sigma)
            elif self.acquisition_function == 'upper_confidence_bound':
                return -self._upper_confidence_bound(mu, sigma)
            elif self.acquisition_function ==
'probability_of_improvement':
                return -self._probability_of_improvement(mu, sigma)
            else:
                raise ValueError(f"Unknown acquisition function:
{self.acquisition_function}")

        # Optimize acquisition function
        bounds = self._format_bounds()

        # Multiple random starting points
        best_result = None
        best_value = float('inf')
```

```
        for _ in range(10):  # Try 10 random starting points
            x0 = self._sample_random_point()

            try:
                result = minimize(acquisition_function, x0,
bounds=bounds, method='L-BFGS-B')

                if result.success and result.fun < best_value:
                    best_result = result
                    best_value = result.fun
            except:
                continue

        if best_result is None:
            # Fallback to random point
            x_suggested = self._sample_random_point()
            acquisition_value = 0
        else:
            x_suggested = best_result.x
            acquisition_value = -best_result.fun

        return {
            'x': x_suggested,
            'acquisition_value': acquisition_value
        }

    def _expected_improvement(self, mu: np.ndarray, sigma: np.ndarray,
xi: float = 0.01) -> np.ndarray:
        """Expected Improvement acquisition function"""

        if np.any(sigma == 0):
            return np.zeros_like(mu)

        f_best = max(self.y_observed) if self.y_observed else 0

        with np.errstate(divide='warn'):
            improvement = mu - f_best - xi
            Z = improvement / sigma
            ei = improvement * self._normal_cdf(Z) + sigma *
self._normal_pdf(Z)

        return np.maximum(ei, 0)
```

```
    def _upper_confidence_bound(self, mu: np.ndarray, sigma:
np.ndarray, kappa: float = 2.576) -> np.ndarray:
        """Upper Confidence Bound acquisition function"""

        return mu + kappa * sigma

    def _probability_of_improvement(self, mu: np.ndarray, sigma:
np.ndarray, xi: float = 0.01) -> np.ndarray:
        """Probability of Improvement acquisition function"""

        if np.any(sigma == 0):
            return np.zeros_like(mu)

        f_best = max(self.y_observed) if self.y_observed else 0

        with np.errstate(divide='warn'):
            Z = (mu - f_best - xi) / sigma
            pi = self._normal_cdf(Z)

        return np.maximum(pi, 0)

    def _normal_cdf(self, x: np.ndarray) -> np.ndarray:
        """Cumulative distribution function of standard normal"""
        return 0.5 * (1 + np.sign(x) * np.sqrt(1 - np.exp(-2 * x**2 /
np.pi)))

    def _normal_pdf(self, x: np.ndarray) -> np.ndarray:
        """Probability density function of standard normal"""
        return np.exp(-0.5 * x**2) / np.sqrt(2 * np.pi)

    def _check_convergence(self) -> bool:
        """Check if optimization has converged"""

        if len(self.y_observed) < 10:
            return False

        # Check if best improvement in last 10 evaluations is small
        recent_improvement = max(self.y_observed[-10:]) -
max(self.y_observed[:-10])

        return recent_improvement < 1e-6
```

```
    def plot_optimization(self, save_path: str = None):
        """Plot optimization progress"""

        setup_matplotlib_for_plotting()

        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # Objective function values over iterations
        axes[0, 0].plot(self.y_observed, 'bo-', label='Observed
values')
        axes[0, 0].axhline(max(self.y_observed), color='red',
linestyle='--',
                           label=f'Best: {max(self.y_observed):.6f}')
        axes[0, 0].set_title('Objective Function Values')
        axes[0, 0].set_xlabel('Function Evaluation')
        axes[0, 0].set_ylabel('f(x)')
        axes[0, 0].legend()
        axes[0, 0].grid(True)

        # Acquisition values over iterations
        if self.optimization_history:
            acquisition_values = [h['acquisition_value'] for h in
self.optimization_history]
            axes[0, 1].plot(acquisition_values, 'go-')
            axes[0, 1].set_title('Acquisition Function Values')
            axes[0, 1].set_xlabel('Iteration')
            axes[0, 1].set_ylabel('Acquisition Value')
            axes[0, 1].grid(True)

        # Parameter evolution (if 2D)
        if self.param_count <= 2:
            X_array = np.array(self.X_observed)

            if self.param_count == 1:
                axes[1, 0].scatter(range(len(X_array)), X_array[:, 0],
                                   c=self.y_observed, cmap='viridis',
s=50)
                axes[1, 0].set_title('Parameter Evolution')
                axes[1, 0].set_xlabel('Evaluation')
                axes[1, 0].set_ylabel(self.parameter_names[0])
            else:
```

```
                scatter = axes[1, 0].scatter(X_array[:, 0], X_array[:,
1],
                                             c=self.y_observed,
cmap='viridis', s=50)
                axes[1, 0].set_title('Parameter Space Exploration')
                axes[1, 0].set_xlabel(self.parameter_names[0])
                axes[1, 0].set_ylabel(self.parameter_names[1])
                plt.colorbar(scatter, ax=axes[1, 0])

        # Convergence analysis
        if len(self.y_observed) > 1:
            moving_average = []
            window = min(10, len(self.y_observed))

            for i in range(len(self.y_observed)):
                start_idx = max(0, i - window + 1)

moving_average.append(np.mean(self.y_observed[start_idx:i+1]))

            axes[1, 1].plot(self.y_observed, 'b-', alpha=0.5,
label='Observed')
            axes[1, 1].plot(moving_average, 'r-', linewidth=2,
label=f'Moving Average (window={window})')
            axes[1, 1].set_title('Convergence Analysis')
            axes[1, 1].set_xlabel('Evaluation')
            axes[1, 1].set_ylabel('f(x)')
            axes[1, 1].legend()
            axes[1, 1].grid(True)

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')

        plt.close()
```

Adaptive Risk Management

Dynamic Risk Adjustment

```
class AdaptiveRiskManager:
    """Adaptive risk management system that adjusts based on market
conditions"""

    def __init__(self, initial_capital: float):
        self.initial_capital = initial_capital
        self.current_capital = initial_capital

        # Risk parameters that adapt
        self.risk_parameters = {
            'max_position_size': 0.1,      # 10% max per position
            'max_portfolio_risk': 0.2,     # 20% max portfolio risk
            'max_drawdown_limit': 0.15,    # 15% max drawdown
            'max_leverage': 3.0,           # 3x max leverage
            'volatility_adjustment': 1.0,  # Multiplier based on
volatility
            'correlation_limit': 0.8       # 80% max correlation
        }

        # Adaptation parameters
        self.adaptation_rates = {
            'conservative': 0.01,  # Very slow adaptation
            'moderate': 0.05,      # Moderate adaptation
            'aggressive': 0.1      # Fast adaptation
        }

        # Market condition indicators
        self.market_indicators = {
            'volatility': 0.2,     # Current market volatility
            'correlation': 0.5,    # Current correlation levels
            'trend_strength': 0.3, # Market trend strength
            'liquidity': 0.8,      # Market liquidity
            'sentiment': 0.6       # Market sentiment
        }

        # Performance tracking
        self.performance_history = []
        self.risk_adaptations = []

    def assess_market_conditions(self, market_data: Dict) -> Dict:
        """Assess current market conditions"""
```

```
        # Calculate volatility (rolling standard deviation)
        price_data = market_data.get('prices', [])
        if len(price_data) > 1:
            returns = np.diff(price_data) / price_data[:-1]
            current_volatility = np.std(returns)
        else:
            current_volatility = self.market_indicators['volatility']

        # Calculate correlation (simplified)
        # In practice, this would analyze correlations between assets
        correlation_data = market_data.get('correlations', {})
        avg_correlation = np.mean(list(correlation_data.values())) if
correlation_data else 0.5

        # Trend analysis (simplified)
        if len(price_data) >= 10:
            recent_prices = price_data[-10:]
            trend_slope = np.polyfit(range(len(recent_prices)),
recent_prices, 1)[0]
            trend_strength = abs(trend_slope) / np.mean(recent_prices)
        else:
            trend_strength = self.market_indicators['trend_strength']

        # Liquidity assessment
        volume_data = market_data.get('volumes', [])
        avg_volume = np.mean(volume_data) if volume_data else 0.8

        # Update market indicators
        self.market_indicators.update({
            'volatility': current_volatility,
            'correlation': avg_correlation,
            'trend_strength': trend_strength,
            'liquidity': avg_volume
        })

        return self.market_indicators.copy()

    def adapt_risk_parameters(self, market_conditions: Dict,
                              adaptation_speed: str = 'moderate') ->
Dict:
        """Adapt risk parameters based on market conditions"""
```

```
        # Get adaptation rate
        adaptation_rate = self.adaptation_rates.get(adaptation_speed,
0.05)

        # Store old parameters for comparison
        old_parameters = self.risk_parameters.copy()

        # Adaptation logic

        # 1. Volatility-based adjustments
        volatility = market_conditions.get('volatility', 0.2)
        if volatility > 0.3:  # High volatility
            # Reduce position sizes and leverage
            self.risk_parameters['max_position_size'] *= (1 -
adaptation_rate * 0.5)
            self.risk_parameters['max_leverage'] *= (1 -
adaptation_rate * 0.3)
            self.risk_parameters['volatility_adjustment'] = min(2.0,
1.0 / volatility)
        elif volatility < 0.1:  # Low volatility
            # Increase position sizes slightly
            self.risk_parameters['max_position_size'] *= (1 +
adaptation_rate * 0.2)
            self.risk_parameters['volatility_adjustment'] = 1.0

        # 2. Correlation-based adjustments
        correlation = market_conditions.get('correlation', 0.5)
        if correlation > 0.7:  # High correlation
            # Reduce correlation limits
            self.risk_parameters['correlation_limit'] *= (1 -
adaptation_rate * 0.2)
        else:  # Low correlation
            # Increase correlation limits slightly
            self.risk_parameters['correlation_limit'] *= (1 +
adaptation_rate * 0.1)

        # 3. Trend-based adjustments
        trend_strength = market_conditions.get('trend_strength', 0.3)
        if trend_strength > 0.5:  # Strong trend
            # Slightly increase position sizes in trending markets
            self.risk_parameters['max_position_size'] *= (1 +
adaptation_rate * 0.1)
```

```
        # 4. Liquidity-based adjustments
        liquidity = market_conditions.get('liquidity', 0.8)
        if liquidity < 0.5:  # Low liquidity
            # Reduce position sizes in illiquid markets
            self.risk_parameters['max_position_size'] *= (1 -
adaptation_rate * 0.4)
            self.risk_parameters['max_portfolio_risk'] *= (1 -
adaptation_rate * 0.2)

        # 5. Drawdown-based adjustments
        current_drawdown = self._calculate_current_drawdown()
        if current_drawdown > 0.1:  # 10% drawdown
            # Reduce risk parameters during drawdowns
            drawdown_factor = max(0.5, 1 - current_drawdown * 2)
            self.risk_parameters['max_position_size'] *=
drawdown_factor
            self.risk_parameters['max_portfolio_risk'] *=
drawdown_factor

        # Ensure parameters stay within reasonable bounds
        self.risk_parameters['max_position_size'] =
np.clip(self.risk_parameters['max_position_size'], 0.01, 0.5)
        self.risk_parameters['max_portfolio_risk'] =
np.clip(self.risk_parameters['max_portfolio_risk'], 0.05, 0.5)
        self.risk_parameters['max_drawdown_limit'] =
np.clip(self.risk_parameters['max_drawdown_limit'], 0.05, 0.3)
        self.risk_parameters['max_leverage'] =
np.clip(self.risk_parameters['max_leverage'], 1.0, 10.0)
        self.risk_parameters['correlation_limit'] =
np.clip(self.risk_parameters['correlation_limit'], 0.3, 0.95)

        # Record adaptation
        adaptation_record = {
            'timestamp': datetime.now(),
            'old_parameters': old_parameters,
            'new_parameters': self.risk_parameters.copy(),
            'market_conditions': market_conditions.copy(),
            'adaptation_rate': adaptation_rate
        }

        self.risk_adaptations.append(adaptation_record)
```

```
        # Calculate what changed
        changes = {}
        for param_name in self.risk_parameters:
            old_val = old_parameters[param_name]
            new_val = self.risk_parameters[param_name]
            if abs(old_val - new_val) > 1e-6:
                changes[param_name] = {
                    'old_value': old_val,
                    'new_value': new_val,
                    'change_percent': ((new_val - old_val) / old_val) *
100
                }

        return changes

    def _calculate_current_drawdown(self) -> float:
        """Calculate current drawdown from peak"""

        if not self.performance_history:
            return 0.0

        current_value = self.current_capital
        peak_value = max([self.initial_capital] + [p['capital'] for p
in self.performance_history])

        return max(0, (peak_value - current_value) / peak_value)

    def evaluate_trade_risk(self, trade_proposal: Dict) -> Tuple[bool,
Dict]:
        """Evaluate trade against current risk parameters"""

        # Get trade details
        trade_size = trade_proposal.get('size', 0)
        trade_value = trade_proposal.get('value', 0)
        asset = trade_proposal.get('asset', 'unknown')

        # Calculate risk metrics
        position_risk = trade_value / self.current_capital
        portfolio_risk = self._calculate_portfolio_risk(trade_proposal)

        # Check against adapted parameters
```

```
        checks = {
            'position_size_limit': position_risk <=
self.risk_parameters['max_position_size'],
            'portfolio_risk_limit': portfolio_risk <=
self.risk_parameters['max_portfolio_risk'],
            'leverage_limit': self._calculate_leverage() <=
self.risk_parameters['max_leverage'],
            'correlation_limit': self._check_correlation_limit(asset),
            'volatility_limit':
self._check_volatility_limit(trade_proposal)
        }

        # Overall risk assessment
        risk_approved = all(checks.values())

        # Generate risk report
        risk_report = {
            'approved': risk_approved,
            'position_risk': position_risk,
            'portfolio_risk': portfolio_risk,
            'checks': checks,
            'risk_score': self._calculate_risk_score(trade_proposal),
            'recommendations':
self._generate_risk_recommendations(checks)
        }

        # Update performance tracking
        self.performance_history.append({
            'timestamp': datetime.now(),
            'capital': self.current_capital,
            'risk_assessment': risk_report
        })

        return risk_approved, risk_report

    def _calculate_portfolio_risk(self, trade_proposal: Dict) -> float:
        """Calculate portfolio risk after trade"""

        # Simplified portfolio risk calculation
        # In practice, this would use more sophisticated risk models
        trade_risk = trade_proposal.get('value', 0) /
self.current_capital
```

```
        # Add correlation and concentration adjustments
        asset = trade_proposal.get('asset', 'unknown')
        concentration_risk = self._calculate_concentration_risk(asset)

        total_risk = trade_risk * (1 + concentration_risk)

        return total_risk

    def _calculate_leverage(self) -> float:
        """Calculate current leverage"""

        # Simplified leverage calculation
        total_exposure = self._calculate_total_exposure()
        leverage = total_exposure / self.current_capital if
self.current_capital > 0 else 0

        return leverage

    def _calculate_concentration_risk(self, asset: str) -> float:
        """Calculate concentration risk for asset"""

        # Simplified concentration risk
        # In practice, this would analyze actual portfolio composition
        asset_exposure = self._get_asset_exposure(asset)
        concentration = asset_exposure / self.current_capital if
self.current_capital > 0 else 0

        return concentration

    def _check_correlation_limit(self, asset: str) -> bool:
        """Check if adding asset exceeds correlation limits"""

        # Simplified correlation check
        # In practice, this would use actual correlation analysis
        current_correlations = self._get_current_correlations()
        avg_correlation = np.mean(list(current_correlations.values()))
if current_correlations else 0.5

        return avg_correlation <=
self.risk_parameters['correlation_limit']
```

```
    def _check_volatility_limit(self, trade_proposal: Dict) -> bool:
        """Check if trade volatility exceeds limits"""

        # Get asset volatility
        asset = trade_proposal.get('asset', 'unknown')
        asset_volatility = self._get_asset_volatility(asset)

        # Adjust for market volatility
        market_volatility = self.market_indicators['volatility']
        adjusted_volatility = asset_volatility * market_volatility *
self.risk_parameters['volatility_adjustment']

        # Check against limits (simplified)
        max_volatility = 0.5  # 50% max volatility
        return adjusted_volatility <= max_volatility

    def _calculate_risk_score(self, trade_proposal: Dict) -> float:
        """Calculate overall risk score for trade (0-1, lower is
better)"""

        risk_factors = []

        # Position size risk
        position_risk = trade_proposal.get('value', 0) /
self.current_capital
        position_score = min(1.0, position_risk /
self.risk_parameters['max_position_size'])
        risk_factors.append(position_score)

        # Portfolio risk
        portfolio_risk = self._calculate_portfolio_risk(trade_proposal)
        portfolio_score = min(1.0, portfolio_risk /
self.risk_parameters['max_portfolio_risk'])
        risk_factors.append(portfolio_score)

        # Leverage risk
        leverage_risk = self._calculate_leverage() /
self.risk_parameters['max_leverage']
        leverage_score = min(1.0, leverage_risk)
        risk_factors.append(leverage_score)

        # Volatility risk
```

```
        volatility_score = min(1.0,
self.market_indicators['volatility'] / 0.5)
        risk_factors.append(volatility_score)

        # Overall risk score (weighted average)
        overall_score = np.mean(risk_factors)

        return overall_score

    def _generate_risk_recommendations(self, checks: Dict) ->
List[str]:
        """Generate risk management recommendations"""

        recommendations = []

        if not checks['position_size_limit']:

recommendations.append("Reduce position size to comply with risk
limits")

        if not checks['portfolio_risk_limit']:
            recommendations.append("Portfolio risk too high - consider
reducing exposure")

        if not checks['leverage_limit']:
            recommendations.append("Leverage exceeds limits -
deleverage or reduce positions")

        if not checks['correlation_limit']:
            recommendations.append("Correlation limits exceeded -
diversify holdings")

        if not checks['volatility_limit']:

recommendations.append("Volatility risk too high - consider lower
volatility assets")

        if not recommendations:
            recommendations.append("Trade meets all risk criteria")

        return recommendations
```

```
    # Helper methods (simplified implementations)
    def _calculate_total_exposure(self) -> float:
        """Calculate total portfolio exposure"""
        return self.current_capital * 1.5  # Simplified

    def _get_asset_exposure(self, asset: str) -> float:
        """Get current exposure to specific asset"""
        return self.current_capital * 0.1  # Simplified

    def _get_current_correlations(self) -> Dict[str, float]:
        """Get current asset correlations"""
        return {'ETH': 0.6, 'USDC': 0.2, 'WBTC': 0.7}  # Simplified

    def _get_asset_volatility(self, asset: str) -> float:
        """Get asset volatility"""
        volatilities = {'ETH': 0.3, 'USDC': 0.01, 'WBTC': 0.4}
        return volatilities.get(asset, 0.2)

    def get_risk_status(self) -> Dict:
        """Get current risk management status"""

        current_drawdown = self._calculate_current_drawdown()
        current_leverage = self._calculate_leverage()

        return {
            'current_capital': self.current_capital,
            'current_drawdown': current_drawdown,
            'current_leverage': current_leverage,
            'risk_parameters': self.risk_parameters.copy(),
            'market_conditions': self.market_indicators.copy(),
            'recent_adaptations': len(self.risk_adaptations),
            'risk_status': self._get_risk_status_level()
        }

    def _get_risk_status_level(self) -> str:
        """Get overall risk status level"""

        drawdown = self._calculate_current_drawdown()
        leverage = self._calculate_leverage()

        if drawdown > 0.15 or leverage > 5.0:
            return "HIGH_RISK"
```

```
        elif drawdown > 0.10 or leverage > 3.0:
            return "MODERATE_RISK"
        elif drawdown > 0.05 or leverage > 2.0:
            return "LOW_RISK"
        else:
            return "NORMAL"
```

This comprehensive module covers adaptive algorithms for MEV strategies, from online learning to evolutionary optimization. The key concepts include:

1. **Online Learning**: Continuously updating models with new data
2. **Multi-Armed Bandits**: Balancing exploration and exploitation
3. **Reinforcement Learning**: Learning optimal policies through interaction
4. **Evolutionary Algorithms**: Population-based optimization
5. **Bayesian Optimization**: Efficient parameter tuning
6. **Adaptive Risk Management**: Dynamic risk adjustment

The exercises will provide hands-on experience implementing these adaptive systems for MEV strategy optimization.

In the next module, we'll explore multi-asset and cross-chain arbitrage systems.