import os import json import time import logging import threading import schedule from datetime import datetime from flask import Flask, request, jsonify from pymongo import MongoClient import redis import numpy as np import pandas as pd import cvxpy as cp from scipy.optimize import minimize from pulp import LpMaximize, LpProblem, LpVariable, lpSum, LpStatus import requests from dotenv import load_dotenv

Load environment variables

load_dotenv()

Configure logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s') logger = logging.getLogger(**name**)

Initialize Flask app

app = Flask(**name**)

Connect to MongoDB

mongo_uri = os.getenv("MONGODB_URI", "mongodb://localhost:27017/amgf") mongo_client = MongoClient(mongo_uri) db = mongo_client.amgf opportunities_collection = db.opportunities predictions_collection = db.predictions allocations_collection = db.allocations metrics_collection = db.metrics

Connect to Redis

redis_uri = os.getenv("REDIS_URI", "redis://localhost:6379") redis_client = redis.from_url(redis_uri)

Constants

ALLOCATION_INTERVAL = int(os.getenv("ALLOCATION_INTERVAL", "3600")) # Default: reallocate every hour MAX_RESOURCE_ALLOCATION = float(os.getenv("MAX_RESOURCE_ALLOCATION", "1000")) # Default: $1000 REINVESTMENT_RATIO = float(os.getenv("REINVESTMENT_RATIO", "0.9")) # Default: 90% reinvestment RESERVE_RATIO = float(os.getenv("RESERVE_RATIO", "0.1")) # Default: 10% reserve MAX_AUTONOMOUS_SHIFT = float(os.getenv("MAX_AUTONOMOUS_SHIFT", "0.2")) # Default: 20% max shift

Service endpoints

OPPORTUNITY_SCANNER_URL = os.getenv("OPPORTUNITY_SCANNER_URL", "http://opportunity-scanner:5000") ROI_PREDICTOR_URL = os.getenv("ROI_PREDICTOR_URL", "http://roi-predictor:5000")

class ResourceAllocator: """ Main class for allocating resources across opportunities. """

```
def __init__(self):
    self.current_allocation = {}
```

```
        self.current_allocation = {}
        self.allocation_history = []
        self.available_resources = MAX_RESOURCE_ALLOCATION
        self.reserve_funds = 0
        self.strategy_weights = {
            "service": 0.4,
            "content": 0.3,
            "marketplace": 0.3
        }
        self.risk_tolerance = 0.6  # 0-1 scale, higher = more risk tolerant

        # Load current allocation if available
        self.load_current_allocation()

    def load_current_allocation(self):
        """Load current allocation from database."""
        current = allocations_collection.find_one({"status": "active"})
        if current:
            self.current_allocation = current.get("allocations", {})
            self.available_resources = current.get("available_resources",
MAX_RESOURCE_ALLOCATION)
            self.reserve_funds = current.get("reserve_funds", 0)
            logger.info(f"Loaded current allocation:
{len(self.current_allocation)} opportunities, {self.available_resources}
available resources")
        else:
            logger.info("No current allocation found")

    def get_opportunities(self):
        """Get opportunities from Opportunity Scanner."""
        try:
            response = requests.get(f"
{OPPORTUNITY_SCANNER_URL}/opportunities")
            if response.status_code == 200:
                data = response.json()
                return data.get("opportunities", [])
            else:
                logger.error(f"Error getting opportunities:
{response.status_code}")
                return []
        except Exception as e:
            logger.error(f"Error connecting to Opportunity Scanner: {str(e)}")
            return []

    def get_predictions(self, opportunities):
        """Get ROI predictions from ROI Predictor."""
        try:
            response = requests.post(f"{ROI_PREDICTOR_URL}/predict/batch",
json=opportunities)
            if response.status_code == 200:
                data = response.json()
                return data.get("predictions", [])
            else:
                logger.error(f"Error getting predictions:
{response.status_code}")
                return []
        except Exception as e:
            logger.error(f"Error connecting to ROI Predictor: {str(e)}")
            return []

    def allocate_resources(self):
        """Allocate resources across opportunities."""
        logger.info("Starting resource allocation")

        # Get opportunities
        opportunities = self.get_opportunities()
        if not opportunities:
            logger.warning("No opportunities found")
            return

        # Get predictions
        predictions = self.get_predictions(opportunities)
        if not predictions:
            logger.warning("No predictions available")
            return
```

```
            return

        # Combine opportunities and predictions
        opportunity_map = {opp["id"]: opp for opp in opportunities}
        prediction_map = {pred["opportunity_id"]: pred for pred in predictions
if "error" not in pred}

        combined_data = []
        for opp_id, opp in opportunity_map.items():
            if opp_id in prediction_map:
                combined_data.append({
                    "opportunity": opp,
                    "prediction": prediction_map[opp_id]
                })

        if not combined_data:
            logger.warning("No valid combined data")
            return

        # Perform allocation
        allocation = self.optimize_allocation(combined_data)

        # Check allocation shift limits
        allocation = self.apply_shift_limits(allocation)

        # Update current allocation
        self.update_allocation(allocation)

        logger.info(f"Resource allocation complete: {len(allocation)}
opportunities allocated")

        return allocation

    def optimize_allocation(self, combined_data):
        """Optimize resource allocation using portfolio optimization."""
        logger.info("Optimizing resource allocation")

        # Extract data
        opportunities = [item["opportunity"] for item in combined_data]
        predictions = [item["prediction"] for item in combined_data]

        # Calculate expected returns and risks
        expected_returns = np.array([pred["predicted_roi"] for pred in
predictions])

        # Calculate risk as a combination of prediction confidence and risk
score
        risks = np.array([
            (pred["confidence_interval"]["upper"] -
pred["confidence_interval"]["lower"]) / pred["predicted_roi"] *
pred["risk_score"]
            for pred in predictions
        ])

        # Calculate implementation times
        implementation_times = np.array([opp["implementation_time"] for opp in
opportunities])

        # Calculate time to revenue
        time_to_revenue = np.array([pred["time_to_revenue"] for pred in
predictions])

        # Calculate opportunity costs
        opportunity_costs = np.array([opp["estimated_value"] for opp in
opportunities])

        # Create optimization problem
        n = len(opportunities)

        # Use PuLP for linear programming
        prob = LpProblem("ResourceAllocation", LpMaximize)

        # Decision variables: allocation amount for each opportunity
        allocation_vars = [LpVariable(f"alloc_{i}", lowBound=0,
upBound=opp["estimated_value"] * 2) for i, opp in
```

```
upbound=opp["estimated_value"] * 2) for i, opp in
enumerate(opportunities)]

    # Binary variables for whether to invest in an opportunity
    binary_vars = [LpVariable(f"select_{i}", cat='Binary') for i in
range(n)]

    # Objective: maximize expected return
    prob += lpSum([allocation_vars[i] * expected_returns[i] for i in
range(n)])

    # Constraint: total allocation cannot exceed available resources
    prob += lpSum(allocation_vars) <= self.available_resources

    # Constraint: allocation must be zero if opportunity is not selected
    for i in range(n):
        prob += allocation_vars[i] <= opportunities[i]["estimated_value"]
* 2 * binary_vars[i]

    # Constraint: limit number of opportunities to prevent spreading too
thin
    max_opportunities = min(20, n)  # Maximum 20 opportunities or all
available
    prob += lpSum(binary_vars) <= max_opportunities

    # Constraint: minimum allocation per opportunity if selected
    min_allocation = 5  # Minimum $5 per opportunity
    for i in range(n):
        prob += allocation_vars[i] >= min_allocation * binary_vars[i]

    # Constraint: category diversification
    service_indices = [i for i, opp in enumerate(opportunities) if
opp["category"] == "service"]
    content_indices = [i for i, opp in enumerate(opportunities) if
opp["category"] == "content"]
    marketplace_indices = [i for i, opp in enumerate(opportunities) if
opp["category"] == "marketplace"]

    # Ensure minimum allocation per category based on strategy weights
    if service_indices:
        prob += lpSum([allocation_vars[i] for i in service_indices]) >=
self.available_resources * self.strategy_weights["service"] * 0.7
    if content_indices:
        prob += lpSum([allocation_vars[i] for i in content_indices]) >=
self.available_resources * self.strategy_weights["content"] * 0.7
    if marketplace_indices:
        prob += lpSum([allocation_vars[i] for i in marketplace_indices])
>= self.available_resources * self.strategy_weights["marketplace"] * 0.7

    # Solve the problem
    prob.solve()

    # Check if solution was found
    if LpStatus[prob.status] != "Optimal":
        logger.warning(f"Optimization did not find optimal solution:
{LpStatus[prob.status]}")
        # Fall back to simpler allocation strategy
        return self.simple_allocation(combined_data)

    # Extract results
    allocation = {}
    for i, opp in enumerate(opportunities):
        amount = allocation_vars[i].value()
        if amount is not None and amount > 0:
            allocation[opp["id"]] = {
                "opportunity_id": opp["id"],
                "amount": round(amount, 2),
                "expected_roi": expected_returns[i],
                "risk_score": predictions[i]["risk_score"],
                "implementation_time": opp["implementation_time"],
                "time_to_revenue": predictions[i]["time_to_revenue"],
                "category": opp["category"],
                "platform": opp["platform"],
                "timestamp": datetime.utcnow().isoformat()
            }
```

```
            }

        return allocation

    def simple_allocation(self, combined_data):
        """Simple allocation strategy as fallback."""
        logger.info("Using simple allocation strategy")

        # Sort by expected ROI
        sorted_data = sorted(combined_data, key=lambda x: x["prediction"]
    ["predicted_roi"], reverse=True)

        # Calculate total allocation
        total_allocation = 0
        allocation = {}

        # Allocate to top opportunities
        for item in sorted_data:
            opp = item["opportunity"]
            pred = item["prediction"]

            # Skip if ROI is negative
            if pred["predicted_roi"] <= 0:
                continue

            # Calculate allocation amount based on estimated value and ROI
            amount = min(opp["estimated_value"] * 1.5,
    self.available_resources * 0.1)

            # Ensure we don't exceed available resources
            if total_allocation + amount > self.available_resources:
                amount = self.available_resources - total_allocation
                if amount <= 0:
                    break

            # Add to allocation
            allocation[opp["id"]] = {
                "opportunity_id": opp["id"],
                "amount": round(amount, 2),
                "expected_roi": pred["predicted_roi"],
                "risk_score": pred["risk_score"],
                "implementation_time": opp["implementation_time"],
                "time_to_revenue": pred["time_to_revenue"],
                "category": opp["category"],
                "platform": opp["platform"],
                "timestamp": datetime.utcnow().isoformat()
            }

            total_allocation += amount

            # Stop if we've allocated to enough opportunities
            if len(allocation) >= 20:
                break

        return allocation

    def apply_shift_limits(self, new_allocation):
        """Apply limits to how much allocation can shift between runs."""
        if not self.current_allocation:
            return new_allocation

        logger.info("Applying shift limits to allocation")

        # Calculate total current allocation
        total_current = sum(item["amount"] for item in
    self.current_allocation.values())

        # Calculate total new allocation
        total_new = sum(item["amount"] for item in new_allocation.values())

        # Calculate maximum allowed shift
        max_shift_amount = total_current * MAX_AUTONOMOUS_SHIFT

        # Check if shift exceeds limit
        shift_amount = abs(total_new - total_current)
```

```
    shift_amount = abs(total_new - total_current)
    if shift_amount > max_shift_amount:
        logger.info(f"Allocation shift exceeds limit: {shift_amount} >
{max_shift_amount}")

        # Scale back new allocation to respect shift limit
        scale_factor = (total_current + max_shift_amount *
np.sign(total_new - total_current)) / total_new

        for opp_id in new_allocation:
            new_allocation[opp_id]["amount"] =
round(new_allocation[opp_id]["amount"] * scale_factor, 2)

    # Check individual opportunity shifts
    for opp_id, new_alloc in new_allocation.items():
        if opp_id in self.current_allocation:
            current_amount = self.current_allocation[opp_id]["amount"]
            new_amount = new_alloc["amount"]

            # Calculate maximum allowed shift for this opportunity
            opp_max_shift = current_amount * MAX_AUTONOMOUS_SHIFT

            # Check if shift exceeds limit
            opp_shift = abs(new_amount - current_amount)
            if opp_shift > opp_max_shift:
                # Limit the shift
                new_amount = current_amount + opp_max_shift *
np.sign(new_amount - current_amount)
                new_allocation[opp_id]["amount"] = round(new_amount, 2)

    return new_allocation

def update_allocation(self, allocation):
    """Update current allocation and store in database."""
    # Calculate total allocation
    total_allocation = sum(item["amount"] for item in allocation.values())

    # Update available resources
    self.available_resources = MAX_RESOURCE_ALLOCATION - total_allocation

    # Update reserve funds
    self.reserve_funds = MAX_RESOURCE_ALLOCATION * RESERVE_RATIO

    # Store previous allocation in history
    if self.current_allocation:
        self.allocation_history.append({
            "allocations": self.current_allocation,
            "timestamp": datetime.utcnow().isoformat(),
            "total_allocated": sum(item["amount"] for item in
self.current_allocation.values())
        })

    # Update current allocation
    self.current_allocation = allocation

    # Store in database
    allocations_collection.update_one(
        {"status": "active"},
        {"$set": {
            "allocations": allocation,
            "available_resources": self.available_resources,
            "reserve_funds": self.reserve_funds,
            "timestamp": datetime.utcnow().isoformat(),
            "total_allocated": total_allocation
        }},
        upsert=True
    )

    # Store allocation history
    if self.allocation_history:
        allocations_collection.insert_one({
            "status": "historical",
            "allocations": self.allocation_history[-1]["allocations"],
            "timestamp": self.allocation_history[-1]["timestamp"],
            "total_allocated": self.allocation_history[-1]
```

```
            total_allocated": self.allocation_history[-1]
["total_allocated"]
        })

    # Update metrics
    metrics_collection.update_one(
        {"metric": "resource_allocation"},
        {"$set": {
            "last_allocation": datetime.utcnow().isoformat(),
            "total_allocated": total_allocation,
            "available_resources": self.available_resources,
            "reserve_funds": self.reserve_funds,
            "opportunity_count": len(allocation)
        }},
        upsert=True
    )

    # Publish event to Redis
    redis_client.publish(
        "events:allocation_updated",
        json.dumps({
            "timestamp": datetime.utcnow().isoformat(),
            "total_allocated": total_allocation,
            "opportunity_count": len(allocation)
        })
    )

def get_current_allocation(self):
    """Get current resource allocation."""
    return {
        "allocations": self.current_allocation,
        "available_resources": self.available_resources,
        "reserve_funds": self.reserve_funds,
        "total_allocated": sum(item["amount"] for item in
self.current_allocation.values()),
        "opportunity_count": len(self.current_allocation),
        "timestamp": datetime.utcnow().isoformat()
    }

def get_allocation_history(self, limit=10):
    """Get allocation history."""
    history = list(allocations_collection.find(
        {"status": "historical"},
        {"_id": 0}
    ).sort("timestamp", -1).limit(limit))

    return history

def adjust_strategy_weights(self, new_weights):
    """Adjust strategy weights."""
    # Validate weights
    if sum(new_weights.values()) != 1.0:
        raise ValueError("Strategy weights must sum to 1.0")

    # Update weights
    self.strategy_weights = new_weights

    # Store in database
    metrics_collection.update_one(
        {"metric": "strategy_weights"},
        {"$set": {
            "weights": new_weights,
            "timestamp": datetime.utcnow().isoformat()
        }},
        upsert=True
    )

def adjust_risk_tolerance(self, risk_tolerance):
    """Adjust risk tolerance."""
    # Validate risk tolerance
    if not 0 <= risk_tolerance <= 1:
        raise ValueError("Risk tolerance must be between 0 and 1")

    # Update risk tolerance
    self.risk_tolerance = risk_tolerance
```

```
        self.risk_tolerance = risk_tolerance

        # Store in database
        metrics_collection.update_one(
            {"metric": "risk_tolerance"},
            {"$set": {
                "value": risk_tolerance,
                "timestamp": datetime.utcnow().isoformat()
            }},
            upsert=True
        )

    def reinvest_profits(self, profits):
        """Reinvest profits into the system."""
        # Calculate reinvestment amount
        reinvestment = profits * REINVESTMENT_RATIO
        reserve = profits * RESERVE_RATIO

        # Update available resources
        self.available_resources += reinvestment
        self.reserve_funds += reserve

        # Store in database
        metrics_collection.update_one(
            {"metric": "reinvestment"},
            {"$set": {
                "profits": profits,
                "reinvestment": reinvestment,
                "reserve": reserve,
                "timestamp": datetime.utcnow().isoformat()
            }},
            upsert=True
        )

        # Update allocation document
        allocations_collection.update_one(
            {"status": "active"},
            {"$set": {
                "available_resources": self.available_resources,
                "reserve_funds": self.reserve_funds
            }}
        )

        # Publish event to Redis
        redis_client.publish(
            "events:profits_reinvested",
            json.dumps({
                "timestamp": datetime.utcnow().isoformat(),
                "profits": profits,
                "reinvestment": reinvestment,
                "reserve": reserve
            })
        )

        return {
            "profits": profits,
            "reinvestment": reinvestment,
            "reserve": reserve,
            "available_resources": self.available_resources,
            "reserve_funds": self.reserve_funds
        }
```

Initialize allocator

allocator = ResourceAllocator()

API Routes

@app.route(‘/health’, methods=[‘GET’]) def health_check(): “““Health check endpoint.”””

return jsonify({“status”: “healthy”, “timestamp”: datetime.utcnow().isoformat()})

@app.route(‘/allocate’, methods=[‘POST’]) def trigger_allocation(): “““Trigger resource allocation.””” try: allocation = allocator.allocate_resources()

```
    return jsonify({
        "status": "success",
        "message": f"Allocation complete. Allocated to {len(allocation)}
opportunities.",
        "total_allocated": sum(item["amount"] for item in
allocation.values()),
        "available_resources": allocator.available_resources,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error during allocation: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/allocation’, methods=[‘GET’]) def get_allocation(): “““Get current resource allocation.””” try: allocation = allocator.get_current_allocation()

```
    return jsonify({
        "status": "success",
        "allocation": allocation,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error getting allocation: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/allocation/history’, methods=[‘GET’]) def get_allocation_history(): “““Get allocation history.””” try: limit = int(request.args.get(‘limit’, 10)) history = allocator.get_allocation_history(limit)

```
    return jsonify({
        "status": "success",
        "history": history,
        "count": len(history),
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error getting allocation history: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/strategy/weights’, methods=[‘POST’]) def adjust_strategy_weights(): “““Adjust strategy weights.””” try: weights = request.json

```
    if not weights:
        return jsonify({
            "status": "error",
            "message": "No weights provided",
            "timestamp": datetime.utcnow().isoformat()
        }), 400

    allocator.adjust_strategy_weights(weights)

    return jsonify({
        "status": "success",
        "message": "Strategy weights updated",
        "weights": weights,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error adjusting strategy weights: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/risk/tolerance’, methods=[‘POST’]) def adjust_risk_tolerance(): “““Adjust risk tolerance.””” try: data = request.json

```
    if not data or "risk_tolerance" not in data:
        return jsonify({
            "status": "error",
            "message": "No risk tolerance provided",
            "timestamp": datetime.utcnow().isoformat()
        }), 400

    risk_tolerance = float(data["risk_tolerance"])
    allocator.adjust_risk_tolerance(risk_tolerance)

    return jsonify({
        "status": "success",
        "message": "Risk tolerance updated",
        "risk_tolerance": risk_tolerance,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error adjusting risk tolerance: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/reinvest’, methods=[‘POST’]) def reinvest_profits(): “““Reinvest profits.””” try: data = request.json

```
    if not data or "profits" not in data:
        return jsonify({
            "status": "error",
            "message": "No profits provided",
            "timestamp": datetime.utcnow().isoformat()
        }), 400

    profits = float(data["profits"])
    result = allocator.reinvest_profits(profits)

    return jsonify({
        "status": "success",
        "message": "Profits reinvested",
        "result": result,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error reinvesting profits: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

def run_scheduler(): “””Run the scheduler in a separate thread.””” logger.info(“Starting scheduler”)

```
# Schedule regular allocations
schedule.every(ALLOCATION_INTERVAL).seconds.do(allocator.allocate_resource
)

# Run the scheduler
while True:
    schedule.run_pending()
    time.sleep(1)
```

if **name** == ‘**main**’: # Start scheduler in a separate thread scheduler_thread = threading.Thread(target=run_scheduler) scheduler_thread.daemon = True scheduler_thread.start()

```
# Run initial allocation
allocator.allocate_resources()

# Start Flask app
app.run(host='0.0.0.0', port=5000)
```