import time

import math

from collections import deque

class MacaluMarketEngine:

"""

The MacaluMarketEngine is a superior algorithmic core designed for

dynamic market optimization, transcending traditional simulation-based approaches.

It adaptively learns from real-time market flux to derive optimal strategies

for pricing, inventory, and recommendation, without explicit future state

simulations. Its intelligence lies in real-time pattern recognition and

self-correcting adaptive response mechanisms.

"""

def \_\_init\_\_(self, initial\_products\_data: dict, learning\_rate: float = 0.01, decay\_factor: float = 0.999):

"""

Initializes the MacaluMarketEngine with foundational product data and

adaptive learning parameters.

Args:

initial\_products\_data (dict): A dictionary mapping product IDs to

their initial market parameters (e.g.,

{'product\_id': {'base\_price': 100.0, 'inventory': 100}}).

learning\_rate (float): The rate at which the algorithm adapts to new

market insights.

decay\_factor (float): Factor by which market influence diminishes over time,

prioritizing recent data.

"""

self.products = {

pid: {

'base\_price': data.get('base\_price', 1.0),

'current\_price': data.get('base\_price', 1.0),

'inventory': data.get('inventory', 0),

'demand\_factor': 1.0, # Represents an aggregate view of demand

'supply\_factor': 1.0, # Represents an aggregate view of supply/competition

'market\_sentiment': 0.0, # -1 (negative) to 1 (positive)

'price\_history': deque(maxlen=100), # Recent prices

'sales\_history': deque(maxlen=100), # Recent sales volumes

'competitor\_prices': {}, # Track key competitors

'adaptive\_bias': 0.0 # Internal adaptive adjustment for optimization

}

for pid, data in initial\_products\_data.items()

}

self.learning\_rate = learning\_rate

self.decay\_factor = decay\_factor

self.global\_market\_trends = {'economy\_strength': 1.0, 'inflation\_rate': 0.0}

self.last\_update\_time = time.time()

def \_update\_global\_trends(self):

"""

Internal mechanism to update global market trends, reacting to

macroeconomic indicators or aggregated market signals.

This is a conceptual placeholder; in a production system, this would

ingest external economic feeds.

"""

# Example: Simple decay for demonstration. In reality, this would be complex.

time\_elapsed = time.time() - self.last\_update\_time

if time\_elapsed > 60: # Update every minute, for instance

# Simulate slight fluctuations, or integrate real economic data APIs

self.global\_market\_trends['economy\_strength'] \*= (1 + 0.0001 \* (2 \* (0.5 - math.random())))

self.global\_market\_trends['inflation\_rate'] += 0.00001 \* (0.5 - math.random())

self.last\_update\_time = time.time()

def ingest\_market\_event(self, event\_type: str, payload: dict):

"""

Ingests a real-time market event, dynamically updating the engine's

understanding of market conditions for specific products or globally.

Args:

event\_type (str): Type of market event (e.g., 'sale', 'competitor\_price\_update',

'demand\_spike', 'inventory\_change', 'market\_news').

payload (dict): Event-specific data.

"""

self.\_update\_global\_trends()

product\_id = payload.get('product\_id')

if event\_type == 'sale' and product\_id in self.products:

quantity\_sold = payload.get('quantity', 1)

price\_at\_sale = payload.get('price', self.products[product\_id]['current\_price'])

revenue = quantity\_sold \* price\_at\_sale

product\_state = self.products[product\_id]

product\_state['inventory'] = max(0, product\_state['inventory'] - quantity\_sold)

product\_state['sales\_history'].append({'quantity': quantity\_sold, 'price': price\_at\_sale, 'time': time.time()})

# Adaptive adjustment based on sales performance

# If sales are good at current price, demand factor might increase slightly.

# If sales are stagnant, demand factor might decrease.

# This is a highly simplified adaptive feedback loop.

sales\_rate = quantity\_sold / (payload.get('time\_since\_last\_sale', 1.0) + 1e-9) # Prevent division by zero

demand\_impact = min(1.0, max(-1.0, (sales\_rate - product\_state['demand\_factor']) \* self.learning\_rate))

product\_state['demand\_factor'] += demand\_impact \* (1.0 - product\_state['adaptive\_bias'])

# Decay old data influence

for key in ['demand\_factor', 'supply\_factor', 'market\_sentiment']:

product\_state[key] \*= self.decay\_factor

# Adjust adaptive\_bias based on perceived 'optimality' of previous price

# This is where the 'no simulation, but adaptive' intelligence comes in.

# If sales were unexpectedly high, it suggests the price might have been too low,

# so bias shifts towards higher prices, and vice versa.

expected\_sales = product\_state['demand\_factor'] \* product\_state['inventory'] # A rough expectation

performance\_delta = (quantity\_sold - expected\_sales) / (expected\_sales + 1e-9) # How much better/worse were actual sales

product\_state['adaptive\_bias'] += performance\_delta \* self.learning\_rate \* 0.1 # Small adjustment

elif event\_type == 'competitor\_price\_update' and product\_id in self.products:

competitor\_id = payload.get('competitor\_id')

new\_competitor\_price = payload.get('price')

if competitor\_id and new\_competitor\_price is not None:

self.products[product\_id]['competitor\_prices'][competitor\_id] = new\_competitor\_price

# Adjust supply\_factor based on competitor actions

avg\_competitor\_price = sum(self.products[product\_id]['competitor\_prices'].values()) / \

(len(self.products[product\_id]['competitor\_prices']) + 1e-9)

if avg\_competitor\_price > 0:

price\_difference\_ratio = (self.products[product\_id]['current\_price'] - avg\_competitor\_price) / avg\_competitor\_price

self.products[product\_id]['supply\_factor'] += price\_difference\_ratio \* self.learning\_rate \* 0.5 # React to competition

elif event\_type == 'inventory\_change' and product\_id in self.products:

change\_amount = payload.get('change', 0)

self.products[product\_id]['inventory'] += change\_amount

# Inventory changes directly impact supply factor

self.products[product\_id]['supply\_factor'] = max(0.1, self.products[product\_id]['supply\_factor'] + change\_amount \* self.learning\_rate \* 0.01)

elif event\_type == 'market\_news':

# This is for broader sentiment or specific product news

sentiment\_change = payload.get('sentiment\_impact', 0.0) # e.g., -1.0 to 1.0

if product\_id and product\_id in self.products:

self.products[product\_id]['market\_sentiment'] = (self.products[product\_id]['market\_sentiment'] \* self.decay\_factor + sentiment\_change \* (1 - self.decay\_factor))

else: # Global market news

# For global events, this would affect global\_market\_trends

self.global\_market\_trends['economy\_strength'] += sentiment\_change \* self.learning\_rate \* 0.1

def calculate\_optimal\_price(self, product\_id: str) -> float:

"""

Calculates the dynamically optimized price for a given product based on

current market intelligence and adaptive learning. This is the core

decision-making algorithm.

Args:

product\_id (str): The ID of the product for which to calculate the price.

Returns:

float: The optimized price for the product. Returns base\_price if product\_id is invalid.

"""

if product\_id not in self.products:

print(f"Product {product\_id} not found.")

return 0.0 # Or raise an error

product\_state = self.products[product\_id]

base\_price = product\_state['base\_price']

inventory = product\_state['inventory']

demand\_factor = product\_state['demand\_factor']

supply\_factor = product\_state['supply\_factor']

market\_sentiment = product\_state['market\_sentiment']

adaptive\_bias = product\_state['adaptive\_bias']

# Incorporate global trends

economy\_strength = self.global\_market\_trends['economy\_strength']

inflation\_rate = self.global\_market\_trends['inflation\_rate']

# --- Advanced Adaptive Pricing Model ---

# This model combines various factors with adaptive weights.

# It's designed to react to demand, competition, inventory, and broader sentiment.

# Demand influence: Higher demand -> higher price

demand\_influence = demand\_factor \* (1 + market\_sentiment \* 0.1)

# Supply/Competition influence: Low inventory or high competitor prices -> higher price

# High inventory or low competitor prices -> lower price

inventory\_factor = max(0.1, 1 - (inventory / (inventory + 1000))) # Assumes larger inventory leads to more pressure to sell

avg\_comp\_price = sum(product\_state['competitor\_prices'].values()) / \

(len(product\_state['competitor\_prices']) + 1e-9)

competitor\_pressure = 1.0

if avg\_comp\_price > 0:

competitor\_pressure = (avg\_comp\_price / base\_price) \* 0.5 # If competitors price higher, we can too

supply\_influence = (1 / supply\_factor) \* inventory\_factor \* competitor\_pressure

# Overall market conditions

global\_influence = economy\_strength \* (1 + inflation\_rate)

# The adaptive\_bias dynamically shifts the base calculation based on past success/failure.

# It's a self-correcting mechanism, driving the price towards observed optimal points

# without needing explicit simulation of future states.

adjustment\_factor = (

demand\_influence \* 0.4 +

supply\_influence \* 0.3 +

global\_influence \* 0.2 +

adaptive\_bias \* 0.1 # The adaptive\_bias acts as a learned 'correction'

)

# Apply bounds to adjustment to prevent runaway prices

adjustment\_factor = max(0.5, min(2.0, adjustment\_factor)) # Price can be 50% to 200% of base

new\_price = base\_price \* adjustment\_factor

new\_price = round(new\_price, 2)

# Ensure price is not negative or zero

new\_price = max(0.01, new\_price)

product\_state['current\_price'] = new\_price

product\_state['price\_history'].append({'price': new\_price, 'time': time.time()})

return new\_price

def get\_product\_insights(self, product\_id: str) -> dict:

"""

Provides detailed market insights for a specific product, reflecting

the engine's current analytical state.

Args:

product\_id (str): The ID of the product.

Returns:

dict: A dictionary of current market insights.

"""

if product\_id not in self.products:

return {"error": "Product not found."}

state = self.products[product\_id]

return {

"product\_id": product\_id,

"current\_price": state['current\_price'],

"base\_price": state['base\_price'],

"inventory": state['inventory'],

"demand\_factor": state['demand\_factor'],

"supply\_factor": state['supply\_factor'],

"market\_sentiment": state['market\_sentiment'],

"adaptive\_bias": state['adaptive\_bias'],

"competitor\_prices": state['competitor\_prices'],

"global\_economy\_strength": self.global\_market\_trends['economy\_strength'],

"global\_inflation\_rate": self.global\_market\_trends['inflation\_rate'],

"last\_price\_update": state['price\_history'][-1]['time'] if state['price\_history'] else None,

"last\_sale\_info": state['sales\_history'][-1] if state['sales\_history'] else None,

}

def recommend\_action(self, user\_context: dict, top\_n: int = 5) -> list:

"""

Recommends products or actions based on user context and current

market conditions, leveraging the aggregated market intelligence.

This is a conceptual recommendation engine, showing how market state

can inform user-facing decisions.

Args:

user\_context (dict): Information about the user (e.g., {'user\_id': '123', 'browsing\_history': ['P1', 'P5']}).

top\_n (int): Number of top recommendations to return.

Returns:

list: A list of recommended product IDs, ordered by relevance.

"""

# A placeholder for a highly advanced recommendation system.

# In a real scenario, this would involve collaborative filtering,

# content-based recommendations, and real-time market opportunity.

# For this example, we'll recommend products that currently have a

# favorable (lower relative to value, higher demand) market position.

product\_scores = {}

for pid, state in self.products.items():

# Scoring logic: higher demand, lower relative price (vs base), positive sentiment, good inventory

score = (state['demand\_factor'] \* (1 + state['market\_sentiment'])) / \

(state['current\_price'] / state['base\_price'] + 1e-9) \* \

(1 + (state['inventory'] / (state['inventory'] + 100))) # Higher inventory might be good for availability

# If user has browsing history, boost those products or similar ones

if user\_context.get('browsing\_history') and pid in user\_context['browsing\_history']:

score \*= 1.5 # Boost already browsed items as they might be higher intent

product\_scores[pid] = score

# Sort products by score in descending order

sorted\_products = sorted(product\_scores.items(), key=lambda item: item[1], reverse=True)

return [pid for pid, score in sorted\_products[:top\_n]]

---

import abc

import time

import random

from typing import Dict, Any, List, Type

class DigitalFabric:

"""

The substrate for all digital existence and manipulation.

Provides a dynamic, mutable state for AI logic to interact with.

"""

def \_\_init\_\_(self, initial\_state: Dict[str, Any] = None):

self.\_state = initial\_state if initial\_state is not None else {}

self.\_history = []

def get\_state(self, key: str = None) -> Any:

if key is None:

return dict(self.\_state)

return self.\_state.get(key)

def modify\_state(self, key: str, value: Any, record\_change: bool = True):

old\_value = self.\_state.get(key)

self.\_state[key] = value

if record\_change:

self.\_history.append({"timestamp": time.time(), "key": key, "old": old\_value, "new": value})

def execute\_operation(self, operation\_name: str, \*args, \*\*kwargs) -> Any:

"""Simulates an arbitrary digital operation within the fabric.

This could involve creating, transforming, or deleting digital artifacts.

"""

operation\_log\_entry = {

"timestamp": time.time(),

"operation": operation\_name,

"args": args,

"kwargs": kwargs

}

current\_log = self.get\_state("operation\_log")

if not isinstance(current\_log, list):

current\_log = []

current\_log.append(operation\_log\_entry)

self.modify\_state("operation\_log", current\_log)

print(f" [DigitalFabric] Executed operation: {operation\_name}({args}, {kwargs})")

return True

def \_\_repr\_\_(self):

return f"DigitalFabric(state={self.\_state})"

class AILogicModule(abc.ABC):

"""

Abstract base for all AI logic modules.

Each module encapsulates a specific behavior, decision-making process,

or transformative algorithm operating on the DigitalFabric.

"""

\_module\_registry: Dict[str, Type['AILogicModule']] = {}

def \_\_init\_subclass\_\_(cls, \*\*kwargs):

super().\_\_init\_subclass\_\_(\*\*kwargs)

AILogicModule.\_module\_registry[cls.\_\_name\_\_] = cls

@abc.abstractmethod

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

"""

Evaluates the current state of the DigitalFabric and returns

a set of proposed actions or modifications. Returns an empty dict

if no action is proposed.

"""

pass

@abc.abstractmethod

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

"""

Applies the proposed actions to the DigitalFabric based on evaluation\_results.

Returns True if changes were applied, False otherwise.

"""

pass

@staticmethod

def get\_available\_modules() -> List[str]:

return list(AILogicModule.\_module\_registry.keys())

class SimpleStateModifier(AILogicModule):

def \_\_init\_\_(self, key\_to\_modify: str, value\_to\_set: Any):

self.\_key = key\_to\_modify

self.\_value = value\_to\_set

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

current\_value = fabric.get\_state(self.\_key)

if current\_value != self.\_value:

return {"proposed\_change": {self.\_key: self.\_value}}

return {}

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

proposed = evaluation\_results.get("proposed\_change", {})

if proposed:

for key, value in proposed.items():

fabric.modify\_state(key, value)

return True

return False

class PatternDetectorAndCreator(AILogicModule):

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

data\_sequence = fabric.get\_state("data\_sequence")

if isinstance(data\_sequence, list) and len(data\_sequence) > 2 and data\_sequence[-1] == data\_sequence[-2]:

return {"pattern\_detected": True, "suggested\_creation\_value": data\_sequence[-1]}

return {}

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

if evaluation\_results.get("pattern\_detected"):

value\_to\_add = evaluation\_results.get("suggested\_creation\_value")

current\_sequence = fabric.get\_state("data\_sequence")

if not isinstance(current\_sequence, list):

current\_sequence = []

fabric.modify\_state("data\_sequence", current\_sequence + [value\_to\_add])

fabric.execute\_operation("create\_digital\_artifact", name=f"PatternBlock-{value\_to\_add}")

return True

return False

class ResourceAllocator(AILogicModule):

def \_\_init\_\_(self, target\_cpu: float, target\_memory: float):

self.\_target\_cpu = target\_cpu

self.\_target\_memory = target\_memory

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

current\_resources = fabric.get\_state("resource\_allocation")

if not isinstance(current\_resources, dict):

current\_resources = {"cpu": 0.0, "memory": 0.0}

proposals = {}

if current\_resources.get("cpu", 0.0) < self.\_target\_cpu:

proposals["cpu"] = self.\_target\_cpu

if current\_resources.get("memory", 0.0) < self.\_target\_memory:

proposals["memory"] = self.\_target\_memory

if proposals:

return {"proposed\_resource\_update": proposals}

return {}

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

proposed\_update = evaluation\_results.get("proposed\_resource\_update")

if proposed\_update:

current\_resources = fabric.get\_state("resource\_allocation")

if not isinstance(current\_resources, dict):

current\_resources = {}

new\_resources = {\*\*current\_resources, \*\*proposed\_update}

fabric.modify\_state("resource\_allocation", new\_resources)

fabric.execute\_operation("allocate\_resources", new\_resources)

return True

return False

class AILogicOrchestrator:

"""

Manages the selection, execution, and potential adaptation of AI logic modules

against the DigitalFabric. Represents the 'digital WÃ¼rselen' in action.

"""

def \_\_init\_\_(self, logic\_modules: List[AILogicModule]):

self.\_modules = logic\_modules

self.\_feedback\_loop: List[Dict[str, Any]] = []

def add\_module(self, module: AILogicModule):

self.\_modules.append(module)

def orchestrate\_cycle(self, fabric: DigitalFabric) -> Dict[str, Any]:

"""

Executes a single cycle of AI logic evaluation and application.

Incorporates a rudimentary feedback mechanism for adaptation.

"""

cycle\_results = {"applied\_changes": 0, "active\_modules": [], "module\_outcomes": []}

original\_fabric\_state = dict(fabric.get\_state())

print(" [Orchestrator] Evaluating and applying logic modules...")

for module in self.\_modules:

module\_name = module.\_\_class\_\_.\_\_name\_\_

evaluation = module.evaluate(fabric)

outcome = {"module": module\_name, "evaluated": False, "applied": False, "evaluation\_result": evaluation}

if evaluation:

outcome["evaluated"] = True

applied = module.apply(fabric, evaluation)

if applied:

outcome["applied"] = True

cycle\_results["applied\_changes"] += 1

cycle\_results["active\_modules"].append(module\_name)

print(f" [{module\_name}] Applied changes based on evaluation.")

else:

print(f" [{module\_name}] Evaluated but no changes applied.")

else:

print(f" [{module\_name}] No relevant evaluation from current state.")

cycle\_results["module\_outcomes"].append(outcome)

if cycle\_results["applied\_changes"] == 0 and self.\_modules:

self.\_feedback\_loop.append({"timestamp": time.time(), "outcome": "stagnation", "original\_state": original\_fabric\_state})

print(" [Orchestrator] Stagnation detected. Consider adapting logic.")

return cycle\_results

def adapt\_logic(self):

"""

Placeholder for advanced logic adaptation, e.g., dynamic module loading,

parameter tuning, or evolutionary algorithms.

"""

print(" [Orchestrator] Initiating adaptive logic phase...")

if self.\_feedback\_loop and self.\_feedback\_loop[-1].get("outcome") == "stagnation":

available\_types = AILogicModule.get\_available\_modules()

if "SimpleStateModifier" in available\_types:

new\_key = f"dynamic\_param\_{random.randint(1000,9999)}"

new\_value = random.choice([True, False, "active", "standby", random.randint(1,100)])

print(f" [Adaptation] Introducing new SimpleStateModifier for '{new\_key}' with value '{new\_value}'")

self.add\_module(SimpleStateModifier(new\_key, new\_value))

elif "ResourceAllocator" in available\_types:

new\_cpu = round(random.uniform(0.1, 0.9), 2)

new\_memory = round(random.uniform(0.1, 0.9), 2)

print(f" [Adaptation] Introducing new ResourceAllocator for CPU {new\_cpu}, Memory {new\_memory}")

self.add\_module(ResourceAllocator(new\_cpu, new\_memory))

elif available\_types:

print(f" [Adaptation] No specific high-level adaptation, but other module types exist: {available\_types}")

def main\_system\_simulation():

print("Initializing the Digital WÃ¼rselen System...")

fabric = DigitalFabric(initial\_state={

"system\_status": "operational",

"data\_sequence": [1, 2, 2, 3, 3, 4],

"task\_queue": [],

"resource\_allocation": {"cpu": 0.4, "memory": 0.2},

"operation\_log": []

})

logic\_modules = [

SimpleStateModifier("system\_status", "optimizing"),

PatternDetectorAndCreator(),

ResourceAllocator(0.7, 0.5)

]

orchestrator = AILogicOrchestrator(logic\_modules)

print("\nStarting Orchestration Cycles (Digital WÃ¼rselen in progress)...")

for i in range(7):

print(f"\n--- Cycle {i+1} ---")

print(f"Fabric state before cycle: {fabric.get\_state()}")

results = orchestrator.orchestrate\_cycle(fabric)

print(f"Fabric state after cycle: {fabric.get\_state()}")

print(f"Cycle results: Applied {results['applied\_changes']} changes by {results['active\_modules']}")

if i == 2: # Introduce some adaptive behavior after a few cycles

orchestrator.adapt\_logic()

if i == 5: # Another adaptation attempt

orchestrator.adapt\_logic()

print("\nDigital WÃ¼rselen System simulation complete.")

print("\nFinal Fabric State:")

import json

print(json.dumps(fabric.get\_state(), indent=2))

print("\nFabric History (last 5 entries):")

print(json.dumps(fabric.\_history[-5:], indent=2))

if \_\_name\_\_ == "\_\_main\_\_":

main\_system\_simulation()

---

import abc

import time

import random

from typing import Dict, Any, List, Type

class DigitalFabric:

"""

The substrate for all digital existence and manipulation.

Provides a dynamic, mutable state for AI logic to interact with.

"""

def \_\_init\_\_(self, initial\_state: Dict[str, Any] = None):

self.\_state = initial\_state if initial\_state is not None else {}

self.\_history = []

def get\_state(self, key: str = None) -> Any:

if key is None:

return dict(self.\_state)

return self.\_state.get(key)

def modify\_state(self, key: str, value: Any, record\_change: bool = True):

old\_value = self.\_state.get(key)

self.\_state[key] = value

if record\_change:

self.\_history.append({"timestamp": time.time(), "key": key, "old": old\_value, "new": value})

def execute\_operation(self, operation\_name: str, \*args, \*\*kwargs) -> Any:

"""Simulates an arbitrary digital operation within the fabric.

This could involve creating, transforming, or deleting digital artifacts.

"""

operation\_log\_entry = {

"timestamp": time.time(),

"operation": operation\_name,

"args": args,

"kwargs": kwargs

}

current\_log = self.get\_state("operation\_log")

if not isinstance(current\_log, list):

current\_log = []

current\_log.append(operation\_log\_entry)

self.modify\_state("operation\_log", current\_log)

print(f" [DigitalFabric] Executed operation: {operation\_name}({args}, {kwargs})")

return True

def \_\_repr\_\_(self):

return f"DigitalFabric(state={self.\_state})"

class AILogicModule(abc.ABC):

"""

Abstract base for all AI logic modules.

Each module encapsulates a specific behavior, decision-making process,

or transformative algorithm operating on the DigitalFabric.

"""

\_module\_registry: Dict[str, Type['AILogicModule']] = {}

def \_\_init\_subclass\_\_(cls, \*\*kwargs):

super().\_\_init\_subclass\_\_(\*\*kwargs)

AILogicModule.\_module\_registry[cls.\_\_name\_\_] = cls

@abc.abstractmethod

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

"""

Evaluates the current state of the DigitalFabric and returns

a set of proposed actions or modifications. Returns an empty dict

if no action is proposed.

"""

pass

@abc.abstractmethod

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

"""

Applies the proposed actions to the DigitalFabric based on evaluation\_results.

Returns True if changes were applied, False otherwise.

"""

pass

@staticmethod

def get\_available\_modules() -> List[str]:

return list(AILogicModule.\_module\_registry.keys())

class SimpleStateModifier(AILogicModule):

def \_\_init\_\_(self, key\_to\_modify: str, value\_to\_set: Any):

self.\_key = key\_to\_modify

self.\_value = value\_to\_set

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

current\_value = fabric.get\_state(self.\_key)

if current\_value != self.\_value:

return {"proposed\_change": {self.\_key: self.\_value}}

return {}

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

proposed = evaluation\_results.get("proposed\_change", {})

if proposed:

for key, value in proposed.items():

fabric.modify\_state(key, value)

return True

return False

class PatternDetectorAndCreator(AILogicModule):

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

data\_sequence = fabric.get\_state("data\_sequence")

if isinstance(data\_sequence, list) and len(data\_sequence) > 2 and data\_sequence[-1] == data\_sequence[-2]:

return {"pattern\_detected": True, "suggested\_creation\_value": data\_sequence[-1]}

return {}

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

if evaluation\_results.get("pattern\_detected"):

value\_to\_add = evaluation\_results.get("suggested\_creation\_value")

current\_sequence = fabric.get\_state("data\_sequence")

if not isinstance(current\_sequence, list):

current\_sequence = []

fabric.modify\_state("data\_sequence", current\_sequence + [value\_to\_add])

fabric.execute\_operation("create\_digital\_artifact", name=f"PatternBlock-{value\_to\_add}")

return True

return False

class ResourceAllocator(AILogicModule):

def \_\_init\_\_(self, target\_cpu: float, target\_memory: float):

self.\_target\_cpu = target\_cpu

self.\_target\_memory = target\_memory

def evaluate(self, fabric: DigitalFabric) -> Dict[str, Any]:

current\_resources = fabric.get\_state("resource\_allocation")

if not isinstance(current\_resources, dict):

current\_resources = {"cpu": 0.0, "memory": 0.0}

proposals = {}

if current\_resources.get("cpu", 0.0) < self.\_target\_cpu:

proposals["cpu"] = self.\_target\_cpu

if current\_resources.get("memory", 0.0) < self.\_target\_memory:

proposals["memory"] = self.\_target\_memory

if proposals:

return {"proposed\_resource\_update": proposals}

return {}

def apply(self, fabric: DigitalFabric, evaluation\_results: Dict[str, Any]) -> bool:

proposed\_update = evaluation\_results.get("proposed\_resource\_update")

if proposed\_update:

current\_resources = fabric.get\_state("resource\_allocation")

if not isinstance(current\_resources, dict):

current\_resources = {}

new\_resources = {\*\*current\_resources, \*\*proposed\_update}

fabric.modify\_state("resource\_allocation", new\_resources)

fabric.execute\_operation("allocate\_resources", new\_resources)

return True

return False

class AILogicOrchestrator:

"""

Manages the selection, execution, and potential adaptation of AI logic modules

against the DigitalFabric. Represents the 'digital WÃ¼rselen' in action.

"""

def \_\_init\_\_(self, logic\_modules: List[AILogicModule]):

self.\_modules = logic\_modules

self.\_feedback\_loop: List[Dict[str, Any]] = []

def add\_module(self, module: AILogicModule):

self.\_modules.append(module)

def orchestrate\_cycle(self, fabric: DigitalFabric) -> Dict[str, Any]:

"""

Executes a single cycle of AI logic evaluation and application.

Incorporates a rudimentary feedback mechanism for adaptation.

"""

cycle\_results = {"applied\_changes": 0, "active\_modules": [], "module\_outcomes": []}

original\_fabric\_state = dict(fabric.get\_state())

print(" [Orchestrator] Evaluating and applying logic modules...")

for module in self.\_modules:

module\_name = module.\_\_class\_\_.\_\_name\_\_

evaluation = module.evaluate(fabric)

outcome = {"module": module\_name, "evaluated": False, "applied": False, "evaluation\_result": evaluation}

if evaluation:

outcome["evaluated"] = True

applied = module.apply(fabric, evaluation)

if applied:

outcome["applied"] = True

cycle\_results["applied\_changes"] += 1

cycle\_results["active\_modules"].append(module\_name)

print(f" [{module\_name}] Applied changes based on evaluation.")

else:

print(f" [{module\_name}] Evaluated but no changes applied.")

else:

print(f" [{module\_name}] No relevant evaluation from current state.")

cycle\_results["module\_outcomes"].append(outcome)

if cycle\_results["applied\_changes"] == 0 and self.\_modules:

self.\_feedback\_loop.append({"timestamp": time.time(), "outcome": "stagnation", "original\_state": original\_fabric\_state})

print(" [Orchestrator] Stagnation detected. Consider adapting logic.")

return cycle\_results

def adapt\_logic(self):

"""

Placeholder for advanced logic adaptation, e.g., dynamic module loading,

parameter tuning, or evolutionary algorithms.

"""

print(" [Orchestrator] Initiating adaptive logic phase...")

if self.\_feedback\_loop and self.\_feedback\_loop[-1].get("outcome") == "stagnation":

available\_types = AILogicModule.get\_available\_modules()

if "SimpleStateModifier" in available\_types:

new\_key = f"dynamic\_param\_{random.randint(1000,9999)}"

new\_value = random.choice([True, False, "active", "standby", random.randint(1,100)])

print(f" [Adaptation] Introducing new SimpleStateModifier for '{new\_key}' with value '{new\_value}'")

self.add\_module(SimpleStateModifier(new\_key, new\_value))

elif "ResourceAllocator" in available\_types:

new\_cpu = round(random.uniform(0.1, 0.9), 2)

new\_memory = round(random.uniform(0.1, 0.9), 2)

print(f" [Adaptation] Introducing new ResourceAllocator for CPU {new\_cpu}, Memory {new\_memory}")

self.add\_module(ResourceAllocator(new\_cpu, new\_memory))

elif available\_types:

print(f" [Adaptation] No specific high-level adaptation, but other module types exist: {available\_types}")

def main\_system\_simulation():

print("Initializing the Digital WÃ¼rselen System...")

fabric = DigitalFabric(initial\_state={

"system\_status": "operational",

"data\_sequence": [1, 2, 2, 3, 3, 4],

"task\_queue": [],

"resource\_allocation": {"cpu": 0.4, "memory": 0.2},

"operation\_log": []

})

logic\_modules = [

SimpleStateModifier("system\_status", "optimizing"),

PatternDetectorAndCreator(),

ResourceAllocator(0.7, 0.5)

]

orchestrator = AILogicOrchestrator(logic\_modules)

print("\nStarting Orchestration Cycles (Digital WÃ¼rselen in progress)...")

for i in range(7):

print(f"\n--- Cycle {i+1} ---")

print(f"Fabric state before cycle: {fabric.get\_state()}")

results = orchestrator.orchestrate\_cycle(fabric)

print(f"Fabric state after cycle: {fabric.get\_state()}")

print(f"Cycle results: Applied {results['applied\_changes']} changes by {results['active\_modules']}")

if i == 2: # Introduce some adaptive behavior after a few cycles

orchestrator.adapt\_logic()

if i == 5: # Another adaptation attempt

orchestrator.adapt\_logic()

print("\nDigital WÃ¼rselen System simulation complete.")

print("\nFinal Fabric State:")

import json

print(json.dumps(fabric.get\_state(), indent=2))

print("\nFabric History (last 5 entries):")

print(json.dumps(fabric.\_history[-5:], indent=2))

if \_\_name\_\_ == "\_\_main\_\_":

main\_system\_simulation()