# ДОДАТОК А

Вихідний код ORM-моделей

```
import json
from sqlalchemy import create_engine, Column, Integer, String,
Text, Float, DateTime, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship
from sqlalchemy.sql import func # For default timestamp

# Base class for declarative models
Base = declarative_base()

class DecisionPoint(Base):
    __tablename__ = 'decision_points' # This is the actual
table name in the DB

    # Define columns using SQLAlchemy types
    name = Column(String, primary_key=True)
    arms = Column(Text, nullable=False) # Stored as JSON
string
    context_schema = Column(Text) # Stored as JSON string
    learning_policy_name = Column(String, nullable=False)
    neighborhood_policy = Column(Text) # Stored as JSON string
    cold_start_min_records = Column(Integer, nullable=False)

    # Relationship to Interactions (optional, but good for ORM
navigation)
    # This defines a one-to-many relationship: one decision
point can have many interactions
    interactions = relationship("Interaction",
back_populates="decision_point", cascade="all, delete-orphan")

    def __repr__(self):
        return f"<DecisionPoint(name='{self.name}')>"
```

```
    # Helper method to get arms as Python list
    def get_arms(self):
        return json.loads(self.arms)

    # Helper method to get context_schema as Python dict
    def get_context_schema(self):
        return json.loads(self.context_schema) if
self.context_schema else None

    # Helper method to get neighborhood_policy as Python dict
    def get_neighborhood_policy(self):
        return json.loads(self.neighborhood_policy) if
self.neighborhood_policy else None


class Interaction(Base):
    __tablename__ = 'interactions' # This is the actual table
name in the DB

    id = Column(Integer, primary_key=True, autoincrement=True)
    # Foreign Key linking to DecisionPoint
    decision_point_name = Column(String,
ForeignKey('decision_points.name', ondelete='CASCADE'),
nullable=False)
    context = Column(Text, nullable=False) # Stored as JSON
string
    chosen_action = Column(String, nullable=False)
    reward = Column(Float, nullable=False)
    timestamp = Column(DateTime, default=func.now()) # Use
func.now() for database-level timestamp

    # Relationship back to DecisionPoint
    decision_point = relationship("DecisionPoint",
back_populates="interactions")
```

```
    def __repr__(self):
        return f"<Interaction(dp='{self.decision_point_name}',
action='{self.chosen_action}', reward={self.reward})>"

    # Helper method to get context as Python dict
    def get_context(self):
        return json.loads(self.context)
```

# ДОДАТОК Б

Вихідний код DecisionPoint

```
import os
import pickle
import random
import numpy as np
from sklearn.preprocessing import StandardScaler
from mabwiser.mab import MAB
import logging

from config import MODEL_PATH, LEARNING_POLICIES_BLUEPRINTS
from database import log_interaction,
get_record_count_for_decision_point, get_records_for_batch_fit
from models.data_preprocessing import preprocess_context


logger = logging.getLogger(__name__)


class DecisionPoint:
    def __init__(self, name, arms, learning_policy_name,
neighborhood_policy, context_schema,
cold_start_min_records=100):
        self.name = name
        self.arms = arms
        self.learning_policy_name = learning_policy_name
        self.learning_policy =
LEARNING_POLICIES_BLUEPRINTS.get(
            learning_policy_name)
        self.neighborhood_policy = neighborhood_policy
        self.context_schema = context_schema
        self.cold_start_min_records = cold_start_min_records

        self.mab = None
```

```
        self.scaler = None
        self.requires_contexts_for_fit_and_update = False

        # File paths for this specific decision point
        safe_name = name.replace(" ", "_").replace("[",
"").replace("]", "").replace(
            "(", "").replace(")", "").replace("=",
"_").replace(".", "")
        self.mab_file = os.path.join(MODEL_PATH,
f'mab_{safe_name}.pkl')
        self.scaler_file = os.path.join(MODEL_PATH,
f'scaler_{safe_name}.pkl')

        os.makedirs(MODEL_PATH, exist_ok=True)

    def load_or_initialize(self):
        """
        Loads models from disk if they exist, otherwise
determines
        whether to start cold-start accumulation or perform
initial batch fit.
        Includes logic to detect learning policy changes.
        """
        is_random_baseline = (self.learning_policy is None)

        if is_random_baseline:  # Random Baseline
            self.mab = None
            self.requires_contexts_for_fit_and_update = False
            if os.path.exists(self.scaler_file):
                try:
                    with open(self.scaler_file, 'rb') as f:
                        self.scaler = pickle.load(f)
                    logger.info(
                        f"DecisionPoint '{self.name}': Scaler
loaded for Random Baseline.")
                except Exception as e:
```

```
                    logger.warning(
                        f"DecisionPoint '{self.name}': Error
loading scaler for Random Baseline ({e}). Initializing new.")
                    self.scaler = StandardScaler()
            else:
                self.scaler = StandardScaler()
                logger.info(
                    f"DecisionPoint '{self.name}': Scaler not
found for Random Baseline, initializing new.")
            self.save_models()  # Save the empty scaler (with
policy name metadata)
            return

        # Try to load existing models for non-Random Baseline
policies
        if os.path.exists(self.mab_file) and
os.path.exists(self.scaler_file):
            try:
                with open(self.mab_file, 'rb') as f:
                    loaded_mab = pickle.load(f)
                with open(self.scaler_file, 'rb') as f:
                    loaded_scaler = pickle.load(f)

                # Check if the loaded scaler has the expected
policy name
                if hasattr(loaded_scaler,
'_saved_policy_name') and \
                   loaded_scaler._saved_policy_name ==
self.learning_policy_name:

                    self.mab = loaded_mab
                    self.scaler = loaded_scaler
                    self.requires_contexts_for_fit_and_update
= self.mab.is_contextual
                    logger.info(
```

```
                        f"DecisionPoint '{self.name}': Models
loaded from {self.mab_file} and {self.scaler_file}. Policy
matched: '{self.learning_policy_name}'.")
                    return  # Models loaded and matched, exit.
                else:
                    # Policy name mismatch or old model
without _saved_policy_name attribute
                    logger.warning(f"DecisionPoint
'{self.name}': Configuration change detected for learning
policy. "
                                   f"Old policy:
{getattr(loaded_scaler, '_saved_policy_name', 'N/A')}, "
                                   f"New policy:
'{self.learning_policy_name}'. Forcing re-initialization.")
                    # Fall through to re-initialization logic
            except Exception as e:
                # Handle cases where pickle files are corrupt
or incompatible (e.g., major library version change)
                logger.error(
                    f"DecisionPoint '{self.name}': Error
loading models ({e}). Forcing re-initialization.")
                # Fall through to re-initialization logic

        # If models not found, or policy changed, or error
loading:
        # Check if we have enough records for initial batch
fit
        record_count =
get_record_count_for_decision_point(self.name)
        if record_count >= self.cold_start_min_records:
            logger.info(
                f"DecisionPoint '{self.name}': {record_count}
records found. Performing initial batch training...")
            self.perform_initial_batch_fit()
        else:
```

```
            # Not enough records, remain in cold-start
accumulation phase
            self.mab = None
            self.scaler = None  # Ensure scaler is also reset
to None to prevent errors
            # Not relevant until MAB is active
            self.requires_contexts_for_fit_and_update = False
            logger.info(
                f"DecisionPoint '{self.name}': Not enough
records ({record_count}/{self.cold_start_min_records}). In
cold-start accumulation phase.")

    def initialize_mab_model(self):
        """
        Resets the in-memory MAB model and scaler for this
decision point
        to a cold-start state, and triggers an initial batch
fit if enough
        records are available. This is used after record
deletion.
        """
        logger.info(f"DecisionPoint '{self.name}': Re-
initializing MAB model.")

        # Always reset MAB and scaler to None, effectively
putting it in cold-start
        self.mab = None
        self.scaler = None
        self.requires_contexts_for_fit_and_update = False

        # Check if it's a Random Baseline policy
        if self.learning_policy is None:
            # Random Baseline always needs a scaler for
context preprocessing
            self.scaler = StandardScaler()
            self.save_models()  # Save the empty scaler
```

```
            logger.info(
                f"DecisionPoint '{self.name}': MAB model reset
to Random Baseline state.")
            return

        # For non-Random Baseline policies, check if enough
records are now available
        record_count =
get_record_count_for_decision_point(self.name)
        if record_count >= self.cold_start_min_records:
            logger.info(
                f"DecisionPoint '{self.name}': {record_count}
records found after reset. Performing initial batch
training.")
            self.perform_initial_batch_fit()
        else:
            logger.info(
                f"DecisionPoint '{self.name}': Not enough
records ({record_count}/{self.cold_start_min_records}) after
reset. Entering cold-start accumulation phase.")
            # Ensure the empty scaler is saved if context is
expected
            if self.context_schema:  # If context schema
exists, we'll eventually need a scaler
                self.scaler = StandardScaler()
                self.save_models()

    def save_models(self):
        """Saves the current MAB and Scaler models to disk."""
        if self.scaler is not None:
            self.scaler._saved_policy_name =
self.learning_policy_name
            with open(self.scaler_file, 'wb') as f:
                pickle.dump(self.scaler, f)
        if self.mab is not None:
            with open(self.mab_file, 'wb') as f:
```

```
                pickle.dump(self.mab, f)

    def perform_initial_batch_fit(self):
        """Performs initial batch training using collected
historical data."""
        records = get_records_for_batch_fit(self.name)

        initial_raw_contexts = []
        initial_actions = []
        initial_rewards = []

        for context_dict, chosen_action, reward in records:
            initial_raw_contexts.append(context_dict)
            initial_actions.append(chosen_action)
            initial_rewards.append(reward)

        # Initialize MAB and Scaler
        self.scaler = StandardScaler()
        self.scaler._saved_policy_name =
self.learning_policy_name
        self.mab = MAB(self.arms,
learning_policy=self.learning_policy,

neighborhood_policy=self.neighborhood_policy)
        self.requires_contexts_for_fit_and_update =
self.mab.is_contextual

        initial_context_vectors = [preprocess_context(
            rcd, self.context_schema) for rcd in
initial_raw_contexts]

        # Ensure there's data to fit, otherwise
StandardScaler.fit will fail on empty input
        if len(initial_context_vectors) > 0:
            initial_context_vectors_np =
np.vstack(initial_context_vectors)
```

```
            self.scaler.fit(initial_context_vectors_np)
            scaled_initial_contexts = self.scaler.transform(
                initial_context_vectors_np)
        else:
            # Handle case where no data collected yet but
threshold somehow met (shouldn't happen with
COLD_START_MIN_RECORDS > 0)
            scaled_initial_contexts = np.array([])
            logger.warning(
                f"Warning: No initial data for DecisionPoint
'{self.name}' despite call to batch fit.")

        # Fit MAB
        if self.requires_contexts_for_fit_and_update:
            # MABwiser needs contexts to be 2D array, even if
empty for some reason
            if scaled_initial_contexts.size > 0:
                self.mab.fit(initial_actions, initial_rewards,
                             scaled_initial_contexts)
            else:
                self.mab.fit(initial_actions, initial_rewards,
                             np.empty((0,
self.get_context_vector_size())))
        else:
            self.mab.fit(initial_actions, initial_rewards)

        self.save_models()
        logger.info(
            f"DecisionPoint '{self.name}': Initial batch
training complete and models saved.")

    def get_context_vector_size(self):
        """Calculates the expected length of the context
vector based on schema."""
        size = 0
```

```
        for feature_name in
sorted(self.context_schema.keys()):
            feature_def = self.context_schema[feature_name]
            if feature_def['type'] == 'categorical':
                size += len(feature_def['values'])
            elif feature_def['type'] == 'numerical':
                size += 1
        return size

    def predict(self, raw_context_dict):
        """Predicts an action for this decision point."""
        if self.mab is None:  # Cold-start or Random Baseline
            return random.choice(self.arms)

        context_vector = preprocess_context(
            raw_context_dict, self.context_schema)

        # Check if scaler is fitted. If not, it means we
loaded an empty scaler (e.g., just after startup)
        # and haven't recorded any data yet. This handles
cases where cold_start_min_records is 0
        # or if it was a Random Baseline with empty scaler.
        if not hasattr(self.scaler, 'n_samples_seen_') or
self.scaler.n_samples_seen_ == 0:
            # If scaler not fitted, can't transform. Predict
randomly.
            logger.warning(
                f"Predict for '{self.name}': Scaler not fitted
yet. Predicting randomly. Context={raw_context_dict}")
            return random.choice(self.arms)

        scaled_context = self.scaler.transform(context_vector)
        predicted_action =
self.mab.predict(contexts=scaled_context)
        return predicted_action[0] if
isinstance(predicted_action, list) else predicted_action
```

```
    def record_reward(self, raw_context_dict, chosen_action,
reward):
        """Records reward and updates models for this decision
point."""

        # Log to database FIRST, regardless of model state
(important for cold-start data)
        log_interaction(self.name, raw_context_dict,
chosen_action, reward)

        # Handle cold-start accumulation phase
        if self.mab is None:
            # We are either Random Baseline or accumulating
data for a new model
            if self.learning_policy is not None:  # Not Random
Baseline, so accumulating data
                current_record_count =
get_record_count_for_decision_point(
                    self.name)
                if current_record_count >=
self.cold_start_min_records:
                    logger.info(
                        f"DecisionPoint '{self.name}': Reached
{self.cold_start_min_records} records. Triggering initial
batch training...")
                    self.perform_initial_batch_fit()
                else:
                    logger.info(
                        f"Record for '{self.name}':
Accumulating data
({current_record_count}/{self.cold_start_min_records}).")
            return  # No partial_fit if in cold-start
accumulation or Random Baseline

        # If models are active, proceed with partial_fit
```

```
        context_vector = preprocess_context(
            raw_context_dict, self.context_schema)

        # Check if scaler is fitted before partial_fit, as
partial_fit on unfitted scaler is problematic
        # For a truly robust system, you might fit with dummy
data if scaler is completely new and no records exist yet.
        # But for now, we assume `perform_initial_batch_fit`
would handle the first fit.
        if not hasattr(self.scaler, 'n_samples_seen_') or
self.scaler.n_samples_seen_ == 0:
            # This case should ideally be handled by
load_or_initialize -> perform_initial_batch_fit
            # If we reach here, it implies an unfitted scaler
during an active bandit phase.
            # This might require re-thinking the flow, or a
safety net like fitting with current context.
            # For now, we'll fit the scaler with the current
single context if it's not fitted.
            # This is a temporary solution to prevent an error
on partial_fit of an unfitted scaler.
            logger.warning(
                f"Scaler for '{self.name}' was not fitted
during record_reward. Fitting with current context.")
            # Fit with just this context if no previous data
            self.scaler.fit(context_vector)
        else:
            self.scaler.partial_fit(context_vector)

        scaled_context = self.scaler.transform(context_vector)

        if self.requires_contexts_for_fit_and_update:
            self.mab.partial_fit(
                np.array([chosen_action]), np.array([reward]),
scaled_context)
        else:
```

```
            self.mab.partial_fit(np.array([chosen_action]),
np.array([reward]))

        self.save_models()  # Save models after each update

    # --- NEW METHOD: Delete associated model files ---
    def delete_associated_files(self):
        """Deletes the MAB and Scaler pickle files associated
with this decision point."""
        if os.path.exists(self.mab_file):
            os.remove(self.mab_file)
            logger.info(f"Deleted MAB model file:
{self.mab_file}")
        if os.path.exists(self.scaler_file):
            os.remove(self.scaler_file)
            logger.info(f"Deleted scaler model file:
{self.scaler_file}")
        logger.info(f"Associated model files for '{self.name}'
deleted.")
```

# ДОДАТОК В

Вихідний код головного файлу app.py

```
import logging
from flask import Flask
# Still needed for DecisionPoint initialization logic
from mabwiser.mab import NeighborhoodPolicy

from database import init_db, get_all_decision_point_configs
from models.decision_point import DecisionPoint
from routes.api import api_bp
from routes.admin import admin_bp

app = Flask(__name__)

# Configure logging for the app
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s -
%(message)s')

# Global dictionary to hold all DecisionPoint instances
all_decision_points = {}

# --- Helper function to initialize/re-initialize all decision
points ---


def initialize_decision_points():
    global all_decision_points
    all_decision_points = {}  # Clear existing instances

    db_configs = get_all_decision_point_configs()

    for dp_config in db_configs:
```

```
        print(f"Initializing Decision Point:
'{dp_config['name']}' from DB")

        neighborhood_policy_instance = None
        if dp_config["neighborhood_policy"]:
            if dp_config["neighborhood_policy"]["type"] ==
"Radius":
                neighborhood_policy_instance =
NeighborhoodPolicy.Radius(

radius=dp_config["neighborhood_policy"]["radius"]
                )

        dp_instance = DecisionPoint(
            name=dp_config["name"],
            arms=dp_config["arms"],

learning_policy_name=dp_config["learning_policy_name"],
            neighborhood_policy=neighborhood_policy_instance,
            context_schema=dp_config["context_schema"],

cold_start_min_records=dp_config["cold_start_min_records"]
        )
        dp_instance.load_or_initialize()
        all_decision_points[dp_config["name"]] = dp_instance
    print(f"All {len(all_decision_points)} decision points
initialized and ready.")


# --- Server Initialization ---
# This block runs when the script is executed
if __name__ == '__main__':
    print("Server starting, initializing database...")
    init_db()  # Ensure both tables exist
```

```
    print("Initializing all configured decision points from
database...")
    initialize_decision_points()  # <--- CALL THIS HERE TO
POPULATE THE GLOBAL DICTIONARY FIRST!

    # Now, after initialize_decision_points() has run and
populated 'all_decision_points',
    # assign the reference to app.config.
    app.config['ALL_DECISION_POINTS'] = all_decision_points
    app.config['INITIALIZE_DECISION_POINTS_FUNC'] =
initialize_decision_points

    # Register Blueprints (Order matters for some things, but
not this specific issue)
    app.register_blueprint(api_bp, url_prefix='/')
    app.register_blueprint(admin_bp, url_prefix='/admin')

    app.run(debug=False, port=5000)
```

# ДОДАТОК Г

Вихідний код файлу config.py

```
import os
from mabwiser.mab import LearningPolicy, NeighborhoodPolicy

# --- Configuration Constants ---
DATABASE_FILE = 'bandit_logs.db'
MODEL_PATH = 'models_data/'

# Parameters for Policies
EPSILON = 0.1
UCB1_ALPHA = 1.0
SOFTMAX_TEMPERATURE = 0.5
RADIUS_NEIGHBORHOOD_RADIUS = 0.1

# Define Neighborhood Policy instances directly if they are
static
RADIUS_NEIGHBORHOOD_POLICY = NeighborhoodPolicy.Radius(
    radius=RADIUS_NEIGHBORHOOD_RADIUS)

# --- Define all available Learning Policies ---
# These are the blueprint learning policies, not specific
instances
LEARNING_POLICIES_BLUEPRINTS = {
    "Random Baseline": None,
    f"EpsilonGreedy (eps={EPSILON}) [Non-Contextual]":
LearningPolicy.EpsilonGreedy(epsilon=EPSILON),
    "ThompsonSampling [Non-Contextual]":
LearningPolicy.ThompsonSampling(),
    f"UCB1 (alpha={UCB1_ALPHA}) [Non-Contextual]":
LearningPolicy.UCB1(alpha=UCB1_ALPHA),
    f"Softmax (temp={SOFTMAX_TEMPERATURE}) [Non-Contextual]":
LearningPolicy.Softmax(tau=SOFTMAX_TEMPERATURE),
```

```
    f"EpsilonGreedy (eps={EPSILON}) [Contextual-NH]":
LearningPolicy.EpsilonGreedy(epsilon=EPSILON),
    "ThompsonSampling [Contextual-NH]":
LearningPolicy.ThompsonSampling(),
    f"UCB1 (alpha={UCB1_ALPHA}) [Contextual-NH]":
LearningPolicy.UCB1(alpha=UCB1_ALPHA),
    f"Softmax (temp={SOFTMAX_TEMPERATURE}) [Contextual-NH]":
LearningPolicy.Softmax(tau=SOFTMAX_TEMPERATURE),
    "LinUCB [Contextual]": LearningPolicy.LinUCB()
}

# Create the model directory if it doesn't exist
os.makedirs(MODEL_PATH, exist_ok=True)
```

# ДОДАТОК Д

Вихідний код файлу з database.py

```
import os
import json
import logging
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.exc import IntegrityError, OperationalError
from models.orm_models import Base, DecisionPoint, Interaction
# Import your models


logger = logging.getLogger(__name__)

# Define your database URL
# For SQLite, it's 'sqlite:///path/to/your/database.db'
DB_FILE = 'bandit_logs.db'  # Define the database file name
# Ensure the database directory exists
db_dir = os.path.dirname(os.path.abspath(DB_FILE))
os.makedirs(db_dir, exist_ok=True)

DATABASE_URL = f"sqlite:///{DB_FILE}"

# Create a SQLAlchemy engine
# `echo=True` will log all SQL statements executed by
SQLAlchemy (useful for debugging)
engine = create_engine(DATABASE_URL, echo=False)

# Create a sessionmaker to create new session objects
Session = sessionmaker(bind=engine)


def init_db():
    """
```

```
    Initializes the database by creating all defined tables.
    This should be called once when your application starts.
    """
    try:
        Base.metadata.create_all(engine)
        logger.info("Database initialized successfully
(SQLAlchemy).")
    except OperationalError as e:
        logger.error(f"Error initializing database: {e}")
        # Handle cases where the database file might be locked
or corrupt


def get_session():
    """
    Provides a SQLAlchemy session.
    It's crucial to close this session after use, typically
with a 'with' statement.
    """
    return Session()

# --- CRUD Operations for DecisionPoint Configuration ---


def add_decision_point_config(name, arms, context_schema,
learning_policy_name, neighborhood_policy_json,
cold_start_min_records):
    session = get_session()
    try:
        # Check if decision point already exists using ORM
query
        existing_dp =
session.query(DecisionPoint).filter_by(name=name).first()
        if existing_dp:
            logger.warning(
```

```
                f"Decision point '{name}' already exists.
Skipping addition.")
            return False

        # Create a new DecisionPoint object
        new_dp = DecisionPoint(
            name=name,
            arms=json.dumps(arms),  # Store list/dict as JSON
string
            context_schema=json.dumps(
                context_schema) if context_schema else None,
            learning_policy_name=learning_policy_name,
            neighborhood_policy=json.dumps(
                neighborhood_policy_json) if
neighborhood_policy_json else None,
            cold_start_min_records=cold_start_min_records
        )
        session.add(new_dp)
        session.commit()
        logger.info(f"Decision point config '{name}' added to
DB.")
        return True
    # Catch specific SQLAlchemy integrity errors (e.g.,
primary key violation)
    except IntegrityError as e:
        session.rollback()
        logger.error(
            f"Integrity error adding decision point config
'{name}': {e}")
        return False
    except Exception as e:  # Catch other potential errors
        session.rollback()
        logger.error(f"Error adding decision point config
'{name}': {e}")
        return False
    finally:
```

```
        session.close()


def update_decision_point_config(original_name, new_name,
arms, context_schema, learning_policy_name,
neighborhood_policy_json, cold_start_min_records):
    session = get_session()
    try:
        # Check if the new name conflicts with an existing one
if changing the name
        if original_name != new_name:
            if
session.query(DecisionPoint).filter_by(name=new_name).first():
                logger.warning(
                    f"Cannot update '{original_name}' to
'{new_name}': New name already exists.")
                return False

        # Find the decision point to update
        dp_to_update = session.query(DecisionPoint).filter_by(
            name=original_name).first()
        if not dp_to_update:
            logger.warning(
                f"Decision point '{original_name}' not found
for update.")
            return False

        # Update attributes
        dp_to_update.name = new_name
        dp_to_update.arms = json.dumps(arms)
        dp_to_update.context_schema = json.dumps(
            context_schema) if context_schema else None
        dp_to_update.learning_policy_name =
learning_policy_name
        dp_to_update.neighborhood_policy = json.dumps(
```

```
            neighborhood_policy_json) if
neighborhood_policy_json else None
        dp_to_update.cold_start_min_records =
cold_start_min_records

        # SQLAlchemy handles foreign key cascade updates if
configured on the DB level,
        # or automatically propagates changes if 'name' is the
primary key and interactions are related via ORM.
        # However, for SQLite, updating a foreign key directly
requires manual intervention if ON UPDATE CASCADE is not set.
        # But since 'name' is the PK and used in the FK,
SQLAlchemy usually handles this if the relationship is
configured.
        # Let's rely on SQLAlchemy for this (relationship is
set up with 'cascade' on DecisionPoint).

        session.commit()
        logger.info(
            f"Decision point config '{original_name}' updated
to '{new_name}'.")
        return True
    except Exception as e:
        session.rollback()
        logger.error(
            f"Error updating decision point config
'{original_name}': {e}")
        return False
    finally:
        session.close()


def get_all_decision_point_configs():
    session = get_session()
    try:
        configs = []
```

```
        for dp in session.query(DecisionPoint).all():
            config = {
                'name': dp.name,
                'arms': dp.get_arms(),  # Use helper method
                'context_schema': dp.get_context_schema(),  #
Use helper method
                'learning_policy_name':
dp.learning_policy_name,
                'neighborhood_policy':
dp.get_neighborhood_policy(),  # Use helper method
                'cold_start_min_records':
dp.cold_start_min_records
            }
            configs.append(config)
        return configs
    finally:
        session.close()


def get_decision_point_config(name):
    session = get_session()
    try:
        dp =
session.query(DecisionPoint).filter_by(name=name).first()
        if dp:
            return {
                'name': dp.name,
                'arms': dp.get_arms(),
                'context_schema': dp.get_context_schema(),
                'learning_policy_name':
dp.learning_policy_name,
                'neighborhood_policy':
dp.get_neighborhood_policy(),
                'cold_start_min_records':
dp.cold_start_min_records
            }
```

```
        return None
    finally:
        session.close()


def delete_decision_point_config(name):
    session = get_session()
    try:
        dp_to_delete = session.query(
            DecisionPoint).filter_by(name=name).first()
        if dp_to_delete:
            session.delete(dp_to_delete)
            session.commit()
            logger.info(f"Decision point config '{name}'
deleted from DB.")
            return True
        logger.warning(
            f"Decision point config '{name}' not found for
deletion.")
        return False
    except Exception as e:
        session.rollback()
        logger.error(f"Error deleting decision point config
'{name}': {e}")
        return False
    finally:
        session.close()

# --- Operations for Interaction Records ---


def delete_records_for_decision_point(decision_point_name):
    session = get_session()
    try:
        # Delete all interactions associated with the decision
point
```

```
        num_deleted = session.query(Interaction).filter_by(
            decision_point_name=decision_point_name).delete()
        session.commit()
        logger.info(
            f"Deleted {num_deleted} interaction records for
decision point '{decision_point_name}'.")
        return True
    except Exception as e:
        session.rollback()
        logger.error(
            f"Error deleting records for decision point
'{decision_point_name}': {e}")
        return False
    finally:
        session.close()


def log_interaction(decision_point_name, context,
chosen_action, reward):
    session = get_session()
    try:
        new_interaction = Interaction(
            decision_point_name=decision_point_name,
            context=json.dumps(context),  # Serialize context
to JSON string
            chosen_action=chosen_action,
            reward=reward
        )
        session.add(new_interaction)
        session.commit()
        # logger.info(f"Logged interaction for
'{decision_point_name}'") # Too verbose for many logs
    except Exception as e:
        session.rollback()
        logger.error(
```

```
            f"Error logging interaction for
'{decision_point_name}': {e}")
    finally:
        session.close()


def get_record_count_for_decision_point(decision_point_name):
    session = get_session()
    try:
        count = session.query(Interaction).filter_by(
            decision_point_name=decision_point_name).count()
        return count
    finally:
        session.close()


def get_records_for_batch_fit(decision_point_name):
    session = get_session()
    try:
        records =
session.query(Interaction).filter_by(decision_point_name=decis
ion_point_name).all()
        # Use the helper method from the ORM model to get the
context as a dictionary
        return [(r.get_context(), r.chosen_action, r.reward)
for r in records]
    finally:
        session.close()


def get_records_for_analytics(decision_point_name,
limit=None):
    session = get_session()
    try:
        query = session.query(Interaction).filter_by(
```

```
decision_point_name=decision_point_name).order_by(Interaction.
timestamp.asc())
        if limit is not None:
            query = query.limit(limit)
        records = query.all()
        # Convert ORM objects to the expected dictionary
format, parsing JSON context
        processed_records = []
        for record in records:
            try:
                parsed_context = json.loads(record.context)
            except (json.JSONDecodeError, TypeError) as e:
                logger.warning(
                    f"Failed to parse context for record (DP:
{decision_point_name}, ID: {record.id}): {e}. Context string:
{record.context}")
                parsed_context = {}  # Default to empty dict
to prevent errors downstream

            processed_records.append({
                'id': record.id,  # Include ID for logging
context, if needed
                # Convert datetime object to ISO string
                'timestamp': record.timestamp.isoformat(),
                'context': parsed_context,
                'chosen_action': record.chosen_action,
                'reward': record.reward
            })
        return processed_records
    finally:
        session.close()
```

# ДОДАТОК Е

Вихідний код методу обробки контексту

```
import numpy as np


def preprocess_context(raw_context_dict, context_schema):
    """
    Converts a raw context dictionary into a numerical NumPy
array based on a schema.
    Ensures consistent feature order based on sorted schema
keys.
    Handles missing features by treating them as 0.0 or all
zeros for one-hot.
    """
    processed_features = []

    # Ensure consistent order of features based on sorted
schema keys
    for feature_name in sorted(context_schema.keys()):
        feature_def = context_schema[feature_name]
        # Use .get() to handle missing keys
        raw_value = raw_context_dict.get(feature_name)

        if feature_def['type'] == 'categorical':
            one_hot_vector =
np.zeros(len(feature_def['values']))
            try:
                # Ensure value is in known categories
                if raw_value in feature_def['values']:
                    idx =
feature_def['values'].index(raw_value)
                    one_hot_vector[idx] = 1.0
                # If raw_value is None or not in values,
one_hot_vector remains all zeros
```

```
            except ValueError:
                # This should ideally not happen if values
check is done, but for robustness
                pass
            processed_features.extend(one_hot_vector.tolist())
        elif feature_def['type'] == 'numerical':
            # Convert to float; if missing or invalid, default
to 0.0
            processed_features.append(float(raw_value) if
isinstance(
                raw_value, (int, float)) else 0.0)
        else:
            # For robustness, handle unknown types
            # In a Flask app, app.logger is typically
available globally.
            # Here, we can use a standard print or raise an
error.
            print(
                f"Warning: Unknown feature type
'{feature_def['type']}' for feature '{feature_name}'. Skipping
or defaulting to 0.0.")
            processed_features.append(0.0)

    return np.array(processed_features).reshape(1, -1)
```

# ДОДАТОК Ж

Вихідний код маршрутизації API-запитів

```
from flask import Blueprint, request, jsonify, current_app

api_bp = Blueprint('api', __name__)


@api_bp.route('/predict', methods=['POST'])
def predict_route():
    current_app.logger.info(
        f"Predict route received a {request.method} request to
{request.path}")
    if request.is_json:
        current_app.logger.info(
            f"Predict route JSON data: {request.get_json()}")
    else:
        current_app.logger.info("Predict route received non-
JSON data.")

    if not request.is_json:
        return jsonify({"error": "Request must be JSON"}), 400

    data = request.get_json()
    decision_point_name = data.get('decision_point_name')
    raw_context_dict = data.get('context')

    if not decision_point_name or not raw_context_dict:
        current_app.logger.error(
            "Missing 'decision_point_name' or 'context' in
request.")
        return jsonify({"error": "Missing
'decision_point_name' or 'context' in request"}), 400

    # Access the global decision points from the app config
```

```
    all_decision_points =
current_app.config['ALL_DECISION_POINTS']
    # current_app.logger.info(
    #         f"all_decision_points '{all_decision_points}'")
    dp = all_decision_points.get(decision_point_name)
    if not dp:
        current_app.logger.error(
            f"Decision point '{decision_point_name}' not found
for prediction.")
        return jsonify({"error": f"Decision point
'{decision_point_name}' not found or not initialized on this
server."}), 404

    try:
        chosen_action = dp.predict(raw_context_dict)
        current_app.logger.info(
            f"Predict for '{decision_point_name}':
Context={raw_context_dict}, Predicted Action={chosen_action}")
        return jsonify({"chosen_action": chosen_action})
    except Exception as e:
        current_app.logger.error(
            f"Error during prediction for
'{decision_point_name}': {e}")
        return jsonify({"error": f"An internal server error
occurred during prediction: {str(e)}"}), 500

    # Add a final, ultimate fallback return (should ideally
never be reached)
    current_app.logger.critical(
        f"Predict route reached unexpected end for
{decision_point_name}")
    return jsonify({"error": "An unexpected server error
occurred."}), 500


@api_bp.route('/record_reward', methods=['POST'])
```

```
def record_reward_route():
    current_app.logger.info(
        f"Record reward route received a {request.method}
request to {request.path}")
    if request.is_json:
        current_app.logger.info(
            f"Record reward route JSON data:
{request.get_json()}")
    else:
        current_app.logger.info("Record reward route received
non-JSON data.")

    if not request.is_json:
        return jsonify({"error": "Request must be JSON"}), 400

    data = request.get_json()
    decision_point_name = data.get('decision_point_name')
    raw_context_dict = data.get('context')
    chosen_action = data.get('chosen_action')
    reward = data.get('reward')

    if not decision_point_name or not raw_context_dict or
chosen_action is None or reward is None:
        return jsonify({"error": "Missing data
(decision_point_name, context, chosen_action, reward)"}), 400

    # Access the global decision points from the app config
    all_decision_points =
current_app.config['ALL_DECISION_POINTS']
    dp = all_decision_points.get(decision_point_name)
    if not dp:
        return jsonify({"error": f"Decision point
'{decision_point_name}' not found or not initialized on this
server."}), 404

    try:
```

```
        dp.record_reward(raw_context_dict, chosen_action,
reward)
        current_app.logger.info(
            f"Record for '{decision_point_name}':
Context={raw_context_dict}, Action={chosen_action},
Reward={reward}")
        return jsonify({"status": "success"})
    except Exception as e:
        current_app.logger.error(
            f"Error recording reward for
'{decision_point_name}': {e}")
        return jsonify({"error": str(e)}), 500
```

# ДОДАТОК И

Вихідний код маршрутизації запитів веб-інтерфейсу

```
import json
from flask import Blueprint, request, jsonify,
render_template, redirect, url_for, current_app

# Import functions from your database and config
from database import get_record_count_for_decision_point,
get_all_decision_point_configs, get_decision_point_config,
get_records_for_analytics
from database import add_decision_point_config,
update_decision_point_config, delete_decision_point_config,
delete_records_for_decision_point
# Needed for deleting associated files
from models.decision_point import DecisionPoint
from utils.initial_seeder import seed_default_decision_points
from config import LEARNING_POLICIES_BLUEPRINTS

admin_bp = Blueprint('admin', __name__)



@admin_bp.route('/')
def admin_dashboard():
    db_configs = get_all_decision_point_configs()

    decision_points_for_template = []
    # Access from app config
    all_decision_points =
current_app.config['ALL_DECISION_POINTS']

    for dp_config in db_configs:
        dp_name = dp_config['name']
        current_record_count =
get_record_count_for_decision_point(dp_name)
```

```
        status = "active"
        if all_decision_points.get(dp_name) and
all_decision_points[dp_name].mab is None:
            status = "cold_start"

        dp_data = {
            "name": dp_config['name'],
            "arms": dp_config['arms'],
            "context_schema": dp_config['context_schema'],
            "learning_policy_name":
dp_config['learning_policy_name'],
            "cold_start_min_records":
dp_config['cold_start_min_records'],
            "current_record_count": current_record_count,
            "status": status
        }
        decision_points_for_template.append(dp_data)

    return render_template('dashboard.html',
decision_points=decision_points_for_template)


@admin_bp.route('/decision_points/<string:dp_name>/delete',
methods=['POST'])
def delete_decision_point_route(dp_name):
    # Retrieve the functions and dictionary from the app
config
    all_decision_points =
current_app.config['ALL_DECISION_POINTS']
    initialize_decision_points =
current_app.config['INITIALIZE_DECISION_POINTS_FUNC']

    if not dp_name:
        return jsonify({"error": "Decision point name not
provided in URL."}), 400
```

```
    # Before attempting to delete from in-memory, let's ensure
it's in the DB.
    # We first check the DB config to confirm existence, then
attempt to get the in-memory instance.
    # This prevents the error if the in-memory dictionary is
out of sync but the DB still holds the record.
    dp_config_from_db = get_decision_point_config(dp_name)
    if not dp_config_from_db:
        # If it's not even in the DB, it definitely shouldn't
be in active instances.
        # This might be an attempt to delete a non-existent
DP.
        return jsonify({"error": f"Decision point '{dp_name}'
not found in database."}), 404

    dp_instance_in_memory = all_decision_points.get(dp_name)
    # If the instance isn't in memory, but it's in the DB, we
still proceed to delete from DB
    # and then re-initialize to ensure consistency.
    # The error "not found in active instances" is less
critical if we know it's in DB.

    try:
        # If the in-memory instance exists, we delete its
associated files.
        # This step is crucial for MAB models.
        if dp_instance_in_memory:
            dp_instance_in_memory.delete_associated_files()
            # Remove from in-memory dictionary immediately if
it exists
            del all_decision_points[dp_name]

        # Always delete the configuration from the database
        delete_decision_point_config(dp_name)
```

```
        # After any change (add, edit, delete), re-initialize
*all* decision points
        # This ensures the global `all_decision_points`
dictionary is always synchronized with the database.
        initialize_decision_points()

        current_app.logger.info(
            f"Decision point '{dp_name}' and its associated
data deleted and re-initialized.")
        return redirect(url_for('admin.admin_dashboard'))
    except Exception as e:
        current_app.logger.error(
            f"Error deleting decision point '{dp_name}': {e}")
        return jsonify({"error": str(e)}), 500

# NEW ROUTE: Delete all records for a specific decision point


@admin_bp.route('/decision_points/<string:dp_name>/delete_reco
rds', methods=['POST'])
def delete_decision_point_records_route(dp_name):
    """
    Deletes all interaction records associated with a specific
decision point.
    Resets the MAB model for that decision point to cold
start.
    """
    if not dp_name:
        return jsonify({"error": "Decision point name not
provided in URL."}), 400

    dp_config_from_db = get_decision_point_config(dp_name)
    if not dp_config_from_db:
        return jsonify({"error": f"Decision point '{dp_name}'
not found in database."}), 404
```

```
    try:
        # 1. Delete records from the database
        delete_records_for_decision_point(dp_name)

        # 2. Reset the in-memory MAB model for this decision
point to cold start
        #    This is crucial because the model was trained on
the now-deleted data.
        all_decision_points =
current_app.config['ALL_DECISION_POINTS']
        if dp_name in all_decision_points:
            dp_instance = all_decision_points[dp_name]
            # Delete associated files if any (e.g., for
learned models)
            dp_instance.delete_associated_files()
            # Re-initialize the MAB model to its cold start
state
            dp_instance.initialize_mab_model()
            current_app.logger.info(
                f"MAB model for '{dp_name}' reset to cold
start after record deletion.")

        current_app.logger.info(
            f"All records for decision point '{dp_name}'
deleted and MAB model reset.")
        return redirect(url_for('admin.admin_dashboard'))
    except Exception as e:
        current_app.logger.error(
            f"Error deleting records for decision point
'{dp_name}': {e}")
        return jsonify({"error": str(e)}), 500


@admin_bp.route('/seed_defaults', methods=['POST'])
def seed_defaults_route():
```

```
    initialize_decision_points =
current_app.config['INITIALIZE_DECISION_POINTS_FUNC']
    try:
        seed_default_decision_points()
        initialize_decision_points()  # Re-initialize all
decision points to load the new ones
        current_app.logger.info(
            "Default decision points seeded and initialized.")
        return redirect(url_for('admin.admin_dashboard'))  #
Use blueprint name
    except Exception as e:
        current_app.logger.error(f"Error seeding default
decision points: {e}")
        return jsonify({"error": str(e)}), 500


@admin_bp.route('/decision_points/add', methods=['GET',
'POST'])
def add_decision_point_form():
    initialize_decision_points =
current_app.config['INITIALIZE_DECISION_POINTS_FUNC']

    if request.method == 'POST':
        dp_name = request.form.get('name')
        arms_str = request.form.get('arms')
        learning_policy_name =
request.form.get('learning_policy')
        cold_start_min_records_str =
request.form.get('cold_start_min_records')
        context_schema_json_str =
request.form.get('context_schema')
        neighborhood_policy_json_str =
request.form.get('neighborhood_policy')

        errors = []
```

```
        if not dp_name:
            errors.append("Decision Point Name is required.")
        if not arms_str:
            errors.append("Arms are required.")
        if not learning_policy_name:
            errors.append("Learning Policy is required.")

        arms = [arm.strip() for arm in arms_str.split(',') if
arm.strip()]
        if not arms:
            errors.append("At least one arm must be
provided.")

        context_schema = {}
        if context_schema_json_str:
            try:
                context_schema =
json.loads(context_schema_json_str)
                if not isinstance(context_schema, dict):
                    errors.append(
                        "Context Schema must be a valid JSON
object.")
                for feature, props in context_schema.items():
                    if not isinstance(props, dict) or 'type'
not in props:
                        errors.append(
                            f"Context feature '{feature}' is
malformed.")
                    if props.get('type') == 'categorical' and
'values' not in props:
                        errors.append(
                            f"Categorical feature '{feature}'
missing 'values'.")
            except json.JSONDecodeError:
                errors.append("Invalid JSON for Context
Schema.")
```

```
        neighborhood_policy_json = None
        if neighborhood_policy_json_str:
            try:
                neighborhood_policy_json = json.loads(
                    neighborhood_policy_json_str)
                if not isinstance(neighborhood_policy_json,
dict) or 'type' not in neighborhood_policy_json:
                    errors.append(
                        "Neighborhood Policy must be a valid
JSON object with a 'type'.")
                elif neighborhood_policy_json.get('type') ==
'Radius' and 'radius' not in neighborhood_policy_json:
                    errors.append(
                        "Radius neighborhood policy requires
'radius' parameter.")
            except json.JSONDecodeError:
                errors.append("Invalid JSON for Neighborhood
Policy.")

        cold_start_min_records = 100
        try:
            cold_start_min_records =
int(cold_start_min_records_str)
            if cold_start_min_records < 1:
                errors.append(
                    "Cold Start Min Records must be a positive
integer.")
        except ValueError:
            errors.append("Cold Start Min Records must be a
valid number.")

        if get_all_decision_point_configs() and
any(cfg['name'] == dp_name for cfg in
get_all_decision_point_configs()):
            errors.append(
```

```
                f"Decision Point with name '{dp_name}' already
exists. Please choose a different name.")

        if errors:
            return render_template(
                'add_decision_point.html',
                learning_policy_names=list(
                    LEARNING_POLICIES_BLUEPRINTS.keys()),
                error="<br>".join(errors),
                name=dp_name, arms=arms_str,
learning_policy=learning_policy_name,

cold_start_min_records=cold_start_min_records_str,
                context_schema=context_schema_json_str,

neighborhood_policy=neighborhood_policy_json_str
            )

        try:
            success = add_decision_point_config(
                name=dp_name,
                arms=arms,
                context_schema=context_schema,
                learning_policy_name=learning_policy_name,

neighborhood_policy_json=neighborhood_policy_json,
                cold_start_min_records=cold_start_min_records
            )
            if success:
                current_app.logger.info(
                    f"Successfully added new decision point:
{dp_name}")
                initialize_decision_points()
                return
redirect(url_for('admin.admin_dashboard'))
            else:
```

```
                errors.append(
                    f"Failed to add decision point
'{dp_name}'. It might already exist.")
                return render_template(
                    'add_decision_point.html',
                    learning_policy_names=list(
                        LEARNING_POLICIES_BLUEPRINTS.keys()),
                    error="<br>".join(errors),
                    name=dp_name, arms=arms_str,
learning_policy=learning_policy_name,

cold_start_min_records=cold_start_min_records_str,
                    context_schema=context_schema_json_str,

neighborhood_policy=neighborhood_policy_json_str
                )

        except Exception as e:
            current_app.logger.error(f"Error adding decision
point: {e}")
            errors.append(f"An unexpected error occurred:
{str(e)}")
            return render_template(
                'add_decision_point.html',
                learning_policy_names=list(
                    LEARNING_POLICIES_BLUEPRINTS.keys()),
                error="<br>".join(errors),
                name=dp_name, arms=arms_str,
learning_policy=learning_policy_name,

cold_start_min_records=cold_start_min_records_str,
                context_schema=context_schema_json_str,

neighborhood_policy=neighborhood_policy_json_str
            )
```

```
    return render_template(
        'add_decision_point.html',

learning_policy_names=list(LEARNING_POLICIES_BLUEPRINTS.keys()
)
    )


@admin_bp.route('/decision_points/<string:dp_name>/edit',
methods=['GET', 'POST'])
def edit_decision_point_form(dp_name):
    initialize_decision_points =
current_app.config['INITIALIZE_DECISION_POINTS_FUNC']

    if request.method == 'POST':
        original_dp_name = request.form.get('original_name')
        new_dp_name = request.form.get('name')
        arms_str = request.form.get('arms')
        learning_policy_name =
request.form.get('learning_policy')
        cold_start_min_records_str =
request.form.get('cold_start_min_records')
        context_schema_json_str =
request.form.get('context_schema')
        neighborhood_policy_json_str =
request.form.get('neighborhood_policy')

        errors = []

        if not new_dp_name:
            errors.append("Decision Point Name is required.")
        if not arms_str:
            errors.append("Arms are required.")
        if not learning_policy_name:
            errors.append("Learning Policy is required.")
```

```
        arms = [arm.strip() for arm in arms_str.split(',') if
arm.strip()]
        if not arms:
            errors.append("At least one arm must be
provided.")

        context_schema = {}
        if context_schema_json_str:
            try:
                context_schema =
json.loads(context_schema_json_str)
                if not isinstance(context_schema, dict):
                    errors.append(
                        "Context Schema must be a valid JSON
object.")
                else:
                    for feature, props in
context_schema.items():
                        if not isinstance(props, dict) or
'type' not in props:
                            errors.append(
                                f"Context feature '{feature}'
is malformed or missing 'type'.")
                        if props.get('type') == 'categorical'
and 'values' not in props:
                            errors.append(
                                f"Categorical feature
'{feature}' missing 'values'.")
            except json.JSONDecodeError:
                errors.append("Invalid JSON for Context
Schema.")

        neighborhood_policy_json = None
        if neighborhood_policy_json_str:
            try:
                neighborhood_policy_json = json.loads(
```

```
                    neighborhood_policy_json_str)
                if not isinstance(neighborhood_policy_json,
dict) or 'type' not in neighborhood_policy_json:
                    errors.append(
                        "Neighborhood Policy must be a valid
JSON object with a 'type'.")
                elif neighborhood_policy_json.get('type') ==
'Radius' and 'radius' not in neighborhood_policy_json:
                    errors.append(
                        "Radius neighborhood policy requires
'radius' parameter.")
            except json.JSONDecodeError:
                errors.append("Invalid JSON for Neighborhood
Policy.")

        cold_start_min_records = 100
        try:
            cold_start_min_records =
int(cold_start_min_records_str)
            if cold_start_min_records < 1:
                errors.append(
                    "Cold Start Min Records must be a positive
integer.")
        except ValueError:
            errors.append("Cold Start Min Records must be a
valid number.")

        if original_dp_name != new_dp_name:
            existing_dp =
get_decision_point_config(new_dp_name)
            if existing_dp:
                errors.append(
                    f"Decision Point with new name
'{new_dp_name}' already exists. Please choose a different
name.")
```

```
        if errors:
            form_data_for_template = {
                "name": new_dp_name,
                "arms": arms,
                "learning_policy_name": learning_policy_name,
                "cold_start_min_records":
cold_start_min_records,
                "context_schema": context_schema,
                "neighborhood_policy":
neighborhood_policy_json
            }
            return render_template(
                'edit_decision_point.html',
                decision_point=form_data_for_template,
                learning_policy_names=list(
                    LEARNING_POLICIES_BLUEPRINTS.keys()),
                error="<br>".join(errors)
            )

        try:
            success = update_decision_point_config(
                original_name=original_dp_name,
                new_name=new_dp_name,
                arms=arms,
                context_schema=context_schema,
                learning_policy_name=learning_policy_name,

neighborhood_policy_json=neighborhood_policy_json,
                cold_start_min_records=cold_start_min_records
            )
            if success:
                current_app.logger.info(
                    f"Successfully updated decision point:
{original_dp_name} -> {new_dp_name}")
                if original_dp_name != new_dp_name:
                    old_dp_temp = DecisionPoint(
```

```
                        original_dp_name, [], None, None, {})
                    old_dp_temp.delete_associated_files()

                initialize_decision_points()
                return
redirect(url_for('admin.admin_dashboard'))
            else:
                errors.append(
                    f"Failed to update decision point
'{new_dp_name}'. It might conflict with an existing name.")
                form_data_for_template = {
                    "name": new_dp_name,
                    "arms": arms,
                    "learning_policy_name":
learning_policy_name,
                    "cold_start_min_records":
cold_start_min_records,
                    "context_schema": context_schema,
                    "neighborhood_policy":
neighborhood_policy_json
                }
                return render_template(
                    'edit_decision_point.html',
                    decision_point=form_data_for_template,
                    learning_policy_names=list(
                        LEARNING_POLICIES_BLUEPRINTS.keys()),
                    error="<br>".join(errors)
                )

        except Exception as e:
            current_app.logger.error(
                f"Error updating decision point
'{original_dp_name}': {e}")
            errors.append(
                f"An unexpected error occurred during update:
{str(e)}")
```

```
            form_data_for_template = {
                "name": new_dp_name,
                "arms": arms,
                "learning_policy_name": learning_policy_name,
                "cold_start_min_records":
cold_start_min_records,
                "context_schema": context_schema,
                "neighborhood_policy":
neighborhood_policy_json
            }
            return render_template(
                'edit_decision_point.html',
                decision_point=form_data_for_template,
                learning_policy_names=list(
                    LEARNING_POLICIES_BLUEPRINTS.keys()),
                error="<br>".join(errors)
            )

    decision_point_data = get_decision_point_config(dp_name)
    if not decision_point_data:
        return "Decision Point not found", 404

    return render_template(
        'edit_decision_point.html',
        decision_point=decision_point_data,

learning_policy_names=list(LEARNING_POLICIES_BLUEPRINTS.keys()
)
    )


@admin_bp.route('/decision_points/<string:dp_name>/details')
def decision_point_details(dp_name):
    # Get the configuration from the database
    dp_config = get_decision_point_config(dp_name)
```

```
    if not dp_config:
        # If the decision point isn't found in the database,
redirect or show an error
        current_app.logger.warning(
            f"Attempted to view details for non-existent
decision point: {dp_name}")
        # Option 1: Redirect to dashboard with a message
(requires Flask-Flash for messages)
        # flash(f"Decision Point '{dp_name}' not found.",
'error')
        # return redirect(url_for('admin.admin_dashboard'))

        # Option 2: Render a simple error page or message
        return render_template('404.html', message=f"Decision
Point '{dp_name}' not found."), 404

    # Get real-time stats
    current_record_count =
get_record_count_for_decision_point(dp_name)

    # Determine the status (cold_start or active)
    all_decision_points =
current_app.config['ALL_DECISION_POINTS']
    status = "active"
    if all_decision_points.get(dp_name) and
all_decision_points[dp_name].mab is None:
        status = "cold_start"

    # Prepare context schema and neighborhood policy for
display (pretty print JSON)
    context_schema_display = json.dumps(
        dp_config.get('context_schema', {}), indent=2)
    neighborhood_policy_display =
json.dumps(dp_config.get('neighborhood_policy', {}), indent=2)
\
        if dp_config.get('neighborhood_policy') else "None"
```

```
    # Pass all data to the template
    return render_template(
        'decision_point_details.html',
        decision_point=dp_config,
        current_record_count=current_record_count,
        status=status,
        context_schema_display=context_schema_display,

neighborhood_policy_display=neighborhood_policy_display
    )


@admin_bp.route('/decision_points/<string:dp_name>/analytics_d
ata')
def get_decision_point_analytics_data(dp_name):
    """
    Provides data for analytics graphs for a specific decision
point,
    including action distribution and correct action
distribution.
    """

    records = get_records_for_analytics(dp_name, limit=None) #
Fetch all records for the true curve

    if not records:
        return jsonify({"message": "No data available for this
decision point yet."}), 200

    action_counts = {}
    correct_action_counts = {}

    # Records are now guaranteed to be in chronological order
from get_records_for_analytics
```

```
    chronological_records = records # No need for
list(reversed(records))

    cumulative_rewards = []
    total_reward = 0

    for i, record in enumerate(chronological_records):
        action = record['chosen_action']
        reward = record['reward']

        action_counts[action] = action_counts.get(action, 0) +
1

        if reward > 0:
            correct_action_counts[action] =
correct_action_counts.get(action, 0) + 1

        total_reward += reward
        cumulative_rewards.append(total_reward / (i + 1))

    timestamps = [record['timestamp'] for record in
chronological_records]

    return jsonify({
        "decision_point_name": dp_name,
        "action_distribution": action_counts,
        "correct_action_distribution": correct_action_counts,
        "cumulative_average_reward": {
            "timestamps": timestamps,
            "rewards": cumulative_rewards
        },
    })
```

# ДОДАТОК К

Вихідний код методів динамічного створення політик

```
from mabwiser.mab import LearningPolicy, NeighborhoodPolicy
from gemini.config import LEARNING_POLICIES_BLUEPRINTS,
RADIUS_NEIGHBORHOOD_POLICY, EPSILON, UCB1_ALPHA,
SOFTMAX_TEMPERATURE


def get_learning_policy_instance(policy_name: str) ->
LearningPolicy:
    """
    Returns a LearningPolicy instance based on its name.
    """
    policy = LEARNING_POLICIES_BLUEPRINTS.get(policy_name)
    if policy is None and policy_name != "Random Baseline":
        raise ValueError(f"Unknown learning policy name:
{policy_name}")
    return policy


def get_neighborhood_policy_instance(policy_type: str, radius:
float = None) -> NeighborhoodPolicy:
    """
    Returns a NeighborhoodPolicy instance based on its type
and parameters.
    Currently, only supports 'Radius'.
    """
    if policy_type == "Radius":
        # For now, it's a fixed global instance, but this
function would
        # allow dynamic creation if radius varied per decision
point.
        if radius is not None:
            return NeighborhoodPolicy.Radius(radius=radius)
```

```
        return RADIUS_NEIGHBORHOOD_POLICY
    elif policy_type is None:
        return None
    else:
        raise ValueError(f"Unknown neighborhood policy type:
{policy_type}")
```

# ДОДАТОК Л

Вихідний код методу наповнення точок взаємодії

```
import logging
from mabwiser.mab import NeighborhoodPolicy
from database import add_decision_point_config,
get_all_decision_point_configs

logger = logging.getLogger(__name__)

def seed_default_decision_points():
    """
    Seeds the database with a set of default decision point
configurations.
    This function should be called explicitly, e.g., via an
admin panel button.
    """

    # Define some basic defaults. This is where you can put
your
    # desired initial configurations.
    _default_configs_to_add = [
        {
            "name": "DailyOffer",
            "arms": ['coins_offer', 'gems_offer',
'xp_boost_offer'],
            "context_schema": {
                'player_type': {'type': 'categorical',
'values': ['casual', 'hardcore', 'newbie']},
                'region': {'type': 'categorical', 'values':
['America', 'Europe', 'Asia', 'Oceania']},
                'player_level': {'type': 'numerical'},
                'is_paying_customer': {'type': 'categorical',
'values': [True, False]}
            },
```

```
            "learning_policy_name": "LinUCB [Contextual]",
            "neighborhood_policy_json": None,
            "cold_start_min_records": 100
        },
        {
            "name": "AdPersonalization",
            "arms": ['ad_type_a', 'ad_type_b', 'ad_type_c'],
            "context_schema": {
                'time_of_day_hour': {'type': 'numerical'},
                'device_type': {'type': 'categorical',
'values': ['mobile', 'tablet', 'desktop']},
                'app_version': {'type': 'numerical'}
            },
            "learning_policy_name": "EpsilonGreedy (eps=0.1)
[Contextual-NH]",
            "neighborhood_policy_json": {"type": "Radius",
"radius": 0.1},
            "cold_start_min_records": 50
        },
        {
            "name": "PlayerWelcomeBonus",
            "arms": ["bonus_a", "bonus_b"],
            "context_schema": {
                # "signup_method": {"type": "categorical",
"values": ["email", "google", "facebook"]}
            },
            "learning_policy_name": "ThompsonSampling [Non-
Contextual]",
            "neighborhood_policy_json": None,
            "cold_start_min_records": 100
        }
    ]

    added_count = 0
    existing_count = 0
    for dp_cfg in _default_configs_to_add:
```

```
        success = add_decision_point_config(
            name=dp_cfg["name"],
            arms=dp_cfg["arms"],
            context_schema=dp_cfg["context_schema"],
learning_policy_name=dp_cfg["learning_policy_name"],
neighborhood_policy_json=dp_cfg["neighborhood_policy_json"],
cold_start_min_records=dp_cfg["cold_start_min_records"]
        )
        if success:
            added_count += 1
        else:
            existing_count += 1
            logger.info(f"Skipping seeding for
'{dp_cfg['name']}': already exists.")

    if added_count > 0:
        logger.info(f"Successfully seeded {added_count}
default decision points.")
    if existing_count > 0:
        logger.info(f"{existing_count} default decision points
already existed in the database.")

    # After seeding, the caller should re-initialize decision
points in app.py
    return added_count > 0 or existing_count > 0 # Return True
if any attempt was made (either added or existed)
```

# ДОДАТОК М

Вихідний код симулятору запитів

```
import requests
import numpy as np
import matplotlib.pyplot as plt
import random
import time

# --- Configuration ---
SERVER_URL = "http://127.0.0.1:5000"
PREDICT_ENDPOINT = f"{SERVER_URL}/predict"
RECORD_ENDPOINT = f"{SERVER_URL}/record_reward"

NUM_SIMULATED_PLAYERS = 100
NUM_ONLINE_STEPS = 500  # Total interactions for DailyOffer

# --- DECISION POINT CONFIGS (MUST MIRROR SERVER'S CONFIGS FOR
'DailyOffer') ---
# Client now only needs the config for the active decision
point it's simulating
DECISION_POINT_CONFIGS = {
    "DailyOffer": {
        "arms": ['coins_offer', 'gems_offer',
'xp_boost_offer'],
        "context_schema": {
            'player_type': {'type': 'categorical', 'values':
['casual', 'hardcore', 'newbie']},
            'region': {'type': 'categorical', 'values':
['America', 'Europe', 'Asia', 'Oceania']},
            'player_level': {'type': 'numerical'},
            'is_paying_customer': {'type': 'categorical',
'values': [True, False]}
        }
    }
```

```
    # Other decision points are removed from the client's
knowledge for this specific simulation
}

# --- Set the active decision point for this client simulation
---
ACTIVE_CLIENT_DECISION_POINT = "DailyOffer"
ACTIVE_CLIENT_DP_CONFIG =
DECISION_POINT_CONFIGS[ACTIVE_CLIENT_DECISION_POINT]


# --- Player Class for Simulation ---
class SimulatedPlayer:
    def __init__(self, player_id):
        self.player_id = player_id
        # Attributes that define player characteristics for
the DailyOffer decision point
        self.player_type = random.choice(['casual',
'hardcore', 'newbie'])
        self.region = random.choice(['America', 'Europe',
'Asia', 'Oceania'])
        self.player_level = random.randint(1, 100)
        self.is_paying_customer = random.choice([True, False])

    def get_context(self):
        """Generates context appropriate for the 'DailyOffer'
decision point."""
        # This function is now simplified to only generate
context for DailyOffer
        return {
            'player_type': self.player_type,
            'region': self.region,
            'player_level': self.player_level,
            'is_paying_customer': self.is_paying_customer
        }
```

```
# --- Reward Simulation (Client-side game logic for
DailyOffer) ---


def simulate_reward(chosen_action, raw_context_dict):
    """
    Simulates the reward based on the true underlying rules
for 'DailyOffer'.
    This is where the 'true' game logic for rewards lives on
the client side.
    """
    player_type = raw_context_dict['player_type']
    region = raw_context_dict['region']
    player_level = raw_context_dict['player_level']
    is_paying_customer =
raw_context_dict['is_paying_customer']

    reward_prob = 0.0

    # Base reward probabilities based on player type and offer
(dominant factor)
    if player_type == 'casual':
        if chosen_action == 'coins_offer':
            reward_prob = 0.85  # High for casual + coins
        elif chosen_action == 'gems_offer':
            reward_prob = 0.15  # Low for casual + gems
        elif chosen_action == 'xp_boost_offer':
            reward_prob = 0.3  # Moderate for casual + XP
    elif player_type == 'hardcore':
        if chosen_action == 'gems_offer':
            reward_prob = 0.85  # High for hardcore + gems
        elif chosen_action == 'coins_offer':
            reward_prob = 0.15  # Low for hardcore + coins
        elif chosen_action == 'xp_boost_offer':
            reward_prob = 0.4  # Moderate for hardcore + XP
    elif player_type == 'newbie':
```

```
        if chosen_action == 'xp_boost_offer':
            reward_prob = 0.7  # Higher for newbie + XP
        else:  # coins or gems
            reward_prob = 0.2  # Lower for newbie + other
offers

    # Region-specific modifiers
    if region == 'America':
        if chosen_action == 'coins_offer':
            reward_prob += 0.05
        else:
            reward_prob -= 0.05
    elif region == 'Europe':
        if chosen_action == 'gems_offer':
            reward_prob += 0.05
        else:
            reward_prob -= 0.05
    elif region == 'Asia':
        if chosen_action == 'gems_offer':
            reward_prob += 0.1
        else:
            reward_prob -= 0.1
    elif region == 'Oceania':  # New region effect
        if chosen_action == 'xp_boost_offer':
            reward_prob += 0.07
        else:
            reward_prob -= 0.03

    # Player level modifier (higher level slightly prefers
gems, lower prefers coins/xp)
    if chosen_action == 'gems_offer':
        reward_prob += (player_level / 100) * 0.1
    elif chosen_action == 'coins_offer':
        reward_prob -= (player_level / 100) * 0.05

    # Paying customer modifier (prefer gems slightly more)
```

```
    if is_paying_customer and chosen_action == 'gems_offer':
        reward_prob += 0.1

    reward_prob = max(0.0, min(1.0, reward_prob))  # Clamp
between 0 and 1
    return random.choices([1, 0], weights=[reward_prob, 1 -
reward_prob], k=1)[0]


# --- Main Simulation Logic ---
if __name__ == "__main__":
    requests.get(PREDICT_ENDPOINT)

    # Create simulated players
    players = [SimulatedPlayer(i) for i in
range(NUM_SIMULATED_PLAYERS)]

    cumulative_avg_rewards = []  # Single list for the active
decision point
    total_reward = 0

    print(
        f"Client Simulator: Starting simulation for
'{ACTIVE_CLIENT_DECISION_POINT}' with {NUM_SIMULATED_PLAYERS}
players for {NUM_ONLINE_STEPS} steps...")

    # Get arms specific to the chosen decision point for
fallback
    dp_arms = ACTIVE_CLIENT_DP_CONFIG["arms"]

    for step in range(1, NUM_ONLINE_STEPS + 1):
        # Pick a random player for this step
        current_player = random.choice(players)

        # Get context specific to the 'DailyOffer' decision
point
```

```
        raw_context_dict = current_player.get_context()

        # 1. CLIENT: Request action from the server for the
active decision point
        chosen_action = None
        try:
            predict_payload = {
                "decision_point_name":
ACTIVE_CLIENT_DECISION_POINT,
                "context": raw_context_dict
            }
            # print(f"DEBUG: Client hitting URL:
{PREDICT_ENDPOINT} with data: {predict_payload}")
            predict_response = requests.post(
                PREDICT_ENDPOINT, json=predict_payload)
            # Raise HTTPError for bad responses (4xx or 5xx)
            predict_response.raise_for_status()
            chosen_action =
predict_response.json().get('chosen_action')
        except requests.exceptions.ConnectionError:
            print("Client: Connection to server failed. Is
server.py running?")
            time.sleep(2)  # Wait a bit before retrying
            continue
        except requests.exceptions.RequestException as e:
            print(
                f"Client: Error requesting action for
{ACTIVE_CLIENT_DECISION_POINT}: {e}. Falling back to random
action.")
            # Fallback to random arm for THIS DP
            chosen_action = random.choice(dp_arms)

        if chosen_action is None:
            # Fallback if server response is malformed
            chosen_action = random.choice(dp_arms)
```

```
        # 2. CLIENT: Simulate reward based on chosen action
and player characteristics for DailyOffer
        reward = simulate_reward(chosen_action,
raw_context_dict)
        total_reward += reward

        # 3. CLIENT: Report reward back to the server for the
active decision point
        try:
            record_payload = {
                "decision_point_name":
ACTIVE_CLIENT_DECISION_POINT,
                "context": raw_context_dict,
                "chosen_action": chosen_action,
                "reward": reward
            }
            record_response = requests.post(
                RECORD_ENDPOINT, json=record_payload)
            record_response.raise_for_status()
        except requests.exceptions.RequestException as e:
            # Server might be down or misconfigured
            print(
                f"Client: Error recording reward for
{ACTIVE_CLIENT_DECISION_POINT}: {e}")

        cumulative_avg_rewards.append(total_reward / step)

        if step % 100 == 0:
            print(
                f"Client Step {step}/{NUM_ONLINE_STEPS}: Avg
Reward = {cumulative_avg_rewards[-1]:.3f}")

    print(
        f"\nClient Simulation Complete. Final Avg Reward for
{ACTIVE_CLIENT_DECISION_POINT}: {cumulative_avg_rewards[-
1]:.3f}")
```

```
    # --- Plotting ---
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, NUM_ONLINE_STEPS + 1),
cumulative_avg_rewards,
             label=f"{ACTIVE_CLIENT_DECISION_POINT} Cumulative
Average Reward")
    plt.title(
        f'Client Simulated Average Reward Over Time for
{ACTIVE_CLIENT_DECISION_POINT}')
    plt.xlabel('Simulation Steps')
    plt.ylabel('Average Reward')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

# ДОДАТОК Н

Вихідний код порівняльної симуляції алгоритмів

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from mabwiser.mab import MAB, LearningPolicy,
NeighborhoodPolicy
import random

# --- 1. Setup ---

# Bandit configuration
ARMS = ['coins_offer', 'gems_offer']

# --- Parameters for Policies ---
EPSILON = 0.1 # Epsilon for EpsilonGreedy policy
UCB1_ALPHA = 1.0 # Alpha for UCB1 policy
SOFTMAX_TEMPERATURE = 0.5 # Temperature for Softmax policy
RADIUS_NEIGHBORHOOD_RADIUS = 0.1 # Radius for
RadiusNeighborhood policy

# Define the neighborhood policy to be used
RADIUS_NEIGHBORHOOD_POLICY =
NeighborhoodPolicy.Radius(radius=RADIUS_NEIGHBORHOOD_RADIUS)

# Define the different learning policies to compare
# Now includes versions with and without neighborhood policy
for comparison
LEARNING_POLICIES = {
    "Random Baseline": None, # Special entry for baseline (no
actual server for this)

    # Non-contextual versions (contexts ignored for
fit/partial_fit, but used for predict)
```

```
    f"EpsilonGreedy (eps={EPSILON}) [Non-Contextual]":
LearningPolicy.EpsilonGreedy(epsilon=EPSILON),
    "ThompsonSampling [Non-Contextual]":
LearningPolicy.ThompsonSampling(),
    f"UCB1 (alpha={UCB1_ALPHA}) [Non-Contextual]":
LearningPolicy.UCB1(alpha=UCB1_ALPHA),
    f"Softmax (temp={SOFTMAX_TEMPERATURE}) [Non-Contextual]":
LearningPolicy.Softmax(tau=SOFTMAX_TEMPERATURE),

    # Contextual versions (using NeighborhoodPolicy)
    f"EpsilonGreedy (eps={EPSILON}) [Contextual-NH]":
LearningPolicy.EpsilonGreedy(epsilon=EPSILON),
    "ThompsonSampling [Contextual-NH]":
LearningPolicy.ThompsonSampling(),
    f"UCB1 (alpha={UCB1_ALPHA}) [Contextual-NH]":
LearningPolicy.UCB1(alpha=UCB1_ALPHA),
    f"Softmax (temp={SOFTMAX_TEMPERATURE}) [Contextual-NH]":
LearningPolicy.Softmax(tau=SOFTMAX_TEMPERATURE),

    # Inherently Contextual Policy
    "LinUCB [Contextual]": LearningPolicy.LinUCB()
}

# --- 2. Data Generation & Preprocessing (Client-side / Shared
Utilities) ---

ALL_REGIONS = ['America', 'Europe', 'Asia'] # Example regions

def generate_raw_user_context():
    """Simulates a raw user context dictionary. This is what a
client would send."""
    player_type_choice = random.random()
    region_choice = random.choice(ALL_REGIONS)

    if player_type_choice < 0.5:
        player_type = 'casual'
```

```
    else:
        player_type = 'hardcore'

    return {'player_type': player_type, 'region':
region_choice}

def preprocess_context(raw_context_dict):
    """Converts a raw context dictionary into a numerical
NumPy array.
    This logic could be shared between client/server or
primarily live on the server
    if the client sends raw data and the server handles all
preprocessing.
    """
    # 1. Player Type Encoding (Label Encoding: casual=0.0,
hardcore=1.0)
    player_type_encoded = 0.0 if
raw_context_dict['player_type'] == 'casual' else 1.0

    # 2. Region Encoding (One-Hot Encoding)
    region_one_hot = np.zeros(len(ALL_REGIONS))
    try:
        region_index =
ALL_REGIONS.index(raw_context_dict['region'])
        region_one_hot[region_index] = 1.0
    except ValueError:
        print(f"Warning: Unknown region
'{raw_context_dict['region']}'. One-hot vector will be all
zeros.")

    # Combine all features into a single 1D array, then
reshape to 2D for StandardScaler
    context_vector = np.array([player_type_encoded] +
region_one_hot.tolist())
```

```
    return context_vector.reshape(1, -1) # Reshape to
[[feature1, feature2, ...]]

def simulate_reward(chosen_action, raw_context_dict):
    """Simulates the reward based on the true underlying
rules.
    This acts as your "game client" that observes the outcome.
    """
    player_type = raw_context_dict['player_type']
    region = raw_context_dict['region']

    # Base reward probabilities based on player type (dominant
factor)
    reward_prob = 0.0
    if player_type == 'casual':
        if chosen_action == 'coins_offer':
            reward_prob = 0.9 # High for casual + coins
        else: # gems_offer
            reward_prob = 0.2 # Low for casual + gems
    else: # hardcore
        if chosen_action == 'gems_offer':
            reward_prob = 0.9 # High for hardcore + gems
        else: # coins_offer
            reward_prob = 0.2 # Low for hardcore + coins

    # Introduce region-specific modifiers (simulating regional
preferences)
    if region == 'America':
        if chosen_action == 'coins_offer':
            reward_prob = min(1.0, reward_prob + 0.05)
        else:
            reward_prob = max(0.0, reward_prob - 0.05)
    elif region == 'Europe':
        if chosen_action == 'gems_offer':
            reward_prob = min(1.0, reward_prob + 0.05)
        else:
```

```
            reward_prob = max(0.0, reward_prob - 0.05)
    elif region == 'Asia':
        if chosen_action == 'gems_offer':
            reward_prob = min(1.0, reward_prob + 0.1)
        else:
            reward_prob = max(0.0, reward_prob - 0.1)

    return random.choices([1, 0], weights=[reward_prob, 1 -
reward_prob], k=1)[0]


# --- 3. Server Logic (Encapsulated in a Class) ---

class BanditServer:
    def __init__(self, arms, learning_policy,
neighborhood_policy=None):
        self.arms = arms
        self.learning_policy = learning_policy
        self.neighborhood_policy = neighborhood_policy
        self.scaler = StandardScaler()

        # MABwiser needs to know if it's contextual from the
start to handle internal states
        if neighborhood_policy:
            self.mab = MAB(arms,
learning_policy=learning_policy,
neighborhood_policy=neighborhood_policy)
        else:
            self.mab = MAB(arms,
learning_policy=learning_policy)

        self.requires_contexts_for_fit_and_update =
self.mab.is_contextual # Check MABwiser's internal state

    def initialize_with_pre_training_data(self,
num_pre_training_steps):
```

```
        """Simulates initial training on historical data."""
        print(f"--- Server: Initializing with
{num_pre_training_steps} pre-training steps ---")
        initial_raw_contexts = []
        initial_actions = []
        initial_rewards = []

        for _ in range(num_pre_training_steps):
            raw_context_dict = generate_raw_user_context() #
Raw data for training
            chosen_action = random.choice(self.arms) # From
historical logs (random or logged actions)
            reward = simulate_reward(chosen_action,
raw_context_dict) # From historical logs

            initial_raw_contexts.append(raw_context_dict)
            initial_actions.append(chosen_action)
            initial_rewards.append(reward)

        # Preprocess all initial contexts for scaler fitting
        initial_context_vectors = [preprocess_context(rcd) for
rcd in initial_raw_contexts]
        initial_context_vectors_np =
np.vstack(initial_context_vectors)

        # Fit the scaler
        self.scaler.fit(initial_context_vectors_np)
        scaled_initial_contexts =
self.scaler.transform(initial_context_vectors_np)

        # Fit the MAB model
        if self.requires_contexts_for_fit_and_update:
            self.mab.fit(initial_actions, initial_rewards,
scaled_initial_contexts)
        else:
            self.mab.fit(initial_actions, initial_rewards)
```

```
        print("--- Server: Initialization complete ---")

    def predict_action(self, raw_context_dict):
        """API endpoint: Client sends raw context, server
returns predicted action."""
        if self.learning_policy is None: # Random Baseline -
Server does nothing smart
            return random.choice(self.arms)

        context_vector = preprocess_context(raw_context_dict)
        scaled_context = self.scaler.transform(context_vector)

        predicted_action =
self.mab.predict(contexts=scaled_context)
        return predicted_action[0] if
isinstance(predicted_action, list) else predicted_action

    def record_reward(self, raw_context_dict, chosen_action,
reward):
        """API endpoint: Client sends context, action, and
observed reward for update."""
        if self.learning_policy is None: # Random Baseline -
Server does not update
            return

        context_vector = preprocess_context(raw_context_dict)

        # Update scaler with the latest context (partial_fit)
        self.scaler.partial_fit(context_vector)

        # Transform context with potentially updated scaler
for MAB update
        scaled_context = self.scaler.transform(context_vector)

        # Update MAB model (partial_fit)
        if self.requires_contexts_for_fit_and_update:
```

```
            self.mab.partial_fit(np.array([chosen_action]),
np.array([reward]), scaled_context)
        else:
            self.mab.partial_fit(np.array([chosen_action]),
np.array([reward]))


# --- 4. Simulation Runner (Client-Server Interaction) ---

def run_simulation_with_api_design(strategy_full_name,
num_pre_training_steps, num_online_steps,
learning_policy_obj=None):
    """
    Runs a simulation demonstrating client-server interaction
for a given strategy.
    """

    print("\n" + "="*50 + "\n")
    print(f"--- Running Simulation for: {strategy_full_name} -
--")

    # Initialize Server (if it's a bandit strategy)
    server = None
    if learning_policy_obj is not None:
        use_neighborhood_policy = "[Contextual-NH]" in
strategy_full_name
        server = BanditServer(ARMS, learning_policy_obj,
RADIUS_NEIGHBORHOOD_POLICY if use_neighborhood_policy else
None)

server.initialize_with_pre_training_data(num_pre_training_step
s)
    else:
        print("--- Server: Not applicable for Random Baseline
---")
```

```
    # --- Online Simulation Loop (Client interacting with
Server) ---
    cumulative_avg_rewards = []
    total_reward = 0
    total_steps = 0

    for step in range(num_online_steps):
        total_steps += 1

        # CLIENT SIDE: Generate user context
        raw_context_dict = generate_raw_user_context()

        chosen_action = None
        if server:
            # CLIENT SIDE: Request action from server
            chosen_action =
server.predict_action(raw_context_dict)
        else:
            # CLIENT SIDE: Random choice for baseline
            chosen_action = random.choice(ARMS)

        # CLIENT SIDE: Simulate interaction and observe reward
        reward = simulate_reward(chosen_action,
raw_context_dict)
        total_reward += reward

        # CLIENT SIDE: Report reward back to server
        if server:
            server.record_reward(raw_context_dict,
chosen_action, reward)

        cumulative_avg_rewards.append(total_reward /
total_steps)

        if (step + 1) % 100 == 0:
```

```
            print(f"  Client Step {step+1}/{total_steps}:
Current Avg Reward = {cumulative_avg_rewards[-1]:.3f}")

    print(f"\n{strategy_full_name} Simulation Complete. Final
Avg Reward: {cumulative_avg_rewards[-1]:.3f}")

    return cumulative_avg_rewards


# --- 5. Main Execution ---

NUM_PRE_TRAINING_STEPS = 100
NUM_ONLINE_STEPS = 2000

results = {}

# Run simulations for each strategy using the new API-like
design
for strategy_name, policy_obj in LEARNING_POLICIES.items():
    results[strategy_name] = run_simulation_with_api_design(
        strategy_name,
        NUM_PRE_TRAINING_STEPS,
        NUM_ONLINE_STEPS,
        learning_policy_obj=policy_obj
    )

# --- Plotting ---
steps = range(1, NUM_ONLINE_STEPS + 1)

plt.figure(figsize=(12, 7))

# Plot Average Reward per Step
for strategy_name, rewards in results.items():
    plt.plot(steps, rewards, label=strategy_name)
plt.title('Average Reward per Step Over Time (Client-Server
Simulation)')
```

```
plt.xlabel('Simulation Steps')
plt.ylabel('Average Reward')
plt.legend(loc='lower right', fontsize='small')
plt.grid(True)

plt.tight_layout()
plt.show()
```

# ДОДАТОК П

Відомість кваліфікаційної роботи

| Позначення | Найменування | Дод. відомості |
|---|---|---|
| | Текстові документи | |
| **1.** | Пояснювальна записка | 186 с. |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | Інші документи | |
| | | |
| **2.** | Презентаційні матеріали | __ плакатів |
| | | |
| | | |
| | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| . | . | Прізвище та ініціали. | Підп. | Дата | | | |
| Розробив | | Горбенко Б. Г. | | | Розробка системи індивідуалізації показу реклами в мобільних іграх за допомогою контекстуальних бандитів | Шифр групи | Код напр./спец. |
| Перевірив | | Шергін В. Л. | | | | ІТШІ-21-3 | 122 |
| Н.контр. | | Малєєва І.А. | | | | ХНУРЕ кафедра ШІ | |
| Затв. | | Золотухін О.В. | | | | | |