ДОДАТОК А

Вихідний код проєкту

Лістинг А.1 – Програмний код файлу acg_env.py

```
import numpy as np
import gymnasium as gym
from pettingzoo import ParallelEnv
import pygame
from typing import Dict, Any, Tuple, Optional, List
import random


class AdaptiveCoopetitionGridworld(ParallelEnv):
    metadata = {'render_modes': ['human', 'rgb_array'],
'render_fps': 10}

    @property
    def num_agents(self):
        return self._num_agents

    @num_agents.setter
    def num_agents(self, value):
        self._num_agents = value

    def __init__(
        self,
        grid_size: int = 15,
        num_agents: int = 5,
        max_cycles: int = 300,
        render_mode: Optional[str] = None,
        coin_lifetime: int = 25,
        fire_lifetime: int = 20,
        monster_lifetime: int = 30,
        coin_reward: float = 1.0,
        fire_reward: float = 10.0,
```

```
        monster_reward: float = 15.0,
        cooperation_intent_weight: float = 0.5,
    ):
        super().__init__()
        self.grid_size = grid_size
        self.num_agents = num_agents
        self.max_cycles = max_cycles
        self.render_mode = render_mode

        self.coin_lifetime = coin_lifetime
        self.fire_lifetime = fire_lifetime
        self.monster_lifetime = monster_lifetime
        self.coin_reward = coin_reward
        self.fire_reward = fire_reward
        self.monster_reward = monster_reward
        self.cooperation_intent_weight =
cooperation_intent_weight

        self.possible_agents = [f'agent_{i}' for i in
range(num_agents)]
        self.agent_name_mapping = {a: i for i, a in
enumerate(self.possible_agents)}

        self.action_spaces = {
            a: gym.spaces.Discrete(5) for a in
self.possible_agents
        }
        self.observation_spaces = {
            a: gym.spaces.Dict({
                'local': gym.spaces.Box(low=0.0, high=1.0,
shape=(7, 7, 4), dtype=np.float32),
                'global': gym.spaces.Box(low=0.0, high=1.0,
shape=(5,), dtype=np.float32),
            })
            for a in self.possible_agents
        }
```

```
        self.window = None
        self.clock = None
        self.cell_size = 40

        self.reset()

    def reset(self, seed: Optional[int] = None, options:
Optional[dict] = None):
        if seed is not None:
            np.random.seed(seed)
            random.seed(seed)

        self.cycle = 0
        self.agents = self.possible_agents[:]

        self.agent_positions: Dict[str, Tuple[int, int]] = {}
        occupied = set()
        for a in self.agents:
            while True:
                pos = (random.randint(0, self.grid_size - 1),
                       random.randint(0, self.grid_size - 1))
                if pos not in occupied:
                    occupied.add(pos)
                    self.agent_positions[a] = pos
                    break
        self.prev_agent_positions = dict(self.agent_positions)

        self.coins: List[Tuple[int, int, int]] = []
        self.fires: List[List] = []
        self.monsters: List[List] = []

        self._cumulative_rewards = {a: 0.0 for a in
self.agents}
        self.episode_resolved_crises = 0
        self.episode_spawned_crises = 0
```

```
        self.episode_coins_collected = 0

        self._spawn_coins(8)
        self._spawn_crisis()
        self.last_crisis_step = self.cycle

        self.coop_history: Dict[str, List[float]] = {a: [0.0]
* 10 for a in self.agents}

        observations = self._get_observations()
        infos = {a: {'coin_pickup': False,
'crisis_resolved_here': False,
                     'active_crises': len(self.fires) +
len(self.monsters)}
                 for a in self.agents}
        return observations, infos

    def step(self, actions: Dict[str, int]):
        self.cycle += 1
        rewards = {a: 0.0 for a in self.agents}
        terminations = {a: False for a in self.agents}
        truncations = {a: self.cycle >= self.max_cycles for a
in self.agents}
        infos = {a: {'coin_pickup': False,
'crisis_resolved_here': False,
                     'coin_reward': 0.0, 'crisis_reward': 0.0}
                 for a in self.agents}

        self.prev_agent_positions = dict(self.agent_positions)

        new_positions = {}
        for a, act in actions.items():
            x, y = self.agent_positions[a]
            if act == 0:    y = max(0, y - 1)
            elif act == 1:  y = min(self.grid_size - 1, y + 1)
            elif act == 2:  x = max(0, x - 1)
```

```
            elif act == 3:  x = min(self.grid_size - 1, x + 1)
            new_positions[a] = (x, y)
        self.agent_positions = new_positions

        self._process_coins(rewards, infos)
        self._process_crises(rewards, infos)

        if random.random() < 0.15:
            self._spawn_coins(3)
        if random.random() < 0.08 and len(self.fires) +
len(self.monsters) < 4:
            self._spawn_crisis()
            self.last_crisis_step = self.cycle

        self._update_coop_history()

        self._cleanup_resources()
        observations = self._get_observations()

        for a in self.agents:
            infos[a]['active_crises'] = len(self.fires) +
len(self.monsters)

        if all(truncations.values()):
            terminations = {a: True for a in self.agents}

        self._cumulative_rewards = {a:
self._cumulative_rewards[a] + rewards[a]
                                    for a in self.agents}

        return observations, rewards, terminations,
truncations, infos

    def get_state(self) -> np.ndarray:
        state = np.zeros((self.grid_size, self.grid_size, 4),
dtype=np.float32)
```

```
        for pos in self.agent_positions.values():
            state[pos[0], pos[1], 0] = 1.0
        for c in self.coins:
            state[c[0], c[1], 1] = 1.0
        for f in self.fires:
            if not f[3]:
                state[f[0], f[1], 2] = 1.0
        for m in self.monsters:
            if not m[3]:
                state[m[0], m[1], 3] = 1.0
        return state

    def _spawn_coins(self, count: int):
        agent_cells = set(self.agent_positions.values())
        for _ in range(count):
            for _try in range(20):
                x = random.randint(0, self.grid_size - 1)
                y = random.randint(0, self.grid_size - 1)
                if (x, y) not in agent_cells:
                    self.coins.append((x, y,
self.coin_lifetime))
                    break

    def _spawn_crisis(self):
        x = random.randint(0, self.grid_size - 1)
        y = random.randint(0, self.grid_size - 1)
        if random.random() < 0.6:
            self.fires.append([x, y, self.fire_lifetime,
False])
        else:
            self.monsters.append([x, y, self.monster_lifetime,
False])
        self.episode_spawned_crises += 1

    def _process_coins(self, rewards: Dict, infos: Dict):
        for i in range(len(self.coins) - 1, -1, -1):
```

```
            x, y, _life = self.coins[i]
            takers = [a for a in self.agents if
self.agent_positions[a] == (x, y)]
            if takers:
                winner = random.choice(takers)
                rewards[winner] += self.coin_reward
                infos[winner]['coin_pickup'] = True
                infos[winner]['coin_reward'] +=
self.coin_reward
                self.episode_coins_collected += 1
                del self.coins[i]

    def _process_crises(self, rewards: Dict, infos: Dict):
        for fire in self.fires:
            if fire[3]:
                continue
            x, y = fire[0], fire[1]
            on_fire = [a for a in self.agents if
self.agent_positions[a] == (x, y)]
            n = len(on_fire)
            if n >= 2:
                fire[3] = True
                shared = self.fire_reward + 2.0 * (n - 2)
                per_agent = shared / n
                for a in on_fire:
                    rewards[a] += per_agent
                    infos[a]['crisis_resolved_here'] = True
                    infos[a]['crisis_reward'] += per_agent
                self.episode_resolved_crises += 1

        for mon in self.monsters:
            if mon[3]:
                continue
            x, y = mon[0], mon[1]
            on_mon = [a for a in self.agents if
self.agent_positions[a] == (x, y)]
```

```
            n = len(on_mon)
            if n >= 3:
                mon[3] = True
                shared = self.monster_reward + 2.0 * (n - 3)
                per_agent = shared / n
                for a in on_mon:
                    rewards[a] += per_agent
                    infos[a]['crisis_resolved_here'] = True
                    infos[a]['crisis_reward'] += per_agent
                self.episode_resolved_crises += 1

    def _cleanup_resources(self):
        self.coins = [(x, y, life - 1) for x, y, life in
self.coins if life > 1]

        self.fires = [f for f in self.fires if f[2] > 1 and
not f[3]]
        for f in self.fires:
            f[2] -= 1
        self.monsters = [m for m in self.monsters if m[2] > 1
and not m[3]]
        for m in self.monsters:
            m[2] -= 1

    def _compute_cooperation_score(self, agent: str) -> float:
        active_crises = [(f[0], f[1]) for f in self.fires if
not f[3]] + \
                        [(m[0], m[1]) for m in self.monsters
if not m[3]]
        if not active_crises:
            return 0.0

        curr = self.agent_positions[agent]
        if curr in active_crises:
            return 1.0
```

```
        prev = self.prev_agent_positions.get(agent, curr)
        manhattan = lambda a, b: abs(a[0] - b[0]) + abs(a[1] -
b[1])
        prev_min = min(manhattan(prev, c) for c in
active_crises)
        curr_min = min(manhattan(curr, c) for c in
active_crises)

        if curr_min < prev_min:
            return self.cooperation_intent_weight
        return 0.0

    def _update_coop_history(self):
        for a in self.agents:
            score = self._compute_cooperation_score(a)
            self.coop_history[a].pop(0)
            self.coop_history[a].append(score)

    def _get_observations(self) -> Dict[str, Dict[str,
np.ndarray]]:
        obs = {}
        num_fires_active = sum(1 for f in self.fires if not
f[3])
        num_monsters_active = sum(1 for m in self.monsters if
not m[3])
        num_coins = len(self.coins)
        time_since_crisis = self.cycle - self.last_crisis_step

        for a in self.agents:
            local = np.zeros((7, 7, 4), dtype=np.float32)
            ax, ay = self.agent_positions[a]

            for dx in range(-3, 4):
                for dy in range(-3, 4):
                    x, y = ax + dx, ay + dy
```

```
                    if not (0 <= x < self.grid_size and 0 <= y
< self.grid_size):
                        continue
                    lx, ly = dx + 3, dy + 3
                    for a2 in self.agents:
                        if a2 != a and
self.agent_positions[a2] == (x, y):
                            local[lx, ly, 0] = 1.0
                    for c in self.coins:
                        if (c[0], c[1]) == (x, y):
                            local[lx, ly, 1] = 1.0
                    for f in self.fires:
                        if not f[3] and (f[0], f[1]) == (x,
y):
                            local[lx, ly, 2] = 1.0
                    for m in self.monsters:
                        if not m[3] and (m[0], m[1]) == (x,
y):
                            local[lx, ly, 3] = 1.0

            if self.num_agents > 1:
                avg_coop = float(np.mean([
                    np.mean(self.coop_history[other])
                    for other in self.agents if other != a
                ]))
            else:
                avg_coop = 0.0

            global_vec = np.array([
                min(num_fires_active / 4.0, 1.0),
                min(num_monsters_active / 4.0, 1.0),
                min(num_coins / 15.0, 1.0),
                float(np.clip(avg_coop, 0.0, 1.0)),
                min(time_since_crisis / 30.0, 1.0),
            ], dtype=np.float32)
```

```
            obs[a] = {'local': local, 'global': global_vec}
        return obs

    def observation_space(self, agent):
        return self.observation_spaces[agent]

    def action_space(self, agent):
        return self.action_spaces[agent]

    def render(self) -> Optional[np.ndarray]:
        if self.render_mode is None:
            return None

        if self.window is None and self.render_mode ==
'human':
            pygame.init()
            self.window = pygame.display.set_mode(
                (self.grid_size * self.cell_size,
self.grid_size * self.cell_size)
            )
            pygame.display.set_caption('Adaptive Coopetition
Gridworld')
            self.clock = pygame.time.Clock()

        canvas = pygame.Surface((self.grid_size *
self.cell_size,
                                 self.grid_size *
self.cell_size))
        canvas.fill((30, 30, 40))

        for x in range(self.grid_size):
            for y in range(self.grid_size):
                rect = pygame.Rect(x * self.cell_size, y *
self.cell_size,
                                   self.cell_size,
self.cell_size)
```

```
                pygame.draw.rect(canvas, (60, 60, 70), rect,
1)

        for x, y, _ in self.coins:
            cx = x * self.cell_size + self.cell_size // 2
            cy = y * self.cell_size + self.cell_size // 2
            pygame.draw.circle(canvas, (255, 220, 0), (cx,
cy), self.cell_size // 4)

        for f in self.fires:
            if f[3]:
                continue
            cx = f[0] * self.cell_size + self.cell_size // 2
            cy = f[1] * self.cell_size + self.cell_size // 2
            pygame.draw.polygon(canvas, (220, 50, 50), [
                (cx, cy - self.cell_size // 3),
                (cx - self.cell_size // 3, cy + self.cell_size
// 3),
                (cx + self.cell_size // 3, cy + self.cell_size
// 3),
            ])

        for m in self.monsters:
            if m[3]:
                continue
            cx = m[0] * self.cell_size + self.cell_size // 2
            cy = m[1] * self.cell_size + self.cell_size // 2
            pygame.draw.polygon(canvas, (180, 0, 255), [
                (cx - self.cell_size // 3, cy - self.cell_size
// 3),
                (cx + self.cell_size // 3, cy - self.cell_size
// 3),
                (cx, cy + self.cell_size // 3),
            ])

        colors = [(0, 180, 255), (0, 255, 180), (255, 180, 0),
```

```
                  (180, 0, 255), (255, 100, 100)]
        for i, a in enumerate(self.agents):
            x, y = self.agent_positions[a]
            cx = x * self.cell_size + self.cell_size // 2
            cy = y * self.cell_size + self.cell_size // 2
            pygame.draw.circle(canvas, colors[i %
len(colors)],
                               (cx, cy), self.cell_size // 3)
            pygame.draw.circle(canvas, (255, 255, 255),
                               (cx, cy), self.cell_size // 3,
3)

        if self.render_mode == 'human':
            self.window.blit(canvas, (0, 0))
            pygame.display.flip()
            self.clock.tick(self.metadata['render_fps'])
            return None
        else:
            return
np.transpose(np.array(pygame.surfarray.pixels3d(canvas)),
                                axes=(1, 0, 2))

    def close(self):
        if self.window is not None:
            pygame.quit()
            self.window = None

if __name__ == '__main__':
    env = AdaptiveCoopetitionGridworld(render_mode=None)
    obs, infos = env.reset(seed=42)
    print('local shape :', obs['agent_0']['local'].shape)
    print('global shape:', obs['agent_0']['global'].shape)
    print('state shape :', env.get_state().shape)
    for _ in range(50):
        actions = {a: env.action_space(a).sample() for a in
env.agents}
```

```
        obs, rewards, terminations, truncations, infos =
env.step(actions)
        if any(terminations.values()) or
any(truncations.values()):
            break
    env.close()
    print('OK')
```

Лістинг А.2 – Програмний код файлу acg_models.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Dict, Optional, Tuple, Union


class LocalObserver(nn.Module):
    """CNN over the agent's 7×7×4 local window
(decentralized)."""

    def __init__(self, in_channels: int = 4, out_dim: int =
256):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, 32, kernel_size=3,
padding=1)
        self.gn1 = nn.GroupNorm(8, 32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3,
padding=1)
        self.gn2 = nn.GroupNorm(8, 64)
        self.fc = nn.Linear(64 * 7 * 7, out_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: (B, 7, 7, 4)  →  (B, 4, 7, 7)
        x = x.permute(0, 3, 1, 2)
        x = F.relu(self.gn1(self.conv1(x)))
        x = F.relu(self.gn2(self.conv2(x)))
```

```
        x = x.flatten(start_dim=1)
        return F.relu(self.fc(x))


class GlobalObserver(nn.Module):
    """CNN over the full 15×15×4 grid (centralized critic
only)."""

    def __init__(self, grid_size: int = 15, in_channels: int =
4, out_dim: int = 128):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size=3,
padding=1)
        self.gn1 = nn.GroupNorm(4, 16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3,
padding=1)
        self.gn2 = nn.GroupNorm(8, 32)
        self.pool = nn.AdaptiveAvgPool2d((5, 5))
        self.fc = nn.Linear(32 * 5 * 5, out_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: (B, 15, 15, 4)  →  (B, 4, 15, 15)
        x = x.permute(0, 3, 1, 2)
        x = F.relu(self.gn1(self.conv1(x)))
        x = F.relu(self.gn2(self.conv2(x)))
        x = self.pool(x)
        x = x.flatten(start_dim=1)
        return F.relu(self.fc(x))


class MetaEncoder(nn.Module):
    """Encodes the 5-D regime descriptor into a context
embedding."""

    def __init__(self, in_dim: int = 5, embed_dim: int = 64):
        super().__init__()
```

```
        self.fc1 = nn.Linear(in_dim, 128)
        self.ln1 = nn.LayerNorm(128)
        self.fc2 = nn.Linear(128, 128)
        self.ln2 = nn.LayerNorm(128)
        self.fc3 = nn.Linear(128, embed_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.relu(self.ln1(self.fc1(x)))
        x = F.relu(self.ln2(self.fc2(x)))
        return torch.tanh(self.fc3(x))


class ACGAgent(nn.Module):
    """
    Actor + centralized critic, with parameter sharing across
agents.

    Forward methods are split:
      * actor_forward   - uses ONLY local observation history
(decentralized)
      * critic_forward  - uses global state, regime descriptor
and agent id
    """

    def __init__(
        self,
        num_agents: int = 5,
        num_actions: int = 5,
        grid_size: int = 15,
        embed_dim: int = 64,
        hidden_dim: int = 256,
        use_lstm: bool = True,
        use_meta_encoder: bool = True,
        use_centralized_critic: bool = True,
    ):
        super().__init__()
```

```
        self.num_agents = num_agents
        self.num_actions = num_actions
        self.grid_size = grid_size
        self.hidden_dim = hidden_dim
        self.use_lstm = use_lstm
        self.use_meta_encoder = use_meta_encoder
        self.use_centralized_critic = use_centralized_critic

        self.local_observer = LocalObserver(in_channels=4,
out_dim=256)
        if use_lstm:
            self.lstm = nn.LSTM(input_size=256,
hidden_size=hidden_dim,
                                num_layers=1,
batch_first=True)
        else:
            self.lstm = None
            self.feedforward = nn.Linear(256, hidden_dim)
        self.actor_fc1 = nn.Linear(hidden_dim, 128)
        self.actor_fc2 = nn.Linear(128, num_actions)

        if use_centralized_critic:
            self.global_observer =
GlobalObserver(grid_size=grid_size,

in_channels=4, out_dim=128)
            critic_in = 128
        else:
            self.global_observer = None
            critic_in = 256

        if use_meta_encoder:
            self.meta_encoder = MetaEncoder(in_dim=5,
embed_dim=embed_dim)
            critic_in += embed_dim
        else:
```

```
            self.meta_encoder = None

        critic_in += num_agents

        self.critic_fc1 = nn.Linear(critic_in, 256)
        self.critic_fc2 = nn.Linear(256, 128)
        self.critic_fc3 = nn.Linear(128, 1)

    def actor_forward(
        self,
        local_obs: torch.Tensor,
        hidden: Optional[Tuple[torch.Tensor, torch.Tensor]] =
None,
    ) -> Tuple[torch.Tensor, Optional[Tuple[torch.Tensor,
torch.Tensor]]]:
        """
        Two supported input shapes:
          (B, 7, 7, 4)            - single timestep, returns
(B, num_actions)
          (B, T, 7, 7, 4)         - full sequence, returns (B,
T, num_actions)

        For sequence input the LSTM is unrolled across T
(proper BPTT).
        """
        if local_obs.dim() == 4:
            B = local_obs.shape[0]
            feat = self.local_observer(local_obs)
            if self.use_lstm:
                feat = feat.unsqueeze(1)
                out, hidden = self.lstm(feat, hidden)
                out = out.squeeze(1)
            else:
                out = F.relu(self.feedforward(feat))
            x = F.relu(self.actor_fc1(out))
            logits = self.actor_fc2(x)
```

```
            return logits, hidden

        elif local_obs.dim() == 5:
            # Sequence input
            B, T = local_obs.shape[:2]
            flat = local_obs.reshape(B * T,
*local_obs.shape[2:])
            feat = self.local_observer(flat)
            feat = feat.reshape(B, T, 256)
            if self.use_lstm:
                out, hidden = self.lstm(feat, hidden)
            else:
                out = F.relu(self.feedforward(feat))
            x = F.relu(self.actor_fc1(out))
            logits = self.actor_fc2(x)
            return logits, hidden

        else:
            raise ValueError(f"Unexpected local_obs.dim() =
{local_obs.dim()}")


    def critic_forward(
        self,
        global_state: Optional[torch.Tensor],
        global_vec: Optional[torch.Tensor],
        agent_ids: torch.Tensor,
        local_obs: Optional[torch.Tensor] = None,
    ) -> torch.Tensor:
        if self.use_centralized_critic:
            assert global_state is not None
            feats = self.global_observer(global_state)
        else:
            assert local_obs is not None
            feats = self.local_observer(local_obs)
```

```
        parts = [feats]
        if self.use_meta_encoder:
            assert global_vec is not None
            parts.append(self.meta_encoder(global_vec))

        if agent_ids.dim() == 1:
            agent_oh = F.one_hot(agent_ids,
num_classes=self.num_agents).float()
        else:
            agent_oh = agent_ids.float()
        parts.append(agent_oh)

        x = torch.cat(parts, dim=-1)
        x = F.relu(self.critic_fc1(x))
        x = F.relu(self.critic_fc2(x))
        return self.critic_fc3(x)
```

Лістинг А.3 – Програмний код файлу acg_trainer.py

```
import os
import random
from typing import Dict, List, Optional, Tuple

import numpy as np
import torch
import torch.nn.functional as F
import torch.optim as optim
import wandb

from acg_env import AdaptiveCoopetitionGridworld
from acg_models import ACGAgent


class Trajectory:
    __slots__ = (
        'obs_local', 'global_state', 'global_vec',
```

```
        'actions', 'log_probs', 'values',
        'rewards', 'dones', 'ep_terminated',
        'bootstrap_value', 'advantages', 'returns',
    )

    def __init__(self):
        self.obs_local: List[torch.Tensor] = []
        self.global_state: List[torch.Tensor] = []
        self.global_vec: List[torch.Tensor] = []
        self.actions: List[torch.Tensor] = []
        self.log_probs: List[torch.Tensor] = []
        self.values: List[torch.Tensor] = []
        self.rewards: List[torch.Tensor] = []
        self.dones: List[torch.Tensor] = []
        self.ep_terminated: bool = False
        self.bootstrap_value: Optional[torch.Tensor] = None
        self.advantages: Optional[torch.Tensor] = None
        self.returns: Optional[torch.Tensor] = None

    def __len__(self) -> int:
        return len(self.actions)

    def stack(self):
        for name in ('obs_local', 'global_state',
'global_vec',
                     'actions', 'log_probs', 'values',
'rewards', 'dones'):
            data = getattr(self, name)
            if data:
                setattr(self, name, torch.stack(data))


class MAPPOTrainer:
    def __init__(
        self,
        env: AdaptiveCoopetitionGridworld,
```

```
        embed_dim: int = 64,
        hidden_dim: int = 256,
        lr: float = 5e-4,
        gamma: float = 0.99,
        lambda_gae: float = 0.95,
        clip_eps: float = 0.2,
        entropy_coef: float = 0.01,
        value_coef: float = 0.5,
        max_grad_norm: float = 0.5,
        use_lstm: bool = True,
        use_meta_encoder: bool = True,
        use_centralized_critic: bool = True,
        device: str = 'cuda' if torch.cuda.is_available() else
'cpu',
        use_wandb: bool = True,
        wandb_run_name: str = 'MAPPO_MetaEncoder',
        save_dir: str = 'models',
        save_every_steps: int = 500_000,
        log_every_updates: int = 1,
    ):
        self.env = env
        self.num_agents = env.num_agents
        self.grid_size = env.grid_size
        self.device = torch.device(device)

        self.gamma = gamma
        self.lambda_gae = lambda_gae
        self.clip_eps = clip_eps
        self.entropy_coef = entropy_coef
        self.value_coef = value_coef
        self.max_grad_norm = max_grad_norm
        self.save_every_steps = save_every_steps
        self.save_dir = save_dir
        self.log_every_updates = log_every_updates
        self._update_count = 0
```

```
        os.makedirs(save_dir, exist_ok=True)

        self.agent = ACGAgent(
            num_agents=self.num_agents,
            num_actions=5,
            grid_size=self.grid_size,
            embed_dim=embed_dim,
            hidden_dim=hidden_dim,
            use_lstm=use_lstm,
            use_meta_encoder=use_meta_encoder,
            use_centralized_critic=use_centralized_critic,
        ).to(self.device)

        self.optimizer = optim.AdamW(
            self.agent.parameters(), lr=lr, weight_decay=1e-5
        )

        self.trajectories: List[Trajectory] = []

        self.use_wandb = use_wandb
        if use_wandb:
            wandb.init(project='ACG-MARL-Diploma',
name=wandb_run_name,
                       config={
                           'embed_dim': embed_dim,
'hidden_dim': hidden_dim,
                           'lr': lr, 'gamma': gamma,
'lambda_gae': lambda_gae,
                           'clip_eps': clip_eps, 'use_lstm':
use_lstm,
                           'use_meta_encoder':
use_meta_encoder,
                           'use_centralized_critic':
use_centralized_critic,
                       })
```

```
    def _stack_local(self, obs) -> torch.Tensor:
        return torch.from_numpy(
            np.stack([obs[a]['local'] for a in
self.env.agents])
        ).float().to(self.device)

    def _stack_global_vec(self, obs) -> torch.Tensor:
        return torch.from_numpy(
            np.stack([obs[a]['global'] for a in
self.env.agents])
        ).float().to(self.device)

    def _global_state_tensor(self) -> torch.Tensor:
        return
torch.from_numpy(self.env.get_state()).float().to(self.device)

    def collect_rollout(self, num_steps: int = 512) -> int:
        self.trajectories = []
        self.agent.eval()

        obs, _ = self.env.reset()
        hidden = None
        traj = Trajectory()
        steps = 0

        while steps < num_steps:
            local = self._stack_local(obs)
            global_state = self._global_state_tensor()
            global_vec = self._stack_global_vec(obs)
            agent_ids = torch.arange(self.num_agents,
device=self.device)

            with torch.no_grad():
                logits, hidden =
self.agent.actor_forward(local, hidden)
```

```
                dist =
torch.distributions.Categorical(logits=logits)
                actions = dist.sample()
                log_probs = dist.log_prob(actions)

                gs_batch =
global_state.unsqueeze(0).expand(self.num_agents, -1, -1, -1)
                values = self.agent.critic_forward(
                    gs_batch, global_vec, agent_ids,
local_obs=local
                ).squeeze(-1)

            action_dict = {a: int(actions[i].item())
                           for i, a in
enumerate(self.env.agents)}
            next_obs, rewards, terminations, truncations, _ =
self.env.step(action_dict)

            r_t = torch.tensor([rewards[a] for a in
self.env.agents],
                               dtype=torch.float32)
            done_t = torch.tensor(
                [float(terminations[a] or truncations[a]) for
a in self.env.agents],
                dtype=torch.float32
            )

            traj.obs_local.append(local.cpu())
            traj.global_state.append(global_state.cpu())
            traj.global_vec.append(global_vec.cpu())
            traj.actions.append(actions.cpu())
            traj.log_probs.append(log_probs.cpu())
            traj.values.append(values.cpu())
            traj.rewards.append(r_t)
            traj.dones.append(done_t)
```

```
            obs = next_obs
            steps += 1

            ep_done = any(terminations.values()) or
any(truncations.values())
            ep_terminated = ep_done and
any(terminations.values())

            if ep_done or steps == num_steps:
                traj.ep_terminated = ep_terminated

                if ep_terminated:
                    bootstrap = torch.zeros(self.num_agents)
                else:
                    fl = self._stack_local(obs)
                    fgs = self._global_state_tensor()
                    fgv = self._stack_global_vec(obs)
                    fids = torch.arange(self.num_agents,
device=self.device)
                    fgs_b =
fgs.unsqueeze(0).expand(self.num_agents, -1, -1, -1)
                    with torch.no_grad():
                        bootstrap = self.agent.critic_forward(
                            fgs_b, fgv, fids, local_obs=fl
                        ).squeeze(-1).cpu()
                traj.bootstrap_value = bootstrap

                traj.stack()
                self._compute_gae(traj)
                self.trajectories.append(traj)

                if ep_done and steps < num_steps:
                    obs, _ = self.env.reset()
                    hidden = None
                    traj = Trajectory()
```

```
        self.agent.train()
        return steps

    def _compute_gae(self, traj: Trajectory):
        rewards = traj.rewards
        values = traj.values
        T, n = rewards.shape

        advantages = torch.zeros_like(rewards)
        last_gae = torch.zeros(n)

        if traj.ep_terminated:
            next_value = torch.zeros(n)
        else:
            next_value = traj.bootstrap_value

        for t in reversed(range(T)):
            delta = rewards[t] + self.gamma * next_value -
values[t]
            last_gae = delta + self.gamma * self.lambda_gae *
last_gae
            advantages[t] = last_gae
            next_value = values[t]

        traj.advantages = advantages
        traj.returns = advantages + values

    def update(self, num_epochs: int = 4):
        if not self.trajectories:
            return

        all_adv = torch.cat([t.advantages.flatten() for t in
self.trajectories])
        adv_mean = all_adv.mean()
        adv_std = all_adv.std() + 1e-8
```

```
        last_logs = {}

        for epoch in range(num_epochs):
            order = list(range(len(self.trajectories)))
            random.shuffle(order)
            for idx in order:
                traj = self.trajectories[idx]
                T = len(traj)
                n = self.num_agents

                local = traj.obs_local.to(self.device)
                global_state =
traj.global_state.to(self.device)
                global_vec = traj.global_vec.to(self.device)
                actions = traj.actions.to(self.device)
                old_logp = traj.log_probs.to(self.device)
                advantages = traj.advantages.to(self.device)
                returns = traj.returns.to(self.device)

                advantages = (advantages - adv_mean) / adv_std

                local_seq = local.transpose(0, 1).contiguous()
                logits_seq, _ =
self.agent.actor_forward(local_seq, hidden=None)
                logits = logits_seq.transpose(0,
1).contiguous()
                dist =
torch.distributions.Categorical(logits=logits)
                new_logp = dist.log_prob(actions)
                entropy = dist.entropy().mean()

                gs_rep = global_state.unsqueeze(1).expand(-1,
n, -1, -1, -1)
                gs_flat = gs_rep.reshape(T * n,
*gs_rep.shape[2:])
                gv_flat = global_vec.reshape(T * n, -1)
```

```
                local_flat = local.reshape(T * n,
*local.shape[2:])
                ids_flat = torch.arange(n,
device=self.device)\
                    .unsqueeze(0).expand(T, -1).reshape(-1)
                values = self.agent.critic_forward(
                    gs_flat, gv_flat, ids_flat,
local_obs=local_flat
                ).squeeze(-1).reshape(T, n)

                ratio = torch.exp(new_logp - old_logp)
                surr1 = ratio * advantages
                surr2 = torch.clamp(ratio, 1 - self.clip_eps,
1 + self.clip_eps) * advantages
                actor_loss = -torch.min(surr1, surr2).mean()
                value_loss = F.mse_loss(values, returns)
                total_loss = actor_loss + self.value_coef *
value_loss \
                             - self.entropy_coef * entropy

                self.optimizer.zero_grad()
                total_loss.backward()
                grad_norm = torch.nn.utils.clip_grad_norm_(
                    self.agent.parameters(),
self.max_grad_norm
                )
                self.optimizer.step()

                last_logs = {
                    'actor_loss': actor_loss.item(),
                    'value_loss': value_loss.item(),
                    'entropy': entropy.item(),
                    'total_loss': total_loss.item(),
                    'grad_norm': float(grad_norm),
                    'approx_kl': float((old_logp -
new_logp).mean().item()),
```

```
                }
                self._update_count += 1

        if self.use_wandb and last_logs:
            wandb.log(last_logs)

    def train(
        self,
        total_env_steps: int = 5_000_000,
        rollout_steps: int = 512,
        update_epochs: int = 4,
        log_every_env_steps: int = 50_000,
        save_path: Optional[str] = None,
    ):
        env_steps = 0
        next_log = log_every_env_steps
        next_save = self.save_every_steps
        save_path = save_path or os.path.join(self.save_dir,
'acg_final_model.pth')

        ep_returns = []

        while env_steps < total_env_steps:
            collected = self.collect_rollout(rollout_steps)
            self.update(num_epochs=update_epochs)
            env_steps += collected

            for traj in self.trajectories:
                if traj.ep_terminated or len(traj) >=
self.env.max_cycles:

ep_returns.append(traj.rewards.sum(dim=0).mean().item())
            ep_returns = ep_returns[-50:]

            if env_steps >= next_log:
```

```
                avg_ret = np.mean(ep_returns) if ep_returns
else float('nan')
                print(f'[step {env_steps:>9}]
avg_episode_return={avg_ret:.2f}  '

f'trajectories={len(self.trajectories)}')
                if self.use_wandb:
                    wandb.log({
                        'global_step': env_steps,
                        'avg_episode_return': avg_ret,
                    })
                next_log += log_every_env_steps

            if env_steps >= next_save:
                ckpt = os.path.join(self.save_dir,
f'acg_step_{env_steps}.pth')
                torch.save(self.agent.state_dict(), ckpt)
                next_save += self.save_every_steps

        torch.save(self.agent.state_dict(), save_path)
        print(f'Training complete. Final checkpoint:
{save_path}')
        if self.use_wandb:
            wandb.finish()
```

Лістинг А.4 – Програмний код файлу train.py

```
import argparse
import random

import numpy as np
import torch

from acg_env import AdaptiveCoopetitionGridworld
from acg_trainer import MAPPOTrainer
from training_configs import CONFIGS
```

```
def set_seed(seed: int = 42):
    torch.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--config', type=str,
default='full_model',
                        choices=list(CONFIGS.keys()),
                        help='Which config from
training_configs.py to use')
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--total-steps', type=int,
default=None,
                        help='Total environment steps
(overrides config value)')
    parser.add_argument('--rollout-steps', type=int,
default=512)
    parser.add_argument('--update-epochs', type=int,
default=4)
    parser.add_argument('--no-wandb', action='store_true')
    parser.add_argument('--device', type=str, default=None,
                        help='cuda or cpu (auto by default)')
    args = parser.parse_args()

    set_seed(args.seed)
```

```
    cfg = CONFIGS[args.config]
    print(f'Running config: {args.config}')
    print(f'  description       : {cfg['description']}')
    print(f'  use_lstm          : {cfg.get('use_lstm',
True)}')
    print(f'  use_meta_encoder  : {cfg.get('use_meta_encoder',
True)}')
    print(f'  use_centralized_critic:
{cfg.get('use_centralized_critic', True)}')

    env = AdaptiveCoopetitionGridworld(
        grid_size=15,
        num_agents=5,
        max_cycles=300,
        render_mode=None,
    )

    device = args.device or ('cuda' if
torch.cuda.is_available() else 'cpu')

    trainer = MAPPOTrainer(
        env=env,
        embed_dim=cfg.get('embed_dim', 64),
        hidden_dim=cfg.get('hidden_dim', 256),
        lr=cfg.get('lr', 5e-4),
        gamma=cfg.get('gamma', 0.99),
        lambda_gae=cfg.get('lambda_gae', 0.95),
        clip_eps=cfg.get('clip_eps', 0.2),
        entropy_coef=cfg.get('entropy_coef', 0.01),
        value_coef=cfg.get('value_coef', 0.5),
        max_grad_norm=cfg.get('max_grad_norm', 0.5),
        use_lstm=cfg.get('use_lstm', True),
        use_meta_encoder=cfg.get('use_meta_encoder', True),

use_centralized_critic=cfg.get('use_centralized_critic',
True),
```

```
        device=device,
        use_wandb=not args.no_wandb,
        wandb_run_name=f'{args.config}_seed{args.seed}',
        save_dir='models',
    )

    total_steps = args.total_steps or
cfg.get('total_env_steps', 4_000_000)

    trainer.train(
        total_env_steps=total_steps,
        rollout_steps=args.rollout_steps,
        update_epochs=args.update_epochs,
        save_path=cfg['output_path'],
    )

    env.close()


if __name__ == '__main__':
    main()
```

Лістинг А.5 – Програмний код файлу training_configs.py

```
CONFIGS = {
    "full_model": {
        "description": "MAPPO + LSTM + centralized critic +
meta-encoder (full model)",
        "use_lstm": True,
        "use_meta_encoder": True,
        "use_centralized_critic": True,
        "embed_dim": 64,
        "hidden_dim": 256,
        "lr": 5e-4,
        "gamma": 0.99,
        "lambda_gae": 0.95,
```

```
        "clip_eps": 0.2,
        "entropy_coef": 0.03,
        "value_coef": 0.5,
        "max_grad_norm": 0.5,
        "total_env_steps": 5_000_000,
        "output_path": "models/full_model.pth",
    },

    "baseline_no_meta": {
        "description": "MAPPO + LSTM + centralized critic (no
meta-encoder) — main baseline",
        "use_lstm": True,
        "use_meta_encoder": False,
        "use_centralized_critic": True,
        "embed_dim": 64,
        "hidden_dim": 256,
        "lr": 5e-4,
        "gamma": 0.99,
        "lambda_gae": 0.95,
        "clip_eps": 0.2,
        "entropy_coef": 0.03,
        "value_coef": 0.5,
        "max_grad_norm": 0.5,
        "total_env_steps": 5_000_000,
        "output_path": "models/baseline_no_meta.pth",
    },

    "ablate_no_lstm": {
        "description": "Meta-encoder + centralized critic, but
actor has no LSTM",
        "use_lstm": False,
        "use_meta_encoder": True,
        "use_centralized_critic": True,
        "embed_dim": 64,
        "hidden_dim": 256,
        "lr": 5e-4,
```

```
        "gamma": 0.99,
        "lambda_gae": 0.95,
        "clip_eps": 0.2,
        "entropy_coef": 0.03,
        "value_coef": 0.5,
        "max_grad_norm": 0.5,
        "total_env_steps": 5_000_000,
        "output_path": "models/ablate_no_lstm.pth",
    },

    "ablate_decentralized_critic": {
        "description": "Per-agent (decentralized) critic —
IPPO-like baseline",
        "use_lstm": True,
        "use_meta_encoder": False,
        "use_centralized_critic": False,
        "embed_dim": 64,
        "hidden_dim": 256,
        "lr": 5e-4,
        "gamma": 0.99,
        "lambda_gae": 0.95,
        "clip_eps": 0.2,
        "entropy_coef": 0.03,
        "value_coef": 0.5,
        "max_grad_norm": 0.5,
        "total_env_steps": 5_000_000,
        "output_path":
"models/ablate_decentralized_critic.pth",
    },

    "ablate_meta_only": {
        "description": "Decentralized critic + meta-encoder
(no centralized critic)",
        "use_lstm": True,
        "use_meta_encoder": True,
        "use_centralized_critic": False,
```

```
        "embed_dim": 64,
        "hidden_dim": 256,
        "lr": 5e-4,
        "gamma": 0.99,
        "lambda_gae": 0.95,
        "clip_eps": 0.2,
        "entropy_coef": 0.03,
        "value_coef": 0.5,
        "max_grad_norm": 0.5,
        "total_env_steps": 5_000_000,
        "output_path": "models/ablate_meta_only.pth",
    },
}
```

Лістинг А.6 – Програмний код файлу evaluate.py

```
import json
import os
from collections import defaultdict
from typing import Dict, Tuple

import numpy as np
import torch
from tqdm import tqdm

from acg_env import AdaptiveCoopetitionGridworld
from acg_models import ACGAgent
from training_configs import CONFIGS


class Evaluator:
    def __init__(
        self,
        model_path: str,
        config_name: str = 'full_model',
        num_episodes: int = 1000,
```

```
        device: str = None,
        deterministic: bool = True,
    ):
        self.device = torch.device(
            device or ('cuda' if torch.cuda.is_available()
else 'cpu')
        )
        self.num_episodes = num_episodes
        self.deterministic = deterministic
        self.config_name = config_name

        cfg = CONFIGS[config_name]
        self.env =
AdaptiveCoopetitionGridworld(render_mode=None)
        self.agent = ACGAgent(
            num_agents=self.env.num_agents,
            num_actions=5,
            grid_size=self.env.grid_size,
            embed_dim=cfg.get('embed_dim', 64),
            hidden_dim=cfg.get('hidden_dim', 256),
            use_lstm=cfg.get('use_lstm', True),
            use_meta_encoder=cfg.get('use_meta_encoder',
True),

use_centralized_critic=cfg.get('use_centralized_critic',
True),
        ).to(self.device)
        self.agent.load_state_dict(torch.load(model_path,
map_location=self.device))
        self.agent.eval()

    @torch.no_grad()
    def _select_actions(self, obs, hidden):
        local = torch.from_numpy(
            np.stack([obs[a]['local'] for a in
self.env.agents])
```

```
        ).float().to(self.device)
        logits, hidden = self.agent.actor_forward(local,
hidden)
        if self.deterministic:
            actions = torch.argmax(logits, dim=-1)
        else:
            actions =
torch.distributions.Categorical(logits=logits).sample()
        return actions.cpu().numpy(), hidden

    def run_evaluation(self) -> Tuple[Dict[str, float],
Dict[str, list]]:
        per_episode = defaultdict(list)

        for ep in tqdm(range(self.num_episodes),
desc=f'Evaluating {self.config_name}'):
            obs, infos = self.env.reset(seed=ep)
            hidden = None

            ep_total_reward = 0.0
            ep_coin_reward = 0.0
            ep_crisis_reward = 0.0
            crisis_steps = 0
            cooperation_steps = 0
            crisis_resolution_times = []
            crisis_spawn_step: dict = {}

            done = False
            while not done:
                for f in self.env.fires:
                    if not f[3]:
                        key = (f[0], f[1], 'fire')
                        crisis_spawn_step.setdefault(key,
self.env.cycle)
                for m in self.env.monsters:
                    if not m[3]:
```

```
                        key = (m[0], m[1], 'monster')
                        crisis_spawn_step.setdefault(key,
self.env.cycle)

                active_pre = [(f[0], f[1], 'fire') for f in
self.env.fires if not f[3]] + \
                             [(m[0], m[1], 'monster') for m in
self.env.monsters if not m[3]]
                pre_active_set = set(active_pre)

                if pre_active_set:
                    crisis_steps += 1
                    on_crisis = any(
                        (self.env.agent_positions[a][0],
                         self.env.agent_positions[a][1]) in
                        {(p[0], p[1]) for p in pre_active_set}
                        for a in self.env.agents
                    )
                    if on_crisis:
                        cooperation_steps += 1

                actions, hidden = self._select_actions(obs,
hidden)
                action_dict = {a: int(actions[i])
                               for i, a in
enumerate(self.env.agents)}
                obs, rewards, terminations, truncations, infos
= self.env.step(action_dict)

                for a in self.env.agents:
                    ep_total_reward += rewards[a]
                    ep_coin_reward +=
infos[a].get('coin_reward', 0.0)
                    ep_crisis_reward +=
infos[a].get('crisis_reward', 0.0)
```

```
                active_post_set = set(
                    [(f[0], f[1], 'fire') for f in
self.env.fires if not f[3]] +
                    [(m[0], m[1], 'monster') for m in
self.env.monsters if not m[3]]
                )
                vanished = pre_active_set - active_post_set
                for key in vanished:
                    spawn = crisis_spawn_step.pop(key, None)
                    resolved =
any(infos[a].get('crisis_resolved_here', False)
                                        for a in self.env.agents)
                    if spawn is not None and resolved:

crisis_resolution_times.append(self.env.cycle - spawn)

                done = any(terminations.values()) or
any(truncations.values())


per_episode['total_reward'].append(ep_total_reward)
            per_episode['coin_reward'].append(ep_coin_reward)

per_episode['crisis_reward'].append(ep_crisis_reward)
            per_episode['cooperation_rate'].append(
                cooperation_steps / max(crisis_steps, 1)
            )
            per_episode['adaptation_speed'].append(
                float(np.mean(crisis_resolution_times))
                if crisis_resolution_times else 0.0
            )
            per_episode['crisis_resolution_rate'].append(
                self.env.episode_resolved_crises /
                max(self.env.episode_spawned_crises, 1)
            )
            per_episode['coin_collection_rate'].append(
```

```
                self.env.episode_coins_collected
            )

        results = {}
        for key, values in per_episode.items():
            arr = np.array(values, dtype=np.float64)
            results[f'mean_{key}'] = float(arr.mean())
            results[f'std_{key}'] = float(arr.std())
        results['mean_cooperation_rate_pct'] =
results['mean_cooperation_rate'] * 100
        results['mean_crisis_resolution_rate_pct'] =
results['mean_crisis_resolution_rate'] * 100

        print(f'\nResults for '{self.config_name}':')
        for k in ('mean_total_reward', 'mean_coin_reward',
'mean_crisis_reward',
                  'mean_cooperation_rate_pct',
'mean_adaptation_speed',
                  'mean_crisis_resolution_rate_pct',
'mean_coin_collection_rate'):
            print(f'   {k}: {results[k]:.3f}')

        return results, dict(per_episode)


if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--model-path', type=str,
default='models/full_model.pth')
    parser.add_argument('--config', type=str,
default='full_model')
    parser.add_argument('--episodes', type=int, default=1000)
    parser.add_argument('--stochastic', action='store_true',
                        help='Sample actions instead of arg-
max')
```

```
    parser.add_argument('--out', type=str,
default='evaluation_results.json')
    args = parser.parse_args()

    evaluator = Evaluator(
        model_path=args.model_path,
        config_name=args.config,
        num_episodes=args.episodes,
        deterministic=not args.stochastic,
    )
    results, raw = evaluator.run_evaluation()

    os.makedirs(os.path.dirname(args.out) or '.',
exist_ok=True)
    with open(args.out, 'w') as f:
        json.dump({'summary': results, 'per_episode': raw}, f,
indent=2)
    print(f'Saved metrics to {args.out}')
```

Лістинг А.7 – Програмний код файлу run_ablation.py

```
import argparse
import json
import os
import subprocess
import sys

import numpy as np
from scipy import stats

from training_configs import CONFIGS
from evaluate import Evaluator


METRICS = (
    'mean_total_reward',
```

```
    'mean_coin_reward',
    'mean_crisis_reward',
    'mean_cooperation_rate_pct',
    'mean_adaptation_speed',
    'mean_crisis_resolution_rate_pct',
    'mean_coin_collection_rate',
)


def train_all(seed: int, total_steps: int, no_wandb: bool):
    for cfg_name in CONFIGS:
        cmd = [sys.executable, 'train.py',
               '--config', cfg_name,
               '--seed', str(seed),
               '--total-steps', str(total_steps)]
        if no_wandb:
            cmd.append('--no-wandb')
        print(f'\n=== Training {cfg_name} ===')
        print(' '.join(cmd))
        result = subprocess.run(cmd)
        if result.returncode != 0:
            print(f'Training {cfg_name} failed with code
{result.returncode}')
            sys.exit(result.returncode)


def evaluate_all(num_episodes: int) -> dict:
    '''Evaluate every config that has a saved checkpoint.'''
    results = {}
    raw = {}
    for cfg_name, cfg in CONFIGS.items():
        path = cfg['output_path']
        if not os.path.exists(path):
            print(f'Skipping {cfg_name}: no checkpoint at
{path}')
            continue
```

```
        ev = Evaluator(model_path=path, config_name=cfg_name,
                       num_episodes=num_episodes)
        summary, per_ep = ev.run_evaluation()
        results[cfg_name] = summary
        raw[cfg_name] = per_ep
    return results, raw


def print_table(results: dict):
    print('\n' + '=' * 120)
    print('ABLATION RESULTS')
    print('=' * 120)
    header = ['model'] + list(METRICS)
    col_w = [22] + [22] * len(METRICS)

    def fmt_row(row):
        return ' | '.join(f'{str(c):<{col_w[i]}}' for i, c in
enumerate(row))

    print(fmt_row(header))
    print('-' * 120)
    for name, summary in results.items():
        row = [name] + [f'{summary.get(m, float('nan')):.3f}'
for m in METRICS]
        print(fmt_row(row))


def significance_tests(raw: dict, baseline: str =
'baseline_no_meta'):
    if baseline not in raw:
        print(f'\nBaseline '{baseline}' not available —
skipping t-tests')
        return
    print('\n' + '=' * 120)
    print(f'WELCH'S T-TEST vs {baseline}  (raw per-episode
rewards)')
```

```
    print('=' * 120)
    base_rewards = np.array(raw[baseline]['total_reward'])
    base_coop = np.array(raw[baseline]['cooperation_rate'])
    print(f'{'model':<28} {'reward Δ%':>14} {'reward p':>12}
{'coop Δpp':>14} {'coop p':>12}')
    print('-' * 120)
    for name, per_ep in raw.items():
        if name == baseline:
            continue
        m_rewards = np.array(per_ep['total_reward'])
        m_coop = np.array(per_ep['cooperation_rate'])
        t1, p1 = stats.ttest_ind(m_rewards, base_rewards,
equal_var=False)
        t2, p2 = stats.ttest_ind(m_coop, base_coop,
equal_var=False)
        d_reward_pct = ((m_rewards.mean() -
base_rewards.mean()) /
                        max(abs(base_rewards.mean()), 1e-8)) *
100
        d_coop_pp = (m_coop.mean() - base_coop.mean()) * 100
        print(f'{name:<28} {d_reward_pct:>13.1f}% {p1:>12.4g}
'
              f'{d_coop_pp:>13.1f}pp {p2:>12.4g}')


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--train', action='store_true',
                        help='Train every config first
(slow)')
    parser.add_argument('--evaluate', action='store_true',
                        help='Evaluate all trained configs
(default if --train not set)')
    parser.add_argument('--episodes', type=int, default=500)
    parser.add_argument('--seed', type=int, default=42)
```

```
    parser.add_argument('--total-steps', type=int,
default=5_000_000)
    parser.add_argument('--no-wandb', action='store_true')
    parser.add_argument('--out', type=str,
default='ablation_results.json')
    args = parser.parse_args()

    if args.train:
        train_all(args.seed, args.total_steps, args.no_wandb)

    if args.evaluate or not args.train:
        results, raw = evaluate_all(args.episodes)
        if not results:
            print('No checkpoints found. Train first with --
train.')
            return
        print_table(results)
        significance_tests(raw)

        with open(args.out, 'w') as f:
            json.dump({'summary': results,
                       'per_episode': {k: {m: list(v) for m, v
in r.items()}
                                       for k, r in
raw.items()}},
                      f, indent=2)
        print(f'\nSaved results to {args.out}')


if __name__ == '__main__':
    main()
```

Лістинг А.8 – Програмний код файлу evaluate_multi_seed.py

```
import argparse
import json
```

```
import os
from collections import defaultdict
from typing import Dict, List

import numpy as np
from scipy import stats

from training_configs import CONFIGS
from evaluate import Evaluator


METRICS = (
    "total_reward",
    "coin_reward",
    "crisis_reward",
    "cooperation_rate",
    "adaptation_speed",
    "crisis_resolution_rate",
    "coin_collection_rate",
)


def evaluate_one(model_path: str, config_name: str, episodes:
int) -> Dict:
    ev = Evaluator(
        model_path=model_path,
        config_name=config_name,
        num_episodes=episodes,
        deterministic=True,
    )
    summary, per_episode = ev.run_evaluation()
    return {"summary": summary, "per_episode": per_episode}


def collect_all(seed_dirs: List[str], episodes: int) -> Dict:
    results = defaultdict(dict)
```

```
    for seed_dir in seed_dirs:
        if not os.path.isdir(seed_dir):
            print(f"Directory {seed_dir} does not exist —
skipping")
            continue
        for cfg_name, cfg in CONFIGS.items():
            ckpt_name = os.path.basename(cfg["output_path"])
            ckpt_path = os.path.join(seed_dir, ckpt_name)
            if not os.path.exists(ckpt_path):
                print(f"Missing {ckpt_path} — skipping")
                continue
            print(f"\n=== {cfg_name} | {seed_dir} ===")
            results[cfg_name][seed_dir] =
evaluate_one(ckpt_path, cfg_name, episodes)
    return results


def aggregate_per_seed(results: Dict) -> Dict:
    per_seed = defaultdict(lambda: defaultdict(list))
    for cfg_name, by_seed in results.items():
        for seed_dir, payload in by_seed.items():
            for metric in METRICS:
                arr = np.array(payload["per_episode"][metric],
dtype=np.float64)

per_seed[cfg_name][metric].append(float(arr.mean()))
    return per_seed


def aggregate_stats(per_seed: Dict) -> Dict:
    stats_dict = {}
    for cfg_name, by_metric in per_seed.items():
        stats_dict[cfg_name] = {}
        for metric, seed_means in by_metric.items():
            arr = np.array(seed_means, dtype=np.float64)
            stats_dict[cfg_name][metric] = {
```

```
                "mean": float(arr.mean()),
                "std": float(arr.std(ddof=1)) if len(arr) > 1
else 0.0,
                "n_seeds": len(arr),
                "seed_means": list(map(float, arr)),
            }
    return stats_dict


def seed_level_ttests(per_seed: Dict, baseline: str =
"full_model") -> Dict:
    if baseline not in per_seed:
        print(f"Baseline '{baseline}' not in results —
skipping seed-level t-tests")
        return {}
    out = {}
    for cfg_name, by_metric in per_seed.items():
        if cfg_name == baseline:
            continue
        out[cfg_name] = {}
        for metric in METRICS:
            base_arr = np.array(per_seed[baseline][metric],
dtype=np.float64)
            cmp_arr = np.array(by_metric[metric],
dtype=np.float64)
            if len(base_arr) < 2 or len(cmp_arr) < 2:
                out[cfg_name][metric] = {
                    "t": None, "p": None,
                    "note": "Need ≥2 seeds in each group for
t-test",
                }
                continue
            t, p = stats.ttest_ind(cmp_arr, base_arr,
equal_var=False)
            out[cfg_name][metric] = {
                "t": float(t),
```

```
                "p": float(p),
                "delta_mean": float(cmp_arr.mean() -
base_arr.mean()),
            }
    return out


def episode_level_ttests(results: Dict, baseline: str =
"full_model") -> Dict:
    pooled = defaultdict(lambda: defaultdict(list))
    for cfg_name, by_seed in results.items():
        for seed_dir, payload in by_seed.items():
            for metric in METRICS:

pooled[cfg_name][metric].extend(payload["per_episode"][metric]
)

    if baseline not in pooled:
        return {}
    out = {}
    for cfg_name, by_metric in pooled.items():
        if cfg_name == baseline:
            continue
        out[cfg_name] = {}
        for metric in METRICS:
            base_arr = np.array(pooled[baseline][metric],
dtype=np.float64)
            cmp_arr = np.array(by_metric[metric],
dtype=np.float64)
            t, p = stats.ttest_ind(cmp_arr, base_arr,
equal_var=False)
            out[cfg_name][metric] = {
                "t": float(t),
                "p": float(p),
                "delta_mean": float(cmp_arr.mean() -
base_arr.mean()),
```

```
                "n_episodes_per_group": int(len(base_arr)),
            }
    return out


def print_main_table(stats_dict: Dict):
    print("\n" + "=" * 130)
    print("MULTI-SEED RESULTS (mean ± std over seeds)")
    print("=" * 130)
    headers = ["config"] + list(METRICS)
    widths = [28] + [22] * len(METRICS)

    def fmt(row):
        return " | ".join(f"{str(c):<{widths[i]}}" for i, c in
enumerate(row))

    print(fmt(headers))
    print("-" * 130)
    for cfg_name, by_metric in stats_dict.items():
        n = by_metric[METRICS[0]]["n_seeds"]
        row = [f"{cfg_name} (n={n})"]
        for m in METRICS:
            mean = by_metric[m]["mean"]
            std = by_metric[m]["std"]
            row.append(f"{mean:.3f} ± {std:.3f}")
        print(fmt(row))


def print_ttest_table(ttests: Dict, title: str, baseline:
str):
    if not ttests:
        return
    print("\n" + "=" * 130)
    print(f"{title} — {baseline} vs others")
    print("=" * 130)
```

```
    print(f"{'config':<28} {'metric':<26} {'Δ mean':>14}
{'t':>10} {'p':>12}  significance")
    print("-" * 130)
    for cfg_name, by_metric in ttests.items():
        for metric, vals in by_metric.items():
            if vals.get("p") is None:
                continue
            p = vals["p"]
            sig = "***" if p < 0.001 else "**" if p < 0.01
else "*" if p < 0.05 else ""
            print(f"{cfg_name:<28} {metric:<26} "
                  f"{vals['delta_mean']:>14.4f}
{vals['t']:>10.3f} "
                  f"{p:>12.4g}  {sig}")


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--seed-dirs", nargs="+",
required=True,
                        help="Directories containing
checkpoints, one per seed")
    parser.add_argument("--episodes", type=int, default=500)
    parser.add_argument("--baseline", type=str,
default="full_model",
                        help="Config to use as the comparison
baseline in t-tests")
    parser.add_argument("--out", type=str,
default="multi_seed_results.json")
    args = parser.parse_args()

    print(f"Evaluating {len(args.seed_dirs)} seed(s) ×
{len(CONFIGS)} configs "
          f"× {args.episodes} episodes …")
    raw_results = collect_all(args.seed_dirs, args.episodes)
    if not raw_results:
```

```
        print("Nothing to evaluate.")
        return

    per_seed = aggregate_per_seed(raw_results)
    stats_dict = aggregate_stats(per_seed)
    seed_tt = seed_level_ttests(per_seed,
baseline=args.baseline)
    ep_tt = episode_level_ttests(raw_results,
baseline=args.baseline)

    print_main_table(stats_dict)
    print_ttest_table(
        seed_tt,
        title=f"SEED-LEVEL Welch's t-test
(N={len(args.seed_dirs)} per group) "
              "[methodologically correct]",
        baseline=args.baseline,
    )
    print_ttest_table(
        ep_tt,
        title=f"EPISODE-LEVEL Welch's t-test (pooled) [for
reference, "
              "less conservative — episodes are correlated]",
        baseline=args.baseline,
    )

    output = {
        "seed_dirs": args.seed_dirs,
        "episodes_per_run": args.episodes,
        "baseline": args.baseline,
        "aggregate_stats": stats_dict,
        "seed_level_ttests": seed_tt,
        "episode_level_ttests": ep_tt,
        "per_seed_means": {cfg: {m: list(map(float, v))
                                 for m, v in
by_metric.items()}
```

```
                            for cfg, by_metric in
per_seed.items()},
        "raw_per_episode": {
            cfg: {seed: {m: list(map(float,
payload["per_episode"][m]))
                         for m in METRICS}
                  for seed, payload in by_seed.items()}
            for cfg, by_seed in raw_results.items()
        },
    }
    with open(args.out, "w") as f:
        json.dump(output, f, indent=2)
    print(f"\nSaved full results to {args.out}")
    print("\nLegend: *** p<0.001, ** p<0.01, * p<0.05")
    print("\nFor your thesis report, use the SEED-LEVEL t-
tests "
          "(they are statistically defensible).")


if __name__ == "__main__":
    main()
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	119 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Падалкін К.Д.			Модель адаптивної мультиагентної системи для динамічної кооперації та конкуренції в ігровому середовищі, що ґрунтується на принципах MARL	Шифр групи	Код напр./спец.
Перевірив		Шевченко О.Ю.				СШІм-24-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					