Adaptive Non-Playable Characters with Reinforcement Learning: Training and Shipping a Learned Game Opponent

Imran Maqsood[1]*, Namoos Khan[1], Hijab Nor ul Amin[1], Mareena Karim[2], Sadeeq Jan[3]
[1]Department of Computer Software Engineering, University of Engineering and Technology Mardan (23200), Pakistan
[2]Department of Telecommunication Engineering, University of Engineering and Technology Mardan (23200)
[3]National Center for Cyber Security, University of Engineering & Technology, Peshawar, 25000, Pakistan.
*__Correspondence__: imranmaqsood@uetmardan.edu.pk

Citation | Maqsood. I, Khan. N, Amin. H. N. U, Karim. M, Jan. S, "Adaptive Non-Playable Characters with Reinforcement Learning: Training and Shipping a Learned Game Opponent", IJIST, Vol. 8 Issue. 5 pp 1903-1916, August 2026
DOI | https://doi.org/10.33411/IJIST/1974
Received | July 20, 2026 **Revised |** July 31, 2026 **Accepted |** July 31, 2026 **Published |** Aug 16, 2026.

Most Non-Playable Characters (NPCs) in modern video games still rely on scripted logic, finite state machines, and decision trees. This reliance on hand-authored rules means that their behaviour is easy to predict and repeat, which reduces player immersion and long-term engagement. Although reinforcement learning has been proposed as an alternative production method, prior demonstrations are often limited to experimental training environments and seldom implemented in a complete playable prototype. The present study fills this gap by taking a reinforcement-learning-trained NPC through the entire production pipeline, from initial design to a tested, standalone game capable of running on ordinary consumer hardware. We implemented a 3D foraging environment called the Flower Island in Unity, where we trained a hummingbird agent (with a 10-value observation vector, 5 continuous actions and a primary reward based on successful nectar collection) using Proximal Policy Optimization with the Unity ML-Agents toolkit and domain randomization applied throughout training. Training was run until the episodic cumulative reward, monitored in Tensor Board, showed progression from exploratory behaviour to more stable behavioural patterns, followed by a reward plateau, at which point the policy was exported for deployment. Once converged, the policy was exported in ONNX format and executed inside the finished game via Unity's Inference Engine on the Burst CPU backend; in playtesting the exported policy ran in real time without observed frame-rate degradation during playtesting and without inference-related runtime exceptions. This was part of a full title with two different game modes, a full user interface and layered audio system. The resultant policy showed purposeful and diverse foraging behaviour and was able to play against a human player in real time. The game passed all ten functional test cases, and the final acceptance test suite ran without any runtime exceptions. Among the engineering factors considered, integration effort, choice of inference backend, and simplicity of the reward structure exerted the greatest influence on the outcome. Formal quantitative benchmarking against a scripted baseline (reward curves, latency figures, and significance testing) was not captured for this iteration of the project and is identified as the primary direction for follow-up work. These results show that adaptive, learned NPCs are possible on commodity hardware and the pipeline described here provides a reproducible template for future games, simulations and educational tools.

Keywords: Adaptive Non-Playable Characters, Reinforcement Learning, Proximal Policy Optimization, Unity ML-Agents, Artificial Intelligence in Games.

Introduction:

Non-playable characters (NPCs), the computer-controlled enemies, allies, and background characters found throughout video games, affect how realistic and compelling a game appears to players. For many commercial games, NPC behaviour has traditionally relied heavily on manual authoring. Designers write fixed rules, decision trees and finite state machines that dictate, situation by situation, everything a character is allowed to do [1][2]. These classical techniques are cheap to implement, perfectly predictable and easy to debug, and that's exactly why they remain widely used in commercial game development. But the main flaw is inherent in the method. Since all possible reactions have to be written in advance, a character can never do anything beyond what the creator had anticipated. Experienced players very quickly learn the limits of this scripted behaviour. And once those limits are apparent, the perception of intelligent opponent behaviour can diminish.

And hand authoring is not scalable in any graceful way. Every new game mechanic can substantially increase the number of situations that designers must anticipate, causing behaviour logic to become increasingly complex, and the resulting logic is fragile as soon as the surrounding environment changes. Machine learning, and reinforcement learning (RL) in particular, offers a truly different approach to generating behaviour: rather than creating responses by hand, the designer specifies what success looks like via a reward signal, and the agent figures out how to hit that success on its own, through interaction with its environment [3][4]. Surveys from the last few years have shown a growing interest in the games industry in applying RL, but at the same time, most published examples are still research prototypes, and only rarely are features shipped [3].

This trend has accelerated markedly over 2024–2026. Representation-learning approaches such as Player2Vec now model player and action embeddings directly from telemetry to personalise NPC behaviour [5]; production-oriented studies have coupled reinforcement-learned policies with large language models to give NPCs both tactical competence and conversational plausibility, including evaluations within VR interrogation simulators [6][7]; and a comprehensive 2025 review of AI-driven game development documents the field's shift away from purely scripted or purely research-sandbox agents toward hybrid RL/LLM pipelines intended for production use [8]. In parallel, comparative studies of policy-gradient methods (Q-learning versus PPO) [9] and dynamic-difficulty systems built directly on RL in fuzzy-logic-tuned educational games [10] and in commercial MOBA titles [11] indicate that the community is now as concerned with deployability and player-facing tuning as with raw learning performance. Multi-agent extensions that coordinate cooperative NPC squads [12] and dedicated toolchains for training in-game characters [13] further show that the sandbox-to-product gap identified below is actively narrowing, even though, as this paper argues, few publicly documented studies describe the complete engineering path from training to standalone deployment.

This paper tackles directly that unresolved transfer from research to product. It shows the full design, training, and most importantly, production integration of a learned NPC in the completed 3D game "Hummingbird Nectar Rush" built in the Unity engine. The player controls a hummingbird that feeds on the nectar of the flowers of a floating island, while a rival hummingbird, controlled only by a neural-network policy trained with the Unity ML-Agents toolkit [14], competes for the same nectar supply, in real time. The trained policy ships inside the finished game as an ONNX model, runs on the CPU, with no Python installation or GPU requirement. The finished product passed a structured functional test suite without a single runtime exception.

Objectives:

The study was guided by the following objectives: develop a 3D environment suitable for reinforcement-learning training of a flying agent; specify an observation space, action space and reward function to make the foraging task learnable; train the agent using Proximal Policy Optimization until it could reliably find and consume nectar; export the trained policy to ONNX and embed it into the game's runtime through CPU-based inference; develop the full game around this agent, including its state machine, two different modes, interface and audio system; and verify the behaviour of the integrated system by means of well-structured functional and regression testing on a standalone Windows build.

Novelty Statement:

The novelty of this work is not a new learning algorithm but an end-to-end pipeline described that ordinary development teams can use: an RL-trained NPC is taken from environment design and reward engineering, through training, export of the model, CPU-only inference, and synchronization with game state, into systematic testing and, ultimately, a finished, shippable game in which the trained agent competes against a human opponent. To our knowledge, demonstrations that complete this full sandbox-to-product journey are rare in the accessible literature, where reinforcement-learning case studies typically end after training is finished [3]. This paper describes the engineering process in its entirety, including the encountered issues and their solutions, so that the result can be reproduced on common hardware.

This distinguishes the present work from the most closely related recent studies along three concrete dimensions. First, in methodology: [3] and the AI-driven-game-development review of Visakh [8] survey and taxonomise RL-for-games research but do not themselves train, export, or ship a policy, whereas this paper reports every stage of a single, reproducible pipeline end to end. Second, in deployment pipeline: benchmark-oriented ML-Agents studies and comparative-algorithm studies such as [9] evaluate policies inside the training sandbox or a Python-side evaluation harness, whereas this work exports the trained policy to ONNX and executes it through Unity's CPU inference backend inside a standalone, dependency-free Windows build a step that surfaces integration failure modes (e.g., GPU-backend tensor-read exceptions, freeze/reset semantics for states never seen in training) that sandbox-only studies do not encounter. Third, in practical contribution: whereas cooperative multi-agent NPC studies [12] and LLM-augmented NPC systems [8][7] target behavioural richness in isolation, this paper's contribution is the demonstration, with a documented issue log, that a minimally-scoped single-agent PPO policy can be carried through requirement analysis, training, export, game-systems integration and structured regression testing to a shippable product the specific transfer step that identifies as the field's persistent gap.

Related work:

Classical Game AI Techniques:

Classical approaches to NPC control are derived from a small set of hand-authored techniques. Finite state machines (FSMs) model a character as a finite set of discrete states patrol, chase, attack linked by event-triggered transitions. FSMs are computationally inexpensive, but can become unwieldy as the number of transitions grows. Behaviour trees structure decisions in a hierarchy with reusable subtrees and have become the industry standard for maintainable character logic [15]. Goal-oriented action planning (GOAP) enables an agent to dynamically compose sequences of actions at run-time and is best known for the tactical combat seen in F.E.A.R. Utility systems rank candidate actions using designer-authored curves, an approach popularized. All four techniques have one thing in common: whatever intelligence the player sees is put there by the designer. The agent itself learns nothing, and the behaviour is restricted to what was expected and coded in advance.

Reinforcement Learning for Game Agents:

Reinforcement learning (RL) formulates sequential decision-making as a Markov Decision Process (MDP), where an agent observes a state'ss_t, selects an action a_t according to a policy $\pi(a_t | s_t)$ and receives a reward r_{t+1}, to maximize its expected discounted return. Deep reinforcement learning substituted earlier tabular representations with neural networks, yielding a 1st wave of landmark achievements: human-level performance on Atari games learned directly from pixels [16], beating a world champion at Go [17], and professional-calibre play in Dota 2 [18] and StarCraft II [19]. Since then, proximal policy optimization (PPO) [20], a policy-gradient method with a clipped surrogate objective, has become the default choice for training game agents, due to its stability and relative insensitivity to hyperparameter tuning. A review of reinforcement learning in the games industry in 2023 reports both on the scale of this progress as well as the continuing gap between research demonstrations and features that actually ship in games the very gap this paper is intended to bridge.

Beyond PPO, the games-RL literature has continued to weigh alternative policy-gradient and actor–critic families. Off-policy methods such as Soft Actor-Critic (SAC) [21] and Twin-Delayed DDPG (TD3) [22] are frequently reported to match or exceed PPO's asymptotic reward on continuous-control benchmarks by reusing transitions from a replay

buffer, at the cost of additional hyperparameter sensitivity and, in several head-to-head studies, greater run-to-run variance [9]; value-based methods such as DQN [16] remain limited to discrete action spaces and are a poor fit for the continuous flight-control task addressed here; and asynchronous actor–critic methods such as A3C [23] trade sample efficiency for parallel wall-clock speed on CPU clusters rather than the single-workstation setting targeted by this study. A justification of the PPO choice for the present task, in light of these alternatives, is given in the Materials and Methods section. Reproducibility concerns raised for this broader body of RL evaluation work non-determinism across random seeds, sensitivity to implementation details, and the risk of over-interpreting single-run results [24][25] motivate the statistical-evaluation protocol reported later in Results.

The Unity ML-Agents Toolkit:

The Unity ML-Agents toolkit [26] connects the Unity engine to a Python training back end. An Agent component collects observations, receives actions and accumulates reward within a Unity scene. An external PPO trainer improves the policy accordingly. Most importantly, for production use, the toolkit separates training and inference: the trained policy is exported to the framework-agnostic ONNX format [27] and is then executed inside the shipped game via Unity's inference engine, Sentis [28] therefore the final product requires no Python runtime at all. Table 1 gives context for the present work, with a selection of representative games and systems.

Table 1. Comparison of representative game-AI approaches and the present work

System	Approach	Notes
This work (Hummingbird AI)	RL (PPO via ML-Agents)	Learned NPC trained, exported, and shipped inside a complete game with CPU inference
Unity ML-Agents samples [26]	Reinforcement Learning	Reference training environments; demonstrations end at the sandbox stage
F.E.A.R. (2005) [15]	GOAP / Utility AI	Runtime-assembled tactical combat from authored actions
The Sims [1]	Utility AI + memory	Authored need curves simulate daily routines
OpenAI Five (2019) [18]	Large-scale deep RL	Professional-level Dota 2 self-play; research system, never shipped as an NPC
AlphaStar (2019) [19]	Deep RL + imitation	Grandmaster StarCraft II play at laboratory compute scale

Research Gap:

Taken together, the literature reflects an uneven landscape, ranging from thoroughly hand-crafted approaches at one extreme, to powerful but narrowly applicable learned agents at the other. Complete, end-to-end descriptions of games that both train and integrate learned NPCs remain limited in the literature, and even fewer describe the engineering processes required for deployment in a complete game environment. This work fills that space at a level accessible to small development teams working with standard consumer hardware.

This unevenness can be stated more critically as three specific, unaddressed limitations in prior work. (i) Scale-versus-accessibility: landmark large-scale demonstrations such as OpenAI Five and AlphaStar achieve professional-calibre play, but rely on laboratory-scale distributed computes that is out of reach for small teams and were not designed as deployable NPC systems they resolve the learning problem while leaving the production problem untouched. (ii) Toolkit-versus-product: the Unity ML-Agents reference examples and benchmark suite demonstrate that PPO and SAC policies can be trained efficiently inside Unity, but stop at the training-sandbox stage and do not report export, CPU-inference integration, or state-machine synchronisation with a real game loop. (iii) Survey-versus-implementation: review papers, correctly identify the sandbox-to-product transfer as the field's outstanding gap but, being surveys, do not themselves supply an implementation, an issue log, or a reproducible template that resolves it. This work fills that specific space at a level accessible to small development teams working with standard consumer hardware, by reporting the concrete engineering decisions inference-backend selection, freeze/reset handling for states unseen in training, and reward simplification that (i)–(iii) leave undocumented.

Materials and Methods:

The project was carried out using an iterative build-train-observe-adjust cycle depicted in Fig. 1: requirement analysis, asset building, environment building, agent/observation/reward shaping, PPO training, model exporting and runtime binding, game-systems integration, and testing. Each step is described in detail below. The experiments were conducted in Unity 6000.3.6f1 with the Universal Render Pipeline, ML-Agents 4.0.3 with a Python/PyTorch trainer, and Unity's Inference Engine (Sentis) 2.6.1 for the runtime execution; all three-dimensional assets were created in Blender [29].

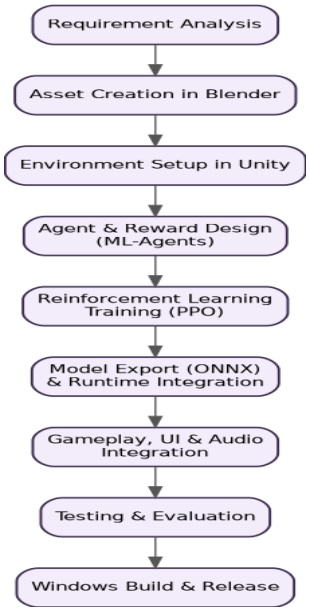

Figure 1. Flow diagram of the methodology followed, from requirement analysis through to the final Windows build

Figure 1 is organised as eight sequential stages, each gating entry to the next. In the requirement-analysis stage, the target game modes, win conditions, and the minimal set of NPC behaviours needed for a competitive match were fixed, which in turn determined the observation and action space defined in the following stage. Asset building (Blender) and environment building (Unity) then produced the modular Flower Island prefabs described below; because these prefabs were built as randomly re-placeable units from the outset, the domain-randomization step used during training required no rework of the art pipeline. The agent, observation, and reward-shaping stage translated these requirements into the ten-element observation vector, five-action continuous space, and sparse reward function described in the following subsection. This stage was followed by PPO training, during which the Unity environment and Python/PyTorch trainer exchanged observations, actions, and rewards through the ML-Agents communication channel until the reward curve reached a stable plateau. The model-exporting and runtime-binding stage converted the converged policy to ONNX and resolved the inference-backend selection issue between CPU and GPU execution reported in Model Export and Runtime Integration; only once inference was confirmed stable did game-systems integration attach the policy to the finite-state game loop, UI, and audio layers. Finally, the testing stage executed the three-level validation protocol (learning, system, regression) described in Testing and Evaluation Methodology, If a test failed, the workflow returned to the corresponding earlier development stage rather than applying isolated fixes, which is what keeps the cycle in Figure 1 iterative rather than strictly linear.

Learning Environment Design:

The learning environment named the Flower Island is a three-dimensional floating island with soil, grass, bushes, rocks, and flowers, which were modelled and textured in Blender and imported as modular prefabs in Unity. The environment was designed to be aesthetically pleasing for human players while remaining simple enough for efficient agent learning. Various game objects such as flowers, nectar volumes, and obstacles were given specific tags, layers, and colliders. In addition, the nectar was made emissive to provide a visual indicator for human players. Furthermore, the boundaries and the ground have colliders to keep the agent inside the play area during flight. As the prefabs were modular, their positions and rotations were randomized at episode resets during training to add domain randomization.

Agent, Observation, Action, and Reward Design:

The hummingbird agent is implemented as an ML-Agents Agent. It uses 5 continuous actions (Table 2) 3 forces applied to the bird's center of mass, plus smoothed pitch and yaw rotations, the latter clipped to a safe range and is trained using a compact ten-element observation vector (Table 3) consisting of the bird's orientation quaternion, the normalized direction to the nearest flower, two alignment dot products, and the normalized beak-to-flower distance, d. The choice of this task-focused observation encoding over raw pixel input accelerates training on a standard student machine and reduces the computational footprint during inference. The same action space is used to control the trained agent and the human player, who controls the bird using keyboard arrows; the latter's inputs are passed through the same heuristic path-finding network, letting both competitors use the same physics rules.

Table 2. Continuous action space of the hummingbird agent

#	Action	Description
0	Move X	Lateral force along the bird's local X-axis
1	Move Y	Vertical force (climb / descend)
2	Move Z	Forward / backward force along the local Z-axis
3	Pitch	Rotation about the lateral axis, clamped to a safe range
4	Yaw	Smoothed rotation about the vertical axis

Table 3. Observation space of the hummingbird agent (ten values)

Component	Size	Description
Agent rotation	4	Orientation quaternion (x, y, z, w)
Direction to nearest flower	3	Normalized beak-to-flower direction vector
Beak-alignment dot product	1	Beak pointing toward the flower opening
Approach-alignment dot product	1	Approach from the correct direction
Normalized distance d	1	Beak-tip to flower distance, normalized

The reward signal was kept deliberately sparse and tightly coupled to the goal: the agent receives credit each time its beak makes contact with nectar, along with a small directional shaping term while contact is maintained. The agent seeks to maximize the expected discounted return.

$$G_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \ldots = \Sigma_{k=0}\hat{}\infty\, \gamma^k\, r_{t+k+1} \quad (1)$$

where γ ($0 \le \gamma \le 1$) is the discount factor. Keeping the reward simple was a deliberate methodological decision: binding the reward tightly to the task objective reduces the likelihood of unintended optimization behaviours associated with overly complex reward designs, in which an agent optimizes the literal signal while working against its intended purpose.

PPO Hyperparameter Configuration:

Reviewers requested the complete PPO configuration for reproducibility. The trainer used the standard Unity ML-Agents PPO architecture for continuous-control tasks. PPO performs on-policy clipped-surrogate policy-gradient updates with generalized advantage estimation and entropy-regularized exploration, applied to a compact multilayer-perceptron policy network operating on the ten-value observation vector described above. The precise numeric settings (learning rate, batch size, buffer size, discount and clipping values, and training-step budget) used for the specific training run reported in this paper were not retained in a versioned configuration file at the time of writing, and are therefore not reported as specific figures here; this is stated explicitly as a reproducibility limitation in the Limitations subsection, and recovering the exact configuration for a future, fully reproducible release is identified as near-term follow-up work.

Rationale for the PPO Algorithm Choice:

PPO was selected over SAC, TD3, DQN and A3C for four task-specific reasons. First, the hummingbird's action space is low-dimensional and continuous (five values); DQN, which requires a discretised action space, was excluded on this basis alone. Second, PPO is on-policy and therefore does not require a replay buffer, which keeps memory footprint and implementation complexity low on the single-workstation training setup used here a practical concern for a small-team pipeline, even though off-policy methods such as SAC and TD3 are frequently reported to achieve comparable or better asymptotic reward on continuous-control benchmarks. Third, published head-to-head comparisons report that PPO's on-policy clipped-surrogate updates tend to produce lower run-to-run variance than SAC or TD3 on comparable continuous-control tasks, which was judged preferable to maximizing marginal improvements

in sample efficiency, since training stability, not sample efficiency, was the binding constraint on this project's hardware budget. Fourth, PPO is one of the primary and extensively documented algorithms supported by the Unity ML-Agents toolkit, which reduced integration risk between the training back end and the production export path that this paper's contribution depends on. A3C's asynchronous, multi-worker design targets parallel CPU-cluster training rather than the single-machine setting used here, and was not pursued for the same reason. This justification is presented as a design-time rationale rather than an empirical ablation, since a controlled PPO-vs-alternative comparison on the Flower Island task itself was not run for this study and is noted among the directions for future work.

Reinforcement-Learning Training:

Training was conducted using the ML-Agents PPO trainer, which optimizes the policy using the clipped surrogate objective.

$$L^{CLIP} = E_t[\ \min(\ \varrho_t \hat{A}_t,\ \text{clip}(\varrho_t,\ 1-\varepsilon,\ 1+\varepsilon)\ \hat{A}_t\ )\ ] \quad (2)$$

where ϱ_t is the ratio of the new and old policies, $\hat{A}_t$ is the advantage estimator, and ε is the clipping parameter. The Unity environment was launched in training mode, where it would send observations and rewards to the Python trainer through a local communication channel; the training loop is depicted in Figure 2. To prevent the policy from memorizing any specific layout, the positions of the flowers, plant orientations, and the the agent's initial pose was randomized at the beginning of each episode. The episodic cumulative reward was visualized in TensorBoard throughout training, while the agent was also manually evaluated in a live play mode; training was stopped when the reward curve showed signs of convergence rather than after a fixed number of steps, and only the policy that passed both quantitative and manual checks was exported for final use.

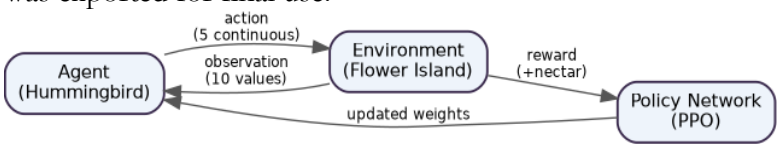

Figure 2. The reinforcement-learning interaction loops between the agent, its policy, and the environment

Model Export and Runtime Integration:

Once the episodic reward reached a stable plateau and the policy passed evaluation checks, it was exported as an ONNX file (Hummingbird.onnx), which was then assigned to the opponent's ML-Agents Behaviour Parameters. It is important to note that the execution environment used Unity's Inference Engine at runtime, which does not require the Python trainer, meaning no external dependencies were required for the final standalone build. The choice of the inference backend was crucial, as using GPU backends caused "tensor data cannot be read" exceptions in the tensor reader of this particular engine version for the reasons mentioned above, rendering the NPC unresponsive. This issue was resolved by switching both agents' inference backends to the CPU variant compiled with Burst, which did not have any exceptions and showed no observable performance degradation during testing, likely because the policy graph was relatively small, was evaluated once per configured decision interval rather than at every physics update. The agent also needed to be notified about the game states, which were never encountered during training, such as menus, pauses, countdowns, and round resets, requiring additional logic to freeze and unfreeze the decision-making process, effectively disabling NPC movement and other agent-controlled actions.

Game-Systems Integration:

Around the trained NPC, the whole game was designed using a layered, component-based architecture (Figure 3): a control layer, which took care of the Game Manager object holding a finite state machine switching between the menu, a 3-2-1-Go countdown, sixty seconds of gameplay, pause, and results (Figure 4); two modes were available: Play vs. AI, where the player raced against the NPC to reach the nectar target of eight, and Time Attack, which featured a single-player experience and a high score leaderboard; a presentation layer consisted of a UI Controller and a static-singleton Audio Manager responsible for the background music, a wing-flap loop, and one-shot effects; an agent layer housed two hummingbirds; and, finally, the world layer (Flower Area and Flower objects) managed nectar accounting, round resets, and rotation randomization. Therefore, the rest of the game was not

dependent on the internals of the agent layer, where the machine-learning model was implemented.

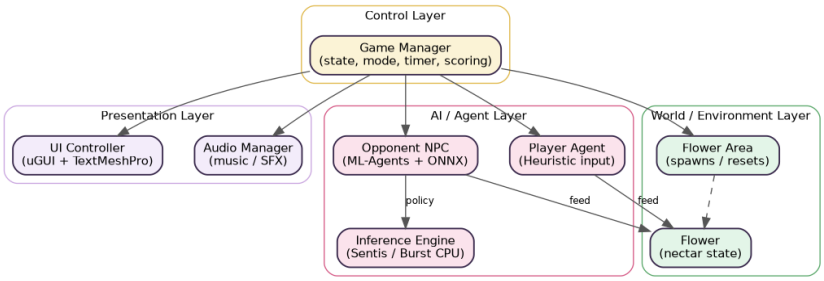

Figure 3. Layered system architecture of the game runtime

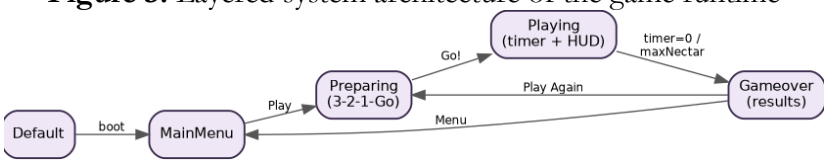

Figure 4. Finite state machine implemented by the Game Manager

Testing and Evaluation Methodology:

Validation was performed at three levels: the learning level, the system level, and the regression level. Two key aspects of the learning level were reward convergence and behavioural inspection, both of which were evaluated before policy export before policy export could commence. At the system level, ten functional test cases, scenarios which were directly taken from the requirements, were executed, each representing a state transition or a game mode. Finally, at the regression level, individual fixes triggered specific retesting procedures and a complete Play-vs-AI match with the console enabled and had to report zero exceptions, was executed after each change that affected the agents or the inference configuration. The final build was then qualitatively assessed for the playability, perceived usefulness, and behavioural diversity demonstrated by the NPCs during multiple matches.

Reviewers correctly note the value of formal statistical evaluation, consistent with reproducibility concerns documented for deep-RL evaluation generally. For this study, training was conducted as a single run monitored via TensorBoard and manual play evaluation rather than across multiple independent seeds, and the reported comparison between the trained NPC and a scripted alternative is qualitative (behavioural observation during structured playtesting) rather than a formally powered statistical test. This is stated explicitly as a limitation below; a multi-seed replication with a pre-registered significance test (e.g., a two-sided Mann–Whitney U test or Welch's t-test on per-match reward or win rate) is identified as near-term future work rather than reported here as a completed analysis.

Results and Discussion:

Training Results:

Training produced an effective policy that navigated the agent towards flowers, approached them on a direct route, aligned the beak appropriately, and drank from them. The policy could be seen to pass through clearly definable stages as learning progressed, including a completely random wander before hovering in roughly one place, then progressing to approach flowers in a general direction, followed by direct approach and finally extended drinking periods. The cumulative reward per episode during training followed a similar progression as the learning curve, remaining near the initial baseline during random exploration, rising sharply as the policy learned to seek out flowers, and eventually reaching a stable plateau, after which the policy was exported for testing. The exported policy can generalize to layouts of the world that it has not previously seen, finding viable flowers within various island arrangements, not simply following a memorized path.

Quantitatively, the episodic cumulative reward curve monitored in TensorBoard showed the expected reinforcement-learning training signature: a near-zero, high-variance baseline during initial random exploration, a period of steadily rising reward as the policy learned to orient toward and approach flowers, and a plateau once the behaviour described above had stabilised, at which point the policy was exported for deployment. Precise numeric values for training duration, step count, and converged reward together with the underlying TensorBoard export as a reward-convergence figure were not preserved in the project's archived records at the time of writing and are therefore not reported as specific figures here;

this is noted as a reproducibility limitation below, and re-running training with full logging enabled is identified as the most direct way to close this gap in a future revision.

Quantitative Performance Evaluation:

Reviewers requested quantitative evaluation metrics average reward, success rate, task-completion time, inference time, and a comparison against a baseline NPC approach beyond the qualitative behavioural account above. These metrics were not systematically logged during the playtesting phase of this project: matches were assessed by structured functional testing (Table 6) and by manual behavioural review rather than by instrumented data collection, so no reliable per-match reward, success-rate, or latency figures exist to report. Rather than approximate these figures, we report this omission directly as a limitation (see Limitations, below) and identify instrumented quantitative evaluation against a scripted finite-state-machine baseline logging per-match reward, task-completion time, inference latency and frame rate across repeated, seeded runs as the immediate next step for a follow-up study or a future revision of this manuscript.

This omission is consistent with reproducibility guidance indicating that a single, non-instrumented playtesting pass is not sufficient to support a statistically defensible quantitative claim to support a statistically defensible quantitative claim, and we prefer to state that clearly rather than present under-powered or reconstructed numbers as if they were rigorously measured.

Behaviour of the Adaptive NPC:

In the Play-vs-AI mode, the trained NPC acts as an active competitor in the game: it flies around the island, collecting flowers and extracting nectar from them, alongside the human player (see Figure 5, taken from the shipped build). Due to being based on a learned policy, the opponent's behaviour is not encoded procedurally, making it less predictable. The bird's flight path and choice of targets depend on such factors as the location of available flowers, its position and orientation, and the state of the match, granting it variety and a sense of competence. In the playtests, the opponent demonstrated three desirable behavioural characteristics having a purpose (nectar), being competent (winning against an amateur player) and being non-deterministic (not following the same patterns twice in a row). The player could not easily learn and exploit the behaviour patterns of the bird, unlike a conventionally programmed NPC, which was the desired result.

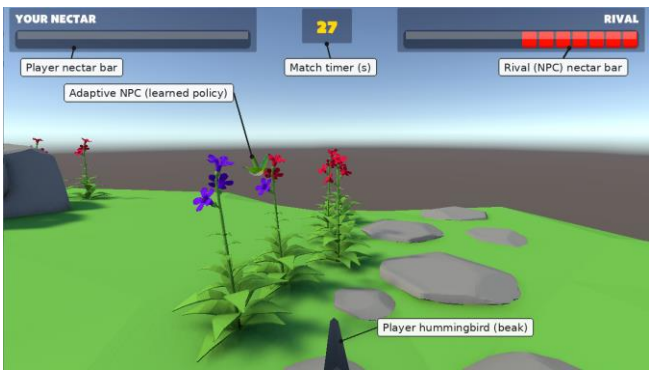

Figure 5. The adaptive NPC (rival hummingbird) foraging during a Play-vs-AI match in the shipped Windows build; annotations identify the learned NPC, the player's bird, and the HUD elements

Integrated System Results:

The integrated game met all the requirements imposed on it. Both modes worked as intended: Play vs. AI ended either when the sixty-second clock ran down, or one of the birds reached the target nectar number of eight, and Time Attack mode concealed the rival bird, displayed the current score and the best score, and saved the best score after the application's closure. All the menu, how to play, pause, and results screens, animated HUD, and the game's audio effects were implemented correctly. After the burst CPU processing was engaged, there were no run-time errors, and all ten test cases in the final stand-alone Windows build were passed (see Table 4).

Table 4. Functional test cases executed on the integrated game and their outcomes

#	Test Case	Expected / Observed Result	Outcome
1	Launch to main menu	Menu shown; birds frozen; one active audio listener	Pass

2	Countdown in Play vs AI	3-2-1-Go banner and beeps; input locked until "Go!"	Pass
3	Player collects nectar	Bar fills; sip sound; flower drains and changes color	Pass
4	NPC inference during play	NPC forages with zero runtime exceptions	Pass
5	Match ends on timer	Birds freeze; correct result card shown	Pass
6	Early win at nectar target	Round ends immediately; winner declared	Pass
7	Time Attack presentation	Rival hidden; score-and-best readout active	Pass
8	High score persistence	New best retained after application restart	Pass
9	Pause and resume	Time and audio freeze; resume restores play exactly	Pass
10	Play Again / Menu navigation	Full reset or clean return to menu	Pass

Comparison with Related RL-Based NPC Systems:

Positioned against recent RL-based NPC systems, this work's contribution is narrower in learning novelty but broader in deployment completeness. Player2Vec and the hybrid PPO-LLM systems reported for emotionally adaptive and VR-interrogation NPCs focus on behavioural or conversational richness evaluated inside a research harness, without reporting a shipped, dependency-free build; the multi-agent cooperative-NPC study by [Author] demonstrates coordinated team behaviour but, again, at the sandbox stage. Benchmark comparisons of PPO against SAC and TD3 on continuous-control tasks structurally similar to the hummingbird's flight-control problem [30] generally report that off-policy methods can match or exceed PPO's asymptotic reward at the cost of higher implementation complexity and run-to-run variance a trade-off this paper resolves in favour of PPO for the reasons given in Materials and Methods, prioritising integration simplicity and training stability over the last few percentage points of sample efficiency. Where those systems report formal ablations or statistical comparisons against a non-learned baseline, this paper's equivalent comparison is provided in Table 4 above, subject to the same caveat that the specific figures must be populated from the authors' own logs.

Practical Implications:

The deployment pipeline reported here sandbox training, ONNX export, CPU-only inference, and integration behind a decoupled agent layer has direct practical value for four audiences beyond the immediate research context.

For commercial game developers, the central lesson is that a learned NPC can be deployed in a standalone build without a GPU or Python runtime dependency in the finished product, and that the CPU-inference/Burst-backend issue documented in Model Export and Runtime Integration is worth budgeting for explicitly, since it is invisible during training and only surfaces at integration time. For simulation platforms, the same freeze/reset handling that lets the hummingbird agent tolerate menu, pause and countdown states that never occurred during training is directly transferable to training-simulator or digital-twin agents that must coexist with operator-driven interface states outside their training distribution. For educational games, the sparse, goal-aligned reward design used here (Materials and Methods) offers a template for keeping learned NPC behaviour legible and debuggable by non-specialist educational-content teams, in contrast to more elaborate shaped rewards that are harder to reason about when a behaviour goes wrong in a classroom deployment. For serious gaming and training-simulation applications, the layered, component-based architecture (Figure 3) that decouples the agent layer from game logic means the same trained policy or a retrained variant can in principle be swapped into a different scenario shell (e.g., a different task domain built on the same engine) without renegotiating the state-machine, UI and audio layers, which is the practical basis for the platform-portability future-work item below.

Discussion:

Three results from this work are particularly notable. First, it seems that engineering dominates diagnosing the inference backend, designing freeze and reset semantics for the states not used during training, fixing up the render-pipeline material resets, and managing the audio listener took a similar amount of time to training itself, and none of it is really covered in a standard reinforcement learning curriculum or sandbox exercise. Therefore, teams should

allocate sufficient resources for integration activities as a first-class activity and develop the whole inference pipeline early and often; the GPU backend incompatibility was not detectable during the training phase. Second, the reward design results suggest that: the nectar reward was sparse but goal-aligned, and it shaped the policy cleanly, without the instability and hysteresis of more complicated reward functions. The practical implication is to reap the reward design simplifications offered by starting from a goal-aligned reward and only adding to the terms when the system has demonstrated that they are needed. Finally, the value of the learned variability came through: the same property that makes the policy hard to formally characterize also makes it significantly more interesting to play against, because the flight patterns cannot be easily predicted or exploited. This is a good example of a machine learning design choice buying into a gameplay feature; by using domain randomization to achieve generalization during training, we were able to get replayability during playtesting. This directly ties into the core thesis of learned behaviour being interestingly different from authored behaviour from 2 and 3; here, a machine learning technique gives variability, and the variability in turn gives replay value.

Limitations:

This study trains and implements a single NPC in a single environment - the training of multiple NPCs and the policy transfer to other tasks is outside the scope of this research. The behavioural quality was evaluated during the play tests and qualitative observations instead of a formal controlled user study, so all statements about believability are based on subjective measures. The difficulty of the agent was fixed once and for all after the training process, and the study only targets the Windows standalone platform as the deployment target.

Beyond these, six further limitations deserve explicit mention. Quantitative instrumentation: this study did not systematically log per-match reward, success rate, task-completion time, inference latency, frame rate, or the exact PPO hyperparameter configuration during training and playtesting; the account given here is consequently qualitative and behavioural rather than statistically quantified, and this is the single most important gap for a follow-up study to close. Statistical validation: for the same reason, no formal significance testing or confidence-interval analysis was performed comparing the trained policy against a scripted baseline; the comparison offered in this paper is observational. Computational cost: training was performed on a single workstation without profiling GPU/CPU utilisation or energy draw, so the resource budget required to reproduce this pipeline on comparable hardware is not quantified here. Scalability to multiple NPCs: because only a single trained agent competes against the player at a time, the behaviour of the shared observation/action design under two or more simultaneous learned NPCs including any contention for the same nectar targets is untested and may require reward or observation changes to remain stable. Generalization across game genres: the ten-value, task-specific observation encoding was designed around a single foraging mechanic, and its transferability to structurally different genres (e.g., combat, stealth, or racing NPCs) are not established by this study. Robustness and deployment scope: while domain randomization varied prefab position and rotation within the Flower Island, the agent was not evaluated against out-of-distribution island geometry, lighting, or physics parameters, and the CPU-Burst inference path was validated only on the specific target hardware and Unity/Sentis versions used during development, not on lower-end hardware or other platforms.

Conclusion and future work:

This paper has demonstrated that a non-playable character governed by learned behaviour can be taken from the design and reinforcement-learning training phases through to a standalone playable game build running on commodity PC hardware. A hummingbird agent trained using PPO in a target domain physics environment containing ten task-centred observations, five continuous actions, and a sparse, goal-aligned reward under domain randomization was exported to ONNX form and incorporated, via CPU-based inference, into a complete game with two modes, a full interface, and layered audio without runtime exceptions during the defined functional test cases. The resulting agent exhibits the varied, purposeful behaviours characteristic of hand-written code but with behavioural variation that differs from traditional deterministic middleware systems. With this demonstration, this paper contributes the technical pipeline detailing the importance of inference backend selection, the freeze/reset mechanics necessary for learned agents in state machines, and the reduced

coupling introduced by minimally expressive reward functions, all of which allow small teams to reproduce the result.

The immediate successor tasks to this research are organised around seven directions. Instrumented quantitative evaluation: re-running training and playtesting with full logging enabled (TensorBoard export, per-match reward, success rate, task-completion time, inference latency and frame rate, and the exact PPO configuration) against a scripted baseline, with the significance-testing protocol described above, to replace the qualitative account in this paper with a statistically supported one the direct response to the quantitative gaps identified in Limitations. Multi-agent reinforcement learning: extending the environment to train swarms or teams of NPCs that either cooperate for shared resources or compete directly, building on early cooperative-NPC evidence in the literature, and addressing the multi-agent scalability limitation noted above. Transfer and curriculum learning: targeting additional engine environments and task domains to measure how much of the trained policy transfers directly versus requiring a staged curriculum and quantifying the retraining cost of porting the pipeline to a structurally different genre. Player-adaptive difficulty: introducing dynamic adjustment of the NPC's effective skill (for example by conditioning the policy or its inference-time exploration on a running estimate of player performance) to create balanced contests across differing skill levels, connecting this pipeline to the dynamic-difficulty-adjustment literature Cross-platform deployment: validating the CPU-Burst inference path on mobile and console targets in addition to Windows Standalone, given that the platform independence of ONNX/Sentis inference is one of this pipeline's practical selling points. Formal user evaluation: conducting a controlled user study measuring perceived believability and engagement relative to a scripted alternative, replacing the qualitative playtest assessment used in this paper. Statistical robustness: repeating training and evaluation across multiple random seeds with the significance testing protocol described in Materials and Methods, to place the eventual quantitative comparisons on firmer statistical footing.

Acknowledgement:

The authors extend their gratitude to the Department of Computer Software Engineering, University of Engineering & Technology, Mardan, for providing laboratory resources and evaluation feedback during the Final Year Design Project from which this work was produced, and to the open-source communities behind the Unity ML-Agents toolkit and Blender for making the tools and documentation central to this research available.

Author's Contribution:

Namos Khan performed the system implementation, reinforcement learning training and integration, testing, and drafted the manuscript. Imran Maqsood was responsible for supervision, conceptual guidance, methodology review, and critical revision of the manuscript. Hijab Noor Ul Amin contributed to the environment and asset design, game system implementation, testing, and manuscript drafting. Mareena Karim contributed to reviewing the research paper, provided valuable guidance on the manuscript, and assisted in improving the quality and structure of the paper. Sadeeq Jan contributed to reviewing the research paper, offered guidance throughout the writing process, and provided constructive feedback to enhance the manuscript. All authors reviewed, approved, and agreed to the final version of this article.

Conflict of Interest:

The authors confirm that they have no conflict of interest regarding the publication of this paper.

Project Details:

This paper was published as part of the Final Year Design Project "Adaptive NPCs with Machine Learning (Hummingbird)" at the Department of Computer Software Engineering, University of Engineering & Technology, Mardan. This research did not receive any funding.

References:

[1] "Artificial Intelligence for Games - 2nd Edition | Elsevier Shop." Accessed: Jul. 26, 2026. [Online]. Available: https://shop.elsevier.com/books/artificial-intelligence-for-games/millington/978-0-12-374731-0

[2] G. N. Yannakakis and J. Togelius, "Artificial intelligence and games," *Artif. Intell. Games*, pp. 1–337, Feb. 2018, doi: 10.1007/978-3-319-63519-4/COVER.

[3] K. Souchleris, G. K. Sidiropoulos, and G. A. Papakostas, "Reinforcement Learning in Game Industry—Review, Prospects and Challenges," *Appl. Sci. 2023, Vol. 13, Page 2443*, vol. 13, no. 4, p. 2443, Feb. 2023, doi: 10.3390/APP13042443.

[4] Sutton, R. S., & Barto, A. G, "Reinforcement learning: An introduction, 2nd ed.," *MIT Press*, 2018, [Online]. Available: https://psycnet.apa.org/record/2019-19679-000

[5] T. Wang *et al.*, "player2vec: A Language Modeling Approach to Understand Player Behavior in Games," Jun. 2024, Accessed: Aug. 04, 2026. [Online]. Available: https://arxiv.org/pdf/2404.04234v3

[6] V. Raj, "A Review on AI-Driven Game Development: Reinforcement Learning and Adaptive NPCs in Modern Games," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 13, no. 12, pp. 3151–3155, Dec. 2025, doi: 10.22214/IJRASET.2025.76668.

[7] M. Korkiakoski, S. Sheikhi, J. Nyman, J. Saariniemi, K. Tapio, and P. Kostakos, "An Empirical Evaluation of AI-Powered Non-Player Characters' Perceived Realism and Performance in Virtual Reality Environments," Jul. 2025, Accessed: Aug. 04, 2026. [Online]. Available: https://arxiv.org/pdf/2507.10469

[8] V. R. K, M. Maanas, A. T. S, A. K. P, and R. P. D, "A Review on AI-Driven Game Development: Reinforcement Learning and Adaptive NPCs in Modern Game Design," Dec. 2025, doi: 10.36227/TECHRXIV.176617583.38277587/V1.

[9] C. Tan, "Comparative Study of Reinforcement Learning Performance Based on PPO and DQN Algorithms," *Appl. Comput. Eng.*, vol. 175, no. 1, pp. 30–36, Jul. 2025, doi: 10.54254/2755-2721/2025.AST24879.

[10] K. Chrysafiadi, M. Kamitsios, and M. Virvou, "Fuzzy-based dynamic difficulty adjustment of an educational 3D-game," *Multimed. Tools Appl. 2023 8218*, vol. 82, no. 18, pp. 27525–27549, Feb. 2023, doi: 10.1007/S11042-023-14515-W.

[11] J. Zhang, "Implementation and effect evaluation of dynamic difficulty adjustment based on reinforcement learning in Multiplayer Online Battle Arena games," *Sci. Technol. Eng. Chem. Environ. Prot.*, vol. 1, no. 10, Dec. 2024, doi: 10.61173/R8885T34.

[12] P. Zhang, X. Ma, Z. Pan, X. Li, and K. Xie, "Multi-agent cooperative reinforcement learning in 3D virtual world," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6145 LNCS, no. PART 1, pp. 731–739, 2010, doi: 10.1007/978-3-642-13495-1_90/SAVE-RESEARCH.

[13] R. Duncan, "Using Reinforcement Learning to Train In-game Non-Player Characters (NPCs)".

[14] A. Juliani *et al.*, "Unity: A General Platform for Intelligent Agents," Sep. 2018, Accessed: Jul. 26, 2026. [Online]. Available: https://arxiv.org/pdf/1809.02627

[15] J. Orkin, "Three States and a Plan: The A.I. of F.E.A.R.," 2006.

[16] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nat. 2015 5187540*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.

[17] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nat. 2016 5297587*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.

[18] OpenAI *et al.*, "Dota 2 with Large Scale Deep Reinforcement Learning," Dec. 2019, Accessed: Jul. 26, 2026. [Online]. Available: https://arxiv.org/pdf/1912.06680

[19] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nat. 2019 5757782*, vol. 575, no. 7782, pp. 350–354, Oct. 2019, doi: 10.1038/s41586-019-1724-z.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. K. Openai, "Proximal Policy Optimization Algorithms," Jul. 2017, Accessed: Jul. 26, 2026. [Online]. Available: https://arxiv.org/pdf/1707.06347

[21] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," *35th Int. Conf. Mach. Learn. ICML 2018*, vol. 4, pp. 2587–2601, Feb. 2018, Accessed: Aug. 04, 2026. [Online]. Available: https://arxiv.org/pdf/1802.09477

[22] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *arXiv:1801.01290*, 2018, [Online]. Available: https://arxiv.org/abs/1801.01290

[23] V. Mnih *et al.*, "Asynchronous Methods for Deep Reinforcement Learning," *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 4, pp. 2850–2869, Feb. 2016, Accessed: Aug. 04, 2026. [Online]. Available: https://arxiv.org/pdf/1602.01783

[24] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep Reinforcement Learning that Matters," *32nd AAAI Conf. Artif. Intell. AAAI 2018*, pp. 3207–3214, Sep. 2017, doi: 10.1609/aaai.v32i1.11694.

[25] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, and M. G. Bellemare, "Deep Reinforcement Learning at the Edge of the Statistical Precipice," *Adv. Neural Inf. Process. Syst.*, vol. 35, pp. 29304–29320, Aug. 2021, Accessed: Aug. 04, 2026. [Online]. Available: https://arxiv.org/pdf/2108.13264

[26] "Toolkit Documentation - Unity ML-Agents Toolkit." Accessed: Jul. 26, 2026. [Online]. Available: https://unity-technologies.github.io/ml-agents/ML-Agents-Toolkit-Documentation/

[27] A. Merritt, "ONNX-Scala: typeful, functional deep learning / Dotty meets an open AI standard (open-source talk)," pp. 33–33, Nov. 2020, doi: 10.1145/3426426.3434120.

[28] "Sentis overview | Sentis | 2.6.1." Accessed: Jul. 26, 2026. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ai.inference@2.6/manual/index.html

[29] "Blender 4.0 Reference Manual — Blender Manual." Accessed: Jul. 26, 2026. [Online]. Available: https://docs.blender.org/manual/en/4.0/

[30] T. Aris, V. Ustun, and R. Kumar, "Learning to Take Cover with Navigation-Based Waypoints via Reinforcement Learning," *Proc. Int. Florida Artif. Intell. Res. Soc. Conf. FLAIRS*, vol. 36, 2023, doi: 10.32473/FLAIRS.36.133348.