Final Project

1. Problem Description

This project focuses on the development of an autonomous game-playing system based on Reinforcement Learning (RL), capable of playing the classic arcade game Pac-Man without human intervention. The system faithfully recreates Pac-Man Level 1, including authentic game mechanics, real-time dynamics, scoring rules, and adversarial ghost behaviors.

2. Technological Aspect

2.1 Technological Survey

The system integrates multiple technological components into a unified architecture:
A grid-based game environment that simulates the Pac-Man maze, pellets, collisions, and scoring mechanics.
An adversarial Ghost AI system, implementing four ghosts (Blinky, Pinky, Inky, Clyde), each with arcade-accurate behavioral logic.
A Reinforcement Learning agent that learns an optimal policy through interaction with the environment.
The architecture is designed to operate in real time (60 FPS) while also supporting headless execution for accelerated training. The modular design enables independent development and testing of the game environment, ghost behaviors, and learning algorithms.

2.2 Current Solutions Survey

For developing a reinforcement learning-based Pac-Man AI, the main technological choices involve selecting a programming language, game framework, and RL library. The primary programming language options are Python (dominant in ML/RL with extensive libraries like PyTorch, TensorFlow, and Stable-Baselines3), C++ (offering high performance but with a steeper learning curve and limited RL ecosystem), and Java (cross-platform but with fewer mature RL frameworks). For game development, Pygame provides a lightweight 2D library with simple integration into Python RL frameworks, Unity offers professional tooling but adds unnecessary overhead for a simple tile-based game, and custom implementations provide maximum control but require significant development time. RL framework options include Stable-Baselines3 (production-ready implementations of DQN and PPO with excellent documentation), TensorFlow/Keras (flexible but requiring more custom implementation), and RLlib (designed for distributed RL, which is overkill for single-machine training).

2.3 Parameters for Solution Comparison

The technological choices were evaluated based on several key parameters, each weighted according to project priorities. Development speed was given high priority due to the semester timeline requiring rapid iteration and prototyping. The RL library ecosystem was considered critical since the project specifically requires DQN and PPO implementations for algorithm comparison. The learning curve received high weight because the team has limited prior experience with some technologies and needs to be productive quickly. Performance was rated as medium priority—the system must achieve 60 FPS rendering and over 1000 FPS headless training, but raw performance is less important than development speed for an academic project. Documentation quality and community support were rated high and medium respectively, as they are essential for troubleshooting and learning but not project-critical. Flexibility for customization (particularly reward functions and state representations) and cross-platform support were both rated medium priority, as the team uses mixed operating systems but these are not blocking factors.

2.4 Decision Justifications

Python 3.9+ was selected as the programming language because it provides the best RL library ecosystem (Stable-Baselines3, Gymnasium, PyTorch are all Python-native), enables rapid prototyping for iterating on reward functions and network architectures, has a low learning curve with all team members already familiar with it, and offers adequate performance through NumPy's C-optimized operations and PyTorch's compiled backend—headless training exceeds 1000 FPS and rendering achieves the required 60 FPS despite Python being slower than C++. Pygame 2.x was chosen as the game framework because it integrates seamlessly with Python (no inter-process communication needed), is lightweight with minimal overhead for a simple 2D game, provides complete control over the game loop for frame-perfect timing (±3 frames accuracy requirement), and easily supports headless mode for fast training. Stable-Baselines3 with PyTorch backend was selected as the RL framework because it provides production-ready, tested implementations of both DQN and PPO (eliminating the need to implement replay buffers or advantage estimation from scratch), has excellent documentation and a simple API that aids learning, is fully compatible with the Gymnasium interface standard, and uses PyTorch's more Pythonic API with dynamic computation graphs that make debugging easier. The technology stack prioritizes development speed and educational value over maximum raw performance, which aligns with the project's focus on comparative RL research within a semester timeline rather than production optimization.

3. Stakeholders' description

3.1 Primary Stakeholders

Development Team (Kylee, Peleg, Shmuel)

- Role: Developers, testers, users
- Interest: Successfully complete course project, demonstrate RL competency
- Requirements: Clear specifications, testable components, good documentation

Course Instructors

- Role: Evaluators
- Interest: Assess understanding of software engineering and RL concepts
- Requirements: Professional documentation, working demonstration, clear analysis

3.2 Secondary Stakeholders

Academic Community

- Role: Future researchers, students
- Interest: Reproducible results, educational resource
- Requirements: Open-source code, clear documentation, sharable findings

4. Functional requirements (mapped stakeholders described above)

4.1 Game Environment Requirements

4.1.1 Maze and Layout

FR-1.1: The system shall implement Pac-Man Level 1 maze with exact tile layout (28×36 grid)
Priority: Critical
Rationale: Foundation for all gameplay

FR-1.2: The maze shall contain 244 pellets: 240 small pellets and 4 power pellets
Priority: Critical
Rationale: Core game objective

FR-1.3: Power pellets shall be positioned at the four maze corners
Priority: Critical
Rationale: Strategic gameplay element

FR-1.4: The maze shall include two side tunnels connecting left and right edges
Priority: High
Rationale: Important strategic element

FR-1.5: The maze shall mark red zones where ghosts cannot turn upward
Priority: High
Rationale: Affects ghost AI behavior and player strategy

4.1.2 Pac-Man Character

FR-2.1: Pac-Man shall move at 80% maximum speed during Level 1
Priority: High
Rationale: Authentic game mechanics

FR-2.2: Pac-Man shall stop for 1 frame (1/60 second) when eating a regular pellet
Priority: Medium
Rationale: Realistic game feel, affects chase dynamics

FR-2.3: Pac-Man shall stop for 3 frames when eating a power pellet
Priority: Medium
Rationale: Authentic mechanics

FR-2.4: Pac-Man shall support cornering (pre-turn and post-turn mechanics)
Priority: Medium
Rationale: Allows precise control and optimal path execution

FR-2.5: Pac-Man shall die upon collision with a non-frightened ghost
Priority: Critical
Rationale: Core game rule

FR-2.6: Pac-Man shall be able to eat frightened ghosts for points
Priority: Critical
Rationale: Core scoring mechanism

4.1.3 Pellet System

FR-3.1: Small pellets shall award 10 points when eaten
Priority: High
Rationale: Scoring system

FR-3.2: Power pellets shall award 50 points when eaten
Priority: High
Rationale: Scoring system

FR-3.3: Eating a power pellet shall trigger frightened mode for all ghosts
Priority: Critical
Rationale: Core gameplay mechanic

FR-3.4: Frightened mode shall last 6 seconds in Level 1
Priority: High
Rationale: Authentic timing

FR-3.5: The system shall track remaining pellet count
Priority: High
Rationale: Level completion detection

FR-3.6: Level shall complete when all 244 pellets are consumed
Priority: Critical
Rationale: Victory condition

4.1.4 Collision Detection

FR-4.1: The system shall detect tile-based collisions between Pac-Man and ghosts
Priority: Critical
Rationale: Core game mechanics

FR-4.2: Collision detection shall use center-point tile occupancy
Priority: High
Rationale: Authentic Pac-Man mechanics

FR-4.3: The system shall detect pellet consumption when Pac-Man occupies pellet tile
Priority: Critical
Rationale: Core game mechanic

4.2 Ghost AI Requirements

4.2.1 Ghost Modes

FR-5.1: Ghosts shall support three distinct modes: scatter, chase, and frightened
Priority: Critical
Rationale: Defines ghost behavior patterns

FR-5.2: Ghosts shall alternate between scatter and chase modes on fixed schedule:

- Scatter: 7 seconds
- Chase: 20 seconds
- Scatter: 7 seconds
- Chase: 20 seconds
- Scatter: 5 seconds
- Chase: 20 seconds
- Scatter: 5 seconds
- Chase: indefinite

Priority: High
Rationale: Authentic arcade timing

FR-5.3: Ghosts shall reverse direction when mode changes occur
Priority: High
Rationale: Visual indicator of mode change

FR-5.4: Frightened mode shall override scatter/chase modes
Priority: Critical
Rationale: Power pellet mechanic

4.2.2 Blinky (Red Ghost) Behavior

FR-4.1: In chase mode, Blinky shall target Pac-Man's current tile directly
Priority: Critical
Rationale: Defines Blinky's personality

FR-4.2: Blinky shall accelerate ("Cruise Elroy" mode) when 20 pellets remain
Priority: High
Rationale: Increases difficulty, authentic behavior

FR-4.3: Blinky shall further accelerate when 10 pellets remain
Priority: High
Rationale: Authentic behavior

FR-4.4: In Cruise Elroy mode, Blinky shall continue chasing during scatter periods
Priority: Medium
Rationale: Authentic behavior

4.2.3 Pinky (Pink Ghost) Behavior

FR-4.1: In chase mode, Pinky shall target 4 tiles ahead of Pac-Man's current direction
Priority: Critical
Rationale: Defines Pinky's ambush strategy

FR-4.2: When Pac-Man faces upward, Pinky's target shall be 4 tiles up and 4 tiles left (overflow bug simulation)
Priority: Medium
Rationale: Authentic quirk from original game

4.2.4 Inky (Cyan Ghost) Behavior

FR-8.1: In chase mode, Inky shall calculate target using Pac-Man position and Blinky position
Priority: Critical
Rationale: Defines Inky's complex behavior

FR-8.2: Inky's target shall be: 2× vector from Blinky to (2 tiles ahead of Pac-Man)
Priority: Critical
Rationale: Authentic Inky logic

FR-8.3: When Pac-Man faces upward, intermediate offset shall be 2 up, 2 left (overflow bug)
Priority: Medium
Rationale: Authentic quirk

4.2.5 Clyde (Orange Ghost) Behavior

FR-9.1: In chase mode, Clyde shall switch behavior based on distance to Pac-Man
Priority: Critical
Rationale: Defines Clyde's "shy" personality

FR-9.2: When >8 tiles from Pac-Man, Clyde shall target Pac-Man directly
Priority: Critical
Rationale: Approach behavior

FR-9.3: When ≤8 tiles from Pac-Man, Clyde shall target scatter corner
Priority: Critical
Rationale: Retreat behavior

4.2.6 Ghost Movement and Pathfinding

FR-10.1: Ghosts shall use pathfinding to navigate toward target tiles
Priority: Critical
Rationale: Core AI behavior

FR-10.2: At intersections, ghosts shall choose direction minimizing Euclidean distance to target

Priority: Critical
Rationale: Authentic pathfinding logic

FR-10.3: Ghosts shall never voluntarily reverse direction (except on mode changes)
Priority: High
Rationale: Authentic behavior constraint

FR-10.4: Ghosts shall respect wall collision
Priority: Critical
Rationale: Basic movement constraint

FR-10.5: Ghosts shall slow down in side tunnels (40% speed)
Priority: High
Rationale: Strategic gameplay element

FR-10.6: Ghosts shall move at 75% of Pac-Man's speed in Level 1 (normal mode)
Priority: High
Rationale: Balanced gameplay

FR-10.7: Frightened ghosts shall move at 50% speed and wander randomly
Priority: Critical
Rationale: Power pellet mechanic

4.2.7 Ghost House Mechanics

FR-11.1: Ghosts shall start each level in the ghost house
Priority: High
Rationale: Initial state

FR-11.2: Pinky shall leave ghost house immediately at level start
Priority: High
Rationale: Authentic behavior

FR-11.3: Inky shall leave after 30 pellets eaten (Level 1)
Priority: Medium
Rationale: Authentic behavior

FR-11.4: Clyde shall leave after 60 pellets eaten (Level 1)
Priority: Medium
Rationale: Authentic behavior

FR-11.5: Eaten ghosts shall return to ghost house and regenerate
Priority: Critical
Rationale: Core game mechanic

FR-11.6: Ghost eyes shall be visible during return journey
Priority: Low
Rationale: Visual feedback

4.2.8 Scatter Mode Targets

FR-12.1: Each ghost shall have a fixed scatter target tile in their respective corner
Priority: High
Rationale: Defines scatter behavior

FR-12.2: Blinky scatter target: Top-right corner
FR-12.3: Pinky scatter target: Top-left corner
FR-12.4: Inky scatter target: Bottom-right corner
FR-12.5: Clyde scatter target: Bottom-left corner
Priority (all): Medium
Rationale: Authentic scatter positions

4.3 Scoring System Requirements

FR-13.1: Small pellet: 10 points
FR-13.2: Power pellet: 50 points
FR-13.3: First frightened ghost: 200 points
FR-13.4: Second frightened ghost: 400 points
FR-13.5: Third frightened ghost: 800 points
FR-13.6: Fourth frightened ghost: 1600 points
Priority (all): High
Rationale: Authentic scoring system

FR-13.7: Score shall accumulate throughout episode
Priority: High
Rationale: Performance metric

FR-13.8: Score shall reset on death
Priority: High
Rationale: Episode-based evaluation

4.4 RL Agent Requirements

4.4.1 State Representation

FR-14.1: Agent shall receive complete game state observation
Priority: Critical
Rationale: Informed decision-making

FR-14.2: State shall include: Pac-Man position, all ghost positions, ghost states, pellet map, valid moves
Priority: Critical
Rationale: Complete information

FR-14.3: State shall be normalized to [0,1] or [-1,1] range
Priority: High
Rationale: Neural network stability

4.4.2 Action Selection

FR-15.1: Agent shall output action from discrete set: {Up, Down, Left, Right}
Priority: Critical
Rationale: Game controls

FR-15.2: Agent shall respect wall constraints (invalid actions become no-op)
Priority: High
Rationale: Prevents impossible moves

FR-15.3: Agent shall make decisions at every frame (60 FPS)
Priority: High
Rationale: Real-time gameplay

4.4.3 Training Interface

FR-14.1: System shall provide Gym-compatible environment interface
Priority: Critical
Rationale: Standard RL framework integration

FR-14.2: Environment shall implement reset(), step(), render() methods
Priority: Critical
Rationale: Gym API compliance

FR-14.3: step() shall return (observation, reward, done, info)
Priority: Critical
Rationale: Standard return format

FR-14.4: System shall support episodic training (restart on death)
Priority: Critical
Rationale: RL training paradigm

4.4.4 Reward Function

FR-14.1: System shall compute reward signal after each action
Priority: Critical
Rationale: RL feedback mechanism

FR-14.2: Reward function shall be configurable via parameters
Priority: High
Rationale: Experimentation flexibility

FR-14.3: System shall support dense reward shaping (frequent small rewards)
Priority: High
Rationale: Learning efficiency

4.4.5 Agent Training

FR-18.1: System shall support training with PPO algorithm
Priority: Critical
Rationale: Primary algorithm

FR-18.2: System shall support training with DQN algorithm
Priority: High
Rationale: Comparison baseline

FR-18.3: Training shall be resumable from checkpoints
Priority: High
Rationale: Long training sessions

FR-18.4: System shall save model weights periodically
Priority: High
Rationale: Prevent data loss

FR-18.5: System shall log training metrics (rewards, loss, epsilon, etc.)
Priority: High
Rationale: Performance monitoring

4.4.6 Agent Evaluation

FR-19.1: System shall support deterministic evaluation mode (no exploration)
Priority: High
Rationale: Fair performance comparison

FR-19.2: System shall run evaluation episodes periodically during training
Priority: High
Rationale: Track learning progress

FR-19.3: System shall compute statistics: mean reward, survival time, pellets eaten
Priority: High
Rationale: Performance metrics

FR-19.4: System shall record evaluation episodes as video
Priority: Medium
Rationale: Qualitative analysis

4.5 Visualization Requirements

FR-20.1: System shall render game state in real-time (60 FPS minimum)
Priority: High
Rationale: Smooth visualization

FR-20.2: Visualization shall display: maze, Pac-Man, ghosts, pellets, score
Priority: Critical
Rationale: Complete game state

FR-20.3: Ghosts shall be color-coded: Red (Blinky), Pink (Pinky), Cyan (Inky), Orange (Clyde)
Priority: High
Rationale: Easy identification

FR-20.4: Frightened ghosts shall turn blue
Priority: High
Rationale: Visual feedback

FR-20.5: System shall support headless mode (no rendering)
Priority: High
Rationale: Fast training

FR-20.6: System shall display current score and pellets remaining
Priority: Medium
Rationale: Gameplay information

FR-20.7: System shall optionally display debug information (ghost targets, modes)
Priority: Low
Rationale: Development aid

4.6 Data Logging and Analysis Requirements

FR-21.1: System shall log episode statistics: total reward, survival frames, pellets eaten, ghosts eaten
Priority: High
Rationale: Performance tracking

FR-21.2: System shall generate training curves: reward vs. episode, loss vs. step
Priority: High
Rationale: Convergence analysis

FR-21.3: System shall export logs to CSV format
Priority: Medium
Rationale: External analysis

FR-21.4: System shall support Tensorboard logging
Priority: High
Rationale: Real-time monitoring

FR-21.5: System shall save hyperparameters with model checkpoints
Priority: Medium
Rationale: Reproducibility

4.7 Configuration Requirements

FR-22.1: All hyperparameters shall be configurable via config file or CLI arguments
Priority: High
Rationale: Experimentation flexibility

FR-22.2: Configurable parameters shall include: learning rate, batch size, discount factor, epsilon decay, network architecture
Priority: High
Rationale: Tuning capability

FR-22.3: System shall validate configuration on startup
Priority: Medium
Rationale: Prevent invalid experiments

5. Non-functional requirements

5.1 Performance Requirements

NFR-1: The game simulation shall run at 60 FPS minimum on mid-range hardware (Intel i5/Ryzen 5, 8GB RAM)
Priority: High
Rationale: Real-time gameplay requirement

NFR-2: Agent decision-making shall take <100ms per action
Priority: High
Rationale: Real-time responsiveness

NFR-3: Training shall support GPU acceleration when available
Priority: High
Rationale: Reduced training time

NFR-4: Headless training shall run at >1000 FPS
Priority: Medium
Rationale: Fast experimentation

NFR-5: Memory usage shall not exceed 4GB during training
Priority: Medium
Rationale: Commodity hardware compatibility

5.2 Accuracy and Correctness Requirements

NFR-6: Ghost behaviors shall match Pac-Man Dossier specifications within 5% error margin
Priority: Critical
Rationale: Authentic game mechanics

NFR-7: Timing mechanics (mode switches, frightened duration) shall be accurate within ±3 frames
Priority: High
Rationale: Gameplay consistency

NFR-8: Score calculation shall be 100% accurate
Priority: High
Rationale: Performance measurement integrity

5.3 Reliability Requirements

NFR-9: System shall handle exceptions gracefully without crashing
Priority: High
Rationale: Robust training

NFR-10: Training shall be resumable after interruption
Priority: High
Rationale: Long experiments

NFR-11: Saved models shall be backward compatible across minor version updates
Priority: Medium
Rationale: Model reuse

NFR-12: System shall validate environment correctness via unit tests (>80% coverage)
Priority: High
Rationale: Code quality

5.4 Usability Requirements

NFR-13: System shall be installable via single pip install command
Priority: High
Rationale: Easy setup

NFR-14: Documentation shall include: setup guide, usage examples, API reference
Priority: High
Rationale: Accessibility

NFR-15: Training script shall require <10 parameters to start basic training
Priority: Medium
Rationale: Ease of use

NFR-16: Visualization shall clearly distinguish game elements (Pac-Man, ghosts, pellets)
Priority: High
Rationale: Interpretability

5.5 Maintainability Requirements

NFR-17: Code shall follow PEP 8 style guidelines
Priority: Medium
Rationale: Code readability

NFR-18: All public functions shall have docstrings
Priority: Medium
Rationale: Code documentation

NFR-19: System shall use modular architecture (separate game, AI, training modules)
Priority: High
Rationale: Code organization

NFR-20: Git repository shall include: README, requirements.txt, .gitignore
Priority: High
Rationale: Standard practices

5.6 Portability Requirements

NFR-21: System shall run on Windows, macOS, and Linux
Priority: High
Rationale: Platform independence

NFR-22: System shall support Python 3.9, 3.10, 3.11
Priority: Medium
Rationale: Compatibility

NFR-23: Dependencies shall be installable via pip, managed by UV during development
Priority: High
Rationale: Easy deployment

5.7 Scalability Requirements

NFR-24: System shall support parallel environment execution (vectorized environments)
Priority: Medium
Rationale: Training speedup

NFR-25: Training architecture shall allow multi-GPU training
Priority: Low
Rationale: Future expansion

5.8 Security Requirements

NFR-26: System shall not require network access during training (optional for logging)
Priority: Low
Rationale: Air-gapped environments

NFR-27: Model checkpoints shall be stored securely with appropriate permissions
Priority: Low
Rationale: Data integrity

5.9 Reproducibility Requirements

NFR-28: All experiments shall use fixed random seeds
Priority: High
Rationale: Reproducible results

NFR-29: System shall log: git commit hash, hyperparameters, environment details
Priority: Medium
Rationale: Experiment tracking

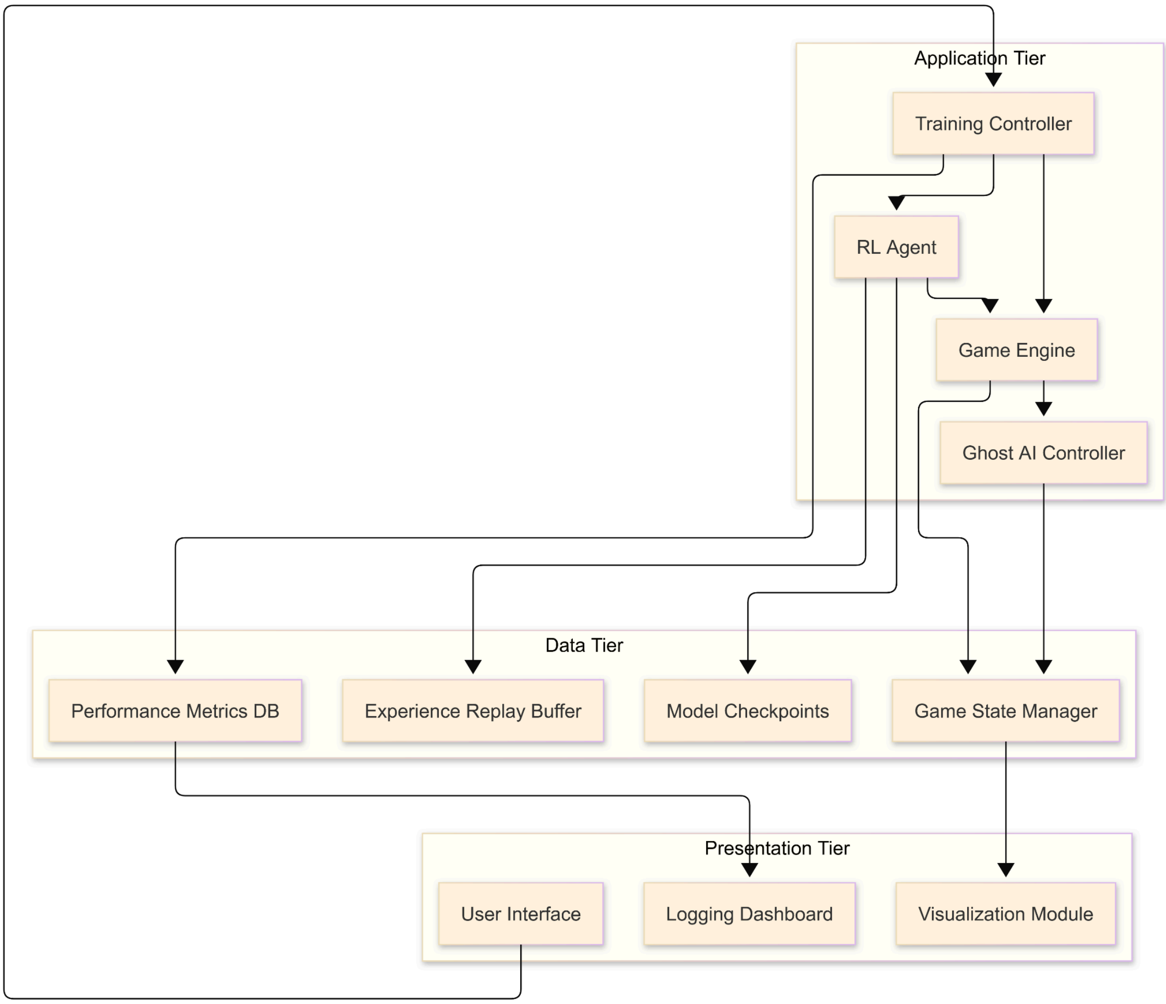
NFR-30: Random number generation shall be deterministic when seed is set
Priority: High
Rationale: Debugging capability

6. Architecture and high-level design

6.1 System Architecture

6.1.1. Architectural Description and Design: Roles, Activities and Data

The Self-Playing Pac-Man AI system follows a data-driven architecture centered on the interaction between three core processing components: the Training Controller, Game Engine and RL Agent.

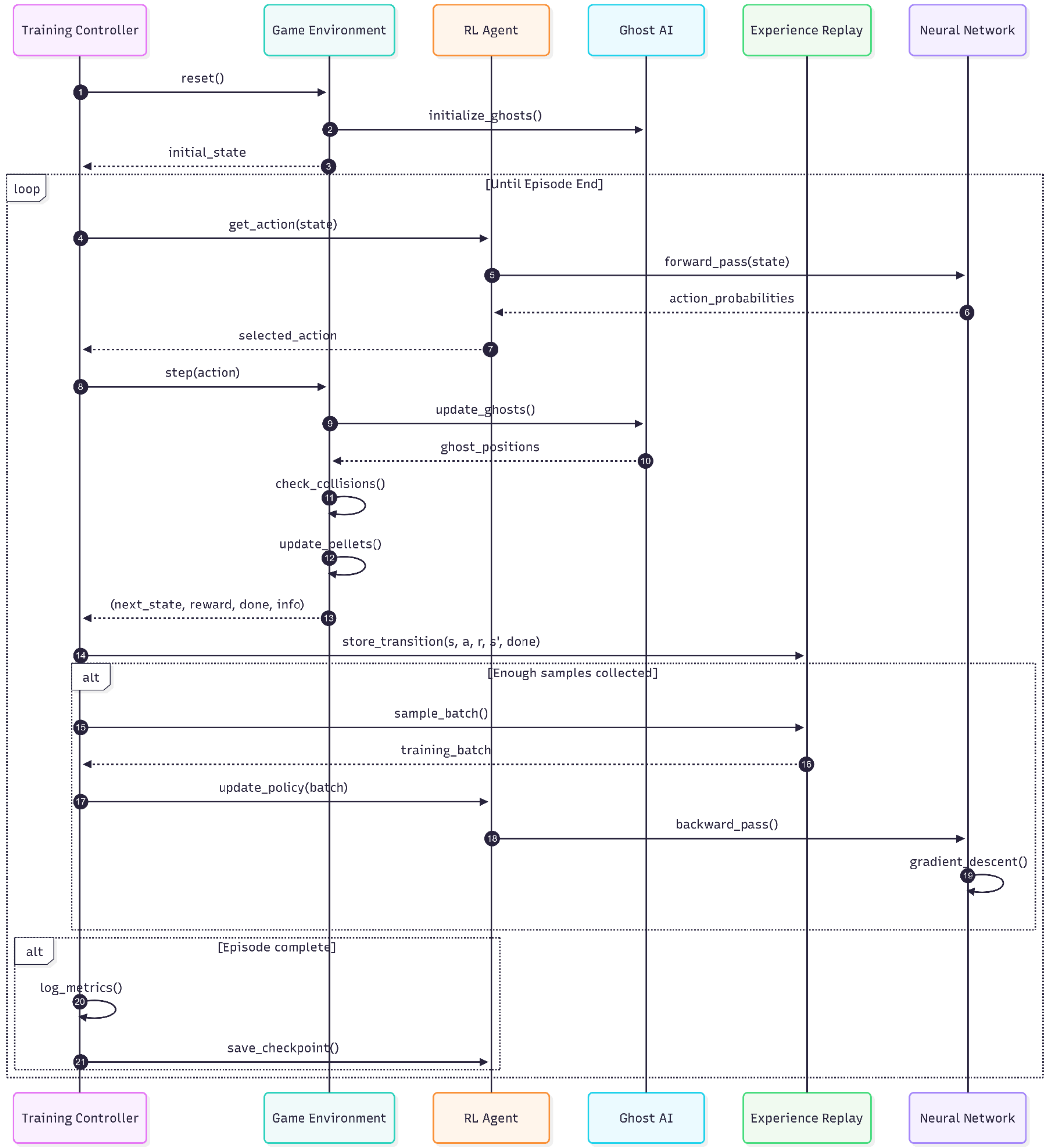


Component Roles:

Component	Role	Primary Activities	Data Managed
Game Engine	Core simulation controller	Update game state, handle collisions, manage pellets	Game state, positions, scores
Ghost AI Controller	Adversarial agents	Pathfinding, mode management, targeting	Ghost states, targets, modes
RL Agent	Learning player	Action selection, learning updates	Policy network, Q-values
Training Controller	Orchestration	Episode management, hyperparameter control	Training metrics, checkpoints
Visualization Module	Real-time display	Render game state, show debug info	Frame buffers, sprites
State Manager	Data persistence	Save/load game configurations	State snapshots, history
Experience Replay	Training data	Store transitions for off-policy learning	(s, a, r, s', done) tuples

6.1.2 Main System Internal Scenario

Scenario: RL Agent Training Episode

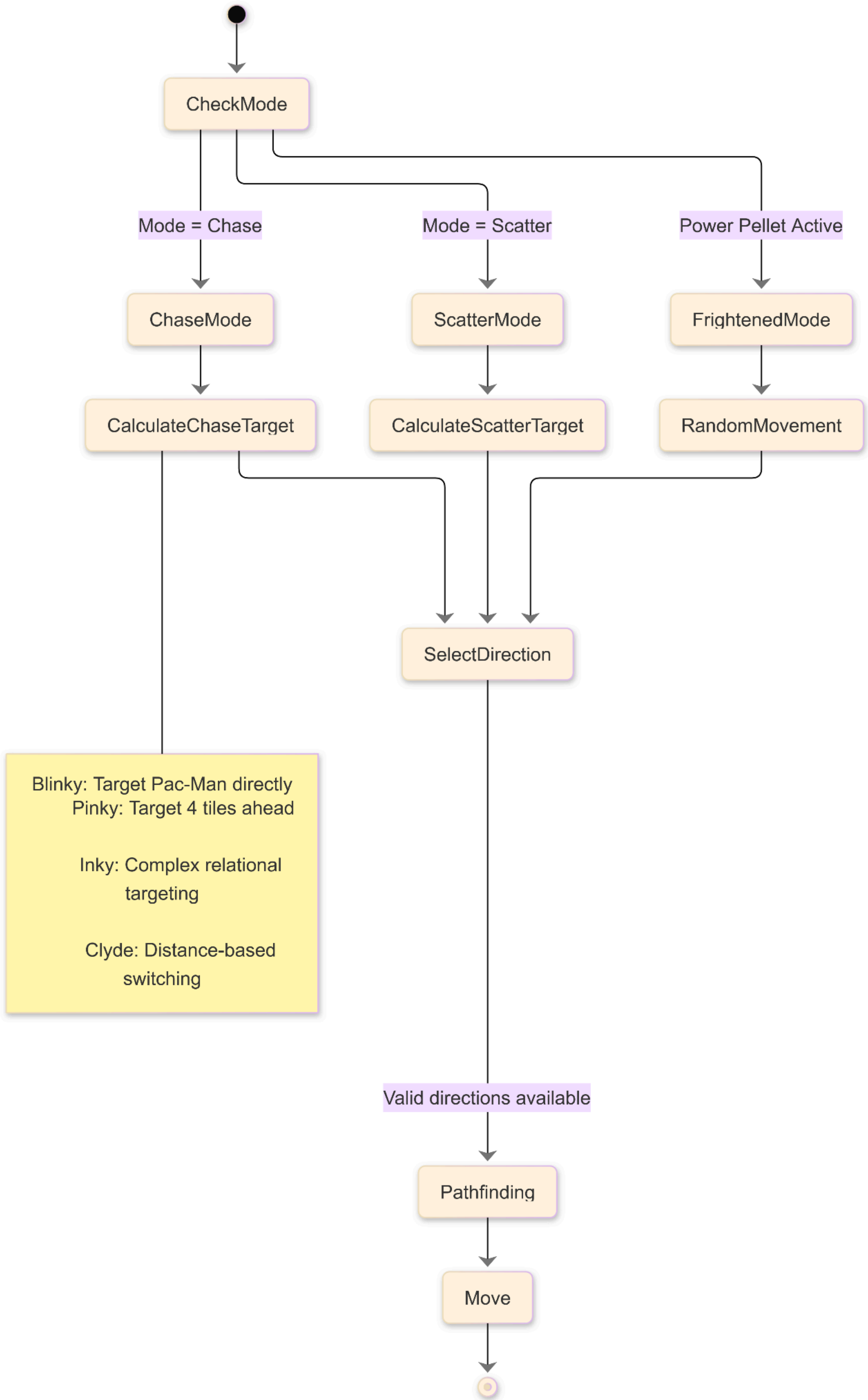


Flow Description:

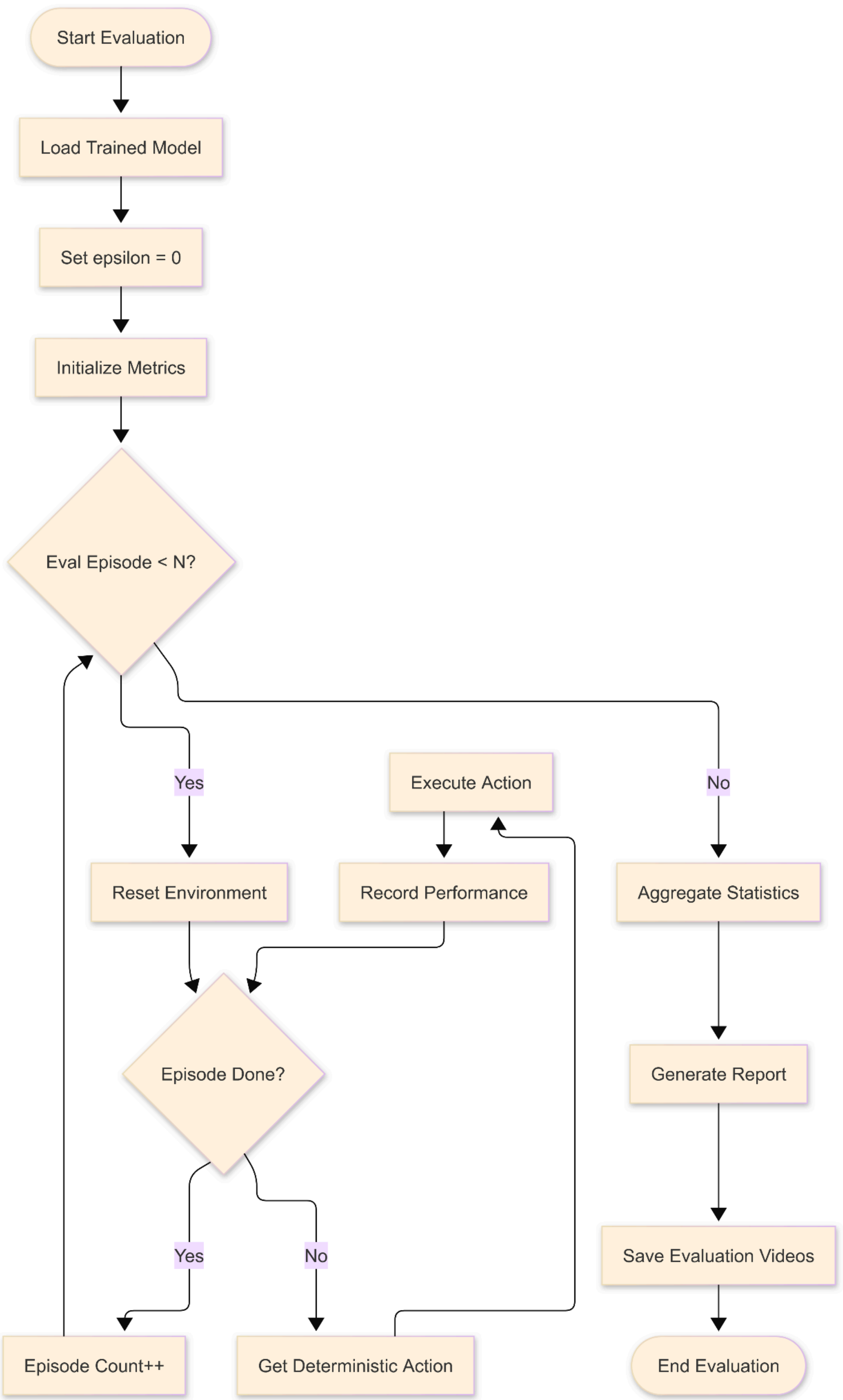
1. **Initialization**: Training controller resets environment, initializes ghosts and agent
2. **Action Selection**: Agent observes state, selects action via epsilon-greedy or policy sampling
3. **Environment Step**: Game engine executes action, ghosts update positions, collisions checked
4. **Reward Calculation**: Reward signal computed based on pellets eaten, survival, ghost interactions
5. **Experience Storage**: Transition stored in replay buffer
6. **Learning Update**: When sufficient samples collected, agent performs gradient descent on sampled batch
7. **Episode Completion**: Metrics logged, model checkpoint saved
8. **Iteration**: Process repeats for configured number of episodes

6.1.3 Additional System Internal Scenarios

Scenario 1: Ghost AI Decision Making



Scenario 2: Model Evaluation Flow

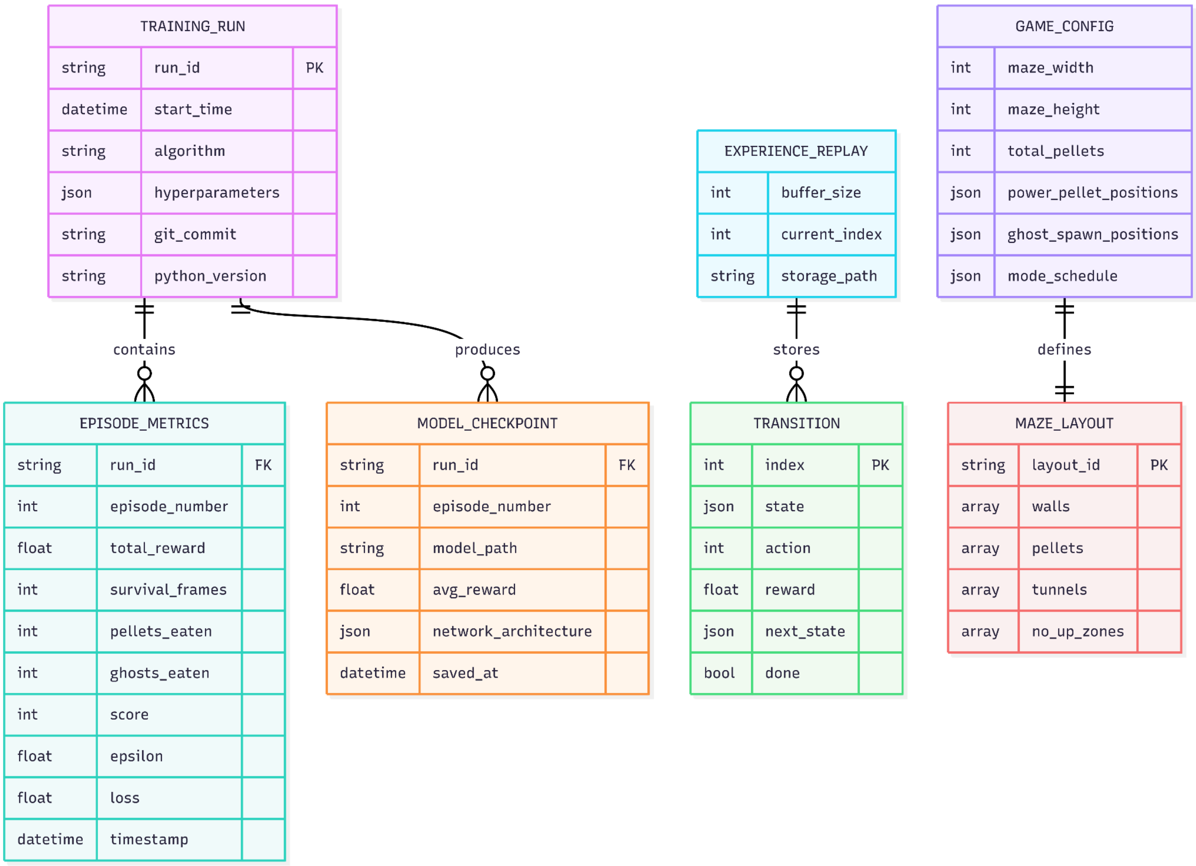


6.2 Design

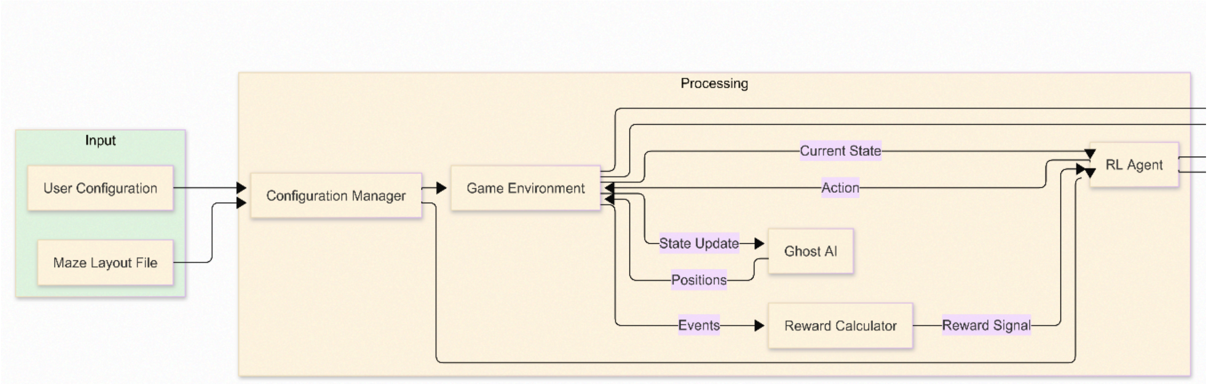
6.2.1 Data Design - Database Description

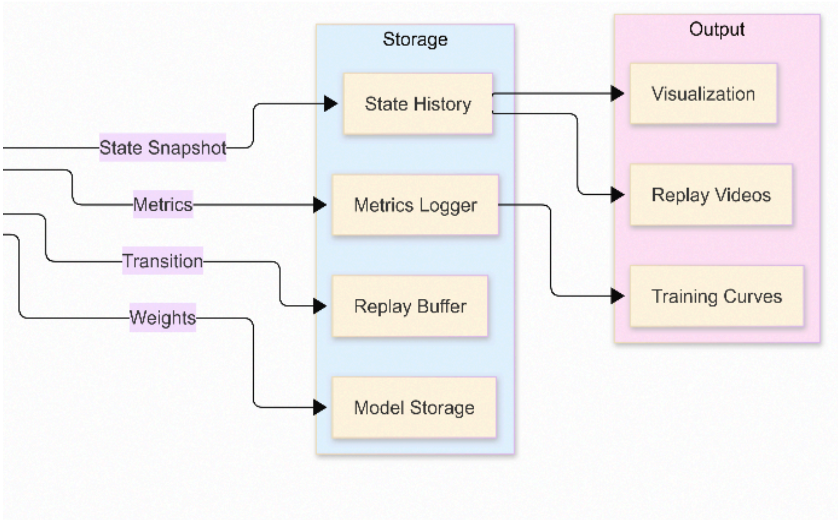
The system uses file-based storage rather than traditional databases, optimized for fast I/O during training:

Storage Schema:



6.2.2 Data Flow – Data Flow Between System Modules

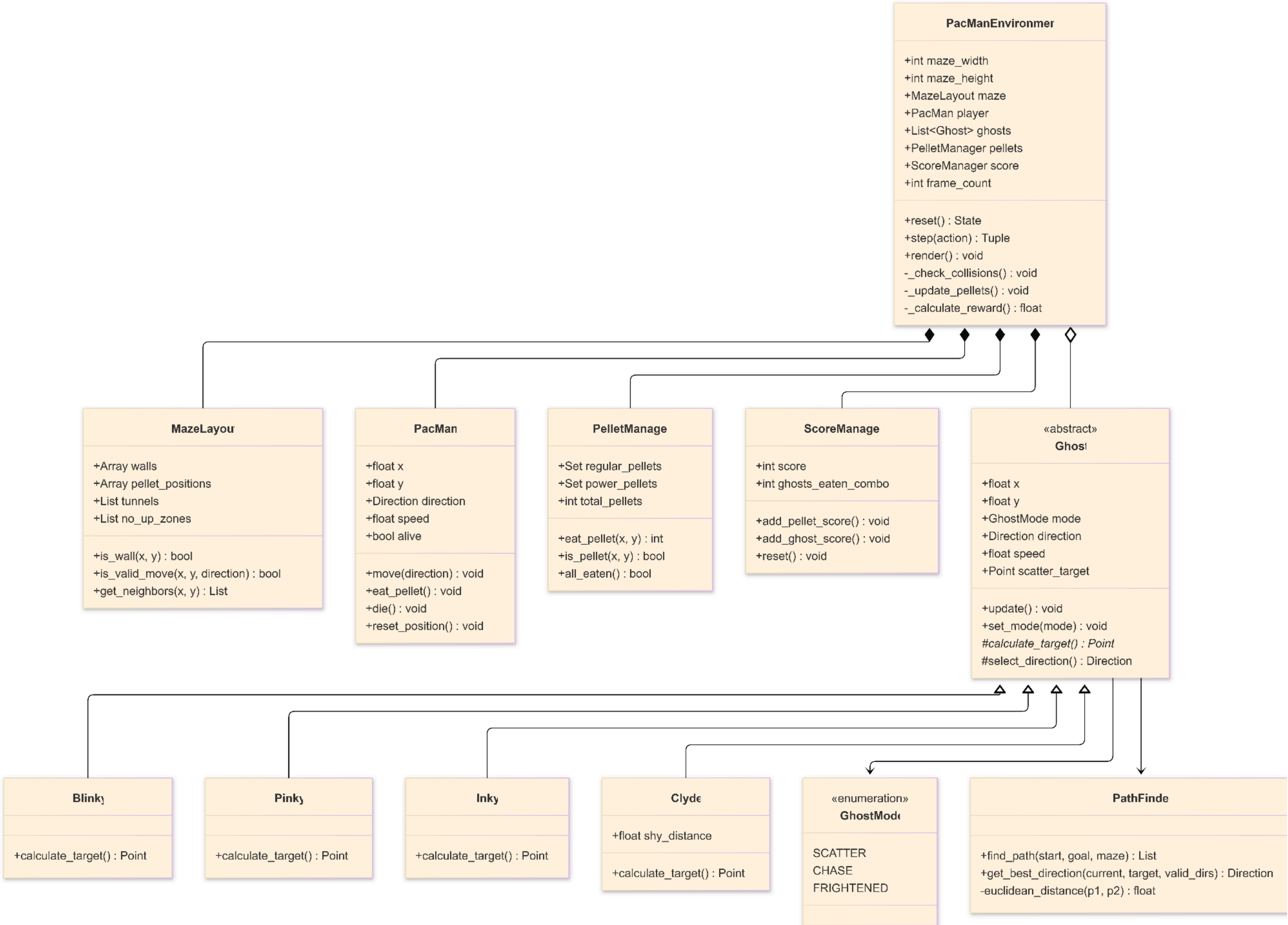


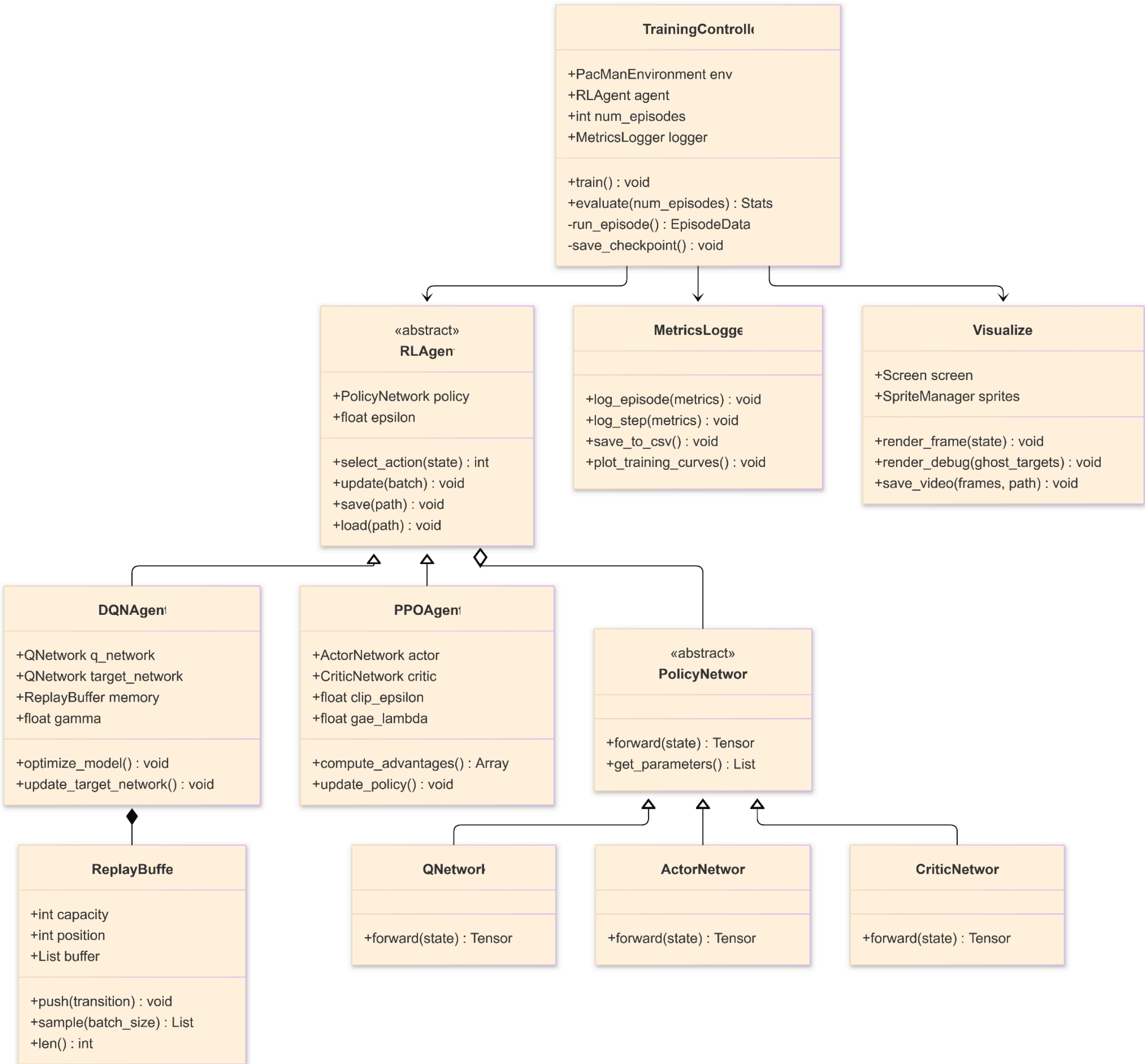


Data Flow Description:

1. **Configuration Flow**:
    - User provides hyperparameters, algorithm choice, training settings
    - Maze layout loaded from JSON file
    - Configuration manager initializes all components
2. **Training Loop Flow**:
    - Environment generates current state observation
    - State passed to RL agent for action selection
    - Action executed in environment
    - Ghost AI updates positions based on mode and targeting rules
    - Reward calculator determines reward signal
    - Transition (s, a, r, s', done) stored in replay buffer
    - Agent samples mini-batch for learning update
3. **Persistence Flow**:
    - State history saved for visualization replay
    - Model checkpoints periodically saved
    - Training metrics logged to CSV and TensorBoard
4. **Output Flow**:
    - Real-time visualization renders current game state
    - Training curves plotted from metrics
    - Episode videos generated from state history

6.2.3 Structural Design - Class Diagram





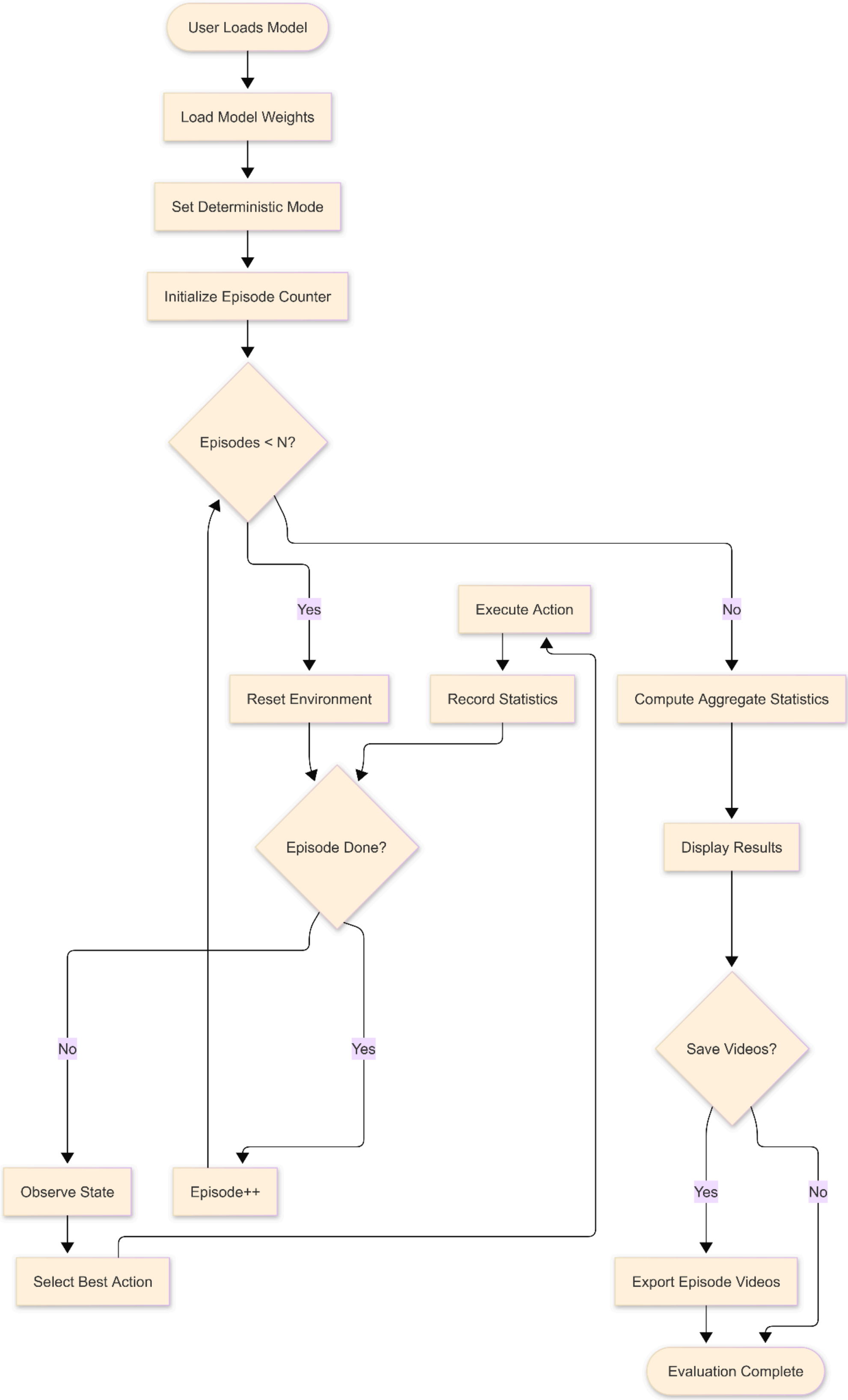
Key Design Patterns:

1. **Strategy Pattern**: Different ghost behaviors (Blinky, Pinky, Inky, Clyde) implement common Ghost interface
2. **Factory Pattern**: Ghost creation based on ghost type
3. **Observer Pattern**: MetricsLogger observes training events
4. **State Pattern**: Ghost mode transitions (Scatter, Chase, Frightened)
5. **Template Method**: RL Agent defines common training structure, DQN/PPO implement specifics

6.2.4 Interactions Design

a. Use Cases

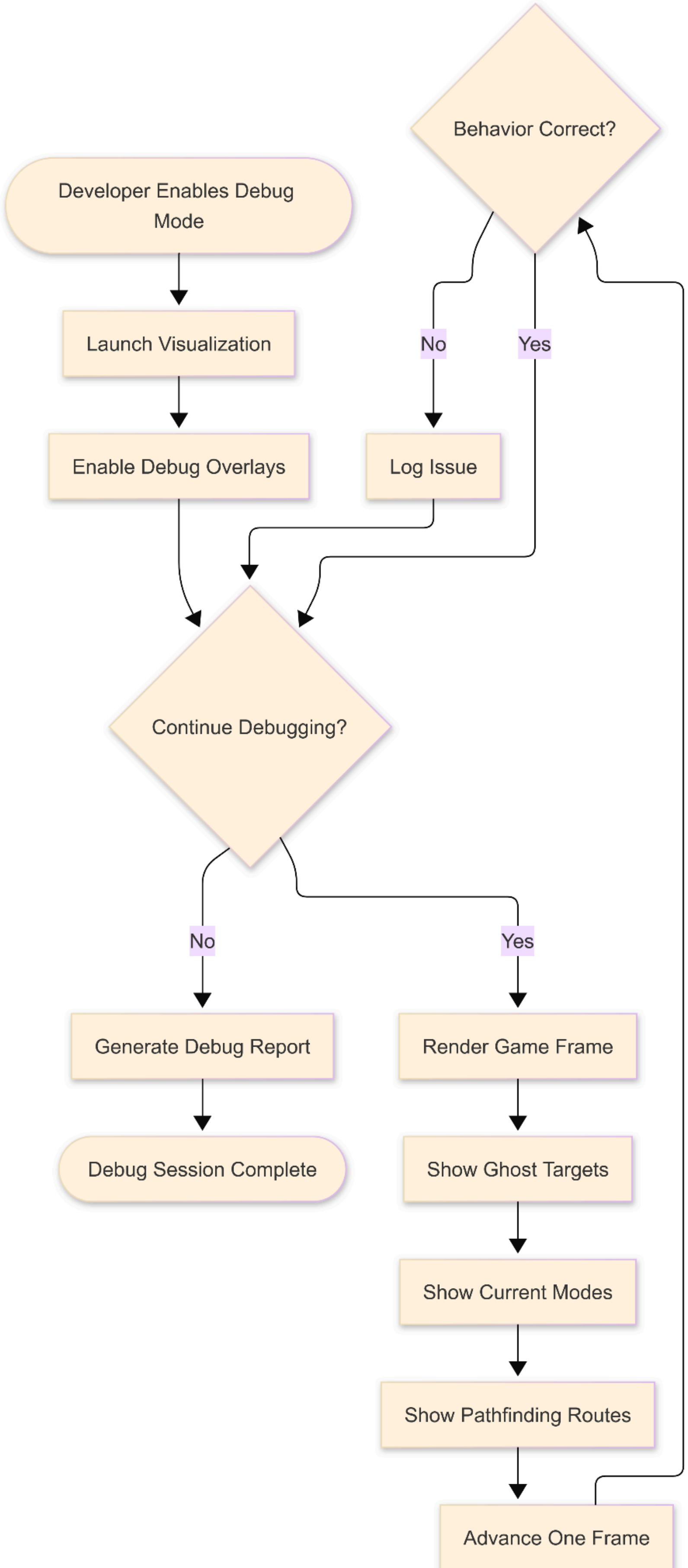
UC-1: Evaluate Trained Model



Actors: Researcher
Preconditions: Trained model

available
Postconditions: Performance statistics computed, optional videos saved

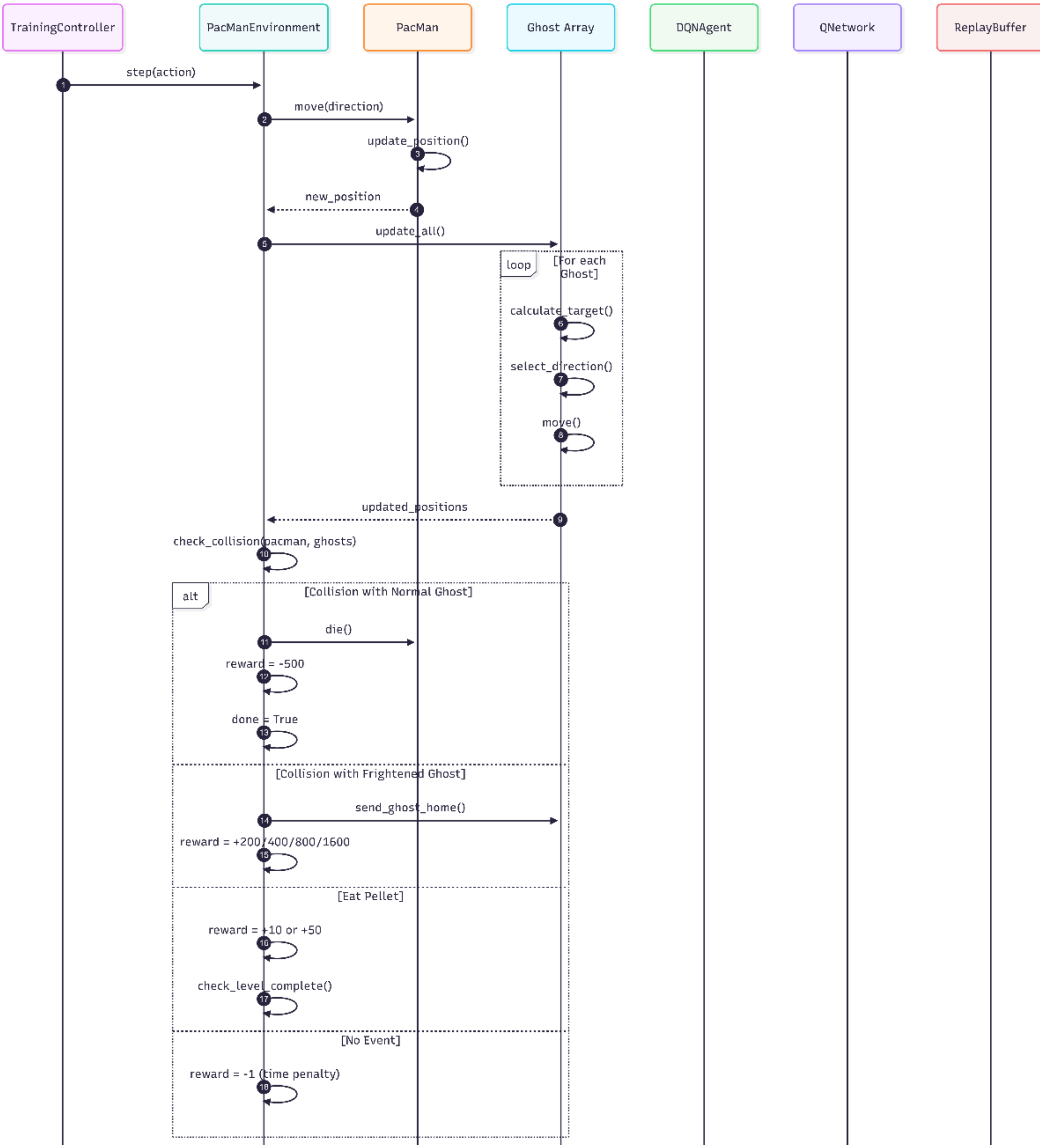
UC-2: Debug Ghost Behavior

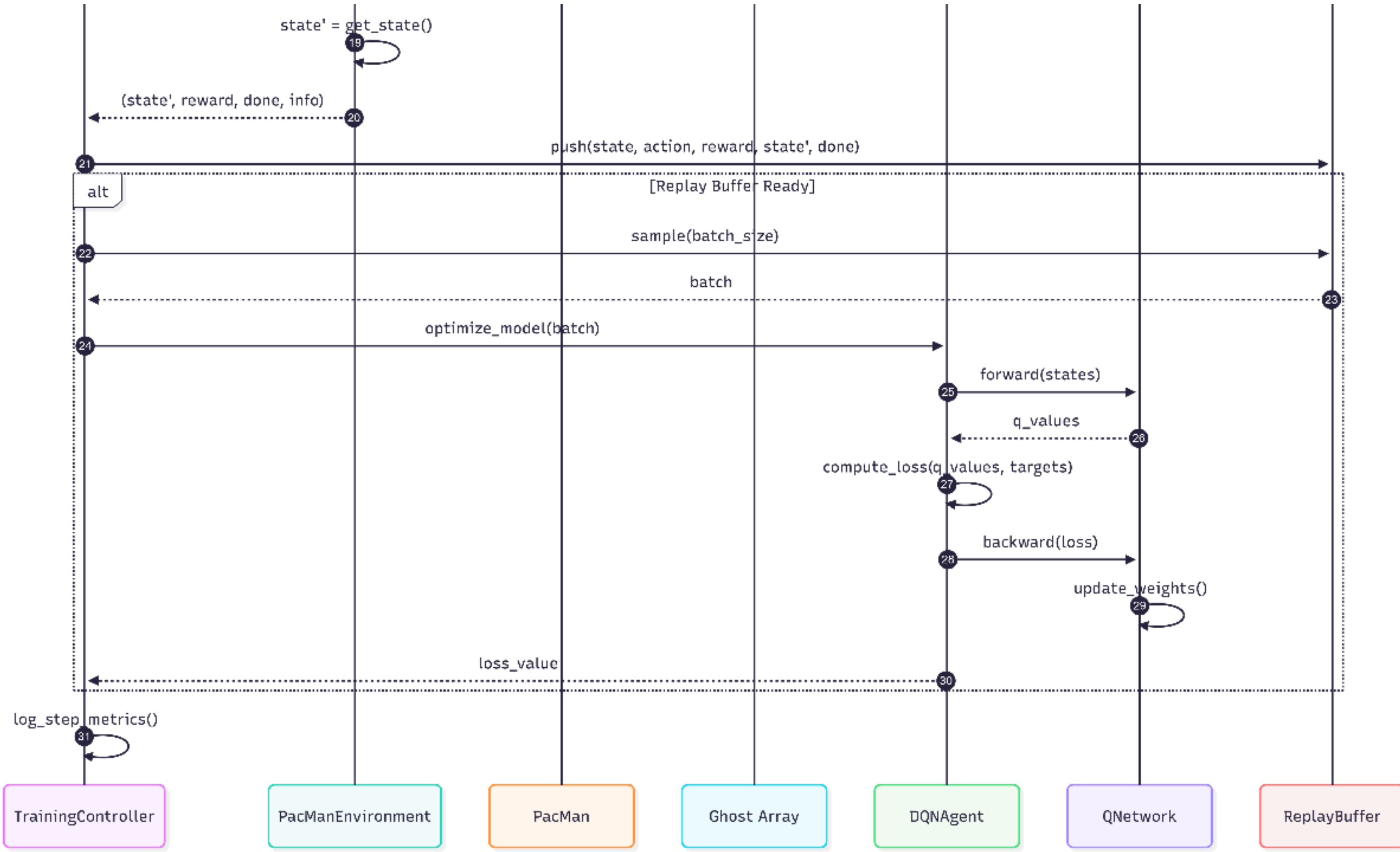


Actors: Developer
Preconditions: Debug mode available, visualization system functional
Postconditions: Ghost behaviors verified, issues documented

b. Sequence Diagram

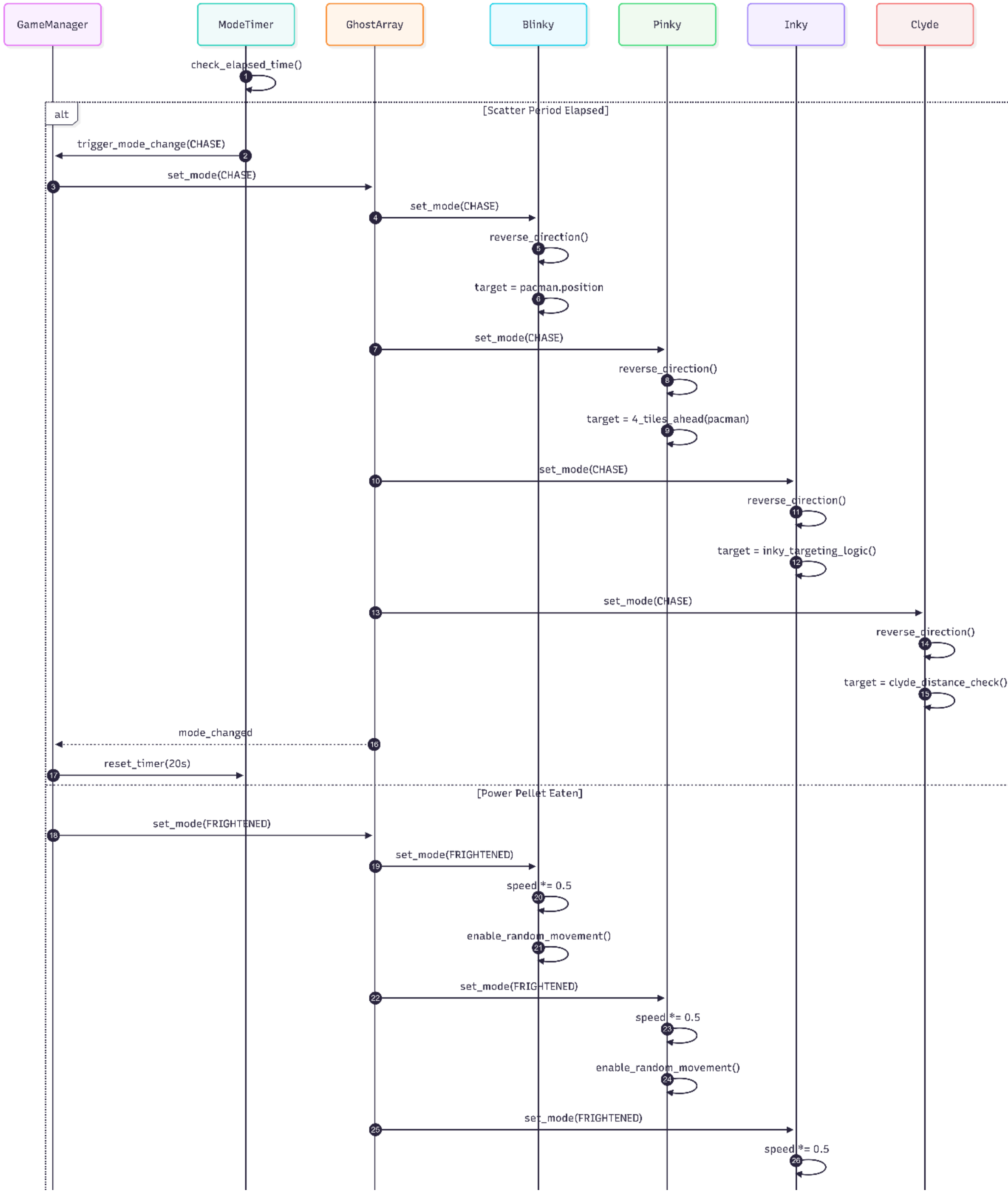
Sequence: Single Training Step

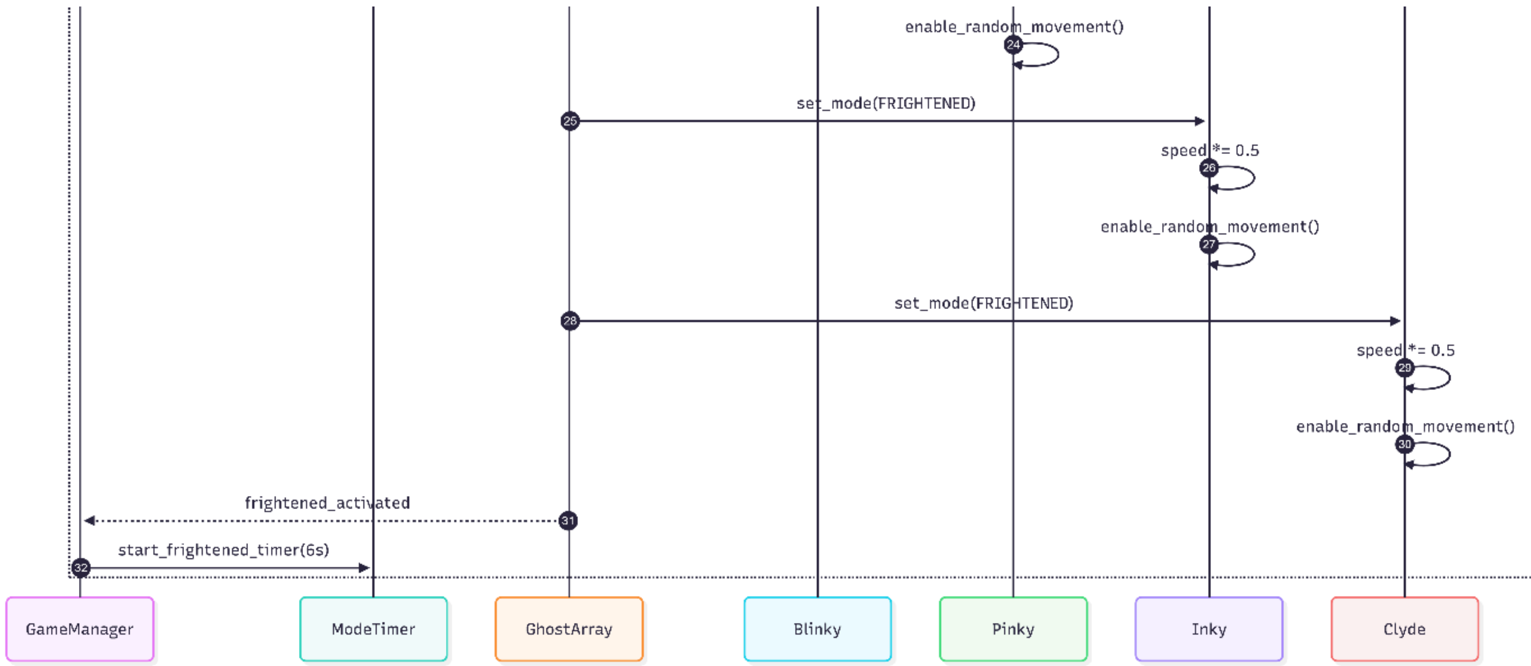




state' = get_state()
(state', reward, done, info)
push(state, action, reward, state', done)
alt
[Replay Buffer Ready]
sample(batch_size)
batch
optimize_model(batch)
forward(states)
q_values
compute_loss(q_values, targets)
backward(loss)
update_weights()
loss_value
log_step_metrics()
TrainingController
PacManEnvironment
PacMan
Ghost Array
DQNAgent
QNetwork
ReplayBuffer

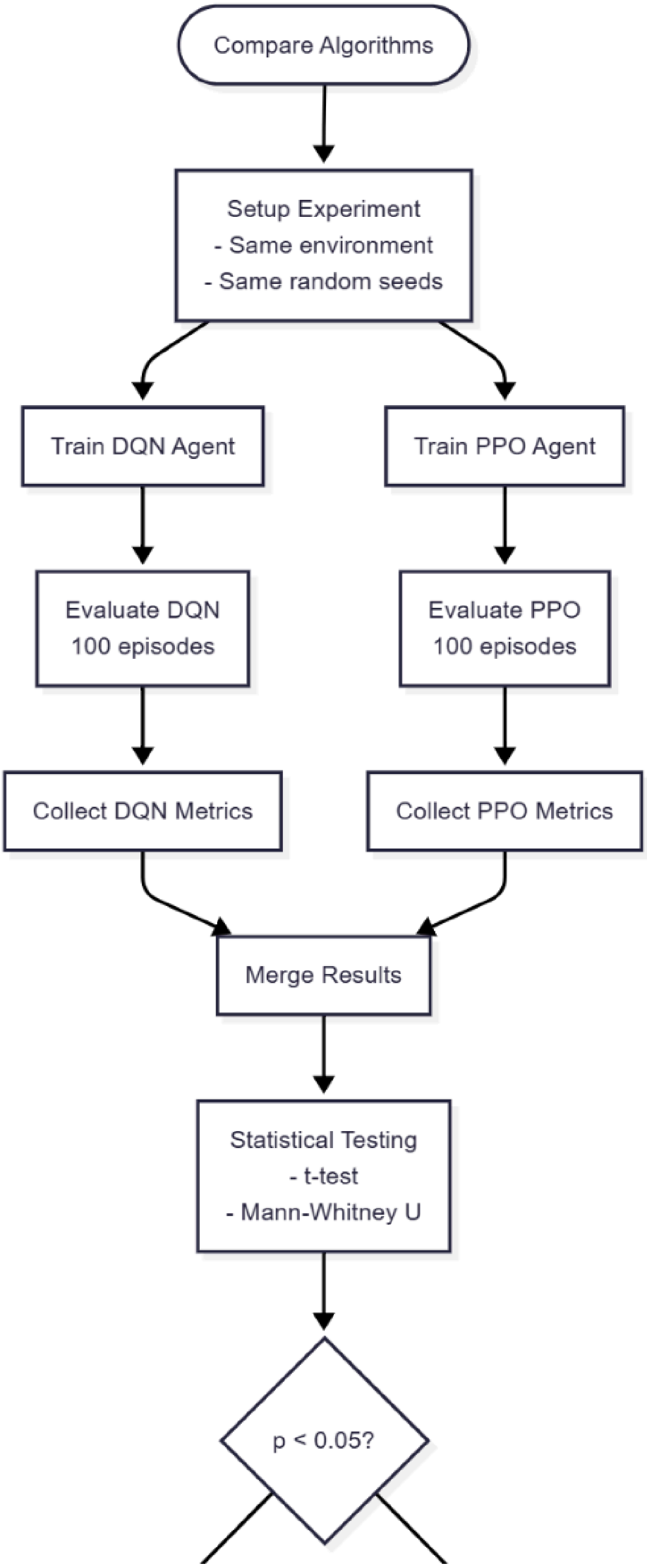
Sequence: Ghost Mode Switching

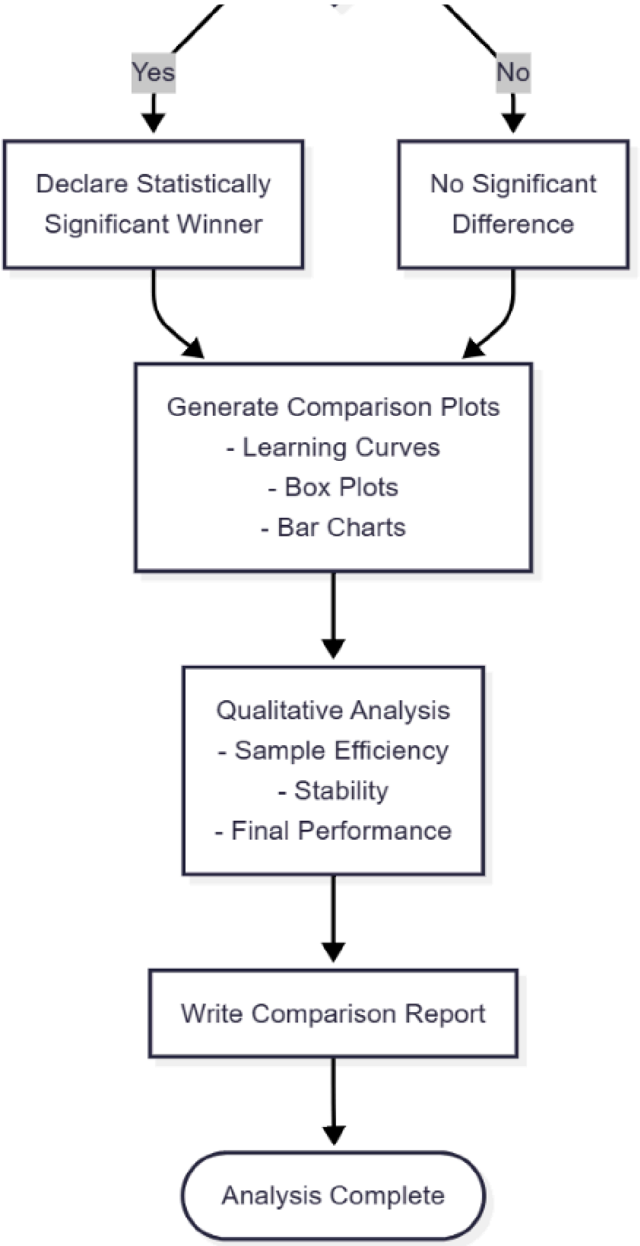



enable_random_movement()
set_mode(FRIGHTENED)
speed *= 0.5
enable_random_movement()
set_mode(FRIGHTENED)
speed *= 0.5
enable_random_movement()
frightened_activated
start_frightened_timer(6s)
GameManager
ModeTimer
GhostArray
Blinky
Pinky
Inky
Clyde

c. Activity Diagram / State / Processes

Activity Diagram: Model Comparison Workflow

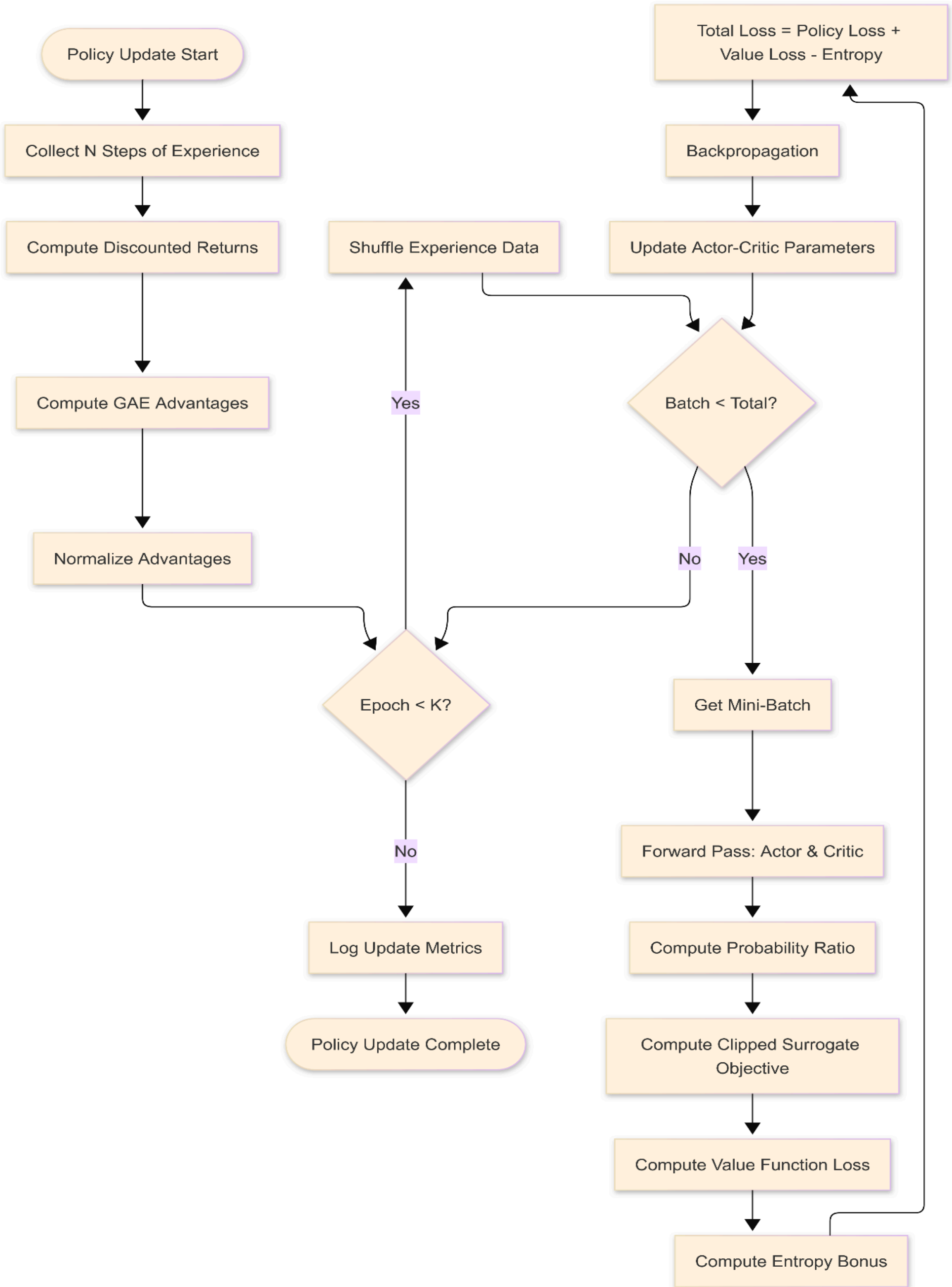




Model Comparison Workflow:

The comparison process trains both DQN and PPO agents independently using identical environment configurations and random seeds to ensure fair evaluation. Each trained agent undergoes 100 evaluation episodes to collect performance metrics (reward, survival time, pellets eaten). Statistical tests (t-test, Mann-Whitney U) determine whether performance differences are statistically significant ($p<0.05$), followed by visualization generation (learning curves, box plots) and qualitative analysis of sample efficiency, training stability, and final performance to produce a comprehensive comparison report.

Process: PPO Policy Update



The PPO policy update process begins by collecting N steps of experience (state-action-reward trajectories) from the environment. It then computes discounted returns and Generalized Advantage Estimation (GAE) to measure how much better each action was compared to the expected value. Finally, it performs K epochs of mini-batch gradient descent, using a clipped surrogate objective to prevent overly large policy updates, while simultaneously updating the value function and adding an entropy bonus to encourage exploration.

6.2.5 Description of Algorithmic Components

a. Deep Q-Network (DQN) Algorithm

DQN learns an optimal action-value function Q(s,a) by iteratively updating a neural network to minimize the temporal difference error between predicted Q-values and target values computed using a separate target network. The agent uses epsilon-greedy exploration, stores experiences in a replay buffer for sample efficiency, and periodically copies weights to the target network for training stability. The core innovation is using experience replay to break correlation between consecutive samples and a fixed target network to prevent divergence during learning.

b. Proximal Policy Optimization (PPO) Algorithm

PPO is an on-policy algorithm that directly optimizes a policy network by collecting trajectories, computing advantages using Generalized Advantage Estimation (GAE), and performing multiple epochs of mini-batch updates. It uses a clipped surrogate objective that limits the size of policy updates, preventing destructively large changes while maintaining sample efficiency. The algorithm simultaneously trains a value function to estimate state values and adds an entropy bonus to encourage exploration.

c. Ghost Pathfinding Algorithm

Each ghost calculates a target tile based on its unique personality: Blinky targets Pac-Man's current tile directly; Pinky targets 4 tiles ahead of Pac-Man's direction (with an overflow bug when facing up that shifts the target 4 tiles up and left); Inky uses complex relational targeting by doubling the vector from Blinky to 2 tiles ahead of Pac-Man; and Clyde targets Pac-Man directly when more than 8 tiles away but retreats to his scatter corner when closer. At each intersection, ghosts select the direction that minimizes Euclidean distance to their target, never reversing except during mode changes, and cannot turn upward in designated red zones near the ghost house.

d. Reward Function Implementation

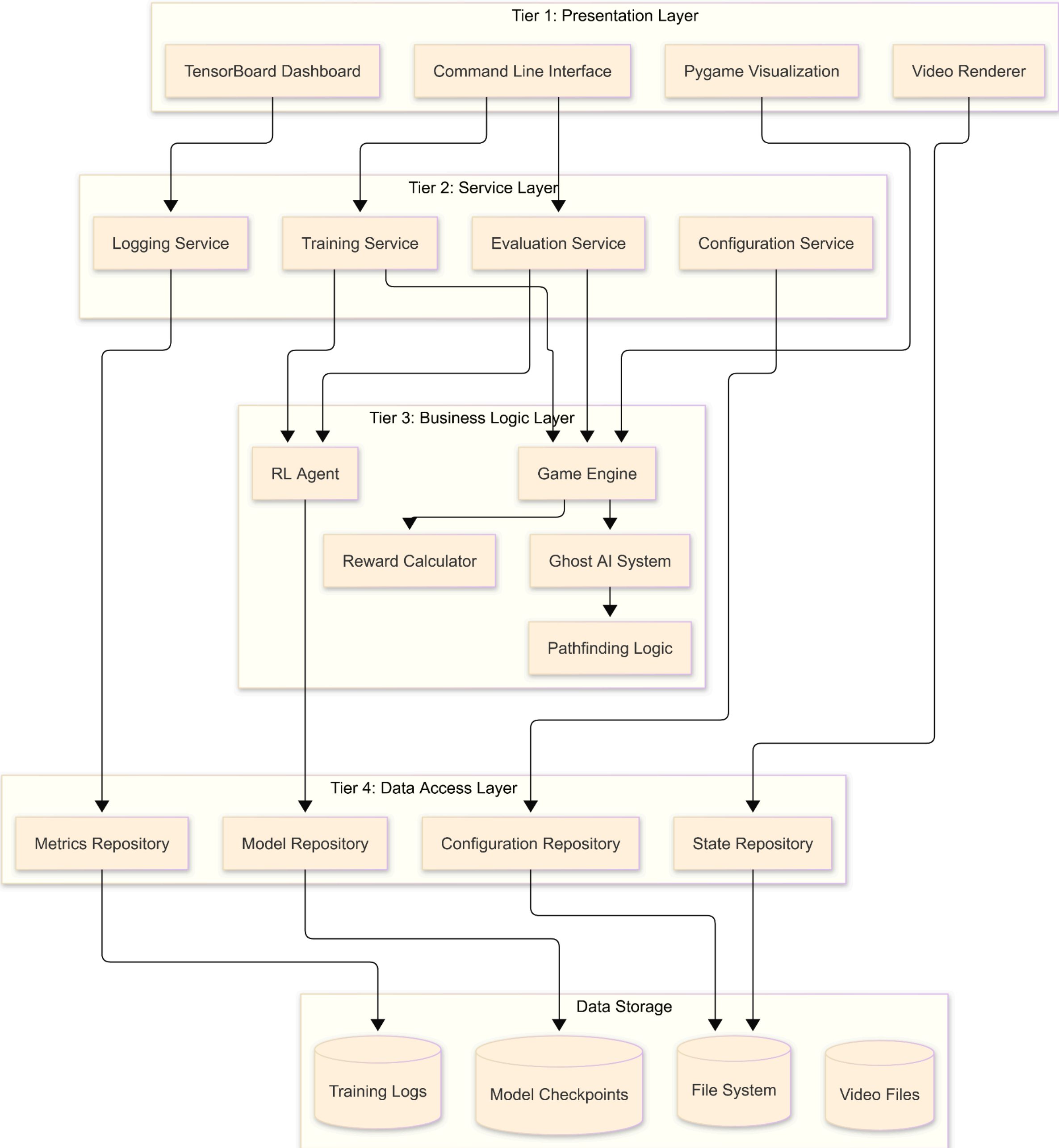
```
Function: calculate_reward(event, game_state)
reward = 0
# Death penalty
if event == GHOST_COLLISION and not ghost.frightened:
    reward = -500
    done = True
    return reward, done
# Pellet rewards
elif event == PELLET_EATEN:
    if pellet.type == REGULAR:
        reward += 10
    elif pellet.type == POWER:
        reward += 50

    # Level completion bonus
    if game_state.pellets_remaining == 0:
        reward += 1000
        done = True
        return reward, done
# Ghost eating rewards (combo multiplier)
elif event == GHOST_EATEN:
    combo = game_state.ghosts_eaten_this_power_pellet
    reward += 200 × (2 ^ combo)  # 200, 400, 800, 1600
# Time penalty (encourage speed)
reward -= 1
```

The reward function provides immediate feedback for all game events to guide the agent's learning. It balances survival (-500 for death), scoring objectives (+10/+50 for pellets, progressive bonuses for ghosts), level completion (+1000), and efficiency (small time penalty to discourage stalling), creating a dense reward signal that encourages both cautious play and strategic risk-taking.

6.2.6 Software Architecture Pattern

N-tier: Data, Logic, Service, Presentation tiers

The system implements a clean 4-tier architecture:



Tier Responsibilities:

Tier 1 - Presentation Layer:

1. User interaction and visualization
2. Input validation
3. Display formatting
4. Real-time rendering
5. Video export

Tier 2 - Service Layer:

6. Orchestration of business logic
7. Transaction management (episodes)
8. Session management
9. Error handling and logging
10. Configuration management

Tier 3 - Business Logic Layer:

11. Core game mechanics
12. AI algorithms
13. Physics simulation
14. Rule enforcement
15. Decision-making logic

Tier 4 - Data Access Layer:

16. CRUD operations for models
17. State persistence
18. Metrics storage
19. Configuration loading
20. File I/O abstraction

Benefits:

21. **Separation of Concerns**: Each tier has distinct responsibilities
22. **Modularity**: Tiers can be developed and tested independently
23. **Maintainability**: Changes in one tier don't cascade to others
24. **Testability**: Each tier can be unit tested in isolation
25. **Scalability**: Tiers can be scaled independently (e.g., parallel training)

7. Low level design

7.1 Pathfinder

```
import math
from typing import Tuple, List, Optional
from enum import Enum

class Direction(Enum):
    UP = (0, -1)
    DOWN = (0, 1)
    LEFT = (-1, 0)
    RIGHT = (1, 0)

    @property
    def opposite(self):
        opposites = {
            Direction.UP: Direction.DOWN,
            Direction.DOWN: Direction.UP,
            Direction.LEFT: Direction.RIGHT,
            Direction.RIGHT: Direction.LEFT
        }
        return opposites[self]

class Pathfinder:
    """
    Implements ghost pathfinding logic for a Pac-Man-style maze.
    Ghosts look one step ahead and choose directions based on
Euclidean distance
    to their target tile.
    """

    # Direction preference order for tie-breaking: up, left,
down, right
    DIRECTION_PREFERENCE = [Direction.UP, Direction.LEFT,
Direction.DOWN, Direction.RIGHT]

    def __init__(self, maze: List[List[int]]):
        """
        Initialize the pathfinder with a maze.

        Args:
            maze: 2D list where 0 = walkable tile, 1 = wall
        """
        self.maze = maze
        self.height = len(maze)
        self.width = len(maze[0]) if maze else 0
```

```
    def is_valid_tile(self, x: int, y: int) -> bool:
        """Check if a tile position is valid and walkable."""
        if x < 0 or x >= self.width or y < 0 or y >= self.height:
            return False
        return self.maze[y][x] == 0

    def get_available_exits(self, x: int, y: int,
current_direction: Direction) -> List[Direction]:
        """
        Get all available exits from a tile, excluding walls and
reverse direction.

        Args:
            x, y: Current tile coordinates
            current_direction: Current direction of travel

        Returns:
            List of valid directions to move
        """
        available = []

        for direction in self.DIRECTION_PREFERENCE:
            # Skip reverse direction
            if direction == current_direction.opposite:
                continue

            # Check if the exit leads to a valid tile
            dx, dy = direction.value
            next_x, next_y = x + dx, y + dy

            if self.is_valid_tile(next_x, next_y):
                available.append(direction)

        return available

    def euclidean_distance(self, x1: int, y1: int, x2: int, y2:
int) -> float:
        """Calculate Euclidean distance between two points."""
        return math.dist((x1, y1), (x2, y2))

    def choose_direction(self, ghost_x: int, ghost_y: int,
                        current_direction: Direction,
                        target_x: int, target_y: int) ->
Direction:
        """
        Choose the next direction for the ghost based on looking
ahead one tile.
```

```
        Args:
            ghost_x, ghost_y: Current ghost position
            current_direction: Ghost's current direction of
travel
            target_x, target_y: Target tile coordinates

        Returns:
            The direction the ghost should take at the next
intersection
        """
        # Look ahead to the next tile along current direction
        dx, dy = current_direction.value
        lookahead_x = ghost_x + dx
        lookahead_y = ghost_y + dy

        # Get available exits from the lookahead tile
        available_exits = self.get_available_exits(lookahead_x,
lookahead_y, current_direction)

        # If only one exit is available, use it
        if len(available_exits) == 1:
            return available_exits[0]

        # If no exits are available (dead end), reverse direction
        if len(available_exits) == 0:
            return current_direction.opposite

        # Multiple exits available - evaluate test tiles
        best_direction = None
        best_distance = float('inf')

        for direction in available_exits:
            # Get test tile position (one tile beyond
intersection in this direction)
            test_dx, test_dy = direction.value
            test_x = lookahead_x + test_dx
            test_y = lookahead_y + test_dy

            # Calculate distance from test tile to target
            distance = self.euclidean_distance(test_x, test_y,
target_x, target_y)

            # Update best direction if this is better
            if distance < best_distance:
                best_distance = distance
                best_direction = direction
            elif distance == best_distance:
```

```
                # Tie-breaking: use direction preference order
                if self.DIRECTION_PREFERENCE.index(direction) <
self.DIRECTION_PREFERENCE.index(best_direction):
                    best_direction = direction

        return best_direction

    def get_next_position(self, x: int, y: int, direction:
Direction) -> Tuple[int, int]:
        """
        Get the next position given current position and
direction.

        Args:
            x, y: Current position
            direction: Direction to move

        Returns:
            Tuple of (next_x, next_y)
        """
        dx, dy = direction.value
        return (x + dx, y + dy)


# Example usage and demonstration
if __name__ == "__main__":
    # Create a simple maze (0 = walkable, 1 = wall)
    maze = [
        [1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 1, 1]
    ]

    pathfinder = Pathfinder(maze)

    # Ghost at position (1, 1), moving right, targeting (5, 5)
    ghost_x, ghost_y = 1, 1
    current_direction = Direction.RIGHT
    target_x, target_y = 5, 5

    next_direction = pathfinder.choose_direction(
        ghost_x, ghost_y, current_direction, target_x, target_y
    )
```

```
    print(f"Ghost at ({ghost_x}, {ghost_y}) moving
{current_direction.name}")
    print(f"Target at ({target_x}, {target_y})")
    print(f"Next direction at intersection:
{next_direction.name}")
```

7.2 ScoreManager

```
class ScoreManager:
     def __init__(self):
          self.score = 0
          self.ghost_eaten_combo = 1

     def reset(self):
          self.score = 0
          self.ghost_eaten_combo = 1

     def add_pellet_score(self):
          self.score += 10

     def add_power_pellet_score(self):
          self.score += 50
          self.ghost_eaten_combo = 1 # The combo should reset
when a power pellet is eaten

     def add_ghost_score(self):
          self.score += (2 ** self.ghost_eaten_combo) * 100    #
200 -> 400 -> 800 -> 1600
          self.ghost_eaten_combo += 1
```

8. Additional Information

Key Algorithms

Deep Q-Network (DQN)

- Uses experience replay to break correlation between consecutive samples
- Employs a separate target network updated every 1,000 steps for stability
- Epsilon-greedy exploration: starts at 1.0, decays to 0.01 over 10,000 steps
- Optimizes mean squared error between predicted and target Q-values

Proximal Policy Optimization (PPO)

- On-policy algorithm with clipped surrogate objective
- Uses Generalized Advantage Estimation (GAE) with λ=0.95
- Performs 4 epochs of updates per data collection
- Balances policy improvement with training stability

Ghost Pathfinding

- Look-ahead algorithm: examines one tile ahead in current direction
- Selects direction minimizing Euclidean distance to target
- Tie-breaking follows priority order: Up → Left → Down → Right
- Never reverses direction except during mode changes

Unique Data Structures

Experience Replay Buffer

- Circular buffer storing (state, action, reward, next_state, done) tuples
- Capacity: 100,000 transitions
- Enables off-policy learning by reusing past experiences
- Uniform random sampling breaks temporal correlations

Pellet Map

- 2D boolean grid (28×36) for efficient pellet presence checking

9. Development Environment Description

Programming Language Selection

Python 3.9+ was chosen for the following reasons:

1. **RL Ecosystem**: Native support for Stable-Baselines3, PyTorch, and Gymnasium
2. **Development Speed**: Rapid prototyping enables quick iteration on reward functions and network architectures
3. **Performance**: NumPy's C-optimized operations and PyTorch's compiled backend provide adequate performance (>1000 FPS headless, 60 FPS rendered)

Framework Choices

Game Framework: Pygame 2.x

- Lightweight 2D graphics with minimal overhead
- Seamless Python integration (no IPC needed)
- Complete game loop control for frame-perfect timing
- Easy headless mode for accelerated training

RL Framework: Stable-Baselines3

- Production-ready DQN and PPO implementations
- Gymnasium-compatible interface
- Excellent documentation and active community
- PyTorch backend for debugging flexibility

Storage Architecture

Local File-Based Storage was selected over cloud storage:

1. **No Network Dependency**: Training runs offline without internet access
2. **Fast I/O**: Local SSD provides low-latency checkpoint saving (<100ms)
3. **Simplicity**: No cloud service configuration or authentication required
4. **Cost**: Zero infrastructure costs for academic project

File Organization:

- Training runs: `/data/training_runs/{run_id}/`
- Model checkpoints: `/data/training_runs/{run_id}/checkpoints/`
- Metrics: CSV files + TensorBoard logs
- Replay buffers: Pickled Python objects for easy serialization

Development Tools

- **Version Control**: Git + GitHub for collaboration
- **Package Management**: UV for development environment
- **Testing**: pytest with >80% code coverage target
- **Code Quality**: Black formatter, mypy type checker
- **Monitoring**: TensorBoard for real-time training visualization

GPU acceleration optional but recommended for faster training.

10. Full Validation Report

```
============================ test session starts ==============================
platform linux -- Python 3.13.7, pytest-9.0.2, pluggy-1.6.0
rootdir: /p/Projects/College/Python/pacman-rl
configfile: pyproject.toml
collected 36 items

test_pathfinder.py .....................                    [ 58%]
test_scoremanager.py ...............                        [100%]

============================= 36 passed in 0.11s ==============================
```

During the validation process, 36 automated tests were collected and executed, covering the core modules of the system:

- **test_pathfinder.py** – Tests for the Pathfinder module, responsible for the ghost navigation logic within the maze.
- **test_scoremanager.py** – Tests for the ScoreManager module, responsible for managing the game's scoring rules and combo mechanics.

Test Results

All tests were completed successfully:

- All test cases passed without failures.
- Total execution time: 0.11 seconds.
- No logical errors, exceptions, or unexpected behaviors were detected.

11. Summary

This document provides comprehensive specifications for developing a Self-Playing Pac-Man AI system using reinforcement learning. The project recreates Pac-Man Level 1 with arcade-authentic mechanics and trains autonomous agents to play optimally against four distinct adversarial ghosts.

Key Documentation Components:

Problem & Approach: The system addresses multi-agent reinforcement learning in real-time adversarial environments through a modular architecture combining game simulation, ghost AI, and RL agents (DQN and PPO).

Requirements: Detailed functional requirements cover game mechanics (maze, pellets, collisions), ghost behaviors (targeting, pathfinding, mode management), RL agent functionality (state representation, training, evaluation), and visualization. Non-functional requirements ensure 60 FPS performance, arcade-accurate timing, and cross-platform compatibility.

Architecture: A 4-tier design separates presentation, service, business logic, and data layers. The core training loop involves the Training Controller orchestrating interactions between the Game Engine, Ghost AI, and RL Agent, with experience stored in replay buffers for learning.

Design Details: Class diagrams define the object-oriented structure with design patterns. Sequence diagrams illustrate training episodes and ghost decision-making. Activity diagrams map training workflows and policy updates.

Implementation Guidance: Low-level designs for the Pathfinder and ScoreManager modules demonstrate the look-ahead algorithm for ghost navigation and progressive scoring mechanics. The technology stack (Python 3.9+, Pygame, Stable-Baselines3) prioritizes development speed and educational value.

Validation: A testing strategy encompasses unit tests (>80% coverage target), integration tests, system tests, and performance benchmarks to verify game mechanics accuracy, ghost behavior authenticity, and RL agent learning progression.

This specification serves as the authoritative blueprint for development, providing clear requirements, architectural decisions, and technical details necessary for successful implementation and evaluation of the Self-Playing Pac-Man AI system.