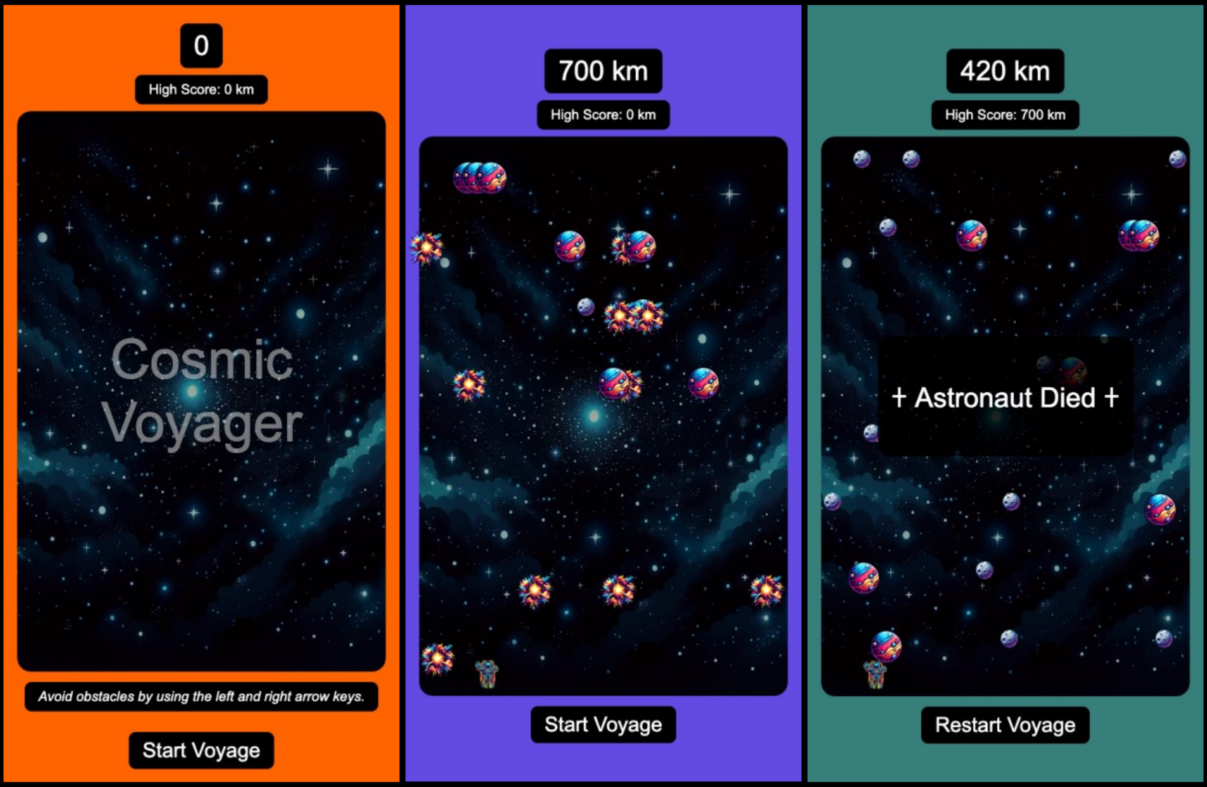
From Concept to Deployment: Building a Web App with an AI-Powered Backend

Introduction, Motivation & Objective

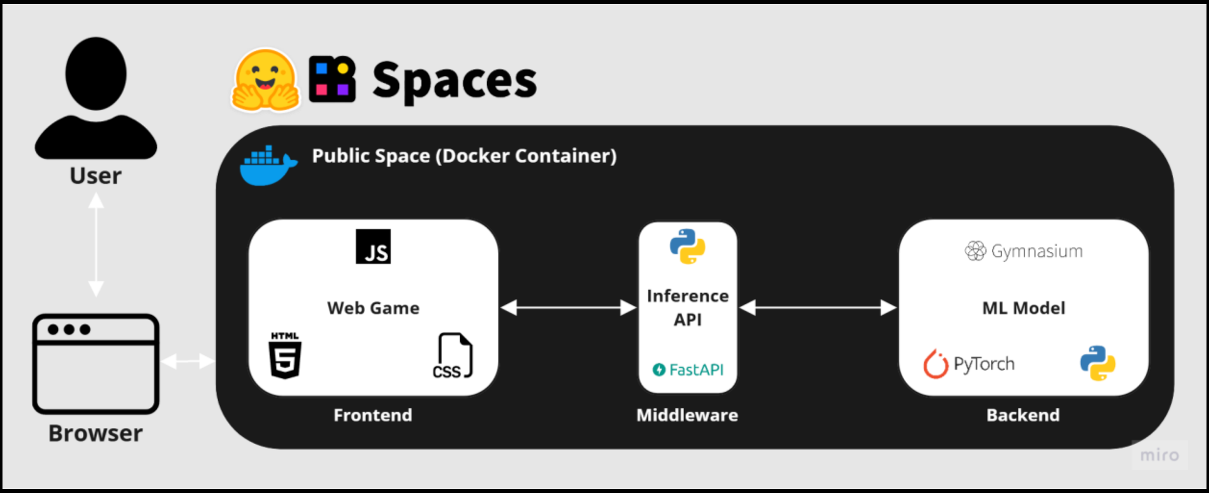
After creating **Cosmic Voyager**, my retro-inspired web game designed to offer a thrilling and accessible gameplay experience, I felt compelled to explore its potential beyond traditional player interactions. My next objective was to develop an **AI mode**, enabling an intelligent agent to autonomously play the game. This presented a fascinating challenge: building a system that could navigate the game environment, dodge obstacles, and maximize its score—all without human intervention. If you want to learn more about the development, feel free to read the separate story on that right [here]. See the original web interface below.



To achieve this, I turned to **Deep Reinforcement Learning (DRL)**, a cutting-edge area of artificial intelligence known for its ability to train agents to solve complex problems in simulated environments. DRL has been successfully applied in various AI agents to master virtual games, from classic arcade challenges to modern 3D simulations, making it an ideal approach for this project. Not only did this endeavor align with my fascination

for DRL, but it also provided an opportunity to gain hands-on experience in applying this powerful technology to a practical, interactive system.
The goal was ambitious but exciting: extend **Cosmic Voyager** into a full-fledged machine learning application to showcase the capabilities of reinforcement learning in creating intelligent, autonomous systems. This project promised to deepen my understanding of AI, enhance my technical skill set, and merge my passion for web development with state-of-the-art machine learning techniques.
In the following, I will discuss the technical details. Have fun!

Application Design

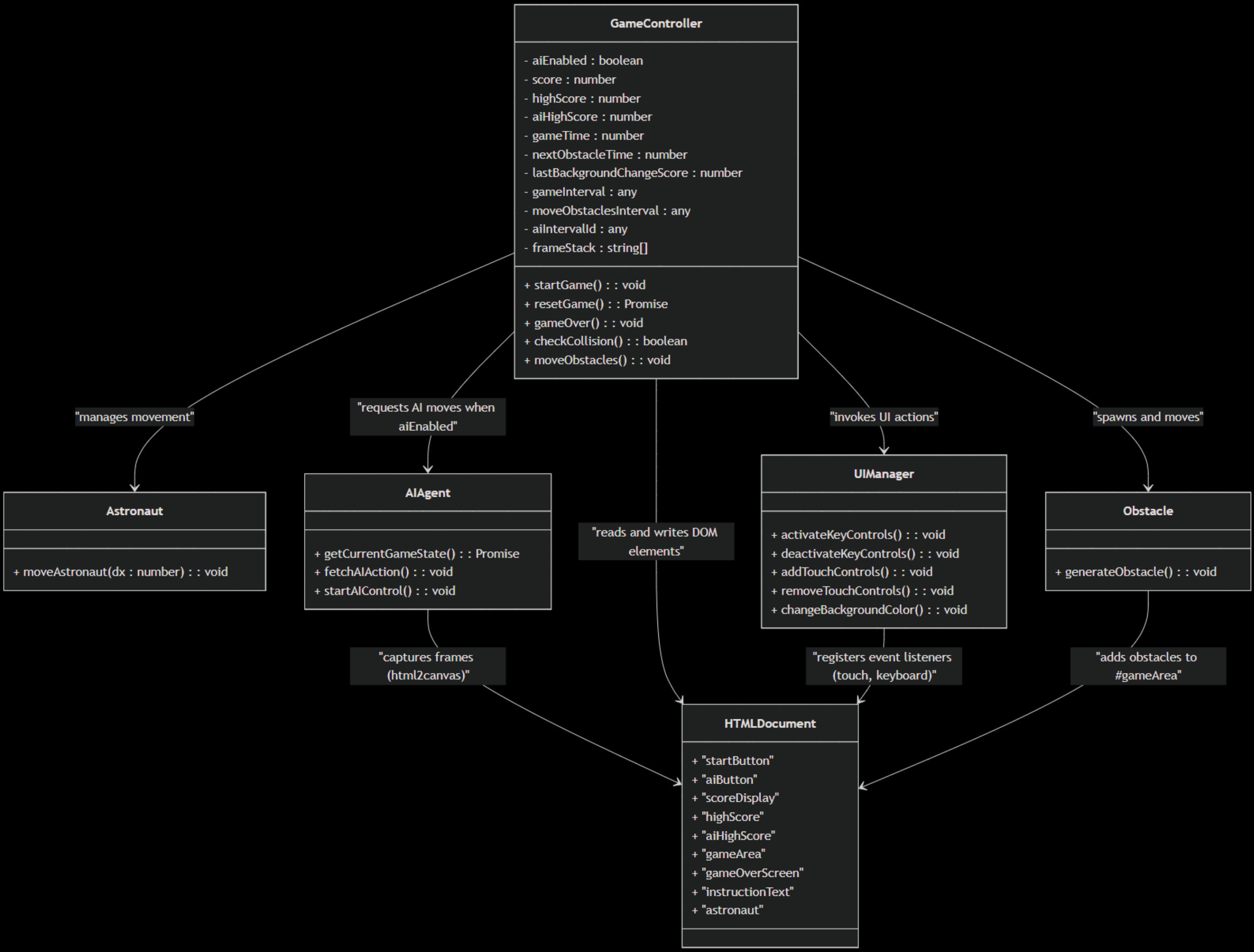


Building upon the foundation of Cosmic Voyager, I transformed the web game into a fully functional machine learning application by integrating a deep reinforcement learning (DRL) model. The software architecture of **Cosmic Voyager** is structured around a clear modular design comprising three primary components: the frontend, middleware, and backend, all deployed within a single Docker container on the Hugging Face Spaces platform.

Frontend (Web Game)

The web game serves as the frontend, developed with HTML, CSS, and JavaScript. It operates entirely within the user's browser, rendering the game environment, managing user interactions, and visualizing gameplay. Players can access the game through a public URL and interact with it via any web browser, ensuring broad accessibility.
The web game is structured around a single-page application model where HTML, CSS, and JavaScript collectively handle rendering, user interaction, and game logic. The core JavaScript code listens for user events—like keyboard presses or screen touches—to move the on-screen astronaut and uses setInterval loops to manage the continuous flow of obstacle generation, collision detection, and score updates. All visual elements (the astronaut, dynamically added obstacles, and the score displays) are part of the DOM,

which is styled via an external CSS file. When AI mode is activated, the client periodically captures the current game state via screenshots of the game area, shrinks and converts the canvas to grayscale, and then accumulates these frames in a buffer. Once the buffer is filled, the client sends the stacked frames via a JSON POST request to a backend endpoint (e.g., “/predict”) for inference. The model in the backend responds with an action (move left, move right, or no movement), which the front-end applies to the astronaut’s position. This loosely couples the client-side game engine with the server-side machine learning model. State management, such as current score, high scores, and AI toggling, is maintained in JavaScript variables and displayed on-screen in real time. The architecture thus allows seamless interplay between manual control and AI-driven actions, all within a single webpage powered by event-driven JavaScript, interval-based game loops, and external AI inference calls.
See the following UML class diagram to learn more about the architecture of the web game.


GameController
- aiEnabled : boolean
- score : number
- highScore : number
- aiHighScore : number
- gameTime : number
- nextObstacleTime : number
- lastBackgroundChangeScore : number
- gameInterval : any
- moveObstaclesInterval : any
- aiIntervalId : any
- frameStack : string[]
+ startGame() : : void
+ resetGame() : : Promise
+ gameOver() : : void
+ checkCollision() : : boolean
+ moveObstacles() : : void
"manages movement"
"requests AI moves when aiEnabled"
"invokes UI actions"
"spawns and moves"
Astronaut
+ moveAstronaut(dx : number) : : void
AIAgent
+ getCurrentGameState() : : Promise
+ fetchAIAction() : : void
+ startAIControl() : : void
"reads and writes DOM elements"
UIManager
+ activateKeyControls() : : void
+ deactivateKeyControls() : : void
+ addTouchControls() : : void
+ removeTouchControls() : : void
+ changeBackgroundColor() : : void
Obstacle
+ generateObstacle() : : void
"captures frames (html2canvas)"
"registers event listeners (touch, keyboard)"
"adds obstacles to #gameArea"
HTMLDocument
+ "startButton"
+ "aiButton"
+ "scoreDisplay"
+ "highScore"
+ "aiHighScore"
+ "gameArea"
+ "gameOverScreen"
+ "instructionText"
+ "astronaut"

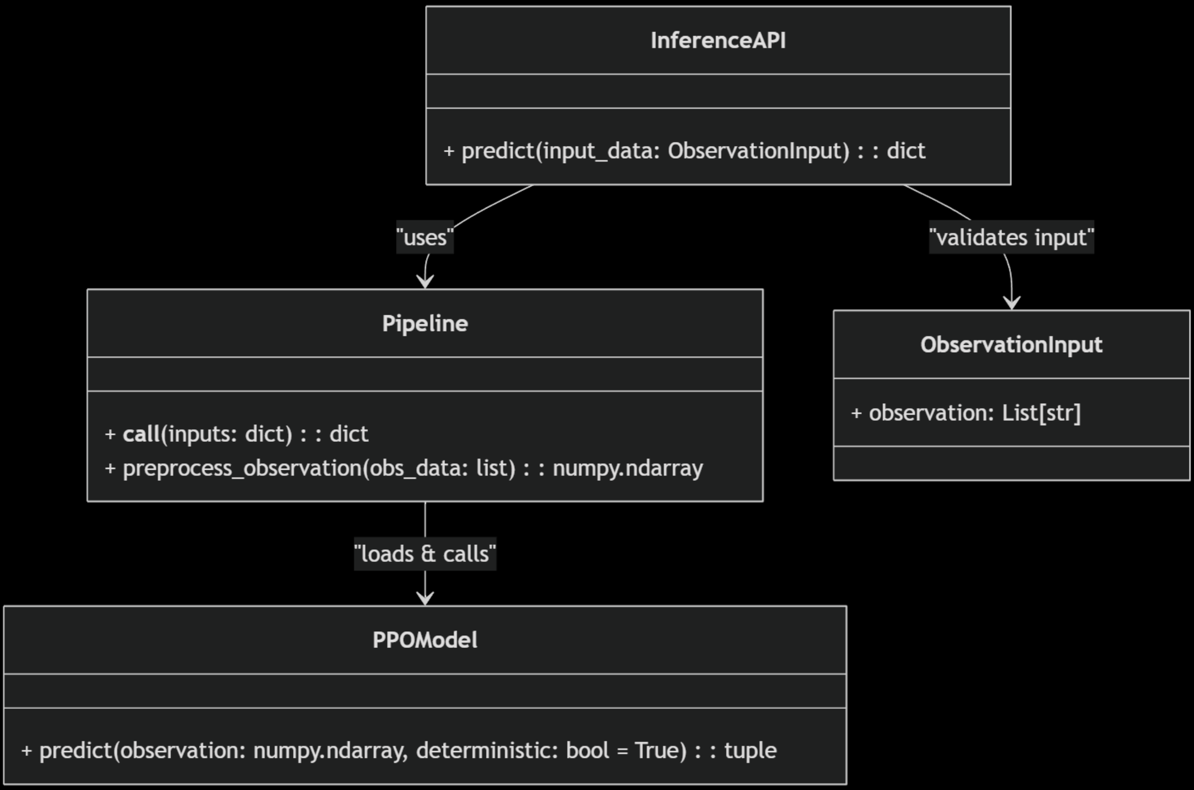
Explanation of Components

1. **GameController**
   Acts as the central “engine” of the game. It controls the main loops (using setInterval), handles the score and time increments, checks for collisions, spawns obstacles, and coordinates transitions between game states (start, reset, game over).
2. **Astronaut**
   Encapsulates logic for player movement (or AI-driven movement). In the code, this is partly done by calling moveAstronaut() to adjust the astronaut’s position within the game area.
3. **Obstacle**
   Handles the creation of different obstacle types (asteroids, planets, supernovas, black holes), their properties (images, size, position), and updates needed (e.g., growth or oscillation). Although in the code these are created on the fly, conceptually we can think of them as a class or a responsibility.
4. **UIManager**
   Manages user interface concerns such as keyboard activation, touch controls for mobile, updating background color, and general display/hide operations. In the code, these responsibilities appear scattered across helper functions but can be grouped conceptually as UI concerns.
5. **AIAgent**
   Handles the AI mode by capturing the game state (with html2canvas), stacking frames, sending a request (fetch('/predict')) to the backend, and applying the returned action (move left, right, or do nothing). This is the bridge between the local JavaScript game and a remote AI inference service.
6. **HTMLDocument**
   Represents the DOM elements—buttons, game area, score displays, and so on. While not literally a “class” in the code, we can show the DOM or external environment in UML to demonstrate the dependencies and references. This clarifies that these UI elements are manipulated by the controllers and managers.

Middleware (Inference API)

The Inference API (application programming interface) functions as the intermediary between the frontend and backend. Developed in Python using the FastAPI framework, it employs a server that loads a trained deep reinforcement learning model at application startup and receives incoming data in the form of multiple image frames. Each frame is transformed into a standardized format and appended into a collection of stacked arrays, which the model then processes to produce a corresponding action output. The API returns the action as a succinct JSON response, incorporating basic validation to ensure

correct data input. In addition, it serves static content from a designated directory and uses appropriate middleware to manage cross-origin requests, making it suitable for seamless integration into real-time interactive systems.
See the following UML class diagram to learn more about the architecture of the Inference API.



InferenceAPI
+ predict(input_data: ObservationInput) : : dict
"uses"
"validates input"
Pipeline
+ **call**(inputs: dict) : : dict
+ preprocess_observation(obs_data: list) : : numpy.ndarray
ObservationInput
+ observation: List[str]
"loads & calls"
PPOModel
+ predict(observation: numpy.ndarray, deterministic: bool = True) : : tuple

Explanation of Components

1. InferenceAPI (the FastAPI-based application) defines the /predict endpoint, which calls the Pipeline instance.
2. Pipeline loads a pre-trained PPOModel (DRL model) and uses it for inference once it has preprocessed the incoming frames.
3. ObservationInput is a data model used for validating the request body.

Backend (ML Model)

The ML/DRL model constitutes the backend, providing the AI agent's decision-making capabilities. Trained using DRL algorithms and OpenAI's Gymnasium framework, the model evaluates the current game state and determines optimal actions to navigate the game's challenges. This integration enables the AI agent to play the game autonomously, showcasing the practical application of reinforcement learning in interactive environments.

Implementation of the Web Game as Frontend

When I developed **Cosmic Voyager**, my goal was to create an engaging and accessible gaming experience that could not only serve as a platform for honing my web development skills but also provide a foundation for building a machine learning application. I wanted the game to evoke nostalgia with its retro-inspired 2D design while ensuring it was easily accessible across devices.

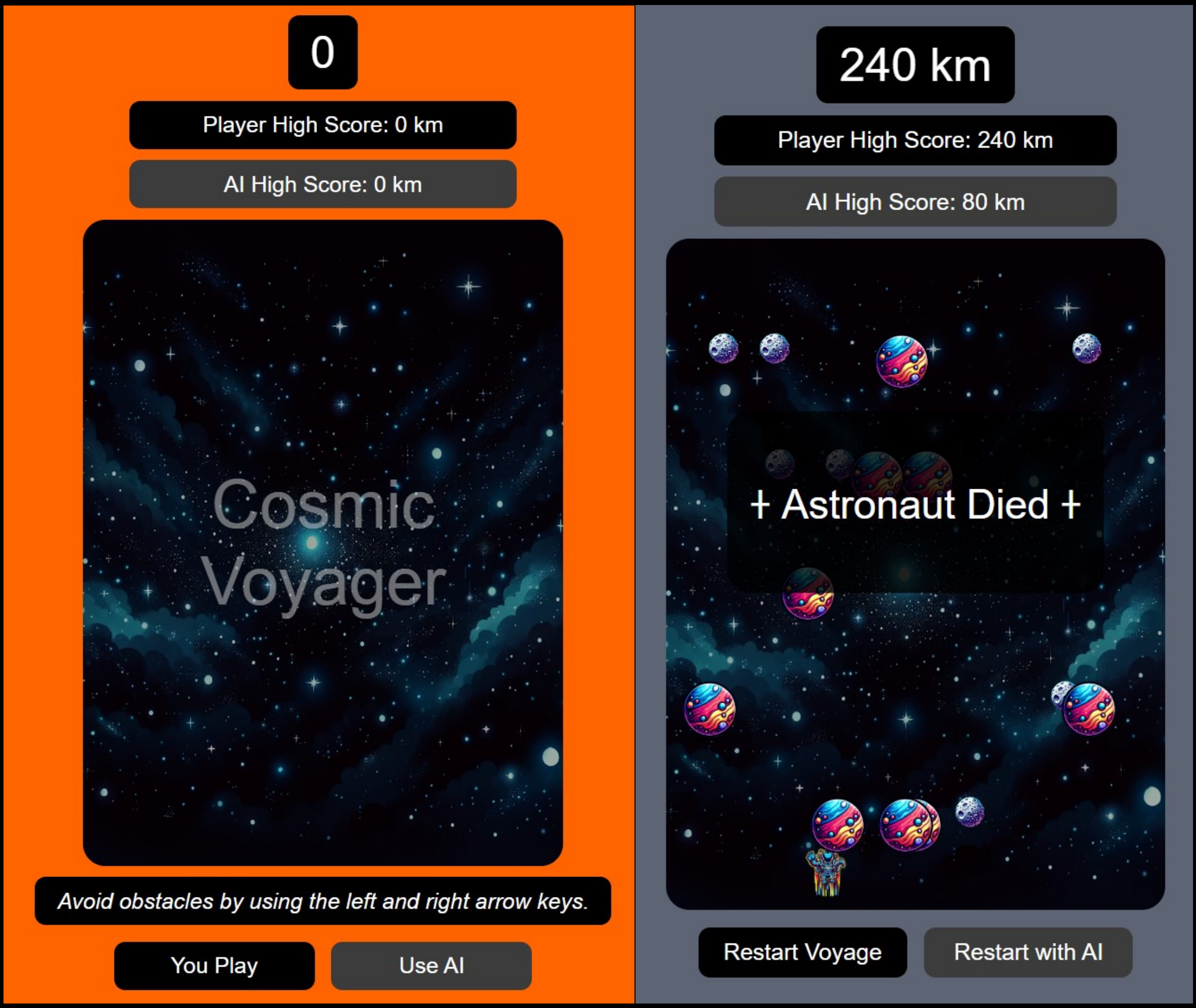
The gameplay centers around an astronaut navigating through an obstacle-filled space environment. I designed the controls to be intuitive, allowing players to use a keyboard or touch inputs to guide the astronaut past asteroids, planets, and other cosmic hazards. To keep the game dynamic and challenging, I incorporated features like increasing obstacle complexity and randomized patterns, ensuring that no two play sessions felt the same. I added a scoring system to motivate players, along with visual elements like background changes to enhance the sense of progression.

I built the game using HTML, CSS, and JavaScript. HTML provided the structure, CSS handled the styling, and JavaScript brought the game to life with logic for obstacle generation, collision detection, and adaptive controls. For the visual assets, I used DALL-E to create the icons and backgrounds, then optimized them to maintain a cohesive and polished cosmic theme.

One of the biggest challenges I faced was ensuring the game worked seamlessly across various devices. I had to make careful adjustments to the layout and controls to create a truly responsive design. Another challenge was balancing the game's difficulty. Through iterative testing, I fine-tuned the obstacle speed and frequency to strike the perfect balance between being fair and challenging.

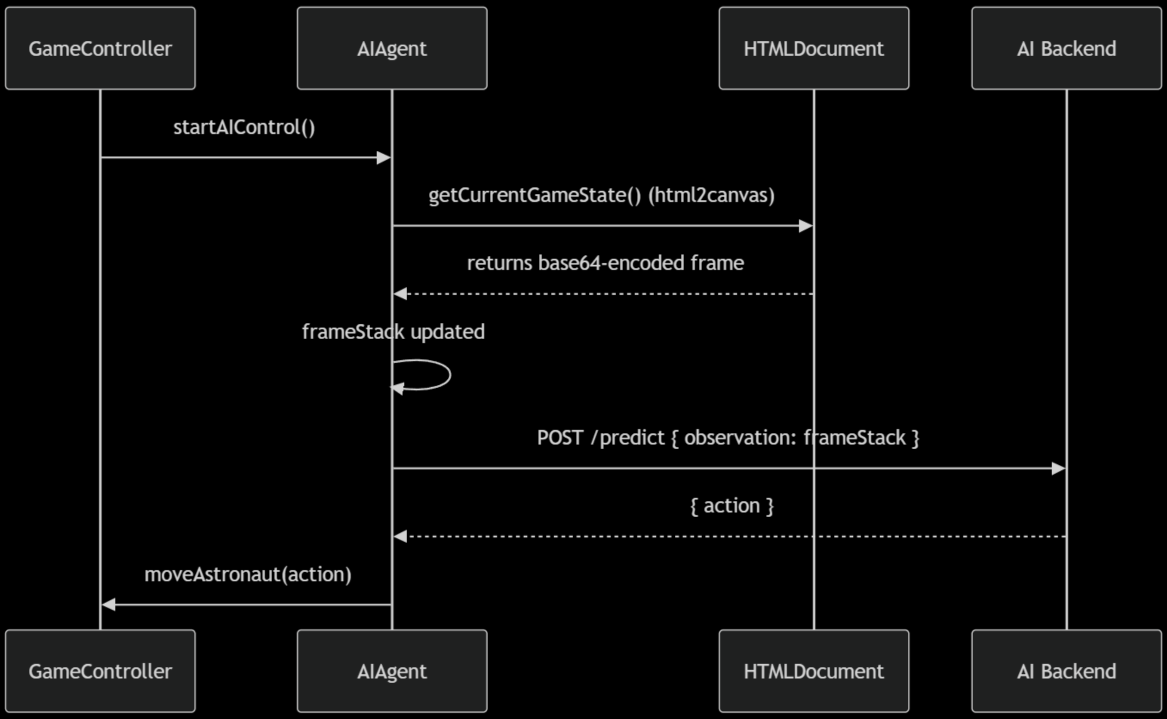
The final product was a functional, visually appealing, and universally accessible game. It not only demonstrated the potential of combining creativity with modern development tools but also laid the groundwork for integrating a machine learning agent. Cosmic Voyager became more than just a game — it became a platform for exploring the exciting intersection of web development and artificial intelligence.

To enable the integration of AI mode into my web game, I made several enhancements to the frontend. One of the key changes I introduced was a new button that allows users to activate AI mode directly from the interface. This addition made it possible to switch seamlessly between manual and AI gameplay modes, providing a flexible and intuitive experience. I also added a separate high-score display specifically for the AI agent, ensuring that its performance was tracked independently from the player's score. This not only highlighted the AI's capabilities but also gave me a clear way to measure its success. See the updated web interface below.



I extended the game logic in the frontend to handle the functionality required for the AI mode. I implemented a process to capture the current game state and send it to the backend via an API, where the AI model would decide on the best action to take. These decisions were then applied in real-time, enabling the AI agent to autonomously navigate the game and strive for high scores. I adapted the reset and start mechanisms to ensure they worked smoothly across both manual and AI modes, making the transition between them seamless for users.
Through these adaptations, I transformed the game into a dual-mode system that supports both interactive and automated gameplay. This evolution not only improved the frontend but also turned the game into a platform where I could explore reinforcement learning in action, while comparing human and AI performance in a fun and engaging way.

The following sequence diagram shows the cyclical nature of AI inference: capturing frames from the DOM, batching them, sending them to the server, receiving an action, and finally applying that action in the GameController.



Development of the Machine Learning Model as Backend

Problem Description and Objective

Training an AI agent to navigate the Cosmic Voyager environment posed an exciting challenge. The objective was to create a system that could play the game autonomously, continuously improving its performance over time. The agent needed to learn the underlying mechanics of the game, such as avoiding obstacles and positioning itself effectively within the game area. This required it to develop strategies that maximize survival and optimize its score.

The problem was defined by the dynamic nature of the environment. The agent faced a barrage of obstacles, increasing in complexity and frequency as the game progressed. To succeed, it had to make quick decisions in real-time, leveraging the information from its surroundings to predict the best course of action. The ultimate goal was for the agent to learn these mechanics and strategies on its own through continuous interaction with the game, gradually getting better with each iteration and hence maximising the score.

Solution

To solve the problem of training an AI agent to master Cosmic Voyager, I turned to Deep Reinforcement Learning (DRL), a cutting-edge approach that has been widely used for training agents in virtual game environments. DRL combines reinforcement learning (RL) with deep neural networks to enable agents to learn complex tasks by interacting with an environment. In this paradigm, an agent observes the environment's state, selects an action, and receives feedback in the form of a reward. This feedback guides the agent in learning to take better actions over time. The goal is for the agent to maximize cumulative rewards, which represent successful behavior in the environment.

In DRL, a neural network approximates either the agent’s policy (a strategy mapping states to actions) or the value function (estimating future rewards from a state). Unlike traditional supervised learning, where the model learns from labeled data, DRL operates through trial and error. The agent explores the environment, experimenting with actions to discover strategies that yield the highest rewards. Over many iterations, the agent converges toward optimal behavior by balancing exploration of new strategies and exploitation of known successful ones.

This approach has been highly effective in applications such as robotics, video games, and autonomous systems, where agents operate in dynamic environments requiring real-time decision-making and adaptation.

General Approach to Training an Agent

1. **Defining the Virtual Environment**
   The first step is to create a virtual environment that serves as the training ground for the agent. This environment must provide observations representing its state, accept actions from the agent, and compute rewards based on the agent’s performance. The environment encapsulates the rules, dynamics, and objectives the agent must learn to navigate.

2. **Designing the Reward Function**
   A well-defined reward function is critical for guiding the agent’s learning process. Rewards incentivize desired behaviors and penalize undesirable ones, effectively shaping the agent's exploration of strategies. The design of the reward function directly impacts the quality and efficiency of the learning process.

3. **Selecting a Training Algorithm**
   Once the environment is ready, a suitable DRL algorithm is chosen to train the agent. These algorithms, such as policy-based or value-based methods, determine how the agent updates its policy or value function based on its interactions with the environment. The choice of algorithm depends on the problem's complexity, the nature of the environment, and the desired outcomes.

4. **Setting Up the Training Process**
   With the algorithm selected, the training process begins. This involves initializing the agent with a neural network architecture capable of processing observations and generating actions. Hyperparameters such as learning rate, discount factor, and exploration parameters are configured to optimize the learning process.

5. **Training the Agent**
   During training, the agent interacts with the environment across multiple episodes, each consisting of a sequence of state transitions, actions, and

rewards. The agent's policy is updated iteratively, using the feedback received to improve its decision-making ability. Initially, the agent's actions are largely exploratory, but as training progresses, it starts exploiting learned strategies to achieve better performance.

6. **Evaluating and Optimizing the Model**
   After initial training, the agent's performance is evaluated against the defined objectives. This step often reveals areas for improvement, prompting adjustments to the training setup. Modifications can include fine-tuning hyperparameters, refining the reward function, or updating the neural network architecture. The process is repeated iteratively to enhance the agent's performance.

7. **Obtaining the Trained Model**
   Once the agent consistently performs well in the environment, the training process concludes, resulting in a trained model. This model encodes the agent's optimal policy, enabling it to make decisions and act autonomously within the environment.

This structured approach ensures that the agent learns effectively and adapts to the challenges of the virtual environment, culminating in a model that demonstrates intelligent and autonomous behavior.

Environment and Reward Setup

To enable the AI agent to train effectively in the Cosmic Voyager game, I created a custom environment tailored to the game's mechanics and training requirements. This environment was implemented using Python and several key libraries, each serving a specific purpose in the setup.

Frameworks and Libraries

1. **Gymnasium**: A framework for creating and interacting with reinforcement learning environments. It provided the structure for defining action spaces, observation spaces, and step/reset methods.

2. **Selenium**: A browser automation tool that enabled direct interaction with the web-based game. It was used to simulate user actions, retrieve game states, and control the gameplay dynamically.

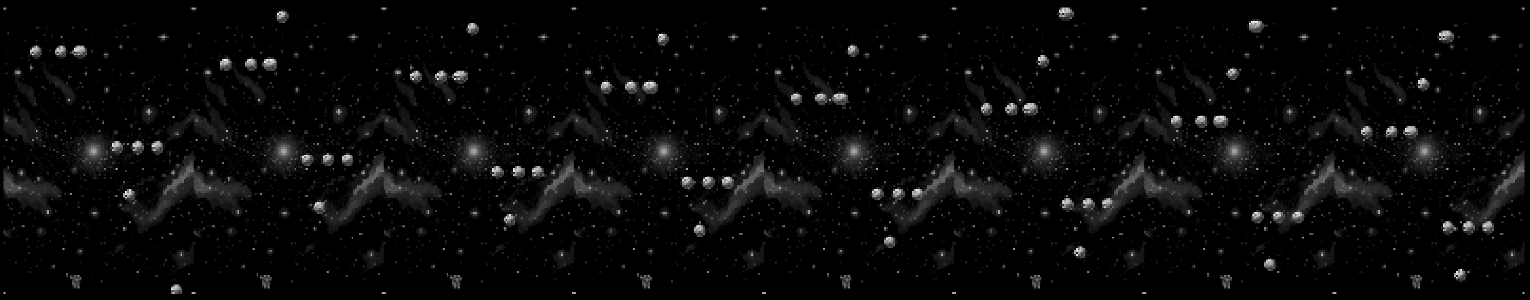
3. **OpenCV**: A library for image processing, used to preprocess the game visuals by converting screenshots into grayscale and resizing them to fixed dimensions.

4. **NumPy**: Essential for handling numerical data, particularly for managing observations and computations within the environment.

5. **Collections (Deque)**: Utilized for stacking frames to provide the agent with temporal context, critical for understanding dynamic environments.

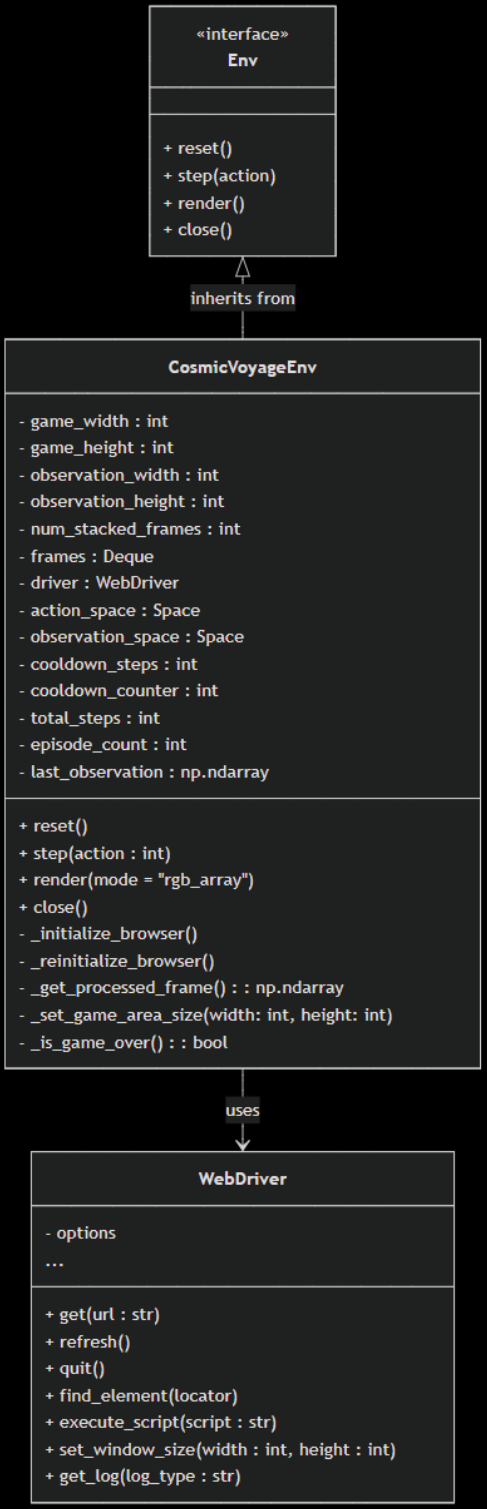
Custom Environment Setup

The environment was designed to simulate the Cosmic Voyager game and enable the agent to interact with it autonomously. The game itself runs in a browser, and the environment uses Selenium to initialize and control the browser session. The game area is adjusted to a fixed size for consistency, and the agent interacts with the game through simulated keyboard inputs, corresponding to discrete actions: no action, move left, or move right.
Observations are generated by capturing screenshots of the game area, preprocessing them using OpenCV, and normalizing pixel values to create a format suitable for training. To provide the agent with temporal context, multiple consecutive frames (8) are stacked together, forming a three-dimensional observation array (example below).



The reward function was designed to encourage survival and penalize collisions. The agent earns a small reward for each step it survives, with additional incentives for achieving high scores. Collisions with obstacles result in negative rewards, signaling the agent to avoid such actions in the future. This reward structure guides the agent in learning optimal strategies, such as positioning itself effectively to dodge incoming hazards.
Through this setup, the environment bridges the gap between the web-based game and the reinforcement learning model, enabling the AI agent to train in a simulated yet highly interactive virtual setting.

See the following UML class diagram to learn more about the architecture of the script setting up the custom environment.

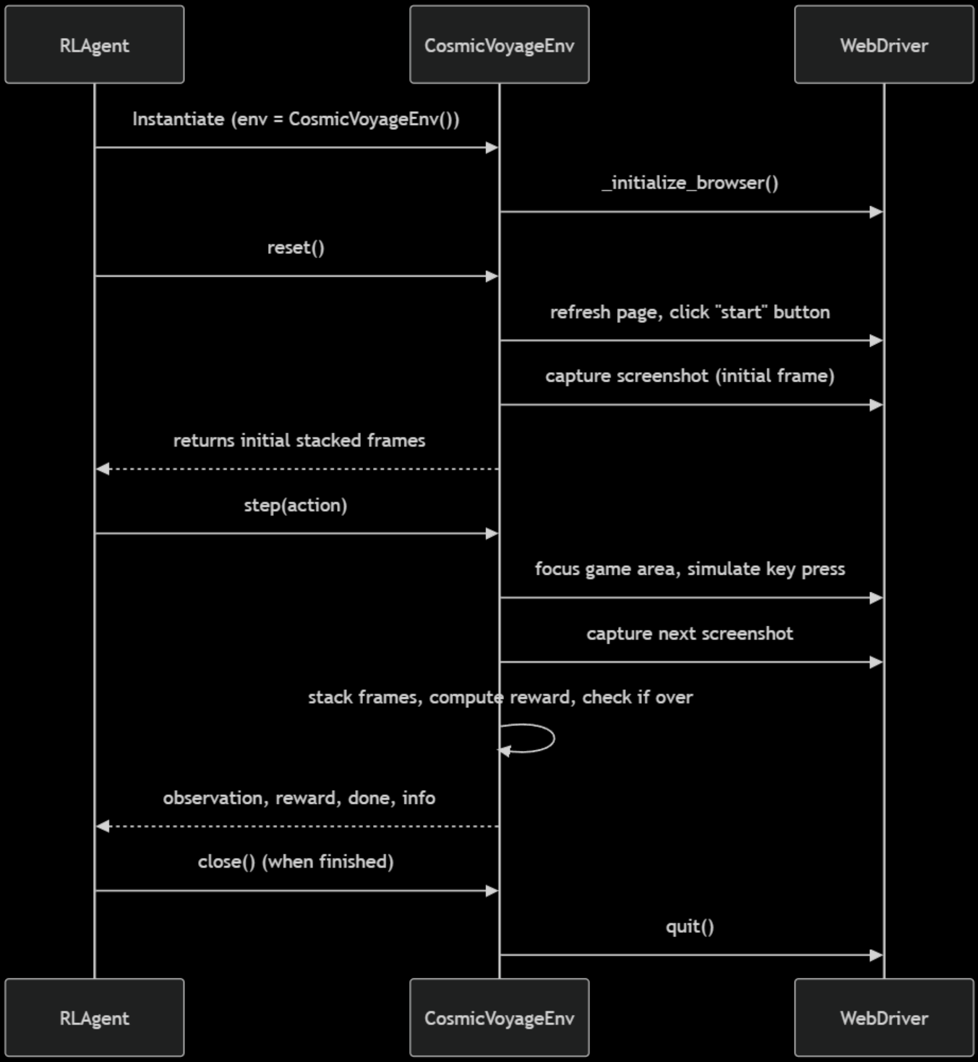


Explanation

- **Env** is the interface or abstract parent class from Gym that CosmicVoyageEnv implements (i.e., gym.Env in code).
- **CosmicVoyageEnv** manages game parameters, browser interaction, and environment logic (resetting, stepping, rendering).
- **WebDriver** (Selenium's Chrome driver in this case) is used internally by CosmicVoyageEnv to load, refresh, and interact with the web game.

The following sequence diagram explains the functionality of the environment script:

1. The **RLAgent** (reinforcement learning code) creates an instance of **CosmicVoyageEnv**, triggering the browser initialization.
2. The agent calls **reset()**, causing the environment to refresh the game page, click the "start" button, and build the initial stacked observation from multiple frames.
3. On **step(action)**, the environment focuses the game area, sends the relevant keyboard action (left, right, or none), takes a screenshot, stacks frames, and calculates the resulting reward.
4. Finally, **close()** properly shuts down the browser and cleans up resources.


RLAgent
CosmicVoyageEnv
WebDriver
Instantiate (env = CosmicVoyageEnv())
_initialize_browser()
reset()
refresh page, click "start" button
capture screenshot (initial frame)
returns initial stacked frames
step(action)
focus game area, simulate key press
capture next screenshot
stack frames, compute reward, check if over
observation, reward, done, info
close() (when finished)
quit()
RLAgent
CosmicVoyageEnv
WebDriver

Training Setup

To train the AI agent for Cosmic Voyager, I carefully designed the training setup to maximize learning efficiency and ensure the agent's performance steadily improved. This involved structuring the training process, defining hyperparameters, incorporating

logging and monitoring tools, and implementing a system for evaluation and continuous optimization.
The training setup starts by initializing the Cosmic Voyager environment and defining an evaluation environment identical in configuration. These environments provide the agent with observations and allow it to interact with the game, learning through trial and error. A policy is initialized with a neural network architecture tailored to process the game's observations effectively.
The training process involves specifying hyperparameters such as learning rate, batch size, and discount factor, which govern the learning dynamics. The Proximal Policy Optimization (PPO) algorithm was chosen. PPO is an advanced reinforcement learning algorithm that optimizes an agent's strategy by balancing exploration and learning stability. It uses a clipping mechanism to limit policy updates, preventing instability and ensuring steady improvement. By approximating the policy and value function with a neural network, PPO efficiently trains agents in complex environments, making it a robust choice for Cosmic Voyager.
During training, the agent interacts with the environment, observing states, taking actions, and receiving rewards. A callback system logs metrics, records videos of the agent's gameplay, and saves model checkpoints at regular intervals. These checkpoints preserve the model's progress, providing recovery points in case of interruptions.

Logging and Evaluation

Detailed logs are generated during training, capturing performance metrics such as average episode reward and length, and entropy loss. Visualizations of the agent's observations and gameplay videos are recorded for analysis. These insights help evaluate the agent's progress and identify areas for improvement. Everything is logged to WandB for streamlined visualization and comparison of different training runs.

Frameworks and Tools

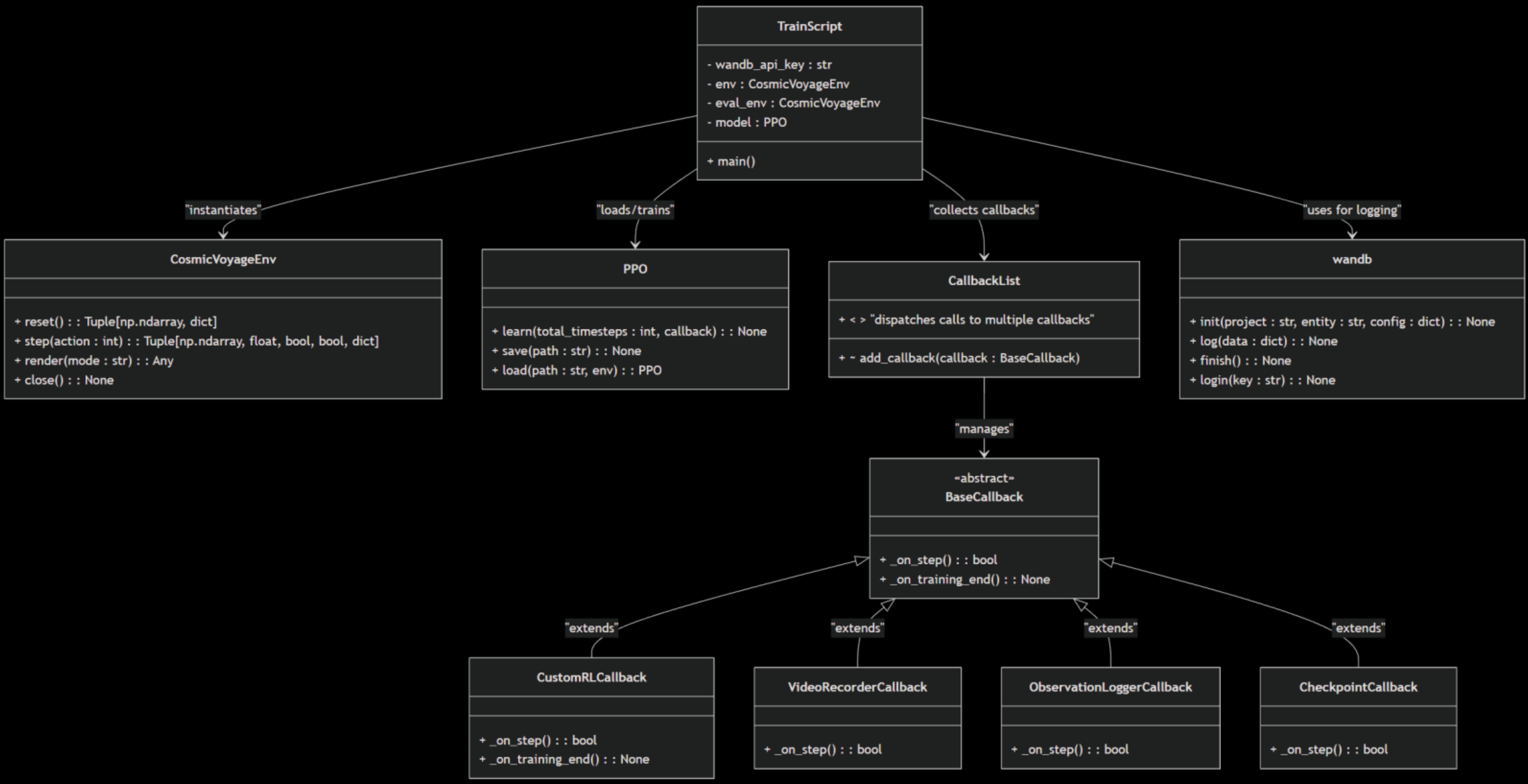
1. **PyTorch**: A flexible and efficient deep learning framework used for training the neural network model that powers the AI agent.

2. **Stable-Baselines3**: A reinforcement learning library providing implementations of advanced algorithms, including the Proximal Policy Optimization (PPO) algorithm used in this project.

3. **WandB (Weights and Biases)**: A tool for experiment tracking and logging, enabling detailed monitoring of the training process.

4. **NumPy**: Essential for numerical computations throughout the training process.

See the following UML class diagram to learn more about the architecture of the script executing the deep reinforcement learning.


TrainScript
- wandb_api_key : str
- env : CosmicVoyageEnv
- eval_env : CosmicVoyageEnv
- model : PPO
+ main()
"instantiates"
"loads/trains"
"collects callbacks"
"uses for logging"
CosmicVoyageEnv
+ reset() : : Tuple[np.ndarray, dict]
+ step(action : int) : : Tuple[np.ndarray, float, bool, bool, dict]
+ render(mode : str) : : Any
+ close() : : None
PPO
+ learn(total_timesteps : int, callback) : : None
+ save(path : str) : : None
+ load(path : str, env) : : PPO
CallbackList
+ < > "dispatches calls to multiple callbacks"
+ ~ add_callback(callback : BaseCallback)
wandb
+ init(project : str, entity : str, config : dict) : : None
+ log(data : dict) : : None
+ finish() : : None
+ login(key : str) : : None
"manages"
«abstract»
BaseCallback
+ _on_step() : : bool
+ _on_training_end() : : None
"extends"
"extends"
"extends"
"extends"
CustomRLCallback
+ _on_step() : : bool
+ _on_training_end() : : None
VideoRecorderCallback
+ _on_step() : : bool
ObservationLoggerCallback
+ _on_step() : : bool
CheckpointCallback
+ _on_step() : : bool

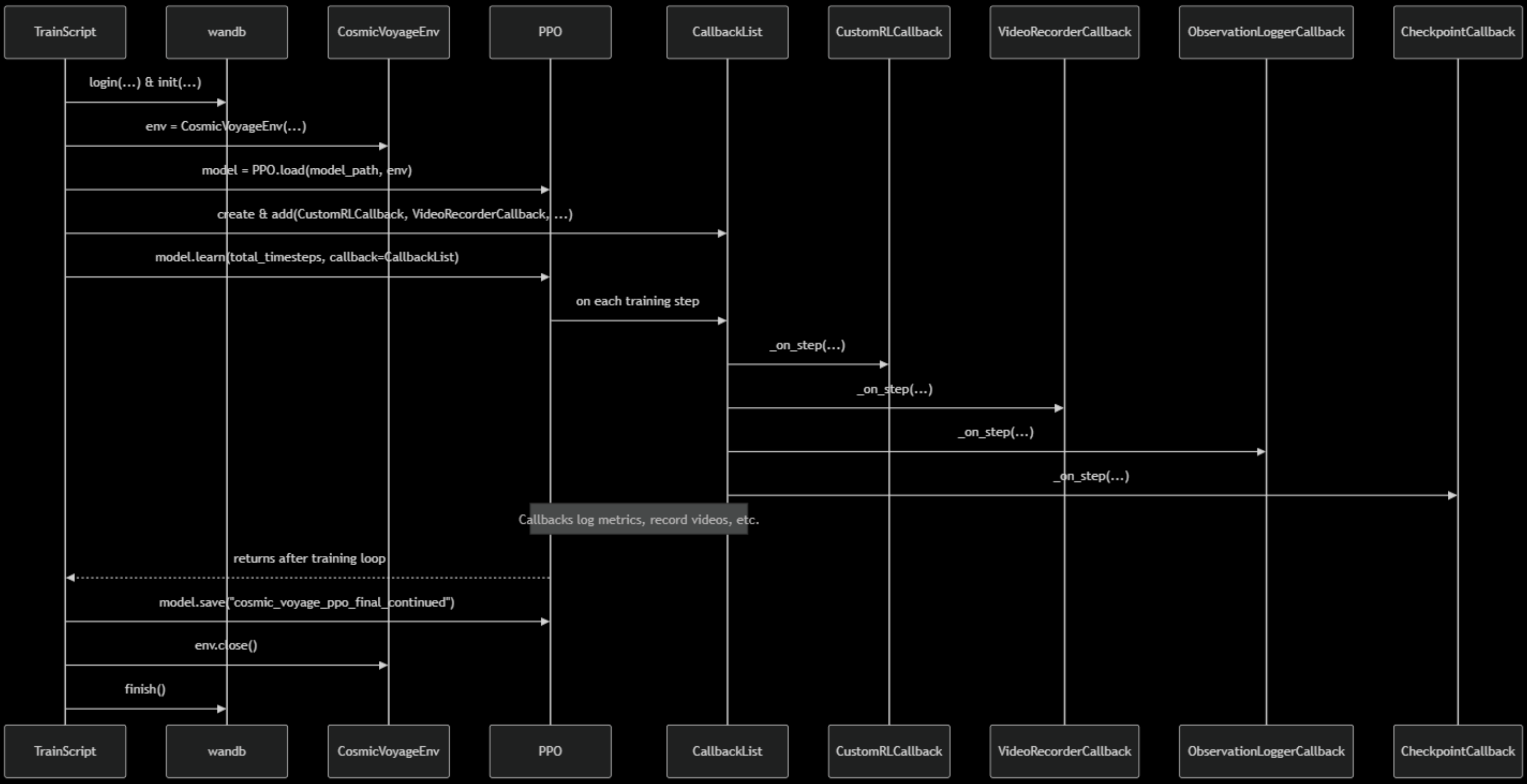
Explanation

- **TrainScript** represents the main script that sets up the environment, loads the model, configures callbacks, and triggers training.
- **CosmicVoyageEnv** is the custom Gym environment for the web game.
- **PPO** is the Stable Baselines3 model class used for training.
- **BaseCallback** is the abstract base class from Stable Baselines3, with various custom callbacks extending it.
- **CallbackList** holds and dispatches calls to each callback.
- **wandb** is used for experiment tracking and logging.

The following sequence diagram explains the functionality of the training script:

1. **TrainScript** logs into Weights & Biases (wandb) and initializes the project tracking.
2. It instantiates **CosmicVoyageEnv** (and a separate evaluation environment), then loads a pre-trained **PPO** model.
3. **CallbackList** is created and populated with various callbacks—**CustomRLCallback**, **VideoRecorderCallback**, **ObservationLoggerCallback**, and **CheckpointCallback**.
4. **TrainScript** calls model.learn(...), triggering the training loop.
5. During each step, the **PPO** triggers the **CallbackList**, which calls each individual callback in turn, logging metrics and recording artifacts.

6. After the training completes, **TrainScript** saves the updated model, closes the environment, and finishes the W&B session.


TrainScript
wandb
CosmicVoyageEnv
PPO
CallbackList
CustomRLCallback
VideoRecorderCallback
ObservationLoggerCallback
CheckpointCallback
login(...) & init(...)
env = CosmicVoyageEnv(...)
model = PPO.load(model_path, env)
create & add(CustomRLCallback, VideoRecorderCallback, ...)
model.learn(total_timesteps, callback=CallbackList)
on each training step
_on_step(...)
_on_step(...)
_on_step(...)
_on_step(...)
Callbacks log metrics, record videos, etc.
returns after training loop
model.save("cosmic_voyage_ppo_final_continued")
env.close()
finish()

Optimization and Final Model

Training is an iterative process, where the initial setup is refined based on evaluation results. Hyperparameters are adjusted, the reward structure may be revisited, and the neural network architecture can be optimized to improve learning. Once the agent achieves consistent performance, the training concludes, resulting in a trained model capable of autonomous gameplay.

By structuring the training setup in this way, I ensured that the agent not only learned the mechanics of Cosmic Voyager but also optimized its strategies to excel in the dynamic game environment.

Model Training & Optimization

The training phase for the Cosmic Voyager AI agent involved an iterative process of experimentation, stabilization, and optimization. My goal was to refine the agent's learning process by tuning hyperparameters, adjusting the network architecture, and addressing challenges such as interrupted training sessions. Ultimately, I built on previous training runs to achieve an agent capable of performing effectively in the dynamic game environment.

Hyperparameter Tuning

Hyperparameter tuning is a critical step in training reinforcement learning models, as it directly impacts the efficiency and quality of the learning process. I experimented with key hyperparameters, including:

- **Learning Rate**: Controlled how quickly the model updated its weights. I used smaller values (e.g., 0.00005) for stability and larger values (e.g., 0.001) for faster updates.
- **Batch Size**: Determined the number of samples processed in a single update. Larger batches (e.g., 512) improved stability, while smaller ones (e.g., 64) allowed faster but noisier updates.
- **Clip Range**: Regulated the magnitude of policy updates to stabilize learning.
- **Entropy Coefficient**: Balanced exploration (trying new strategies) and exploitation (refining known strategies).
- **GAE Lambda and Discount Factor (Gamma)**: Adjusted the agent's sensitivity to immediate vs. long-term rewards.
- **Entropy Coefficient**: Adjusted to balance exploration and exploitation.

Although I manually tuned these hyperparameters, AutoML could have streamlined the process. AutoML (Automated Machine Learning) leverages algorithms to automatically optimize hyperparameters, architectures, and training strategies, reducing the need for manual experimentation. Prominent frameworks like Optuna, Ray Tune, and Google AutoML are designed for this purpose, making them valuable tools for reinforcement learning projects.

Initial Test Runs and Stabilizing the Training Process

I began with several short test runs to explore different neural network architectures and hyperparameters. The network complexity was varied by adjusting the number of hidden layers and nodes per layer, impacting the agent's capacity to learn complex behaviors. For example, I tested configurations with smaller architectures (e.g., fewer layers and nodes) for quicker training but less capacity, and larger architectures for potentially better long-term performance. This experimentation can be observed in the gray, blue and dark green graphs with the lowest reward and episode length in the plots of the training metrics below.
Training interruptions caused by PC shutdowns, updates, and errors were a significant early challenge. To address this, I implemented mechanisms to save trained models at regular intervals. This allowed me to resume training from the last saved state, avoiding the need to start from scratch. However, it is important to note that continuing with a pretrained model constrains certain choices: the training algorithm and network architecture cannot be changed, as the pretrained model builds on the preexisting training runs.

CUDA and Training Performance

I conducted the training on my local NVIDIA RTX 3060 GPU with 12GB VRAM using CUDA (Compute Unified Device Architecture). CUDA is a parallel computing platform developed by NVIDIA that allows deep learning frameworks to leverage GPU acceleration, significantly speeding up computations. The RTX 3060 provided sufficient computational power for the task, enabling smooth training even with larger network architectures and longer episodes.

Logging and Evaluation

Throughout the training, I logged key metrics:

- **Average Episode Length**: Indicated how long the agent could survive in the game.
- **Average Reward per Episode**: Reflected the agent's overall performance and progression.
- **Entropy Loss**: Monitored the balance between exploration and exploitation.

Additionally, I recorded gameplay videos at certain intervals to visually evaluate the agent's strategies. These visual inspections provided insights into how the agent adapted to challenges and whether further adjustments were needed. You can download the training videos right [here] to see how the agent performed at the beginning, in between and at the end of training.

Long-Term Training and Optimization

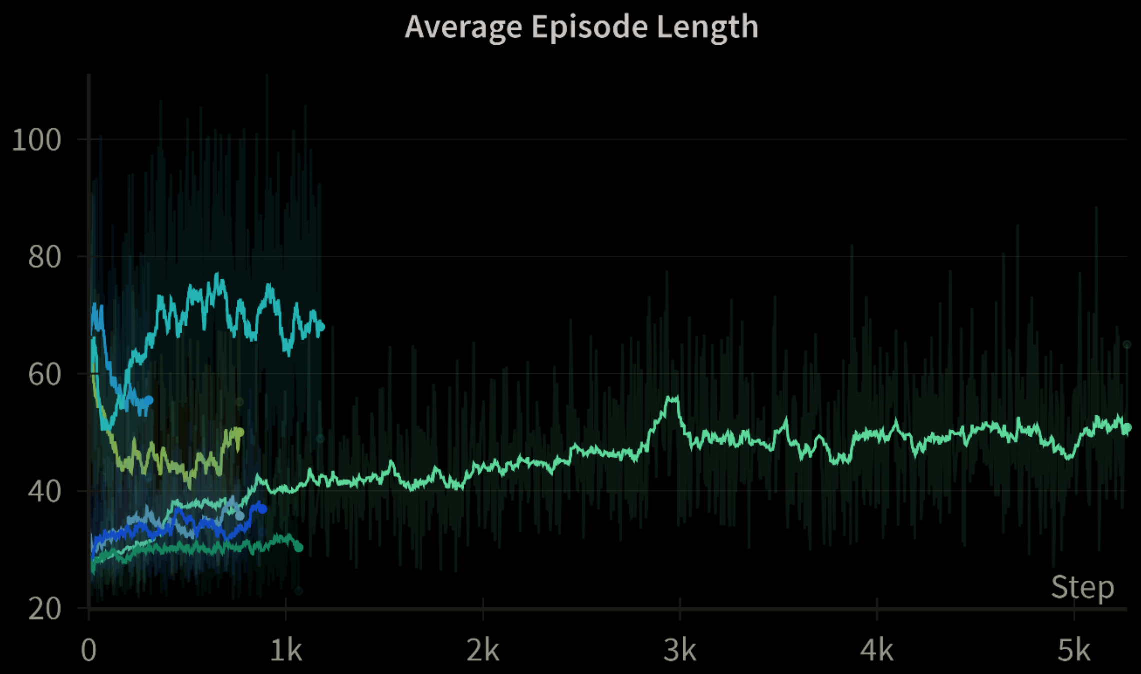
Once I stabilized the training process and identified a suitable architecture and hyperparameters, I initiated a long-term training run lasting nearly 15 days (long light green graph in the plots below). During this period, I monitored the logged metrics to ensure consistent progress.
Despite occasional plateaus in performance, iterative adjustments to hyperparameters, such as reducing the clip range and increasing the entropy coefficient, helped overcome stagnation and further improved the agent's learning. Earlier improvement attempts failed, as evidenced by declining metrics in the light blue and green graphs. The final training run lasted 2.5 days, while building on the model from the previous 15-day run, culminating in a model capable of navigating the Cosmic Voyager environment autonomously with a high level of competence.
I successfully optimized the training, resulting in significant performance gains, even though the training plateaued toward the end, the final turquoise graph with the highest reward and episode lenght indicates improved stability and performance. This final model represents the culmination of extensive experimentation, stabilization, and optimization, achieving the desired performance objectives.

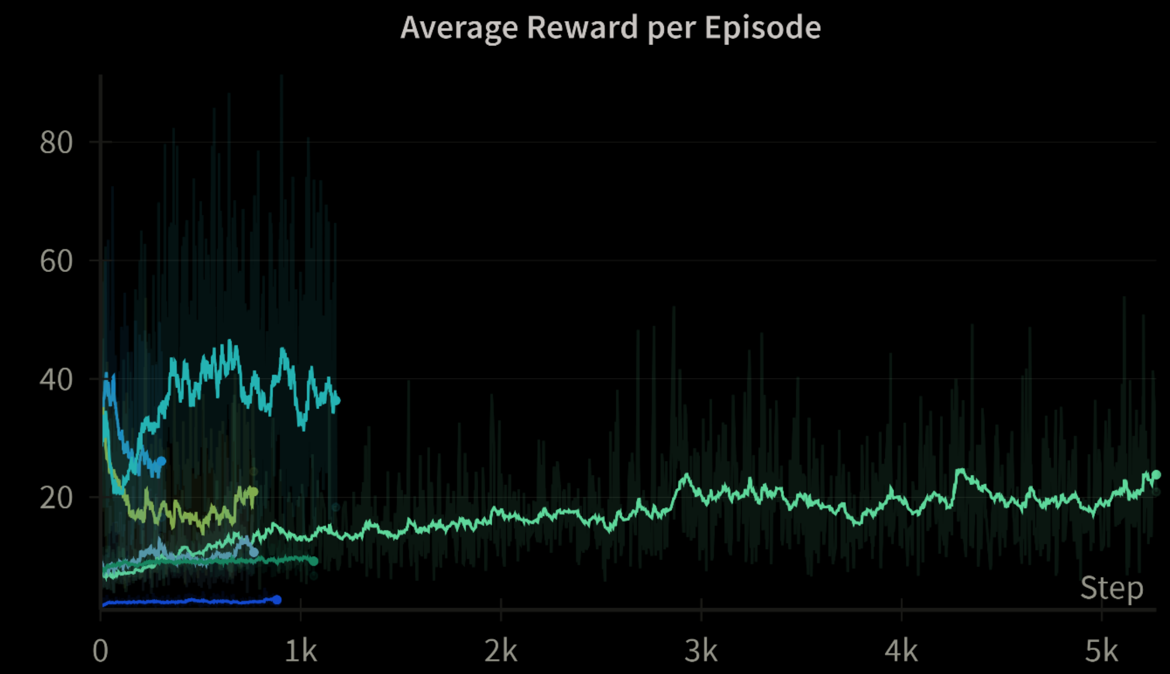
Final Model and Output of the Training Process

The training process produced a model zip folder with a size of about 210 mb containing all the components needed to use and deploy the trained AI agent. Key files include policy.pth, which holds the trained neural network weights defining the agent's decision-making strategy, and policy.optimizer.pth, which stores the optimizer state, enabling seamless continuation of training. The data file encapsulates the trained parameters and hyperparameters, effectively preserving the agent's learning. Supporting files such as _stable_baselines3_version ensure compatibility by recording the library version used, while pytorch_variables.pth stores additional metadata from PyTorch. The system_info file documents the hardware and software specifications of the training environment, ensuring reproducibility.

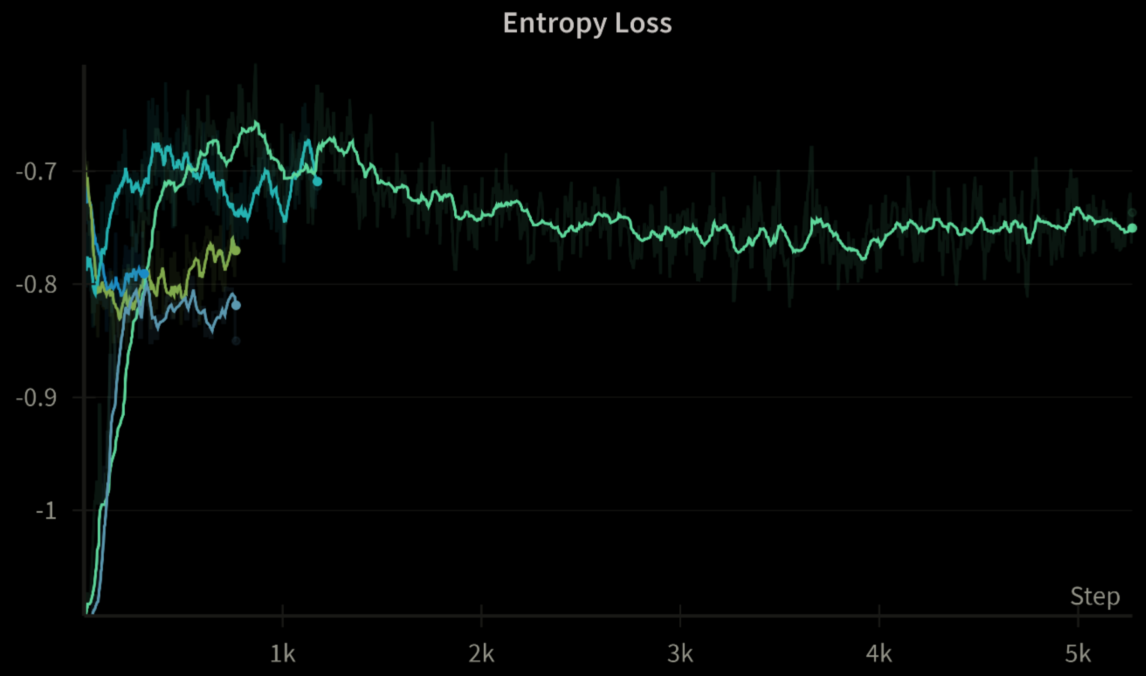
This output is comprehensive, enabling further fine-tuning, evaluation, or deployment of the trained model with minimal setup. You can access the model on Hugging Face right [here].

See the graphs for the respective runs in the plots for the average reward and episode length below:





Graphs for the entropy loss over time for the respective training runs are shown below. Early on, high entropy indicates broad exploration, and as the model learns, the policy becomes more deterministic—lowering entropy. Eventually, each curve settles around a stable level, reflecting a balance between exploration and exploitation.



Implementation of the Inference API as Middleware

The inference API plays a crucial role in the overall architecture by serving as the bridge between the frontend (web game) and the backend (machine learning model). It handles requests from the game, processes observations from the virtual environment, and communicates with the trained ML model to generate actions for the AI agent. The API is built using modern frameworks and libraries, providing robust and efficient communication between the components. This inference API ensures efficient and reliable communication between the game and the ML model, enabling the AI agent to operate autonomously and adapt to dynamic game environments. By leveraging modern tools like FastAPI, the implementation achieves high performance and scalability, essential for real-time decision-making in interactive systems.

Frameworks and Libraries

1. **FastAPI**: A modern, high-performance web framework for building APIs. It is used to define the endpoints, handle HTTP requests, and manage the communication pipeline between the frontend and backend.

2. **Stable-Baselines3**: Provides the tools to load and interact with the pretrained Proximal Policy Optimization (PPO) model, which powers the AI agent's decision-making.

3. **Torch**: The PyTorch library facilitates model inference by enabling efficient processing of neural network predictions.

4. **NumPy**: Handles numerical operations, such as preprocessing input data into the correct format for the ML model.

5. **Pillow (PIL)**: Processes image data, resizing and converting input frames to grayscale for compatibility with the model.

6. **Base64**: Encodes and decodes image data transmitted as strings between the frontend and backend.

Functionality of the Inference API

The inference API consists of an endpoint (/predict) that handles requests from the frontend. The primary function of this endpoint is to accept observations (game states) as input, preprocess these inputs, feed them into the ML model, and return the predicted actions to the frontend.

At the core of the API is the Pipeline class, which initializes by loading the pretrained ML model. This model, saved in PyTorch format, uses the PPO algorithm to make predictions. The Pipeline class defines a callable interface that takes in observations, preprocesses them, and returns the model's predicted actions.

- **Preprocessing Observations**:
  The API receives the game state as a stack of 8 consecutive frames, encoded as Base64 strings. During preprocessing, the frames are decoded, resized to 100x150 pixels, and converted to grayscale. The pixel values are normalized to fall between 0 and 1, and the frames are stacked into a single NumPy array with the required dimensions ((1, 8, 150, 100)), which includes a batch dimension.

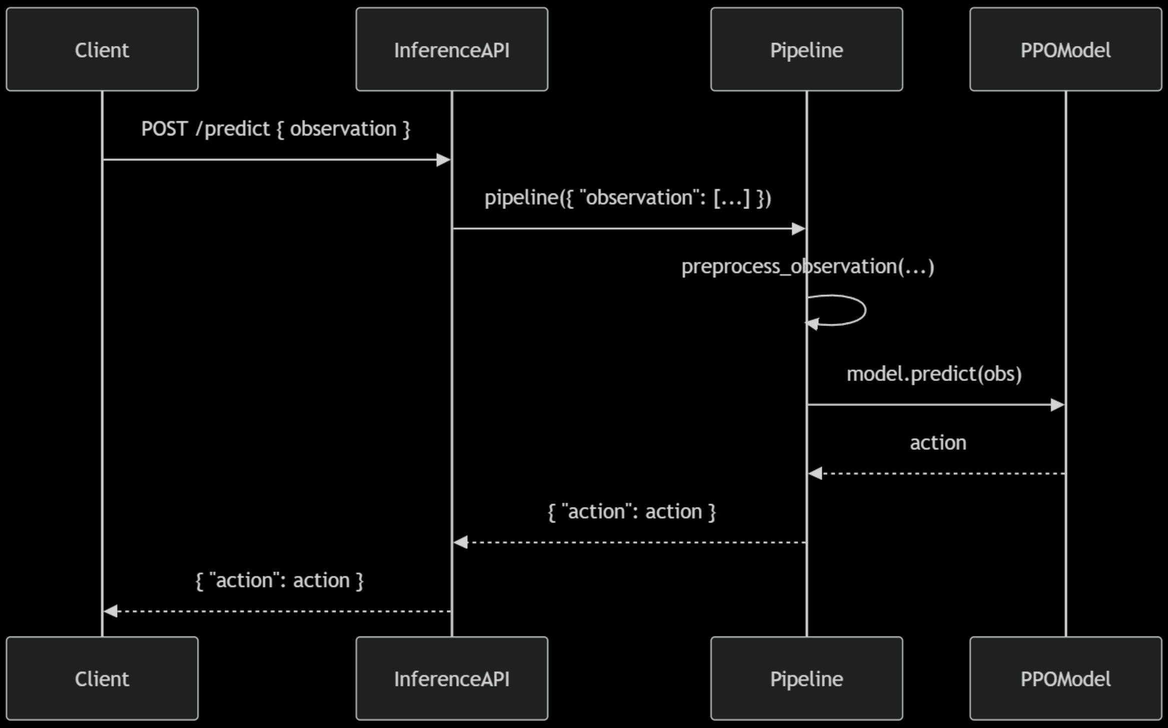
- **Model Inference**:
  The preprocessed observation is passed to the pretrained model, which predicts the optimal action for the current game state. The prediction is deterministic to ensure consistency, and the resulting action is returned as a JSON response to the frontend.

- **Error Handling**:
  The API includes robust error handling for scenarios such as invalid or insufficient input frames. If an issue arises during preprocessing or prediction, the API responds with an appropriate error message and status code.

- **Static File Hosting and CORS Support**:
  The API also serves static files, enabling seamless integration with the frontend by hosting the game interface directly. Additionally, a CORS middleware ensures that the frontend can interact with the API from any domain, simplifying development and deployment.

API Workflow

The following sequence diagram illustrates the funtionality. The client sends a POST request with a collection of image frames to an endpoint, which validates and transforms these inputs into a standardized format before passing them to the loaded ML model. After the model determines an action, the system sends a concise JSON response back to the client.



Deployment & Testing

Deployment is a critical phase in delivering machine learning applications to users, and concepts like DevOps and MLOps are foundational in achieving efficient, reliable, and scalable deployment processes. DevOps refers to the set of practices that combine software development (Dev) and IT operations (Ops) to automate and streamline the processes of building, testing, and deploying applications. It emphasizes continuous integration and deployment (CI/CD) to ensure consistent and rapid delivery of software. MLOps, on the other hand, extends these principles to machine learning workflows, focusing on automating and managing the lifecycle of ML models, from training and deployment to monitoring and updates. These approaches were partially applied in my project to ensure smooth deployment and maintenance.

Deployment on Hugging Face Spaces

To make my application publicly accessible, I deployed it on Hugging Face Spaces, a platform designed for hosting machine learning applications and demos. Hugging Face Spaces provides an intuitive interface for deploying projects, supporting various frameworks and seamless integration with Git repositories. By leveraging Docker, I containerized the application to ensure consistency across environments, streamline deployment, and simplify scaling.

DevOps principles were applied through the use of containerization and automation. Containerizing the application with Docker ensured that the deployment environment was consistent with the development environment, eliminating "it works on my machine" issues. Additionally, the Dockerfile acted as a single source of truth for the runtime configuration, making it easy to rebuild and deploy the application in other environments. From an MLOps perspective, deploying the trained machine learning model as part of the container ensured that the entire ML pipeline—from model training to deployment—was reproducible and maintainable. The integration of all components (inference API, static files, and trained model) into a single container simplified model serving and deployment and increased performance by reducing latency. This approach aligned with MLOps practices of ensuring reliability, reproducibility, and traceability in ML workflows.
The deployment process began by uploading the trained model (as a zip file), the inference script, and the static web game files to the Hugging Face Space. I then developed a comprehensive Dockerfile to configure the environment required to run the ML model and application. This Dockerfile automated the setup of system libraries, Python dependencies, and runtime commands, embodying the DevOps principle of infrastructure as code.
Further, using a specific base image (python:3.9) and fixed library versions minimizes bugs due to version mismatches. Containerization allows for easy scalability and integration into CI/CD pipelines, enabling automated testing and deployment with every update. The Dockerfile also supports model lifecycle management by integrating the correct pre-trained model for inference, ensuring straightforward updates or replacements in the future.

Final Deployment and Accessibility

Once the Dockerfile was executed, Hugging Face Spaces automatically built the environment within the container. The deployment process provided a seamless path from development to production, ensuring the application was consistent across all environments. By hosting the application on Hugging Face Spaces, users can interact with the AI-powered game via a public URL, engaging with the AI agent directly from their web browsers without the need for additional installations.

This deployment strategy reflects modern DevOps and MLOps practices:

- FastAPI and Docker streamlined development and deployment.
- Containerization ensured consistency and portability.
- Hugging Face Spaces simplified hosting and integration, embodying the principles of accessibility and scalability central to MLOps.

By combining the principles of DevOps and MLOps with the technical infrastructure provided by Hugging Face Spaces and Docker, I achieved a robust and user-friendly deployment process. This approach ensures that the application is not only accessible but also maintainable and ready for future updates or scaling needs.

Testing Approach

In my project, testing focused primarily on debugging and error handling, using printouts to the command line to identify and address runtime issues. Additionally, I leveraged Chrome's developer-specific tools—particularly the Network and Console tabs—to observe the application's behavior in real time and diagnose potential issues. While I did not implement formal unit tests with predefined test cases, I conducted manual testing for individual components. For the frontend (web game), I manually tested its behavior to ensure it functioned as desired, verifying user interactions and game logic in isolation. This corresponds to a unit testing approach, though performed manually rather than through automated frameworks. Moving to integration testing, I tested the interaction between the three components—frontend, inference API, and ML model—by initializing the inference API locally and confirming that the AI mode worked as expected. This involved verifying that the game correctly sent observations to the API and received corresponding actions from the ML model. Finally, I performed system testing by deploying the complete application to the Hugging Face Space and ensuring the entire system, including all integrated components, worked cohesively in the target environment. While the testing process was largely manual, this approach ensured each component and the full application functioned reliably from development to deployment.

Implementation Effort

Web Game

The development of Cosmic Voyager was an incredibly efficient process, thanks to the use of generative AI tools like ChatGPT and DALL-E 3. I completed the game design phase, which included brainstorming mechanics, sketching gameplay flow, and creating visual elements, in just about 10 hours. These tools made it easy to iterate quickly and refine ideas, streamlining the entire process. Implementing the game, including coding the HTML structure, CSS styling, and JavaScript logic, took another 20 hours, resulting in a compact yet functional codebase of around 850 lines. ChatGPT was instrumental during this phase, helping me solve challenges like obstacle generation and collision detection, while I stayed actively involved to ensure I understood and guided the development logic. Extending the game to support the AI mode added another 10 hours, as I made the necessary adjustments to integrate the autonomous agent. Fine-tuning the game's parameters, like obstacle speed, player responsiveness, and pacing, required several iterations to achieve a balance between challenge and playability. In total, the combination of AI-assisted development and focused testing enabled me to create a polished and engaging game in just 40 hours, highlighting how effectively generative AI can enhance the development process.

ML Model

The implementation of the machine learning model for Cosmic Voyager was a substantial effort that required a great deal of focus and iteration. I relied on OpenAI's GPT-o1 model as a programming assistant throughout the process, which was incredibly helpful in tackling challenges and debugging. Setting up the custom environment, with its 350 lines of code, was particularly demanding. It involved configuring Selenium for browser interaction and ensuring everything worked as intended, which took about 20 hours. Creating the training script, another 300 lines of code, required a similar amount of effort as I worked through debugging and integration issues.
Experimenting with different algorithms, architectures, and hyperparameters during the initial training runs took approximately 30 hours. This phase involved a lot of trial and error to refine the process and ensure consistent results. Setting up and executing multiple training runs, along with fine-tuning the hyperparameters, took an additional 25 hours before I arrived at the final model.
Developing the inference script, about 100 lines of code, and performing integration testing took another 15 hours to ensure the API worked seamlessly on my local machine. Creating the Dockerfile (50 lines of code) for containerization and testing the setup took about 5 hours. Finally, completing the deployment and making the final adjustments required around 5 more hours. In total, I spent about 120 hours over several iterative phases, combining experimentation, debugging, and optimization to successfully implement the machine learning model for Cosmic Voyager.

Developing Cosmic Voyager as a complete application, from the web game to the machine learning model, was a challenging yet rewarding experience, spanning about four weeks of full-time effort. Throughout the process, I relied heavily on generative AI tools like ChatGPT-o1, which were incredibly helpful. They provided invaluable support in coding, brainstorming ideas, and solving problems quickly. However, I realized that these tools couldn’t replace me—they needed my guidance, high-level understanding of the project, and hands-on involvement to achieve the desired outcomes. There were times when I had to solve issues myself, ensuring the project stayed on track.
Looking back, I’m impressed by how much I could accomplish with the help of AI tools. They excelled at reasoning through problems and providing coding solutions, but ultimately, it was my understanding and decision-making that brought Cosmic Voyager to life. This project taught me how powerful the combination of human expertise and AI assistance can be when creating something innovative and impactful.

Performance Optimization

When testing the AI mode, I observed that the agent's performance was not as strong as expected. This can be attributed to several factors.
First, the training process of the ML model could be further optimized. While the current training yielded a functional agent, improvements in the algorithm, architecture, and hyperparameters might enhance learning. For instance, experimenting with alternative algorithms, increasing the model's complexity, or extending the training duration could result in better performance. Leveraging AutoML tools, as previously mentioned, could streamline this optimization process by automating much of the experimentation and tuning.
Second, the deployment environment impacts performance. The agent performed noticeably better on my local machine, where it benefited from the computing power of my NVIDIA RTX 3060 GPU. On Hugging Face Spaces, I use a free space, which offers only limited CPU-based resources. This constrained environment may introduce latency during model inference, slightly delaying the agent's actions. In a fast-paced game like Cosmic Voyager, even small delays can cause the agent to fail quickly, significantly affecting its overall performance.
Lastly, generalization issues may also contribute to suboptimal performance. The model might be overfitted to the specific setup used during training. For example, the game area changes dynamically for different screen sizes, but the model was trained on a fixed virtual screen size. To address this, parallel training on environments with varying screen dimensions and combining the results could improve the model's ability to generalize across different setups, leading to more robust performance in real-world deployments.
These adjustments, both in training and deployment, hold potential to significantly enhance the agent's effectiveness in AI mode.

Conclusion

The development of **Cosmic Voyager**, from design to deployment, was an incredibly rewarding experience that showcased the immense power of generative AI tools. These tools were invaluable across every stage of the project—assisting with code generation, debugging, image creation, framework selection, and providing guidance on technologies and best practices.
The fact that I could implement such a complex project, spanning web game development, machine learning, and deployment, entirely on my own and in such a short time, highlights the paradigm shift brought about by generative AI in software development. It’s remarkable how these tools enabled me to work effectively without deep prior implementation knowledge in this domain.
That said, the success of the project also underscores the critical role of human oversight. I had to guide the AI tools carefully, defining the general solution design and

approach while steering the development process. My understanding of the overall architecture and ability to identify and address specific issues—such as library version conflicts and incompatibilities—were essential in navigating challenges where the AI tools struggled or got stuck. This reinforces the notion that while AI tools significantly amplify productivity, they cannot entirely replace human expertise and decision-making (yet).

Overall, this project was both demanding and immensely fulfilling. It offered an incredibly steep learning curve, allowing me to delve into new domains and tackle a wide range of technical challenges. At the same time, it was a lot of fun, as I witnessed the power of GenAI tools in action, transforming an ambitious idea into a fully functional application. Watching my agent learn step by step and steadily improve its performance was an incredibly exciting experience.

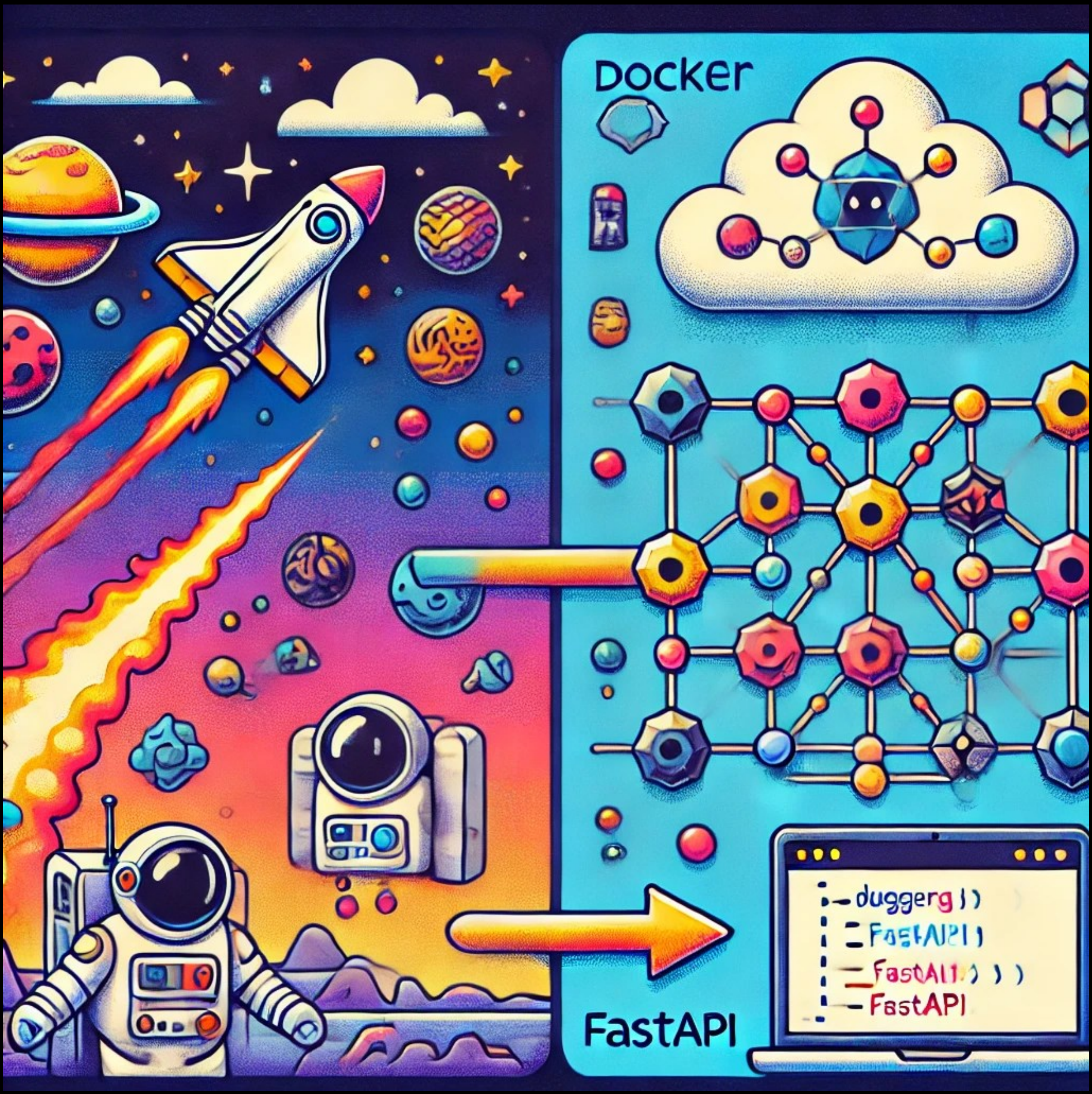
This project has demonstrated the evolving role of AI in software development, where humans and AI collaborate to achieve results that would have seemed daunting not long ago.

If you're interested in diving deeper into the code, you can find my GitHub repository for this project linked [here]. I hope you enjoy playing *Cosmic Voyager* as much as I enjoyed creating it!

You can play the deployed game yourself and try to achieve a score above 2000 as well as try the AI mode right [here].

Thanks and have fun!

Best, Andy



Finally, a nice illustration for the project from **DALL·E**.