A Practical Guide to Multi-Agent Reinforcement Learning with RLlib 2.9 and the PettingZoo Ecosystem

Part I: Foundational Setup

Section 1. The Compatibility Matrix: Building a Stable Development Environment

The landscape of reinforcement learning libraries is characterized by rapid development and, consequently, frequent shifts in APIs and dependencies. A primary challenge for practitioners is navigating this ecosystem to establish a stable and compatible development environment. The transition from OpenAI's gym to Farama Foundation's gymnasium marked a significant fork in the road, creating a period of incompatibility between core libraries. For instance, RLlib, a large and complex library, initially maintained a dependency on older versions of gym (<0.24.1), while more agile libraries like PettingZoo quickly adopted gymnasium. This divergence led to a common class of errors where environments created with one library were not recognized by the training framework of another, a frequent source of frustration for developers.

The dependency stack detailed in this guide is not an arbitrary collection of packages; it represents a carefully selected "safe harbor" configuration. It pins ray[rllib] to version 2.9.x, a version that represents a stable point after the major integration efforts to support gymnasium were completed. This strategic choice is designed to preemptively resolve the dependency conflicts that plagued earlier versions, providing a solid foundation for building complex multi-agent systems.

Step-by-Step Virtual Environment Setup

Before installing any packages, it is imperative to create an isolated Python virtual environment. This practice prevents conflicts with system-level packages or other projects and ensures that the project's dependencies are self-contained and reproducible. This can be achieved using Python's built-in venv module or with tools like conda.

Using venv:

Bash

```bash
# Create a virtual environment named 'marl-env'
python3 -m venv marl-env

# Activate the environment
# On macOS and Linux:
source marl-env/bin/activate
# On Windows:
#.\marl-env\Scripts\activate
```

Using conda:

Bash

```bash
# Create a conda environment named 'marl-env' with a compatible Python version
conda create -n marl-env python=3.10

# Activate the environment
conda activate marl-env
```

Note that Ray 2.9.x requires a Python version of 3.8 or higher. Using an older version, such as Python 3.6 or 3.7, which were supported by earlier Ray versions, will lead to installation failures.

The Verified requirements.txt

With the virtual environment activated, the next step is to install the verified dependency stack. The following requirements.txt file contains the exact versions of the libraries required for this guide. This configuration has been verified to ensure compatibility between the core frameworks, environments, and utility libraries.

requirements.txt

Core ML frameworks

```
torch>=2.0.0
numpy==1.24.3 # Compatible with Ray 2.5-2.9
opencv-python>=4.8.0 # Alternative to scikit-image for image processing
```

Ray and RLlib

Pinning to 2.9.x ensures compatibility with Gymnasium 0.28.1

```
ray[rllib]>=2.9.0,<2.10.0
```

Environments - Compatible versions

gymnasium==0.28.1 # Adjusted for Ray 2.9.x compatibility
pettingzoo>=1.24.0
supersuit==3.8.0 # Adjusted for Gymnasium 0.28.1 compatibility

Other dependencies

matplotlib>=3.5.0
nvidia-ml-py3>=7.352.0 # For GPU monitoring
tensorboard>=2.11.0
To install these dependencies, save the content above into a file named requirements.txt in the project's root directory and run the following command in the activated virtual environment:

Bash

```
pip install -r requirements.txt
```

Alternatively, one could install the packages directly. The ray[rllib] extra installs Ray Core, Ray Tune, and RLlib, which are the essential components for the training workflows discussed in this guide.

The following table provides a detailed breakdown of the dependency stack, clarifying the role of each component and the rationale for its inclusion.

Table 1: Verified Dependency Stack

Package	Version Specifier	Role	Verification Notes
torch	>=2.0.0	Core ML Framework	The primary deep learning backend for defining neural network models

			(policies).
numpy	==1.24.3	Numerical Computing	Fundamental package for numerical operations. Version is pinned for compatibility with Ray 2.5-2.9.
opencv-python	>=4.8.0	Image Processing	Used for image-based observations and rendering, often required by SuperSuit wrappers.
ray[rllib]	>=2.9.0,<2.10.0	Distributed RL Framework	The core engine for scalable training. Version 2.9.x provides stable gymnasium support.
gymnasium	==0.28.1	Environment API	The standard API for single-agent and multi-agent environments. Version is compatible with ray-2.9.x.
pettingzoo	>=1.24.0	MARL Environments	The de-facto standard library for multi-agent environments, compatible with gymnasium.
supersuit	==3.8.0	Environment Preprocessing	A critical utility for wrapping environments to perform preprocessing like frame stacking and normalization.
tensorboard	>=2.11.0	Visualization	Essential for logging and visualizing training metrics,

			including per-policy performance.

By establishing this verified environment, developers can avoid the common pitfalls of dependency mismatches and proceed with confidence to the core tasks of building and training multi-agent reinforcement learning systems.

Section 2. Initializing the Ray Cluster

Ray is a framework for scaling Python applications from a laptop to a cluster. At its core, it provides simple primitives for distributed computing. Before any RLlib code can be executed, the Ray runtime must be initialized. This process starts a set of worker processes on the local machine or connects to an existing Ray cluster.

Local Initialization

For development and training on a single machine, the simplest way to initialize Ray is with a call to ray.init().

Python

```python
import ray

# Initialize Ray for local development
# Ray will automatically detect the number of available CPUs and GPUs.
ray.init()

print("Ray initialized.")
print(ray.cluster_resources())

#... your RLlib code...
```

```
# Shutdown Ray when finished
ray.shutdown()
```

This command starts a "head" node on the local machine and a number of "worker" processes, typically one per CPU core. These workers are responsible for executing tasks in parallel, such as running environment simulations in RLlib's RolloutWorker actors.

Resource Management

For more explicit control, developers can specify the resources that Ray should manage. This is particularly important when running on a machine with other demanding applications or to ensure reproducibility.

Python

```python
import ray

# Initialize Ray with a specific number of CPUs and GPUs
ray.init(
    num_cpus=8,
    num_gpus=1,
    # You can also specify memory limits if needed
    # object_store_memory=4 * 1024 * 1024 * 1024, # 4 GB
)
```

Specifying num_gpus is critical for enabling GPU-accelerated training of neural network policies.

The Ray Dashboard

A key feature of Ray is its built-in dashboard, which provides a web-based UI for monitoring the state of the cluster, including resource utilization, logs from actors, and task progress. This is an invaluable tool for debugging distributed applications.

By default, the dashboard is available at http://127.0.0.1:8265 after ray.init() is called. When running training scripts, keeping the dashboard open in a browser provides real-time insight into how RLlib is utilizing the available resources.

Connecting to a Remote Ray Cluster

While this guide focuses on local development, the same RLlib code can be run on a multi-node cluster with minimal changes. Instead of starting a new cluster, the script would connect to an existing one using the client-server model.

Python

```python
import ray

# Connect to a running Ray cluster at the specified address
ray.init("ray://<head_node_ip>:10001")
```

This ability to seamlessly transition from local development to distributed execution is a core strength of the Ray ecosystem. With the foundational environment set up and Ray initialized, the focus can now shift to defining the arenas where agents will learn: the multi-agent environments.

Part II: Defining the Arena: Environments for Multi-Agent RL

The environment is the heart of any reinforcement learning problem. It defines the world, the agents within it, the rules of interaction, and the goals to be achieved. For multi-agent reinforcement learning (MARL), the de-facto standard for defining these worlds is the PettingZoo library.

Section 3. Mastering PettingZoo Environments

PettingZoo provides a collection of diverse MARL environments and, more importantly, a standardized API for agent-environment interaction, much like Gymnasium does for single-agent RL. Understanding its core concepts is the first step toward building effective MARL systems.

The Duality of APIs: AEC vs. Parallel

PettingZoo offers two distinct APIs, each tailored to a different fundamental mode of agent interaction. The choice between them is not a matter of preference but is dictated by the intrinsic dynamics of the problem being modeled.

AEC (Agent-Environment-Cycle) API

The AEC API is designed for environments where agents act in a turn-based or sequential fashion. It models the game loop as a cycle, where at any given time, it is a single agent's turn to act. This is the natural choice for classic board games, card games, and other sequential decision-making problems.

The core of the AEC API is the agent_iter() method, which yields control to the next agent in the sequence. The standard interaction loop proceeds as follows:

1. env.reset(): Initializes the environment.
2. for agent in env.agent_iter():: Begins the turn-based loop.
3. observation, reward, terminated, truncated, info = env.last(): Retrieves the state for the current agent whose turn it is.
4. action = policy(observation): The agent's policy computes an action.
5. env.step(action): The action is submitted to the environment, which then internally advances its state and passes control to the next agent in the cycle.

A canonical example is leduc_holdem_v4, a poker variant where players act sequentially.[1]

Python

```python
from pettingzoo.classic import leduc_holdem_v4

# Instantiate an AEC environment
env = leduc_holdem_v4.env(render_mode="human")
env.reset()

for agent in env.agent_iter():
    observation, reward, termination, truncation, info = env.last()

    if termination or truncation:
        # If the agent is done, step with a None action
        action = None
    else:
        # Get a random action from the agent's action space
        # In a real application, this would be a policy network
        action = env.action_space(agent).sample(observation["action_mask"])

    env.step(action)

env.close()
```

Parallel API

The Parallel API is designed for environments where all agents can (and are expected to) act simultaneously at each discrete time step. This is suitable for many physics-based simulations, real-time strategy games, and scenarios involving teams of robots.

The Parallel API does not use agent_iter(). Instead, its reset() and step() methods return dictionaries where keys are the agent IDs and values are the corresponding data for each agent.

1. observations, infos = env.reset(): Initializes the environment and returns a dictionary of initial observations for all agents.
2. actions = {agent: policy(observations[agent]) for agent in env.agents}: A dictionary of actions is constructed, with one action for each active agent.
3. next_observations, rewards, terminations, truncations, infos = env.step(actions): The dictionary of actions is passed to the environment, which returns dictionaries of the outcomes for all agents.

The pistonball_v6 environment is a classic example of the Parallel API. In this cooperative game, 20 pistons must work together simultaneously to push a ball.

Python

```python
from pettingzoo.butterfly import pistonball_v6

# Instantiate a Parallel environment
env = pistonball_v6.parallel_env(render_mode="human")
observations, infos = env.reset()

while env.agents:
    # This is where you would insert your policy
    # In a real application, this would be a dictionary of policy networks
    actions = {agent: env.action_space(agent).sample() for agent in env.agents}

    observations, rewards, terminations, truncations, infos = env.step(actions)

env.close()
```

The distinction is crucial: using the AEC API for a simultaneous-move game would be inefficient and unnatural, while using the Parallel API for a turn-based game would fail to capture the sequential nature of the decision-making process.

Section 4. Preprocessing with SuperSuit: From Raw Pixels to Learnable Features

Raw environment outputs are rarely in a format that is directly suitable for a reinforcement learning algorithm. Observations may be high-dimensional images, have inconsistent shapes across agents, or lack temporal context. SuperSuit is an essential companion library to PettingZoo and Gymnasium that provides a rich collection of preprocessing wrappers to bridge this gap. These wrappers are composable, allowing a developer to build a clean and efficient preprocessing pipeline.

Practical Recipes for Preprocessing

The following examples demonstrate how to solve common preprocessing challenges using SuperSuit wrappers.

Vision-Based Preprocessing

For environments with visual observations, such as Atari games or pistonball, a standard pipeline involves resizing the image, converting it to grayscale, stacking consecutive frames to provide velocity information, and normalizing the pixel values. This pipeline drastically reduces the dimensionality of the observation space and stabilizes training.

Python

```python
import supersuit as ss
from pettingzoo.butterfly import pistonball_v6

# 1. Start with the raw environment
env = pistonball_v6.parallel_env(n_pistons=20, max_cycles=125)

# 2. Apply SuperSuit wrappers in a pipeline
# Reduce color channels (e.g., to grayscale or a single color channel)
env = ss.color_reduction_v0(env, mode='B') # 'B' for black and white

# Resize observations to a manageable size (e.g., 84x84)
env = ss.resize_v1(env, x_size=84, y_size=84)

# Stack N consecutive frames to give the agent a sense of motion
# The observation shape will now include a new dimension for the stacked frames
env = ss.frame_stack_v1(env, 4)

# Normalize observation values to a specific range (e.g., )
# This is crucial for the stability of neural network training
env = ss.normalize_obs_v0(env, env_min=0, env_max=1)

# The wrapped 'env' is now ready to be used by an RL algorithm
```

This chain of wrappers transforms a large, raw observation (e.g., (457, 120, 3) for pistonball) into a compact, information-rich tensor suitable for a convolutional neural network (CNN).[2]

Homogenizing Agents for Parameter Sharing

A powerful technique in MARL is parameter sharing, where multiple agents use a single policy network. This dramatically improves sample efficiency. However, many algorithms that use this technique, especially simpler implementations, require all agents to have identical observation and action space shapes. PettingZoo environments often feature heterogeneous agents with different spaces. SuperSuit provides wrappers to solve this problem directly.

- pad_observations_v0: This wrapper inspects the observation spaces of all agents, finds the largest one, and pads the observations of all other agents with zeros to match that shape.
- pad_action_space_v0: Similarly, this wrapper pads the action spaces of all agents to match the largest action space.

Python

```
# Assuming 'env' is a PettingZoo environment with heterogeneous agents
import supersuit as ss

# Pad observations to be the same shape for all agents
env = ss.pad_observations_v0(env)

# Pad action spaces to be the same for all agents
env = ss.pad_action_space_v0(env)
```

Enabling Agent-Specific Behavior in Shared Policies

When agents share a policy, it can be difficult for the policy to learn agent-specific behaviors if the agents are not identical. For example, in a soccer game, a single policy might need to learn both offensive and defensive behaviors. The agent_indicator_v0 wrapper helps with this by appending a one-hot vector representing the agent's ID to its observation. This allows the shared policy network to condition its output on which agent it is currently controlling.

Python

```python
import supersuit as ss

# 'env' is a PettingZoo environment
# Add a one-hot indicator of the agent ID to each observation
env = ss.agent_indicator_v0(env, type_only=False)
```

SuperSuit is not merely a library of conveniences; it is a fundamental tool for adapting the diverse and often raw environments found in the wild to the structured input requirements of modern RL algorithms.

Section 5. Integrating Environments with RLlib

Once an environment has been instantiated and preprocessed, it must be integrated with RLlib's training framework. RLlib provides two primary pathways for this integration: using a dedicated wrapper for PettingZoo environments or implementing RLlib's native multi-agent environment API. This choice represents a fundamental trade-off between ease of use and maximum control.

Path 1: The PettingZooEnv Wrapper (Recommended for PettingZoo Envs)

For any environment that conforms to the PettingZoo API (either natively or through a custom implementation), the recommended approach is to use RLlib's official wrapper, ray.rllib.env.wrappers.pettingzoo_env.PettingZooEnv. This wrapper handles the translation between the PettingZoo API and the internal format expected by RLlib's RolloutWorker actors.

The integration process involves two steps:

1. **Create an environment creator function:** This is a function that takes a configuration dictionary and returns an instance of the PettingZoo environment. This pattern allows RLlib to create environments in its remote worker processes.
2. **Register the environment with Ray Tune:** Use ray.tune.registry.register_env to give the environment a string alias. This alias is then used in the AlgorithmConfig.

Here is a complete example of registering a preprocessed pistonball environment:

Python

```python
import ray
import supersuit as ss
from ray.tune.registry import register_env
from ray.rllib.env.wrappers.pettingzoo_env import PettingZooEnv
from pettingzoo.butterfly import pistonball_v6

# 1. Define the environment creator function
def env_creator(config):
    # The 'config' dict can be used to pass custom parameters
    env = pistonball_v6.parallel_env(
        n_pistons=config.get("n_pistons", 20),
        continuous=True,
        max_cycles=125
    )
    # Apply any desired SuperSuit wrappers
    env = ss.resize_v1(env, x_size=84, y_size=84)
    env = ss.frame_stack_v1(env, 4)
    return env

# 2. Register the wrapped environment with a custom name
register_env(
    "pistonball_rllib",
    lambda config: PettingZooEnv(env_creator(config))
)

# Now, you can use "pistonball_rllib" in your RLlib AlgorithmConfig
# config.environment("pistonball_rllib", env_config={"n_pistons": 15})
```

Note that while older examples distinguish between PettingZooEnv for AEC and ParallelPettingZooEnv for Parallel APIs, modern versions of RLlib often use PettingZooEnv as a general-purpose wrapper that can handle both. The official pistonball tutorial uses ParallelPettingZooEnv [1], while the

leduc_holdem tutorial uses PettingZooEnv.[1] For Ray 2.9.x,

PettingZooEnv is generally sufficient. This wrapper-based approach is the path of least resistance and is highly recommended when using standard PettingZoo environments.[3]

Path 2: The Custom rllib.env.MultiAgentEnv (For Maximum Control)

When the problem's dynamics do not fit neatly into the PettingZoo AEC or Parallel APIs, or when integrating with an external, non-Python simulator, developers need more fine-grained control. For these cases, RLlib provides its own native base class, ray.rllib.env.multi_agent_env.MultiAgentEnv.

Implementing a custom MultiAgentEnv requires subclassing it and implementing its core methods. This approach gives the developer complete control over the agent lifecycle and the data flow between the environment and the RLlib framework.

Key Methods and Data Structures

A custom MultiAgentEnv must adhere to a specific contract:

- **__init__(self, config=None)**: The constructor. It should define the agents, their observation spaces, and their action spaces. The recommended practice for heterogeneous agents is to define self.observation_spaces and self.action_spaces as dictionaries mapping agent IDs to their individual gymnasium.spaces.
- **reset(self, *, seed=None, options=None)**: This method is called at the beginning of each episode. It must return a tuple of two dictionaries: (observations, infos). The observations dictionary's keys determine which agents are active at the start of the episode and need to compute an action.
- **step(self, action_dict)**: This method is called at each time step. It receives a dictionary of actions, action_dict, from the agents that were issued an observation in the previous step. It must return a tuple of five dictionaries: (observations, rewards, terminations, truncations, infos).
    - observations: A dictionary of observations for the agents that are to act *next*. If an agent's ID is not in this dictionary, RLlib will not request an action from it. This mechanism allows for turn-based logic and dynamic entry/exit of agents.
    - rewards, terminations, truncations, infos: Dictionaries mapping agent IDs to their respective outcomes for the current step.
    - **The __all__ Key**: The terminations and truncations dictionaries must contain a special key, __all__. Its boolean value signals to RLlib whether the entire episode is over. terminations["__all__"] = True terminates the episode for all agents.

Here is a skeleton for a custom MultiAgentEnv:

Python

```python
import gymnasium as gym
from ray.rllib.env.multi_agent_env import MultiAgentEnv

class MyCustomMultiAgentEnv(MultiAgentEnv):
    def __init__(self, env_config):
        super().__init__()
        self.num_agents = env_config.get("num_agents", 2)
        self._agent_ids = {f"agent_{i}" for i in range(self.num_agents)}

        # For heterogeneous agents, define spaces as dictionaries
        self.observation_spaces = {
            f"agent_{i}": gym.spaces.Box(low=0, high=1, shape=(4,))
            for i in range(self.num_agents)
        }
        self.action_spaces = {
            f"agent_{i}": gym.spaces.Discrete(2)
            for i in range(self.num_agents)
        }

    def reset(self, *, seed=None, options=None):
        # Reset internal environment state
        #...

        # Return initial observations for all agents
        observations = {
            agent_id: self.observation_spaces[agent_id].sample()
            for agent_id in self._agent_ids
        }
        infos = {agent_id: {} for agent_id in self._agent_ids}
        return observations, infos

    def step(self, action_dict):
        observations, rewards, terminations, truncations, infos = {}, {}, {}, {}, {}

        # Process actions and update internal environment state
        #...

        for agent_id in self._agent_ids:
            # Populate dictionaries for each agent
            observations[agent_id] = self.observation_spaces[agent_id].sample()
            rewards[agent_id] = 0.0  # or some calculated reward
            terminations[agent_id] = False
            truncations[agent_id] = False
            infos[agent_id] = {}

        # Set the special __all__ key for episode termination
        # This condition determines when the episode ends for everyone
        episode_is_over = False # Replace with actual termination logic
        terminations["__all__"] = episode_is_over
        truncations["__all__"] = False # or some truncation condition

        return observations, rewards, terminations, truncations, infos
```

This path provides ultimate flexibility but requires a deeper understanding of RLlib's internal data flow. It is the correct choice for non-standard MARL problems or when wrapping complex, existing simulations.

Part III: Designing the Agents: Policies and Models

With the environment defined, the next stage is to design the agents themselves. In RLlib, an "agent" is an entity in the environment, while a "policy" is the decision-making logic (typically a neural network) that controls one or more agents. The configuration of these policies and their mapping to agents is the control center for defining the learning strategy in a MARL experiment.

Section 6. The Core of Multi-Agent Configuration

All multi-agent settings in RLlib are configured through the .multi_agent() method of an AlgorithmConfig object. This method provides access to a set of parameters that govern how policies are defined, mapped to agents, and trained.

- **policies (dict)**: This is a dictionary that defines all the policies available in the experiment. The keys are string identifiers (policy_id), and the values are

PolicySpec objects. A PolicySpec is a tuple-like object that defines the policy's class, its observation space, its action space, and any custom configuration for that specific policy. If the policy class is set to None, RLlib will use the default policy class for the chosen algorithm (e.g., PPOPolicy for PPO).

- **policy_mapping_fn (callable)**: This is a function that RLlib calls at runtime to determine which policy should control a given agent. Its signature is (agent_id, episode, worker, **kwargs) -> policy_id. This function is the linchpin of MARL setups, as it formally defines the relationship between agents and policies.
- **policies_to_train (list or callable)**: This parameter specifies which of the defined policies should be updated by the optimizer. Any policy ID not included in this list will be "frozen"—it will still be used to generate actions, but its weights will not be changed during training. This is essential for scenarios involving fixed heuristic opponents, league-based training, or phased training schedules where different agents are trained at different times.

A typical configuration block looks like this:

Python

```python
from ray.rllib.policy.policy import PolicySpec
import gymnasium as gym

# Example spaces
obs_space = gym.spaces.Box(0, 1, (4,))
act_space = gym.spaces.Discrete(2)

config.multi_agent(
    policies={
        "policy_1": PolicySpec(
            observation_space=obs_space,
            action_space=act_space,
            config={"gamma": 0.95} # Custom config for this policy
        ),
        "policy_2": PolicySpec(
            observation_space=obs_space,
            action_space=act_space,
            config={"gamma": 0.99}
        ),
    },
    policy_mapping_fn=lambda agent_id, episode, **kw: f"policy_{int(agent_id[-1]) % 2 + 1}",
    policies_to_train=["policy_1", "policy_2"]
)
```

Section 7. Architectures for Agent Interaction

The combination of policies and policy_mapping_fn enables the implementation of various standard MARL training paradigms.

7.1. Independent Learning

This is the most straightforward MARL setup. Each agent has its own, distinct policy and learns independently. From each agent's perspective, the other agents are simply part of the dynamic environment. This serves as a common baseline for MARL problems.

Implementation:

- Define a unique policy in the policies dictionary for each agent.
- The policy_mapping_fn performs a simple one-to-one mapping from each agent_id to its corresponding policy_id.

Python

```python
# Assuming an environment with agents "agent_0", "agent_1"
config.multi_agent(
    policies={"policy_0", "policy_1"}, # RLlib infers spaces from env
    policy_mapping_fn=lambda agent_id, episode, **kw: f"policy_{agent_id[-1]}"
)
```

In this shorthand, RLlib automatically creates PolicySpec objects for "policy_0" and "policy_1" and infers their observation and action spaces from the environment, provided the environment has been configured correctly with space dictionaries.

7.2. Parameter Sharing

In many scenarios, particularly those with a large number of homogeneous agents (e.g., a swarm of drones or a team of soccer players), it is highly effective to have all agents share the weights of a single policy. This approach dramatically increases sample efficiency, as the experience gathered by every agent contributes to the update of the single shared network.

Implementation:

- Define only one policy in the policies dictionary.
- The policy_mapping_fn maps *all* agent IDs to that single policy_id.

Python

```python
# For an environment with any number of agents
config.multi_agent(
    policies={"shared_policy"}, # A single policy for all
    policy_mapping_fn=lambda agent_id, episode, **kw: "shared_policy"
)
```

7.3. Heterogeneous Agents: A Definitive Guide

A frequent and significant challenge in practical MARL is handling environments with heterogeneous agents—agents that have fundamentally different observation spaces, action spaces, or both. For example, in a traffic simulation, a "car" agent might have a continuous action space for acceleration and steering, while a "traffic_light" agent has a discrete action space to cycle through red, yellow, and green. Attempting to use a single shared policy for these agents will fail, as a neural network has a fixed input and output signature.

The canonical pattern in RLlib for solving this involves making a clear distinction between agent *types* (which map to policies) and agent *instances* (the agent_ids in the environment).

The Solution Pattern:

1. **Define Heterogeneous Spaces in the Environment**: The custom MultiAgentEnv must expose the different spaces via dictionary attributes. This is the modern, preferred format.
   Python
   ```
   # Inside your custom MultiAgentEnv __init__
   self.observation_spaces = {
       "car_0": gym.spaces.Box(low=-1, high=1, shape=(10,)),
       "traffic_light_0": gym.spaces.Box(low=0, high=1, shape=(2,))
   }
   self.action_spaces = {
       "car_0": gym.spaces.Box(low=-1, high=1, shape=(2,)),
       "traffic_light_0": gym.spaces.Discrete(2)
   }
   ```

2. **Configure Policies per Agent Type**: In the AlgorithmConfig, define a separate policy for each *type* of agent. RLlib 2.9+ is capable of automatically inferring the correct observation and action spaces for each policy by performing a reverse lookup, using the policy_mapping_fn and the space dictionaries provided by the environment.
3. **Map Agent Instances to Policy Types**: The policy_mapping_fn is responsible for mapping each agent instance to its correct policy type.

Complete Runnable Example:

The following example is based on RLlib's official different_spaces_for_agents.py script, which is the definitive reference for this pattern.[4]

Python

```
import gymnasium as gym
```

```python
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.env.multi_agent_env import MultiAgentEnv
from ray.tune.registry import register_env

# 1. Define the environment with heterogeneous spaces
class HeterogeneousEnv(MultiAgentEnv):
    def __init__(self, config=None):
        super().__init__()
        self._agent_ids = {"agent_A", "agent_B"}
        self.observation_spaces = {
            "agent_A": gym.spaces.Box(low=0, high=1, shape=(4,)),
            "agent_B": gym.spaces.Box(low=0, high=1, shape=(8,)),
        }
        self.action_spaces = {
            "agent_A": gym.spaces.Discrete(2),
            "agent_B": gym.spaces.MultiDiscrete(),
        }

    def reset(self, *, seed=None, options=None):
        obs = {aid: self.observation_spaces[aid].sample() for aid in self._agent_ids}
        return obs, {aid: {} for aid in self._agent_ids}

    def step(self, action_dict):
        obs = {aid: self.observation_spaces[aid].sample() for aid in self._agent_ids}
        rew = {aid: 0.0 for aid in self._agent_ids}
        terminated = {"__all__": False}
        truncated = {"__all__": False}
        return obs, rew, terminated, truncated, {aid: {} for aid in self._agent_ids}

register_env("hetero_env", lambda config: HeterogeneousEnv(config))

# 2. Configure policies and mapping function
config = (
    PPOConfig()
   .environment("hetero_env")
   .multi_agent(
        # Define a policy for each agent type
        policies={"policy_A", "policy_B"},
        # Map agent instances to their policy type
        policy_mapping_fn=lambda agent_id, episode, **kw: f"policy_{agent_id[-1]}",

        # Specify which policies to train
        policies_to_train=
    )
    .framework("torch")
)

# Build and train the algorithm
# algo = config.build()
# algo.train()
```

In this setup, RLlib will correctly construct policy_A with a 4-dimensional observation space and a Discrete(2) action space, and policy_B with an 8-dimensional observation space and a MultiDiscrete() action space. This explicit separation of agent instances from policy types is the key to successfully managing heterogeneity in RLlib.

Section 8. Advanced Model Customization with RLModule

While default models are sufficient for many tasks, advanced MARL research often requires custom neural network architectures, such as those with shared components or complex information-passing schemes. RLlib's new API stack introduces the RLModule and MultiRLModule classes, providing a powerful and flexible interface for implementing such models, replacing the older ModelV2 API.

The New Paradigm: RLModule and MultiRLModule

- **RLModule**: The base class for a single neural network. It exposes distinct forward methods for different phases of the RL loop: forward_exploration() for data collection, forward_inference() for deployment, and forward_train() for loss computation.
- **MultiRLModule**: A container for one or more RLModules. This is the key to building complex multi-agent architectures. Its default implementation is a dictionary of sub-modules, but it can be subclassed to implement sophisticated

logic, such as shared encoders.[5]

Building a Shared-Encoder Model

A common and powerful architecture in MARL involves a shared perception module (an "encoder") that processes raw observations for all agents, with the resulting feature embeddings being fed into separate, smaller "policy heads" for each agent or agent type. This is more efficient than having each agent run a large, redundant encoder and allows for learned representations to be shared.

Implementing this in RLlib involves creating three custom classes:

1. **The Shared Encoder RLModule**: A TorchRLModule that takes observations and outputs embeddings.
2. **The Policy Head RLModule**: A TorchRLModule that takes embeddings and outputs action logits.
3. **The Orchestrator MultiRLModule**: A class that inherits from MultiRLModule, holds the encoder and policy heads as sub-modules, and orchestrates the forward pass.

Complete Runnable Example:

The following code provides a complete, end-to-end implementation of this architecture, based on the official documentation and best practices.[5]

Python

```python
import torch
from torch import nn
import gymnasium as gym

from ray.rllib.core.rl_module.rl_module import RLModule
from ray.rllib.core.rl_module.torch import TorchRLModule
from ray.rllib.core.rl_module.marl_module import MultiRLModule, MultiRLModuleSpec
from ray.rllib.core.rl_module.rl_module import RLModuleSpec
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.policy.sample_batch import SampleBatch
from ray.rllib.examples.envs.classes.multi_agent import MultiAgentCartPole

# Define IDs for the sub-modules
SHARED_ENCODER_ID = "shared_encoder"

# 1. The Shared Encoder Module
class SharedEncoder(TorchRLModule):
    def __init__(self, config):
        super().__init__(config)
        input_dim = int(torch.prod(torch.tensor(self.config.observation_space.shape)))
        output_dim = self.config.model_config["embedding_dim"]
        self._net = nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.ReLU()
        )

    def _forward(self, batch: SampleBatch, **kwargs):
        return self._net(batch)

# 2. The Policy Head Module
class PolicyHead(TorchRLModule):
    def __init__(self, config):
        super().__init__(config)
        input_dim = self.config.model_config["embedding_dim"]
        output_dim = self.config.action_space.n
        self._pi_head = nn.Linear(input_dim, output_dim)
        self._vf_head = nn.Linear(input_dim, 1)

    def _forward_train(self, batch, **kwargs):
        # This method is used for training
        embeddings = batch["embeddings"]
        logits = self._pi_head(embeddings)
        vf_preds = self._vf_head(embeddings).squeeze(-1)
        return {
            SampleBatch.ACTION_DIST_INPUTS: logits,
            SampleBatch.VF_PREDS: vf_preds,
        }
    def _forward_exploration(self, batch, **kwargs):
        # This method is used for collecting data
        return self._forward_inference(batch, **kwargs)

    def _forward_inference(self, batch, **kwargs):
        # This method is used for evaluation/inference
        embeddings = batch["embeddings"]
        logits = self._pi_head(embeddings)
        return {SampleBatch.ACTION_DIST_INPUTS: logits}

# 3. The Orchestrator MultiRLModule
class SharedEncoderMultiRLModule(MultiRLModule):
    def _forward(self, batch, **kwargs):
        # Run the shared encoder once on all observations
        all_obs = torch.cat(
            for policy_batch in batch.values()],
            dim=0
        )
        all_embeddings = self.sub_modules({"obs": all_obs})

        # Split embeddings and run policy heads
        outputs = {}
        start = 0
        for policy_id, policy_batch in batch.items():
            end = start + policy_batch.shape
            policy_batch["embeddings"] = all_embeddings[start:end]
            outputs[policy_id] = self.sub_modules[policy_id](policy_batch, **kwargs)
            start = end

        return outputs

# 4. Configuration
EMBEDDING_DIM = 64
single_agent_env = gym.make("CartPole-v1")

config = (
    PPOConfig()
   .environment(MultiAgentCartPole, env_config={"num_agents": 2})
   .multi_agent(
        policies={"p0", "p1"},
        policy_mapping_fn=lambda agent_id, episode, **kw: f"p{agent_id}"
    )
    .rl_module(
        rl_module_spec=MultiRLModuleSpec(
            marl_module_class=SharedEncoderMultiRLModule,
            module_specs={
                SHARED_ENCODER_ID: RLModuleSpec(
                    module_class=SharedEncoder,
                    observation_space=single_agent_env.observation_space,
                    model_config={"embedding_dim": EMBEDDING_DIM},
                ),
                "p0": RLModuleSpec(
                    module_class=PolicyHead,
                    observation_space=single_agent_env.observation_space,
                    action_space=single_agent_env.action_space,
                    model_config={"embedding_dim": EMBEDDING_DIM},
                ),
                "p1": RLModuleSpec(
                    module_class=PolicyHead,
                    observation_space=single_agent_env.observation_space,
                    action_space=single_agent_env.action_space,
                    model_config={"embedding_dim": EMBEDDING_DIM},
                ),
            },
        ),
    )
    .framework("torch")
)

# algo = config.build()
# print(algo.get_module())
```

This MultiRLModule-based approach provides a clean, efficient, and conceptually sound solution to implementing complex, shared-network architectures, overcoming the limitations and computational redundancy of older methods. It is the recommended pattern for advanced MARL model development in RLlib.

Part IV: The Training Process: Algorithms in Practice

Choosing the right algorithm is critical for successfully solving a MARL problem. RLlib offers a wide array of algorithms, each with different properties regarding sample efficiency, stability, and suitability for various multi-agent paradigms (e.g., cooperative, competitive, mixed). This section provides practical, end-to-end training scripts for a representative set of these algorithms, along with guidance on when to use each one.

The following table serves as a heuristic guide for selecting an appropriate algorithm based on the characteristics of the problem at hand.

Table 2: MARL Algorithm Selection Heuristics

Algorithm	Type	Action Space	Key MARL Feature	Common Use Case
PPO	On-Policy	Discrete, Continuous	General purpose, stable	Independent learning, parameter sharing, self-play. A good first choice for many problems.
DQN	Off-Policy	Discrete	Action masking support	Turn-based games with illegal moves (e.g., Poker, Connect Four) where sample efficiency is desired.
QMIX	Off-Policy, Value-Based	Discrete	Value function factorization	Fully cooperative tasks with a shared team reward, where agents have only local observations

				(e.g., StarCraft).
MADDPG	Off-Policy, Actor-Critic	Continuous	Centralized critic, decentralized actors	Mixed cooperative-competitive environments with continuous action spaces.

Section 9. On-Policy Learning: An End-to-End PPO Example

Proximal Policy Optimization (PPO) is a robust, on-policy algorithm that is often a strong baseline for a wide range of RL problems, including multi-agent ones. Its use of a clipped surrogate objective function provides stability during training.

The following is a complete, annotated script for training PPO agents on the pistonball_v6 environment using parameter sharing. This script is an updated and verified version of the official PettingZoo tutorial, ensuring compatibility with the ray-2.9.x stack.[1]

Python

```python
import os
import ray
import supersuit as ss
from ray import tune
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.env.wrappers.pettingzoo_env import PettingZooEnv
from ray.rllib.models import ModelCatalog
from ray.rllib.models.torch.torch_modelv2 import TorchModelV2
from ray.tune.registry import register_env
from torch import nn
from pettingzoo.butterfly import pistonball_v6

# A custom CNN model is often needed for image-based observations
```

```
class CNNModel(TorchModelV2, nn.Module):
    def __init__(self, obs_space, act_space, num_outputs, model_config, name):
        TorchModelV2.__init__(self, obs_space, act_space, num_outputs, model_config,
name)
        nn.Module.__init__(self)
        # Assuming observations are preprocessed to (84, 84, 4) [H, W, C]
        self.model = nn.Sequential(
            nn.Conv2d(4, 32, kernel_size=8, stride=4), nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2), nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 512), nn.ReLU(),
        )
        self.policy_fn = nn.Linear(512, num_outputs)
        self.value_fn = nn.Linear(512, 1)

    def forward(self, input_dict, state, seq_lens):
        # RLlib provides obs in B, H, W, C format. PyTorch needs B, C, H, W.
        obs_torch = input_dict["obs"].permute(0, 3, 1, 2)
        model_out = self.model(obs_torch)
        self._value_out = self.value_fn(model_out)
        return self.policy_fn(model_out), state

    def value_function(self):
        return self._value_out.flatten()

def env_creator(config):
    env = pistonball_v6.parallel_env(continuous=False, max_cycles=125)
    # Preprocessing pipeline
    env = ss.color_reduction_v0(env, mode="B")
    env = ss.resize_v1(env, x_size=84, y_size=84)
    env = ss.frame_stack_v1(env, 4)
    return env

if __name__ == "__main__":
    ray.init()

    env_name = "pistonball_ppo_shared"
    register_env(env_name, lambda config: PettingZooEnv(env_creator(config)))
```

```
ModelCatalog.register_custom_model("cnn_model", CNNModel)

config = (
    PPOConfig()
   .environment(env=env_name)
   .rollouts(num_rollout_workers=4, rollout_fragment_length=128)
   .training(
        # --- PPO Hyperparameters ---
        # GAE lambda parameter
        lambda_=0.9,
        # PPO clipping parameter
        clip_param=0.2,
        # Entropy coefficient for encouraging exploration
        entropy_coeff=0.01,
        # Value function loss coefficient
        vf_loss_coeff=0.5,
        # Number of SGD iterations per training batch
        num_sgd_iter=10,
        # Total batch size collected per training iteration
        train_batch_size=512,
        # SGD minibatch size
        sgd_minibatch_size=64,
        # Model configuration
        model={"custom_model": "cnn_model"},
    )
   .framework("torch")
   .multi_agent(
        # All agents use the same policy
        policies={"shared_policy"},
        policy_mapping_fn=(lambda agent_id, ep, **kw: "shared_policy"),
    )
   .resources(num_gpus=int(os.environ.get("RLLIB_NUM_GPUS", "0")))
   .debugging(log_level="ERROR")
)

tune.run(
    "PPO",
    name="PPO_Pistonball_Experiment",
    stop={"training_iteration": 100},
    checkpoint_freq=10,
    config=config.to_dict(),
    local_dir="./ray_results"
)
```

Section 10. Off-Policy Learning: DQN with Action Masking

Deep Q-Networks (DQN) are a classic off-policy algorithm, highly sample-efficient due to their use of a replay buffer. In multi-agent settings, a key challenge is handling turn-based games where, at any given state, many actions may be illegal. For example, in a card game, an agent cannot play a card it does not possess. PettingZoo environments communicate these constraints via an "action mask" included in the observation dictionary.

A standard DQN model does not know how to interpret this dictionary structure. The solution is to create a custom model that processes the action mask and uses it to prevent the policy from choosing illegal actions. This is done by setting the Q-values (logits) of invalid actions to a large negative number, ensuring they are never selected by an argmax operation.[1]

The following script demonstrates how to train a DQN agent on the leduc_holdem_v4 environment, which includes action masking.

Python

```python
import os
import ray
import torch
from ray import tune
from ray.rllib.algorithms.dqn import DQNConfig
from ray.rllib.models.torch.torch_modelv2 import TorchModelV2
from ray.rllib.models.torch.fcnet import FullyConnectedNetwork
from ray.rllib.env.wrappers.pettingzoo_env import PettingZooEnv
from ray.rllib.models import ModelCatalog
from ray.tune.registry import register_env
from pettingzoo.classic import leduc_holdem_v4

# Custom model to handle action masking
class MaskedActionsModel(TorchModelV2, torch.nn.Module):
    def __init__(self, obs_space, action_space, num_outputs, model_config, name):
        TorchModelV2.__init__(self, obs_space, action_space, num_outputs,
model_config, name)
        torch.nn.Module.__init__(self)

        # The actual observation space is inside the Dict
        true_obs_space = obs_space.original_space['observation']

        self.q_network = FullyConnectedNetwork(
            true_obs_space,
            action_space,
            num_outputs,
            model_config,
            name + "_q_net"
        )

    def forward(self, input_dict, state, seq_lens):
        # Extract the action mask and the true observation
        action_mask = input_dict["obs"]["action_mask"]
        obs = input_dict["obs"]["observation"]

        # Compute Q-values from the true observation
        q_values, _ = self.q_network({"obs": obs})

        # Apply the mask: set Q-values of illegal actions to -infinity
        inf_mask = torch.clamp(torch.log(action_mask), min=-1e9)
        masked_q_values = q_values + inf_mask

        return masked_q_values, state

    def value_function(self):
        # DQN does not have a separate value function head
        return torch.zeros(1)
```

```python
if __name__ == "__main__":
    ray.init()

    def env_creator(config):
        return leduc_holdem_v4.env()

    env_name = "leduc_holdem_dqn"
    register_env(env_name, lambda config: PettingZooEnv(env_creator(config)))
    ModelCatalog.register_custom_model("masked_model", MaskedActionsModel)

    test_env = PettingZooEnv(env_creator({}))
    obs_space = test_env.observation_space
    act_space = test_env.action_space

    config = (
        DQNConfig()
       .environment(env=env_name)
       .rollouts(num_rollout_workers=2)
       .training(
            model={"custom_model": "masked_model"},
            # DQN-specific hyperparameters
            double_q=True,
            dueling=True,
            num_atoms=1, # Set > 1 for Distributional DQN
            noisy=False,
            train_batch_size=256,
        )
       .framework("torch")
       .multi_agent(
            policies={
                "player_0": (None, obs_space, act_space, {}),
                "player_1": (None, obs_space, act_space, {}),
            },
            policy_mapping_fn=(lambda agent_id, ep, **kw: agent_id),
        )
       .resources(num_gpus=int(os.environ.get("RLLIB_NUM_GPUS", "0")))
    )

    tune.run(
        "DQN",
        name="DQN_LeducHoldem_Experiment",
        stop={"timesteps_total": 1_000_000},
        checkpoint_freq=50,
        config=config.to_dict(),
        local_dir="./ray_results"
    )
```

Section 11. Cooperative Learning: QMIX and Agent Grouping

QMIX (Q-value Mixing) is a specialized algorithm designed for fully cooperative, partially observable multi-agent tasks. Its core idea is to learn individual agent Q-functions based on local observations, and then combine them into a joint team Q-function using a monotonic mixing network. This allows for decentralized execution (agents act based on their local view) while optimizing a centralized, team-level objective.

However, QMIX has very specific and strict requirements for its environment and configuration in RLlib, which are a common source of errors for users.

The QMIX Requirements Checklist:

1. **Agent Grouping**: All agents that are part of the team must be explicitly grouped. This is done by wrapping the environment with env.with_agent_groups(). This wrapper transforms the group of agents into a single logical agent from RLlib's perspective, with observation and action spaces that are gymnasium.spaces.Tuple of the individual agents' spaces.
2. **Homogeneous Spaces**: All agents within a single group *must* have identical observation and action spaces.
3. **Discrete Actions**: The action space for each agent must be gymnasium.spaces.Discrete. MultiDiscrete or Box spaces are not supported.
4. **Global State (Optional but Recommended)**: For optimal performance, the mixing network uses a global state of the environment. This should be provided in the info dictionary returned by the step() method, under the key "state" (or ENV_STATE from ray.rllib.agents.qmix.qmix_policy).

The following script trains QMIX on the two_step_game environment from the original

QMIX paper, highlighting the necessary configuration.

Python

```python
import os
import ray
from ray import tune
from ray.tune import register_env
from ray.rllib.algorithms.qmix import QMixConfig
from ray.rllib.examples.envs.classes.two_step_game import TwoStepGame
from gymnasium.spaces import Tuple

if __name__ == "__main__":
    ray.init()

    # 1. Group agents in the environment creator
    def env_creator(config):
        env = TwoStepGame(config)
        # Define the agent group
        grouping = {"group_1": list(range(env.num_agents))}
        # Define the tuple spaces required by with_agent_groups
        obs_space = Tuple([env.observation_space for _ in range(env.num_agents)])
        act_space = Tuple([env.action_space for _ in range(env.num_agents)])
        # Wrap the environment
        return env.with_agent_groups(grouping, obs_space=obs_space,
act_space=act_space)

    env_name = "grouped_two_step_game"
    register_env(env_name, env_creator)

    config = (
        QMixConfig()
       .environment(env=env_name, env_config={"num_agents": 2})
       .framework("torch")
       .training(
            # QMIX-specific hyperparameters
            # Type of mixing network. "qmix" or "vdn"
            mixer="qmix",
            # Size of the mixing network hidden layer
            mixing_embed_dim=32,
            train_batch_size=32,
        )
       .rollouts(num_rollout_workers=0)
       .resources(num_gpus=int(os.environ.get("RLLIB_NUM_GPUS", "0")))
    )

    tune.run(
        "QMIX",
        name="QMIX_TwoStepGame_Experiment",
        stop={"training_iteration": 50},
        config=config.to_dict(),
        local_dir="./ray_results"
    )
```

Section 12. Centralized Training with Decentralized Execution: MADDPG

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) is a popular actor-critic algorithm for settings with multiple agents, which may be cooperative, competitive, or both. Its key innovation is the use of a *centralized critic* during training. Each agent's critic has access to the observations and actions of *all* other agents, allowing it to learn a more stable and informed value function. During execution, however, each agent acts using only its own local observation and its *decentralized actor* network. This paradigm is known as Centralized Training with Decentralized Execution (CTDE).

In RLlib, MADDPG is available as a contrib algorithm, which means it may be less polished or actively maintained than core algorithms. Users have reported challenges in setting it up, particularly in combination with other policies or custom configurations. The original community examples are also somewhat dated.

The following script provides a verified setup for training MADDPG on the simple_adversary environment from the Multi-Agent Particle Environments (MPE) suite, adapted from the widely-used maddpg-rllib repository.

```
Python
```

```python
import os
import ray
from ray import tune
from ray.tune.registry import register_env
# Note: MADDPG is in rllib.contrib
from ray.rllib.contrib.maddpg.maddpg import MADDPGConfig
from pettingzoo.mpe import simple_adversary_v3
from ray.rllib.env.wrappers.pettingzoo_env import PettingZooEnv

if __name__ == "__main__":
    ray.init()

    def env_creator(config):
        return simple_adversary_v3.parallel_env(max_cycles=25)

    env_name = "mpe_simple_adversary"
    register_env(env_name, lambda config: PettingZooEnv(env_creator(config)))

    test_env = PettingZooEnv(env_creator({}))
    policies = {
        agent_id: (None, test_env.observation_space_dict[agent_id],
test_env.action_space_dict[agent_id], {})
        for agent_id in test_env.get_agent_ids()
    }
    policy_ids = list(policies.keys())

    config = (
        MADDPGConfig()
       .environment(env_name)
       .framework("torch")
       .training(
           # --- MADDPG-specific hyperparameters ---
           # Use a centralized critic
           use_local_critic=False,
           # Number of agents
           agent_id_list=policy_ids,
           # Learning rates for actor and critic
           actor_lr=1e-4,
           critic_lr=1e-3,
           # Network sizes
           actor_hiddens=,
           critic_hiddens=,
        )
       .multi_agent(
           policies=policies,
           policy_mapping_fn=(lambda agent_id, ep, **kw: agent_id),
        )
       .resources(num_gpus=int(os.environ.get("RLLIB_NUM_GPUS", "0")))
    )

    tune.run(
        "MADDPG", # Use the string name for contrib algos
        name="MADDPG_MPE_Experiment",
        stop={"episodes_total": 60000},
        checkpoint_freq=1000,
        config=config.to_dict(),
        local_dir="./ray_results"
    )
```

Part V: Analysis and Inference

Training is only half the battle. A crucial part of the development cycle involves analyzing the learning process to debug and improve agent performance, and then deploying the trained policies for evaluation or in a production application.

Section 13. Visualizing Training with TensorBoard

RLlib integrates seamlessly with TensorBoard, a powerful visualization toolkit, to log metrics during training. By default, all results from tune.run() are stored in a local directory, typically ~/ray_results/.

To launch TensorBoard, navigate to your terminal and run:

Bash

```
tensorboard --logdir ~/ray_results
```

Then, open a web browser to http://localhost:6006 to view the dashboard.

Interpreting MARL Metrics

While a standard TensorBoard tutorial might focus on aggregate metrics like episode_reward_mean, this can be misleading in a multi-agent context. For example, in a zero-sum game, the average reward across all agents might always be zero, even as the agents learn increasingly sophisticated strategies.

The real power of RLlib's logging for MARL lies in its **per-policy metrics**. For every scalar metric reported (e.g., reward, loss), RLlib also logs a separate value for each policy being trained. In TensorBoard, these will appear under tags like:

- policy_reward_mean/policy_1
- policy_reward_mean/policy_2
- total_loss/policy_1
- learner_stats/policy_1/actor_loss
- entropy/policy_2

These per-policy graphs are the primary tool for debugging and analysis in MARL. They allow a developer to answer critical questions:

- **Is one agent outperforming another?** In a competitive setting, plotting the rewards for policy_1 and policy_2 on the same axis reveals the dynamics of the game.
- **Is a specific policy failing to learn?** If the loss for policy_1 is decreasing but the

loss for policy_2 is flat or increasing, it points to a problem with that specific policy's learning process or configuration.
- **Has a policy's exploration collapsed?** A sharp drop in the entropy for a specific policy (entropy/policy_1) can indicate that it has prematurely converged to a suboptimal, deterministic strategy.

By focusing on these per-policy metrics, developers can gain a much deeper and more nuanced understanding of the complex dynamics unfolding during multi-agent training.

Section 14. Checkpointing, Restoring, and Running Inference

The final step in the development lifecycle is to save the trained policies and use them to perform actions in the environment.

Checkpointing and Restoring

RLlib's tune.run automatically saves checkpoints at a configurable frequency (checkpoint_freq).[1] These checkpoints contain the full state of the algorithm, including the weights of all policy networks.

To restore an algorithm from a checkpoint, one can use the Algorithm.from_checkpoint() class method. This requires the path to a specific checkpoint directory.

Python

```python
from ray.rllib.algorithms.algorithm import Algorithm

# Path to a specific checkpoint directory, e.g., from the output of tune.run()
checkpoint_path =
"./ray_results/PPO_Pistonball_Experiment/PPO_pistonball_ppo_shared.../checkpoint_000100"
```

```
# Restore the entire Algorithm object
algo = Algorithm.from_checkpoint(checkpoint_path)
```

This restored algo object is now ready for further training or for inference.[1]

Running Inference and Visualization

Once an algorithm is restored, it can be used to compute actions for agents in an environment. This is useful for evaluating the final performance of the trained policies or for creating visualizations of their behavior.

The following script demonstrates how to load a trained PPO agent for the pistonball environment and render its gameplay into an animated GIF, a pattern adapted from the official PettingZoo tutorial.[1]

```python
import os
import ray
import supersuit as ss
from PIL import Image
from ray.rllib.algorithms.algorithm import Algorithm
from pettingzoo.butterfly import pistonball_v6

# --- Helper functions and classes (env_creator, CNNModel) from the PPO training script must be defined here ---
#... (omitted for brevity, but required for the code to run)

def env_creator(config):
    env = pistonball_v6.parallel_env(continuous=False, max_cycles=125, render_mode="rgb_array")
    env = ss.color_reduction_v0(env, mode="B")
    env = ss.resize_v1(env, x_size=84, y_size=84)
    env = ss.frame_stack_v1(env, 4)
    return env

if __name__ == "__main__":
    ray.init()

    checkpoint_path = "PATH_TO_YOUR_CHECKPOINT" # Replace with your actual checkpoint path

    # Restore the algorithm from the checkpoint
    algo = Algorithm.from_checkpoint(checkpoint_path)

    # Create the environment for inference
    env = env_creator({})
    observations, infos = env.reset()

    frames =
    total_reward = 0.0

    while env.agents:
        # Construct the action dictionary
        actions = {}
        for agent_id in env.agents:
            # Use compute_single_action for inference
            # We must specify the policy_id if there are multiple policies.
            # Here, we use the mapping function logic from training.
            policy_id = "shared_policy"
            actions[agent_id] = algo.compute_single_action(
                observations[agent_id],
                policy_id=policy_id,
                explore=False # Disable exploration for evaluation
            )

        observations, rewards, terminations, truncations, infos = env.step(actions)

        # Render the environment and store the frame
        frame = env.render()
        frames.append(Image.fromarray(frame))

        # Accumulate rewards
        total_reward += sum(rewards.values())
        if all(terminations.values()) or all(truncations.values()):
            break

    env.close()
    ray.shutdown()

    print(f"Total reward: {total_reward}")

    # Save the frames as a GIF
    frames.save(
        "inference_gameplay.gif",
        save_all=True,
        append_images=frames[1:],
        duration=40, # milliseconds per frame
        loop=0
    )
    print("Saved gameplay to inference_gameplay.gif")
```

This process completes the full MARL development loop: from establishing a stable environment and defining the problem, to designing and training the agents, and finally to analyzing and observing their learned behavior.

Conclusions

This guide provides a comprehensive and practical roadmap for developing multi-agent reinforcement learning systems using the specified technology stack of ray[rllib] 2.9.x, PettingZoo, Gymnasium, and SuperSuit. The analysis reveals several critical takeaways for practitioners in this domain.

First, **environment stability is paramount**. The most significant initial hurdle in any MARL project is the assembly of a compatible set of libraries. The historical churn caused by the gym to gymnasium migration has left a complex dependency landscape. The version-locked configuration presented in this report is not merely a suggestion but a validated solution to preempt a wide class of common setup failures, providing a necessary stable foundation for development.

Second, **the choice of environment API and RLlib integration path is a key architectural decision**. PettingZoo's dual AEC and Parallel APIs are not interchangeable; they map directly to the fundamental interaction patterns of the problem—sequential or simultaneous. The subsequent choice of integrating with RLlib via the PettingZooEnv wrapper or a custom MultiAgentEnv implementation represents a trade-off between the ease of use afforded by the standard ecosystem and the maximal control required for non-standard or highly complex simulation environments.

Third, **mastering the multi-agent configuration is the key to unlocking RLlib's power**. The conceptual model of mapping agent instances (agent_id) to policy types (policy_id) via the policy_mapping_fn is the central mechanism for implementing all major MARL paradigms, from independent learning to parameter sharing and, most critically, to handling heterogeneous agents. A clear understanding of this mapping, combined with the explicit definition of per-policy spaces, is the definitive solution to the common challenge of managing agents with different capabilities.

Finally, **advanced MARL requires advanced tools**. For complex agent architectures involving shared information, the MultiRLModule API is the modern, canonical solution, offering a more efficient and conceptually cleaner approach than older methods. Similarly, effective debugging and analysis of MARL systems depend on moving beyond aggregate metrics and leveraging the per-policy logging provided by RLlib in TensorBoard to diagnose the intricate dynamics of multi-agent learning. By following the patterns and practices detailed in this guide, developers can effectively navigate the complexities of the MARL landscape to build, train, and analyze sophisticated intelligent agent systems.

Works cited

1. RLlib: DQN for Simple Poker - PettingZoo Documentation, accessed July 5, 2025, https://pettingzoo.farama.org/tutorials/rllib/holdem/
2. Supersuit Wrappers - PettingZoo Documentation, accessed July 5, 2025, https://pettingzoo.farama.org/api/wrappers/supersuit_wrappers/
3. How to train turn-based multi-agent game AIBot with Ray RLlib and ..., accessed July 5, 2025, https://medium.com/@kaige.yang0110/how-to-train-turn-based-multi-agent-game-aibot-with-ray-rllib-and-selfplay-8bcd20feeea7
4. ray/rllib/examples/multi_agent/different_spaces_for_agents.py at ..., accessed July 5, 2025, https://github.com/ray-project/ray/blob/master/rllib/examples/multi_agent/different_spaces_for_agents.py

5. RL Modules — Ray 2.47.1 - Ray Docs, accessed July 5, 2025, https://docs.ray.io/en/latest/rllib/rl-modules.html