Event-Driven Trainer Architecture for MARL in ROS2-Compatible Simulations

Oliver Wolf[1], Andreas Schwung[2] and Dorothea Schwung[3]

***Abstract*— Many Multi-Agent Reinforcement Learning (MARL) frameworks integrate training orchestration, agent learning logic, and environment execution into a unified pipeline. This has the effect of reducing the transparency of the data flow, embedding temporal assumptions into the control flow, and making integration of complex simulation systems–particularly robotics middleware such as ROS2–unnecessarily difficult. In such systems, the processes of step completion and observation availability are not inherently aligned with a fixed synchronous loop.**

The present paper proposes an open, event-driven trainer architecture for MARL that separates these concerns at the system boundary. The trainer emits a deterministic event order within a single orchestration loop and is responsible only for constructing temporally consistent transition tuples. The learning functionality, encompassing optimization, replay, update schedules, and algorithm-specific state, is integrated within agent implementations. The interaction between the trainer, agents, and environments is expressed through well-defined events and typed transition objects. These objects are implemented via Python dataclasses and type hints. This approach elucidates the data dependencies and facilitates runtime validation via invariants. These invariants include event-order and transition-completeness checks.

The paper provides proof-of-concept evidence for these architectural claims and includes a ROS2-compatible prototype connector in the accompanying open-source artifact.

***Index Terms*– Multi-Agent Reinforcement Learning, Event-Driven Architectures, Modular Training Frameworks, Agent-Based Systems, ROS2 Integration**

I. Introduction

Multi-Agent Reinforcement Learning (MARL) addresses distributed decision-making problems in which multiple autonomous agents interact with a shared environment. Applications range from cooperative control and resource allocation to robotic multi-agent systems [1], [2]. Despite substantial algorithmic progress [2], practical research and integration are often constrained by framework architecture: many MARL libraries remain monolithic, hide critical data flow assumptions, or rely on closed training pipelines that are hard to adapt when experiments deviate from the "standard" rollout-and-update pattern.

[1]Oliver Wolf is with Faculty of Electrical Engineering and Information Technology, Hochschule Düsseldorf University of Applied Sciences, Münsterstraße 156, 40476 Düsseldorf, Germany oliver.wolf@hs-duesseldorf.de

[2]Andreas Schwung with the Department of Automation Technology South Westphalia University of Applied Science 59494 Soest, Germany schwung.andreas@fh-swf.de

[3]Dorothea Schwung with the Center for Digitalization and Digitality (ZDD), Hochschule Düsseldorf University of Applied Sciences, Münsterstraße 156, 40476 Düsseldorf, Germany dorothea.schwung@hs-duesseldorf.de

In particular, tightly integrated trainer pipelines often conflate (i) *when* a transition is semantically complete, (ii) *when* an agent is allowed to learn from it, and (iii) *which* data is available at action selection versus update time. This makes non-standard schedules–e.g., batch updates running concurrently with action selection–hard to express without invasive changes to the orchestration code. Moreover, centralized training schemes (e.g., shared-gradient or joint-value updates, as in QMIX [3]) require explicit control over what is synchronized globally and what remains local, which monolithic pipelines often obscure via framework-specific control flow.

These constraints become acute in robotics. Integrating MARL with ROS2/Gazebo-style simulation stacks frequently requires explicit synchronization across middleware-driven observation streams, variable-latency actuation, and simulator clocks [4]. In tightly coupled training systems, temporal assumptions about when a step is "complete" are embedded into control flow; adapting them to middleware-driven execution can require invasive changes across orchestration, logging, and algorithm code. Surveys and benchmarking studies repeatedly highlight that system design choices (interfaces, coupling, hidden assumptions) have direct impact on reproducibility and extensibility in MARL research [2], [5].

This work introduces an event-driven trainer architecture that treats orchestration as a minimal, explicit system component. The trainer provides a deterministic *event ordering* and constructs temporally consistent transition tuples; it does not implement learning. Agents encapsulate all algorithmic choices (replay, optimization, update schedules), and environments remain black-box transition generators. A typed `StepData` contract and an explicit event stream make the data flow inspectable and enforceable by runtime invariants, while still allowing user-defined payload extensions without modifying the trainer core.

The main contributions of this work are summarized as follows:

- **Event-driven MARL trainer architecture:** An event-driven trainer that separates orchestration from agent learning logic and environment execution, with a deterministic event ordering within a single orchestration loop.
- **Data-level synchronization via typed contracts and runtime checks:** Construction of temporally consistent transitions using typed data contracts; invariants over event order and transition completeness can be validated at runtime without relying on wall-clock assumptions.

- **Open and modular design for heterogeneous agent architectures:** Learning algorithms, replay strategies, and update rules are fully encapsulated within agents, enabling heterogeneous multi-agent learning schemes without changing the trainer core.
- **ROS2/Gazebo integration pattern:** A middleware-compatible integration pattern that synchronizes on completed transitions; the artifact includes a ROS2-compatible prototype connector illustrating this pattern.
- **Open-source artifact:** A reference implementation with environment adapters and experiment runners corresponding to the proof-of-concept results described in this paper.

II. Related Work

Numerous libraries address MARL from different perspectives, as documented in comparative surveys and benchmarking studies [2], [5]. PettingZoo establishes a standardized interface for multi-agent environments and promotes reuse of benchmarks by defining clear semantics for agent–environment interaction [6]. Its primary focus is the environment side; training orchestration and learning logic remain external.

MARLlib and RLlib follow a more tightly integrated approach, coupling training pipelines, algorithm implementations, and distributed execution [7], [8]. While this provides strong automation and scalability, it can restrict structural flexibility when researchers need to explicitly separate semantic roles in the pipeline–data collection, action computation, and learning updates are often forced into a single coupled control path. As a result, implementing alternative information-sharing schemes, shared-gradient computations, or non-standard update schedules across agents becomes cumbersome [2], [8]. Temporal assumptions about rollout execution and update coupling are commonly embedded in the training infrastructure, which can obstruct decoupled batch updates that should not delay action freshness in middleware-driven environments [2], [7].

Earlier work by the authors explored reinforcement learning–based coordination in tightly scoped multi-robot scenarios [9], [10]. In these studies, specialized trainer logic was implemented for cooperative tasks involving a fixed number of robots with predefined interaction patterns. While effective for the targeted scenarios, the training orchestration was inherently task-specific: assumptions about agent roles, synchronization, and update timing were embedded directly into the trainer implementation.

As a consequence, these approaches do not generalize beyond the concrete experimental setups for which they were designed. Adapting them to different numbers of agents, alternative coordination schemes, or middleware-driven execution models (e.g., ROS2) would require substantial restructuring of the trainer logic itself. The present work is explicitly motivated by this limitation and aims to decouple training orchestration from scenario- and algorithm-specific assumptions.

Several works have explored the integration of reinforcement learning with ROS-based simulation and execution environments [11], [12]. These approaches typically focus on single-agent learning or rely on tightly coupled execution loops, where synchronization between action execution and observation updates is handled implicitly.

UniROS [13] proposes an asynchronous ROS-integrated reinforcement learning framework in which a single robot interacts with a learning agent under middleware-driven execution. While this design enables flexible real-time interaction, it does not provide a notion of a synchronized multi-agent transition. For MARL, however, learning updates depend on temporally consistent joint transitions across agents. Asynchronous per-agent interaction without an explicit synchronization point complicates the construction of such transitions and limits applicability to general MARL settings.

The approach presented in this paper emphasizes *data-level synchronization and explicit contracts*. Instead of providing a fully integrated training pipeline or simulation stack, the framework defines a minimal but precise interface between trainer, agents, and environment: the trainer advances only on completed transitions and exposes synchronization points as events, preserving algorithmic freedom while enabling ROS2-compatible integration and maintaining the semantic requirements of MARL.

III. System Description

To motivate the architectural design, this section first clarifies the system-level responsibilities that arise when constructing MARL transition data under heterogeneous execution models. The framework addresses an orchestration problem in MARL: the correct construction of transition tuples in the presence of multiple interacting components. Environments, agents, and side processes may execute with different internal timing models, while learning algorithms require temporally consistent transitions to ensure correct updates [14].

Conceptually, the architecture enforces a "construction kit" separation: it makes explicit *what* is needed for action selection, *when* learning is permitted to consume data, and *where* joint versus per-agent information is formed. This reduces implicit coupling and clarifies which semantic dependencies must hold at each event boundary.

From the trainer's perspective, the responsibility is *data synchronization at completed transitions*. The trainer ensures that observations, actions, rewards, and successor states are combined into complete and well-defined transition tuples, regardless of how components execute internally.

A. Reinforcement Learning Foundations and System Roles

Reinforcement Learning (RL) is characterized by an interaction loop in which an agent observes the environment, selects an action, and receives a reward and a successor observation. Learning updates require that this interaction can be represented as temporally consistent transition data.

Formally, the interaction is modeled as a Markov Decision Process (MDP) defined by $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ [14]. Learning algorithms consume transitions of the form (s_t, a_t, r_t, s_{t+1}). Correctness depends on the integrity of these tuples: each selected action must be unambiguously associated with the environment response it caused.

Algorithm classes differ in when they consume transitions. On-policy methods require data generated by the current policy, while off-policy methods decouple collection and updates via stored experience [14], [15]. Both rely on complete transitions; bootstrapped updates in particular require successor information.

In MARL, constructing consistent transition data is complicated by concurrent decision-making and shared environment dynamics. Agents may act sequentially or in parallel, and environments may process joint actions with internal scheduling. As concurrency increases, implicit assumptions about execution order become fragile. The system-level requirement remains unchanged: each agent's selected action must match the corresponding environment response so that per-agent and joint transitions are well-defined.

B. Motivation for Event-Driven Orchestration

Many benchmarks assume strictly synchronous environments that advance in discrete, blocking steps. In such settings, transition construction is implicitly aligned with control flow. Complex simulation systems break this assumption: physics engines, distributed simulators, and middleware-driven platforms such as ROS2 and Gazebo decouple simulation time from the trainer's execution [4], [11].

At the same time, modern MARL systems frequently decouple environment progression and learning updates. Off-policy and batch-based methods may update parameters independently of action selection. In multi-agent settings, action selection may occur in parallel while the environment processes joint actions under its own timing model.

These characteristics motivate an orchestration model that encodes correctness via *semantic milestones* rather than wall-clock timing: (i) a joint action is defined for a step, and (ii) a complete environment response is available so that the resulting transition tuple is well-defined. An event-driven trainer makes these milestones explicit and advances only when data dependencies are satisfied.

While ROS-integrated MARL systems have demonstrated middleware-driven learning setups [13], the focus here is a strict separation of concerns: the trainer guarantees transition integrity; learning remains entirely within agents.

IV. Event-Driven Trainer Architecture

The following section describes how the motivated event-driven orchestration model is realized in the trainer through an explicit event sequence and execution structure.

A. Event Sequence and Execution Model

Training orchestration is expressed as an ordered sequence of semantically meaningful events emitted by the trainer as shown in Fig. 1. Events denote logical milestones in the

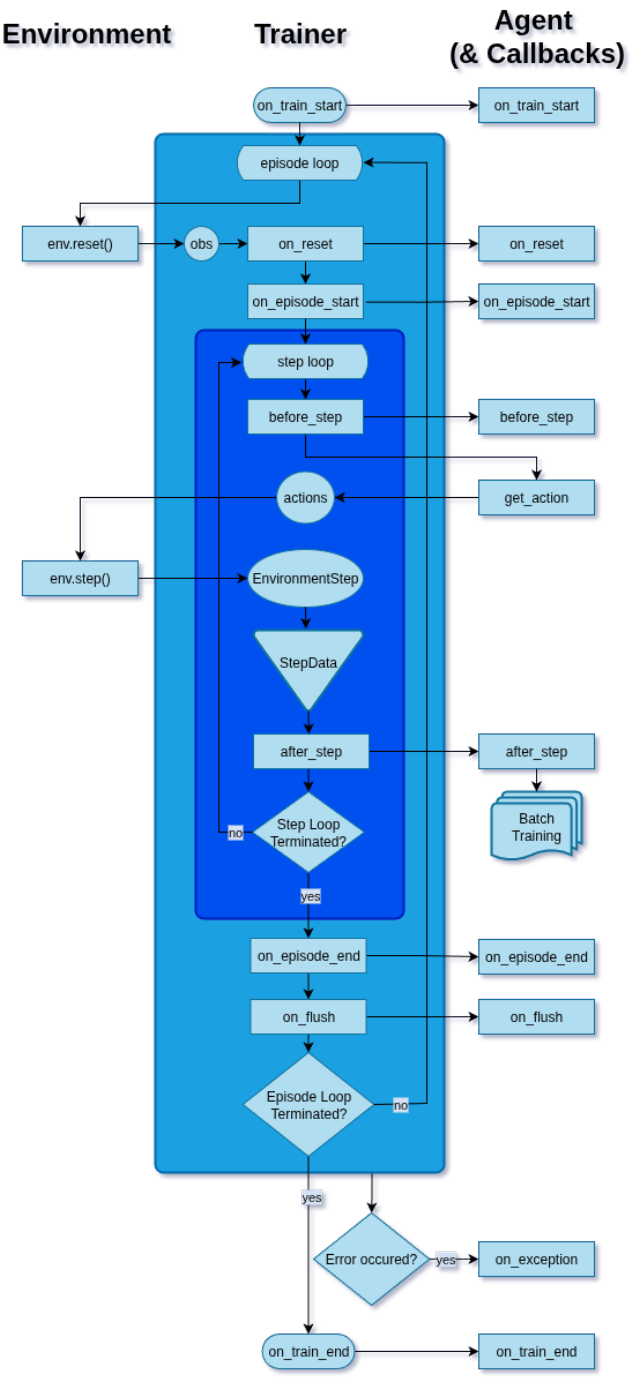

Fig. 1. Event-driven control flow of the proposed MARL trainer architecture. The trainer orchestrates training by managing the episode and step loops, while the environment is treated as a black-box component exposing reset and step transitions via a typed contract. Agents and callbacks subscribe to the same event stream and react to lifecycle events such as `on_train_start`, `on_episode_start`, and `after_step`. Within each step, actions are requested from the agents, applied to the environment, and the resulting transition is encapsulated in a `StepData` object. Learning updates (including optional replay/batch updates) are triggered after environment interaction and remain decoupled from the control flow. Errors in trainer execution or environment interaction can be reported via `on_exception`. For middleware-driven environments, blocking synchronization can be bounded by timeouts, so that missing or delayed agent responses enter this exception path rather than stalling the trainer indefinitely.

construction of environment transitions and learning updates rather than execution timing.

Training proceeds as a sequence of episodes, each consisting of repeated environment interactions. At the beginning of training, the trainer emits `on_train_start`. For each episode, the trainer emits `on_episode_start`, resets the environment and synchronizes initial observations via `on_reset`. Within an episode, the trainer emits `before_step` prior to action selection and `after_step` once a complete environment transition has been observed.

Actions are requested from the agents after `before_step` by calling their `get_action` hook and are applied to the environment as a joint action. A step is complete only after the environment returns a full transition, which is encapsulated in a `StepData` object and propagated with `after_step`. Learning updates are triggered exclusively after this event and remain decoupled from the control flow.

When termination or truncation conditions are detected, the trainer emits `on_episode_end`, followed by `on_flush` to allow components to finalize buffered state. Training concludes with `on_train_end`.

Determinism in this architecture refers to the strict ordering of trainer-emitted events and step indices within a single orchestration loop. Environment dynamics and agent policies may be stochastic or internally asynchronous, but the trainer advances only once a complete environment response is available, yielding a well-defined event sequence.

B. Trainer Responsibilities

The trainer governs progression through the event sequence. It does not implement learning logic, optimization routines, or algorithm-specific state. Its responsibilities are: (i) coordinate interaction between agents and environment, (ii) construct temporally consistent transition data, and (iii) expose completed transitions through events in a typed form.

For each step, the trainer collects agent actions associated with the current observations, forwards the joint action to the environment, and waits for a complete environment response. In the reference implementation, step completion is exposed through a blocking environment `step()` call that returns a complete transition.

Beyond step orchestration, the trainer manages lifecycle events (training start/end, episode start/end, reset) and provides controlled shutdown behavior via termination-related events. This keeps coordination centralized while maintaining clear boundaries between orchestration, learning, and observation.

C. Environment Interface and Responsibilities

The environment is the authoritative source of state transitions and rewards. It processes joint actions and returns successor observations, rewards, and termination indicators. The interface follows established multi-agent semantics as formalized by Gymnasium and PettingZoo [6], [16]: at each interaction, the environment accepts a joint action mapping over active agents and returns a complete response.

The environment is treated as a black box with respect to internal execution. It may be synchronous or asynchronous and may depend on middleware clocks or message passing. The returned transition may therefore be computed or may represent a snapshot of an otherwise concurrent simulation state. A step is complete only when the environment response is available, which is the sole criterion for advancing the event sequence.

Reset semantics are analogous: a reset is complete only once initial observations for all active agents are available. This ensures a consistent initial state representation independent of internal initialization procedures.

D. Agent Interface and Learning Responsibilities

Agents encapsulate all learning-related functionality: policy representation, action selection, replay buffer management, optimization, update rules, and algorithm-specific state. The architecture imposes no restrictions on internal learning paradigms.

Within the event sequence, agents provide actions when prompted and may produce metrics or auxiliary information. Action selection is the only agent operation that directly influences environment interaction. Agents are guaranteed access to complete per-agent observation mappings at defined synchronization points, enabling unambiguous association of selected actions with the resulting environment transition.

Learning updates are intentionally not synchronized by the trainer and may occur independently of environment execution. This supports on-policy, off-policy, and batch-based methods, as well as heterogeneous agent populations with different information requirements and update schedules.

E. Callbacks, Logging, and Side Processes

Auxiliary components subscribe to the same event stream for logging, metric aggregation, visualization or checkpointing. Callbacks are observational in the sense that their return values are ignored and they are not part of the action-selection path. Integrity checks (e.g., event-order and transition-completeness invariants) and logging are natural callback subscribers.

By exposing training milestones through a unified event mechanism, the architecture enables adding or replacing side processes without entangling them with orchestration or learning logic.

F. Conceptual ROS2–Gazebo Integration

The architectural separation and completed-transition synchronization support integration of ROS2/Gazebo-based simulation systems, where simulation time, middleware clocks, and the training process may operate on different time scales.

Because the trainer synchronizes on complete transitions, simulator execution can remain middleware-driven with variable latency: a ROS-based environment advances according to its own timing model, while the trainer proceeds only once a fully resolved transition is available. The accompanying artifact includes a ROS2-compatible prototype connector that exposes a PettingZoo-parallel interface with blocking step

completion and demonstrates the event-driven synchronization model end-to-end.

In the prototype connector, step completion is defined by middleware-driven commit milestones rather than wall-clock time: each agent publishes an action command, while observations are updated asynchronously via ROS topics. The PettingZoo-parallel wrapper blocks until all agents have reached a target commit index, then returns a complete transition for the trainer. The connector is intentionally minimal: it is designed to make the synchronization contract explicit and to demonstrate feasibility under variable-latency middleware execution, not to provide a production-hardened robotics stack.

V. Implementation Examples and Proof-of-Concept Studies

The following section empirically evaluates the architectural claims derived in the preceding sections, with particular emphasis on the conceptual ROS2/Gazebo integration described above. Specifically, the experiments assess whether the proposed event-driven trainer indeed preserves transition correctness and stable control behavior when environment execution, observation updates, and learning updates operate on different and potentially incompatible timing models.

Rather than benchmarking learning performance, the presented studies focus on system-level properties that follow directly from the architectural design: the explicit separation of orchestration and learning, completed-transition synchronization, and the absence of hidden wall-clock assumptions in control flow. All experiments are executed using the reference implementation provided in the accompanying open-source artifact.[1]

A. Evaluation Goals and Claims Under Test

The proof-of-concept studies are designed to validate the following architectural claims introduced earlier in the paper:

- **Data-level synchronization via completed transitions:** The trainer advances exclusively on semantically complete environment transitions, yielding well-defined and temporally consistent transition tuples independent of simulator or middleware timing.
- **Decoupling of learning and control flow:** Learning updates can be executed asynchronously without blocking action selection or modifying trainer orchestration, preventing stale actions in middleware-driven environments.
- **Viability for ROS2/Gazebo-style execution models:** The architecture supports environments with asynchronous observations and variable-latency execution while preserving deterministic event ordering at the trainer level.

[1]An open-source reference artifact accompanies this paper and implements all described architectural components (trainer core, event interfaces, typed data contracts, environment adapters, example Agents and corresponding experiment runners) as well as the proof-of-concept experiments. The artifact is available at `https://github.com/kids-ei/marl-eventloop-trainer`.

B. QMIX with a ROS2/Gazebo Prototype Environment

1) Rationale: This study evaluates the proposed event-driven trainer architecture under middleware-driven execution. QMIX [3] is selected as a representative centralized MARL algorithm because it relies on batch-based training updates while executing decentralized policies, making it sensitive to delays between observation availability, action selection, and environment execution.

The goal of this study is not to benchmark QMIX performance, but to assess whether the proposed trainer preserves correct transition construction and stable interaction when learning updates are decoupled from the control path in a ROS2/Gazebo-style execution model.

2) Experimental Setup: A ROS2-compatible prototype environment is used in which three TurtleBot3 agents interact in a shared simulated space. Environment progression is indexed by a monotonically increasing commit counter, which reflects completed middleware update cycles. From this, an *action_age* metric is derived that measures the delay, in commit units, between observation availability and action execution.

Two trainer configurations are compared. In the *asynchronous* configuration, QMIX batch updates are executed in a separate non-blocking CUDA thread, allowing the trainer to continue requesting actions while learning updates are running. In the *sequential* configuration, batch updates are executed inline and block the control path until training completes. Apart from the placement of the learning update, both configurations use identical trainer logic, event ordering, and environment interaction.

Experiments are conducted at control rates of 10 Hz and 20 Hz. Reported action_age values correspond to the average over all three agents, resulting in fractional values when individual agents experience slightly different commit offsets.

3) Results and Discussion: Fig. 2 shows clear qualitative differences between asynchronous and sequential training under middleware-driven execution. In the asynchronous configuration, action_age remains close to the nominal range across both control rates, indicating that actions are computed from near-fresh observations and applied with minimal delay. The stability signal predominantly remains in the normal operating regime, with deviations occurring primarily during reset and re-synchronization phases.

In contrast, the sequential configuration exhibits a consistently elevated action_age due to blocking batch updates. This effect is already visible at 10 Hz and becomes substantially more pronounced at 20 Hz, where reduced timing slack amplifies the impact of blocking behavior. Correspondingly, the stability signal shows frequent warnings and truncation events, indicating degraded temporal alignment between observation updates and control decisions.

Importantly, these effects arise without any changes to the trainer's orchestration logic or event ordering. The only difference between configurations is whether learning updates are allowed to block the control path. This confirms that the observed behavior is a consequence of update scheduling

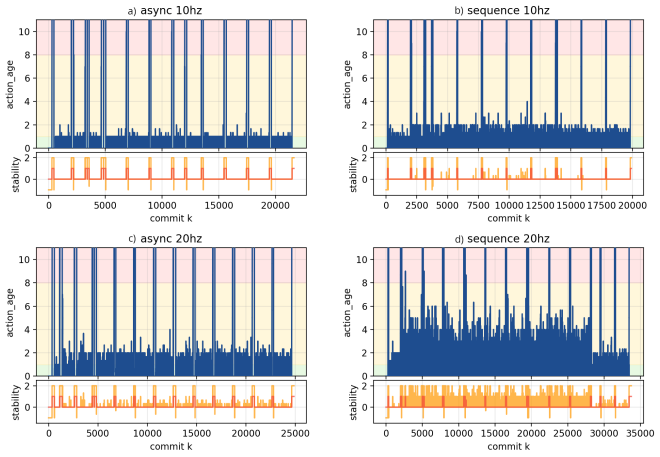

Fig. 2. QMIX in a ROS2/Gazebo prototype: asynchronous (left) vs. sequential (right) training at 10 Hz (top) and 20 Hz (bottom). Upper subplots show the average action_age over three TurtleBot3 agents (fractional values due to averaging). Lower subplots show the stability signal: orange bars indicate waiting for first action (-1), normal operation (0), warnings for action_age $>$ 1 (1), and truncation for action_age $>$ 8 (2); truncation is highlighted in red. The asynchronous, non-blocking setup yields consistently lower action_age and more stable behavior, especially at higher control rates.

rather than environment- or algorithm-specific assumptions embedded in the trainer.

Overall, this study demonstrates that decoupling learning updates from action selection is critical for stable interaction with ROS2/Gazebo-style environments. As a proof-of-concept, the results show that the proposed event-driven trainer supports centralized MARL algorithms under middleware-driven execution while preserving deterministic event ordering and completed-transition synchronization.

From an architectural perspective, the experiment highlights that the critical failure mode in middleware-driven MARL systems is not stochasticity or simulator latency per se, but implicit coupling between learning updates and the control path. By expressing synchronization exclusively through completed transitions and allowing learning to proceed asynchronously, the trainer avoids embedding timing assumptions into control flow. This separation is particularly important for centralized methods such as QMIX, where batch updates can be computationally intensive but need not influence action freshness if decoupled correctly.

VI. Conclusion

This paper introduced an event-driven trainer architecture for multi-agent reinforcement learning that makes data dependencies explicit and separates training orchestration from learning logic and environment execution. By advancing exclusively on completed transitions and exposing synchronization points as deterministic events, the trainer avoids hidden temporal assumptions commonly embedded in control flow.

The presented ROS2/Gazebo proof-of-concept demonstrates that this architectural separation enables stable and responsive interaction in middleware-driven environments, even when learning updates are computationally intensive and temporally decoupled from action selection. In particular, the results show that asynchronous, non-blocking learning updates can be integrated without modifying trainer orchestration or compromising transition correctness.

Overall, the proposed architecture provides a minimal yet expressive foundation for MARL systems that must operate under heterogeneous timing models. By shifting synchronization from wall-clock assumptions to explicit data-level contracts, it supports reproducible experimentation and clean integration with complex simulation stacks such as ROS2, while leaving algorithmic design choices entirely to the agents.

References

[1] T. T. Nguyen *et al.*, "Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications," *IEEE Transactions on Cybernetics*, vol. 50, no. 9, pp. 3826–3839, 2020.

[2] P. Hernandez-Leal *et al.*, "A survey and critique of multiagent deep reinforcement learning," *Autonomous Agents and Multi-Agent Systems*, vol. 33, no. 6, pp. 750–797, 2019.

[3] T. Rashid *et al.*, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018. [Online]. Available: https://arxiv.org/abs/1803.11485

[4] M. Aljamal *et al.*, "Comprehensive review of robotics operating system-based reinforcement learning in robotics," *Applied Sciences*, vol. 15, no. 4, p. 1840, 2025.

[5] G. Papoudakis *et al.*, "Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks," in *NeurIPS 2021 Track on Datasets and Benchmarks*, 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/a8baa56554f96369ab93e4f3bb068c22-Paper-round1.pdf

[6] J. K. Terry *et al.*, "Pettingzoo: Gym for multi-agent reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 15 032–15 043, 2021. [Online]. Available: https://arxiv.org/abs/2009.14471

[7] E. Liang *et al.*, "Rllib: Abstractions for distributed reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018, pp. 3053–3062. [Online]. Available: https://proceedings.mlr.press/v80/liang18b.html

[8] S. Hu *et al.*, "Marllib: A scalable and efficient multi-agent reinforcement learning library," *Journal of Machine Learning Research*, vol. 24, no. 77, pp. 1–27, 2023. [Online]. Available: https://jmlr.org/papers/v24/23-0378.html

[9] D. Schwung *et al.*, "An application of reinforcement learning algorithms to industrial multi-robot stations for cooperative handling operation," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 194–199.

[10] A. Schwung *et al.*, "Cooperative robot control in flexible manufacturing cells: Centralized vs. distributed approaches," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1, 2019, pp. 233–238.

[11] N. Gonzalez Lopez *et al.*, "gym-gazebo2, a toolkit for reinforcement learning using ros 2 and gazebo," 2019, arXiv:1903.06278. [Online]. Available: https://arxiv.org/abs/1903.06278

[12] M. Martini *et al.*, "Pic4rl-gym: a ros2 modular framework for robots autonomous navigation with deep reinforcement learning," 2022, arXiv:2211.10714. [Online]. Available: https://arxiv.org/abs/2211.10714

[13] J. Kapukotuwa *et al.*, "Uniros: Ros-based reinforcement learning across simulated and real-world robotics," *Sensors*, vol. 25, no. 18, p. 5679, 2025.

[14] R. S. Sutton *et al.*, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[15] T. Haarnoja *et al.*, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018. [Online]. Available: https://arxiv.org/abs/1801.01290

[16] M. Towers *et al.*, "Gymnasium: A standard interface for reinforcement learning environments," 2024, arXiv:2407.17032. [Online]. Available: https://arxiv.org/abs/2407.17032