B.Comp. Dissertation

Competition Platform for AI Tasks

By

Tan Yuanhong

Department of Computer Science

School of Computing

National University of Singapore

2021/22

B.Comp. Dissertation

Competition Platform for AI Tasks

By

Tan Yuanhong

Department of Computer Science

School of Computing

National University of Singapore

2021/22

Project No: H247080
Advisor: Dr. Akshay Narayan, Prof. Leong Tze Yun

Abstract

Entering the second decade of the 21st century, machine learning (ML) and artificial intelligence (AI) are the new must-haves for computer science education. Just like traditional algorithms, AI education requires evaluation of AI algorithms taught in class. However, while the data-oriented tasks like classification and regression have well-adopted platforms such as Kaggle, simulation-oriented tasks that are most common for reinforcement learning (RL) algorithms are yet to have a feature-complete online judging solution. This project aims to build a complete RL algorithm evaluation solution that is extensible, scalable, and easy-to-use. It covers four critical components of judging RL algorithms: a simulation environment framework, an auto-grading framework, a scalable and secure sandbox for arbitrary code execution, and finally a modern web application as the entry point. Besides comprehensive documentation and many quality-of-life new features, this project greatly improves the performance over existing internal platform (aiVLE 1.0) with a task queue system that not only distributes tasks fairly, but also dynamically adapts to the load of each worker node thanks to its resource awareness. Experimental results show that the new system provides equal task distribution among worker nodes and nearly three times of resource utilization improvement over aiVLE 1.0 ($\approx 85\%$ vs $\approx 30\%$).

Project Nature:
Implementation, Experimentation, Simulation

Subject Descriptors:
C.2.4: Distributed Systems
I.2: Artificial Intelligence

Project Keywords:
Artificial Intelligence
Machine Learning

Implementation Software and Hardware:
Ubuntu 20.04, Firejail 0.9.68, Python 3.8, Django 4.0, Celery 5.2

Acknowledgement

Without the support of many people, this final year project would not have been possible. I would like to thank my advisors Dr. Narayan and Prof. Leong, who offered an unimaginable amount of guidance throughout this project with great kindness. Also thanks to School of Computing for their compute cluster nodes that made the experiments possible.

Finally, I would like to thank my friends and family for their support and love during the hard times of the COVID-19 pandemic. And special thanks my significant other for her patiently listening to my annoyingly frequent mentions of this project.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing popularity in both artificial intelligence (AI) research and industry application, more and more institutions are considering AI education as an integral part of their computer science or even general education curriculum. However, in spite of the algorithmic nature of AI education, there are surprisingly few platforms and tools to host AI competitions/assignments and automatically grade the submissions. Unlike traditional algorithms and data structure courses that have many online judges available (to name a few, Codeforces, Kattis, UVa Online Judge), AI courses often rely on arbitrary grading scripts or in-house solutions to host their assignments/contests. There are well-known data science competition platforms like Kaggle that are suitable for prediction/classification tasks, but the area of reinforcement learning (RL) tasks[1] remains surprisingly untouched. Therefore, this project tries to fill the gap of evaluating algorithms that solve interactive tasks. We aim to provide a complete solution from building RL[2] environments, writing test cases, to eventually hosting these tasks on a massively scalable platform.

[1]To be precise, here tasks mean environments that are typically used for RL algorithms. The differentiating characteristic of such tasks is that they are interactive (in contrast to comparing the output against the ground truth like what Kaggle and other traditional online judges do).

[2]The evaluation model we use is suitable for almost all AI/ML tasks (including classification and regression), but we will focus on RL in this work.

1.1 Background

1.1.1 Reinforcement Learning Tasks

Reinforcement learning (RL) is a branch of machine learning that maps observations (possibly with history as well) to actions such that certain metrics are maximized (Sutton & Barto, 2018). The metrics which agents try to maximize are called a reward or reinforcement, thus the name "reinforcement" learning. In a broader sense, any AI task is RL task: we can generalize any input as the observation, any output as the action and any target as the reward. But for the sake of simplicity, in this report, by RL task we mean sequential decision problems (SDP), where the outcome of agent's decision depends on a sequence of decisions (Russell, 2010).

A sequential decision problem is fully described by state space, action space, transition model and a reward function (Russell, 2010). A transition model is a function that maps (current state, history states, action) to a probability of reaching any possible state. If the transition model does not depend on history state, we call it Markovian as the probability only depends on the current state instead of the history of all earlier states. A reward function maps current state[3] to a numerical reward.

The solution to a sequential decision problem is called a policy. It is a function that maps current state (and sometimes history states) to an action. We always assume that the domain of the policy function is the entire observation space so that the agent knows what action to take under any possible circumstance. The goal of RL algorithms is to come up with such policy functions that maximize their expected reward under different settings (e.g., unknown underlying transition model, partial observability, etc.).

From the definition of sequential decision problems, we can summarize the following execution loop of any typical SDP:

[3]Reward function may also depend on action and next state, but one that only depends on the current state is the most common variant, and this difference is not fundamental.

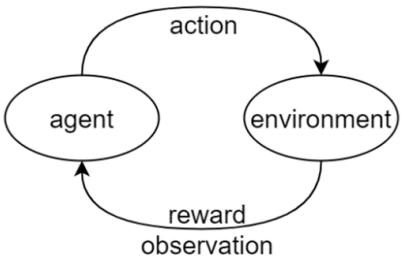

Figure 1.1: Action-observation-reward Loop

In other words, the environment implements the transition model $P(s_j|s_i, a)$ and reward function $r(s_i)$ and returns the next observation o_j and reward r_j to the agent, after which the agent takes another step of action a_{j+1} based on $[o_j, o_{j-1}, \cdots, o_0]$ and $[r_j, r_{j-1}, \cdots, r_0]$. Note that o_i and s_i may or may not be the same depending on whether the environment is fully observable.

1.2 Related Work

1.2.1 Reinforcement Learning Environment Frameworks And Collections

RL research, like many computer science topics, requires concretizing the theory with implementation and experiments as they are equally important. That is where RL environment frameworks and collections like the Arcade Learning Environment (ALE) (Bellemare et al., 2012) and OpenAI Gym (Brockman et al., 2016) come into play. Such an effort is incredibly important for the rapid growth of the RL community: it not only saves researchers a lot of emulation development time, but also provides benchmark environments to evaluate RL tasks similar to how CIFAR-10 (Giuste & Vizcarra, 2020) provides benchmark datasets in image classification.

Besides collecting many classical environments as a benchmark package, an important contribution of OpenAI Gym is its widely accepted application programming interface (API). This common interface (see Code 1.1) proves to be sufficient for not only hundreds of self-contained environments included in OpenAI Gym, but also many more third-party environments. When it comes to turning retro video games to RL environments, ALE eventually adopted Gym's API

and merged into Gym. Moreover, Gym Retro (Nichol et al., 2018) adds even more emulated systems such as Nintendo Game Boy and Sega Genesis to the Gym ecosystem. Therefore, it is safe to say that following this interface gives us enough flexibility to implement most of the desired environments.

From the source code of OpenAI Gym, a compatible environment needs to implement the methods and attributes shown in Figure 1.2:

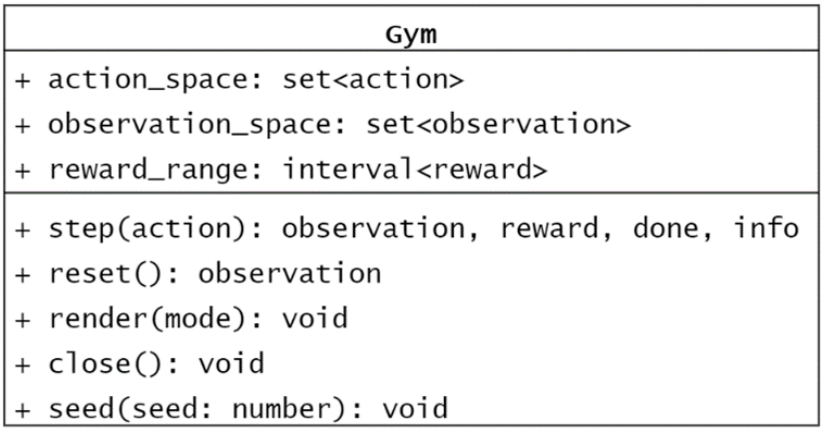

Figure 1.2: OpenAI Gym Class Diagram

These abstractions (i.e., `action_space`, `observation_space` and `step(action)`) are in line with the academic definition of sequential decision problems (SDP). Thus, in the following example of a typical OpenAI Gym agent program (Code 1.1), there is also a loop structure similar to the action-observation-reward loop in SDPs (Figure 1.3):

```
import gym

env = gym.make('CartPole-v0')
for i_episode in range(100):
    observation = env.reset()
    for t in range(100):
        action = decide(observation, reward)
        observation, reward, done, info = env.step(action)
env.close()
```

Code 1.1: OpenAI Gym Agent Example

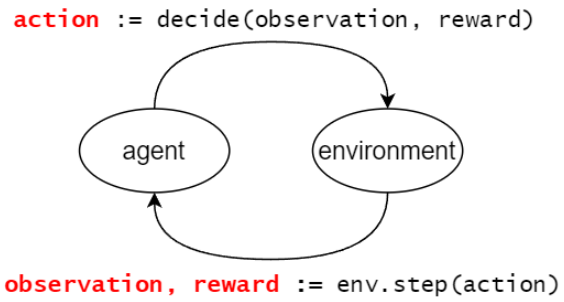

Figure 1.3: Action-observation-reward Loop in OpenAI Gym

Since this project is supposed to be designed for future multi-agent task support from ground up, we also looked into the multi-agent environment support of OpenAI Gym: although authors of OpenAI Gym listed it as a future direction when they published the paper in 2016, as of 2022, there is still no official support for multi-agent tasks. There are some third-party environments that support multi-agent tasks, like ma-gym (Koul, 2019). These implementations usually convert multi-agent observation, reward, and action into a vector, length of which is equal to the number of participating agents in the environment.

1.2.2 Online Judging System

Online judges first emerged to address the problem of automatically grading programming assignments conveniently, fairly and securely (Kurnia et al., 2001). Without any manual intervention after the system is configured properly, online judge systems need to handle 1) receiving submissions, 2) compiling and executing source codes, 3) comparing outputs against the correct answers, and 4) logging the result. For fairness, RAM and CPU usage will be monitored and limited. Usually, there is a hard time limit after which any submission will be forced to terminate. For security, execution of arbitrary submission needs to be contained in a safe environment with heavily restricted privilege levels such that only necessary operations are allowed. Any other attempt (e.g., accessing the network, read/write on filesystem, etc.) will be blocked and preferably reported to the system administrator.

Among all the challenges in building a successful online judge, security is probably the most technically difficult aspect. Some online judges were used in an exam setting where students had

to use dedicated terminals for access, and a system administrator is involved in monitoring the security issues (Arnow & Barshay, 1999). For automated security measures, there are generally two approaches: 1) source code or binary scanning, and 2) operating system level access control.

Source code or binary scanning has no runtime performance penalty, but it requires an exhaustive check on all possible malicious tokens/system calls. It is unrealistic to have a perfectly secure source code scanner and every supported language requiring a new set of scanning rules makes this approach non-sustainable (Kurnia et al., 2001). Besides, it is impossible to scan the binary for programs written in a dynamic programming language such as Python as they execute inside an interpreter.

Access control based on operating system kernel security features is much more foolproof and widely applicable. Such techniques are also called OS-level virtualization because from the perspective of programs inside a "namespace", it can only see partial filesystem and devices that are assigned to this "namespace". Since the introduction of user namespaces in Linux kernel 3.8, containerization technology has become increasingly popular to provide isolation between processes while being able to share nearly all hardware resources with minimal performance overhead (Jain, 2020).

With the rapid growth of cloud and distributed computing, there are also novel online judges that attempt to adopt these cutting-edge technologies. A distributed online judge architecture tries to address the problem of scalability: an online judge needs to execute student submissions, and with more students and courses, computation required grows linearly. It is difficult to scale vertically (making computers more powerful) but easy to scale horizontally (having more computers) especially in recent years with the emergence of cloud computing (Barroso et al., 2013). Evaluating AI tasks has an even stronger demand for horizontal scalability as neural networks or sophisticated searching algorithms are notoriously computationally heavy. Notable implementations of distributed online judge include MetaOJ (Wang et al., 2021), which separates data storage, web application and judgers into three components that can be deployed to the cloud independently (Figure 1.4).

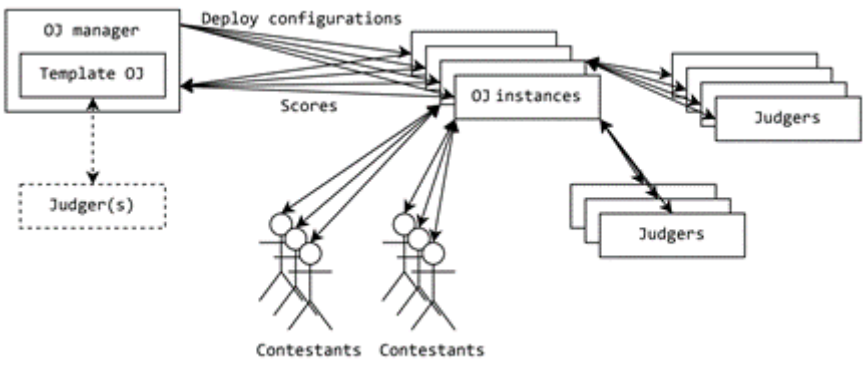

Figure 1.4: MetaOJ Architecture (Wang et al., 2021)

1.2.3 ML/RL Benchmark Platforms

In contrast to competition platforms on the education side, researchers are more interested in a benchmark platform that provides a collection of tasks to compare different RL algorithms. Such benchmarks are crucial for reproducible and verifiable research in RL. According to Vanschoren and Blockeel (2009), an interpretable and reproducible experiment consists of four parts: algorithm, parameters, evaluation method and dataset. For a benchmark, evaluation method and dataset are fixed while contributors will compare different algorithms and parameters. In the context of RL benchmark, this means packaging the simulation environment and evaluation metrics behind a simple set of common interfaces for both training and testing.

However, as machine learning models (along with their training processes) get much more sophisticated, reproducibility is affected by not only those four points – random seeds, version of external libraries and even specific model of CPU/GPU could greatly affect the result of the same code (Isdahl & Gundersen, 2019). Therefore, according to "Ten Simple Rules for Reproducible Computational Research" by (Sandve et al., 2013), for an RL benchmark platform to be reproducible, it should also: 1) have strict versioning[4] of environments and evaluation metrics; 2) use deterministic methods whenever possible and fix random seeds throughout the process; and 3) have a record of all raw data.

Considering the criteria listed above, OpenAI Gym (Brockman et al., 2016) is possibly one

[4]Meaning "any" change to the implementation of either environment or evaluation metrics should be reflected in a different version number. This also includes changes to the version of external dependencies.

of the closest to an ideal benchmark: it explicitly embraces strict versioning, has an interface to fix random seeds, and includes many well-accepted tasks. Unfortunately, it only handles the simulation side of RL benchmark while leaving the evaluation to the users. For a complete RL evaluation framework to be used in teaching and research, we also need to provide the common scoring criteria.

1.2.4 AI Competition Platforms

aiVLE (AI Virtual Learning Environment) is the inspiration and foundation of this project. It is an RL task evaluation system built by the CS4246 teaching team since 2019. We will call the old aiVLE as aiVLE 1.0 henceforth. Do note that aiVLE 1.0 refers to the system as a whole — it does not refer to any specific component (e.g., web, runner, runner-kit). aiVLE 1.0 addresses the need for evaluating an OpenAI Gym agent in a GPU-accelerated environment, the architecture of which is illustrated in Figure 1.5:

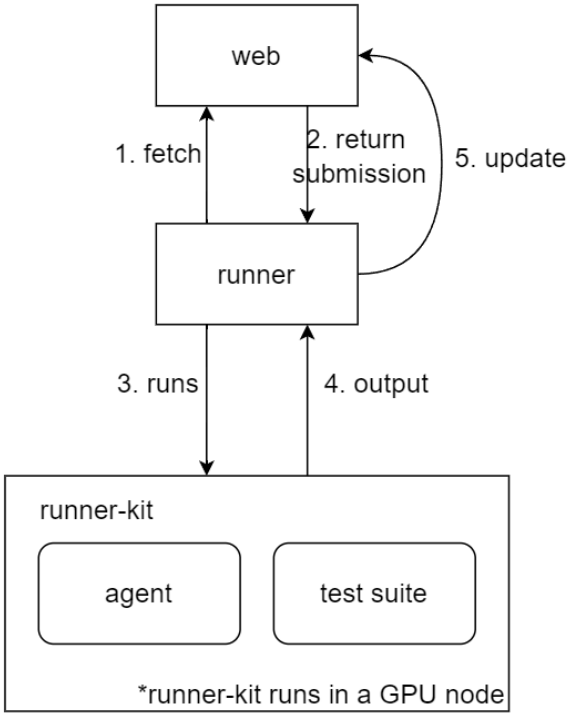

Figure 1.5: Architecture of aiVLE 1.0

Due to urgency and time constraints, the author of aiVLE 1.0 prioritized "making it work"

over "doing it right", which leaves us four pain points[5] that impact its **extensibility**:

1. Lack of documentation. For example, 1) deployment documentation for the runner instances (programs that evaluate student submissions), 2) architecture overview of each component and the entire system, and 3) inline documentation such as function usage, class definition. Without these documentation, new developers and users will take a much longer time to adopt and contribute to the system.

2. Need more coverage of software engineering best practices such as proper level of abstraction, avoiding lengthy functions/files for interpretability, etc.

3. No frontend-backend separation. Navigating and understanding the codebase is difficult as there is no clear boundary between frontend and backend code. Moreover, using JavaScript for frontend interactivity would make the codebase look even more convoluted.

4. No multi-agent support.

As for **scalability**, lack of proper task queue makes it difficult to scale aiVLE 1.0 to more machines: its workers get evaluation tasks by requesting server for the latest submissions *periodically*. Worker-side polling has risks for traffic congestion and race condition, and the consequent lack of load balancing further hurts aiVLE 1.0's potential to scale well with more workers.

[5]The list here is not exhaustive: more limitations and considerations will be discussed in Design and Implementation, along with solutions to these pain points.

Chapter 2

Project Objective

2.1 Problem Statement

The goal of this project is to provide a complete solution for RL algorithm evaluation that is extensible, scalable, and easy-to-use. It consists of four tightly integrated components (collectively called aiVLE 2.0[6]):

1. aiVLE Gym: An OpenAI Gym (Brockman et al., 2016) compatible RL environment framework with agent-environment separation and official support for multi-agent tasks.
2. aiVLE Grader: An auto grading framework for aiVLE Gym tasks.
3. aiVLE Worker: A security sandbox to execute arbitrary code submissions safely in a resource restricted environment. And a massively scalable worker client for evaluation.
4. aiVLE Web: A web application for hosting RL competitions.

2.2 Motivation

Courses like CS4246, which teach AI, and are different from traditional programming/algorithm courses, need a system to automate the process of collecting and evaluating programming as-

[6]Do note that aiVLE 2.0 refers to the new system architecture **as a whole** – every sub-project has an independent versioning that does not necessarily share the major revision number 2 because components like aiVLE Gym and aiVLE Grader do not exist in aiVLE 1.0.

signments on AI tasks. aiVLE 1.0 provides instant feedback on programming assignments that require varied computational power such as GPU or other specialized processing units. However, as mentioned in Section 1.2.4, aiVLE 1.0 has many pain points. For example, it lacks extensibility (e.g., does not support multi-agent task), scalability (e.g., no concurrency safety for many workers), and documentation for more courses to make use of the platform. As for another similar platform called Botzone (Zhou et al., 2018), although it is built for external users, limitations like no GPU support and no common interface make it infeasible for many tasks.

Secondly, an open source RL competition platform with good documentation and software engineering best practices has potential impact beyond education purposes. On the one hand, such a platform could also be useful for benchmarking AI algorithms – the consistency required for grading assignments is perfect for comparing research as well. On the other hand, examples like AlphaGo (Silver et al., 2016) proves the effectiveness of combining supervised learning from past match data with unsupervised RL – match history collected on the platform could be useful for training models for corresponding tasks.

Lastly, there is demand for such a platform. Both CS3243 and CS4246 have assignments on RL algorithms, and many non-RL tasks such as brute force or informed search, satisfiability problems, can be solved using similar execution loops. It is a safe bet that such a platform will benefit many AI courses (including CS2109S and CS3244) by using it alongside platforms like Kaggle to cover most AI algorithm evaluation scenarios.

2.3 Roadmap

There are three stages planned for this project. During the first semester, we delivered satisfactory results for the first two stages. During the winter break, we conducted real-world deployment and testing to evaluate the feasibility of advancing the project to the third stage. During the second semester, we focused on fixing problems found during the test, and delivering features as mentioned in the third stage.

2.3.1 Stage 1: Framework

Before developing the web platform for hosting the competitions, we first need frameworks for 1) creating environment and 2) evaluating agent's performance – aiVLE Gym and Grader respectively. These frameworks are independent from the web platform, therefore could be used separately for other education or research purposes as well. Besides these two platform-independent frameworks, a security sandbox named aiVLE Worker, is also necessary for hosting competitions. These three components will be implemented in the mentioned order as the latter have dependency on the former.

2.3.2 Stage 2: Platform

With the frameworks ready, the second stage is to have a web platform that achieves basic online judge functionality, which includes but is not limited to: 1) user registration and authentication, 2) creating/joining courses, 3) creating/modifying/submitting tasks, 4) showing the evaluation result. The target of this stage is to have a *minimum viable product* that CS4246 can use to host single-agent assignments with GPU-accelerated evaluation.

2.3.3 Stage 3: Advanced Solution

Stage 1 and 2 provide a solid foundation for further extension and exploration, but primarily focus on delivering features that are necessary (thus **minimally** viable) for CS4246 teaching. Stage 3 aims to further extend the use case of our web platform. First, we will perform a real-world deployment and stress test of the system, which 1) validates the stability and performance gain of the new architecture, and 2) irons out unexpected behaviors and discrepancies in the deployment instructions. Second, we will improve the documentation of the source code and use cases, which is crucial for the maintainability and upgradability of this project. Lastly, we will add advanced features that are so-called "quality-of-life updates"[7] but are achievable within the time constraint (e.g., Sections 3.3.3 for resource awareness, 3.4.3 for load balancing).

[7]There are also many other nice-to-have features that did not appear in this report, like more user-friendly frontend and course administration tools. To keep the report relatively concise, we only cover features that are deemed significant or technically challenging.

Chapter 3

Design and Implementation

As mentioned in Section 2.1 Problem Statement, there are four components in aiVLE 2.0: aiVLE Gym, Grader, Worker and Web. As a high-level overview, Figure 3.1 shows the relationship between the components: the system can be roughly divided into client-side (where users write and run the agents locally) and server-side (where the platform hosts the competition and evaluates the submissions in a security sandbox). On the client side, we have **aiVLE Gym** (Section 3.1) responsible for simulation. **aiVLE Grader** (Section 3.2) calls the agent implementation to interact with the environment and records the simulations. On the server side, student submissions are evaluated inside the security sandbox of **aiVLE Worker** (Section 3.3), ensuring fair resource allocation and secure code execution. We also have a worker cluster consisting of many worker nodes, communicating with **aiVLE Web** (Section 3.4) in an orchestrated manner. This cluster is designed to be fault tolerant and massively scalable. And of course, aiVLE Web also provides a frontend that you can access from a browser.

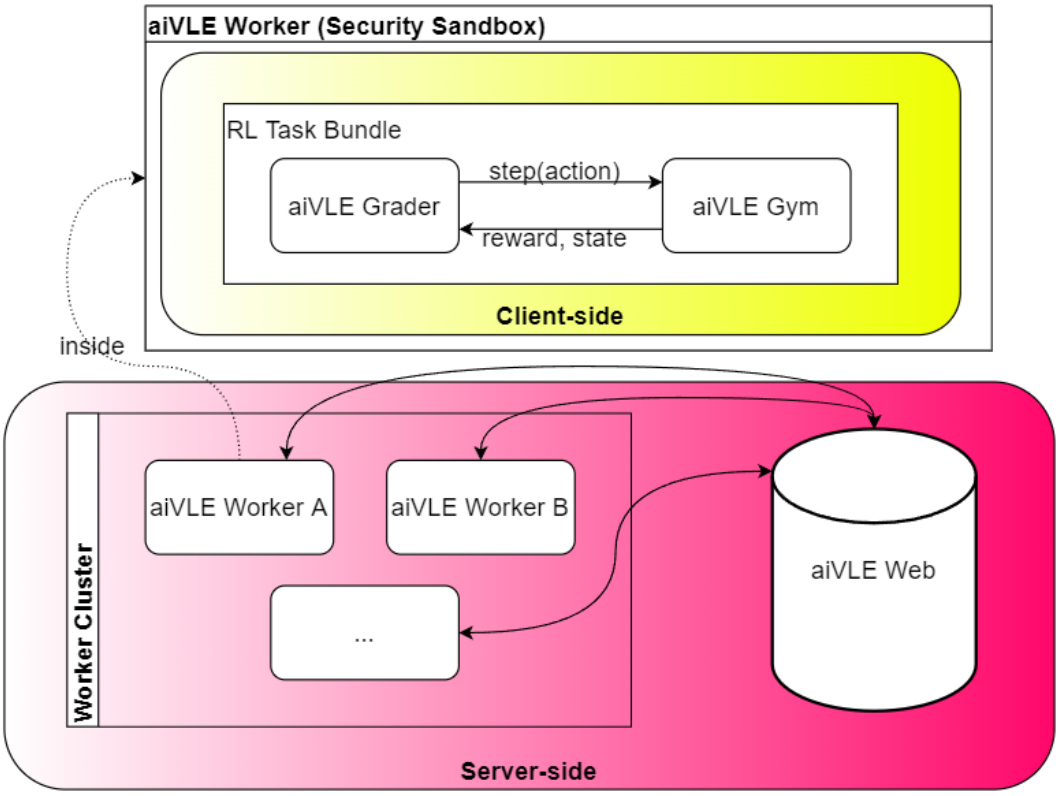

Figure 3.1: aiVLE 2.0 Architecture Overview

In this chapter, we will discuss about the design and implementation of all these four components in detail, the order of which is also the chronological order of development: the later components may have dependencies on the earlier ones, and the earlier components generally do not depend on the later ones. For example, aiVLE Gym can be used as general-purpose multi-agent environment framework outside of the aiVLE Web platform.

3.1 aiVLE Gym - Separating Agents from Environment

I have released a stable version of aiVLE Gym with all planned features implemented. The source consists of ∼1K lines of code, including several example environments and full documentation. The package is published to PyPI (https://test.pypi.org/project/aivle-gym) so that aiVLE Gym can be installed and imported like any other Python package. A brief user guide can be found at https://edu-ai.github.io/aivle-docs/user-guide/aivle-gym/.

3.1.1 Motivation

aiVLE Gym makes multi-agent competition possible by separating agents from the environment. In a two-agent OpenAI Gym (Brockman et al., 2016) task, we write agent code as shown in Code 3.1:

```
env = gym.make("PongDuel-v0") # Two-player Ping Pong game

for ep_i in range(100):
    done_n = [False for _ in range(env.n_agents)]
    ep_reward = 0
    obs_n = env.reset()
    env.render()
    while not all(done_n):
        action_0 = decide_0(obs_n[0], reward_n[0])
        action_1 = decide_1(obs_n[1], reward_n[1])
        action_n = [action_0, action_1]
        obs_n, reward_n, done_n, info = env.step(action_n)
        ep_reward += sum(reward_n)
        env.render()
    print('Episode #{} Reward: {}'.format(ep_i, ep_reward))

env.close()
```

Code 3.1: OpenAI Gym agent example

Note that in a multi-agent scenario, `observation`, `reward` and `done` are all vectors - each element corresponds to one of the agents. Similarly, when you call `env.step()`, you should provide `actions` for every agent in this simulation. Such design is acceptable when we perform these multi-agent experiments offline in a controlled manner. However, in a competition setting, when it comes to multi-agent tasks, you cannot make decisions for your opponent agents. Therefore, separating agents from the environment simulation is necessary. From the perspective of each agent, it is just like a single-agent environment – the only difference is that the environment is affected by actions taken by other agents as well. Figure 3.2 and Figure 3.3 show the architectural differences between *OpenAI* Gym and *aiVLE* Gym (each colored box represents a separate process; solid arrows represent inter-process or network communication):

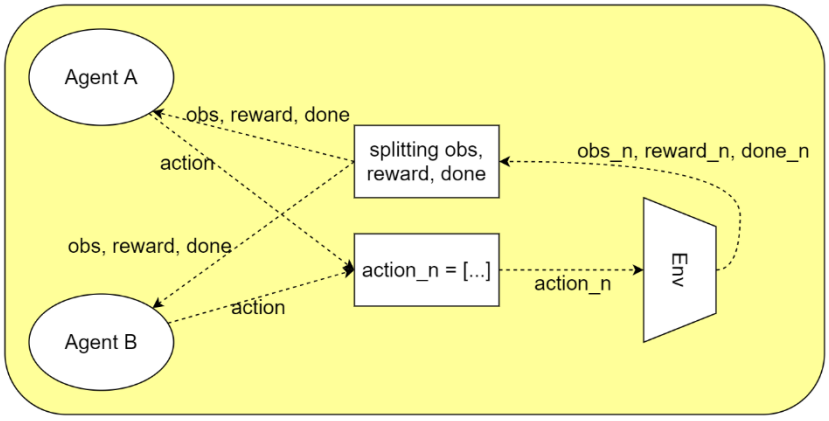

Figure 3.2: Multi-agent Architecture in OpenAI Gym

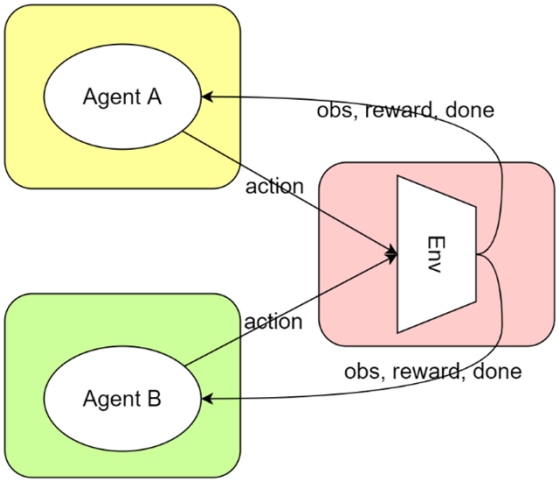

Figure 3.3: Multi-agent Architecture in aiVLE Gym

3.1.2 Design Goal

Unless mentioned otherwise, all design goals in this chapter are achieved in the released implementation.

The design goal of aiVLE Gym is to keep full compatibility with OpenAI Gym on the agent side. On the environment side, it lets you convert from existing OpenAI Gym environments with minor adaptation. More specifically:

In single-agent case, on the agent side, traditional environment (simulation happens within agent process) and aiVLE environment (simulation happens outside of agent process) should be interchangeable. On the environment side, author can reuse existing OpenAI Gym compatible

environment by implementing a serializer that serializes `action`, `observation`, and `info` to JSON compatible objects (Code 3.2).

```
class CartPoleEnvSerializer(EnvSerializer):
    def action_to_json(self, action):
        return action
    def json_to_action(self, action_json):
        return action_json

    '''Because numpy.array objects are not JSON-serializable by default,
    we provide custom methods to marshal/unmarshal observations.
    As shown in this example, if action/observation/info are JSON-serializable
    to begin with, you just return the original value.
    '''
    def observation_to_json(self, obs):
        return obs.tolist()
    def json_to_observation(self, obs_json):
        return numpy.array(obs_json)

    def info_to_json(self, info):
        return info
    def json_to_info(self, info_json):
        return info_json
```

Code 3.2: aiVLE Gym Environment Serializer Example

In multi-agent case, on the agent side, the API behaves just like normal single-agent OpenAI Gym environment. On the environment side, author can reuse existing ma-gym environment (Koul, 2019) by implementing a serializer along with several metadata fields.

3.1.3 Agent-Environment Communication

Since agents and environment are separated, we need an inter-process communication channel between them. aiVLE Gym uses a lightweight yet high-performance messaging library ZeroMQ (Akgul, 2013), which has comprehensive support for many synchronous and asynchronous messaging patterns that are essential to this project. There are two primary challenges for multi-agent tasks when it comes to agent-environment communication:

1. The judge should receive and respond to requests asynchronously - it needs to wait for all agents' actions before stepping forward in the environment, then decide on what observa-

tions/rewards to respond to each of the agents.

2. Certain operations (e.g., resetting the simulation) must be performed strictly once for each episode, but since each agent will initialize the episode on their own, judge-side will unavoidably receive multiple requests.

To summarize, the judge-side environment needs to implement a "barrier" synchronization mechanism that not only realizes synchronous rendezvous of agent requests, but also performs additional tasks upon the "first-comer" and "last-leaver".

Therefore, we propose the deterministic finite automaton (DFA) as shown in Figure 3.4:

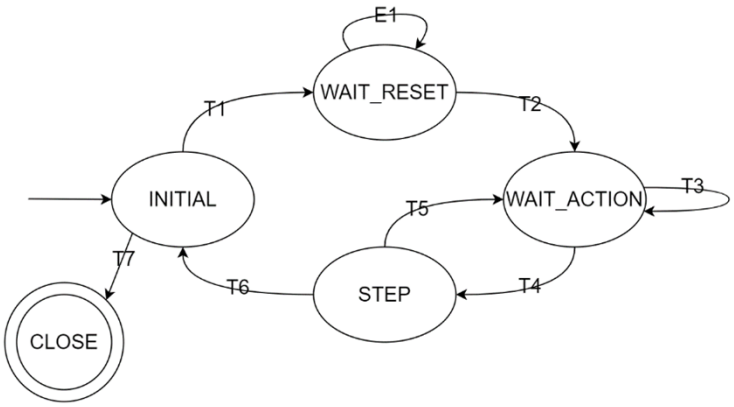

Figure 3.4: aiVLE Gym Multi-agent Environment Communication Automaton

This DFA is key to keeping complicated communication details transparent to both agent and environment logic – both can write common synchronous code while the framework deals with the underlying asynchronous logic. Details of this DFA can be found in the Appendix B.1.

3.2 aiVLE Grader - Evaluating Agents Using Test Suites

I have released a stable version of aiVLE Grader with support for 1) OpenAI Gym, 2) aiVLE Gym single-agent, and 3) aiVLE Gym multi-agent environment. The codebase consists of ∼600 lines of framework code and ∼300 lines of example test suites for all three supported use cases. Similar to aiVLE Gym, the package is released to PyPI (https://test.pypi.org/project/aivle-grader) for in-production usage. A brief user guide can be found at https://edu-ai.github.io/aivle-docs/user-guide/aivle-grader/.

3.2.1 Design Goal

Unlike competitive programming style problems or machine learning prediction tasks, evaluating RL agents is much more complicated than comparing students' output against a standard answer. Fortunately, with the common programming interfaces provided by OpenAI/aiVLE Gym, we may create a framework that standardizes/modularizes the initialization, execution, and conclusion of RL agent evaluation. The ultimate goal of this framework, when writing a grader for agents in any OpenAI/aiVLE Gym environment, is to:

1. Make the built-in components complete enough such that for most use cases using built-in ones would be sufficient;

2. Make each component self-contained without complicated inter-dependencies (i.e., following the single responsibility principle) when writing a custom component.

3.2.2 Key Abstractions

There are three key abstractions to aiVLE Grader: *agent*, *evaluator*, and *test case* as summarized in Figure 3.5.

Agent only has two methods: `reset` to reset internal states, `step` to return an action from provided observation. It is flexible enough to allow agents to memorize the history, whilst restrictive enough to prohibit agents from modifying the inner-workings of the environment.

Evaluator records the entire execution process and produces a score when the session terminates. It utilizes the common pattern of most RL tasks (i.e., the action-observation-reward loop): each session consists of many episodes, and each episode consists of many discrete steps. By inserting hook functions to these critical points, an evaluator practically records everything about the evaluation session.

Test case is a bootstrap for evaluation sessions. It wraps *agent*, *environment*, and *evaluator* along with necessary initialization parameters into an object with one simple `evaluate` method. It also offloads certain chore (e.g., time limit) away from the user.

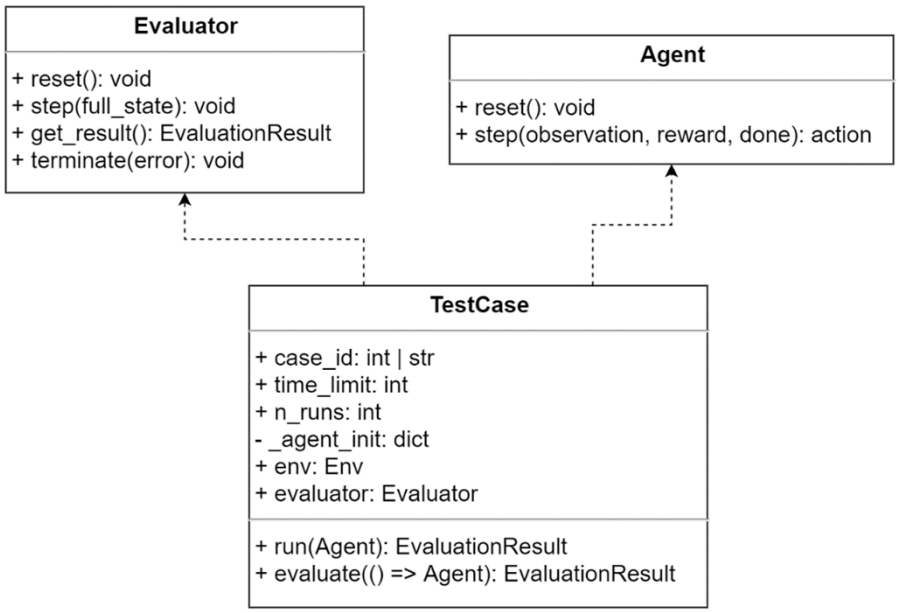

Figure 3.5: Class Diagram for aiVLE Grader

3.3 aiVLE Worker - Secure and Scalable Grading Client

I have released the second stable version of aiVLE Worker (relative to the first version released in Semester 1). The most significant upgrade in the new version is its resource awareness (see Section 3.3.3). It is validated on both CPU-only machines and GPU nodes with CUDA drivers (GTX 1050Ti for CUDA 10, RTX 3070 for CUDA 11). Moreover, it has been deployed and tested on several GPU nodes of the SoC Compute Cluster (see Section 4.1). The codebase consists of ∼700 LoC, also with examples and detailed inline documentation. A brief user guide can be found at https://edu-ai.github.io/aivle-docs/user-guide/aivle-worker/.

Unlike aiVLE Gym and Grader that are packages that are meant to be imported in other scripts, aiVLE Worker is a self-contained client that is executable out-of-the-box. Thus aiVLE Worker is published to PyPI (https://test.pypi.org/project/aivle-worker) as a command-line tool. Users may install aiVLE Worker from PyPI and use it like any regular program as shown in Figure 3.6.

```
(venv) (base) ➜ temp aivle-worker -h
usage: aivle-worker [-h] [--queue QUEUE] [--concurrency CONCURRENCY] [--worker-name WORKER_NAME] [--zmq-port ZMQ_PORT]

Run aiVLE worker client.

optional arguments:
  -h, --help            show this help message and exit
  --queue QUEUE         aiVLE task queue name, default as 'default'
  --concurrency CONCURRENCY
                        maximum number of concurrent evaluation, default as 1
  --worker-name WORKER_NAME
                        worker name, default as 'celery'
  --zmq-port ZMQ_PORT   ZMQ port used for communication between monitor, warden and main worker thread, default as 15921
```

Figure 3.6: aiVLE Worker Command-line Interface

3.3.1 Design Goal

For security purposes, the implementation is expected to achieve:

1. **File access restriction**: agent program should have no access to directories that may contain sensitive data (e.g., private keys), and should have read only access to files necessary for its execution (e.g., Python binary, dependencies, agent, and environment source code).

2. **Network restriction**: agent program should have no access to the Internet. Otherwise, 1) it may benefit from extra computation resources; 2) it may send out confidential runtime details (e.g., configuration of simulation environment) that allow users to fine-tune their program; both of which make the competition unfair.

3. **Resource limit**: agent program should be terminated and reported if it exceeds RAM, VRAM or time limit.

For scalability, in addition to the task queue system that manages all worker nodes and distributes evaluation jobs fairly and efficiently to the workers (see Section 3.4.2), on the worker side, a client that requires little permission and setup would be extremely helpful. It benefits horizontal scalability because 1) adding new worker nodes is easier, 2) more machines can be used as worker nodes (e.g., shared GPU servers, programming lab PCs). Thus, what we expect from this solution are:

1. **Managed concurrency**: it should be able to run as many evaluation jobs concurrently as the hardware resource permits, while being able to detect and terminate processes that

consume excessive RAM/VRAM.

2. **Minimal permission requirement**: if we have access to run the submission locally, the environment should be able to operate as a worker node (i.e., sudo is not required).

3. **Minimal dependency requirement**: any Linux machine with kernel version ≥ 5.4 and Python version ≥ 3.8 should be able to run the worker client.

4. **Moderate overhead**: compared to traditional OJ, aiVLE can trade some overhead (both startup and runtime) for absolute essentials like GPU support. However, to achieve a certain level of throughput, warmup time as long as several minutes is still unacceptable.

3.3.2 Security Solution

There are three mainstream security solutions for our consideration: 1) virtual machine (VM) such as Virtualbox, 2) container such as Docker/Podman, and 3) sandbox such as Firejail. All three solutions are capable of fulfilling the security requirements (i.e., file access, network and resource utilization restriction). So here we mainly focus on the scalability objectives: **minimal permission requirement** means we prefer a rootless solution; **minimal dependency requirement** means additional setups like "nVIDIA container runtime" are not welcomed; **moderate overhead** means we prefer smaller (runtime) overhead and faster startups. The corresponding areas of interest are compared in Table 3.1:

	VM	**Container**		**Sandbox**
	VirtualBox	Docker	Podman	Firejail
Rootless	No	Yes	Yes	Yes
Level of isolation	Very high	High		Medium
Overhead	High	Low		None
Startup time	∼15s	∼3s		∼0.05s
GPU support	No	Yes, with NVIDIA container runtime		Yes

Table 3.1: Comparison of Mainstream Security Solutions (Note: there are some caveats to the claims listed in this table as detailed in Appendix C.1)

From the comparison, we conclude that containers and sandboxes are preferred for their rootless operation and much shorter startup time compared with VMs. Additionally, there are several must-haves for the candidate solution:

1. GPU-support. Many recent RL algorithms are practically impossible to run without a GPU. This eliminates VM without PCI passthrough.

2. Server needs to be shared. This eliminates VM with PCI passthrough as it requires exclusive access. This also eliminates Podman and Docker, as they prevent the GPU from being shared by any other container with root access.

Therefore, Firejail is the only option left. I built a custom security profile to expose only the necessary file system and devices to processes inside the sandbox. I also used Firejail to impose CPU affinity (i.e., core count) and RAM usage limit on each sandbox.

3.3.3 Resource Awareness

The architecture of aiVLE Worker's resource awareness is illustrated in Figure 3.7. In the following subsections, we will discuss the objectives and design of related components, namely `monitor` and `warden` modules.

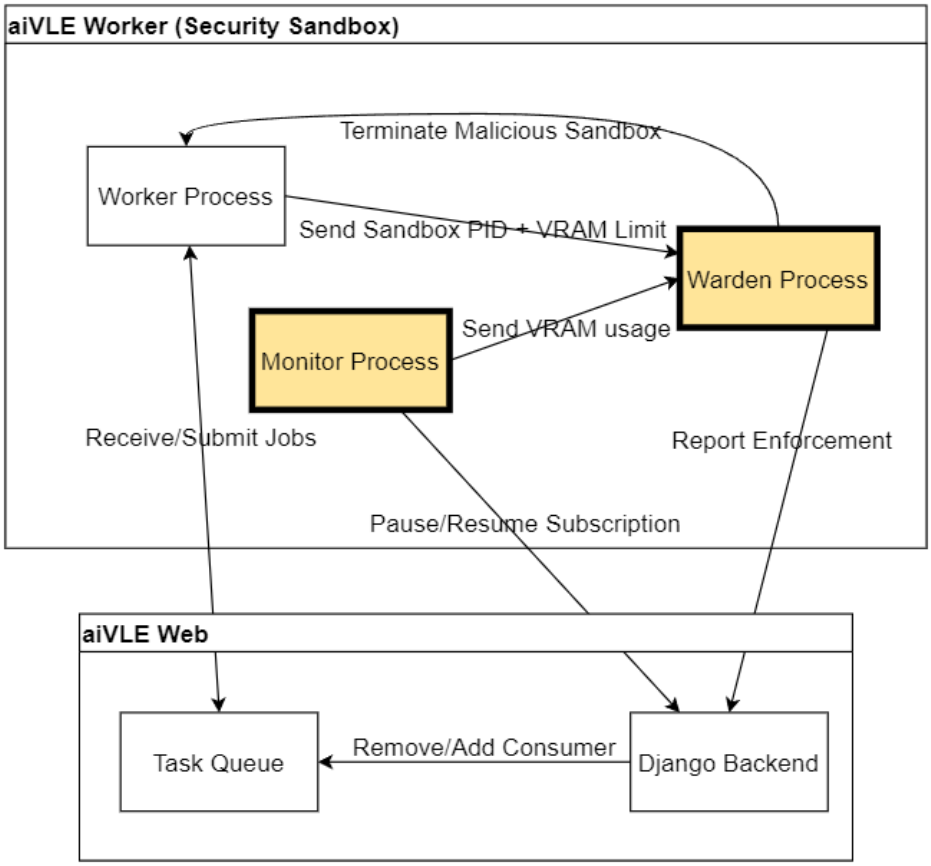

Figure 3.7: aiVLE Worker Resource Awareness Architecture

Resource Monitoring - `monitor` module

To achieve resource-sensitive load balancing, and to enforce resource limit, we need to monitor CPU/GPU utilization and RAM/VRAM usage and take actions accordingly. Therefore we implement the `monitor` module inside aiVLE Worker that runs in parallel with the main worker process. The `monitor` module:

1. Monitors the system utilization periodically
2. Controls the worker's task queue subscription according to prefetched threshold
3. Sends monitoring metrics to the `warden` module to enable resource limit enforcement

The first objective is achieved by the execution flow illustrated in Figure 3.8 with the help

of `psutil`[8] for CPU/RAM monitoring and `py3nvml`[9] for GPU monitoring. The last objective involves inter-process communication that will be discussed later in detail.

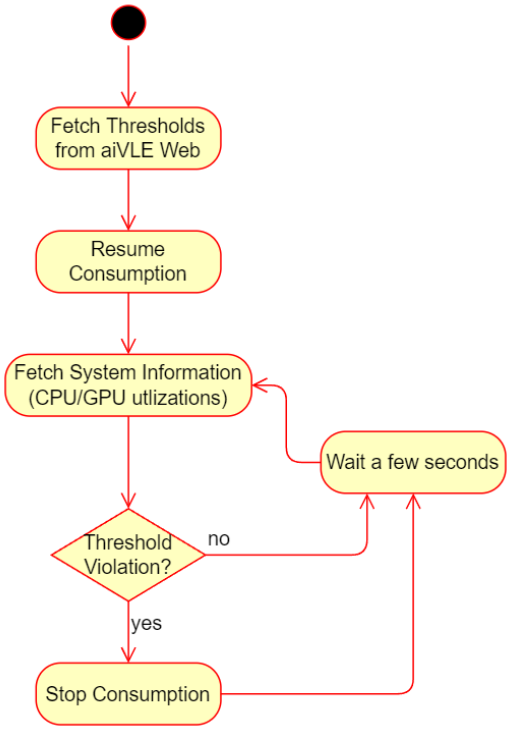

Figure 3.8: Execution Flow of `monitor` module

For the second objective, the reason why `monitor` module is also responsible for controlling the task queue subscription is that: Celery (the Python task queue library we use) does not allow the worker itself to pause **consuming** tasks - the worker can only **shutdown** itself entirely, and only the master server can pause **sending** tasks to a certain worker. So we have two possible designs: either sending monitoring metrics periodically to the master server and let the master server have full control, or make the workers fetch the threshold on startup and request the master server to pause whenever the threshold is violated. The second option is our choice because it incurs much less communication overhead and a fixed, prefetch threshold works fine in our use case.

[8]https://pypi.org/project/psutil/
[9]https://github.com/fbcotter/py3nvml

Limit Enforcement - `warden` module

The `warden` module is responsible for:

1. Collecting task information from `worker` module
2. Collecting system resource information from `monitor` module
3. Terminating sandboxes that violate resource restrictions and reporting to aiVLE Web[10]

Terminating sandboxes and reporting to aiVLE Web is straightforward - killing the corresponding PID and making an HTTP request. What makes `warden`'s job challenging is its first two objectives: there are three modules that run in parallel: `worker`, `monitor` and `warden`; they need to exchange information in order to collaborate. This is where ZeroMQ (Akgul, 2013) comes handy: only the `worker` knows the mapping from sandbox PID to task information (i.e., VRAM limit), and only the `monitor` knows the VRAM usage of each process. Both need to send their data to the `warden` process which oversees all running jobs and terminates jobs that violate restrictions. Figure 3.9 shows the inter-process communication among these three:

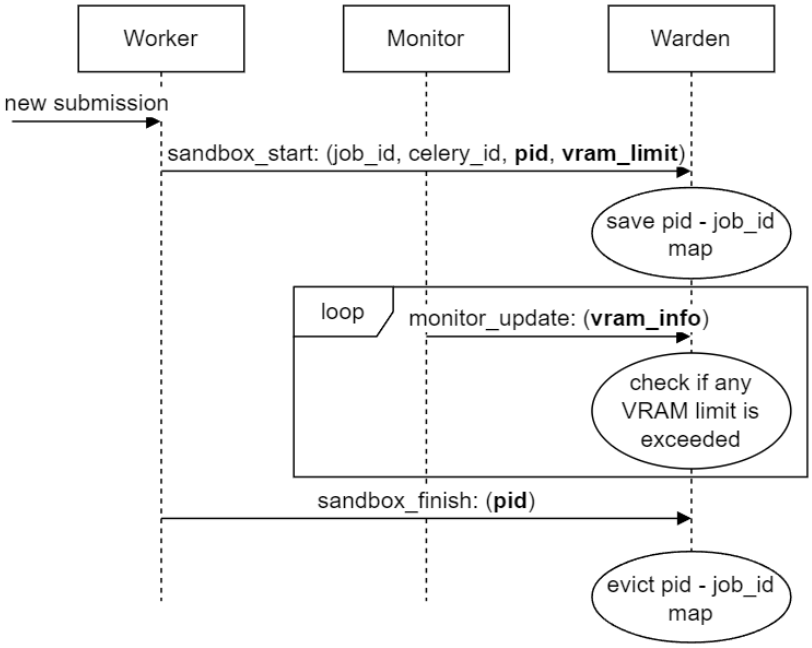

Figure 3.9: aiVLE Worker Inter-process Communication

[10]We use "master server" and aiVLE Web interchangeably in the context of task queue because they run in the same monolithic Python server application.

For `warden`'s case, every time it receives an update from `monitor`, it goes through the list of active sandboxes, queries the process hierarchy[11] to calculate the total VRAM usage of each sandbox, and finally checks if any sandbox needs to be terminated.

3.4 aiVLE Web - AI Competition Platform

aiVLE Web consists of a Django backend that is ~2.5K LoC and a React frontend that is ~2K LoC. An active instance is deployed on the server and is accessible from https://cs5446.comp.nus.edu.sg (frontend) and https://cs5446-api.comp.nus.edu.sg (API and admin panel). Figure 3.10 shows a few screenshots of the React frontend. Note that we used mobile-sized screenshots due to limited space, but the design is responsive (i.e., it adapts to different screen sizes).

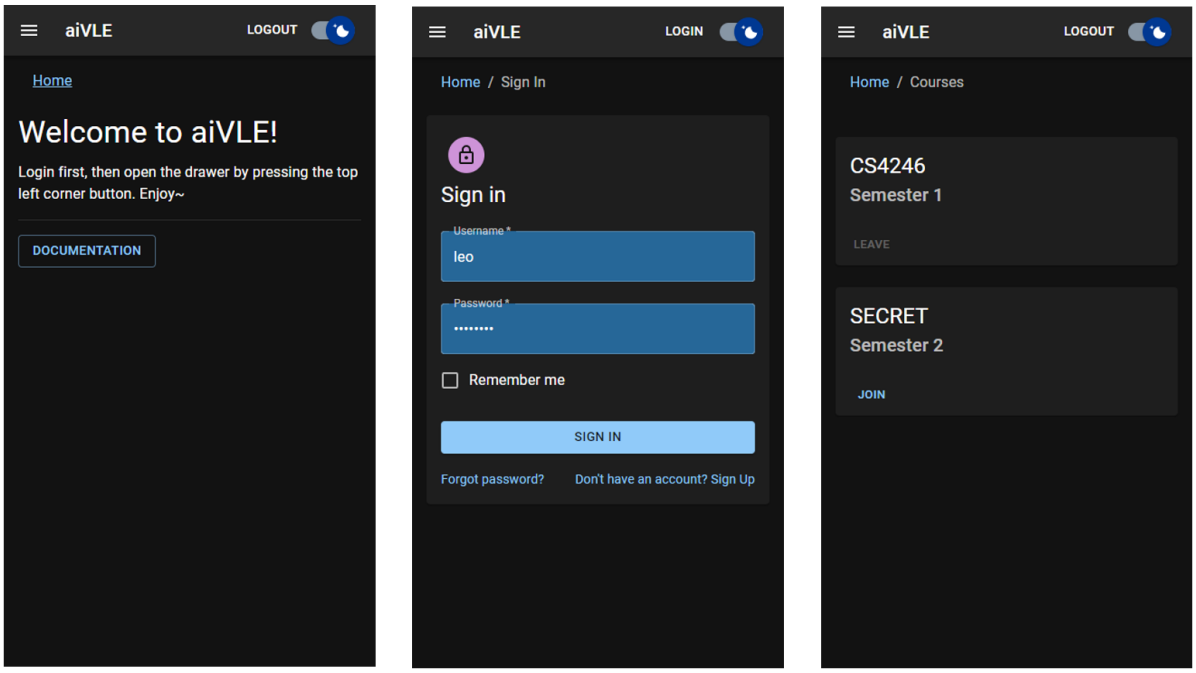

Figure 3.10: aiVLE Web Frontend Screenshots

[11] This is necessary as most frameworks like PyTorch will spawn new processes for computation. Generally the process that is directly utilizing GPU is no longer the parent process. And there might be multiple processes in the same sandbox that consumes VRAM. By traversing the process tree recursively, we prevent intentional or unintentional attempts of "downplaying" the resource consumption of certain sandboxes.

3.4.1 Design Goal

There are three primary considerations to the design of aiVLE web: extensibility, scalability, and usability.

Extensibility means the architecture should be extensible for future upgrades. One of the most significant architectural changes is the frontend-backend separation. This allows sophisticated data processing without making the backend codebase and API bloated - we will discuss one such scenario (course administration) in Section 3.4.5 for a tool named aiVLE CLI. In addition, to demonstrate the extensibility of our database model, we prepared a proposal on how to extend the existing codebase to support multi-agent competition (see Appendix D).

Scalability means the platform should be able to spread the evaluation workload, the most computationally heavy task, over many worker machines. We will discuss this objective in detail with a highly available and fault tolerant task queue (Section 3.4.2) and resource-sensitive load balancing (Section 3.4.3).

Usability means the platform should provide all the features the users (e.g., CS4246 teaching team) need for a successful semester of teaching. While there are many features we could discuss, due to space limit, we pick the role-based permission model (Section 3.4.4) as a typical example of how we carefully balance complexity and flexibility during the development process.

3.4.2 Highly Available and Fault Tolerant Task Queue

This is a continuation of the scalability of (grading) workers. In the section for aiVLE Worker, we addressed the problem of running the worker on as many computers as possible with as little configuration/permission as possible. Here we address the problem of 1) coordinating the communication between the workers and the backend server, and 2) distributing grading tasks to the workers efficiently for shorter waiting time and higher resource utilization.

The Problem

In aiVLE 1.0, there is no concept of a "task queue". All pending tasks are stored in the database with the status "QUEUED". Communication between the worker (or runner as per

the term used in aiVLE 1.0) and server is *half-duplex by polling*. In other words, the worker makes **periodic** requests to the server for new ungraded submissions:

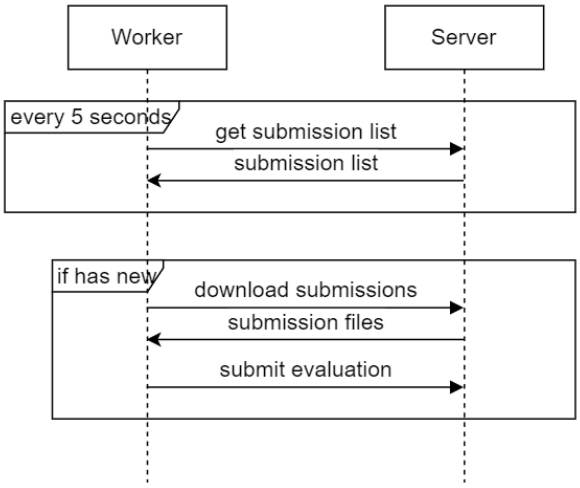

Figure 3.11: Old aiVLE "Task Queue"

There are three critical limitations to this approach:

1. Worker-side polling does not scale well: there is no mechanism in orchestrating the timing and order of workers polling the server for new jobs. The severity of possible traffic spike (i.e., many workers poll the server at the same time due to lack of coordination) increases linearly with the number of worker nodes.

2. Race condition: if two worker pull submissions at the same time, both will get the same ungraded submission. Such redundant work will get more significant with more worker nodes, which hurts the overall efficiency of the worker cluster.

3. No concurrency in each worker[12]: each individual worker only polls for new job when it has no submission to grade. In other words, each worker can only grade one submission at a time.

4. No load balancing: the backend has no control over which worker grades which submission, therefore load balancing is virtually impossible

[12]In the actual use of aiVLE 1.0, we run multiple worker clients on the same machine for disguised concurrency. Besides the inconvenience of manually starting multiple worker clients, this approach makes managing resources difficult as each client runs without knowing the existence of others even if they are on the same machine.

The Solution

Similar to the idea of extracting the responsibility of data storage/management into a separate database backend, we delegate the messaging tasks to a message queue (MQ). Conceptually, message queue enables asynchronous communication between clients (who submit tasks) and workers (who finish tasks). As for implementing the concept in our Python web application, we used Celery framework with RabbitMQ message queue broker/backend. Figure 3.12 illustrates how the MQ-based task queue works:

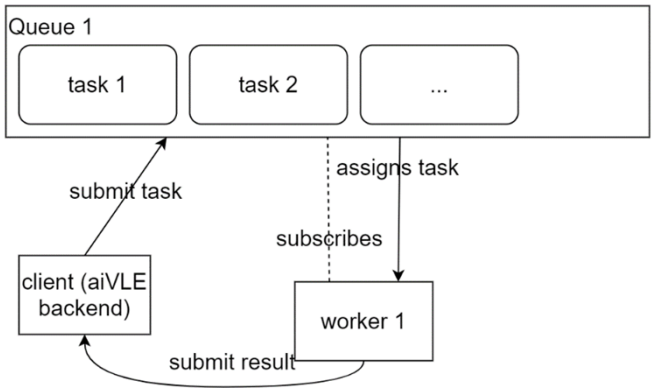

Figure 3.12: Message Queue Based Task Queue

Every worker listens to one or more "queues", where the message broker is responsible for allocating tasks fairly among workers in each queue. When an evaluation job arrives, aiVLE backend will submit the job to an appropriate queue (i.e., private queue if user has dedicated workers available, otherwise public CPU/GPU queue according to task specification) and wait for the assigned worker to submit evaluation result. The task will remain in the queue until it is processed (i.e., message queue is persistent). A randomly generated task ID is used to authenticate worker's submission – only the worker whom the broker assigned the task to will have this ID. This approach not only reduces the number of requests to be $O(n)$ where n is the number of evaluation jobs, but also ensures fair distribution of tasks among workers. Moreover, we also benefit from other standard features of message queue such as automatic retry and heartbeat checks.

3.4.3 Resource-sensitive Load Balancing

By using message queue for task distribution, we already have some primitive load balancing - Celery with RabbitMQ backend by default dispatches messages to all consumers in round-robin style, therefore all consumers are expected to consume the same number of tasks from the same queue over a fixed period of time. Although this is a huge improvement over no load balancing at all, from some stress testing using real-world workload, we find it necessary to take system load into consideration. Specifically, the primitive method of load balancing by number of tasks works poorly when certain tasks are much more resource intensive. For example, assume both worker A and worker B receive 5 submissions, but the ones for worker A consumes 30% of total VRAM while the ones for worker B takes only 10% of total VRAM. There are two serious implications in this imaginary scenario:

1. **Unfairness**: suppose the worker is configured to run at most 8 concurrent evaluations, the fourth and fifth submissions arriving at worker A will not have sufficient VRAM. They will likely receive runtime errors which are unacceptable.

2. **Inefficiency**: suppose the task queue is configured to retry the failed job by putting a new evaluation job into the same queue, since worker A is still accepting submissions, the retry attempt may take additional fails before finally arriving at worker B.

The key to solving such unwanted behavior is 1) to monitor the available system resources, 2) to stop processing new tasks when the available resources fall below certain thresholds. Most heavy-lifting is handled on the worker side (see Section 3.3.3). As for aiVLE Web, it uses `add_consumer` and `cancel_consumer` Celery APIs for resuming and pausing consuming respectively. Nevertheless, there are two points to note in this solution:

1. It does not cancel already running jobs. Instead, it only stops dispatching new tasks to the worker where the threshold is met. This behavior is acceptable in our case since we only need to promise submissions with a certain amount of resources to run the job, and

our solution guarantees[13] the promised amount of CPU/RAM/VRAM at the beginning of any evaluation job.

2. It does not balance the resource utilization among all workers. `cancel_consumer` simply removes the worker from the list of consumers to dispatch tasks to. To achieve resource utilization balance, aiVLE Web needs to understand each worker's utilization in real-time and dispatch tasks accordingly. We think the overhead (i.e., network traffic for reporting utilization data) and complexity (i.e., load balancing algorithm) of this design significantly outweighs the potential benefits.

Besides addressing the previously mentioned implications, resource-sensitive load balancing greatly increases the system utilization. Without resource awareness, aiVLE 1.0 could only achieve disguised concurrency by running multiple worker clients on the same machine. To ensure the machines will not be overloaded, we could only "play it safe" by always under-utilizing the resources. For example, the current configuration of aiVLE 1.0 only allows 3 concurrent evaluations, resulting in an average of $\sim 30\%$ of volatile GPU usage. In comparison, we can now achieve up to 85% GPU utilization rate, and with resource-sensitive load balancing, overloading the system is no longer a concern. This translates to roughly three times of a utilization improvement.

3.4.4 Role-based Permission Model

In Django we have two major types of permission model: per-model permission (built-in authentication system) and per-object permission (django-guardian). Per-model permission means the finest granularity is model-level, that is we may only check if the user has read/write/edit/delete access to the model. Per-object permission is the finest granularity possible as we can establish arbitrary permission between any object and any user. In fact, the worst possible space complexity is $O(mnk)$ where m is the number of objects, n is the number of users and k is the number of permissions.

[13] Technically speaking, this guarantee does not hold between two checks on system information, but we can easily reduce the impact by performing the check more frequently. In our experience, one second interval is more than enough.

In our use case, model-based permission is too coarse-grained: a student should only be able to view the tasks in the course he/she enrolled in, rather than all the tasks available on the platform. On the other hand, object-based permission is overkill: there might be hundreds of thousands of job records in the database, storing each user's permission to each job record is simply too wasteful. Therefore, we propose a solution that is the sweet spot between the two extremes: role-based permission model.

The entity relationship diagram of aiVLE Web is illustrated in Figure 3.13:

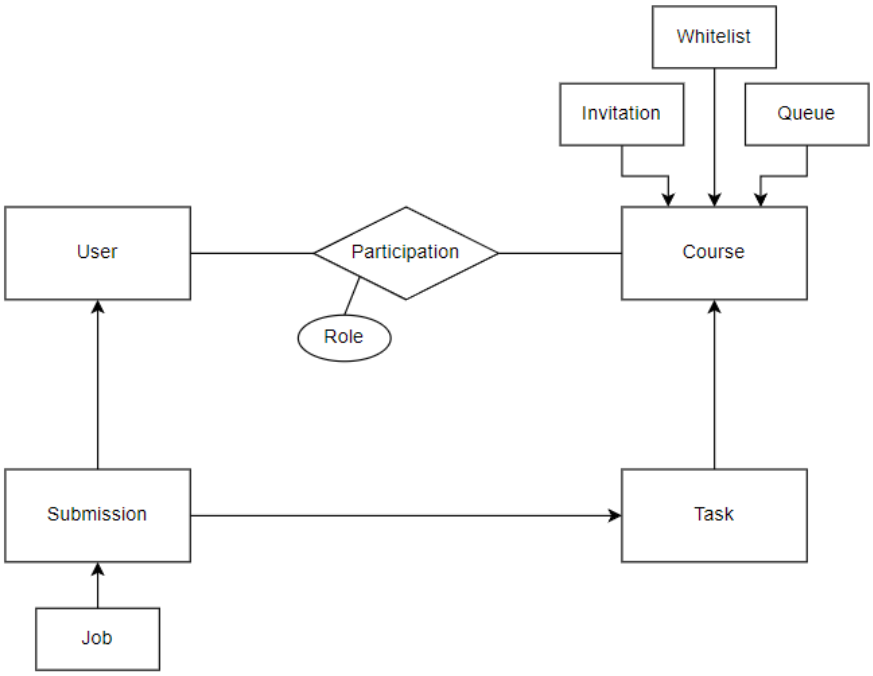

Figure 3.13: aiVLE Web Entity Relationship Diagram

In the relational entity Participation, we record the `Role` of the `User` in a `Course`. Possible roles (as defined in `app.model.participation`) are: administrator (ADM), lecturer (LEC), teaching assistant (TA), student (STU) and guest (GUE). Note that a user can have different roles in different courses, and a superuser has access to everything.

The idea of role-based permission model is centered around one's role in the corresponding course: to determine one's access to any object, we first find the object's related course, then check if the user has access to this object in the context of this related course. For example, if we need to know if the user has view access to a certain `Job`, we can follow the arrows

in the diagram: Job → Submission → Task → Course → Participation → User to find the corresponding role, and check if this role has `job.view` permission according to the permission lookup table (see Table 3.2).

By introducing the `Participation` relational entity and `has_perm(course, user, permission)` utility function, we compressed the worst case $O(mnk)$ space complexity to $O(nk)$ and it is a significant improvement - in reality the number of objects m significantly outweighs the number of users n or permissions k. We can summarize different roles' access (as defined in `aiVLE.settings.ROLES`) using a permission matrix in Table 3.2:

		Admin	Lecturer	TA	Student	Guest
Task	View opened tasks	x	x	x	x	x
	View all tasks	x	x	x		
	Add task	x	x			
	Edit task	x	x	x		
	Delete task	x	x			
Submission	View own submissions	x	x	x	x	
	View all submissions	x	x	x		
	Add submission (under own name)	x	x	x	x	
	Download submission	x	x			

Table 3.2: Partial Permission Lookup Table

3.4.5 aiVLE CLI - Course Administration Utility

aiVLE CLI is an interactive command-line tool written in (∼ 500 lines of) Go. It is cross-compiled[14] to native binary on Windows, Linux and macOS (Intel + Apple Silicon) so that the facilitators can enjoy the same convenience on any platform they prefer. Users make use of this program by answering questions and selecting choices as shown in Figure 3.14.

[14]Cross-compilation means using platform A to build a binary that runs on platform B.

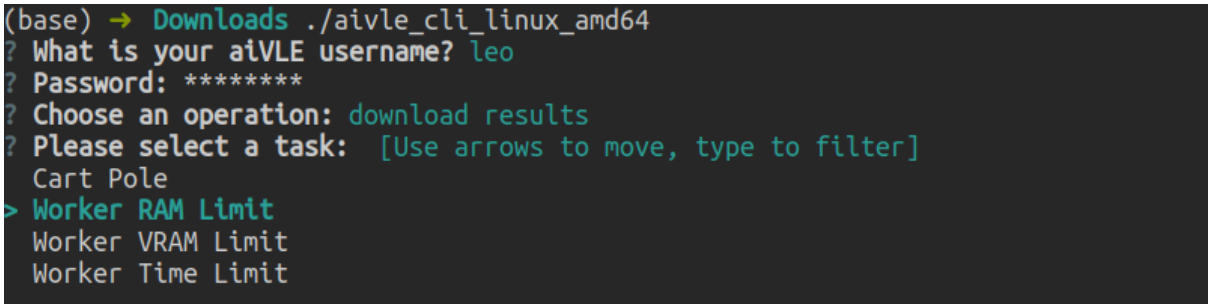

Figure 3.14: aiVLE CLI Usage

There are two reasons behind building a separate application for course administration on the client side instead of integrating all these functionalities into aiVLE Web. First, it is easier to develop a command-line program compared with both a backend API (logic) and corresponding web frontend (user interface). For a command-line app, work required for the user interface is negligible and we only need to focus on delivering the logic. Second, KISS (keep it simple, stupid) principle. Ideally, if a complex functionality can be implemented with several smaller yet more general operations without significant performance or usability penalty, we should avoid providing such a complex API and let the caller do the composition. Note that such composition is impossible without frontend-backend separation, since before the separation the only way of interacting with the data is to use the website, and the website itself, while friendly to us humans, is not as useful as raw data to computer programs.

Currently the aiVLE CLI has 4 features:

1. Download submissions: given a task, you may download all submissions (evaluation result included) or only those are marked for grading.

2. Download evaluation results: given a task, you may download a CSV file with username, grade, and evaluation details.

3. Upload student emails to the course whitelist: the utility can parse and upload student roster Excel file exported from LumiNUS.

4. Get API token from username and password.

Chapter 4

Deployment and Testing

aiVLE 2.0, especially aiVLE Worker and aiVLE Grader, is designed to be highly scalable and easy to deploy. However, just like any distributed systems, being able to run the individual components on a single machine is one thing, having all the components cooperate on separate machines to achieve actual distribution is another. Thus, to pave way for production deployment in the next academic year, and to demonstrate the actual performance of such a distributed system, we performed a complete deployment using several SoC cluster nodes and did several experiments/benchmarks on the system.

There are three sections covering both the deployment and tests this chapter: Section 4.1 describes the deployment environment and deployment steps. Section 4.2 discusses the experiment of load balancing among **multiple** worker nodes. Section 4.3 discusses the experiment of running evaluation jobs concurrently on a **single** worker node.

4.1 Deployment

Note that this deployment happened during the winter break (November 2021 to January 2022). Therefore some later updates to the aiVLE Web and Worker were not included during this deployment (most notably, resource-sensitive load balancing as mentioned in Section 3.3.3 and Section 3.4.3). However, this does not affect the test results as the experiments are designed to have no system overloading (more on that later).

In specific, the exact versions used are (tag with GitHub link):

- aiVLE Web: deploy-1 (https://github.com/edu-ai/aivle-web/releases/tag/deploy-1)
- aiVLE Worker: v0.1.2 (https://github.com/edu-ai/aivle-worker/releases/tag/v0.1.2)

4.1.1 Environment

aiVLE Worker is deployed on SoC compute cluster `xgpg0, xgpg1, xgpg2` with the following configuration:

- Operating System (Code 4.1): Ubuntu 20.04 LTS with Linux kernel version 5.4.0
- GPU (Code 4.2): NVIDIA A100-PCI, Driver 495.29, CUDA 11.5
- CPU: 2x AMD Epyc 7352, in total 48 cores/96 threads, base clock 2.3GHz
- RAM: 256GiB DDR4

```
> cat /proc/version
Linux version 5.4.0-91-generic (buildd@lcy01-amd64-017)
(gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)) #102-Ubuntu SMP Fri Nov 5 16:31:28 UTC 2021
```

Code 4.1: Deployment Environment - Operating System

```
> nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.29.05    Driver Version: 495.29.05    CUDA Version: 11.5     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-PCI...  On   | 00000000:01:00.0 Off |                    0 |
| N/A   49C    P0    36W / 250W |      0MiB / 40536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
```

Code 4.2: Deployment Environment - GPU

aiVLE Web is deployed on a Linode Nanode 1GB VPS (virtual private server) with 1 virtual CPU core and 1 GiB of RAM. We carefully picked the parameters to avoid the aiVLE Web server being the bottleneck of all experiments (see Section 4.2.1).

4.1.2 Limitations

As with every distributed system, even if the system works on a few machines, it does not necessarily mean the system will work or scale well to dozens or even hundreds of machines. While we design the system to be highly scalable, and the underlying technologies (i.e., Celery as Python library and RabbitMQ as message queue broker) have proven to be effective on hundreds of machines, we are never certain until we actually scale the system to that many nodes and put it under pressure.

Unfortunately, for the time being, we are unable to materialize such a large-scale experiment: as we could only secure access to three machines in the SoC compute cluster. Also, at the moment, the tasks were not demanding enough to put the system at such a scale under considerable pressure. For example, on the `xgpg*` nodes used for this experiment, every evaluation takes ∼10 seconds to finish, which is too short for even tens of worker nodes. For the evaluation subsystem to be stressed, we at least need to keep the task queue "filled". In other words, the rate of submitting new jobs into the task queue should be comparable to the rate of workers finishing jobs. However, with each job only taking ∼10 seconds, suppose we have 50 workers, our rate of consumption would be $50/10 = 5$ jobs per second (every 10 seconds we can process 50 jobs). If we assume each submission to be ∼20MiB[15], it would require at least 100MiB per second of network throughput, which is already approaching the limit of gigabit Ethernet available on most machines.

Therefore, with our current resources and setup, we could not properly evaluate the scalability potential of aiVLE 2.0. However, our experiment setup is on par with the resources available to the CS4246 teaching team, and aiVLE 2.0 is shown to have a much higher utilization rate of computational power. As a result, aiVLE 2.0 can process submissions with much smaller

[15]Many deep learning models are much larger, so we are having a relatively conservative estimation here.

delays. In addition, properties like fair distribution and small worker overhead remain stable regardless of the number of nodes, so they were properly evaluated in our experiments.

To summarize, we are cautiously optimistic about the scalability of aiVLE 2.0 as we do not yet have experiment data to support it, but we are confident that it can support CS4246 teaching much more effectively from the results.

4.1.3 Steps

To simulate the production deployment process, we started everything afresh. The following steps[16] are sufficient for any deployment from the ground up:

1. Prepare the message queue broker: either by installing RabbitMQ or using cloud message queue provider such as CloudAMQP. In our case, we installed RabbitMQ on the same VPS with aiVLE Web.

2. Install and start aiVLE Web on the master server. The aiVLE Web Read-me[17] is the definitive guide on this process.

3. Setup the users, courses, tasks in aiVLE Web via its RESTful API or Django admin panel.

4. Prepare the worker nodes with necessary dependencies (i.e., Firejail, Pip, Virtualenv, CUDA drivers). In our case, we requested the cluster admins to install Firejail as all other dependencies are already available.

5. Install and start aiVLE Worker on the worker nodes. aiVLE Worker Read-me[18] describes this process in detail.

There are some issues we found during the deployment process. None of which is critical in a sense that we eventually found workarounds without modifying any existing codebase or design. You may find the details in Appendix E.

[16] Detailed deployment guide with common Q&A on the documentation website: https://edu-ai.github.io/aivle-docs/dev-guide/deployment-guide/

[17] https://github.com/edu-ai/aivle-web#readme

[18] https://github.com/edu-ai/aivle-worker#readme

4.2 Load Balance Experiment

In this section, we will discuss the experiments about load balancing evaluation jobs among **multiple** worker nodes. There are three objectives for this series of experiments:

1. Show that the evaluation system scales well horizontally to several worker nodes.
2. Show that the tasks are distributed evenly among worker nodes.
3. Show that the worker nodes are fully utilized during stressful load.

Raw logs and analyzing scripts can be found in aiVLE experiment logs repository[19]. Correspondence between experiment setup and log file index is shown in Table 4.1.

Web log index	Worker log index	Node count	Concurrency of worker	Submission count	Concurrency of submission
5	2	3	8	100	100
6	3	2	8	100	100
7	4	1	8	100	100

Table 4.1: Load Balancing Experiment Setup

4.2.1 Methodology

Since the evaluation task and all worker nodes are exactly the same, we have four parameters/variables to adjust:

1. Node count: number of active nodes during the experiment.
2. Concurrency of worker: maximum number of concurrent evaluation jobs allowed on *each* worker node.
3. Submission count: number of evaluation jobs submitted to the task queue.
4. Concurrency of submission: maximum number of threads submitting jobs concurrently. Note that if this number is comparable to the submission count, master server will be

[19] https://github.com/edu-ai/aivle-experiment-logs

blocked until all submissions are accepted (i.e., it will not distribute evaluation jobs to any of the worker nodes until all submissions are well-received)[20].

In this section and Section 4.3, submission count and concurrency of submission are both 100. This means before the first job is assigned to any of the worker nodes, 100 jobs are already queued. This is to ensure there will always be sufficient tasks for the workers to work on - if concurrency of submission is significantly smaller than the number of submissions, then the rate of submitting job may dictate the rate of workers finishing jobs, which is undesirable for our stress-oriented experiments.

For load balance experiment, we **fix concurrency on all workers to be 8**, measure **time taken** to finish 100 submissions with

- 1 node (`xgpg0`)
- 2 nodes (`xgpg0,1`)
- 3 nodes (`xgpg0,1,2`)

On the master server (aiVLE Web), we logged the critical phases of each submission with a timestamp, in specific, there is a timestamped record when

1. submission is received
2. evaluation job is submitted to task queue
3. evaluation job is picked up by a worker
4. evaluation job is being worked on by a worker
5. evaluation job is terminated (either finished or failed)

The start time is defined to be the earlier of 1) the latest "submission is received" record or 2) the earliest "evaluation job is picked up by a worker" record. The finish time is defined to be

[20]This is because the submission rate is significantly larger than how fast the master server can process requests, and these submissions will pile up in the requests queue. Since the master server processes requests on a first-come, first-served basis, it is effectively blocked before all submissions are accepted. Of course this number cannot grow indefinitely as any machine will eventually run out of available threads or network bandwidth. Consequently, the submission count cannot grow indefinitely either as it should be comparable to the concurrency of submission. Section 4.2.1 will discuss the preliminary experiments that aim to find these upper bounds.

the latest "evaluation job is terminated" record. The total time is defined to be the difference between the finish time and the start time. The detailed method of calculating time taken to finish all submissions can be found in the analyze script. Note that the focus of this experiment is on the **evaluation subsystem**, not the entire system. For example, before we can start evaluation jobs, we need to upload the student submission first. However, the upload itself could take a significant amount of time depending on the server workload. Such preparation time is an irrelevant variable and should be excluded from our calculation. The definition of the start time here, combined with our effort to adjust the parameters such that evaluation starts only after all submissions are received, achieved such exclusion.

On each worker node, we also logged the timestamped GPU utilization rate and VRAM usage periodically (Figure 4.1). This helps us understand whether all worker nodes are busy most of the time - unnecessary idling is a sign of poor load balancing.

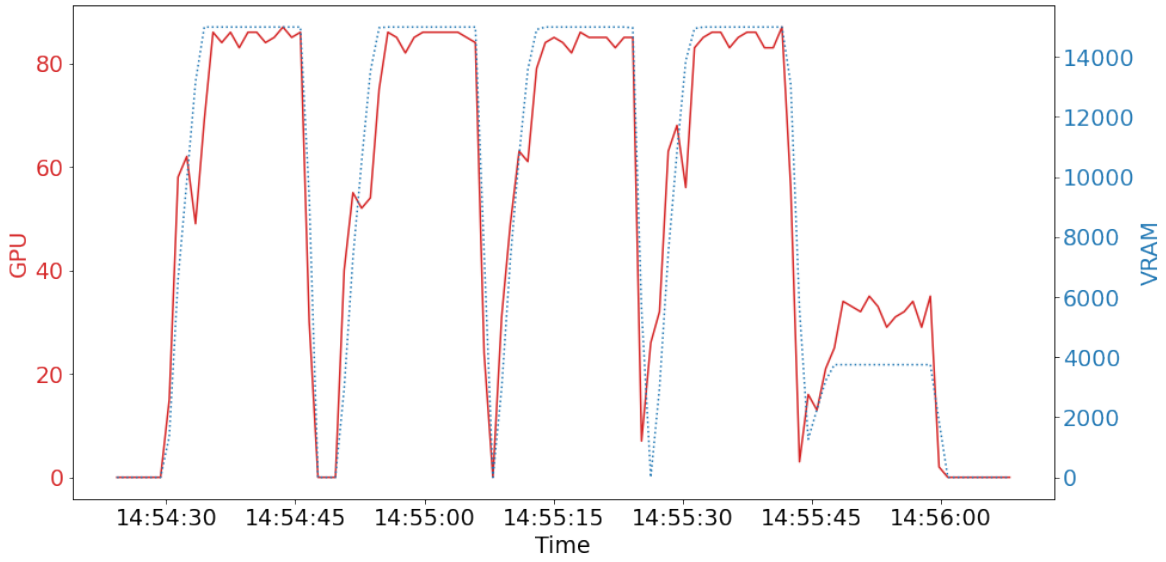

Figure 4.1: GPU/VRAM Utilization Plot from `xgpg0_2.log`

Choice of Parameters

There are two parameters that remain unexplained: number of total submissions (100) and maximum concurrent evaluations on each worker (8).

For the maximum concurrency allowed on each worker, we want the concurrency to be as

large as possible (so that we properly "stress" the system) without overwhelming the system (so that the scaling overhead is mostly because of our task queue system instead of overwhelmed worker nodes). Since our evaluation task is mostly GPU-bounded, we performed experiments on a single machine with concurrent evaluation ranging from 2 to 16 and observed both the **GPU** utilization and **VRAM** utilization. However, with 40GiB of VRAM, it is hard to overload the VRAM even with 16 concurrent jobs as each job takes less than 2GiB of VRAM. On the other hand, GPU utilization increases linearly with number of concurrent jobs until more than 8 jobs. Increasing more jobs will not significantly increase the GPU utilization after 8 jobs, indicating that the machine could only handle up to 8 jobs without significantly degraded per-job performance.

For the total submission count, it is bounded by the maximum concurrent submission our server and network can handle. We performed preliminary experiments with 1500 (`debug.log.1`) and 250 (`debug.log.4`) total submissions, and found the submission client could not reach sufficient request concurrency (i.e., large enough such that no evaluation will happen before all submissions are received) in both cases. In fact, for `debug.log.1` and `debug.log.4`, the average time taken to finish evaluating one submission is ∼25 seconds, and we did observe evaluation jobs started before submissions are concluded. Given that the optimal time (i.e., time taken without any communication overhead) is ∼15 seconds, this confirms our intuition that not queuing up all submissions before starting evaluation severely hurts the accuracy of these experiments. On the contrary, for scenarios listed in Table 4.1, the average time is ∼18 seconds, which is much closer to the optimal time, and ∼3 seconds of overhead seems acceptable considering it needs to download and decompress the student submission first.

4.2.2 Results

First, for the performance of load balancing, below are the times for each test case:

- 1 node: 235.426s (baseline)
- 2 nodes: 128.261s (91.78%)

- 3 nodes: 92.475s (84.86%)

The percentage is the scaling efficiency defined by $\frac{\text{Optional time}}{\text{Measured time}} \times 100\%$ where *Optimal Time* is defined as $\frac{t_0}{N}$ where N is the number of nodes and t_0 is the time taken with one node. We have observed satisfactory scaling efficiency in both 2-node and 3-node configurations.

Second, for the fairness of load balancing, below are the numbers of jobs assigned to each worker node (when there are more than one node):

- 2 nodes: {`celery@xgpg0: 50, celery@xgpg1: 50`}
- 3 nodes: {`celery@xgpg0: 34, celery@xgpg1: 36, 'celery@xgpg2': 30`}

The jobs are nearly equally distributed in both cases, meaning that our round-robin load balancing mechanism is working as expected.

Third, for the utilization of system resources, since our task is GPU-bound, Figure 4.1 plots the GPU and VRAM utilization for one of the worker nodes (others are similar). We have observed stable GPU utilization of $\sim 85\%$ at peak, and a busy rate of $\sim 96\%$, meaning we have very good utilization of worker node resources.

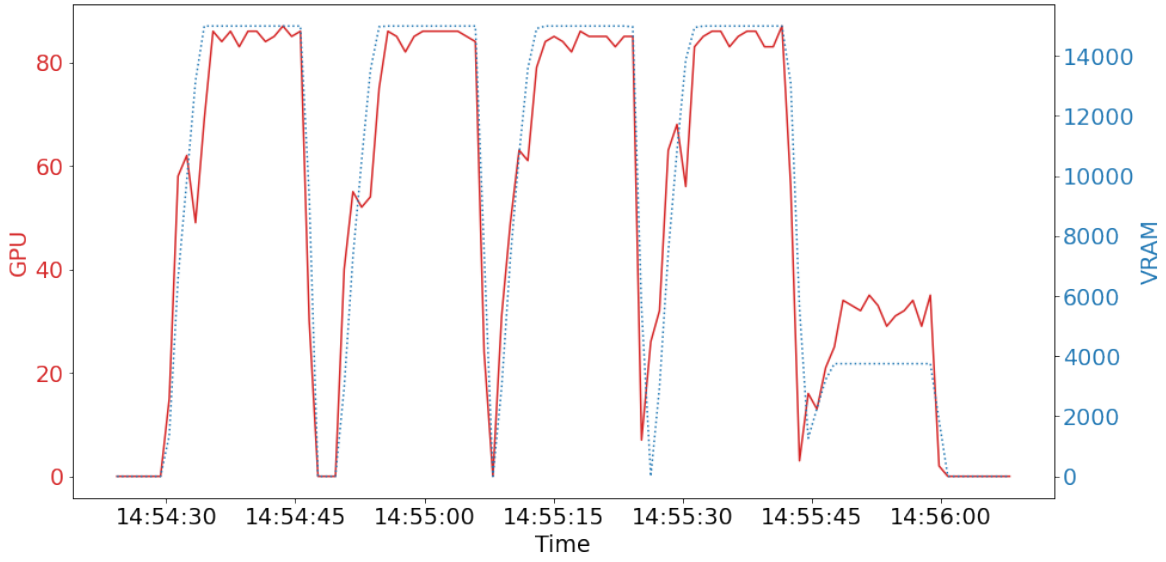

Figure 4.1: GPU/VRAM Utilization Plot from `xgpg0_2.log` (repeated from page 42)

4.3 Concurrency Experiment

In this section, we will discuss the experiments about running multiple evaluation jobs **concurrently** on a **single** worker node. The objective is to show that our worker node can process multiple submissions in parallel with linear scalability.

Raw logs and analyzing scripts can be found in aiVLE experiment logs repository. For correspondence between experiment setup and log file index, please refer to Table 4.2.

Web log index	Worker log index	Node count	Concurrency of worker	Submission count	Concurrency of submission
7	4	1	8	100	100
8	5	1	4	100	100
9	6	1	2	100	100
10	7	1	1	100	100

Table 4.2: Per-worker Concurrency Experiment Setup

4.3.1 Methodology

For explanation of parameters/variables, and the method of measuring total time please refer to Section 4.2.1. For the per-worker concurrency experiment, we activate **only one** worker node, and measure **time taken** to finish 100 submissions with the **concurrency of worker** ranging from 1, 2, 4 to 8.

4.3.2 Results

- concurrency = 1: 1558.811s (baseline)
- concurrency = 2: 859.237s (90.71%)
- concurrency = 4: 422.744s (92.18%)
- concurrency = 8: 235.426s (82.77%)

Similar to the first part of Section 4.2.2, the percentage in the end is the scaling efficiency. Its definition is also similar by changing the definition of N to be the concurrency number. We have achieved over 90% efficiency when the system is not too heavy-loaded, and over 80% efficiency when the system is at its absolute limit. We are satisfied with such per-instance concurrency performance.

Chapter 5

Conclusion

5.1 Summary

We started this project in pursuit of a complete solution for RL algorithm evaluation that is extensible, scalable, and easy-to-use, which are critical for the success of such a platform. During the two semesters of this final year project, we

- implemented RL environment framework that supports agent-environment separation and multi-agent competition;
- provided automated and secure evaluation for RL tasks;
- designed and deployed distributed evaluation subsystem based on message queue;
- improved the web application for hosting competitions by 1) restructuring code, 2) writing comprehensive documentation, and 3) adding new features (e.g., email verification, password reset, invitation token, course whitelist, etc.).

For the RL environment framework, we have invited another FYP student Ho Hol Yin for an early adoption in his project "A unified testbed for AI teaching and research". The feedback has been positive especially in terms of OpenAI Gym compatibility.

For the evaluation subsystem, we deployed the system to SoC compute cluster during the winter break and performed several performance experiments. We have validated the fair dis-

tribution of evaluation jobs, achieved up to 3 times of utilization improvement over the previous solution and demonstrated the new system's scaling capability of supporting CS4246 teaching.

For the competition-hosting web platform, by working with Dr. Narayan closely, we have addressed most of the pain points from CS4246 teaching team. More importantly, the documentation is now sufficient for fresh deployments and future upgrades/maintenance.

5.2 Limitations

We note the following limitations in the current implementation of aiVLE 2.0:

First, as mentioned in Section 4.1.2, due to limited resources, we could not test aiVLE 2.0's scalability on more machines (say dozens or even hundreds of nodes). Once we put aiVLE 2.0 into CS4246 production environment, we wish to onboard more stakeholders so that we can gradually scale up the system and explore its true potentials.

Second, aiVLE 2.0 platform does not provide multi-agent competition support. We started with aiVLE Gym that works great for multi-agent competitions, but once we try to extend aiVLE Web to support multi-agent tasks, we encountered much more difficulties than we anticipated. Most notably, we found it challenging to implement a skill rating and tournament system that balances efficiency and fairness. Nevertheless, we managed to implement the infrastructure and abstractions for the tournament system (see Appendix D), and we definitely hope to materialize the proposal by implementing actual tournament algorithms on top of the infrastructure in the future.

Lastly, although I emphasized the importance of maintainability repeatedly in this project by adopting many software engineering best practices such as comprehensive documentation and maintainable architecture, there is still some room for improvement, especially on the frontend code – unlike backend development that I am familiar with, this project is my first serious attempt with frontend development. Looking at the codebase after a year of experience, we think the frontend code needs some refactoring to achieve similar level of maintainability as the backend.

5.3 Future Work

aiVLE 2.0 has the potential to be an even more comprehensive AI competition platform. Our efforts in improving the project's maintainability are not only for the operation of CS4246 assignment grading, but also for the future upgrades from whoever is interested in making the platform even more advanced.

In particular, a concrete implementation of multi-agent tournament and skill rating system would be a huge upgrade - it opens up the possibility of multi-agent competition that is more interactive and competitive.

For the evaluation subsystem, we feel there is still room for improvement with respect to the load balancing strategy - currently we distribute tasks fairly by round-robin, and pauses assigning new jobs to nodes that are under pressure. However, if we managed to solve the significant communication overhead of reporting utilization data to the master server in real-time, we could implement some load balancing algorithm that balances the real system load among the workers.

Lastly, since aiVLE Gym is designed to function independently outside of aiVLE platform, its capability of separating environment from agents could be useful for academic research in, for example, human-in-the-loop machine learning. Besides AI education, we hope to see more application of these frameworks in the world of RL research, including but not limited to multi-agent experiments built with aiVLE Gym, and benchmarks packaged with aiVLE Grader.

References

Akgul, F. (2013). *Zeromq*. Packt Publishing.

Arnow, D., & Barshay, O. (1999). On-line programming examinations using web to teach. *SIGCSE Bull.*, *31*(3), 21–24. https://doi.org/10.1145/384267.305835

Barroso, L. A., Clidaras, J., & Hlzle, U. (2013). *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Morgan & Claypool Publishers.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2012). The Arcade Learning Environment: An Evaluation Platform for General Agents. *arXiv e-prints*, Article arXiv:1207.4708, arXiv:1207.4708.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv e-prints*, Article arXiv:1606.01540, arXiv:1606.01540.

Elo, A. E. (1978). *The rating of chessplayers, past and present*. Arco Pub. http://www.amazon.com/Rating-Chess-Players-Past-Present/dp/0668047216

Giuste, F. O., & Vizcarra, J. C. (2020). CIFAR-10 Image Classification Using Feature Ensembles. *arXiv e-prints*, Article arXiv:2002.03846, arXiv:2002.03846.

Isdahl, R., & Gundersen, O. E. (2019). Out-of-the-box reproducibility: A survey of machine learning platforms. *2019 15th International Conference on eScience (eScience)*, 86–95. https://doi.org/10.1109/eScience.2019.00017

Jain, S. (2020). *Linux containers and virtualization: A kernel perspective*. https://doi.org/10.1007/978-1-4842-6283-2

Koul, A. (2019). Ma-gym: Collection of multi-agent environments based on openai gym.

Kurnia, A., Lim, A., & Cheang, B. (2001). Online judge. *Computers & Education*, *36*(4), 299–315. https://doi.org/https://doi.org/10.1016/S0360-1315(01)00018-5

Nichol, A., Pfau, V., Hesse, C., Klimov, O., & Schulman, J. (2018). Gotta Learn Fast: A New Benchmark for Generalization in RL. *arXiv e-prints*, Article arXiv:1804.03720, arXiv:1804.03720.

Russell, S. J. N. P. (2010). *Artificial intelligence : A modern approach*. Prentice-Hall.

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLOS Computational Biology*, *9*(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J.,

Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction, 2nd ed.* The MIT Press.

Vanschoren, J., & Blockeel, H. (2009). A community-based platform for machine learning experimentation. *Lecture Notes in Computer Science*, *5782*, 750–754. https://doi.org/10.1007/978-3-642-04174-7_56

Wang, M., Han, W., & Chen, W. (2021). Metaoj: A massive distributed online judge system. *Tsinghua science and technology*, *26*(4), 548–557. https://doi.org/10.26599/TST.2020.9010016

Zhou, H., Zhang, H., Zhou, Y., Wang, X., & Li, W. (2018). Botzone: An online multi-agent competitive platform for ai education. *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 33–38. https://doi.org/10.1145/3197091.3197099

Appendix A

List of Links

A.1 Deployed Website

- Main website: https://cs5446.comp.nus.edu.sg
- Administration site: https://cs5446-api.comp.nus.edu.sg
- Web API explorer: https://cs5446-api.comp.nus.edu.sg/swagger

A.2 GitHub Repositories

- Source code
  - aiVLE Gym: https://github.com/edu-ai/aivle-gym
  - aiVLE Grader: https://github.com/edu-ai/aivle-grader
  - aiVLE Worker: https://github.com/edu-ai/aivle-worker
  - aiVLE Web Backend: https://github.com/edu-ai/aivle-web
  - aiVLE Web Frontend: https://github.com/le0tan/aivle-fe
  - aiVLE CLI: https://github.com/edu-ai/aivle-cli
- Raw Experiment Data & Logs: https://github.com/edu-ai/aivle-experiment-logs

A.3 PyPI Packages

- aiVLE Gym: https://test.pypi.org/project/aivle-gym/
- aiVLE Grader: https://test.pypi.org/project/aivle-grader/
- aiVLE Worker: https://test.pypi.org/project/aivle-worker/

A.4 Documentation

- Official documentation website: aiVLE Docs: https://edu-ai.github.io/aivle-docs/
  - Deployment guide: https://edu-ai.github.io/aivle-docs/dev-guide/deployment-guide/
  - User guide: https://edu-ai.github.io/aivle-docs/user-guide/
- Engineering documentation (i.e., inner-workings and design considerations)
  - aiVLE Gym: https://edu-ai.github.io/aivle-docs/dev-guide/assets/design_doc_aivle_gym.pdf
  - aiVLE Grader: https://edu-ai.github.io/aivle-docs/dev-guide/assets/design_doc_aivle_grader.pdf
  - aiVLE Worker: https://edu-ai.github.io/aivle-docs/dev-guide/assets/design_doc_aivle_worker.pdf
  - aiVLE Web: https://edu-ai.github.io/aivle-docs/dev-guide/assets/design_doc_aivle_web.pdf

Appendix B

aiVLE Gym Design

B.1 Multi-agent Communication DFA

In this section we will define the aiVLE Gym communication DFA in a more mathematical (i.e., "rigorous" and detailed) fashion:

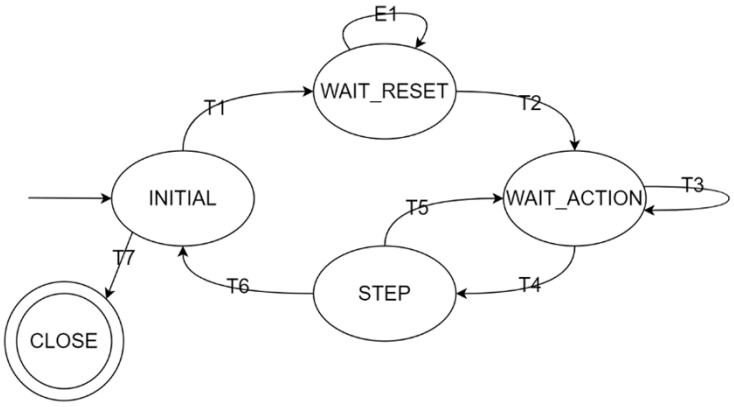

Figure B.1: aiVLE Gym Multi-agent Communication DFA

1. States: INITIAL, WAIT_RESET, WAIT_ACTION, STEP, CLOSE
2. Initial state q_0: INITIAL
3. Accept (terminal) states F: CLOSE
4. Input symbols: method (e.g. reset/step/close) and other conditions

To make this DFA a mathematically rigorous one, the domain of transition function needs to be the Cartesian product of input symbols and states. However, since there are many symbols and states involved, listing them exhaustively takes too much space. For the sake of simplicity, we omitted many self-transitioning paths - if transitioning condition is not satisfied, we assume there's a self-transition path. "Meaningful" transitions are described below:

- T1
  - condition: method is "reset"
  - artifact: reset the underlying base Gym environment, label this agent as already reset, save this agent's router ID
- E1
  - condition: method is "reset" and this sender hasn't reset before
  - artifact: label this agent as already reset, save this agent's router ID, trigger an input symbol of "E1" (This trigger is conceptually equivalent immediately checking if we can transit to the next state. Details can refer to the implementation.)
- T2
  - condition: input symbol of "E1", all agents are labelled as have reset
  - artifact: send initial observation to all agents, clear reset labels, clear router ID mappings
- T3
  - condition: method is "step"
  - artifact: label this agent as already stepped, save this agent's router ID, save this agent's action, trigger an input symbol of "T3"
- T4
  - condition: input symbol of "T3", all agents are labelled as have stepped
  - artifact: step forward in the base environment, send observation/reward/done/info to all agents, trigger an input symbol of "T4"
- T5
  - condition: input symbol is "T4", some of the agents still have ongoing episode
  - artifact: clear "have stepped" labels on all agents, clear router ID mappings
- T6
  - condition: input symbol is "T4", none of the agents still have ongoing episode
  - artifact: same as T5
- T7
  - condition: method is "close"
  - artifact: close the underlying environment

By implementing this DFA carefully, the multi-agent judge environment abstract base class is capable of handling any order of incoming agent requests. Most importantly, agent-side can expect responses synchronously therefore keep all their expectations about a normal single-agent Gym environment. Note that these intricate details are not of users' (both the agent author and environment author) concern. The environment author only needs to provide implementations for the abstract methods and our library will handle the rest.

Appendix C

aiVLE Worker Design

C.1 Comparison of Mainstream Security Solutions

There are several "asterisks" to the claims listed in Table 3.1, which we are going to address in detail in this appendix section.

First, by "rootless" we meant no root privilege is required *after* the initial setup. All solutions listed, including Firejail, requires root privilege to install. However, using it afterwards does not require root privilege.

Second, why in Docker and Podman is considered to be "rootless", but we still say that they "prevent the GPU from being shared by any other container with root access"? It is because both Docker and Podman have a "rootless mode", which allows application to run without root access, however it also forces the GPU to run on rootless mode. Take Docker as an example, Docker Engine 20.10 introduced rootless Docker daemon[21]. But just like Podman, in rootless mode to make NVidia container runtime work you still need to set no-cgroups = true[22], therefore breaks root access from containers[23].

Third, the "level of isolation" of Firejail is labelled as "medium" in the table. In fact, Firejail security entirely relies on how strict the security profile is – by default there is no restriction at all. The recommended approach according to Firejail documentation[24] is to start with a strictest possible profile, then gradually white-listing features until the application inside the sandbox works fine. Since the full lockdown provided by Firejail is very strict, we consider it as a "secure-enough" alternative.

Lastly, VM is labelled to have no GPU support in the table. However, VM does support **exclusive** GPU access by technologies such as PCI passthrough. Exclusivity is the keyword here: we require the solution to work on shared instances, therefore we ignore such infeasible solutions.

[21] https://docs.docker.com/engine/security/rootless/
[22] https://www.redhat.com/en/blog/how-use-gpus-containers-bare-metal-rhel-8
[23] https://github.com/NVIDIA/nvidia-container-runtime/issues/85
[24] https://firejail.wordpress.com/documentation-2/building-custom-profiles/

Appendix D

Proposal for a Multi-agent Tournament System

In the previous version of aiVLE 2.0, we already have infrastructure for running arbitrary evaluation as a `Job` instance. Here we propose an extension to support multi-agent tournament on aiVLE (both matchmaking and evaluation).

This design is very high-level and abstract: even though all relevant code is concrete and runnable, the matchmaking algorithm/strategy, which happens to be the most important and sophisticated part of tournament systems, is not provided. On the other hand, this also means that our design is not limited to any specific tournament type or matchmaking algorithm.

D.1 Tournament Abstraction

Before we dive into the actual API design, we need to understand what is common to all tournaments. A tournament consists of one or many rounds of matches. The participants in every match of a certain round are determined before the round starts, and the order of matches within a round does not matter. After every round, we determine whether the tournament has concluded, or generate the match schedule for the next round.

Here we show the generality of such description by applying it to a real-world championship series: the Overwatch League (OWL)[25] championship:

OWL has regular season matches and playoffs. The schedule for regular season is determined before the season starts, and is not affected by the outcome of any matches within. So we may consider the entire regular season as one round.

The list of teams advanced to the playoffs is determined by the regular season results, so starting the playoffs also means starting a new round. Here we take regular season tournament format of OWL 2021 as an example (Figure D.1):

[25]The Overwatch League is a professional e-sports league for the video game Overwatch. Its structure and operation is similar to other American sports championships such as NBA (National Basketball Association).

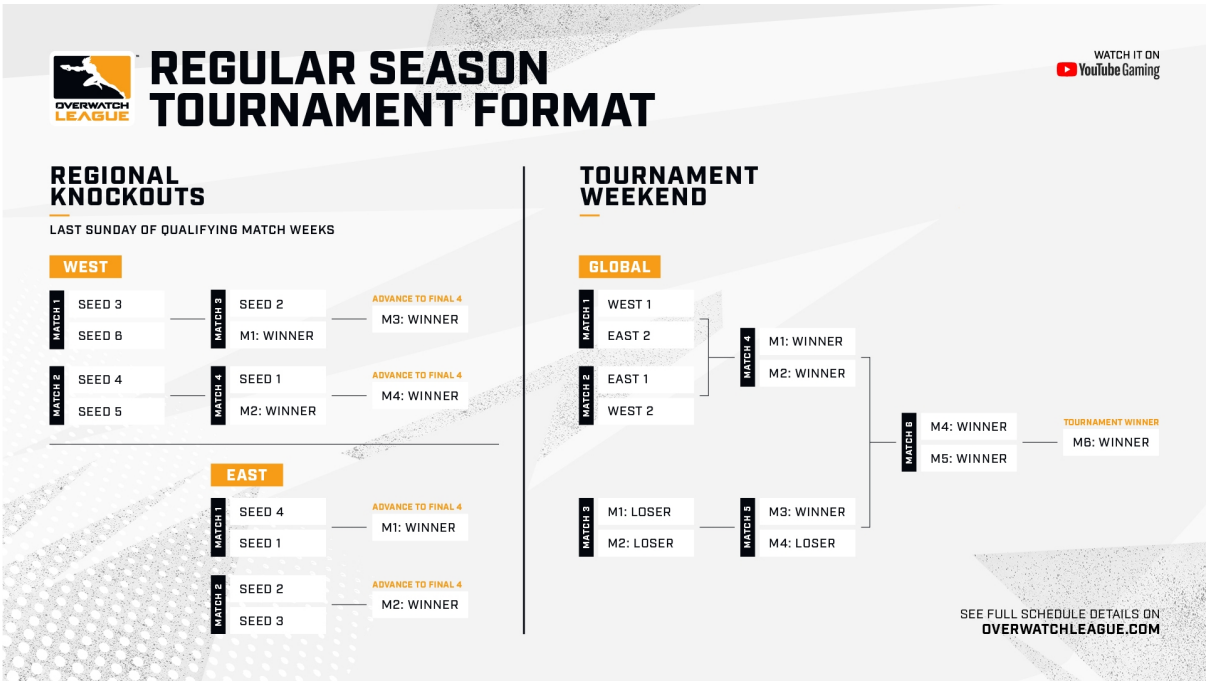

Figure D.1: OWL 2021 Tournament Format

Regional knockouts are single elimination.

- Round 1: (west) match 1, 2; (east) match 1, 2
- Round 2: (west) match 3, 4

Global finals are double elimination.

- Round 3: (global) match 1, 2
- Round 4: (global) match 3, 4
- Round 5: (global) match 5
- Round 6: (global) match 6 - the tournament final

In fact, this description of tournament is so powerful that when we let every round to include only one match, we can implement general skill-rating algorithms such as Elo Rating (Elo, 1978).

D.2 API Design

D.2.1 Overview

Now that we have a powerful abstraction of tournaments, we are ready to describe the API design. In specific, a complete tournament consists of the following steps:

1. `GET /api/v1/tasks/<task_pk>/start_matchmaking` with parameters:

- List of submission IDs
- Matchmaker type (e.g., round-robin)

2. A `MatchmakingSession` object is created and stored in DB
3. `kickstart` method of the `MatchmakingSession`'s `Matchmaker` is called. We get a list of `MatchParticipant`s and `Match`es. We store the `MatchParticipant`s and schedule the `Match`es just like scheduling single-agent evaluation jobs.
4. Every call to `submit_job` RESTful API will trigger a signal[26] to check if the job has a corresponding `MatchmakingSession`[27]:
   (a) If yes, check if all `pending_jobs` in the `MatchmakingSession` are finished
   (b) If also yes, call `schedule` method of the `MatchmakingSession`'s `Matchmaker` instance
      i. Schedule the returned matches by creating `Job`s if necessary - at the same time reset `pending_jobs` of the `MatchmakingSession`
      ii. Update the ratings of the `MatchParticipant`s using the return value
      iii. If concluded, set `ongoing` of the `MatchmakingSession` to `False`

D.2.2 Models

There are two new models required for multi-agent tournament: `MatchParticipant` and `MatchmakingSession`. In addition, we need a `Match` wrapper class to represent the list of participants involved in a match. However, unlike the other two models that need to be stored in the database, `Match` is temporary. The concrete implementation of these models is shown in Code D.1.

[26] For more details, please check https://docs.djangoproject.com/en/4.0/topics/signals/

[27] For backward compatibility (i.e., supporting evaluation jobs that is not part of any tournament), we allow a `Job` to have no corresponding `MatchmakingSession`.

```python
from django.db import models

class MatchParticipant(models.Model):
    submission = models.ForeignKeyField(Submission)
    rating = models.FloatField()


class MatchmakingSession(models.Model):
    participants = models.ManyToManyField(MatchParticipant)
    pending_jobs = models.ManyToManyField(Job)
    ongoing = models.BooleanField()

class Match():
"""
Match is a list of submission IDs corresponding to the participants of this match.
"""
    def __init__(self, ids: List[int]):
        self.ids = ids
```

Code D.1: Multi-agent Tournament Models

D.2.3 Changes to Existing Models

1. `Job` model needs to support more than one related `Submission`s - this can be easily achieved by changing `ForeignKeyField` to a `ManyToManyField`.

2. `Job` needs to bind to a `MatchmakingSession` so that when the `Job` is submitted, the `Matchmaker` can be notified. One possible way:
   `matchmaking_session = models.ForeignKeyField(MatchmakingSession, null=True)`

D.2.4 Matchmaking Logic Abstraction

As shown in Section D.2.1, the decision-making of the entire matchmaking process is abstracted as the `kickstart` and `schedule` method of a `Matchmaker`. This is exactly where the tournament rules and/or matchmaking algorithm should be implemented. Below is an abstract base class for `Matchmaker`:

```python
class BaseMatchmaker():
"""
Matchmaker class must be stateless - every time a new instance of Matchmaker will be created.
"""
    def kickstart(self, participants: List[MatchParticipant]) -> (List[MatchParticipant], List[Match]):
        pass

    def schedule(self, participants: List[MatchParticipant], jobs: List[Job]) -> (bool, List[MatchParticipant], List[Match]):
        pass
```

Code D.2: `Matchmaker` Abstract Base Class

Explanation of input parameters:

- Pool of participants - in our case, each participant has
  - Submission ID
  - Current rating/score
- Match history, in particular, every match should at least have
  - IDs of all participating submissions
  - Outcome of the match

Explanation of the outputs:

- Whether this tournament has concluded or not
- List of updated rating/score for all submissions
- List of scheduled matches (if the tournament hasn't concluded yet)

Appendix E

Deployment Issues and Solutions

E.1 RabbitMQ May Be Blocked by the Firewall

If the worker node resides behind a firewall that restricts access to certain ports (especially under a whitelist policy where only selected ports are available), then you are likely to find RabbitMQ unusable - its underlying protocol, Advanced Message Queuing Protocol (AMQP), uses port 5671/5672 by default. There are three possible solutions:

1. Change RabbitMQ port to one that is allowed by the worker node firewall. Do note that AMQP is not based on HTTP/HTTPS, so before changing the port to 80/443 you need to ensure that not only the broker server has those ports available, but also that both the master server's and worker nodes' firewall allows non-HTTP traffic via port 80/443.

2. Deploy RabbitMQ internally. If the firewall only blocks external access and allows internal access to all ports (like SoC firewall), and all worker nodes reside within the local network, then deploying the RabbitMQ inside the local network would be an uncompromising[28] solution.

3. Switch RabbitMQ to HTTP/HTTPS-based broker such as AWS SQS. Do note that this solution greatly affects the capability of Celery task scheduling - for example, remote worker control is impossible with SQS. As a result, later advanced features like resource-sensitive load balancing would not work under SQS.

In our case, the SoC firewall blocks most ports on external IP address, and forcing RabbitMQ to use port 80 was futile. In the end we switched to AWS SQS as a workaround. It did not affect any existing functionality then - features such as resource-sensitive load balancing came later in the second semester.

[28] Given you have enough privilege to install RabbitMQ on one of the machines, which I did not have at the time of these experiments. In late March 2022, Dr. Narayan finally managed to have SoC approve an instance with `sudo` root access for me, and we confirmed this approach works as expected internally.

E.2 Firejail Version Requirement

Firejail security profile is not forward compatible, and the latest Firejail version is determined by the OS version, therefore the default security profile of aiVLE Worker does not work on `xgpd0` which has Ubuntu 16.04 installed. It works as expected on both Ubuntu 18.04 and Ubuntu 20.04 with their latest Firejail version respectively.

To avoid future confusions, we listed Ubuntu 20.04 and Firejail 0.9.62 as requirements for aiVLE Worker on its Read-me.

E.3 PyTorch Installation Issue with Latest nVIDIA GPU

Although it has been two years since the launch of RTX 30-series GPU[29], PyTorch official PIP channel still has not supported CUDA 11 (which is the minimum CUDA version for 30-series GPU). So instead of

```
pip3 install torch torchvision torchaudio
```

, we need to use

```
pip3 install torch==1.10.1+cu113 torchvision==0.11.2+cu113 torchaudio==0.10.1+cu113 -f
↪  https://download.pytorch.org/whl/cu113/torch_stable.html
```

This means the author of the task needs to be aware of the CUDA version of their allocated grading machines, and adjust their `requirements.txt` accordingly.

[29]The same applies to data center GPUs launched after RTX 30-series, such as nVIDIA A100 used in our deployment.