aiVLE Web - AI competition platform

Website: aiVLE - AI Competition Platform

API: Api Root – Django REST framework

Admin panel: Django site admin

Frontend repo: GitHub - le0tan/aivle-fe: React frontend for aiVLE

Backend repo: GitHub - edu-ai/aivle-web: Django backend for aiVLE (AI Virtual Learning Environment)

Recall the three goals of aiVLE:

- Creating reinforcement learning environment
- Evaluating reinforcement learning algorithms (i.e. agents)
- Hosting agent/bot competitions

aiVLE is the final piece - a web application to host and participate in AI competitions. There are three primary considerations to the design of aiVLE web: extensibility, scalability and usability.

Extensibility

Frontend-backend separation

// allows complex frontend interactions/optimizations, especially real-time interaction

Decouple submission from evaluation

// basically in multi-agent case, one submission will correspond to many evaluations (matches)

Real-time communication extension

// move some of the original aiVLE worker design stuff to here...

Scalability

How to balance scalability and operation cost is probably one of the most challenging problems to solve. On the one hand, a centralized database or web application can only be scaled up (i.e. having a more powerful server), which is sometimes not viable. On the other hand, decentralizing the database by sharding/duplicating, and/or distributing load to multiple instances of the same web application incurs operational costs and complicates the

deployment/upgrade process. By analyzing the characteristics of our specific application, we can conclude the following:

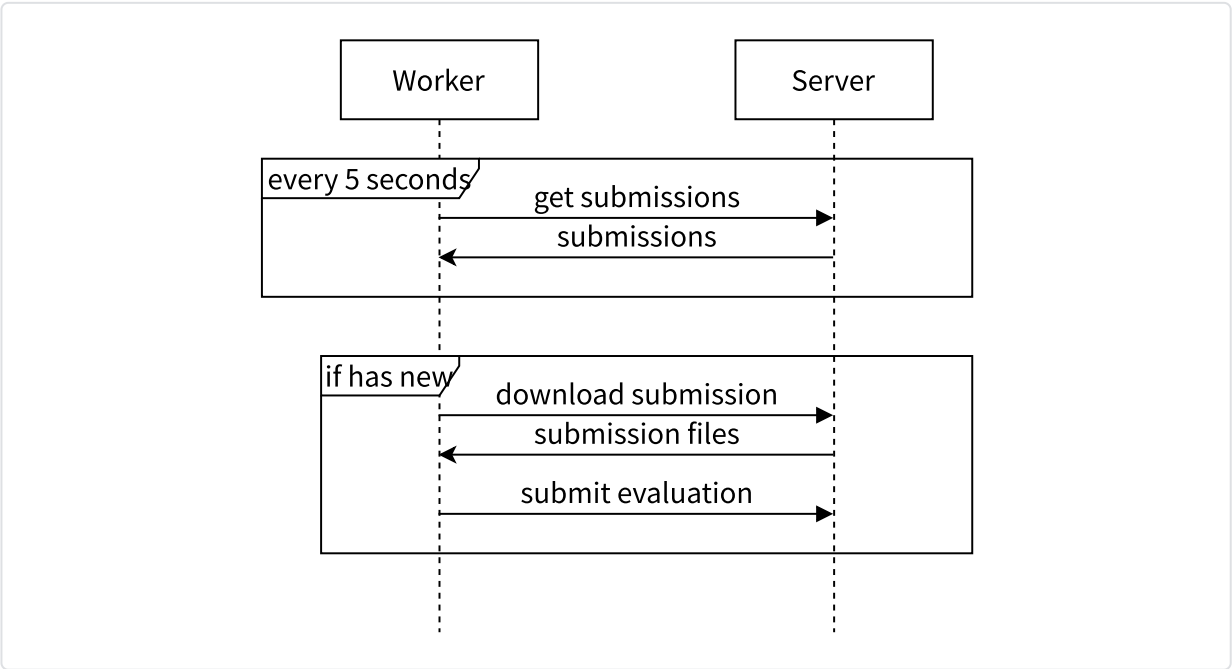
1. Most computation time is spent on grading.
    a. Implication: there will be many grading workers that need to be utilized/coordinated by the backend efficiently.
2. Most user-side interaction is read only and not time-sensitive (up to tens of seconds)
    a. Implication: client-side caching will greatly reduce the server load.
3. Traffic peak is predictable. Most students will not spam the website when there's no assignment to submit. The traffic almost always peaks during the submission period of certain tasks, especially several days before the deadline.
    a. Implication: no need for automatic scaling up. But manually scaling up should be easy enough for every lecturer to perform on their own.

High availability task queue

This is a continuation of 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 many workers and the single backend, 2) distributing grading tasks to the workers efficiently for shorter waiting time and higher resource utilization.

Old implementation (polling)

In original aiVLE, there was no concept of "task queue". All pending tasks are stored in the DB with the status "QUEUED". Communication between worker (or runner as per the term used by original author) and server is half-duplex by polling. In other words, the worker makes a periodic request to the server for new ungraded submissions:

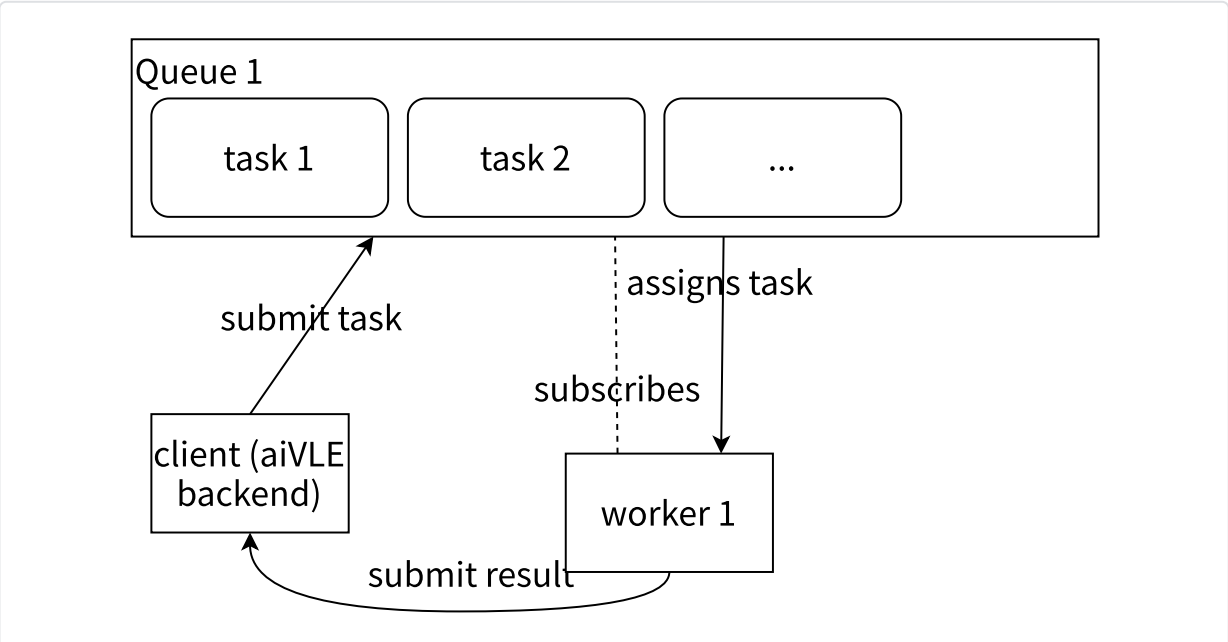


There are three critical limitations to this apprach:

1. Worker-side polling doesn't scale well: there is zero mechanism in orchestrating the timing and order of workers polling the server for new jobs. The severity of possible traffic peak (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. Possible race condition: if two worker pull submissions at the same time, both of them will get the same ungraded submission, and there's no mechanism to prevent them from doing duplicate work. This doesn't cause big trouble because we currently have very few worker nodes, and each node doesn't have concurrency (see point 3). But such race condition will severely affect scalability and efficiency when we try to add more nodes.
3. No concurrency or load balancing on workers: currently, each 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. And because the backend has no control over which worker grades which submission, load balancing is impossible.

New implementation (message queue)

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. Conceptually, message queue enables asyncronous communication between clients (who submit tasks) and workers (who finish tasks). Below is a diagram illustrating how the MQ-based task queue works:



Every worker listens to one or more "queues", where the message broker will be responsible for allocating tasks fairly among workers available for each queue. When an evaluation job comes, aiVLE backend will submit the job to an appropriate queue (i.e. private queue if user has dedicate workers available, otherwise public CPU/GPU queue according to task specification) and wait for the assigned worker to submit evaluation result. 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 # of requests to be O(n) where n is # of evaluation jobs, but also ensures fair assignment of tasks among workers.

Additional benefits:

1. Worker-level time limit enforcement: Setting Time Limit on specific task with celery - Stack Overflow
2. Heartbeat check on workers (workload on broker, not on aiVLE backend): Monitoring and Management Guide - Celery 5.1.2 documentation
3. Automatic retry of tasks (so that aiVLE backend can have a much longer interval of checking the submission queue for jobs that failed because of workers instead of faulty submission): Calling Tasks - Celery 5.1.2 documentation

(NEW) 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 cases, 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, 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 entirely ours to blame.
2. Inefficiency: suppose the task queue is configured to automatically 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 to finally arrive 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 of the heavy-lifting is handled on the worker side, for aiVLE Web, it merely uses \texttt{add_consumer} and \texttt{cancel_consumer} Celery APIs for resuming and pausing consuming respectively. There are two points to note in this solution:

1. It does not cancel already running jobs - it only stops receiving new tasks when the threshold is met. This behavior is acceptable in our case since we only need to promise students with a certain amount of resources to run their submissions. And our solution guarantees\footnote{Technically speaking this guarantee is not true 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.} 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. \texttt{cancel_consumer} simply removes the worker from the list of consumers to dispatch tasks to. To achieve resource utlization balance, aiVLE Web needs to understand each worker's utlization in real-time and dispatch tasks accordingly. We think the overhead and complexity of this design significantly outweights the potential benefits.

Single page application (SPA) frontend

One significant shortcoming of (dynamic) server-side rendered page (e.g. Django templates) is that if user refreshes the same page repeatedly, every refresh will be a request to the server for a new HTML response. We can of course add caching in front of database layer to avoid stressing the DB, but nonetheless a lot of traffic will arrive at the server, potentially crash the not-so-powerful SoC allocated VMs. SPA makes client-side caching and spam prevention very easy: SPA makes it possible to implement a local cache such that even if the user keeps refreshing the page, API request to the backend will always be client-side rate limited.

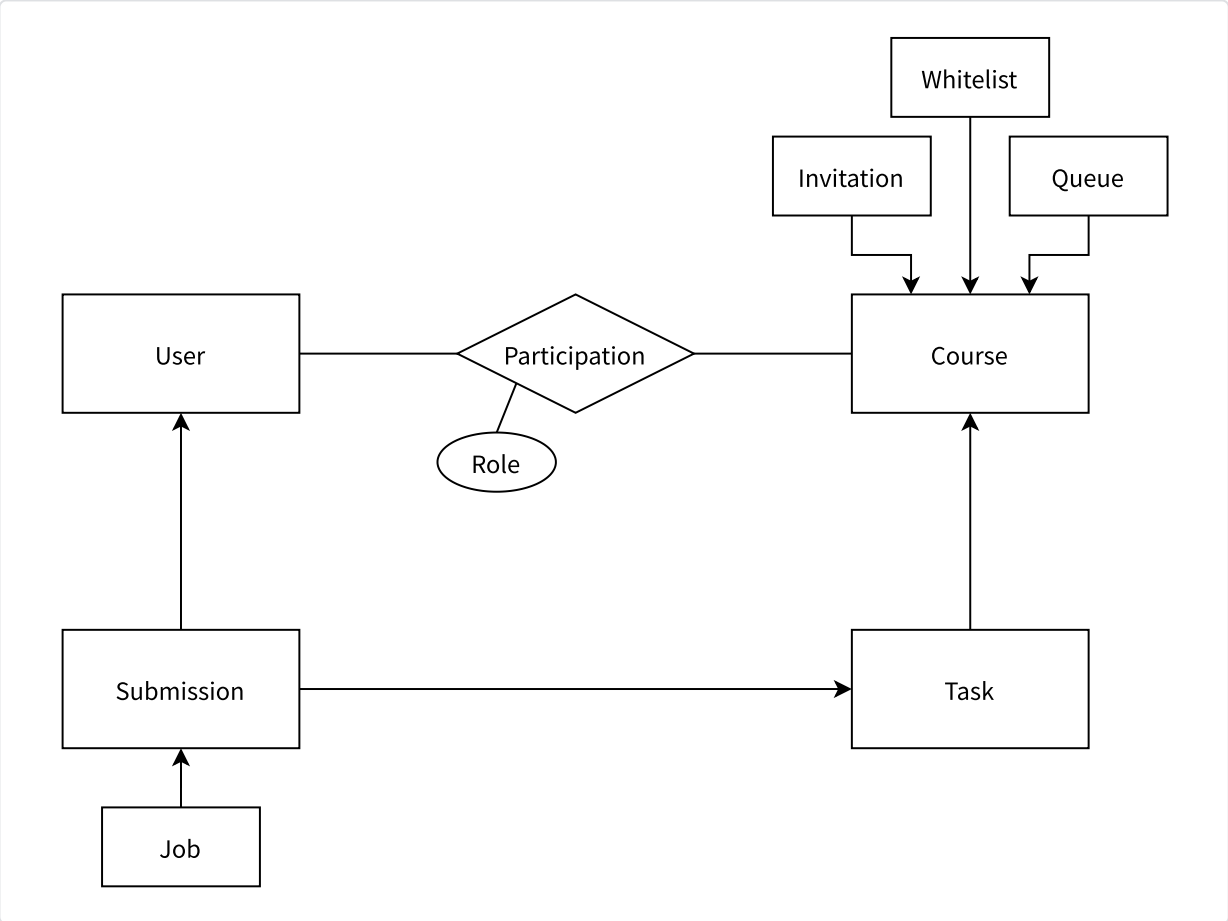
Usability

(UPDATED) 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, meaning 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.

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 too overkill as well: 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\footnote{This can be avoided by properly configuring django-guardian, but its flexibility does allow arbitrary permission on any object as mentioned earlier. In fact, if we restrict django-guardian to avoid such wasteful behavior, we essentially have the soon-to-be-discussed role-based permission model.}. Therefore, we propose a sweet spot between the two extremes: role-based permission model.

The entity relationship diagram of aiVLE Web is illustrated below:



In the relational entity Participation, we record the "role" of the User in a Course. Possible roles (as defined in `app.model.participation`) are: admin (ADM), lecturer (LEC), teaching assistant (TA), student (STU), guest (GUE).

> ❗ Note that a user can have different roles in different courses.
>
> Superuser who has access to absolutely everything is out of discussion here.

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\footnote{The utility function \texttt{has_perm} automatically queries the role given the course and the user, so its argument is the course instead of the role.}, and check if this role has \texttt{job.view} permission according to the permission lookup table.

By introducing the Participation relational entity and \texttt{has_perm(course, user, permission)}, we compressed the worst case $O(mnk)$ space complexity to $O(nk)$ and it is a huge improvement - realistically the number of objects significantly outweighs the number of users or permissions. We can summarize different roles' access (as defined in `aiVLE.settings.ROLES`) using a permission matrix:

		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	x		

(NEW) Superuser-only operations

For security purposes, we reserve the Django admin panel only for the superuser. The superuser is allowed to view and edit anything stored in the database (not the passwords since they're salted hashes).

Nearly all operations can be done via the RESTful API, and all such operations are access controlled. However, there are still some operations that can only be performed by the superuser via the Django admin panel.

- Creating a course
- Assign the first admin to a course

> ❗ As long as there is at least one admin or lecturer inside the course, that user is able to invite new users into the course by creating an invitation token. And that user can also change the role of any participating user other than himself.