aiVLE Worker - Secure & Scalable Grading Client

GitHub repo: https://github.com/edu-ai/aivle-worker

Objectives

Security

Any system that deals with user inputs needs to sanitize them before letting these arbitrary inputs take any effect. In this sense, an online judge system like aiVLE deals with the most dangerous user input possible - it needs to execute arbitrary code uploaded by users. Therefore a robust and foolproof security solution is necessary to ensure not only the fairness of judging, but also nominal operation of aiVLE services.

What we expect from this solution is:

- File access restriction: agent program should
  - have no access to directories that may contain sensitive data (e.g. private keys)
  - have read only access to files necessary for its execution (e.g. Python binary, dependencies, agent and environment source code)
- Network restriction: agent program should have no access to Internet. Otherwise, (1) it may have extra computation resources by delegating the task to an external party; (2) it may send out confidential execution details (e.g. configuration of simulation environment) that allows users to fine-tune their program; both of which makes the competition unfair.
- Resource limit
  - RAM limit
  - CPU affinity (core limit)
- Convenient dependency setup

Scalable

Scalability depends on both the scheduling on the server-side and the execution on the worker-side. In the context of scaling workers easily, a client that requires little permission and setup would be extremely helpful - when the surge of submission arrives, we don't need to pay a hefty bill to buy a lot of cloud servers, nor need to apply for additional Tembusu nodes. Reserving one of the labs with PCs powerful enough to run submissions should be able to save the day.

Therefore, what we expect from this solution is:

- Minimal permission requirement: as long as we have access to run the submission locally, the environment should be able to operate as a worker node.
- Minimal dependency requirement: any Linux machine with Python/Virtualenv/Pip should be able to run the worker client without additional dependencies.
- 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, crazy warmup time like several minutes is still unacceptable.

Comparison for mainstream security sandbox solutions

There are three mainstream methods of sandboxing user submission executable:

- Virtual machine (Virtualbox)
- Container (Docker, Podman)
- Sandbox (Firejail)

The main areas of interest are compared in the table below:

	VM	Container		Sandbox
	Virtualbox	Docker	Podman	Firejail
Rootless*	No	No	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

- Rootless: all four solutions require root access to install.
  - Docker requires a daemon to run with root access and there's a known security issue where any user with "docker" group access is effectively granted root access (ref: https://keiran.scot/privilege-escallation-with-docker-56dc682a6e17).
  - Podman requires root to install but it has a dedicated rootless mode (ref: https://github.com/containers/podman/blob/main/docs/tutorials/rootless_tutorial.md). However, to run CUDA application under Podman rootless mode, the admin needs to modify the config of NVIDIA container runtime (ref: https://www.redhat.com/en/blog/how-use-gpus-containers-bare-metal-rhel-8). This will make the shared GPU to run exclusively on rootless mode (i.e. Docker will lose access to GPU).

    - Firejail doesn't require any privilege.
- Level of isolation:
    - VMs uses hardware-level virtualization thus they have the highest level of isolation.
    - Containers and sandboxes like Fire jail uses OS-level virtualization like Linux namespaces and cgroup. The difference between the two is: containers are configured such that by default host OS is almost completely isolated from the containers (e.g. from inside the container you have no access to host OS files by default). Sandboxes like Firejail relies on "security profile" to manually configure areas of isolation - without any profile, Firejail does nothing at all!
- GPU support
    - Traditionally, VMs like Virtualbox don't have access to devices like GPU. However, a technology called PCI passthrough has emerged in recent years (ref: https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/pcipassthrough.html#:~:text=The%20PCI%20passthrough%20module%20is,drivers%20for%20this%20particular%20device.) that allows allocation of a GPU to a VM. The caveat is that PCI passthrough requires the device to be allocated **exclusively** to a VM instance.
    - Containers are more flexible in terms of GPU support. With a special NVIDIA container runtime, containers are able to share GPUs while keeping their OS-level virtualization.
    - Firejail executes binary just like a normal un-sandboxed program - anything that runs on the host OS will run just fine in Firejail.

Why Firejail is chosen

There are several requirements our solution needs to meet:

1. GPU-support is a must. Otherwise many agents simply become inaccesible.
    a. This eliminates VM without PCI passthrough.
2. Root access (after installation) is not available. SoC cluster admin doesn't allow that.
    a. This eliminates Docker as being able to create Docker conatiner is equivalent to having root access.
3. Server needs to be shared.
    a. This eliminates VM with PCI passthrough as it requires exclusive access.
    b. This also eliminates Podman

Therefore, Firejail is our only sensible security solution.

Design Considerations

Empty

Dependency management

Virtualenv

> ❗ How to use different Python versions?
>
> Ans: specify /usr/bin/Pythonx.x when creating virtualenv

Security profile

> ❗ Why share certain files? (i.e. /tmp directory)
>
> Ans: avoid too much disk operation (e.g. frequently copying PyTorch wastes too much time and hurts the longevity of hard disks)

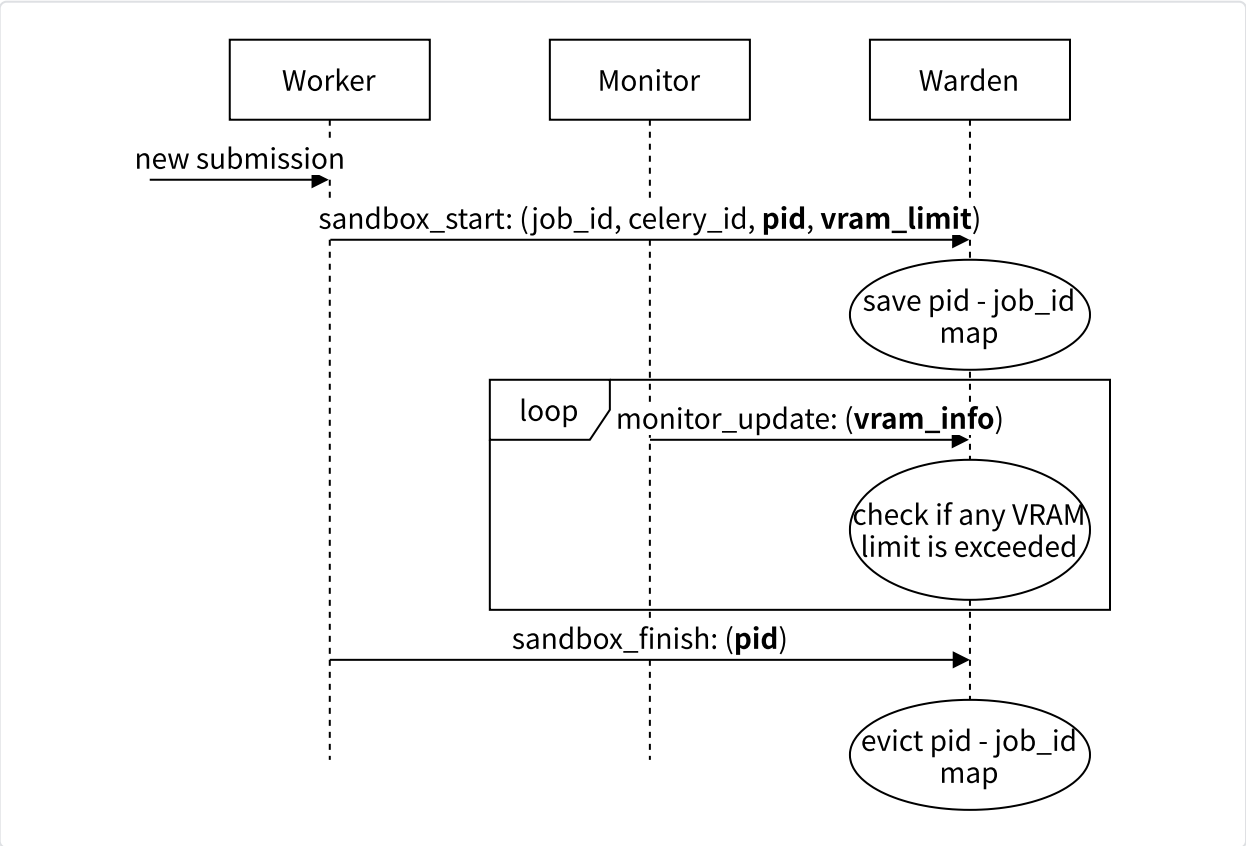
(NEW) Resource monitoring - `monitor` module

To achieve resouce-sensitive load balancing as described in aiVLE Web section, we need to monitor CPU/GPU utlization, RAM/VRAM usage in real-time and control the worker's subscription to the task queue accordingly in real time. Therefore we implement the \texttt{monitor} module inside aiVLE Worker that runs in parallel with the main worker process, which

1. Monitors the system utilization periodically
2. Control the worker's task queue subscription according to prefetched threshold
3. Send monitoring metrics to the \texttt{warden} module to enable resource limit enforcement

From the description above, you may already realize that there are three processes that run in parallel: worker, monitor and warden. And they need to exchange information in order to collaborate. This is where ZeroMQ comes handy (again): 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 violates restrictions. Below is a diagram for the inter-process communication among these three:



Libraries

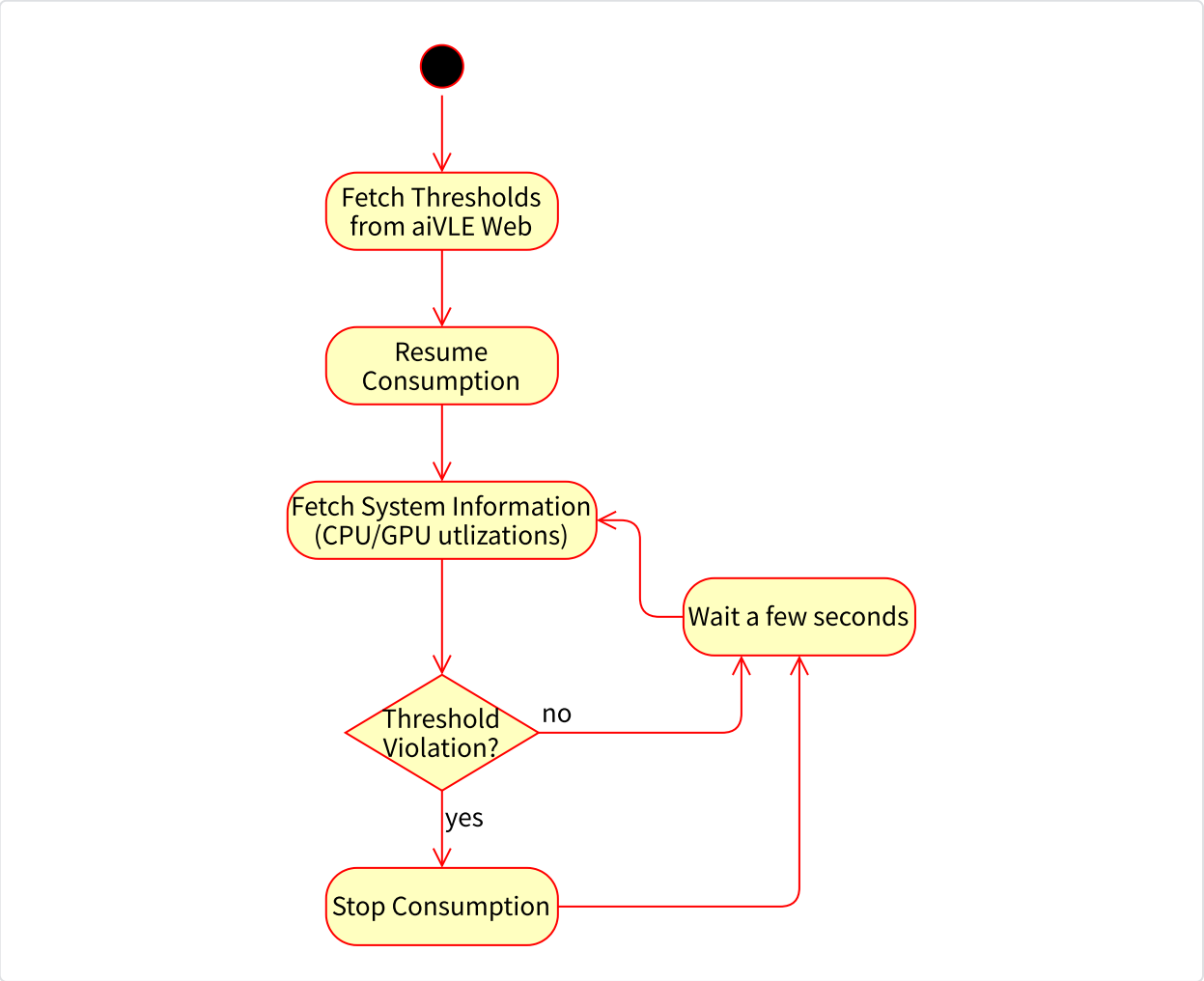
- GitHub - giampaolo/psutil: Cross-platform lib for process and system monitoring in Python
  - Monitor CPU and RAM
- py3nvml
  - Monitor nVIDIA GPU usage and VRAM usage

Basic Idea

- In a separate process
  - **Reason**: `worker_main` function which starts the main worker thread is blocking
  - Use `psutil` and `py3nvml` to monitor resource utilization periodically
  - Whenever a pre-defined threshold is reached, **call an API in aiVLE Web** to let it stop sending task to this worker
    - celery.app.control — Celery 5.2.3 documentation - `add_consumer`
  - When the utilization falls below a threshold, **call an API in aiVLE Web** to let it resume sending task to this worker
    - celery.app.control — Celery 5.2.3 documentation - `cancel_consumer`

Details

Below is a diagram illustrating the execution flow of the monitor process:



There are a few points to consider:

- How to fetch the "pre-defined threshold"?
  - Ans: bind minimum CPU/GPU/VRAM to the evaluation queue, pre-defined threshold is the maximum of all the queues this worker is listening to.
- How about monitoring the behavior of one specific submission?
  - Ans: *it depends*. We can easily restrict the total memory and CPU time by using Firejail's `--rlimit-as=number` and `--rlimit-cpu=number` respectively.

    However, when it comes to GPU utilization rate and VRAM usage, it gets complicated - from OS design point of view, GPU is a non-essential device that relies on proprietary drivers, therefore we must use nVIDIA tools to impose these limits. nVIDIA doesn't show individual utilization by process, therefore such limit is practically impossible. nVIDIA does provide VRAM breakdown by process, so here's the rough plan of how to get total VRAM usage of a certain submission.
    - From `aivle-worker/tasks.py` we know that `evaluate(job_id)` is the Celery function that's called whenever a task is assigned to the worker. So we let `run_submission()` receive an additional argument `job_id`. After getting the

PID of newly created sandbox, it sends PID, job ID, VRAM limit via ZeroMQ to the monitor process.

- The monitor process periodically checks the total VRAM usage of each PID (to prevent workaround of spawning subprocesses, we need to monitor all child processes), and kills processes that exceed the VRAM limit.
- After killing the parent process, the monitor also needs to report failure to aiVLE Web

(NEW) Resource limit enforcement - `warden` module

Celery doesn't allow the worker itself to stop consuming tasks - the worker can only choose to shutdown itself entirely. And the \texttt{monitor} module is periodically checking the system load, so we need to have the \texttt{warden} module that is responsible for enforcing the resource limits by terminating the job externally and report the reason to aiVLE Web. As mentioned above, both the main worker process and the monitor process send information to the warden process. In particular, it receives

1. (from worker) mapping from sandbox PID to corresponding task information
2. (from monitor) VRAM usage of each process

Everytime it receives an update from the monitor, it goes through the list of active sandboxes, figuring out process hierarchy\footnote{This is necessary as most frameworks like PyTorch will spawn new processes for calculation. Generally the process that is utilizing GPU is no longer the parent process. And there might be multiple processes in the same sandbox that consumes VRAM.} to calculate the total VRAM usage of each sandbox, and finally checks if any sandbox needs to be terminated.

TO-DO

- [x] ~~Worker node resource monitoring (GPU/CPU/RAM usage report)~~
- [x] ~~venv: manage local virtual environments (given Python version and requirements.txt, download virtual env;~~
- [x] ~~firejail: run downloaded agents and Gym in a specified venv~~
- [x] ~~client: communicate with aiVLE web server~~