Modal Compute Guide
CS224R (Spring 2026)

Adapted from CS336 Spring 2026 Modal Compute Guide

1 Purpose and scope

This document is a practical introduction to Modal for running Python (especially GPU) workloads.

What this guide covers: authentication, defining images and apps, running functions remotely, volumes, parallelism (`map` / `starmap`), GPUs, secrets, shells, and finding logs in the Modal UI.

2 Quickstart

1. Create a Modal account (e.g. sign in with Google using your Stanford email, unless staff say otherwise).
2. Redeem the course compute code from the email from the teaching team with subject **Your CS224R Modal Compute Credits** via https://modal.com/credits (if you have not received credits yet please fill out `https://forms.gle/gYkTgAqkEvd9Ybcj7`.
3. Install the CLI in the project where you work:

```
uv add modal
# or: pip install modal
```

4. Authenticate:

```
uv run modal setup
# or: modal setup
```

5. Run a small script locally, then the same file with `modal run` once you have wrapped work in `@app.function` / `@app.local_entrypoint` (see below).

3 Security and good practice

- Do not commit API keys, passwords, or private dataset paths into `git`. Prefer Modal secrets for tokens.
- Your Modal account and runs are yours; still follow Stanford's honor code and each assignment's collaboration rules for code and writeups.

4 Turning a local script into a Modal job

A practical recipe:

1. Keep underlying Python logic unchanged when possible.
2. Define a `modal.App` and an `Image` with the packages you need (PyTorch, Gymnasium, simulators, etc.).
3. Decorate expensive work with `@app.function(...)`. Set CPU or GPU, volumes, and timeouts as needed.
4. Add `@app.local_entrypoint()`. Start it from your terminal with `modal run`, using the script path your handout gives.
5. From that entrypoint, call `.remote(...)`, `.map(...)`, or `.starmap(...)` so heavy work runs remotely, not inline on your laptop.

4.1 Fill-in-the-blanks template

Most of the assignments in this class look the same from Modal's point of view: there is a training script you already run locally, and you want the whole thing to happen on a remote machine—maybe with a GPU—and you want the logs to survive after the job ends. The template below handles all of that. Every line marked `# <-- CHANGE` is something we've swapped out in the homework

```
import modal
import subprocess, shutil

app = modal.App("cs224r-hwN")  # <-- CHANGE

# 1) Image -- your pip (and optionally apt) dependencies.
#    Check the requirements.txt or conda_env.yml in your hw repo.
image = (
    modal.Image.debian_slim(python_version="3.10")
    # If the assignment uses MuJoCo or offscreen rendering, uncomment:
    # .apt_install("libgl1-mesa-glx", "libosmesa6-dev", "patchelf")
    # .env({"MUJOCO_GL": "egl"})
    .pip_install(  # <-- CHANGE: whatever is in your conda_env.yml (or
       alternatively you can make a requirements.txt and just list that)
        "torch", "numpy", "gym==0.25.2", "tensorboard",
    )
)

# 2) Volume -- this allows checkpoints to persist after container shutdown
   .
volume = modal.Volume.from_name("cs224r-results", create_if_missing=True)

# 3) Mount -- ships your local homework folder to the container.
code = modal.Mount.from_local_dir(
    ".",  # <-- CHANGE: path to your hw folder on your laptop
    remote_path="/root/hw",
)

@app.function(
```

```
    image=image,
    mounts=[code],
    volumes={"/output": volume},
    gpu="T4",         # <-- CHANGE: to another GPU type if you want
    timeout=3600,     # <-- CHANGE: how long (seconds) before Modal kills
        the run
)
def train():
    import os
    os.chdir("/root/hw")

    # <-- CHANGE: whatever command you would run on your own machine
    subprocess.run(
        ["python", "your_script.py", "--your_flag", "value"],
        check=True,
    )

    # Copy the output directory your script wrote to the persistent volume
        . This is the directory that will persist after the container shuts
         down.
    shutil.copytree("data", "/output/hwN_run", dirs_exist_ok=True)  # <--
        CHANGE
    volume.commit()

@app.local_entrypoint()
def main():
    train.remote()
```

Then just run `modal run your_modal_file.py`. Your code goes to the container through the mount, training runs there, and the results land on the volume. You can look at them afterwards with `modal shell` or from another job.

Tip: detaching long-running jobs. For long-running experiments (e.g., Atari, HalfCheetah), you can use `--detach` to run in the background so you don't need to keep your terminal open:

```
uv run modal run --detach src/scripts/modal_run_dqn.py -- --cfg
   experiments/dqn/mspacman.yaml
```

The job will continue running on Modal's servers even after you close your terminal. You can monitor its progress from the Modal dashboard.

4.2 Volumes and persistence

The local disks of workers running Modal code are ephemeral. After a job finishes, all the data you save is deleted. To persist files, you use Modal Volumes . This is where you put your datasets, preprocessed data, checkpoints, and evaluation outputs.

```
import modal
from pathlib import Path

app = modal.App("volume-demo")
data_vol = modal.Volume.from_name("my-volume", create_if_missing=True)
image = modal.Image.debian_slim()
```

```
@app.function(image=image, volumes={"/data": data_vol})  # mount data_vol
    at /data; files there persist for later jobs
def write_data() -> None:
    Path("/data/data.txt").write_text("this will be persisted")
    data_vol.commit()
```

We highly suggest you read the official volume guide for your Modal version.

4.3 Runnning Jobs in Parallel

If the same function should run independently on many inputs, use `.map` .

```
@app.local_entrypoint()
def main():
    for path in download_or_train.map(list_of_jobs):
        print(path)
```

This is cleaner than manually looping over .remote(...) calls, and it makes the "one function, many inputs" pattern obvious.

Large sweeps can spend credits quickly and may queue during busy periods. Use `max_containers` to cap how many workers run at once:

```
@app.function(image=image, volumes={"/data": data_vol}, max_containers=3)
def run_experiment(seed: int):
    ...
    return result

@app.local_entrypoint()
def sweep():
    for r in run_experiment.map(range(12)):
        print(r)
```

5 Example: toy behavioral cloning with a placeholder environment

Heere is an example of how to code you've already written to run on Modal.

Local script (single run):

```
import gymnasium as gym
import numpy as np
from sklearn.tree import DecisionTreeClassifier

def collect_expert_data(env_id: str, n_episodes: int = 50):
    env = gym.make(env_id)
    observations, actions = [], []
    for _ in range(n_episodes):
        obs, _ = env.reset()
        done = False
        while not done:
```

```
            action = 1 if obs[2] > 0 else 0
            observations.append(obs)
            actions.append(action)
            obs, _, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
    return np.array(observations), np.array(actions)

def main():
    env_id = "YOUR_ENV"
    obs, actions = collect_expert_data(env_id, n_episodes=50)
    model = DecisionTreeClassifier(max_depth=5)
    model.fit(obs, actions)

    env = gym.make(env_id)
    obs, _ = env.reset()
    total_reward = 0
    for _ in range(500):
        action = int(model.predict([obs])[0])
        obs, reward, terminated, truncated, _ = env.step(action)
        total_reward += reward
        if terminated or truncated:
            break
    print("Total reward:", total_reward)

if __name__ == "__main__":
    main()
```

Modal version: parallel hyperparameter grid. Each remote call is self-contained (collect, train, evaluate), so you do not ship large tensors from your laptop. `.starmap` dispatches many jobs subject to your limits and `max_containers`. A small Modal Volume stores one JSON file per config under `/data/runs/` so you can inspect past sweeps from a `modal shell` or a later job without re-running.

```
import json
import modal
import numpy as np
from pathlib import Path

app = modal.App("bc-sweep-demo")  # logical grouping for Modal dashboard

bc_results = modal.Volume.from_name(
    "bc-sweep-demo-results", create_if_missing=True
)  # durable disk shared across runs (not the ephemeral container FS)

image = (
    modal.Image.debian_slim()
    .pip_install("gymnasium", "scikit-learn", "numpy")
)  # dependencies installed on the remote worker image

# Body runs on Modal; /data is the volume mount; max_containers caps
   concurrent workers
@app.function(image=image, volumes={"/data": bc_results}, max_containers
   =3)
def train_and_eval(n_episodes: int, max_depth: int) -> dict:
```

```python
    import gymnasium as gym  # import here so the worker image resolves
        them
    from sklearn.tree import DecisionTreeClassifier

    env_id = "YOUR_ENV"

    def collect_expert_data(n_episodes):
        env = gym.make(env_id)
        observations, actions = [], []
        for _ in range(n_episodes):
            obs, _ = env.reset()
            done = False
            while not done:
                action = 1 if obs[2] > 0 else 0
                observations.append(obs)
                actions.append(action)
                obs, _, terminated, truncated, _ = env.step(action)
                done = terminated or truncated
        return np.array(observations), np.array(actions)

    obs, actions = collect_expert_data(n_episodes)
    model = DecisionTreeClassifier(max_depth=max_depth)
    model.fit(obs, actions)

    env = gym.make(env_id)
    obs, _ = env.reset()
    total_reward = 0
    for _ in range(500):
        action = int(model.predict([obs])[0])
        obs, reward, terminated, truncated, _ = env.step(action)
        total_reward += reward
        if terminated or truncated:
            break

    result = {
        "n_episodes": n_episodes,
        "max_depth": max_depth,
        "reward": float(total_reward),
    }

    run_dir = Path("/data/runs")  # lives on the mounted volume, not local
        disk
    run_dir.mkdir(parents=True, exist_ok=True)
    out_path = run_dir / f"n{n_episodes}_d{max_depth}.json"
    out_path.write_text(json.dumps(result))  # one file per hyperparameter
        pair
    bc_results.commit()  # flush volume writes so the next job sees them

    return result

@app.local_entrypoint()  # `modal run this_file.py` starts here on your
    laptop
def main():
    configs = [
```

```
        {"n_episodes": n, "max_depth": d}
        for n in [10, 50, 200]
        for d in [2, 5, 10]
    ]
    results = list(train_and_eval.starmap(configs))  # parallel remote
        calls; cap via max_containers
    best = max(results, key=lambda r: r["reward"])
    print("Best config:", best)
```

For heavier models, reuse the same volume pattern for checkpoints (serialized with `pickle`, `joblib`, or PyTorch). Pick a volume name that is unique to you if you share a Modal org with others.

6 GPU jobs

Request a GPU on the decorator for remote runs. Valid `gpu=` strings depend on Modal's current offerings and your account; check Modal's GPU documentation.

```
import torch

def do_work():
    device = "cuda" if torch.cuda.is_available() else "cpu"
    x = torch.randn(1024, 1024, device=device)
    return x.sum().item()

@app.function(image=gpu_image, gpu="YOUR_GPU_TYPE")  # example gpu="A100"
def remote_train():
    return do_work()
```

The `gpu="YOUR_GPU_TYPE"` line only affects remote runs, e.g. `run_with_gpu.remote()`.

`run_with_gpu.local()` runs in your local Python process and uses whatever CPU or GPU your laptop or workstation already has.

7 Interactive shell

Modal shell starts a shell with the same image (and optional volumes) as a given function—handy for debugging imports or inspecting mount paths:

```
modal shell path/to/your_script.py::your_function
```

8 Secrets

Do not print secret values. Define secrets in the CLI and attach them to functions:

```
modal secret create wandb WANDB_API_KEY=your_token_here
```

```
import os
import modal

wandb_secret = modal.Secret.from_name("wandb")
```

```
@app.function(image=my_image, secrets=[wandb_secret])
def log_run():
    _ = os.environ["WANDB_API_KEY"]
    ...
```

9 Debugging

Debugging a job that only fails inside a remote container is painful if you rely on `print` statements alone. Modal provides several first-class debugging tools—interactive breakpoints, shells into live containers, hot reloading, and debug logs—documented in full at Modal's developing and debugging guide. The rest of this section is a short tour of the pieces you will reach for most often in this class.

9.1 Interactive breakpoints

You can drop straight into the Python debugger (`pdb`) from a Modal function by putting Python's built-in `breakpoint()` call anywhere in your code and running the app with the `--interactive` / `-i` flag:

```
uv run modal run -i src/scripts/modal_run_train.py
# or: modal run -i src/scripts/modal_run_train.py
```

When execution reaches the `breakpoint()` call, Modal wires the remote container's stdin/stdout to your terminal and you get a live `pdb` prompt—you can inspect tensors, step through code, and evaluate Python expressions on the remote machine exactly as if it were local. For a richer REPL, `pip_install("ipython")` in your image and call `modal.interact()` followed by `IPython.embed()`.

Example: a breakpoint in HW2. Below is a breakpoint set inside the `eval()` method of the `Workspace` class in HW2's `train_off_policy.py`. This is a natural place to stop if you want to inspect the agent's action, the reward the environment just returned, and the current `time_step` before the reward is accumulated.

You set breakpoints in one of two ways:

- **In code:** add a line `breakpoint()` where you want to stop. Cursor will render a red dot in the gutter next to that line.
- **In the IDE:** click in the gutter to the left of a line number in Cursor/VS Code. This is equivalent to inserting a `breakpoint()` for local runs but only takes effect under a debugger—for remote Modal runs you should still add the literal `breakpoint()` call to the file you `modal run -i`.

9.2 Attaching to a running container

If a job is already running and looks stuck, you do not have to kill it. List active containers and attach an interactive shell:

```
modal container list
modal shell <container-id>
```

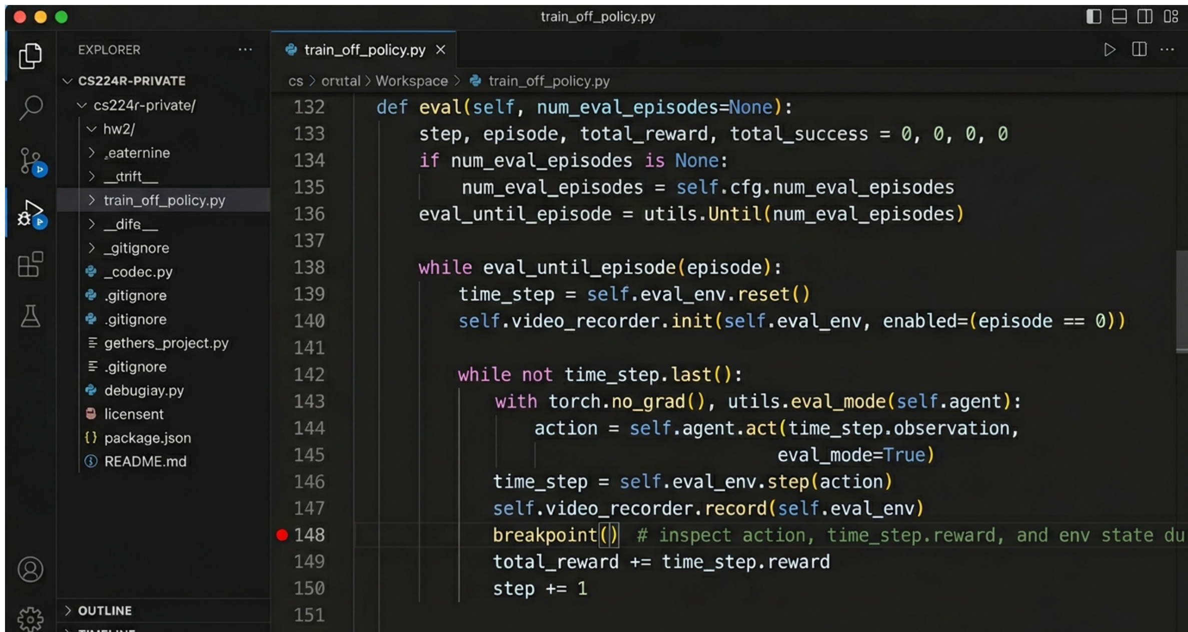

Figure 1: A breakpoint set on line 148 of `train_off_policy.py` inside the evaluation loop. The red dot in the gutter marks the breakpoint; when the file is run with `modal run -i`, execution will pause here and you will be dropped into `pdb` on the remote container.

The debug shell comes with `vim`, `nano`, `ps`, `strace`, `py-spy`, and `curl` preinstalled, so you can poke at the filesystem, sample stack traces of a hung Python process, or inspect environment variables without restarting the job. You can also run a single command without opening a full shell:

```
modal container exec <container-id> ls /root
```

9.3 Debugging the image itself

When the problem is "my imports fail" or "the file I expected is not on the container," the fastest fix is usually to open a shell with the same image your function uses:

```
modal shell path/to/your_script.py::your_function
```

This spins up a fresh container with your function's `Image`, `Secrets`, and `NetworkFileSystems` attached, so you can experimentally `pip install` things, check `/data` mounts, and iterate on build commands before baking them into the image.

9.4 Hot reloading with `modal serve`

For web endpoints or cron-scheduled functions, `modal serve path/to/file.py` watches your source file and pushes changes up to the remote App automatically—no restart required. This is useful when the assignment involves a small HTTP endpoint or when you want fast feedback on a function that runs on a schedule.

9.5 More verbose logs

If the issue is not obvious from the default output, raise the Modal client's log level:

```
MODAL_LOGLEVEL=DEBUG modal run path/to/your_script.py
MODAL_TRACEBACK=1 modal run path/to/your_script.py  # client-side stack
    traces
```

To turn on debug logs *inside* the remote container, pass `MODAL_LOGLEVEL` through a `modal.Secret`; see the debug logs section of Modal's guide for an example.

9.6 Live container profiling

If a container appears stuck, open your function on the Modal dashboard, go to the **Containers** tab, and use **Live Profiling** to see a real-time view of the Python call stack executing on the remote worker. This is the quickest way to distinguish "my code is in a tight loop" from "my code is blocked on I/O" without touching the container.

For the full set of flags, container profiling, IPython workflows, and dialog-handling details, see Modal's developing and debugging guide.

10 Dashboard and logs

Open modal.com, sign in, and browse your apps and run history (logs, failures, timing). Your workspace is individual; use whatever filters the UI provides to find recent jobs.

11 Summary

- Redeem course credits, run `modal setup`, then define an `App`, `Image`, and `@app.function` entrypoints.
- Use volumes for checkpoints and data; treat default worker filesystems as temporary.
- Use `map` / `starmap` for parallel jobs; `max_containers` helps control cost and concurrency.
- Attach `gpu=` for remote GPU; use secrets for API keys; use `modal shell` to debug the remote environment.
- This guide is about Modal mechanics—assignment-specific choices stay in your own code.