CloudAI Benchmark Framework 1.7.1

Table of contents

Overview

CloudAI benchmark framework aims to develop an industry standard benchmark focused on grading Data Center (DC) scale AI systems in the cloud. The primary motivation is to provide automated benchmarking on various systems.

This document contains the following chapters:

Getting Started

CloudAI officially targets Python 3.14 while remaining compatible with Python 3.10 and newer.

```
git clone git@github.com:NVIDIA/cloudai.git
cd cloudai
uv run cloudai --help
```

> **Note**
>
> For more instructions on how to set up access for `enroot`, see Installation Requirements.

pip-based Installation

See the preferred Python version in the `.python-version` file and make sure you have it installed (For installation, see Custom Python Version Installation). Python 3.10 and newer remain supported. Follow these steps:

```
git clone git@github.com:NVIDIA/cloudai.git
cd cloudai
python3.14 -m venv venv
source venv/bin/activate
pip install -e .
```

Custom Python Version Installation

If your system Python version is not supported, you can install a custom version using the uv tool:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
source $HOME/.local/bin/env
uv venv --seed  # picks the python version from .python-version
                # --seed installs pip and setuptools
source .venv/bin/activate
```

Key Concepts

CloudAI operates on three main schemas:

- **System Schema**: Describes the system, including the scheduler type, node list, and global environment variables
- **Test Schema**: An instance of a test template with custom arguments and environment variables
- **Test Scenario Schema**: A set of tests with dependencies and additional descriptions about the test scenario

These schemas enable CloudAI to be flexible and compatible with different systems and configurations.

CloudAI Modes Usage Examples

The following are the global options for the `cloudai` command:

- `--log-file <path>`: specify a file to log output; by default `debug.log` in the current directory is used. Contains log entries of level `DEBUG` and higher
- `--log-level <level>`: specify logging level for standard output; default is `INFO`

Available Modes

- run
- dry-run
- generate-report
- install
- uninstall
- list
- verify-configs

Usage Examples

run

This mode runs workloads. It automatically installs prerequisites if they are not met.

```
cloudai run\
    --system-config
conf/common/system/example_slurm_cluster.toml\
    --tests-dir conf/common/test\
    --test-scenario conf/common/test_scenario/sleep.toml
```

dry-run

This mode simulates running experiments without actually executing them. This is useful for verifying configurations and testing experiment setups.

```
cloudai dry-run\
    --system-config
conf/common/system/example_slurm_cluster.toml\
    --tests-dir conf/common/test\
    --test-scenario conf/common/test_scenario/sleep.toml
```

generate-report

This mode generates reports under the scenario directory. It automatically runs as part of the `run` mode after experiments are completed.

```
cloudai generate-report\
    --system-config
conf/common/system/example_slurm_cluster.toml\
    --tests-dir conf/common/test\
    --test-scenario conf/common/test_scenario/sleep.toml\
    --result-dir /path/to/result_directory
```

install

This mode installs test prerequisites. For more details, refer to the Installation Requirements guide. It automatically runs as part of the `run` mode if prerequisites are not met.

```
cloudai install\
    --system-config
conf/common/system/example_slurm_cluster.toml\
    --tests-dir conf/common/test\
    --test-scenario conf/common/test_scenario/sleep.toml
```

uninstall

This mode is the opposite of the install mode. This mode removes installed test prerequisites.

```
cloudai uninstall\
    --system-config
conf/common/system/example_slurm_cluster.toml\
    --tests-dir conf/common/test\
    --test-scenario conf/common/test_scenario/sleep.toml
```

list

This mode lists internal components available within CloudAI.

```
cloudai list <component_type>
```

verify-configs

This mode verifies the correctness of system, test, and test scenario configuration files.

```
# verify all at once
cloudai verify-configs conf
# verify a single file
cloudai verify-configs
conf/common/system/example_slurm_cluster.toml
# verify all scenarios using specific folder with Test TOMLs
cloudai verify-configs --tests-dir conf/release/spcx/l40s/test
conf/release/spcx/l40s/test_scenario
```

Documentation Update History

Version	Date	Description
v1.5.0	April xx, 2026	Initial release of this documentation version.

Tutorial

This chapter outlines a tutorial on how to utilize the CloudAI framework. Please follow the steps in the same sequence to ensure successful execution:

Creating a Docker Image

To create a Docker image, follow these steps:

1. **Set Up the GitLab Repository:** Start by setting up a repository on GitLab to host your docker image. For this example, use `gitlab-url.com/cloudai/nccl-test`.
2. **Write the Dockerfile:**

   ```
   FROM nvcr.io/nvidia/pytorch:24.02-py3
   ```

3. **Build and Push the Docker Image:** Build the docker image with the Dockerfile and upload it to the designated repository:

```
docker build -t gitlab-url.com/cloudai/nccl-test .
docker push gitlab-url.com/cloudai/nccl-test
```

4. **Verify the Docker Image:** Test the docker image by running it with `srun` to verify that the docker image runs correctly:

```
srun \
   --mpi=pmix \
   --container-image=gitlab-url.com/cloudai/nccl-test \
   all_reduce_perf_mpi \
   --nthreads 1 \
   --ngpus 1 \
   --minbytes 128 \
   --maxbytes 16G \
   --stepbytes 1M \
   --op sum \
   --datatype float \
   --root 0 \
   --iters 100 \
   --warmup_iters 50 \
   --agg_iters 1 \
   --average 1 \
   --parallel_init 0 \
   --check 1 \
   --blocking 0 \
   --cudagraph 0 \
   --stepfactor 2
```

Preparing Configuration Files

CloudAI is fully configurable via a set of TOML configuration files. You can find examples of these files under `conf/common`. In this guide, we will use the following configuration files:

1. `CONFIGS_DIR/system.toml` - Describes the system configuration.
2. `CONFIGS_DIR/tests/nccl_test.toml` - Describes the test to run.
3. `CONFIGS_DIR/scenario.toml` - Describes the test scenario configuration.

Testing Definition

Test definition is a Pydantic model that describes the arguments of a test. Such models should be inherited from the `TestDefinition` class:

```
class MyTestCmdArgs(CmdArgs):
     an_arg: str | list[str]
     docker_image_url: str = "nvcr.io/nvidia/pytorch:24.02-py3"
class MyTestDefinition(TestDefinition):
    cmd_args: MyTestCmdArgs
```

Notice that `cmd_args.docker_image_url` uses `nvcr.io/nvidia/pytorch:24.02-py3`, but you can use the Docker image from Step 1.

`an_arg` has a mixed type of `str | list[str]`, so in a TOML config it can be defined as either:

```
an_arg = "a single string"
```

Or

```
an_arg = ["list", "of", "strings"]
```

When a list is used, CloudAI will automatically generate multiple test cases for each value in the list.

A custom test definition should be registered to handle relevant Test Configs. For this, `Registry()` object is used:

```
Registry().add_test_definition("MyTest", MyTestDefinition)
Registry().add_test_template("MyTest", MyTest)
```

Relevant Test Configurations should specify `test_template_name = MyTest` to use the custom test definition.

System Configuration

System configuration describes how the system configuration works. You can find more examples of system configuration under `conf/common/system/`. The example below is for demonstration purposes. The following is the `CONFIGS_DIR/system.toml` file:

```
name = "my-cluster"
scheduler = "slurm"
install_path = "./install"
output_path = "./results"
cache_docker_images_locally = true
default_partition = "<YOUR PARTITION NAME>"
mpi = "pmix"
gpus_per_node = 8
ntasks_per_node = 8
[[partitions]]
name = "partition_1"
```

Replace `<YOUR PARTITION NAME>` with the name of the partition you want to use. You can find the partition name by running `sinfo` on the cluster.

Testing Configuration

Test configuration describes a particular test configuration to be run. It is based on test definition and will be used in a test scenario. Below is the

`CONFIGS_DIR/tests/nccl_test.toml` file, definition is based on the built-in `NcclTest` definition:

```
name = "nccl_test_all_reduce_single_node"
description = "all_reduce"
test_template_name = "NcclTest"
[cmd_args]
subtest_name = "all_reduce_perf_mpi"
ngpus = 1
minbytes = "8M"
maxbytes = "16G"
iters = 5
warmup_iters = 3
stepfactor = 2
```

You can find more examples under `conf/common/test`. In a test schema file, you can adjust arguments as shown above. In the `cmd_args` section, you can provide different values other than the default values for each argument. In `extra_cmd_args`, you can provide additional arguments that will be appended after the NCCL test command. You can specify additional environment variables in the `extra_env_vars` section.

Running Experiments

Test Scenario uses test description from Step 5. Below is the `CONFIGS_DIR/scenario.toml` file:

```
name = "nccl-test"
[[Tests]]
id = "allreduce.1"
num_nodes = 1
test_name = "nccl_test_all_reduce_single_node"
time_limit = "00:20:00"
[[Tests]]
id = "allreduce.2"
num_nodes = 1
test_name = "nccl_test_all_reduce_single_node"
time_limit = "00:20:00"
  [[Tests.dependencies]]
  type = "start_post_comp"
  id = "allreduce.1"
```

Notes on the test scenario:

1. `id` is a mandatory field and must be unique for each test.

2. The `test_name` specifies the test definition from one of the Test TOML files. Node lists and time limits are optional.

3. If needed, `nodes` should be described as a list of node names as shown in a Slurm system. Alternatively, if groups are defined in the system schema, you can ask CloudAI to allocate a specific number of nodes from a specified partition and group. For example, `nodes = ['PARTITION:GROUP:16']` allocates 16 nodes from group `GROUP` and partition `PARTITION`.

4. There are three types of dependencies: `start_post_comp`, `start_post_init` and `end_post_comp`.

    - `start_post_comp` means that the current test should be started after a specific delay of the completion of the depending test.

    - `start_post_init` means that the current test should start after the start of the depending test.

- `end_post_comp` means that the current test should be completed after the completion of the depending test.

All dependencies are described as a depending test name. The name should be taken from the test name as set in the test scenario.

To generate NCCL test commands without actual execution, use the `dry-run` mode. You can review `debug.log` (or other file specified with `--log-file`) to see the generated commands from CloudAI. Please note that group node allocations are not currently supported in the `dry-run` mode.

```
cloudai dry-run \
    --test-scenario CONFIGS_DIR/scenario.toml \
    --system-config CONFIGS_DIR/system.toml \
    --tests-dir CONFIGS_DIR/tests/
```

You can run NCCL test experiments with the following command. Whenever you run CloudAI in the `run` mode, a new directory will be created under the results directory with the timestamp. In the directory, you can find the results from the test scenario including stdout and stderr. Once completed successfully, you can find generated reports under the directories as well.

```
cloudai run \
    --test-scenario CONFIGS_DIR/scenario.toml \
    --system-config CONFIGS_DIR/system.toml \
    --tests-dir CONFIGS_DIR/tests/
```

Generating Reports

Once the test scenario is completed, it is possible to generate reports using the following command:

```
cloudai generate-report \
   --test-scenario CONFIGS_DIR/scenario.toml \
   --system-config CONFIGS_DIR/system.toml \
   --tests-dir CONFIGS_DIR/tests/ \
   --result-dir results/2024-06-18_17-40-13/
```

`--result-dir` accepts one scenario run result directory.

Test in Scenario

It is possible to override some args or even fully define a workload inside a scenario file:

```
name = "nccl-test"
[[Tests]]
id = "allreduce.in.scenario"
num_nodes = 1
time_limit = "00:20:00"
name = "nccl_test_all_reduce_single_node"
description = "all_reduce"
test_template_name = "NcclTest"
  [Tests.cmd_args]
  subtest_name = "all_reduce_perf_mpi"
  ngpus = 1
  minbytes = "8M"
  maxbytes = "16G"
  iters = 5
  warmup_iters = 3
  stepfactor = 2
[[Tests]]
id = "allreduce.override"
num_nodes = 1
test_name = "nccl_test_all_reduce_single_node"
time_limit = "00:20:00"
  [Tests.cmd_args]
  stepfactor = 4
```

`allreduce.in.scenario` fully defines a workload; in this case `test_name` must not be set, while `name`, `description` and `test_template_name` must be set.

`allreduce.override` overrides only `stepfactor` arg from the test defined in the tests directory.

If a scenario contains only fully defined tests, `--tests-dir` arg is not required.

User Guide

The purpose of this chapter is to help users utilize and understand the different components of the CloudAI framework. The chapter covers topics such as adding new tests and downloading datasets for running NeMo-launcher.

System Schema

In this section, we introduce the concept of the system schema, explain the meaning of each field, and describe how the fields should be used. The system schema is a TOML file that allows users to define a system's configuration.

```
name = "example-cluster"
scheduler = "slurm"
install_path = "./install"
output_path = "./results"
default_partition = "partition_1"
mpi = "pmix"
gpus_per_node = 8
ntasks_per_node = 8
cache_docker_images_locally = true
[[partitions]]
name = "partition_1"
  [[partitions.groups]]
  name = "group_a"
  nodes = ["node-[001-025]"]
  [[partitions.groups]]
  name = "group_b"
  nodes = ["node-[026-050]"]
[[partitions]]
name = "partition_2"
[global_env_vars]
# NCCL Specific Configurations
NCCL_IB_GID_INDEX = "3"
NCCL_IB_TIMEOUT = "20"
NCCL_IB_QPS_PER_CONNECTION = "4"
# Device Visibility Configuration
MELLANOX_VISIBLE_DEVICES = "0,3,4,5,6,9,10,11"
CUDA_VISIBLE_DEVICES = "0,1,2,3,4,5,6,7"
```

Field Descriptions

Field Descriptions

Field	Description

name	Specifies the name of the system. Users can choose any name that is convenient for them.
scheduler	Indicates the type of system. It should be one of the supported types, currently `slurm` or `standalone`. `slurm` refers to a system with the Slurm scheduler, while `standalone` refers to a single-node system without any slave nodes. Other values are possible depending on the available schedulers supported by CloudAI.
install_path	Specifies the path where test prerequisites are installed. Docker images are downloaded to this path if the user chooses to cache Docker images.
output_path	Defines the default path where outputs are stored. Whenever a user runs a test scenario, a new subdirectory will be created under this path.
default_partition	Specifies the default partition where jobs are scheduled.
partitions	Describes the available partitions and nodes within those partitions.

[optional] groups	Within the same partition, users can define groups of nodes. The group concept can be used to allocate nodes from specific groups in a test scenario schema. For instance, this feature is useful for specifying topology awareness. Groups represent logical partitioning of nodes and users are responsible for ensuring no overlap across groups.
mpi	Indicates the Process Management Interface (PMI) implementation to be used for inter-process communication.
gpus_per_node and **ntasks_per_node**	These are Slurm arguments passed to the `sbatch` script and `srun`.

cache_docker_images_locally	Specifies whether CloudAI should cache remote Docker images locally during installation. If set to `true`, CloudAI will cache the Docker images, enabling local access without needing to download them each time a test is run. This approach saves network bandwidth but requires more disk capacity. If set to `false`, CloudAI will allow Slurm to download the Docker images as needed when they are not cached locally by Slurm.
global_env_vars	Lists all global environment variables that will be applied globally whenever tests are run.

RunAI Scheduler

When using RunAI as the scheduler, additional fields must be specified in the system schema TOML file. The following is a list of required fields and how to set them:

```
name = "runai-cluster"
scheduler = "runai"
install_path = "./install"
output_path = "./results"
base_url = "http://runai.example.com"       # The URL of your
RunAI system, typically the same as used for the web interface.
user_email = "your_email"              # The email address used
to log into the RunAI system.
app_id = "your_app_id"                      # Obtained by
creating an application in the RunAI web interface.
app_secret = "your_app_secret"              # Obtained together
with the app_id.
project_id = "your_project_id"              # Project ID assigned
or created in the RunAI system (usually an integer).
cluster_id = "your_cluster_id"              # Cluster ID in UUID
format (e.g., a69928cc-ccaa-48be-bda9-482440f4d855).
```

- After logging into the RunAI web interface, navigate to Access → Applications and create a new application to obtain `app_id` and `app_secret`
- Use your assigned project and cluster IDs. Contact your administrator if they are not available
- All other fields follow the same semantics as in the Slurm system schema (e.g., `install_path`, `output_path`)

Test Scenario Schema

A test scenario is a set of tests with specific dependencies between them. A test scenario is described in a TOML schema file. The following is an example of a test scenario file:

```
name = "nccl-test"
[[Tests]]
id = "Tests.1"
test_name = "nccl_test_all_reduce"
num_nodes = "2"
time_limit = "00:20:00"
[[Tests]]
id = "Tests.2"
test_name = "nccl_test_all_gather"
num_nodes = "2"
time_limit = "00:20:00"
  [[Tests.dependencies]]
  type = "start_post_comp"
  id = "Tests.1"
[[Tests]]
id = "Tests.3"
test_name = "nccl_test_reduce_scatter"
num_nodes = "2"
time_limit = "00:20:00"
  [[Tests.dependencies]]
  type = "start_post_comp"
  id = "Tests.2"
```

The `name` field is the test scenario name, which can be any unique identifier for the scenario. Each test has a section name, following the convention `Tests.1`, `Tests.2`, etc., with an increasing index. The `name` of a test should be specified in this section and must correspond to an entry in the test schema. If a test in a test scenario is not present in the test schema, CloudAI will not be able to identify it.

There are two ways to specify nodes:

- Using the `num_nodes` field as shown in the example
- Specifying nodes explicitly like `nodes = ["node-001", "node-002"]`

Note: When an explicit node list is provided (e.g., `nodes = ["node-001", "node-002"]`), CloudAI lets Slurm apply the arbitrary distribution policy for task placement.

Alternatively, you can utilize the groups feature in the system schema to specify nodes like `nodes = ['PARTITION_NAME:GROUP_NAME:NUM_NODES']`, which allocates `num_nodes` from the group name in the specified partition. You can also use `nodes = ['PARTITION_NAME:GROUP_NAME:max_avail']`, which allocates all the available nodes from the group name in the specified partition.

You can optionally specify a time limit in the Slurm format. Tests can have dependencies. If no dependencies are specified, all tests will run in parallel.

CloudAI supports three types of dependencies:

- `start_post_init`
- `start_post_comp`
- `end_post_comp`

Dependencies of a test can be described as a subsection of the test. It requires other tests' `id` and dependency `type`.

- `start_post_init` means the test starts after the prior test begins, with a specified delay.
- `start_post_comp` means the test starts after the prior test completes.
- `end_post_comp` means the test ends when the prior test completes.

Agents Configuration

For DSE workloads, you can pass agent configuration via `agent_config` in the scenario.

`BaseAgentConfig` includes:

- `random_seed`: controls deterministic random behavior in agents.
- `start_action`: controls the very first action strategy (`"random"` or `"first"`).

- `rewards` (optional): nested table mapped to `RewardOverrides`. When present, `CloudAIGymEnv` uses it for constraint failures and for substituting failed-metric values in observations.
    - `constraint_failure`: reward returned when `TestDefinition.constraint_check` fails on a step. If omitted or unset, the environment uses `-1.0`.
    - `metric_failure`: value written into the observation vector when a metric is missing or is the canonical error sentinel `METRIC_ERROR` (a distinct object from numeric metrics, so a valid `-1.0` result is not mistaken for an error). If omitted or unset, observations use `-1.0` for that slot.

Example:

```
[[Tests]]
id = "Tests.1"
test_name = "nccl_test_all_reduce"
agent = "grid_search"
agent_steps = 10
agent_metrics = ["default"]
agent_reward_function = "inverse"
  [Tests.agent_config]
  random_seed = 123
  start_action = "first"
    [Tests.agent_config.rewards]
    constraint_failure = -5.0
    metric_failure = 0.0
```

When an agent honors `start_action = "first"`, it should start from `CloudAIGymEnv.first_sweep` (the sweep is built from first values of each sweep parameter). `start_action = "random"` means starting from a random action, typically seeded by `random_seed`.

Custom agents may extend the `BaseAgentConfig` and offer more parameters to configure.

DSE parameter exclusions

CloudAI builds the DSE parameter space implicitly from list-valued fields under `cmd_args`, list-valued `extra_env_vars`, and list-valued `num_nodes`. Most lists mean "try each value", but some workload settings are real list-valued configuration, such as worker port lists or ordered benchmark phases.

Use `dse_excluded_args` when a list under `cmd_args` should stay intact instead of becoming a sweep dimension. Entries must be dot-separated paths that start with `cmd_args.` and may point to either a single field or a parent object. Matching is prefix-based, so excluding `cmd_args.foo` also excludes nested list-valued fields such as `cmd_args.foo.bar` from DSE parameter discovery.

```
[[Tests]]
id = "Tests.1"
test_name = "my_test"
dse_excluded_args = ["cmd_args.lmcache.lmcache_worker_ports"]
  [Tests.cmd_args.lmcache]
  chunk_size = [256, 512]
  lmcache_worker_ports = [8788, 8789, 8790, 8791]
```

In this example, `cmd_args.lmcache.chunk_size` is still swept, while `cmd_args.lmcache.lmcache_worker_ports` is passed through as one list value. The exclusion does not remove or mutate the field; it only prevents CloudAI from adding that path to the DSE parameter space.

`dse_excluded_args` currently applies only to `cmd_args` paths. It does not exclude list-valued `extra_env_vars` or `num_nodes`; those lists are still interpreted as sweep dimensions. To exclude many nested list fields at once, exclude their common parent path. Common examples are `cmd_args.aiperf_phases` and `cmd_args.lmcache.lmcache_worker_ports`.

Metric errors and report strategies

Report generation strategies signal a failed or missing metric by returning the singleton `METRIC_ERROR` from `cloudai.core` (type `MetricErrorSentinel`; `float(METRIC_ERROR)` is `-1.0` for numeric use). The declared return type of `ReportGenerationStrategy.get_metric` is `MetricValue` (`float | MetricErrorSentinel`).

When branching on success vs failure, use **identity** (`value is METRIC_ERROR`), not equality to `-1.0`, so a legitimate metric of `-1.0` is not treated as an error. The gym observation path uses that identity check before applying `agent_config.rewards.metric_failure`.

Configuring HTTP Data Repository

The HTTP Data Repository is currently supported for Slurm systems only. To enable access, you must update your system schema file and create a credential file in your CloudAI project's root directory.

The following steps are required to configure the HTTP Data Repository:

- Step 1: Update the System Schema File
- Step 2: Create the Credential File
- Step 3: Usage

Updating the System Schema File

Add the following section to your system schema TOML file (e.g., `system_schema.toml`):

```
[data_repository]
endpoint = "https://my-data-endpoint.com"
```

Replace the endpoint with your actual data repository URL.

Creating the Credential File

In the root of your CloudAI project (i.e., the current working directory), create a file named `.cloudai.toml` with the following content:

```
[data_repository]
token = "<your-api-token-here>"
```

Replace `<your-api-token-here>` with your actual token.

Usage

Both the endpoint and token must be valid for the HTTP Data Repository to function correctly. If either is missing or incorrect, data will not be posted.

Using Test Hooks in CloudAI

A test hook in CloudAI is a specialized test that runs either before or after each main test in a scenario, providing flexibility to prepare the environment or clean up resources. Hooks are defined as pre-test or post-test and referenced in the test scenario's TOML file using `pre_test` and `post_test` fields.

```
name = "nccl-test"
pre_test = "nccl_test_pre"
post_test = "nccl_test_post"
[[Tests]]
id = "Tests.1"
test_name = "nccl_test_all_reduce"
time_limit = "00:20:00"
```

CloudAI organizes hooks in a dedicated directory structure:

- Hook directory: All hook configurations reside in `conf/hook/`
- Hook test scenarios: Place pre-test hook and post-test hook scenario files in `conf/hook/`
- Hook tests: Place individual tests referenced in hooks within `conf/hook/test/`

In the execution flow, pre-test hooks run before the main test, which only executes if the pre-test completes successfully. Post-test hooks follow the main test, provided the prior steps succeed. If a pre-test hook fails, the main test and its post-test hook are skipped.

```
name = "nccl_test_pre"
[[Tests]]
id = "Tests.1"
test_name = "nccl_test_all_reduce"
time_limit = "00:20:00"
```

Downloading DeepSeek Weights

To run DeepSeek R1 tests in CloudAI, you must download the model weights in advance. These weights are distributed via the NVIDIA NGC Registry and must be manually downloaded using the NGC CLI.

The following steps are required to download the DeepSeek weights:

- Step 1: Installing NGC CLI
- Step 2: Configuring NGC CLI
- Step 3: Downloading the Weights
- Step 4: Verifying the Download

Installing NGC CLI

Download and install the NGC CLI using the following commands:

```
wget --content-disposition
https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-
apps/ngc_cli/versions/3.64.2/files/ngccli_linux.zip -O
ngccli_linux.zip
unzip ngccli_linux.zip
chmod u+x ngc-cli/ngc
echo "export PATH=\"$PATH:$(pwd)/ngc-cli\"" >> ~/.bash_profile &&
source ~/.bash_profile
```

This will make the `ngc` command available in your terminal.

Configuring NGC CLI

Authenticate your CLI with your NGC API key by running:

```
ngc config set
```

When prompted, paste your API key, which you can obtain from https://org.ngc.nvidia.com/setup.

Downloading the Weights

Navigate to the directory where you want the DeepSeek model weights to be stored, then run:

```
ngc registry model download-version nim/deepseek-ai/deepseek-r1-
instruct:hf-5dde110-nim-fp8 --dest .
```

This command will create a folder named:

```
deepseek-r1-instruct_vhf-5dde110-nim-fp8/
```

inside your current directory.

Verifying the Download

Make sure the full model has been downloaded by checking the folder size:

```
du -sh deepseek-r1-instruct_vhf-5dde110-nim-fp8
```

The expected size is approximately 642 GB. If it's significantly smaller, remove the folder and re-run the download.

Slurm

The following are subtopics related to Slurm:

Single sbatch vs Per-case sbatch

CloudAI supports two modes for submitting Slurm jobs:

1. (default) Per-case sbatch mode: each case is submitted as a separate sbatch job, allowing flexible scheduling and dependency management.
    - Suits best when cases can run in parallel or there are no dependencies between cases.
2. Single sbatch mode: all cases are submitted together in a single sbatch job and share the same nodes. Each next case starts after the previous one completes. To enable it one needs to pass `--single-sbatch` flag to `cloudai run` command (works only for Slurm systems).
    - Suits best for jobs when cases need to run on the same nodes for performance reasons.
    - There is no support for dependency management between cases yet; all jobs run one after another.

Assuming two cases in a scenario like this:

```
[[Tests]]
id = "nccl.all_reduce"
test_name = "nccl-all_reduce"
num_nodes = 2
time_limit = "00:20:00"
[[Tests]]
id = "nccl.all_gather"
test_name = "nccl-all_gather"
num_nodes = 2
time_limit = "00:20:00"
```

Regular output directory structure (some files are omitted for clarity):

```
$ tree results/scenario
results/scenario
    nccl.all_gather
        0
            cloudai_sbatch_script.sh
            env_vars.sh
            metadata/
            stderr.txt
            stdout.txt
            test-run.toml
    nccl.all_reduce
        0
            cloudai_sbatch_script.sh
            env_vars.sh
            metadata/
            stderr.txt
            stdout.txt
            test-run.toml
```

Single sbatch mode output directory structure:

```
$ tree results/scenario
results/scenario
    cloudai_sbatch_script.sh
    common.err
    common.out
    metadata/
    nccl.all_gather
        0
            env_vars.sh
            stderr.txt
            stdout.txt
            test-run.toml
    nccl.all_reduce
        0
            env_vars.sh
            stderr.txt
            stdout.txt
            test-run.toml
    slurm-job.toml
```

Most of the files are the same: output files, env vars script, test run metadata. The difference for single sbatch mode is that `slurm-job.toml` is created for the entire job as well as `cloudai_sbatch_script.sh`. Extra `common.err`/`common.out` files are created for general sbatch outputs.

Extra srun and sbatch Arguments

CloudAI forms sbatch script and srun commands following internal rules. Users can affect the generation by setting special arguments in the system TOML file.

For example (in a system TOML file):

```
extra_sbatch_args = [
  "--section=4",
  "--other-arg val"
]
```

It will result in sbatch file content like this:

```
... # CloudAI set sbatch arguments
#SBATCH --section=4
#SBATCH --other-arg val
...
```

Another example (in a system TOML file):

```
extra_srun_args = "--arg=val --other-arg=other-val"
```

Will result in srun command inside sbatch script like this:

```
srun ... --arg=val --other-arg=other-val ...
```

Container Mounts

CloudAI runs all slurm jobs using containers. To simplify file system related tasks, CloudAI mounts the following directories into the container:

1. Test output directory (`<output_path>/<scenario_name_with_timestamp>/<test_name>/<iterati` , like `results/scenario_2024-06-18_17-40-13/Tests.1/0`) is mounted as `/cloudai_run_results` .

2. Test specific mounts can be specified in TOML files:

```
extra_container_mounts = [
  "/path/to/mount1:/path/in/container1",
  "/path/to/mount2:/path/in/container2"
]
[cmd_args]
...
```

These mounts are not verified for validity and do not override default mounts.

3. Test specific mounts can be mounted in-code.

Head Node without Shared Storage Available on Compute Nodes

When compute nodes do not share the file system with head node, `--enable-cache-without-check` for `run` and `dry-run` skips the real check for cache existence, but still builds all paths correctly. The flow is like this:

1. *[on the head node]* run `cloudai install`.
2. *[on the head node]* copy cache to compute nodes.
3. Modify `system.toml` to set compute nodes' installation root.
4. Run `cloudai run --enable-cache-without-check ...`.

Dev Details

`SlurmCommandGenStrategy` defines abstract method `_container_mounts(tr: TestRun)` that must be implemented by every subclass. This method is used in `SlurmCommandGenStrategy.container_mounts(tr: TestRun)` (defined as `@final`) where mounts like `/cloudai_run_results` (default mount), `TestDefinition.extra_container_mounts` (from Test TOML) and test specific mounts (defined in-code) are added.

Nsys Tracing

Users can enable Nsys tracing for any workload when running via Slurm. Note that `nsys` should be available on the compute nodes; CloudAI does not manage it.

The configuration fields are:

```
enable: bool = True
nsys_binary: str = "nsys"
task: str = "profile"
output: Optional[str] = None
sample: Optional[str] = None
trace: Optional[str] = None
force_overwrite: Optional[bool] = None
capture_range: Optional[str] = None
capture_range_end: Optional[str] = None
cuda_graph_trace: Optional[str] = None
gpu_metrics_devices: Optional[str] = None
extra_args: list[str] = []
```

Fields with `None` value are not passed to `nsys` command.

Reporting

This chapter describes the reporting system in CloudAI. In this chapter, we will cover the following topics:

Overview

CloudAI has two reporting levels:

- per-test (per each case in a test scenario)
- per-scenario (per each test scenario)

All reports are generated after the test scenario is completed as part of the main CloudAI process. For Slurm, this means that the login node is used to generate reports.

Per-test reports are linked to a particular workload type (e.g. `NcclTest`). All per-test reports are implemented as part of the `per_test` scenario report and can be enabled or disabled via a single configuration option; see Enabling, Disabling and Configuring Reports.

To list all available reports, users can use `cloudai list-reports`. Use verbose output to also print report configurations.

General Flow

- All reports should be registered via `Registry()` (`.add_report()` or `.add_scenario_report()`)

- Scenario reports are configurable via system config (Slurm-only for now) and scenario config
- Configuration in a scenario config has the highest priority. Next, system config is checked. Then it defaults to report config from the registry
- Finally, the report is generated (or not) according to this final config

Enabling, Disabling and Configuring Reports

> **Note**
>
> Only scenario-level reports can be configured.

Enabling or disabling a report needs to be done in the system configuration:

```
[reports]
per_test = { enable = false }
status = { enable = true }
```

Reporting Registration

Report registration is done via `Registry` class:

```
Registry().add_scenario_report("per_test", PerTestReporter,
ReportConfig(enable=True))
```

Reporting Configuration Implementation

Each report can define its own configuration, which is constructed and passed as an argument to `Registry.add_scenario_report`. The `reports` field is parsed during

TOML reading and the respective Pydantic model is created for it.

For example, a custom report configuration can be defined as follows:

```
class CustomReportConfig(ReportConfig):
    greeting: str
```

```
Registry().add_scenario_report("custom", CustomReport,
CustomReportConfig(greeting="default value"))
```

And it can be used in a test scenario as follows:

```
[reports]
custom = { enable = true, greeting = "Hello, world!" }
```

Systems

This chapter lists all the systems supported by CloudAI. The attributes shown for each system can be set in TOML configuration files.

System	Scheduler Value
Slurm	`slurm`
Kubernetes	`kubernetes`
RunAI	`runai`
LSF	`lsf`
Standalone	`standalone`

Slurm

pydantic model cloudai.systems.slurm.slurm_system.SlurmSystem[source]

Represents a Slurm system.

field default_partition*: str [Required]*

field partitions*: List[SlurmPartition] [Required]*

field account*: str | None = None*

field distribution*: str | None = None*

field mpi*: str = 'pmix'*

field gpus_per_node*: int | None = None*

field ntasks_per_node*: int | None = None*

field cache_docker_images_locally*: bool = False*

field scheduler*: str = 'slurm'*

field monitor_interval*: int = 60*

field extra_srun_args*: str | None = None*

field extra_sbatch_args*: list[str] [Optional]*

field container_mount_home*: bool = False*

field data_repository*: DataRepositoryConfig | None = None*

field reports*: dict[str, ReportConfig] | None = None*

field name*: str [Required]*

field install_path*: Path [Required]*

field output_path*: Path [Required]*

field hf_home_path*: Path [Optional]*

field global_env_vars*: dict[str, Any] [Optional]*

pydantic model cloudai.systems.slurm.slurm_system.SlurmPartition[source]

Represents a partition within a Slurm system.

field name*: str [Required]*

field groups*: List[SlurmGroup] = []*

pydantic model cloudai.systems.slurm.slurm_system.SlurmGroup[source]

Represents a group of nodes within a partition.

field name*: str [Required]*

field nodes*: List[str] [Required]*

pydantic model cloudai.systems.slurm.slurm_system.DataRepositoryConfig[source]

Configuration for a data repository.

field endpoint*: str [Required]*

field verify_certs*: bool = True*

Kubernetes

pydantic model
cloudai.systems.kubernetes.kubernetes_system.KubernetesSystem[source]

Represents a Kubernetes system.

field kube_config_path*: Path [Required]*

field default_namespace*: str [Required]*

field scheduler*: str = 'kubernetes'*

field monitor_interval*: int = 1*

field gpus_per_node*: int = 1*

field use_host_network*: bool | None = None*

field name*: str [Required]*

field install_path*: Path [Required]*

field output_path*: Path [Required]*

field hf_home_path*: Path [Optional]*

field global_env_vars*: dict[str, Any] [Optional]*

RunAI

pydantic model cloudai.systems.runai.runai_system.RunAISystem[source]

RunAISystem integrates with the RunAI platform to manage and monitor jobs and nodes.

field scheduler*: str = 'runai'*

field monitor_interval*: int = 60*

field base_url*: str [Required]*

field user_email*: str [Required]*

field app_id*: str [Required]*

field app_secret*: str [Required]*

field project_id*: str [Required]*

field cluster_id*: str [Required]*

field name*: str [Required]*

field install_path*: Path [Required]*

field output_path*: Path [Required]*

field hf_home_path*: Path [Optional]*

field global_env_vars*: dict[str, Any] [Optional]*

pydantic model cloudai.systems.runai.runai_node.RunAINode[source]

Represent a node in the RunAI cluster.

field status*: NodeStatus = NodeStatus.UNKNOWN*

field conditions*: List[Dict[str, Any]] [Optional]*

field taints*: List[Dict[str, Any]] [Optional]*

field node_pool*: str = '' (alias 'nodePool')*

field created_at*: str = '' (alias 'createdAt')*

field gpu_type*: str | None = None (alias 'gpuType')*

field gpu_count*: int | None = None (alias 'gpuCount')*

field nvlink_domain_uid*: str | None = None (alias 'nvLinkDomainUid')*

field nvlink_clique_id*: str | None = None (alias 'nvLinkCliqueId')*

field name*: str = ''*

field id*: str = ''*

field cluster_uuid*: str = '' (alias 'clusterUuid')*

field updated_at*: str = '' (alias 'updatedAt')*

LSF

pydantic model cloudai.systems.lsf.lsf_system.LSFSystem[source]

Represents an LSF system.

field queues*: List[LSFQueue] [Optional]*

A list of queues in the LSF system, filled in automatically

field account*: str | None = None*

field scheduler*: str = 'lsf'*

field project_name*: str | None = None*

field default_queue*: str | None = None*

field monitor_interval*: int = 60*

field app*: str | None = None*

field os_version*: str | None = None*

field name*: str [Required]*

field install_path*: Path [Required]*

field output_path*: Path [Required]*

field hf_home_path*: Path [Optional]*

field global_env_vars*: dict[str, Any] [Optional]*

pydantic model cloudai.systems.lsf.lsf_system.LSFQueue[source]

Represents a queue within the LSF system.

field name*: str [Required]*

field groups*: List[LSFGroup] = []*

pydantic model cloudai.systems.lsf.lsf_system.LSFGroup[source]

Represents a group of nodes within a queue.

field name*: str [Required]*

field nodes*: List[str] [Required]*

Standalone

pydantic model
cloudai.systems.standalone.standalone_system.StandaloneSystem[source]

Class representing a Standalone system.

This class is used for systems that execute commands directly without a job scheduler.

field scheduler*: str = 'standalone'*

field monitor_interval*: int = 1*

field name*: str [Required]*

field install_path*: Path [Required]*

field output_path*: Path [Required]*

field hf_home_path*: Path [Optional]*

field global_env_vars*: dict[str, Any] [Optional]*

Workloads

This chapter contains automatically generated documentation for all CloudAI workloads. Each workload provides specific functionality for running different types of tests and benchmarks.

Available Workloads

Test	Slurm	Kubernetes	RunAI	Standalone
AIConfigurator	□	□	□	□
AI Dynamo	□	□	□	□
Bash Command	□	□	□	□
Chakra Replay	□	□	□	□
DDLB	□	□	□	□
DeepEP Benchmark	□	□	□	□
DynamoMocker	□	□	□	□
fio	□	□	□	□
JaxToolbox workloads (DEPRECATED)	□	□	□	□
MegatronRun	□	□	□	□
MegatronBridge	□	□	□	□
NCCL	□	□	□	□
NeMo v1.0 aka NemoLauncher (DEPRECATED)	□	□	□	□
Nemo Run	□	□	□	□
NIXL Bench	□	□	□	□
NIXL EP	□	□	□	□
NIXL KVBench	□	□	□	□
NIXL CTPerf	□	□	□	□

Sleep	□	□	□	□
SGLang	□	□	□	□
Slurm Container	□	□	□	□
Triton Inference	□	□	□	□
UCC	□	□	□	□
vLLM	□	□	□	□

Adding New Workloads

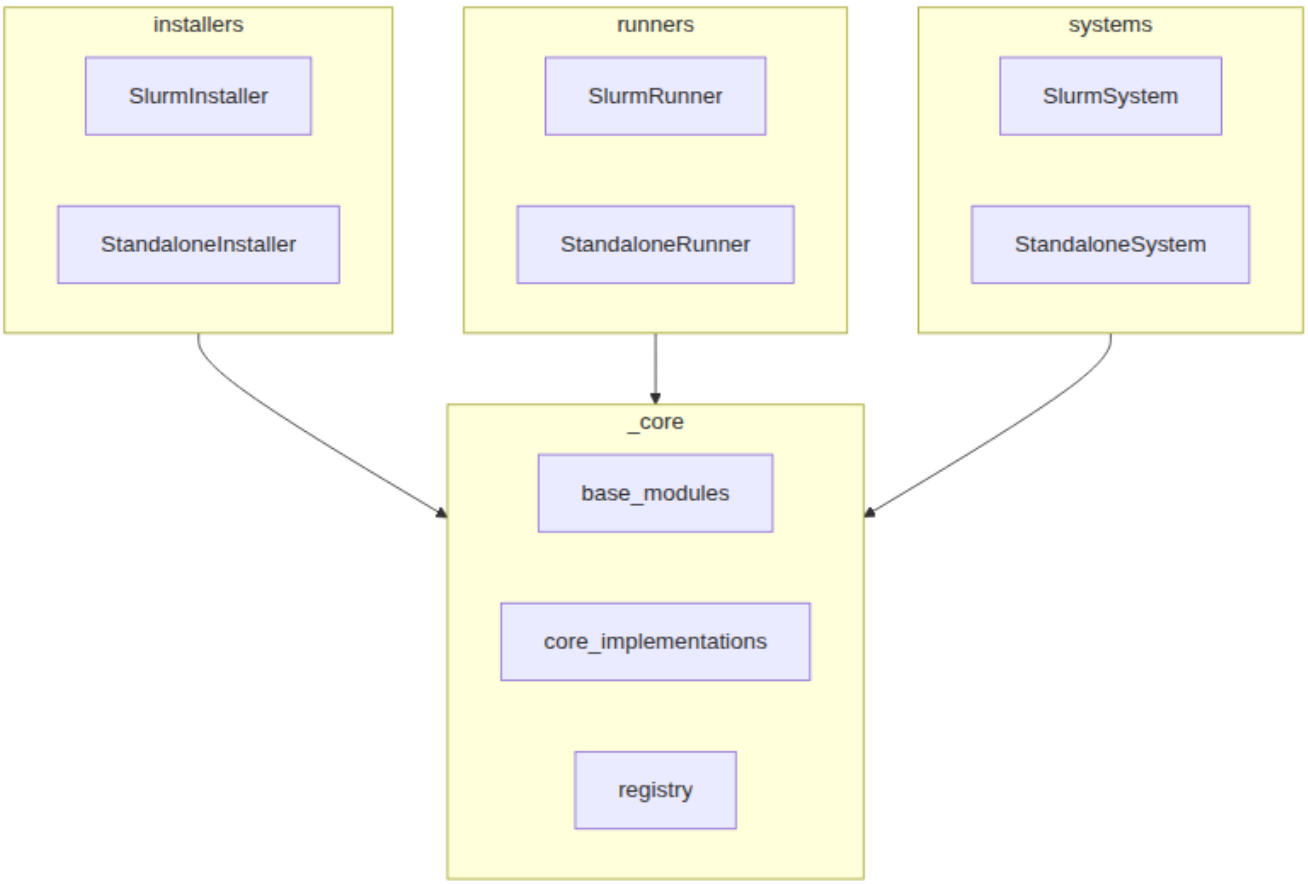
To add documentation for a new workload:

1. Add docstrings to your Python classes and methods.
2. Create a reStructuredText file in `doc/workloads/` (e.g., `my_workload.rst`).
3. Add it to the table above.

The documentation will be automatically generated during the build process.

Development

This chapter targets developers who want to contribute to the project's core.



Core Modules

We use import-linter to ensure no core modules import higher level modules.

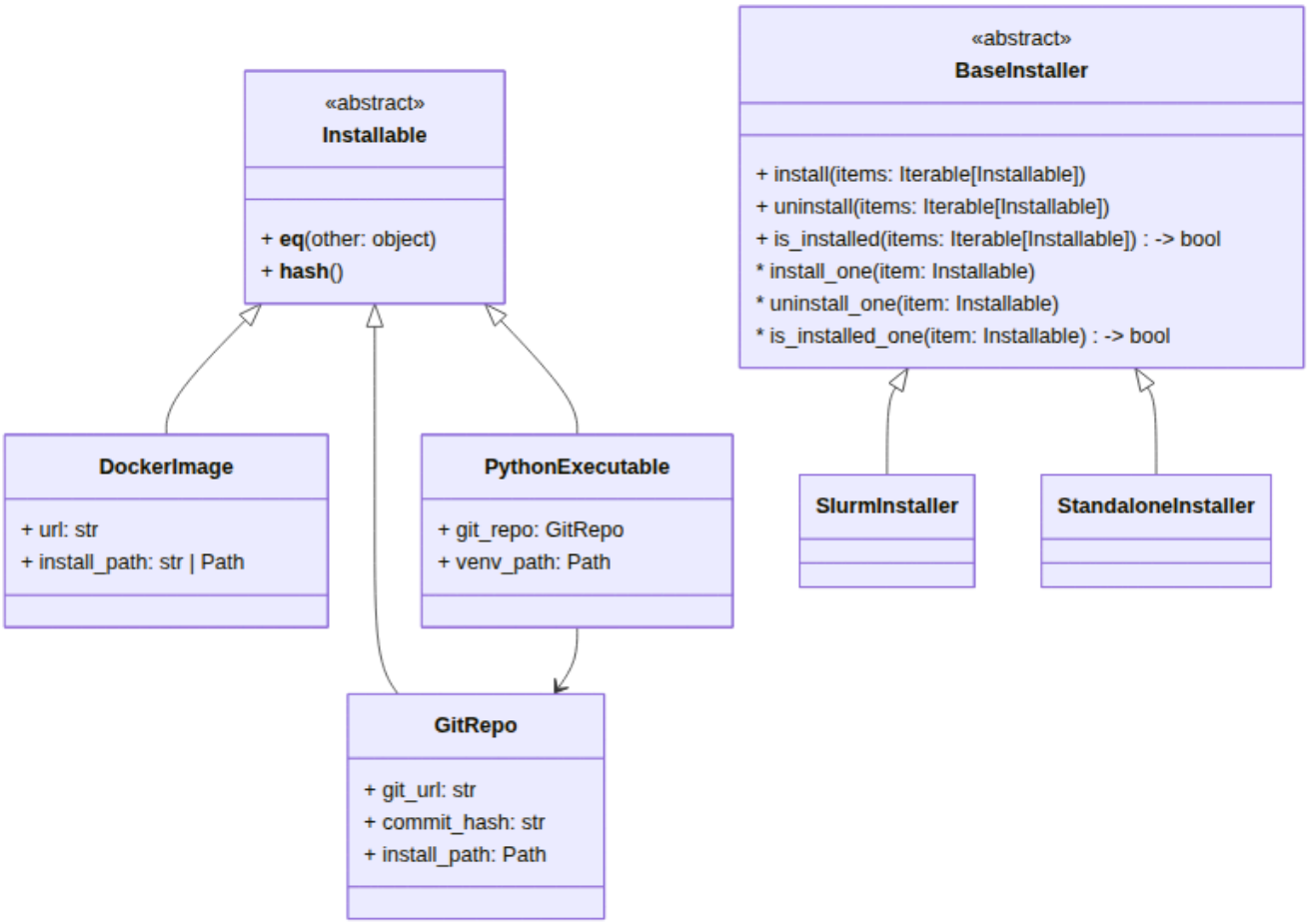
`Registry` object is a singleton that holds implementation mappings. Users can register their own implementations to the registry or replace the default implementations.

Cache

Some prerequisites can be installed. For example:

Docker images, git repos with executable scripts, etc. All such “installables” are kept under the system’s `install_path`.

Installables are shared among all tests. Therefore, if any number of tests use the same installable, it is installed only once for a particular system TOML.



Troubleshooting

This section guides you through identifying the root cause of issues, determining whether they stem from system infrastructure or a bug in CloudAI.

Identifying the Root Cause

If you encounter issues running a command, start by reading the error message to understand the root cause. We strive to make our error messages and exception messages as readable and interpretable as possible.

System Infrastructure vs. CloudAI Bugs

To determine whether an issue is due to system infrastructure or a CloudAI bug, follow these steps:

1. **Check stdout Messages:** If CloudAI fails to run a test successfully, it will be indicated in the stdout messages that a test has failed.
2. **Review Log Files:**
    - Navigate to the output directory and review `debug.log`, stdout, and stderr files
    - `debug.log` contains detailed steps executed by CloudAI, including generated commands, executed commands, and error messages
3. **Analyze Error Messages:** By examining the error messages in the log files, you can understand the type of errors CloudAI encountered.
4. **Examine Output Directory:** If a test fails without explicit error messages, review the output directory of the failed test. Look for `stdout.txt`, `stderr.txt`, or any generated files to understand the failure reason.
5. **Manual Rerun of Tests:**
    - To manually rerun the test, consult the `debug.log` for the command CloudAI executed

- Look for an `sbatch` command with a generated `sbatch` script
- Execute the command manually to debug further

If the problem persists, please report the issue at https://github.com/NVIDIA/cloudai/issues/new/choose. When you report an issue, please make sure it is reproducible. Follow the issue template and provide any necessary details, such as the hash commit used, system settings, any changes in the schema files, and the command.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF

ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.