EvalAI Documentation

Release 1.1

CloudCV Team

Aug 29, 2023

Contents

1 Introduction **3**
- 1.1 Features . . . 3

2 Installation **5**
- 2.1 Dependencies . . . 5
- 2.2 Installation instructions . . . 5

3 Host challenge **7**
- 3.1 Host challenge using github . . . 8
- 3.2 Host prediction upload based challenge . . . 9
- 3.3 Host code-upload based challenge . . . 10
- 3.4 Host static code-upload based challenge . . . 11
- 3.5 Host a remote evaluation challenge . . . 13

4 Challenge configuration **15**

5 Writing Evaluation Script **21**
- 5.1 Writing an Evaluation Script . . . 21
- 5.2 Writing Code-Upload Challenge Evaluation . . . 23
- 5.3 Writing Remote Evaluation Script . . . 26
- 5.4 Writing Static Code-Upload Challenge Evaluation Script . . . 27

6 Approve a challenge (for forked version) **31**
- 6.1 Step 1: Approve challenge using django admin . . . 31
- 6.2 Step 2: Reload submission worker . . . 31

7 Participate in a challenge **33**
- 7.1 1. Visit eval.ai . . . 33
- 7.2 2. Sign up or Log in . . . 33
- 7.3 3. Choose challenge . . . 33
- 7.4 4. Challenge Page . . . 33
- 7.5 5. Create Participant Team . . . 34

8 Make Submission Public **35**

9 Make Submission Private **37**

10 Pull Request **39**

11 Contributing guidelines **41**
11.1 Setting things up . . . 41
11.2 Finding something to work on . . . 41
11.3 Instructions to submit code . . . 41

12 Architecture **43**
12.1 Django . . . 43
12.2 Django Rest Framework . . . 43
12.3 Amazon SQS . . . 43
12.4 PostgreSQL . . . 43
12.5 Angular JS . . . 43

13 Architectural decisions **45**
13.1 URL Patterns . . . 45
13.2 Processing submission messages asynchronously . . . 46
13.3 Submission Worker . . . 46

14 Directory structure **47**
14.1 Django apps . . . 47
14.2 Settings . . . 48
14.3 URLs . . . 48
14.4 Frontend . . . 48
14.5 Scripts . . . 48
14.6 Test Suite . . . 48
14.7 Management Commands . . . 49

15 Submission **51**
15.1 How is a submission processed? . . . 51
15.2 How does submission worker function? . . . 51
15.3 How is submission made? . . . 52
15.4 Format of submission messages . . . 53
15.5 How workers process submission message . . . 53
15.6 Notes . . . 54

16 Migrations **55**
16.1 Creating a migration . . . 55

17 Cite **57**

18 Frequently Asked Questions **59**
18.1 Q. How to start contributing? . . . 59
18.2 Q. What are the technologies that EvalAI uses? . . . 59
18.3 Q. Where could I learn GitHub Commands? . . . 59
18.4 Q. Where could I learn Markdown? . . . 59
18.5 Q. What to do when coverage decreases in your pull request? . . . 59
18.6 Q. How to setup EvalAI using virtualenv? . . . 60
18.7 Common Errors during installation . . . 60

19 Glossary **65**
19.1 Challenge . . . 65
19.2 Challenge host . . . 65
19.3 Challenge host team . . . 65
19.4 Challenge phase . . . 65
19.5 Challenge phase split . . . 66
19.6 Dataset . . . 66

19.7 Dataset split . . . 66
19.8 EvalAI . . . 66
19.9 Leaderboard . . . 66
19.10 Phase . . . 66
19.11 Participant . . . 66
19.12 Participant team . . . 67
19.13 Submission . . . 67
19.14 Submission worker . . . 67
19.15 Team . . . 67
19.16 Test annotation file . . . 67

20 Indices and tables **69**

EvalAI is an open source platform for evaluating and comparing machine learning (ML) and artificial intelligence algorithms (AI) at scale.

It is built to provide a scalable solution to the research community to fulfill the critical need of evaluating machine learning models and agents acting in an environment against annotations or with a human-in-the-loop.

Contents:

CHAPTER 1

Introduction

EvalAI aims to build a centralized platform to host, participate, and collaborate in Artificial Intelligence (AI) challenges organized around the globe and hope to help in benchmarking progress in AI.

1.1 Features

1.1.1 Custom evaluation protocol

We allow creation of an arbitrary number of evaluation phases and dataset splits, compatibility using any programming language, and organizing results in both public and private leaderboards.

1.1.2 Remote evaluation

Certain large-scale challenges need special compute capabilities for evaluation. If the challenge needs extra computational power, challenge organizers can easily add their own cluster of worker nodes to process participant submissions while we take care of hosting the challenge, handling user submissions, and maintaining the leaderboard.

1.1.3 Evaluation inside RL environments

EvalAI lets participants submit code for their agent in the form of docker images which are evaluated against test environments on the evaluation server. During evaluation, the worker fetches the image, test environment, and the model snapshot and spins up a new container to perform evaluation.

1.1.4 CLI support

EvalAI-CLI is designed to extend the functionality of the EvalAI web application to your command line to make the platform more accessible and terminal-friendly.

1.1.5 Portability

EvalAI was designed with keeping in mind scalability and portability of such a system from the very inception of the idea. Most of the components rely heavily on open-source technologies – Docker, Django, Node.js, and PostgreSQL.

1.1.6 Faster evaluation

We warm-up the worker nodes at start-up by importing the challenge code and pre-loading the dataset in memory. We also split the dataset into small chunks that are simultaneously evaluated on multiple cores. These simple tricks result in faster evaluation and reduces the evaluation time by an order of magnitude in some cases.

CHAPTER 2

Installation

2.1 Dependencies

EvalAI can run on Linux, Cloud, Windows, and macOS platforms. Please install docker and docker-compose before getting started with the installation of EvalAI.

2.2 Installation instructions

Once you have installed docker and docker-compose, please follow these steps to setup EvalAI on your local machine.

1. Get the source code on to your machine via git

```
git clone https://github.com/Cloud-CV/EvalAI.git evalai && cd "$_"
```

2. Build and run the Docker containers. This might take a while.

```
docker-compose up --build
```

3. That's it. Open web browser and hit the URL http://127.0.0.1:8888. Three users will be created by default which are listed below:

If you are facing any issue during installation, please see our common errors during installation page.

CHAPTER 3

Host challenge

EvalAI supports hosting challenges with different configurations. Challenge organizers can choose to customize most aspects of the challenge but not limited to:

- Evaluation metrics
- Language/Framework to implement the metric
- Number of phases and data-splits
- Daily / monthly / overall submission limit
- Number of workers evaluating submissions
- Evaluation on remote machines
- Provide your AWS credentials to host code-upload based challenge
- Show / hide error bars on leaderboard
- Public / private leaderboards
- Allow / block certain email addresses to participate in the challenge or phase
- Choose which fields to export while downloading challenge submissions

We have hosted challenges from different domains such as:

- Machine learning (2019 SIOP Machine Learning Competition)
- Deep learning (Visual Dialog Challenge 2019)
- Computer vision (Vision and Language Navigation)
- Natural language processing (VQA Challenge 2019)
- Healthcare (fastMRI Image Reconstruction)
- Self-driving cars (CARLA Autonomous Driving Challenge)

We categorize the challenges in two categories:

1. **Prediction upload based challenges**: Participants upload predictions corresponding to ground truth labels in the form of a file (could be any format: `json`, `npy`, `csv`, `txt` etc.)

   Some of the popular prediction upload based challenges that we have hosted are shown below:

   If you are interested in hosting prediction upload based challenges, then click here.

2. **Code-upload based challenges**: In these kind of challenges, participants upload their training code in the form of docker images using EvalAI-CLI.

   We support two types of code-upload based challenges -

   - Code-Upload Based Challenge (without Static Dataset): These are usually reinforcement learning challenges which involve uploading a trained model in form of docker images and the environment is also saved in form of a docker image.
   - Static Code-Upload Based Challenge: These are challenges where the host might want the participants to upload models and they have static dataset on which they want to run the models and perform evaluations. This kind of challenge is especially useful in case of data privacy concerns.

   We support two types of code-upload based challenges -

   - Code-Upload Based Challenge (without Static Dataset): These are usually reinforcement learning challenges which involve uploading a trained agent in form of docker images and the environment is also saved in form of a docker image.
   - Static Code-Upload Based Challenge: These are challenges where the host might want the participants to upload models and they have static dataset on which they want to run the models and perform evaluations. This kind of challenge is especially useful in case of data privacy concerns.

   Some of the popular code-upload based challenges that we have hosted are shown below:

   If you are interested in hosting code-upload based challenges, then click here. If you are interested in hosting static code-upload based challenges, then click here.

   A good reference would be the Habitat Re-arrangement Challenge 2022.

3.1 Host challenge using github

3.1.1 Step 1: Use template

Use EvalAI-Starters template. See this on how to use a repository as template.

3.1.2 Step 2: Generate github token

Generate your github personal acccess token and copy it in clipboard.

Add the github personal access token in the forked repository's secrets with the name `AUTH_TOKEN`.

3.1.3 Step 3: Setup host configuration

Now, go to EvalAI to fetch the following details -

1. `evalai_user_auth_token` - Go to profile page after logging in and click on `Get your Auth Token` to copy your auth token.
2. `host_team_pk` - Go to host team page and copy the `ID` for the team you want to use for challenge creation.

3. `evalai_host_url` - Use `https://eval.ai` for production server and `https://staging.eval.ai` for staging server.

3.1.4 Step 4: Setup automated update push

Create a branch with name `challenge` in the forked repository from the `master` branch. Note: Only changes in `challenge` branch will be synchronized with challenge on EvalAI.

Add `evalai_user_auth_token` and `host_team_pk` in `github/host_config.json`.

3.1.5 Step 5: Update challenge details

Read EvalAI challenge creation documentation to know more about how you want to structure your challenge. Once you are ready, start making changes in the yaml file, HTML templates, evaluation script according to your need.

3.1.6 Step 6: Push changes to the challenge

Commit the changes and push the `challenge` branch in the repository and wait for the build to complete. View the logs of your build.

If challenge config contains errors then a `issue` will be opened automatically in the repository with the errors otherwise the challenge will be created on EvalAI.

3.1.7 Step 7: Verify challenge

Go to Hosted Challenges to view your challenge. The challenge will be publicly available once EvalAI admin approves the challenge.

To update the challenge on EvalAI, make changes in the repository and push on `challenge` branch and wait for the build to complete.

3.2 Host prediction upload based challenge

3.2.1 Step 1: Setup challenge configuration

We have created a sample challenge configuration that we recommend you to use to get started. Use EvalAI-Starters template to start. See this on how to use a repository as template.

3.2.2 Step 2: Edit challenge configuration

Open `challenge_config.yml` from the repository that you cloned in step-1. This file defines all the different settings of your challenge such as start date, end date, number of phases, and submission limits etc.

Edit this file based on your requirement. For reference to the fields, refer to the challenge configuration reference section.

3.2.3 Step 3: Edit evaluation script

Next step is to edit the challenge evaluation script that decides what metrics the submissions are going to be evaluated on for different phases.

Please refer to the writing evaluation script to complete this step.

3.2.4 Step 4: Edit challenge HTML templates

Almost there. You just need to update the HTML templates in the `templates/` directory of the bundle that you cloned.

EvalAI supports all kinds of HTML tags which means you can add images, videos, tables etc. Moreover, you can add inline CSS to add custom styling to your challenge details.

Congratulations! you have submitted your challenge configuration for review and EvalAI team has notified about this. EvalAI team will review and will approve the challenge.

If you have issues in creating a challenge on EvalAI, please feel free to contact us at team@cloudcv.org create an issue on our GitHub issues page.

3.3 Host code-upload based challenge

3.3.1 Step 1: Setup challenge repository

Steps to create a code-upload based challenge are somewhat similar to what it takes to create a prediction upload based challenge.

We have created a sample challenge repository that we recommend you to use to get started. Use EvalAI-Starters template to start. See this on how to use a repository as template.

3.3.2 Step 2: Edit challenge configuration

Open `challenge_config.yml` from the repository that you cloned in step-1. This file defines all the different settings of your challenge such as start date, end date, number of phases, and submission limits etc. Edit this file based on your requirement.

Please ensure the following fields are set to the following values for code-upload based challenges:

- `remote_evaluation : True`
- `is_docker_based : True`

In order to perform evaluation, you might also need to create an EKS cluster on AWS. This is because we expect to use docker containers for both - the agent, and the environment. See AWS Elastic Kubernetes Service to learn more about what EKS is and how it works.

We need the following details for the EKS cluster in order to perform evaluations in case you are using your own AWS account:

- `aws_account_id: <AWS Account ID>`
- `aws_access_key_id: <AWS Access Key ID>`
- `aws_secret_access_key: <AWS Secret Access Key>`
- `aws_region: <AWS Region>`

These details need to be emailed us at team@cloudcv.org. The EvalAI team will set up the infrastructure in your AWS account. For reference to the other fields, refer to the challenge configuration reference section.

3.3.3 Step 3: Edit evaluation code

Next step is to create code-upload challenge evaluation that decides what metrics the submissions are going to be evaluated on for different phases.

For code-upload challenges, the environment image is expected to be created by the host and the agent image is to be pushed by the participants.

Please refer to the Writing Code-Upload Challenge Evaluation section to complete this step.

3.3.4 Step 4: Edit challenge HTML templates

Almost there. You just need to update the HTML templates in the `templates/` directory of the bundle that you cloned.

EvalAI supports all kinds of HTML tags which means you can add images, videos, tables etc. Moreover, you can add inline CSS to add custom styling to your challenge details.

Please include a detailed `submission_guidelines.html` as it is usually not as straightforward for the participants to upload submissions for code-upload challenges.

The participants are expected to submit links to their agent docker images using `evalai-cli`. Here is an example of a command:

```
evalai push <image>:<tag> --phase <phase_name>
```

Please refer to the documentation for more details on this.

A good example of submission guidelines for code-upload challenges is present here.

Congratulations! you have submitted your challenge configuration for review and EvalAI team has notified about this. EvalAI team will review and will approve the challenge.

3.4 Host static code-upload based challenge

3.4.1 Step 1: Setup challenge repository

Steps to create a static code-upload based challenge are very similar to what it takes to create a prediction upload based challenge and code-upload based challenge.

We have created a sample challenge repository that we recommend you to use to get started. Use EvalAI-Starters template to start. See this on how to use a repository as template.

3.4.2 Step 2: Edit challenge configuration

Open `challenge_config.yml` from the repository that you cloned in step-1. This file defines all the different settings of your challenge such as start date, end date, number of phases, and submission limits etc. Edit this file based on your requirement.

Please ensure the following fields are set to the following values for static code-upload based challenges:

- `remote_evaluation : True`

- `is_docker_based : True`
- `is_static_dataset_code_upload : True`

In order to perform evaluation, you might also need to create an EKS cluster on AWS. This is because we use docker containers for managing the evaluation environment, and as well as model container. See AWS Elastic Kubernetes Service to learn more about what EKS is and how it works.

We need the following details for the EKS cluster in order to perform evaluations in case you are using your own AWS account:

- `aws_account_id: <AWS Account ID>`
- `aws_access_key_id: <AWS Access Key ID>`
- `aws_secret_access_key: <AWS Secret Access Key>`
- `aws_region: <AWS Region>`

These details need to be emailed us at team@cloudcv.org. The EvalAI team will set up the infrastructure in your AWS account.

For reference to the other fields, refer to the challenge configuration reference section.

3.4.3 Step 3: Save the static dataset using EFS

Use AWS EFS to store the static dataset file(s) on which the evaluation is to be performed. By default an `EFS` file system is created and the file system ID is stored in `efs_id` in the Challenge Evaluation Cluster and then the file system is mounted on the instances inside the cluster.

3.4.4 Step 4: Create a sample submission Dockerfile

Create a Dockerfile showing the participants how to install their requirements and run the submission script which produces the predictions file. This docker image is then run on the cluster to perform predictions.

An template Dockerfile is shown below:

```
FROM nvidia/cuda:11.2.0-cudnn8-runtime-ubuntu20.04

RUN apt-get update &&\
    DEBIAN_FRONTEND=noninteractive apt-get install -y python3 &&\
    apt-get install -y

# ADITIONAL PYTHON DEPENDENCIES
COPY requirements.txt ./
RUN pip install -r requirements.txt

WORKDIR /app

# COPY WHATEVER OTHER SCRIPTS YOU MAY NEED
COPY trained_model /trained_model
# SPECIFY THE ENTRYPOINT SCRIPT
CMD ["python", "-u", "submission.py"]
```

This docker image will run `submission.py` script and the script will save the predictions at the specified location.

3.4.5 Step 5: Edit evaluation script

Next step is to write the evaluation script to compute the metrics for each submission.

Please refer to the Writing Static Code-Upload Challenge Evaluation section to complete this step.

3.4.6 Step 6: Prepare detailed documentation

Prepare a detailed documentation describing the following details:

- Input/Output Format: Details about the model input format and the expected model output format for the participants with examples.
- Expected Input/Output Files Names: The documentation should also contain where the dataset is expected to be stored in the docker container, where to save the file, and the output file name.
- Docker commands/script: The docker commands/script to create the docker container from the Docker file.
- Submission Command: The `evalai-cli` command to push the container.
- Any other installation, processing, training tips required for the task.

It is recommended to look at the example of My Seizure Gauge Forecasting Challenge 2022 which contain extensively described steps and documentation for everything, along with tips for every step of challenge participation.

3.4.7 Step 7: Edit challenge HTML templates

Almost there. You just need to update the HTML templates in the `templates/` directory of the bundle that you cloned.

EvalAI supports all kinds of HTML tags which means you can add images, videos, tables etc. Moreover, you can add inline CSS to add custom styling to your challenge details.

Please include a detailed `submission_guidelines.html` as it is usually not as straightforward for the participants to upload submissions for static code-upload challenges.

The participants are expected to submit links to their model docker images using `evalai-cli`. Here is an example of a command:

```
evalai push <image>:<tag> --phase <phase_name>
```

Please refer to the documentation for more details on this.

A good example of submission guidelines is present here.

Congratulations! you have submitted your challenge configuration for review and EvalAI team has notified about this. EvalAI team will review and will approve the challenge.

3.5 Host a remote evaluation challenge

3.5.1 Step 1: Set up the challenge

Follow host challenge using github section to set up a challenge on EvalAI.

3.5.2 Step 2: Edit challenge configuration

Set the `remote_evaluation` parameter to `True` in `challenge_config.yaml`. This challenge config file defines all the different settings of your challenge such as start date, end date, number of phases, and submission limits etc.

Edit this file based on your requirement. For reference to the fields, refer to the challenge configuration reference section.

Please ensure the following fields are set to the following values:

- `remote_evaluation : True`

Refer to the following documentation for details on challenge configuration.

3.5.3 Step 3: Edit remote evaluation script

Next step is to edit the challenge evaluation script that decides what metrics the submissions are going to be evaluated on for different phases. Please refer to Writing Remote Evaluation Script section to complete this step.

3.5.4 Step 4: Set up remote evaluation worker

1. Create conda environment to run the evaluation worker. Refer to conda's create environment section to set up a virtual environment.
2. Install the worker requirements from the `EvalAI-Starters/remote_challenge_evaluation` present here:

```
cd EvalAI-Starters/
pip install remote_challenge_evaluation/requirements.txt
```

3. Start evaluation worker:

```
cd EvalAI-Starters/remote_challenge_evaluation
python main.py
```

If you have issues in creating a challenge on EvalAI, please feel free to contact us at team@cloudcv.org create an issue on our GitHub issues page.

CHAPTER 4

Challenge configuration

Following fields are required (and can be customized) in the `challenge_config.yml`.

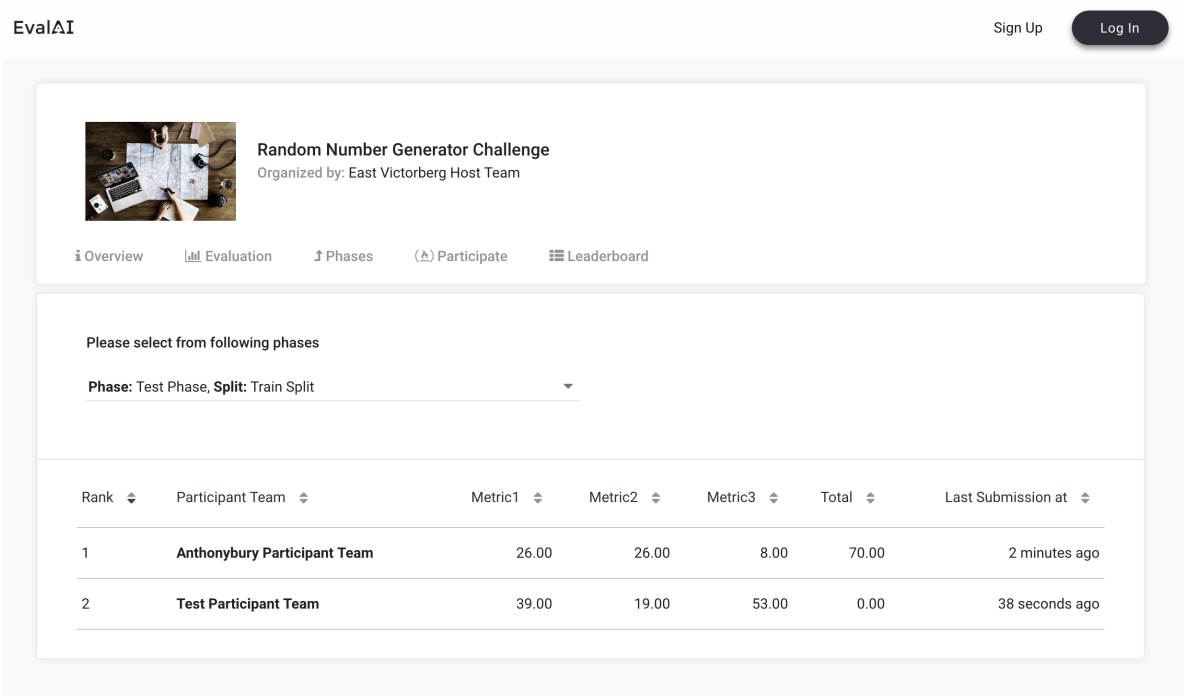
- **title**: Title of the challenge
- **short_description**: Short description of the challenge (preferably 140 characters max)
- **description**: Long description of the challenge (use a relative path for the HTML file, e.g. `templates/description.html`)
- **evaluation_details**: Evaluation details and details of the challenge (use a relative path for the HTML file, e.g. `templates/evaluation_details.html`)
- **terms_and_conditions**: Terms and conditions of the challenge (use a relative path for the HTML file, e.g. `templates/terms_and_conditions.html`)
- **image**: Logo of the challenge (use a relative path for the logo in the zip configuration, e.g. `images/logo/challenge_logo.jpg`). **Note**: The image must be in jpg, jpeg or png format.
- **submission_guidelines**: Submission guidelines of the challenge (use a relative path for the HTML file, e.g. `templates/submission_guidelines.html`)
- **evaluation_script**: Python script which will decide how to evaluate submissions in different phases (path of the evaluation script file or folder relative to this YAML file. For e.g. `evaluation_script/`)
- **remote_evaluation**: True/False (specify whether evaluation will happen on a remote machine or not. Default is `False`)
- **is_docker_based**: True/False (specify whether the challenge is docker based or not. Default is `False`)
- **is_static_dataset_code_upload**: True/False (specify whether the challenge is static dataset code upload or not. Default is `False`)
- **start_date**: Start DateTime of the challenge (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10) in `UTC` time zone
- **end_date**: End DateTime of the challenge (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10) in `UTC` time zone

- **published**: True/False (Boolean field that gives the flexibility to publish the challenge once approved by EvalAI admin. Default is `False`)
- **allowed_email_domains**: A list of domains allowed to participate in the challenge. Leave blank if everyone is allowed to participate. (e.g. `["domain1.com", "domain2.org", "domain3.in"]` Participants with these email domains will only be allowed to participate.)
- **blocked_emails_domains**: A list of domains not allowed to participate in the challenge. Leave blank if everyone is allowed to participate. (e.g. `["domain1.com", "domain2.org", "domain3.in"]` Participants with these email domains will not be allowed to participate.)
- **leaderboard**: A leaderboard for a challenge on EvalAI consists of following subfields:
  - **id**: Unique positive integer field for each leaderboard entry
  - **schema**: Schema field contains the information about the rows of the leaderboard. A schema contains two keys in the leaderboard:
    1. `labels`: Labels are the header rows in the leaderboard according to which the challenge ranking is done.
    2. `default_order_by`: This key decides the default sorting of the leaderboard based on one of the labels defined above.
    3. `metadata`: This field defines additional information about the metrics that are used to evaluate the challenge submissions.

The leaderboard schema for the sample challenge configuration looks like this:

```
leaderboard:
  - id: 1
    schema:
      {
        "labels": ["Metric1", "Metric2", "Metric3", "Total"],
        "default_order_by": "Total",
        "metadata": {
        "Metric1": {
          "sort_ascending": True,
          "description": "Metric Description",
        }
      }
```

The above leaderboard schema will look something like this on leaderboard UI:



- **challenge_phases**:

  There can be multiple challenge phases in a challenge. A challenge phase in a challenge contains the following subfields:

  - **id**: Unique integer identifier for the challenge phase
  - **name**: Name of the challenge phase
  - **description**: Long description of the challenge phase (set the relative path of the HTML file, e.g. `templates/challenge_phase_1_description.html`)
  - **leaderboard_public**: True/False (a Boolean field that gives the flexibility to Challenge Hosts to either make the leaderboard public or private. Default is `False`)
  - **is_public**: True/False (a Boolean field that gives the flexibility to Challenge Hosts to either hide or show the challenge phase to participants. Default is `False`)
  - **is_submission_public**: True/False (a Boolean field that gives the flexibility to Challenge Hosts to either make the submissions by default public/private. Note that this will only work when the `leaderboard_public` property is set to true. Default is `False`)
  - **start_date**: Start DateTime of the challenge phase (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10)
  - **end_date**: End DateTime of the challenge phase (Format: YYYY-MM-DD HH:MM:SS, e.g. 2017-07-07 10:10:10)
  - **test_annotation_file**: This file is used for ranking the submission made by a participant. An annotation file can be shared by more than one challenge phase. (Path of the test annotation file relative to this YAML file, e.g. `annotations/test_annotations_devsplit.json`)
  - **codename**: Unique id for each challenge phase. Note that the codename of a challenge phase is used to map the results returned by the evaluation script to a particular challenge phase. The codename specified here should match with the codename specified in the evaluation script to perfect mapping.

- **max_submissions_per_day**: A positive integer that tells the maximum number of submissions per day to a challenge phase. (Optional, Default value is `100000`)
- **max_submissions_per_month**: A positive integer that tells the maximum number of submissions per month to a challenge phase. (Optional, Default value is `100000`)
- **max_submissions**: A positive integer that decides the maximum number of total submissions that can be made to the challenge phase. (Optional, Default value is `100000`)
- **default_submission_meta_attributes**: These are the default metadata attributes that will be displayed for all submissions, the metadata attributes are `method_name`, `method_description`, `project_url`, and `publication_url`.

```
default_submission_meta_attributes:
- name: method_name
  is_visible: True
- name: method_description
  is_visible: True
- name: project_url
  is_visible: True
- name: publication_url
  is_visible: True
```

- **submission_meta_attributes**: These are the custom metadata attributes that participants can add to their submissions. The custom metadata attributes are `TextAttribute`, `SingleOptionAttribute`, `MultipleChoiceAttribute`, and `TrueFalseField`.

```
submission_meta_attributes:
  - name: TextAttribute
    description: Sample
    type: text
    required: False
  - name: SingleOptionAttribute
    description: Sample
    type: radio
    options: ["A", "B", "C"]
  - name: MultipleChoiceAttribute
    description: Sample
    type: checkbox
    options: ["alpha", "beta", "gamma"]
  - name: TrueFalseField
    description: Sample
    type: boolean
    required: True
```

- **is_restricted_to_select_one_submission**: True/False (indicates whether to restrict a user to select only one submission for the leaderboard. Default is `False`)
- **is_partial_submission_evaluation_enabled**: True/False (indicates whether partial submission evaluation is enabled. Default is `False`)
- **allowed_submission_file_types**: This is a list of file types that are allowed for submission (Optional Default is `.json, .zip, .txt, .tsv, .gz, .csv, .h5, .npy`)

- **dataset_splits**:

  Dataset splits define the subset of test-set on which the submissions will be evaluated on. Generally, most challenges have three splits:

  1. **train_split** (Allow participants to make a large number of submissions, let them see how they are doing, and let them overfit)

2. **test_split** (Allow a small number of submissions so that they cannot mimic test_set. Use this split to decide the winners for the challenge)
3. **val_split** (Allow participants to make submissions and evaluate on the validation split)

A dataset split has the following subfields:

- **id**: Unique integer identifier for the split
- **name**: Name of the split (it must be unique for every split)
- **codename**: Unique id for each split. Note that the codename of a dataset split is used to map the results returned by the evaluation script to a particular dataset split in EvalAI's database. Please make sure that no two dataset splits have the same codename. Again, make sure that the dataset split's codename match with what is in the evaluation script provided by the challenge host.

- **challenge_phase_splits**:

  A challenge phase split is a relation between a challenge phase and dataset splits for a challenge (many to many relation). This is used to set the privacy of submissions (public/private) to different dataset splits for different challenge phases.

  - **challenge_phase_id**: Id of `challenge_phase` to map with
  - **leaderboard_id**: Id of `leaderboard`
  - **dataset_split_id**: Id of `dataset_split`
  - **visibility**: It will set the visibility of the numbers corresponding to metrics for this `challenge_phase_split`. Select one of the following positive integers based on the visibility level you want: (Optional, Default is `3`)

- **leaderboard_decimal_precision**: A positive integer field used for varying the leaderboard decimal precision. Default value is `2`.
- **is_leaderboard_order_descending**: True/False (a Boolean field that gives the flexibility to challenge host to change the default leaderboard sorting order. It is useful in cases where you have error as a metric and want to sort the leaderboard in increasing order of error value. Default is `True`)

CHAPTER 5

Writing Evaluation Script

5.1 Writing an Evaluation Script

Each challenge has an evaluation script, which evaluates the submission of participants and returns the scores which will populate the leaderboard. The logic for evaluating and judging a submission is customizable and varies from challenge to challenge, but the overall structure of evaluation scripts are fixed due to architectural reasons.

Evaluation scripts are required to have an `evaluate()` function. This is the main function, which is used by workers to evaluate the submission messages.

The syntax of evaluate function is:

```
def evaluate(test_annotation_file, user_annotation_file, phase_codename, **kwargs):
    pass
```

It receives three arguments, namely:

- `test_annotation_file`: It represents the local path to the annotation file for the challenge. This is the file uploaded by the Challenge host while creating a challenge.
- `user_annotation_file`: It represents the local path of the file submitted by the user for a particular challenge phase.
- `phase_codename`: It is the `codename` of the challenge phase from the challenge configuration yaml. This is passed as an argument so that the script can take actions according to the challenge phase.

After reading the files, some custom actions can be performed. This varies per challenge.

The `evaluate()` method also accepts keyword arguments. By default, we provide you metadata of each submission to your challenge which you can use to send notifications to your slack channel or to some other webhook service. Following is an example code showing how to get the submission metadata in your evaluation script and send a slack notification if the accuracy is more than some value `X` (X being 90 in the example given below).

```
def evaluate(test_annotation_file, user_annotation_file, phase_codename, **kwargs):

    submission_metadata = kwargs.get("submission_metadata")
```

(continues on next page)

(continued from previous page)

```
    print submission_metadata

    # Do stuff here
    # Set `score` to 91 as an example

    score = 91
    if score > 90:
        slack_data = kwargs.get("submission_metadata")
        webhook_url = "Your slack webhook url comes here"
        # To know more about slack webhook, checkout this link: https://api.slack.com/
↪incoming-webhooks

        response = requests.post(
            webhook_url,
            data=json.dumps({'text': "*Flag raised for submission:* \n \n" +␣
↪str(slack_data)}),
            headers={'Content-Type': 'application/json'})

    # Do more stuff here
```

The above example can be modified and used to find if some participant team is cheating or not. There are many more ways for which you can use this metadata.

After all the processing is done, this `evaluate()` should return an output, which is used to populate the leaderboard. The output should be in the following format:

```
output = {}
output['result'] = [
            {
                'train_split': {
                    'Metric1': 123,
                    'Metric2': 123,
                    'Metric3': 123,
                    'Total': 123,
                }
            },
            {
                'test_split': {
                    'Metric1': 123,
                    'Metric2': 123,
                    'Metric3': 123,
                    'Total': 123,
                }
            }
        ]

return output
```

Let's break down what is happening in the above code snippet.

1. `output` should contain a key named `result`, which is a list containing entries per dataset split that is available for the challenge phase in consideration (in the function definition of `evaluate()` shown above, the argument: `phase_codename` will receive the *codename* for the challenge phase against which the submission was made).
2. Each entry in the list should be a dict that has a key with the corresponding dataset split codename (`train_split` and `test_split` for this example).

3. Each of these dataset split dict contains various keys (`Metric1`, `Metric2`, `Metric3`, `Total` in this example), which are then displayed as columns in the leaderboard.

5.2 Writing Code-Upload Challenge Evaluation

Each challenge has an evaluation script, which evaluates the submission of participants and returns the scores which will populate the leaderboard. The logic for evaluating and judging a submission is customizable and varies from challenge to challenge, but the overall structure of evaluation scripts is fixed due to architectural reasons.

In code-upload challenges, the evaluation is tighly-coupled with the agent and environment containers:

1. The agent interacts with environment via actions and provides a 'stop' signal when finished.
2. The environment provides feedback to the agent until 'stop' signal is received, episodes run out or the time limit is over.

The starter templates for code-upload challenge evaluation can be found here.

The steps to configure evaluation for code-upload challenges are:

1. **Create an environment**: There are few steps involved in creating an environment:
    1. *Edit the evaluator_environment*: This class defines the environment (a gym environment or a habitat environment) and other related attributes/methods. Modify the `evaluator_environment` containing a gym environment shown here:

```
class evaluator_environment:
    def __init__(self, environment="CartPole-v0"):
        self.score = 0
        self.feedback = None
        self.env = gym.make(environment)
        self.env.reset()

    def get_action_space(self):
        return list(range(self.env.action_space.n))

    def next_score(self):
        self.score += 1
```

    There are three methods in this example:

    - `__init__`: The initialization method to instantiate and set up the evaluation environment.
    - `get_action_space`: Returns the action space of the agent in the environment.
    - `next_score`: Returns/updates the reward achieved.

    You can add custom methods and attributes which help in interaction with the environment.

    2. *Edit the Environment service*: This service is hosted on the gRPC server to get actions in form of messages from the agent container. Modify the lines shown here:

```
class Environment(evaluation_pb2_grpc.EnvironmentServicer):
    def __init__(self, challenge_pk, phase_pk, submission_pk, server):
        self.challenge_pk = challenge_pk
        self.phase_pk = phase_pk
        self.submission_pk = submission_pk
        self.server = server
```

(continues on next page)

(continued from previous page)

```
    def get_action_space(self, request, context):
        message = pack_for_grpc(env.get_action_space())
        return evaluation_pb2.Package(SerializedEntity=message)

    def act_on_environment(self, request, context):
        global EVALUATION_COMPLETED
        if not env.feedback or not env.feedback[2]:
            action = unpack_for_grpc(request.SerializedEntity)
            env.next_score()
            env.feedback = env.env.step(action)
        if env.feedback[2]:
            if not LOCAL_EVALUATION:
                update_submission_result(
                    env, self.challenge_pk, self.phase_pk, self.submission_pk
                )
            else:
                print("Final Score: {0}".format(env.score))
                print("Stopping Evaluation!")
                EVALUATION_COMPLETED = True
        return evaluation_pb2.Package(
            SerializedEntity=pack_for_grpc(
                {"feedback": env.feedback, "current_score": env.score,}
            )
        )
```

You can modify the relevant parts of the environment service in order to make it work for your case. You would need to serialize and deserialize the response/request to pass messages between the agent and environment over gRPC. For this, we have implemented two methods which might be useful:

- `unpack_for_grpc`: This method deserializes entities from request/response sent over gRPC. This is useful for receiving messages (for example, actions from the agent).
- `pack_for_grpc`: This method serializes entities to be sent over a request over gRPC. This is useful for sending messages (for example, feedback from the environment).

Note: This is a basic description of the class and the implementations may vary on a case-by-case basis.

3. *Edit the requirements file*: Change the requirements file according to the packages required by your environment.
4. *Edit environment Dockerfile*: You may choose to modify the Dockerfile that will set up and run the environment service.
5. *Edit the docker environment variables*: Fill in the following information in the `docker.env` file:

```
AUTH_TOKEN=<Add your EvalAI Auth Token here>
EVALAI_API_SERVER=<https://eval.ai>
LOCAL_EVALUATION = True
QUEUE_NAME=<Go to the challenge manage tab to get challenge queue name.>
```

6. *Create the docker image and upload on ECR*: Create an environment docker image for the created `Dockerfile` by using:

```
docker build -f <file_path_to_Dockerfile>
```

Upload the created docker image to ECR:

```
aws ecr get-login-password --region <region> | docker login --username AWS --
↪password-stdin <aws_account_id>.dkr.ecr.<region>.amazonaws.com
docker tag <image_id> <aws_account_id>.dkr.ecr.<region>.amazonaws.com/<my-
↪repository>:<tag>
docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.com/<my-repository>:
↪<tag>
```

Detailed steps for uploading a docker image to ECR can be found here.

7. *Add environment image the challenge configuration for challenge phase*: For each challenge phase, add the link to the environment image in the challenge configuration:

```
...
challenge_phases:
    - id: 1
    ...
    - environment_image: <docker image uri>
...
```

Example References:

- Habitat Benchmark: This file contains description of an evaluation class which evaluates agents on the environment.

2. **Create a starter example**: The participants are expected to submit docker images for their agents which will contain the policy and the methods to interact with the environment.

   Like environment, there are a few steps involved in creating the agent:

   1. *Create a starter example script*: Please create a starter agent submission and a local evaluation script in order to help the participants perform sanity checks on their code before making submissions to EvalAI.

      The `agent.py` file should contain a description of the agent, the methods that the environment expects the agent to have, and a `main()` function to pass actions to the environment.

      We provide a template for `agent.py` here:

```
import evaluation_pb2
import evaluation_pb2_grpc
import grpc
import os
import pickle
import time

time.sleep(30)

LOCAL_EVALUATION = os.environ.get("LOCAL_EVALUATION")

if LOCAL_EVALUATION:
    channel = grpc.insecure_channel("environment:8085")
else:
    channel = grpc.insecure_channel("localhost:8085")

stub = evaluation_pb2_grpc.EnvironmentStub(channel)

def pack_for_grpc(entity):
    return pickle.dumps(entity)

def unpack_for_grpc(entity):
```

(continues on next page)

(continued from previous page)

```
    return pickle.loads(entity)

flag = None

while not flag:
    base = unpack_for_grpc(
        stub.act_on_environment(
            evaluation_pb2.Package(SerializedEntity=pack_for_grpc(1))
        ).SerializedEntity
    )
    flag = base["feedback"][2]
    print("Agent Feedback", base["feedback"])
    print("*"* 100)
```

Other Examples: - A random agent from Habitat Rearrangement Challenge 2022

2. *Edit the requirements file*: Change the requirements file according to the packages required by an agent.
3. *Edit environment Dockerfile*: You may choose to modify the Dockerfile which will run the `agent.py` file and interact with environment.
4. *Edit the docker environment variables*: Fill in the following information in the `docker.env` file:

```
LOCAL_EVALUATION = True
```

Example References:

- Habitat Rearrangement Challenge 2022 - Random Agent: This is an example of a dummy agent created for the Habitat Rearrangement Challenge 2022 which is then sent to the evaluator (here, Habitat Benchmark) for evaluation.

5.3 Writing Remote Evaluation Script

Each challenge has an evaluation script, which evaluates the submission of participants and returns the scores which will populate the leaderboard. The logic for evaluating and judging a submission is customizable and varies from challenge to challenge, but the overall structure of evaluation scripts is fixed due to architectural reasons.

The starter template for remote challenge evaluation can be found here.

Here are the steps to configure remote evaluation:

1. **Setup Configs**:

   To configure authentication for the challenge set the following environment variables:

   1. AUTH_TOKEN: Go to profile page -> Click on `Get your Auth Token` -> Click on the Copy button. The auth token will get copied to your clipboard.
   2. API_SERVER: Use `https://eval.ai` when setting up challenge on production server. Otherwise, use `https://staging.eval.ai`
   3. QUEUE_NAME: Go to the challenge manage tab to fetch the challenge queue name.
   4. CHALLENGE_PK: Go to the challenge manage tab to fetch the challenge primary key.
   5. SAVE_DIR: (Optional) Path to submission data download location.

2. **Write `evaluate` method**: Evaluation scripts are required to have an `evaluate()` function. This is the main function, which is used by workers to evaluate the submission messages.

The syntax of evaluate function for a remote challenge is:

```
def evaluate(user_submission_file, phase_codename, test_annotation_file = None,␣
↪**kwargs)
    pass
```

It receives three arguments, namely:

- `user_annotation_file`: It represents the local path of the file submitted by the user for a particular challenge phase.
- `phase_codename`: It is the `codename` of the challenge phase from the challenge configuration yaml. This is passed as an argument so that the script can take actions according to the challenge phase.
- `test_annotation_file`: It represents the local path to the annotation file for the challenge. This is the file uploaded by the Challenge host while creating a challenge.

You may pass the `test_annotation_file` as default argument or choose to pass separately in the `main.py` depending on the case. The `phase_codename` is passed automatically but is left as an argument to allow customization.

After reading the files, some custom actions can be performed. This varies per challenge.

The `evaluate()` method also accepts keyword arguments.

IMPORTANT : If the `evaluate()` method fails due to any reason or there is a problem with the submission, please ensure to raise an `Exception` with an appropriate message.

5.4 Writing Static Code-Upload Challenge Evaluation Script

Each challenge has an evaluation script, which evaluates the submission of participants and returns the scores which will populate the leaderboard. The logic for evaluating and judging a submission is customizable and varies from challenge to challenge, but the overall structure of evaluation scripts are fixed due to architectural reasons.

The starter templates for static code-upload challenge evaluation can be found here. Note that the evaluation file provided will be used on our submission workers, just like prediction upload challenges.

The steps for writing an evaluation script for a static code-upload based challenge are the same as that for [prediction-upload based challenges] section (evaluation_scripts.html#writing-an-evaluation-script).

Evaluation scripts are required to have an `evaluate()` function. This is the main function, which is used by workers to evaluate the submission messages.

The syntax of evaluate function is:

```
def evaluate(test_annotation_file, user_annotation_file, phase_codename, **kwargs):
    pass
```

It receives three arguments, namely:

- `test_annotation_file`: It represents the local path to the annotation file for the challenge. This is the file uploaded by the Challenge host while creating a challenge.
- `user_annotation_file`: It represents the local path of the file submitted by the user for a particular challenge phase.
- `phase_codename`: It is the `codename` of the challenge phase from the challenge configuration yaml. This is passed as an argument so that the script can take actions according to the challenge phase.

After reading the files, some custom actions can be performed. This varies per challenge.

The `evaluate()` method also accepts keyword arguments. By default, we provide you metadata of each submission to your challenge which you can use to send notifications to your slack channel or to some other webhook service. Following is an example code showing how to get the submission metadata in your evaluation script and send a slack notification if the accuracy is more than some value `X` (X being 90 in the example given below).

```
def evaluate(test_annotation_file, user_annotation_file, phase_codename, **kwargs):

    submission_metadata = kwargs.get("submission_metadata")
    print submission_metadata

    # Do stuff here
    # Set `score` to 91 as an example

    score = 91
    if score > 90:
        slack_data = kwargs.get("submission_metadata")
        webhook_url = "Your slack webhook url comes here"
        # To know more about slack webhook, checkout this link: https://api.slack.com/
↪incoming-webhooks

        response = requests.post(
            webhook_url,
            data=json.dumps({'text': "*Flag raised for submission:* \n \n" +␣
↪str(slack_data)}),
            headers={'Content-Type': 'application/json'})

    # Do more stuff here
```

The above example can be modified and used to find if some participant team is cheating or not. There are many more ways for which you can use this metadata.

After all the processing is done, this `evaluate()` should return an output, which is used to populate the leaderboard. The output should be in the following format:

```
output = {}
output['result'] = [
            {
                'train_split': {
                    'Metric1': 123,
                    'Metric2': 123,
                    'Metric3': 123,
                    'Total': 123,
                }
            },
            {
                'test_split': {
                    'Metric1': 123,
                    'Metric2': 123,
                    'Metric3': 123,
                    'Total': 123,
                }
            }
        ]

return output
```

Let's break down what is happening in the above code snippet.

1. `output` should contain a key named `result`, which is a list containing entries per dataset split that is available for the challenge phase in consideration (in the function definition of `evaluate()` shown above, the argument: `phase_codename` will receive the *codename* for the challenge phase against which the submission was made).
2. Each entry in the list should be a dict that has a key with the corresponding dataset split codename (`train_split` and `test_split` for this example).
3. Each of these dataset split dict contains various keys (`Metric1`, `Metric2`, `Metric3`, `Total` in this example), which are then displayed as columns in the leaderboard.

A good example of a well-documented evaluation script for static code-upload challenges is My Seizure Gauge Forecasting Challenge 2022.

CHAPTER 6

Approve a challenge (for forked version)

Note: If you are hosting the challenge on eval.ai, then you cannot approve your challenge. It will be approved by EvalAI team. You can skip this section.

Once a challenge config has been uploaded, the challenge has to be approved by the EvalAI Admin (i.e. you if you are setting up EvalAI yourself on your server) to make it available to everyone. Please follow the following steps to approve a challenge (if you are):

Let's assume that we want to approve a challenge with name `Random Number Generator Challenge`.

6.1 Step 1: Approve challenge using django admin

1. Login to EvalAI's django admin panel, and you will see the list of challenges
2. Click on the challenge that you want to approve and scroll to bottom to check the following two fields.
   - Approved By Admin
   - Publically Available

   Now, save the challenge. The challenge has been successfully approved by the administrator and is also publicly visible to the users.

6.2 Step 2: Reload submission worker

Since you have just approved the challenge, the submission worker has to be reloaded so that it can fetch the evaluation script and other related files for your challenge from the database. Now reload the submission worker using the following command:

Run the following command:

```
docker-compose restart worker
```

Submission worker has been successfully reloaded!

Now, the challenge is ready to accept submissions from participants.

If you have issues in hosting a challenge on forked version of EvalAI, please feel free to create an issue on our GitHub Issues Page.

CHAPTER 7

Participate in a challenge

You have to create an account on EvalAI and a participant team in order to participate in a challenge.

If you are already familiar with the flow of EvalAI, you may want to skip this section else please follow the following steps to participate in a challenge (VQA Challenge 2017 in this example):

7.1 1. Visit eval.ai

Open EvalAI website.

7.2 2. Sign up or Log in

Sign Up and fill in your credentials or log in if you have already registered.

After signing up you would be on the dashboard page.

7.3 3. Choose challenge

Then, go to challenges section and choose an active challenge.

7.4 4. Challenge Page

After reading the challenge instructions on the challenge page, you can participate in the challenge.

7.5 5. Create Participant Team

Create a participant team if there isn't any or you can select from the existing ones.

Click on 'Participate' tab after selecting a team.

Tada! you have successfully participated in a challenge.

CHAPTER 8

Make Submission Public

Let's assume that you want to make your latest submission public in `William Challenge`.

1. Go to `My Submissions` Tab of the challenge page, select the phase and scroll horizontally.
2. Now, to make the first submission public, click on the checkbox under the column `Show on Leaderboard`. It will turn into a green checkmark.

CHAPTER 9

Make Submission Private

Let's assume that you want to make your latest submission private in `William Challenge`.

1. Go to `My Submissions` Tab of the challenge page, select the phase and scroll horizontally.
2. Now, to make the first submission private, click on the green checkmark under the column `Show on Leaderboard`. It will turn into a black checkbox.

If you face any issues, please feel free to create an issue on our GitHub Issues Page.

CHAPTER 10

Pull Request

Contributing to EvalAI is really easy. Just follow these steps to get started.

Step 1: Fork

1. Fork the EvalAI repository from the repository.

Step 2: Selecting an issue

1. Select a suitable issue that will be easy for you to fix. Moreover, you can also take the issues based on their labels. All the issues are labelled according to its difficulty.
2. After selecting an issue, ask the maintainers of the project to assign it to you and they will assign it based on its availability.
3. Once it gets assigned, create a branch from your fork's updated master branch using the following command: `git checkout -b branch_name`
4. Start working on the issue.

Step 3: Committing Your Changes

1. After making the changes, you need to add your files to your local git repository.
2. To add your files, use the following commands:

- To add only modified files, use `git add -u`
- To add a new file, use `git add file_path_from_local_git_repository`
- To add all files, use `git add .`

1. Once you have added your files, you need to commit your changes. Always **create a very meaningful commit message related to the changes that you have done. Try to write the commit message in present imperative tense. Also namespace the commit message so that it becomes self-explanatory by just looking at the commit message.** For example,

    Docs: Add verbose setup docs for ubuntu

Step 4: Creating a Pull Request

1. Before creating a Pull Request, you need to first rebase your branch with the upstream master.

2. To rebase your branch, use following commands: `git fetch upstream` `git rebase upstream/master`
3. After rebasing, push the changes to your forked repository. `git push origin branch_name`
4. After pushing the code, create a Pull Request.
5. When creating a pull request, be sure to add a comment including these keywords, and also mention any maintainer to reviewing it.

Note:

- If you have any doubts, don't hesitate to ask in the comments. You may also add in any relevant content.
- After the maintainers review your changes, fix the code as suggested. Don't forget to add, commit, and push your code to the same branch.

Once you have completed the above steps, you have successfully created a Pull Request to EvalAI.

CHAPTER 11

Contributing guidelines

Thank you for your interest in contributing to EvalAI! Here are a few pointers on how you can help.

11.1 Setting things up

To set up the development environment, follow the instructions in our README.

11.2 Finding something to work on

EvalAI's issue tracker is good place to start. If you find something that interests you, comment on the thread and we'll help get you started.

Alternatively, if you come across a new bug on the site, please file a new issue and comment if you would like to be assigned. Existing issues are tagged with one or more labels, based on the part of the website it touches, its importance etc., which can help you select one.

If neither of these seem appealing, please post on our channel and we will help find you something else to work on.

11.3 Instructions to submit code

Before you submit code, please talk to us via the issue tracker so we know you are working on it.

Our central development branch is *development*. Coding is done on feature branches based off of development and merged into it once stable and reviewed. To submit code, follow these steps:

1. Create a new branch off of development. Select a descriptive branch name.

```
git fetch upstream
git checkout master
```

(continues on next page)

(continued from previous page)

```
git merge upstream/master
git checkout -b your-branch-name
```

We highly encourage using black to format your code. It sticks to PEP8 for the most part and is in line with the rest of the repo. We have already set up pre-commit hooks to run black and flake8. To activate the hooks, you just need to run the following comamnd once:

```
pre-commit install
```

2. Commit and push code to your branch:
   - Commits should be self-contained and contain a descriptive commit message.
   - Please make sure your code is well-formatted and adheres to PEP8 conventions (for Python) and the airbnb style guide (for JavaScript). For others (Lua, prototxt etc.) please ensure that the code is well-formatted and the style consistent.
   - Please ensure that your code is well tested.

   ```
   git commit -a -m "{{commit_message}}"
   git push origin {{branch_name}}
   ```

3. Once the code is pushed, create a pull request:
   - On your GitHub fork, select your branch and click "New pull request". Select "master" as the base branch and your branch in the "compare" dropdown. If the code is mergeable (you get a message saying "Able to merge"), go ahead and create the pull request.
   - Check back after some time to see if the Travis checks have passed, if not you should click on "Details" link on your PR thread at the right of "The Travis CI build failed", which will take you to the dashboard for your PR. You will see what failed / stalled, and will need to resolve them.
   - If your checks have passed, your PR will be assigned a reviewer who will review your code and provide comments. Please address each review comment by pushing new commits to the same branch (the PR will automatically update, so you don't need to submit a new one). Once you are done, comment below each review comment marking it as "Done". Feel free to use the thread to have a discussion about comments that you don't understand completely or don't agree with.
   - Once all comments are addressed, the reviewer will give an LGTM ('looks good to me') and merge the PR.

Congratulations, you have successfully contributed to Project EvalAI!

CHAPTER 12

Architecture

EvalAI helps researchers, students, and data scientists to create, collaborate, and participate in various AI challenges organized around the globe. To achieve this, we leverage some of the best open source tools and technologies.

12.1 Django

Django is the heart of the application, which powers our backend. We use Django version 1.11.23.

12.2 Django Rest Framework

We use Django Rest Framework for writing and providing REST APIs. Its permission and serializers have helped write a maintainable codebase.

12.3 Amazon SQS

We currently use Amazon SQS for queueing submission messages which are then later on processed by a Python worker.

12.4 PostgreSQL

PostgreSQL is used as our primary datastore. All our tables currently reside in a single database named `evalai`

12.5 Angular JS

Angular JS is a well-known framework that powers our frontend.

CHAPTER 13

Architectural decisions

This is a collection of records for architecturally significant decisions.

13.1 URL Patterns

We follow a very basic, yet strong convention for URLs, so that our rest APIs are properly namespaced. First of all, we rely heavily on HTTP verbs to perform **CRUD** actions.

For example, to perform **CRUD** operation on *Challenge Host Model*, the following URL patterns will be used.

- `GET /hosts/challenge_host_team` - Retrieves a list of challenge host teams
- `POST /hosts/challenge_host_team` - Creates a new challenge host team
- `GET /hosts/challenge_host_team/<challenge_host_team_id>` - Retrieves a specific challenge host team
- `PUT /hosts/challenge_host_team/<challenge_host_team_id>` - Updates a specific challenge host team
- `PATCH /hosts/challenge_host_team/<challenge_host_team_id>` - Partially updates a specific challenge host team
- `DELETE /hosts/challenge_host_team/<challenge_host_team_id>` - Deletes a specific challenge host team

Also, we have namespaced the URL patterns on a per-app basis, so URLs for *Challenge Host Model*, which is in the *hosts* app, will be

```
/hosts/challenge_host_team
```

This way, one can easily identify where a particular API is located.

We use underscore _ in URL patterns.

13.2 Processing submission messages asynchronously

When a submission message is made, a REST API is called which saves the data related to the submission in the database. A submission involves the processing and evaluation of `input_file`. This file is used to evaluate the submission and then decide the status of the submission, whether it is *FINISHED* or *FAILED*.

One way to process the submission is to evaluate it as soon as it is made, hence blocking the participant's request. Blocking the request here means to send the response to the participant only when the submission has been made and its output is known. This would work fine if the number of the submissions made is very low, but this is not the case.

Hence we decided to process and evaluate submission message in an asynchronous manner. To process the messages this way, we need to change our architecture a bit and add a Message Framework, along with a worker so that it can process the message.

Out of all the awesome messaging frameworks available, we have chosen Amazon Simple Queue Service (SQS) because it can support decoupled environments. It allows developers to focus on application development, rather than creating their own sophisticated message-based applications. It also eliminates queuing management tasks, such as storage. SQS also works with AWS resources, so you can use it to make reliable and scalable applications on top of an AWS infrastructure.

For the worker, we went ahead with a normal python worker, which simply runs a process and loads all the required data in its memory. As soon as the worker starts, it listens on a SQS queue named `evalai_submission_queue` for new submission messages.

13.3 Submission Worker

The submission worker is responsible for processing submission messages. It listens on a queue named `evalai_submission_queue`, and on receiving a message for a submission, it processes and evaluates the submission.

One of the major design changes that we decided to implement in the submission worker was to load all the data related to the challenge in the worker's memory, instead of fetching it every time a new submission message arrives. So the worker, when starting, fetches the list of active challenges from the database and then loads it into memory by maintaining the map `EVALUATION_SCRIPTS` on challenge id. This was actually a major performance improvement.

Another major design change that we incorporated here was to dynamically import the challenge module and to load it in the map instead of invoking a new python process every time a submission message arrives. So now whenever a new message for a submission is received, we already have its corresponding challenge module being loaded in a map called `EVALUATION_SCRIPTS`, and we just need to call

```
EVALUATION_SCRIPTS[challenge_id].evaluate(*params)
```

This was again a major performance improvement, which saved us from the task of invoking and managing Python processes to evaluate submission messages. Also, invoking a new python process every time for a new submission would have been really slow.

CHAPTER 14

Directory structure

14.1 Django apps

EvalAI is a Django-based application, hence it leverages the concept of Django apps to properly namespace the functionalities. All the apps can be found in the `apps` directory situated in the root folder.

Some important apps along with their main uses are:

- **Challenges**

This app handles all the workflow related to creating, modifying, and deleting challenges.

- **Hosts**

This app is responsible for providing functionalities to the challenge hosts/organizers.

- **Participants**

This app serves users who want to take part in any challenge. It contains code for creating a Participant Team, through which they can participate in any challenge.

- **Jobs**

One of the most important apps, responsible for processing and evaluating submissions made by participants. It contains code for creating a submission, changing the visibility of the submission and populating the leaderboard for any challenge.

- **Web**

This app serves some basic functionalities like providing support for contact us or adding a new contributor to the team, etc.

- **Accounts**

As the name indicates, this app deals with storing and managing data related to user accounts.

- **Base**

A placeholder app which contains the code that is used across various other apps.

14.2 Settings

Settings are used across the backend codebase by Django to provide config values on a per-environment basis. Currently, the following settings are available:

- **dev**

Used in development environment

- **testing**

Used whenever test cases are run

- **staging**

Used on staging server

- **production**

Used on production server

14.3 URLs

The base URLs for the project are present in `evalai/urls.py`. This file includes URLs of various applications, which are also namespaced by the app name. So URLs for the `challenges` app will have its app namespace in the URL as `challenges`. This actually helps us separate our API based on the app.

14.4 Frontend

The whole codebase for the frontend resides in a folder named `frontend` in the root directory

14.5 Scripts

Scripts contain various helper scripts, utilities, python workers. It contains the following folders:

- **migration**

Contains some of the scripts which are used for one-time migration or formatting of data.

- **tools**

A folder for storing helper scripts, e.g. a script to fetch pull request

- **workers**

One of the main directories, which contains the code for submission worker. Submission worker is a normal python worker which is responsible for processing and evaluating submission of a user. The command to start a worker is:

```
python scripts/workers/submission_worker.py
```

14.6 Test Suite

All of the codebase related to testing resides in `tests` folder in the root directory. In this directory, tests are namespaced according to the app, e.g. tests for `challenges` app lives in a folder named `challenges`.

14.7 Management Commands

To perform certain actions like seeding the database, we use Django management commands. Since the management commands are common throughout the project, they are present in `base` application directory. At the moment, the only management command is `seed`, which is used to populate the database with some random values. The command can be invoked by calling

```
python manage.py seed
```

CHAPTER 15

Submission

15.1 How is a submission processed?

We are using REST APIs along with Queue based architecture to process submissions. When a participant makes a submission for a challenge, a REST API with URL pattern `jobs:challenge_submission` is called. This API does the task of creating a new entry for submission model and then publishes a message to exchange `evalai_submissions` with a routing key of `submission.*.*`.

```
User Submission  --> API  --> Publish  --> SQS Queue  --> Submission
                              message                      worker(s)
```

Exchange receives the message and then routes it to the queue `submission_task_queue`. At the end of `submission_task_queue` are workers (scripts/workers/submission_worker.py) which processes the submission message.

The worker can be run with

```
# assuming the current working directory is where manage.py lives
python scripts/workers/submission_worker.py
```

15.2 How does submission worker function?

Submission worker is a python script which mostly runs as a daemon on a production server and simply acts as a python process in a development environment. To run submission worker in a development environment:

```
python scripts/workers/submission_worker.py
```

Before a worker fully starts, it does the following actions:

- Creates a new temporary directory for storing all its data files.

- Fetches the list of active challenges from the database. Active challenges are published challenges whose start date is less than present time and end date greater than present time. It loads all the challenge evaluation scripts in a variable called `EVALUATION_SCRIPTS`, with the challenge id as its key. The maps looks like this:

```
EVALUATION_SCRIPTS = {
    <challenge_pk> : <evalutaion_script_loaded_as_module>,
    ....
}
```

- Creates a connection with SQS queue by using the AWS supplied credentials as environment variables.
- After the connection is successfully created, creates an exchange with the name `evalai_submissions` and two queues, one for processing submission message namely `submission_task_queue`, and other for getting add challenge message.
- `submission_task_queue` is then bound with the routing key of `submission.*.*` and add challenge message queue is bound with a key of `challenge.add.*` Whenever a queue is bound to a exchange with any key, it will route the message to the corresponding queue as soon as the exchange receives a message with a key.
- Binding to any queue is also accompanied with a callback which basically takes a function as an argument. This function specifies what should be done when the queue receives a message.

e.g. `submission_task_queue` is using `process_submission_callback` as a function, which means that when a message is received in the queue, `process_submission_callback` will be called with the message passed as an argument.

Expressing it informally it will be something like

> *Queue*: Hey *Exchange*, I am `submission_task_queue`. I will be listening to messages from you on binding key of `submission.*.*`
>
> *Exchange*: Hey *Queue*, Sure! When I receive a message with a routing key of `submission.*.*`, I will give it to you
>
> *Queue*: Thanks a lot.
>
> *Queue*: Hey *Worker*, Just for the record, when I receive a new message for submission, I want `process_submission_callback` to be called. Can you please make a note of it?
>
> *Worker*: Sure *Queue*, I will invoke `process_submission_callback` whenever you receive a new message.

When a worker starts, it fetches active challenges from the database and then loads all the challenge evaluation scripts in a variable called `EVALUATION_SCRIPTS`, with challenge id as its key. The map would look like

```
EVALUATION_SCRIPTS = {
    <challenge_pk> : <evalutaion_script_loaded_as_module>,
    ....
}
```

After the challenges are successfully loaded, it creates a connection with the SQS queue `evalai_submission_queue` and listens to it for new submissions.

15.3 How is submission made?

When the user makes a submission on the frontend, the following actions happen sequentially

- As soon as the user submits a submission, a REST API with the URL pattern `jobs:challenge_submission` is called.
- This API fetches the challenge and its corresponding challenge phase.
- This API then checks if the challenge is active and challenge phase is public.
- It fetches the participant team's ID and its corresponding object.
- After all these checks are complete, a submission object is saved. The saved submission object includes **participant team id** and **challenge phase id** and **username** of the participant creating it.
- At the end, a submission message is published to exchange `evalai_submissions` with a routing key of `submission.*.*`.

15.4 Format of submission messages

The format of the message is

```
{
    "challenge_id": <challenge_pk_here>,
    "phase_id": <challenge_phase_pk_here>,
    "submission_id": <submission_pk_here>
}
```

This message is published with a routing key of `submission.*.*`

15.5 How workers process submission message

Upon receiving a message from `submission_task_queue` with a binding key of `submission.*.*`, `process_submission_callback` is called. This function does the following:

- It fetches the challenge phase and submission object from the database using the challenge phase id and submission id received in the message.
- It then downloads the required files like input_file, etc. for submission in its computation directory.
- After this, the submission is run. Submission is initially marked in **RUNNING** state. The `evaluate` function of `EVALUATION_SCRIPTS` map with key of the challenge id is called. The `evaluate` function takes in the annotation file path, the user annotation file path, and the challenge phase's code name as arguments. Running a submission involves temporarily updating `stderr` and `stdout` to different locations other than standard locations. This is done so as to capture the output and any errors produced when running the submission.
- The output from the `evaluate` function is stored in a variable called `submission_output`. Currently, the only way to check for the occurrence of an error is to check if the key `result` exists in `submission_output`.
  - If the key does not exist, then the submission is marked as **FAILED**.
  - If the key exists, then the variable `submission_output` is parsed and `DataSetSplit` objects are created. LeaderBoardData objects are also created (in bulk) with the required parameters. Finally, the submission is marked as **FINISHED**.
- The value in the temporarily updated `stderr` and `stdout` are stored in files named `stderr.txt` and `stdout.txt` which are then stored in the submission instance.
- Finally, the temporary computation directory allocated for this submission is removed.

15.6 Notes

- REST API with url pattern `jobs:challenge_submission`. Here *jobs* is application namespace and *challenge_submission* is instance namespace. You can read more about url namespace

CHAPTER 16

Migrations

> Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into. - Django Migration Docs

16.1 Creating a migration

- We recommend you to create migrations per app, where the changes are only about a single issue or feature.

```
# migration only for jobs app
python manage.py makemigrations jobs
```

- Always create named migrations. You can name migrations by passing `-n` or `--name` argument

```
python manage.py makemigrations jobs -n=execution_time_limit
# or
python manage.py makemigrations jobs --name=execution_time_limit
```

- While creating migrations on local environment, don't forget to add development settings.

```
python manage.py makemigrations
```

The following is an example of a complete named migration for the `jobs` app, wherein a execution time limit field is added to the `Submission` model:

```
python manage.py makemigrations jobs --name=execution_time_limit
```

- Files create after running `makemigrations` should be committed along with other files
- While creating a migration for your concerned change, it may happen that some other changes are also there in the migration file. Like adding a `execution_time_limit` field on `Submission` model also brings in the change for `when_made_public` being added. In that case, open an new issue and clearly mention the

issue over there. If possible fix the issue yourself, by opening a new branch and creating migrations only for the concerned part. The idea here is that a commit should only include its concerned migration changes and nothing else.

CHAPTER 17

Cite

If you are using [EvalAI] for hosting challenges, please cite the following technical report:

```
@article{EvalAI,
    title   =  {EvalAI: Towards Better Evaluation Systems for AI Agents},
    author  =  {Deshraj Yadav and Rishabh Jain and Harsh Agrawal and Prithvijit
                Chattopadhyay and Taranjeet Singh and Akash Jain and Shiv Baran
                Singh and Stefan Lee and Dhruv Batra},
    year    =  {2019},
    volume  =  arXiv:1902.03570
}
```

CHAPTER 18

Frequently Asked Questions

18.1 Q. How to start contributing?

EvalAI's issue tracker is good place to start. If you find something that interests you, comment on the thread and we'll help get you started. Alternatively, if you come across a new bug on the site, please file a new issue and comment if you would like to be assigned. Existing issues are tagged with one or more labels, based on the part of the website it touches, its importance etc., which can help you select one.

18.2 Q. What are the technologies that EvalAI uses?

Please refer to Technologies Used

18.3 Q. Where could I learn GitHub Commands?

Refer to GitHub Guide.

18.4 Q. Where could I learn Markdown?

Refer to Markdown Guide.

18.5 Q. What to do when coverage decreases in your pull request?

Coverage decreases when the existing test cases don't test the new code you wrote. If you click coverage, you can see exactly which all parts aren't covered and you can write new tests to test the parts.

18.6 Q. How to setup EvalAI using virtualenv?

We have removed the documentation for setting up using virtual environment since the project has grown and different developers face different dependency issues. We recommend to setup EvalAI using docker based environment.

18.7 Common Errors during installation

18.7.1 Q. While using `pip install -r dev/requirement.txt`

```
  Writing manifest file 'pip-egg-info/psycopg2.egg-info/SOURCES.txt'
  Error: You need to install postgresql-server-dev-X.Y for building a server-side␣
↪extension or
  libpq-dev for building a client-side application.
  ----------------------------------------
  Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-build-
↪qIjU8G/psycopg2/
```

Use the following commands in order to solve the error:

1. `sudo apt-get install postgresql`
2. `sudo apt-get install python-psycopg2`
3. `sudo apt-get install libpq-dev`

18.7.2 Q. While using `pip install -r dev/requirement.txt`

```
Command "python setup.py egg_info" failed with error code 1 in
/private/var/folders/c7/b45s17816zn_b1dh3g7yzxrm0000gn/T/pip-build- GM2AG/psycopg2/
```

Firstly check that you have installed all the mentioned dependencies. Then, Upgrade the version of postgresql to 10.1 in order to solve it.

18.7.3 Q. Getting an import error

```
Couldn't import Django,"when using command python manage.py migrate
```

Firstly, check that you have activated the virtualenv. Install python dependencies using the following commands on the command line

```
cd evalai
pip install -r requirements/dev.txt
```

18.7.4 Q. Getting Mocha Error

```
Can not load reporter "mocha",it is not registered
```

Uninstall karma and then install

```
npm uninstall -g generator-karma && npm install -g generator-angular.
```

18.7.5 Q. While trying to execute `bower install`

```
bower: command not found
```

Execute the following command first :

```
npm install -g bower
```

18.7.6 Q. While trying to execute `gulp dev:runserver`

```
gulp: command not found
```

Execute the following command first

```
npm install -g gulp-cli
```

18.7.7 Q. While executing `gulp dev:runserver`

```
events.js:160
throw er; // Unhandled 'error' event
^
Error: Gem sass is not installed.
```

Execute the following command first :

```
gem install sass
```

18.7.8 Q. While trying to install `npm config set proxy http://proxy:port` on Ubuntu, I get the following error:

```
ubuntu@ubuntu-Inspiron-3521:~/Desktop/Python-2.7.14$ npm install -g angular-cli
npm ERR! Linux 4.4.0-21-generic
npm ERR! argv "/usr/bin/nodejs" "/usr/bin/npm" "install" "-g" "angular-cli"
npm ERR! node v4.2.6
npm ERR! npm  v3.5.2
npm ERR! code ECONNRESET

npm ERR! network tunneling socket could not be established, cause=getaddrinfo␣
↪ENOTFOUND proxy proxy:80
npm ERR! network This is most likely not a problem with npm itself
npm ERR! network and is related to network connectivity.
npm ERR! network In most cases you are behind a proxy or have bad network settings.
npm ERR! network
npm ERR! network If you are behind a proxy, please make sure that the
npm ERR! network 'proxy' config is set properly.  See: 'npm help config'
```

(continues on next page)

(continued from previous page)

```
npm ERR! Please include the following file with any support request:
npm ERR!     /home/ubuntu/Desktop/Python-2.7.14/npm-debug.log
```

To solve, execute the following commands:

1. `npm config set registry=registry.npmjs.org`

If the above does not work, try deleting them by following commands:

1. `npm config delete proxy`
2. `npm config delete https-proxy`

Then, start the installation process of frontend once more.

18.7.9 Q. While using docker, I am getting the following error on URL http://localhost:8888/:

```
Cannot Get \
```

Try removing the docker containers and then building them again.

18.7.10 Q. Getting the following error while running `python manage.py seed`:

```
Starting the database seeder. Hang on... Exception while running run() in 'scripts.
↪seed' Database successfully seeded
```

Change the python version to 2.7.x . The problem might be because of the python 3.0 version.

18.7.11 Q. Getting the following error while executing command `createdb evalai -U postgres`:

```
createdb: could not connect to database template1: FATAL: Peer authentication failed␣
↪for user "postgres"
```

Try creating a new user and then grant all the privileges to it and then create a db.

18.7.12 Q. Getting the following error while executing `npm install`:

```
npm WARN generator-angular@0.16.0 requires a peer of generator-
karma@>=0.9.0 but none was installed.
```

Uninstall and then install karma again and also don't forget to clean the global as well as project npm cache. Then try again the step 8.

18.7.13 Q. While running the unit tests, I am getting the error similar to as shown below:

```
_____________________ ERROR collecting tests/unit/web/test_views.py _____________________
import file mismatch:
imported module 'tests.unit.web.test_views' has this __file__ attribute:
  /path/to/evalai/tests/unit/web/test_views.py
which is not the same as the test file we want to collect:
  /code/tests/unit/web/test_views.py
HINT: remove __pycache__ / .pyc files and/or use a unique basename for your test file␣
↪modules
```

It appears that you are trying to run `pytest` in a docker container. To fix this, delete the `__pycache__` and all `*.pyc` files using the following command:

```
find . | grep -E "(__pycache__|\.pyc|\.pyo$)" | xargs rm -rf
```

18.7.14 Q. Getting the following error:

```
ERROR: for db Cannot start service db: driver failed programming external␣
↪connectivity on endpoint evalai_db_1␣
↪(2163096de9aac6561b4f699bb1049acd0ce881fbaa0da28e47cfa9ca0ee1199f): Error starting␣
↪userland proxy: listen tcp 0.0.0.0:5432: bind: address already in use
```

The following solution only works on Linux.

Execute : `sudo netstat -lpn |grep :5432`

The output of the above would be in the following form:

```
tcp 0 0 127.0.0.1:5432 0.0.0.0:* LISTEN 25273/postgres
```

Execute the following command:

```
sudo kill 25273 ## This would vary and you can change with the output in the first␣
↪step
```

18.7.15 Q. Getting the following error when using Docker:

```
ERROR : Version in "./docker-compose.yml" is unsupported. You might be seeing this␣
↪error because you are using wrong Compose file version.
```

Since, the version of compose file is 3. You might be using a docker version which is not compatible. You can upgrade your docker engine and try again.

18.7.16 Q. Getting the following error while running `python manage.py runserver --settings=settings.dev`:

```
Starting the database seeder. Hang on...
Are you sure you want to wipe the existing development database and reseed it? (Y/N)
Exception while running run() in 'scripts.seed'
```

Try clearing the postgres database manually and try again.

18.7.17 Q. Getting the following error while executing `gulp dev:runserver`:

```
/usr/lib//nodejs/gulp//bin/gulp.js:132
        gulpInst.start.apply(gulpInst, toRun)l
                                             ^
TypeError: Cannot read properly 'apply of undefined'
```

Execute the following command:

```
rm -rf node_modules/
rm -rf bower_components
npm install
bower install
```

18.7.18 Q. While trying to build EvalAI from the master branch and run the command docker-compose up:

```
ERROR: Service 'celery' failed to build: pull access denied for evalai_django,␣
↪repository does not exist or may require 'docker login': denied: requested access␣
↪to the resource is denied
```

Please make sure to clone EvalAI in its default directory with name evalai. This happens because the parent directory changes the name of docker images. For instance, the image evalai_django gets renamed to evalai_dev_django if your directory is renamed to EvalAI_dev.

CHAPTER 19

Glossary

19.1 Challenge

An event, run by an institute or organization, wherein a number of researchers, students, and data scientists participate and compete with each other over a period of time. Each challenge has a start time and generally an end time too.

19.2 Challenge host

A member of the host team who organizes a challenge. In our system, it is a form of representing a user. This user can be in the organizing team of many challenges, and hence for each challenge, its challenge host will be different.

19.3 Challenge host team

A group of challenge hosts who organizes a challenge. They are identified by a unique team name.

19.4 Challenge phase

A challenge phase represents a distinct stage of the challenge. Over a period of time, challenge organizers have started to use the challenge phase as a way to:

1. Decide when to evaluate submissions on a subset of the test-set or when to evaluate on the whole test-set (for e.g, VQA Challenge 2019)
2. Use different challenge phases as different tracks of the same challenge (for e.g., CARLA Autonomous Driving Challenge)

19.5 Challenge phase split

A challenge phase split is the relation between a challenge phase and dataset splits for a challenge with a many-to-many relation. This is used to set the privacy of submissions (public/private) to different dataset splits for different challenge phases.

19.6 Dataset

A dataset in EvalAI is the main entity in which an AI challenge is based on. Participants are expected to make submissions corresponding to different splits of the corresponding dataset.

19.7 Dataset split

A dataset is generally divided into different parts called dataset split. Generally, a dataset has three different splits:

- Training set
- Validation set
- Test set

19.8 EvalAI

EvalAI is an open-source web platform that aims to be the state of the art in AI. Its goal is to help AI researchers, practitioners, and students to host, collaborate, and participate in AI challenges organized around the globe.

19.9 Leaderboard

The leaderboard can be defined as a scoreboard listing the names of the teams along with their current scores. Currently, each challenge has its own leaderboard.

19.10 Phase

A challenge can be divided into many phases (or challenge phases). A challenge phase can have the same or different start and end date than the challenge start and end date.

19.11 Participant

A member of the team competing against other teams for any particular challenge. It is a form of representing a user. A user can participate in many challenges, hence for each challenge, its participant entry will be different.

19.12 Participant team

A group of one or more participants who are taking part in a challenge. They are identified uniquely by a team name.

19.13 Submission

A way of submitting your results to the platform, so that it can be evaluated and ranked amongst others. A submission can be public or private, depending on the challenge.

19.14 Submission worker

A python script which processes submission messages received from a queue. It does the heavy lifting task of receiving a submission, performing mandatory checks, and then evaluating the submission and updating its status in the database.

19.15 Team

A model, present in `web` app, which helps CloudCV register new contributors as a core team member or simply an open source contributor.

19.16 Test annotation file

This is generally a file uploaded by a challenge host and is associated with a challenge phase. This file is used for ranking the submission made by a participant. An annotation file can be shared by more than one challenge phase. In the codebase, this is present as a file field attached to challenge phase model.

CHAPTER 20

Indices and tables

- genindex
- modindex
- search