STADLE

Release 0.0.4

Kiyoshi Nakayama and Anthony Maddalone

Jan 04, 2023

CONTENTS

STADLE documentation is divided into several sections on the left sidebar. If you're unable to find what you're looking for, please let us know! Further information on STADLE and its features can be found on our website. If this is your first time here, you may want to start with the Overview section.

CHAPTER

ONE

TABLE OF CONTENTS:

1.1 Overview

1.1.1 Introduction to STADLE

Our STADLE is a paradigm-shifting intelligence centric platform for federated, collaborative, and continuous learning. STADLE stands for Scalable Traceable Adaptive Distributed Learning platform.

In particular, Federated learning, the core capability of STADLE, continuously collects Machine Learning models from distributed devices or clients, aggregates the collective intelligence, and brings it back to the local devices. Therefore, Federated Learning (FL) using our STADLE solves the fundamental problems that commonly appear in ML systems such as

- **Privacy**: FL does not require users to upload raw data to cloud servers, thus it improves the privacy-preserving aspect of AI systems by not collecting personal or sensitive data in the cloud.
- **Learning Efficiency**: Training a huge amount of data in a centralized manner is really inefficient sometimes. With distributed learning framework life federated learning, you can efficiently utilize any distributed resources to make learning faster and more inexpensive.
- **Real-Time Intelligence**: It often takes a lot of time to deliver desired intelligence, and sometimes all the value is gone by the time it is delivered. FL usually does not have to require a huge data pipeline, computational resources, and storages to deliver high-quality intelligence.
- **Communication Load**: The amount of traffic generated by FL dramatically decreases from classical AI systems due to the difference in data type exchanged.
- **Low Latency**: The delay in communication to obtain collective intelligence can be dramatically reduced by employing decentralized FL servers located at edge servers.

Our STADLE platform is horizontally designed and enhances the capability of FL.

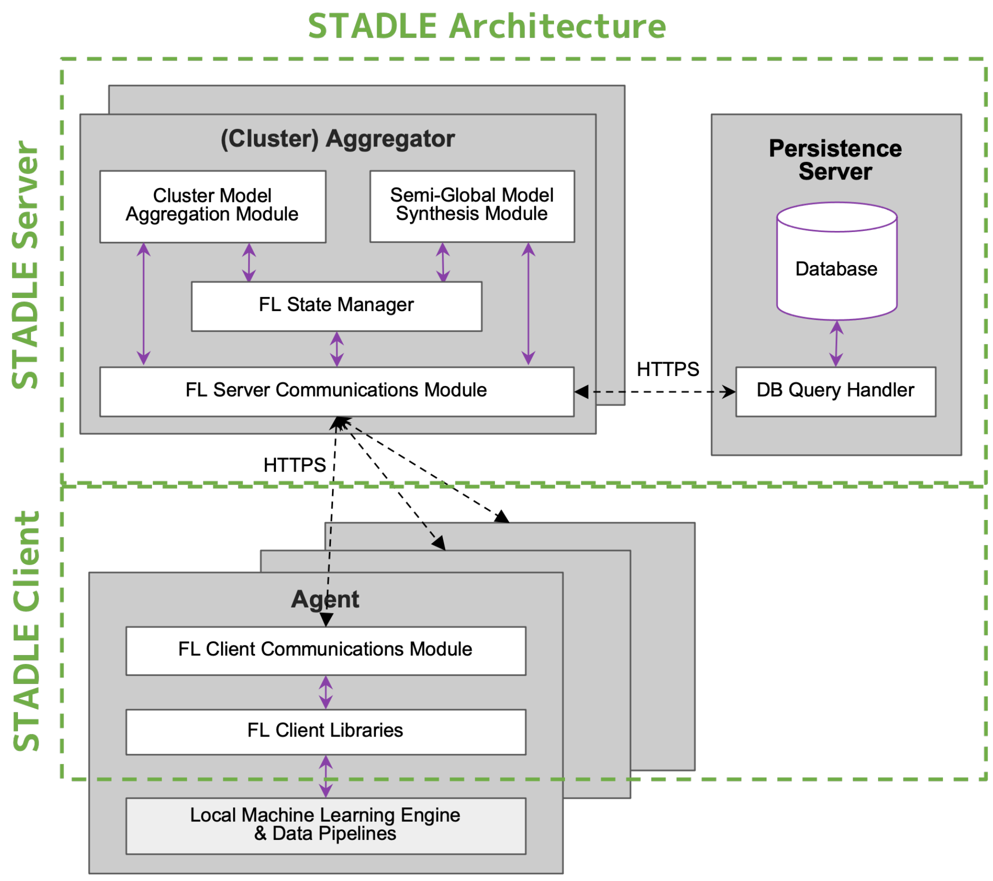
- **Scalability**: Decentralized FL servers in STADLE realize the load-balancing to accommodate more users and create semi-global models which do not require a central master aggregator node.
- **Traceability**: Our platform has the performance tracking capability that monitors and manages the transition of collective intelligence models in the decentralized system.
- **Usability**: Our SaaS platform (stadle.ai) provides GUI for users to keep track of the distributed learning process as well as manage and control the aggregators and agents intuitively.
- **Resiliency**: In the federated learning process, systems often fail and are disconnected from each other. The STADLE ensures the continuous learning that is critical to the operation in real settings for many applications.
- **Adaptability**: Static intelligence produced by big data systems gets outdated and underperforms easily while the adaptive intelligence is created by a well designed distributed resilient learning framework that perfectly sync up the local and global models.

1.1.2 STADLE Architecture

There are 3 main components in STADLE.

- Persistence Server
  - A core functionality that helps keeping track of various database entries and ML models.
  - Packaged as a command inside the *stadle* library that can be used as *stadle persistence-server [args]* in a terminal.
- (Cluster) Aggregator
  - A core FL server functionality that accepts ML models from distributed agents and conducts aggregation processes.
  - Packaged as a command inside the *stadle* library that can be used as *stadle aggregator [args]* in a terminal.
- (Distributed) Agent
  - A core functionality and libraries that help integrating the local ML engine and/or models into the STADLE platform.
  - In charge of communicating with *stadle* core functions.
  - Packaged inside the *stadle-client* library as a class that can be used as *from stadle import BasicClient / IntegratedClient* in the local ML code.
  - *class BasicClient / IntegratedClient* is used to integrate training, testing, validation functions of the local ML process.

All those components are connected using HTTPS and exchange machine learning models with each other.


STADLE Architecture
STADLE Server
STADLE Client
(Cluster) Aggregator
Cluster Model Aggregation Module
Semi-Global Model Synthesis Module
FL State Manager
FL Server Communications Module
Persistence Server
Database
DB Query Handler
HTTPS
HTTPS
Agent
FL Client Communications Module
FL Client Libraries
Local Machine Learning Engine & Data Pipelines

1.1.3 Initial Base Model Upload Process

The first step of running a federated learning process is to register the initial ML model which we call a *Base Model*. The architecture of the base model will be used in the entire process of FL by all the aggregators and agents. We call the agent that uploads the initial base model an *admin agent*. The base model info could include the ML model itself as well as the model type such as PyTorch, the time it was generated, the initial performance data, etc. The base model can be also used as the very first semi-global model (SG model) to be downloaded by the other agents. This process can happen just once unless you want to start a new federated learning process from the beginning.

1.1.4 Federated Learning Cycle with STADLE

Figure below is the federated learning cycle showing the overall process of how federated learning is continuously conducted between an aggregator and an agent. Here it only describes a single-agent case, but in real case and operation, there are many agents dispersed into distributed devices.

The agents other than the admin agent will request the global model that is an updated federated ML model in order to train it locally with local data or deploy it to its own application.

Once the agent gets the updated model from the aggregator and deploys it, the agent basically procees to “training” to retrain the ML model locally with new data that is obtained afterwards. Again, these local data will not be shared with the aggregator and stay local within the distributed devices.

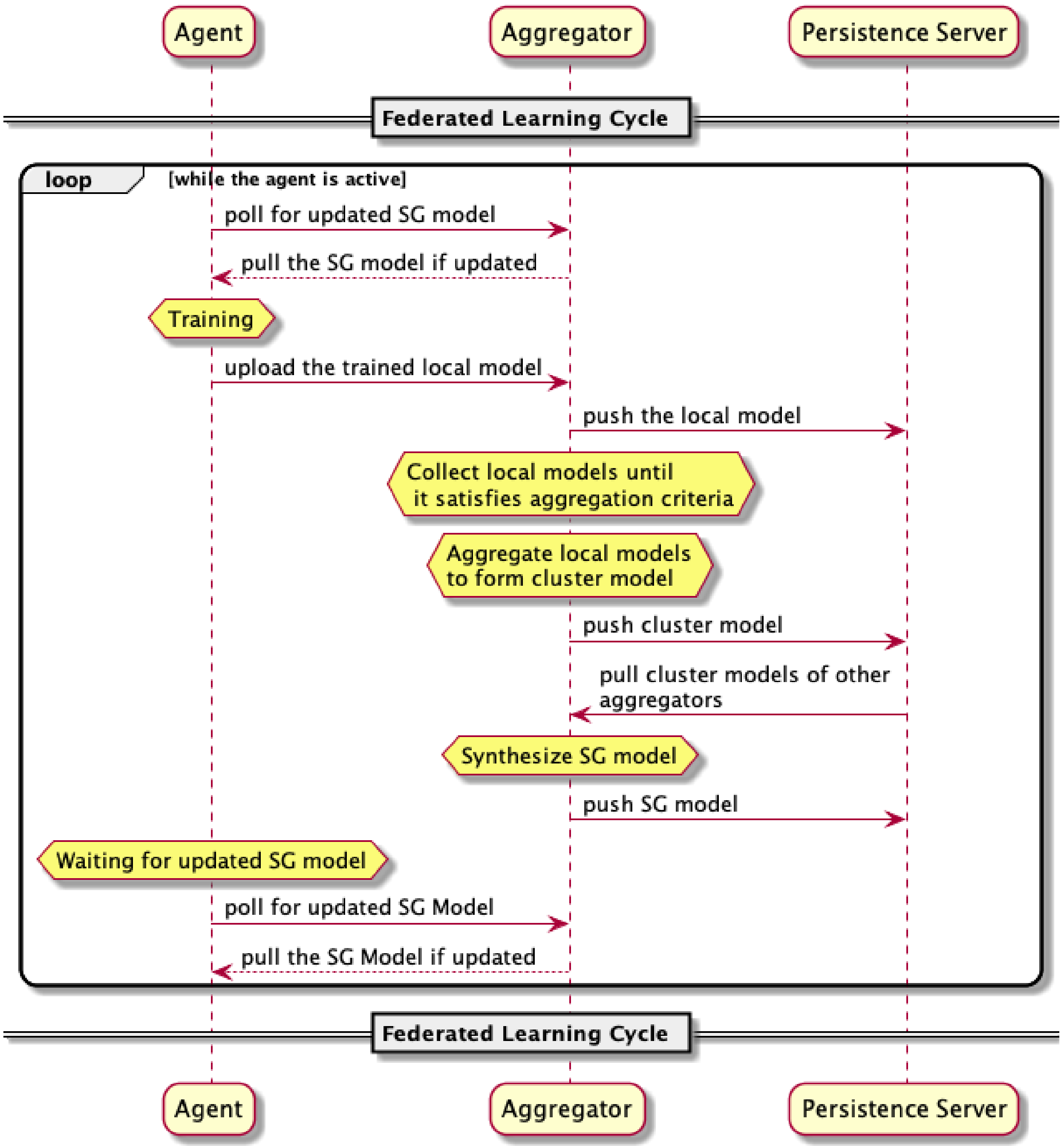
After training the local ML model (that of course has the same architecture as the global/base model of the federated learning), the agent calls FL client API to send the model to the aggregator.

Aggregator receives the model and pushes the model to the database. The aggregator keeps track of the number of collected ML models and it will keep accepting the local ML models as long as the federation round is open. The round can be closed with any defined criteria such as the aggregator receiving enough ML models to be federated. When the criteria are met, the aggregator aggregates the local ML models and produces an updated cluster global model.

Then, the aggregator starts to collect other cluster models formed by other aggregators to synthesize a semi-global model (SG model), and the SG model is the one that is sent back to agents. If there is only one aggregator, the SG model is going to be the same as the cluster model formed by the aggregator.

During that process above, agents constantly keep polling to the aggregator if the SG model is realy or not. Then, the updated SG model is sent back to the agent.

After receiving the updated SG model, the agent deploys and retrains it whenever that is ready and repeats this process until the termination criteria are met for the federated learning. In many cases, there are no termination conditions to stop this federated learning and retraining process.


Agent
Aggregator
Persistence Server
Federated Learning Cycle
loop
[while the agent is active]
poll for updated SG model
pull the SG model if updated
Training
upload the trained local model
push the local model
Collect local models until it satisfies aggregation criteria
Aggregate local models to form cluster model
push cluster model
pull cluster models of other aggregators
Synthesize SG model
push SG model
Waiting for updated SG model
poll for updated SG Model
pull the SG Model if updated
Federated Learning Cycle
Agent
Aggregator
Persistence Server

1.1.5 Client-Side Local Training Cycle

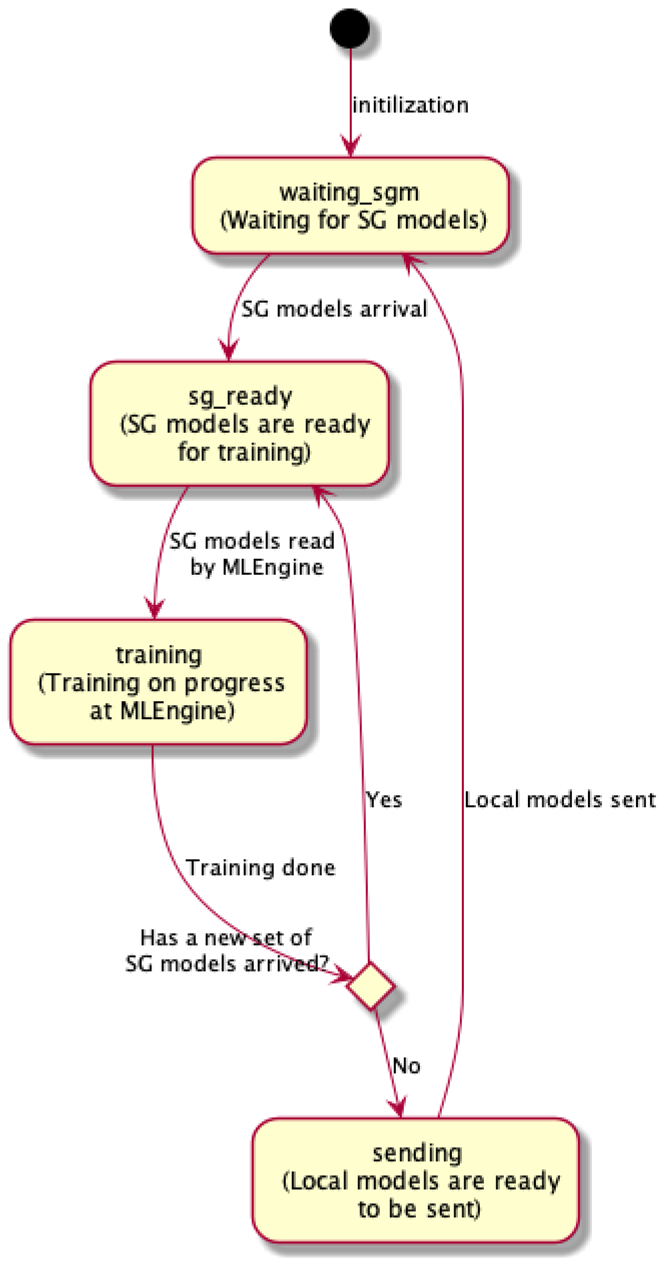
It may be helpful to understand the FL client states when integrating STADLE to your ML applications. Figure below is the state transition of an agnet for local ML training.

(1) While an agent is waiting for the SG model (*waiting_sgm* state), the agent queries the aggregator for updates to the global model (a.k.a. ML model exchanged between the aggregator and agent). Basically, a polling method is used to query the updated global model every seconds/minutes/hours/days.

(2) If the SG model is available (*sg_ready* state), the agent downloads the synthesized SG model that has been updated by the aggregator. These parameters of the SG models can be merged with the local ML model that is to be trained. Before the agent feeds the downloaded SG model to its ML model, the agent can calculate an output and store the new input and the feedback from the process.

(3) The agent can proceed with the local (re)training process (*training* state). After the training is done, if the agent has received a new updated SG model, it basically throw away the retrained model and use the new SG model for retraining. In this case, the agent goes back *sg_ready* state.

(5) Updates made to the ML model is cached so it can be sent to the aggregator when local training is done. Then, the agent sends its updated local ML model to an aggregator by setting the agent state as *sending*.



Ready to get started? Great! Click here for *Quickstart*.

1.2 Quickstart

Starting using the STADLE platform is as easy as just following the 3 steps below.

1.2.1 STEP 1: Set up STADLE Server

Just go to the stadle.ai and sign up for a free account. Then, login to your account, create a new project, and initiate an aggregator. Just wait for a few seconds to get the aggregator running.

Please also follow the step-by-step instructions on the *Start Yout Project* page in *User Guide*.

Note: With your free account, you will be able to create one project and initiate one aggregator. The number of agents that can be connected to the aggregator will be also limited.

1.2.2 STEP 2: Install STADLE Client

The STADLE client can be installed using the following commands.

First upgrade the pip.

```
pip install --upgrade pip
```

Then, install the *stadle-client*.

```
pip install stadle-client
```

Note: The environment needs to be *Python 3.8.0* or newer.

1.2.3 STEP 3: Run Local STADLE Example Codes

The next phase is to utilize STADLE libraries by importing them in your local ML processes. You can quickly run and test some of the sample applications from STADLE examples in which some of the key STADLE client libraries are already connected. To do so, just download the local example codes from the repo.

```
git clone https://github.com/tie-set/stadle_examples.git
```

After downloading the sample codes, just follow the instruction of the README on how to run those applications with the STADLE client-side APIs.

For example, to run the minimal example using PyTorch, just go to the *"stadle_examples/minimal_examples/pytorch"* folder. Then, modify the config files for both admin and ML agents. In particular, *"aggr_ip"* should be the "IP Address" and *"reg_port"* should be the "Port to Connect", both shown on the STADLE dashboard.

Then, run the admin agent with the following command (In this case, the script is named minimal_admin_agent.py). The admin agent uploads the base model that defines the architecture of the ML models that will be aggregated.

```
python minimal_admin_agent.py
```

Then, go to the STADLE dashboard and update the page after a few seconds. You can check the name of the uploaded base model on the dashboard. You can run multiple agents with different agent names. In this case, the name of the local ML process script is “minimal_fl_agent.py”. For example, you can run the ML agents like

```
python minimal_fl_agent.py --agent_name AGEMT_NAME_01
python minimal_fl_agent.py --agent_name AGEMT_NAME_02
```

On the STADLE dashboard, you will see the number of connected ML agents and downloadable recent global and local models as well as the best performing model.

You have successfully completed all the steps to conduct a STADLE project properly. The aggregation process for each round can be checked and managed on Aggregation Management page. Also, the configuration information and system status of aggregators and agents can be checked on Config Info & Settings page.

You will also be able to download the recent global, local, best performance models as well as keep track of the performance of ML models on the STADLE dashboard.

1.3 Usage

This section covers the requisite steps for integrating STADLE with a basic deep learning training process. Please refer to *Quickstart* to set up the client environment to connect to STADLE. Also, please download the sample codes from here in which the STADLE libraries are already integrated.

1.3.1 STADLE Aggregator Functionalities

The STADLE aggregators can be configured through the stadle.ai dashboard as explained in the Quickstart and its User Guide.

Creating Project

Once you create your own account, the first thing you will be doing is to create a new project on Overview page. In one project, you will be able to assign an AI model such as VGG16. If you would like to federate many AI models, you will have to create multiple projects for each AI model to be aggregated as the architecture of the AI model needs to be consistent among all the agents that are connected to the aggregator.

Note: With your free account, you will be able to create only one project.

Initiating Aggregator

Once your create a project, you will be able to initiate aggregator(s) on Overview page. If you would like to set up decentralized aggregation with multiple aggregators, you can initiate multiple aggregator instances within one project so that semi-global model will be created.

Note: With your free account, you will be able to initiate only one aggregator.

Downloading Models

You will be able to download the most recent global ML models as well as the most recent local models and bestperformance models on the STADLE dashboard.

Completing Current Round

This fanctionality provides the ability to wrap up the current round of aggregation. An aggregator needs to collect the certain number of ML models in order to proceed with the aggregation process. However, you can force the aggregation to happen even if there are not enough local models collected from agents by executing "Complete Current Round" functionality.

Aggregation Threshold

This specifies how much local models need to be collected over the active agents connected to the aggregator. For example, if the "Aggergation Threshold" is 0.7, we need 70% of local models from the active agents.

Agent Timeout

This feature provides the time out functionality that disconnects active agents if the aggregator has not heard from the agents after the seconds specified by the user. For example, if the timeout value is 30 and an agent is stopped or disconnected from the network for 30 seconds, the aggregator sets this agent's status as TIMEOUT. If the agent's status becomes TIMEOUT, this agent is not counted as an active agent and excluded from the aggregation process unless it is connected to the aggregator again. If the timeout value is 0, then this agent timeout functionality itself is disabled.

Aggregation Method Selection

While FedAvg is used as a default aggregation method as a powerful approach for many applications, you can pick up the most suitable aggregation method for your ML application. The aggregation methods that are currently supported include FedAvg, Geometric Median, Coordinate-Wise Median, Krum, and Krum Averaging.

Synthesize Semi-Global Models

STADLE supports decentralized architecture of aggregators where multiple aggregators can be set up to synthesize the another layer of global models, which we call Semi-Global Models (SG Models). Semi-Global Models are STADLE's powerful approach to create the global model in a decentralized manner so that you can scale the federated learning horizontally.

Aggregation Management

On the Aggregation Management page, you will be able to check the Current Round, the Maximum Number of Connectable Active Agents, the Number of Active Agents Participating, the Number of Local Models Needed for Aggregation, and the Number of Local Models Collected.

Performance Tracking

Performance of the uploaded local ML models for each aggregation round can be tracked on the Dashboard as well as Performance Tracking page. You can monitor the learning process for each metrics of your ML models there.

Stopping & Restarting aggregators

You can stop/restart aggregators on the Config Info & Settings page. The aggregator status then becomes "INACTIVE" or "ACTIVE" after successfully stoping/restarting the aggregators, respectively.

1.3.2 Client-side STADLE Integration

This section will cover the process of integrating STADLE with existing PyTorch code used to train a CNN on the CIFAR-10 dataset.

Local Training Code

The following is a breakdown of the PyTorch code serving as the example DL process:

```python
import sys

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

from vgg import VGG
```

This section imports sys and the requisite PyTorch libraries for future use. In addition, a predefined VGG model is imported from the model definition file.

```python
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

trainset = torchvision.datasets.CIFAR10(
    root='data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=64, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(
    root='data', train=False, download=True, transform=transform_test)
```

(continues on next page)

(continued from previous page)

```python
testloader = torch.utils.data.DataLoader(
    testset, batch_size=64, shuffle=False, num_workers=2)
```

This section loads in the CIFAR-10 dataset (downloading it if necessary) and applies the transforms to each image to help augment the dataset for robust training.

```python
device = 'cuda'

num_epochs = 200
lr = 0.001
momentum = 0.9

model = VGG('VGG16').to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=lr,
                      momentum=momentum, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
```

This section sets the device to perform training on (GPU in this case) and fixes some training-specific parameters. It then creates the initial model object and the PyTorch objects used to optimize the model parameters during the training process.

```python
for epoch in range(num_epochs):
    print('\nEpoch: %d' % (epoch + 1))

    model.train()
    train_loss = 0
    correct = 0
    total = 0

    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        loss.backward()
        optimizer.step()

        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

        sys.stdout.write('\r'+f"\rEpoch Accuracy: {(100*correct/total):.2f}%")
    print('\n')

    if ((epoch + 0) % 5 == 0):
        model.eval()
        test_loss = 0
        correct = 0
```

(continues on next page)

(continued from previous page)

```
        total = 0

        with torch.no_grad():
            for batch_idx, (inputs, targets) in enumerate(testloader):
                inputs, targets = inputs.to(device), targets.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, targets)

                test_loss += loss.item()
                _, predicted = outputs.max(1)
                total += targets.size(0)
                correct += predicted.eq(targets).sum().item()

        acc = 100.*correct/total
        print(f"Accuracy on val set: {acc}%")
```

Finally, this section handles the actual training of the model. Training on the train dataset occurs every epoch, and validation set accuracy is computed every five epochs.

In summary, this code trains the VGG-16 model on the CIFAR-10 dataset for 200 epochs.

Integration with BasicClient

In STADLE, the purpose of a client is to act as an interface between the model training being done locally and the FL process managed by STADLE's other components. `BasicClient` is an implementation of the STADLE client, intended for cases where maximal control of the FL process or minimal integration are desired.

The process of integrating with STADLE using the BasicClient can be broken down into four steps:

1. Create and properly configure the BasicClient object
2. Connect the BasicClient to STADLE (via an aggregator)
3. Modify the training loop to send a model to STADLE after some period of local training and to wait to receive the aggregated model as a checkpoint to resume local training.
4. Disconnect from STADLE when training is complete

The CIFAR-10 example will be used to show how these steps can be implemented.

Step 1: Create/Configure BasicClient

First, BasicClient has to be imported from the `stadle` library; this is done with

```
from stadle import BasicClient
```

The BasicClient object can then be created. The configuration information of the BasicClient can be set by passing a config file path through the constructor. Refer to *Config File Documentation* for details on the config file parameters.

```
client_config_path = r"/path/to/config/file.json"
stadle_client = BasicClient(config_file=client_config_path)
```

Alternatively, specific config parameter values can be set directly with the BasicClient constructor. Information on the config file and these parameters, as well as all subsequent function calls, can be found at *Client API Documentation*.

Step 2: Connect BasicClient to STADLE

The connection between the BasicClient and the aggregator it is configured to connect to can then be opened with

```
stadle_client.connect(model)
```

Note that we pass the recently-intialized model (in this case, the VGG model) to the client for use as a container for the aggregated parameters received each round.

Step 3: Modify Training Loop

The local training code previously shown trains the VGG model for 200 epochs. In order to apply federated learning to this training process, these 200 epochs must be broken into numerous short local training periods. For this example, these local training periods will be two epochs long; thus, 100 aggregation rounds of two epochs each will be run.

After one such training period, all of the CIFAR-10 "agents" connected to an aggregator send their locally-trained models to the aggregator, waiting to receive the aggregated model before starting the next training period with the received model. The following shows an example of how this can be done within the main training loop of the local training code:

```
for epoch in range(num_epochs):
    print('\nEpoch: %d' % (epoch + 1))

    """
    Addition for STADLE integration
    """
    if (epoch % 2 == 0):
        # Don't send model at beginning of training

    if (epoch != 0):
        stadle_client.send_trained_model(agent.target_net)

    sg_model_dict = stadle_client.wait_for_sg_model()

    model.load_state_dict(sg_model_dict)

    model.train()
    train_loss = 0
    correct = 0
    total = 0

    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        loss.backward()
        optimizer.step()

        _, predicted = outputs.max(1)
        total += targets.size(0)
```

(continues on next page)

(continued from previous page)

```
        correct += predicted.eq(targets).sum().item()

        sys.stdout.write('\r'+f"\rEpoch Accuracy: {(100*correct/total):.2f}%")
    print('\n')
```

Step 4: Disconnect from STADLE

Finally, the BasicClient can be disconnected with

```
stadle_client.disconnect()
```

once all training rounds have completed or some other condition has been met.

Integration with IntegratedClient

Using the `IntegratedClient` allows for the management of the local training process to be passed to STADLE, as opposed to the more hands-off approach taken by the BasicClient. As a result, the integration process to be able to use the IntegratedClient is slightly more in-depth.

This process can be broken down into x steps:

1. Create and properly configure the IntegratedClient object
2. Construct a training, cross-validation, and test function (segmentation of the local training process) and pass the functions to the IntegratedClient
3. Construct a termination function to determine when to stop the FL process
4. Connect the IntegratedClient to STADLE and start the entire FL process

Similarly to the BasicClient, the CIFAR-10 example will be used to show how these steps can be implemented.

Step 1: Create/Configure IntegratedClient

IntegratedClient is imported from the `stadle` library; this is done with

```
from stadle import IntegratedClient
```

The BasicClient object can then be created and configured like the BasicClient:

```
client_config_path = r"/path/to/config/file.json"
stadle_client = IntegratedClient(config_file=client_config_path)
```

Alternatively, specific config parameter values can be set directly with the IntegratedClient constructor. Information on the config file and these parameters, as well as all subsequent function calls, can be found at *Client API Documentation*.

Step 2: Construct Local Training Functions

When STADLE manages the local training part of the FL process, it works with abstracted versions of the training, cross-validation, and test functions. As a result, any specific implementations of these functions must match these abstractions in format. The following are template implementations of the functions in question:

Train Function:

```python
def train(model, data, **kwargs):
    # Use data to locally train model
    # kwargs used to pass general parameters to function

    return locally_trained_model, average_training_loss
```

Cross-Validation Function:

```python
def cross_validate(model, data, **kwargs):
    # Use data to compute accuracy or other performance metric (validation set)
    # kwargs used to pass general parameters to function

    return acc, ave_loss
```

Test Function:

```python
def test(model, data, **kwargs):
    # Use data to compute accuracy or other performance metric (test set)
    # kwargs used to pass general parameters to function

    return acc, ave_loss
```

The IntegratedClient will go through the following steps to fulfill the agent-side role in FL:

1. Check termination function output, continue if false
2. Receive previous round aggregated model from aggregator
3. Run cross_validate function on aggregated model
4. Run train function to train model locally
5. Run cross_validate function on locally-trained model
6. Send locally-trained model to aggregator

The CIFAR-10 local training example code can then be segmented into these functions in the following way:

Train Function (CIFAR-10):

```python
def train(model, data, **kwargs):
    lr = float(kwargs.get("lr")) if kwargs.get("lr") else 0.001
    momentum = float(kwargs.get("momentum")) if kwargs.get("momentum") else 0.9
    epochs = int(kwargs.get("epochs")) if kwargs.get("epochs") else 2
    device = kwargs.get("device") if kwargs.get("device") else 'cpu'

    model = model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr,
```

(continues on next page)

(continued from previous page)

```
                                    momentum=momentum, weight_decay=5e-4)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)

    ave_loss = []

    for epoch in range(epochs):  # loop over the dataset multiple times

        print('\nEpoch: %d' % (epoch + 1))

        model.train()
        train_loss = 0
        correct = 0
        total = 0
        for batch_idx, (inputs, targets) in enumerate(trainloader):
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            ave_loss.append(train_loss)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    ave_loss = sum(ave_loss) / len(ave_loss)

    model = model.to('cpu')

    return model, ave_loss
```

Cross-Validation Function (CIFAR-10):

```
def cross_validate(test_model, data, **kwargs):
    device = kwargs.get("device") if kwargs.get("device") else 'cpu'

    test_model = test_model.to(device)

    correct = 0
    total = 0
    overall_accuracy = 0

    with torch.no_grad():
        for (inputs, targets) in data:
            inputs, targets = inputs.to(device), targets.to(device)
            # calculate outputs by running images through the network
            outputs = test_model(inputs)
            # the class with the highest energy is what we choose as prediction
```

(continues on next page)

(continued from previous page)

```python
            _, predicted = torch.max(outputs.data, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()

    overall_accuracy = 100 * correct / total
    print('Accuracy of the network on the 10000 test images: %d %%' % (overall_accuracy))

    # prepare to count predictions for each class
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    with torch.no_grad():
        for (inputs, targets) in data:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = test_model(inputs)
            _, predictions = torch.max(outputs, 1)
            # collect the correct predictions for each class
            for target, prediction in zip(targets, predictions):
                if prediction == target:
                    correct_pred[classes[target]] += 1
                total_pred[classes[target]] += 1

    # print accuracy for each class
    # Capture average accuracy across all classes
    for classname, correct_count in correct_pred.items():
        accuracy = 100 * float(correct_count) / total_pred[classname]
        print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                              accuracy))
    return overall_accuracy, 0
```

We can use the same implementation for the test function in this case, simply changing the dataset passed to the function.

Test Function (CIFAR-10):

```python
def test(test_model, data, **kwargs):
    device = kwargs.get("device") if kwargs.get("device") else 'cpu'

    test_model = test_model.to(device)

    correct = 0
    total = 0
    overall_accuracy = 0

    with torch.no_grad():
        for (inputs, targets) in data:
            inputs, targets = inputs.to(device), targets.to(device)
            # calculate outputs by running images through the network
            outputs = test_model(inputs)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()

```

(continues on next page)

(continued from previous page)

```
    overall_accuracy = 100 * correct / total
    print('Accuracy of the network on the 10000 test images: %d %%' % (overall_accuracy))

    # prepare to count predictions for each class
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    with torch.no_grad():
        for (inputs, targets) in data:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = test_model(inputs)
            _, predictions = torch.max(outputs, 1)
            # collect the correct predictions for each class
            for target, prediction in zip(targets, predictions):
                if prediction == target:
                    correct_pred[classes[target]] += 1
                total_pred[classes[target]] += 1

    # print accuracy for each class
    # Capture average accuracy across all classes
    for classname, correct_count in correct_pred.items():
        accuracy = 100 * float(correct_count) / total_pred[classname]
        print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                              accuracy))
    return overall_accuracy, 0
```

Step 3: Construct Termination Function

The termination function is a user-defined function that controls when an agent exits a FL process. The function is run by the agent at the beginning of each round, and the agent exits if the function retuns True.

One simple termination function is to return True after a certain number of rounds has passed; the following is an implementation of such a function:

```
def judge_termination(**kwargs) -> bool:
    """
    Decide if it finishes training process and exits from FL platform
    :param training_count: int - the number of training done
    :param sg_arrival_count: int - the number of times it received SG models
    :return: bool - True if it continues the training loop; False if it stops
    """

    keep_running = True
    client = kwargs.get('client')
    current_fl_round = client.federated_training_round

    if current_fl_round >= int(kwargs.get("round_to_exit")):
        keep_running = False
        client.stop_model_exchange_routine()
    return keep_running
```

Step 4: Setup, Connect IntegratedClient to STADLE

The following is example code to set up the IntegratedClient with the previously defined functions and start the FL process:

```
parser = argparse.ArgumentParser(description='STADLE CIFAR10 Training')
parser.add_argument('--lr', default=0.1, type=float, help='learning rate')
parser.add_argument('--lt_epochs', default=3)

args = parser.parse_args()

device = 'cuda'

model = VGG('VGG16')
```

Read in learning rate and number of local training epochs from command line arguments, set training device and define model to be trained.

```
trainset = torchvision.datasets.CIFAR10(
        root='data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(
        trainset, batch_size=64, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(
        root='data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(
        testset, batch_size=64, shuffle=False, num_workers=2)
```

Use the same CIFAR-10 datasets as the local training example

```
stadle_client.set_termination_function(judge_termination, round_to_exit=20, client=stadle_client)
stadle_client.set_training_function(train, trainloader, lr=args.lr, epochs=args.lt_epochs, device=device, agent_name=args.agent_name)
stadle_client.set_cross_validation_function(cross_validate, testloader, device=device)
stadle_client.set_testing_function(test, testloader)
```

Pass functions to IntegratedClient for use in internal training loop

```
stadle_client.set_bm_obj(model)
stadle_client.start()
```

Set the container model for the client, then start the agent FL process

1.3.3 Running Client-Side STADLE Components

After starting the requisite server-side STADLE components, there is one final step that must be run to fully initialize an FL process with STADLE and prepare for agent connections. The component responsible for this is called the *admin agent* - its role in this case is to send the model structure and information to the persistence server for use in converting between specific model frameworks and the framework-agnostic model representation used by STADLE. The following is example admin agent code for the CIFAR-10 example:

```
from stadle import AdminAgent
from stadle import BaseModelConvFormat
from stadle.lib.entity.model import BaseModel
from stadle.lib.util import admin_arg_parser

from vgg import VGG
```

This section imports the required objects from STADLE, as well as a function for reading command line arguments and the VGG model. The BaseModel object acts as a container for information on the model being trained with STADLE, and is passed to the AdminAgent to be sent to the persistence server.

```
base_model = BaseModel("PyTorch-CIFAR10-Model", VGG('VGG16'), BaseModelConvFormat.
↪pytorch_format)
```

The specific BaseModel object is then created with the VGG16 model structure and information.

```
args = admin_arg_parser()
admin_agent = AdminAgent(config_file=args.config_path, simulation_flag=args.simulation,
                         aggregator_ip_address=args.ip_address, reg_socket=args.reg_port,
                         exch_socket=args.exch_port, model_path=args.model_path, base_
↪model=base_model,
                         agent_running=args.agent_running)

admin_agent.preload()
admin_agent.initialize()
```

The command line arguments are parsed and used to create the AdminAgent object, along with the base model. The preload function prepares the base model to be sent (converting to agnostic representation internally) and the initialize function sends the base model information, preparing all of the aggregators to connect to agents by extension.

After the admin agent is run, the main agent client-side code can freely be run. In summary, the order to run components is as follows:

1. Start persistence server
2. Start aggregator(s)
3. Run admin agent (only once)
4. Run agent(s) - client-side code

1.4 Documentation

1.4.1 Client API Documentation

AdminAgent

`stadle.`**`AdminAgent`**

Class of Admin Agent to register initial base models.

BasicClient

`stadle.`**`BasicClient`**(*config_file: str = None*, *simulation_flag=True*, *aggregator_ip_address: str = None*, *reg_port: str = None*, *exch_port: str = None*, *model_path: str = None*, *agent_running: bool = True*)

Create BasicClient using passed-in parameters or parameters from config file (passed-in parameters take priority), used to connnect to a STADLE aggregator and begin participation in FL process

Parameters

- **`config_file`** – Specifies the path of the aggregator config file to read parameter values from, if not provided in the respective constructor parameter. Defaults to value of agent_config_path environmental variable (normally set to setups/config_agent.json) if no path is provided.
- **`simulation_flag`** – Determines if client should operate in simulation mode for testing, or production mode; simulation mode uses the default aggregator token and displays debug information at runtime.
- **`aggregator_ip_address`** – IP address of the aggregator instance to connect to.
- **`reg_port`** – Port to be used to create port for registering through aggregator.
- **`exch_port`** – *Deprecated*
- **`model_path`** – Path to folder used for local storage (client state, id, local and sg models).
- **`agent_running`** – Flag to determine if agent should actively participate in model exchange with aggregator.

Returns

Configured BasicClient object

`stadle.BasicClient.`**`send_trained_model`**(*model*, *perf_values*)

Extract weights from locally-trained model and send weights to aggregator.

Parameters

- **`model`** – Locally-trained model to extract weights from.
- **`perf_values`** – A dictionary containing key-value pairs for different performance metrics to be displayed in STADLE Ops. Valid keys are {'performance','accuracy','loss_training','loss_valid','loss_test','f_score','reward'}.

Returns

False if new aggregated model was received during local training process (nothing sent in this case), True otherwise `stadle.BasicClient.`**`wait_for_sg_model`**`()`

Blocking function that waits to receive the aggregated model from the aggregator.

Returns
Model object with aggregated weights from previous round.

`stadle.BasicClient.`**`set_bm_obj`**`(`*model*`)`

Set container model object in IntegratedClient for use when converting to/from agnostic format.

Parameters
`model` – Used as a container to store aggregated model weights (for ease of use in local training).

`stadle.BasicClient.`**`disconnect`**`()`

Disconnect client and exit from FL process participation.

IntegratedClient

`stadle.`**`IntegratedClient`**`(`*config_file: str = None*, *simulation_flag=True*, *aggregator_ip_address: str = None*, *reg_port: str = None*, *exch_port: str = None*, *model_path: str = None*, *agent_running: bool = True*`)`

Create IntegratedClient using passed-in parameters or parameters from config file (passed-in parameters take priority), used to connnect to a STADLE aggregator and begin participation in FL process.

Parameters

- **`config_file`** – Specifies the path of the aggregator config file to read parameter values from, if not provided in the respective constructor parameter. Defaults to value of agent_config_path environmental variable (normally set to setups/config_agent.json) if no path is provided.
- **`simulation_flag`** – Determines if client should operate in simulation mode for testing, or production mode; simulation mode uses the default aggregator token and displays debug information at runtime.
- **`aggregator_ip_address`** – IP address of the aggregator instance to connect to.
- **`reg_port`** – Port to be used to create port for registering through aggregator.
- **`exch_port`** – *Deprecated*
- **`model_path`** – Path to folder used for local storage (client state, id, local and sg models).
- **`agent_running`** – Flag to determine if agent should actively participate in model exchange with aggregator.

Returns
Configured IntegratedClient object

`stadle.IntegratedClient.`**`set_training_function`**`(`*fn*, *train_data*, ***kwargs*`)`

Pass model training function, data, and associated arguments to the IntegratedClient for use during local training.

Model training function must take model, data, and keys of kwargs as arguments. It must also return the trained model and a training performance metric (float value).

Parameters

- **`fn`** – Function to perform model training using train_data and kwargs.
- **`train_data`** – Data object provided to training function during FL process.
- **`**kwargs`** – Additional required arguments for training function, passed to the function each time it is called.

`stadle.IntegratedClient.`**`set_cross_validation_function`**(*fn*, *cross_validation_data*, ***kwargs*)

Pass model validation function, data, and associated arguments to the IntegratedClient for use during FL process.

Model validation function must take model, data, and keys of kwargs as arguments. It must also return two performance metrics (float values).

Parameters

- **`fn`** – Function to perform model training using cross_validation_data and kwargs.
- **`cross_validation_data`** – Data object provided to validation function during FL process.
- **`**kwargs`** – Additional required arguments for validation function, passed to the function each time it is called.

`stadle.IntegratedClient.`**`set_testing_function`**(*fn*, *test_data*, ***kwargs*)

Pass model test function, data, and associated arguments to the IntegratedClient for use at end of FL process.

Model test function must take model, data, and keys of kwargs as arguments. It must also return two performance metrics (float values).

Parameters

- **`fn`** – Function to perform model training using test_data and kwargs.
- **`test_data`** – Data object provided to validation function during FL process.
- **`**kwargs`** – Additional required arguments for test function, passed to the function when it is called.

`stadle.IntegratedClient.`**`set_termination_function`**(*fn*, ***kwargs*)

Pass agent termination function and associated arguments to the IntegratedClient for use in managing the FL process.

Parameters

- **`fn`** – Function to determine if agent should stop participation and disconnect. Must return either True or False.
- **`**kwargs`** – Required arguments for termination function, passed to the function each time it is called.

`stadle.IntegratedClient.`**`set_bm_obj`**(*model*)

Set container model object in IntegratedClient for use when converting to/from agnostic format.

Parameters

`model` – Used as a container to store aggregated model weights (for ease of use in local training).

`stadle.IntegratedClient.`**`start`**()

Start FL process defined by functions passed to IntegratedClient. STADLE then manages both the client-side and server-side of FL.

1.4.2 Config File Documentation

Configuration of Agent

This JSON file, for example *config_agent.json* file, is read by STADLE admin and ML agents for initial setup. Here is the sample content of the JSON file.

```
{
    "agent_name": "default_agent"
    "model_path": "./data/agent",
    "local_model_file_name": "lms.binaryfile",
    "semi_global_model_file_name": "sgms.binaryfile",
    "state_file_name": "state",
    "aggr_ip": "localhost",
    "reg_port": "8765",
    "init_weights_flag": 1,
    "token": "stadle12345",
    "simulation": "False",
    "exch_port": "0000"
}
```

- *agent_name*: A unique name of the agent that users can define. - e.g. *default_agent*
- *model_path*: A path to a local director in the agent machine to save local models and some state info. - e.g. *./data/agent*
- *local_model_file_name*: A file name to save local models in the agent machine. - e.g. *lms.binaryfile*
- *semi_global_model_file_name*: A file name to save the latest semi-global models in the agent machine. - e.g. *sgms.binaryfile*
- *state_file_name*: A file name to store the agent state in the agent machine. - e.g. *state*
- *aggr_ip*: An aggregator IP address for agents to connect. - e.g. *localhost*, *123.456.789*
- *reg_port*: A port number used by agents to join an aggregator for the first time. - e.g. *8765*
- *init_weights_flag*: A flag used for initializing weights. - e.g. *1*
- *token*: A token that is used for registration process of agents. Agents need to have the same token to be registered in the STADLE system. - e.g. *stadle12345*
- *simulation*: A flag used to enable a simulation mode. - e.g. *True*
- *exch_port*: A port number used to upload local models to an aggregator from an agent. Agents will get to know this port from the communications with an aggregator. - e.g. *7890*

1.5 Contributing

Thanks for your interest in contributing to STADLE! Read on to learn what would be helpful and how you could contribute to the STADLE community.

1.5.1 Developers

When developing stadle, make sure to install the STADLE in develop mode. This mode allows the developer to observe the changes made to the code without installing STADLE each time an update is made to the source.

To do debug the application, use the following command.

```
python -m venv DEVSTADLE
source DEVSTADLE/bin/activate
```

Then, change the directrory to the STADLE source, and run the following command.

```
python setup.py develop
```

Additionally, to include tests, install as follows.

```
pip install -e ".[dev]"
```

Run the test cases,

```
pytest test/
```

Note: If you are using the STADLE outside the source folder, make sure you `copy` the `setups` and `prototypes` folders to your workspace to test things out.

1.5.2 Using Docker

After changing the directory to the STADLE source, build Docker compose

```
docker-compose build
```

Then, start Docker compose

```
docker-compose up
```

Also, reach out with your issues or proposals to improve STADLE.

1.5.3 Bug Reports

Please check/submit issues here.

1.5.4 Tech Support

Please reach out to our technical support team via `support@tie-set.com`.

INDEX