Turbinia

Oct 16, 2021

Contents

Turbinia is an open-source framework for deploying, managing, and running distributed forensic workloads.

The source code is available from https://github.com/google/turbinia

Contents:

CHAPTER 1

User documentation

1.1 Turbinia Quick Installation Instructions

1.1.1 Overview

Turbinia can be run on the Google Cloud Platform, on local machines, or in a hybrid mode. See the "how it works" documentation for more details on what the architecture looks like for each of these installation types. This doc covers the recommended quick installation instructions for Cloud installations. This uses terraform configs that are part of the Forseti Security repository to automate deployment of Turbinia into an existing GCP Project. If you want to install Turbinia in hybrid or local only mode, or want to install Turbinia manually (not recommended), see here for details.

1.1.2 Installation

The following steps can be performed on any Linux machine (Ubuntu 18.0.4 recommended), and Cloud Shell is one easy way to get a shell with access to your GCP resources.

GCP Project Setup

- Create or select a Google Cloud Platform project in the Google Cloud Console.
- Determine which GCP zone and region that you wish to deploy Turbinia into. Note that one of the GCP dependencies is Cloud Functions, and that only works in certain regions, so you will need to deploy in one of the supported regions.
- Install google-cloud-sdk.
  - Note: If you are doing this from cloud shell you shouldn't need this step.
- Run `gcloud auth login` to authenticate. This may require you to copy/paste url to browser.
- Run `gcloud auth application-default login`

Deploy Turbinia

- Download the Terraform CLI from here.
- Clone the Forseti Security repository and change to the path containing the configs
  - `git clone https://github.com/forseti-security/osdfir-infrastructure/`
  - `cd osdfir-infrastructure`
- Configuration
  - By default this will create one Turbinia server instance and one worker instance. If you want to change the number of workers, edit the `modules/turbinia/variables.tf` file and set the `turbinia_worker_count` variable to the number of workers you want to deploy.
  - To adjust the GCP zone and region you want to run Turbinia in, edit the `modules/turbinia/variables.tf` file and change the `gcp_zone` and `gcp_region` variables as appropriate to reflect your GCP project's zone and region.
  - If you want to use docker to run Turbinia tasks, please follow the instructions here to enable docker.
  - Running the following commands will leave some state information under the current directory, so if you wish to continue to manage the number of workers via Terraform you should keep this directory for later use. Alternatively, if you wish to store this information in GCS instead, you can edit `main.tf` and change the `bucket` parameter to the GCS bucket you wish to keep this state information in. See the Terraform documentation for more information.
  - The current configuration does not enable alert notifications by default. Please see *here* for the instructions
  - If you are running multiple workers on a given host and within containers, ensure that you are mapping the host `OUTPUT_DIR` path specified in the configuration file `.turbiniarc` to the containers so that they can properly update the `RESOURCE_STATE_FILE`.
- Initialize terraform and apply the configuration
  - `./deploy.sh --no-timesketch`
    - If the `--no-timesketch` parameter is not supplied, Terraform will also create a Timesketch instance in the same project, and this can be configured to ingest Turbinia timeline output and report data. See the Documentation on this for more details.
    - When prompted for the project name, enter the project you selected during setup.

This should result in the appropriate cloud services being enabled and configured and GCE instances for the server and the worker(s) being started and configured. The Turbinia configuration file will be deployed on these instances as `etc/turbinia/turbinia.conf`. If you later want to increase the number of workers, you can edit the `turbinia/variables.tf` file mentioned above and re-run `terraform apply` To use Turbinia you can use the virtual environment that was setup by the `deploy.sh` script.To activate the virtual environment, run the following command `source ~/turbinia/bin/activate` and then use `turbiniactl`. For more information on how to use Turbinia please visit the user manual.

Client configuration (optional)

If you want to use the command line tool, you can SSH into the server and run `turbiniactl` from there. The `turbiniactl` command can be used to submit Evidence for processing or see the status of existing and previous processing requests. If you'd prefer to use turbiniactl on a different machine, follow the following instructions to configure the client. The instructions are based on using Ubuntu 18.04, though other versions of Linux should be compatible.

- Follow the steps from GCP Project setup above to install the SDK and authenticate with gcloud.

- Install some python tooling:
  - `apt-get install python3-pip python3-wheel`
- Install the Turbinia client.
  - Note: You may want to install this into a virtual-environment with venv or pipenv to reduce potential dependency conflicts and isolate these packages into their own environment.
  - `pip3 --user install turbinia`
- If running on the same machine you deployed Turbinia from, you can generate the config with terraform
  - `terraform output turbinia-config > ~/.turbiniarc`
- Otherwise, if you are running from a different machine you'll need to copy the Turbinia config from the original machine, or from the Turbinia server from `/etc/turbinia/turbinia.conf`.

Grafana SMTP Setup

If you want to receive alert notifications from Grafana, you'll need to setup a SMTP server for Grafana. To configure a SMTP server, you need to add the following environment variables to `Grafana env` section in `osdfir-infrastructure/modules/monitoring/main.tf`

```
    {
      name = "GF_SMTP_ENABLED"
      value = "true"
    }, {
      name = "GF_SMTP_HOST"
      value = "smtp.gmail.com:465" # Replace this if you're not using gmail
    }, {
      name = "GF_SMTP_USER"
      value = "<EMAIL ADDRESS HERE>"
    }, {
      name = "GF_SMTP_PASSWORD"
      value = "<PASSWORD>"
    }, {
      name = "GF_SMTP_SKIP_VERIFY"
      value = "true"
    }, {
      name = "GF_SMTP_FROM_ADDRESS"
      value = "<EMAIL ADDRESS THAT SHOWS AS THE SENDER>"
    }
```

NOTE

By default Gmail does not allow less secure apps to authenticate and send emails. For that reason, you'll need to allow less secure apps to access the provided Gmail account.

Once completed:

- login to the Grafana Dashboard.
- Select Alerting and choose "Notification channels".
- Fill the required fields and add the email addresses that will receive notification.
- Click "Test" to test your SMTP setup.
- Once everything is working, click "Save" to save the notification channel.

1.2 Turbinia Local and Manual Installation Instructions

1.2.1 Introduction

Turbinia can be run on the Google Cloud Platform, on local machines, or in a hybrid mode. See the "how it works" documentation for more details on what the architecture looks like for each of these installation types. This page covers the local installation as well as the manual steps for hybrid and cloud installation. **If you are setting up a GCP cloud or hybrid installation, it is highly recommended and much simpler to use the terraform installation method to bootstrap those environments.**

Each section of this document indicates which installation types it applies to (cloud, hybrid or local), so you only need to follow the relevant steps for your installation type.

Prerequisites

Turbinia requires all worker nodes to have equal access to all Evidence data. For Google Cloud this means using Google Cloud Storage (GCS), but for local and hybrid configurations the easiest setup is to have NFS or a SAN mounted on a common path on each worker. All output should also be written to this common directory so that when new Evidence is generated by a Task, that the other worker nodes can access it for processing. Turbinia can also write output to GCS even when running locally (see the `GCS_OUTPUT_PATH` variable in the config).

1.2.2 Installation

To run Turbinia it's recommended that you have at least two machines or cloud instances, one for the server and one or more for workers. In a small or development setup, you can also run both the server and worker on a single instance.

GCP Setup

This section is required for cloud and hybrid configurations.

- Create or select a Google Cloud Platform project in the Google Cloud Console.
- Determine which GCP zone and region that you wish to deploy Turbinia into. Note that one of the GCP dependencies is Cloud Functions, and that only works in certain regions, so you will need to deploy in one of the supported regions.
- Enable Cloud Functions.
- Follow the instructions to:
  - Enable Cloud Pub/Sub
  - Create a new Pub/Sub topic and subscription (pull type with 600s timeout). These can use the same base name (the part after `topics/` and `subscription/` in the paths).
  - Please take a note of the topic name for the configuration steps, as this is what you will set the `PUBSUB_TOPIC` config variable to.
- Enable Cloud Datastore
  - Go to Datastore in the cloud console
  - Hit the `Create Entity` button
  - Select the same region that you selected in the previous steps. No need to create any Entities after selecting your region

Create a GCE Instance for the Server

This section is required for cloud configurations.

- Create a new GCE instance from a recent version of Debian or Ubuntu (currently 18.0.4 is recommended). Once the host is configured we'll later clone the disk to use for the workers.
  - This should work on other Linux flavors, but these are untested, and the installation steps would need to be adapted.

Create a Google Cloud Storage (GCS) Bucket

This section is required for cloud installations.

- Create a new GCS bucket and take note of the bucket name as this will be referenced later by the `GCS_OUTPUT_PATH` variable.

Core Installation

This section is required for cloud, hybrid and local configurations.

Preparation

All steps listed below must be completed on all servers/workers unless otherwise noted. If you are installing in a cloud environment, you will only have a server VM at this point, and we will later clone this instance to create the workers.

- Install dependencies
  - `sudo apt-get install python-dev build-essential python-setuptools python-pip python-virtualenv liblzma-dev git john`
- Create a turbinia user with password-less sudo access
  - `sudo adduser --disabled-password turbinia`
  - `echo "turbinia ALL = (root) NOPASSWD: /bin/mount,/bin/umount,/sbin/losetup" | sudo tee -a /etc/sudoers.d/turbinia`
- Prepare configuration directory:
  - `sudo mkdir /etc/turbinia`
  - `sudo chown turbinia /etc/turbinia`
- Checkout git at release branch for config and other setup
  - `git clone https://github.com/google/turbinia.git`
  - `cd turbinia`
  - `git branch -l | grep release`
  - `git checkout <latest release branch from previous step>`

Installation and Configuration

- Log in as turbinia
  - `sudo su - turbinia`
- Install Turbinia
  - Note: You may want to install this into a virtual-environment with venv or pipenv to reduce potential dependency conflicts and isolate these packages into their own environment.
  - `pip3 install turbinia` for the server
  - `pip3 install turbinia[worker]` for the worker
  - `pip3 install turbinia[dev]` if you want to run tests or get the development dependencies.
  - If you are running a local installation:
    - `pip3 install turbinia[local]`
- Install Worker binary dependencies
  - You can install Plaso from the GIFT PPA, or see here for other packaged installations.
  - There are a few other binary dependencies that are not packaged with Ubuntu or PyPi, so these will need to be installed manually: bulk_extractor, hindsight (this one is technically in PyPi, but since it's not Python3 yet it needs to be installed separately) and Volatility. Alternately you can disable the Jobs that have those dependencies. You can do this by setting the `DISABLED_JOBS` in the `/etc/turbinia/turbinia.conf` config after it is installed (see below). You can see the list of Jobs with `turbiniactl listjobs` after everything is set up.
- Create and configure the Turbinia configuration file.
  - `cp <git clone path>/turbinia/config/turbinia_config_tmpl.py /etc/turbinia/turbinia.conf`
  - Alternately, you can either put the file in `/home/$USER/.turbiniarc` or in another directory and then point the `TURBINIA_CONFIG_PATH` environment variable to that directory.
  - Edit the config file to match your local installation details.
    - Match the `PUBSUB_TOPIC` variable in the configuration to the name of the topic you created in GCP.
    - If you are running Turbinia locally, make sure to set `GCS_OUTPUT_PATH` to `None`.
    - For local and hybrid installations:
      - Set `SHARED_FILESYSTEM = True`
    - For local installations:
      - Set `STATE_MANAGER = 'Redis'`
      - Set `TASK_MANAGER = 'Celery'`
      - Configure the `CELERY*`, `KOMBU*` and `REDIS*` variables as appropriate for your config.
    - Set the following
  - Configure the `OUTPUT_DIR`, `TMP_DIR`, and `MOUNT_DIR_PREFIX` to match your local system. On the worker nodes, create the corresponding directories and make sure they are owned by the turbinia user.
- Configure the init scripts to run Turbinia on start
  - `cp <git clone path>/turbinia/tools/turbinia@.service /etc/systemd/system/turbinia@server` for the server

  - `cp <git clone path>/turbinia/tools/turbinia@.service /etc/systemd/system/turbinia@psqworker` for a GCP worker
  - `cp <git clone path>/turbinia/tools/turbinia@.service /etc/systemd/system/turbinia@celeryworker` for a local (non-cloud) installation.
  - Follow the instructions at the top of the `turbinia/tools/turbinia@.service` file to enable these services.

GCP Installation

This section is required for cloud and hybrid configurations.

- Install google-cloud-sdk
- Create a scoped service account (this is the best option) with the following roles:
  - `Cloud Datastore User`: Used by PSQ to store result data, and in the future by the Task Manager to store queryable task data
  - `Pub/Sub Editor`: Used by clients to talk to Turbinia, and by the Task Manager to talk to workers
  - `Storage Object Admin` and `Storage Legacy Bucket Reader`: Only required on the GCS bucket used by Turbinia, if any. See GCP Turbinia for details
  - `Compute Instance Admin`: Used to list instances and to attach disks to instances
  - `Service Account User`: Used when attaching disks
  - `Cloud Functions Developer`: Used by turbiniactl to query task status
- Create a new key for your service account, and save it on server/workers and then configure init scripts in `/etc/systemd/system/turbinia*` to point to it by setting the `GOOGLE_APPLICATION_CREDENTIALS` var similar to `ExecStartPre=+/bin/sh -c '/bin/systemctl set-environment GOOGLE_APPLICATION_CREDENTIALS="/home/turbinia/turbinia-service-account-creds.json"'`
- Add the service account to the gcloud auth
  - `gcloud auth list`
  - `gcloud auth activate-service-account --key-file=$GOOGLE_APPLICATION_CREDENTIALS`
- Alternately you can run Turbinia under your own credentials (NOT RECOMMENDED except for development environments)
  - Run `gcloud auth login` (may require you to copy/paste url to browser)
  - Run `gcloud auth application-default login`

Create GCP workers

This section is required for cloud configurations.

- Stop the server instance that has been configured above.
- Create a new image from the server VM's disk.
- Create a new Instance Template using the newly created image.
- Create a new Managed Instance Group from the newly created Instance Template.

Deploy Cloud Functions

This section is required for cloud and hybrid configurations.

- `cd <git clone path>/tools/gcf_init && ./deploy_gcf.py`

1.2.3 Local Turbinia

Running Turbinia locally using Docker. This setup does not require Google Cloud Platform.

See here for detailed instructions.

1.3 Turbinia GKE Installation Instructions

1.3.1 Introduction

Turbinia can be run within Google Kubernetes Engine (https://cloud.google.com/kubernetes-engine). This allows Turbinia Workers to scale based on processing demand. Currently, this is done through scaling on CPU utilization, which is determined when available Turbinia Workers process Tasks and reach a pre-defined CPU threshold. The GKE architecture closely resembles the cloud architecture with GKE being used to scale Turbinia Woker pods.

All steps in this document are required for getting Turbinia running on GKE. Please ensure you have followed the GCP setup prior to configuring the GKE cluster as it's required for Turbinia components to properly function together.

Prerequisites

GKE is only supported for Google Cloud so a Google Cloud Project is required to work from. Additionally, all GCP components specified below must be enabled so please follow the GCP steps outlined prior to setting up GKE.

1.3.2 Installation

Please follow these steps for configuring Turbinia for GCP use and then running it within GKE. Ensure that the `.turbiniarc` config file has been updated appropriately.

GCP Setup

Follow these steps prior to configuring GKE for Turbinia.

- Create or select a Google Cloud Platform project in the Google Cloud Console.
- Determine which GCP zone and region that you wish to deploy Turbinia into. Note that one of the GCP dependencies is Cloud Functions, and that only works in certain regions, so you will need to deploy in one of the supported regions.
- Enable Cloud Functions.
- Follow the instructions to:
  - Enable Cloud Pub/Sub
  - Create a new Pub/Sub topic and subscription (pull type with 600s timeout). These can use the same base name (the part after `topics/` and `subscription/` in the paths).

  - Please take a note of the topic name for the configuration steps, as this is what you will set the `PUBSUB_TOPIC` config variable to.
- Enable Cloud Datastore
  - Go to Datastore in the cloud console
  - Hit the `Create Entity` button
  - Select the same region that you selected in the previous steps. No need to create any Entities after selecting your region
- Create a new GCS bucket and take note of the bucket name as this will be referenced later by the `GCS_OUTPUT_PATH` variable.
- Deploy Cloud Functions `cd <git clone path>/tools/gcf_init && ./deploy_gcf.py`

GKE Setup

- Enable the Kubernetes Engine API.
- Create or select a Google Kubernetes Engine cluster in the Google Cloud Console.
  - Choose the GKE Standard type.
  - Choose any name for the cluster.
  - The Zone should match the GCP Zone configuration (e.g. wherever Zone cloud functions & Datastore was deployed to).
  - Choose number of nodes based on processing & cost requirements.
  - Change machine type based on processing & cost requirments.
  - In the Node Pools -> Security tab, change access scopes to "Allow full access to all cloud APIs"
- Alternatively, a GKE cluster can be created via gcloud.

```
gcloud container clusters create CLUSTER_NAME \
    --release-channel None \
    --zone COMPUTE_ZONE \ # Should match GCS/GCP Zones.
    --node-locations COMPUTE_ZONE # Should match GCS/GCP Zones.
    --num_nodes 3 \ # Change based on processing requirements.
    --machine-type e2-medium # Change based on processing requirements.
    --scopes "https://www.googleapis.com/auth/cloud-platform"
```

Turbinia GKE Deployment

- Connect to cluster through `gcloud container clusters get-credentials <CLUSTER_NAME> --zone <ZONE> --project <PROJECT_NAME>`.
- Clone the latest Turbinia branch and `cd <git clone path>/k8s/gcp-pubsub`.
- Ensure that the zone and region in the Turbinia config file are equal to the zone and region you created your k8s cluster in.
- The `image` variable can be optionally changed in the `turbinia-worker.yaml` and `turbinia-server.yaml` files to chose the docker images used during deployment.
- In the `turbinia-worker.yaml` file, ensure that the path in the volume labeled `lockfolder` matches the Turbinia config variable `TMP_RESOURCE_DIR`.

- Ensure that the `.turbiniarc` config file has been properly configured with required GCP variables.
- Deploy the Turbinia infrastructure by executing `./setup-pubsub.sh <PATH TO CONFIG>`.
- The Turbinia infrastructure can be destroyed by executing `./destroy-pubsub.sh`.

Making processing requests in GKE

- You can either make requests via setting up a local `turbiniactl` client or through connecting to the server through the following steps.
- Connect to cluster through `gcloud container clusters get-credentials <CLUSTER_NAME> --zone <ZONE> --project <PROJECT_NAME>`.
- Use `kubectl get pods` to get a list of running pods.
- Identify the pod named `turbinia-server-*` and exec into it via `kubectl exec --stdin --tty [CONTAINER-NAME] -- bash`
- Use `turbiniactl` to kick off a request to process evidence.

1.4 How Turbinia Works

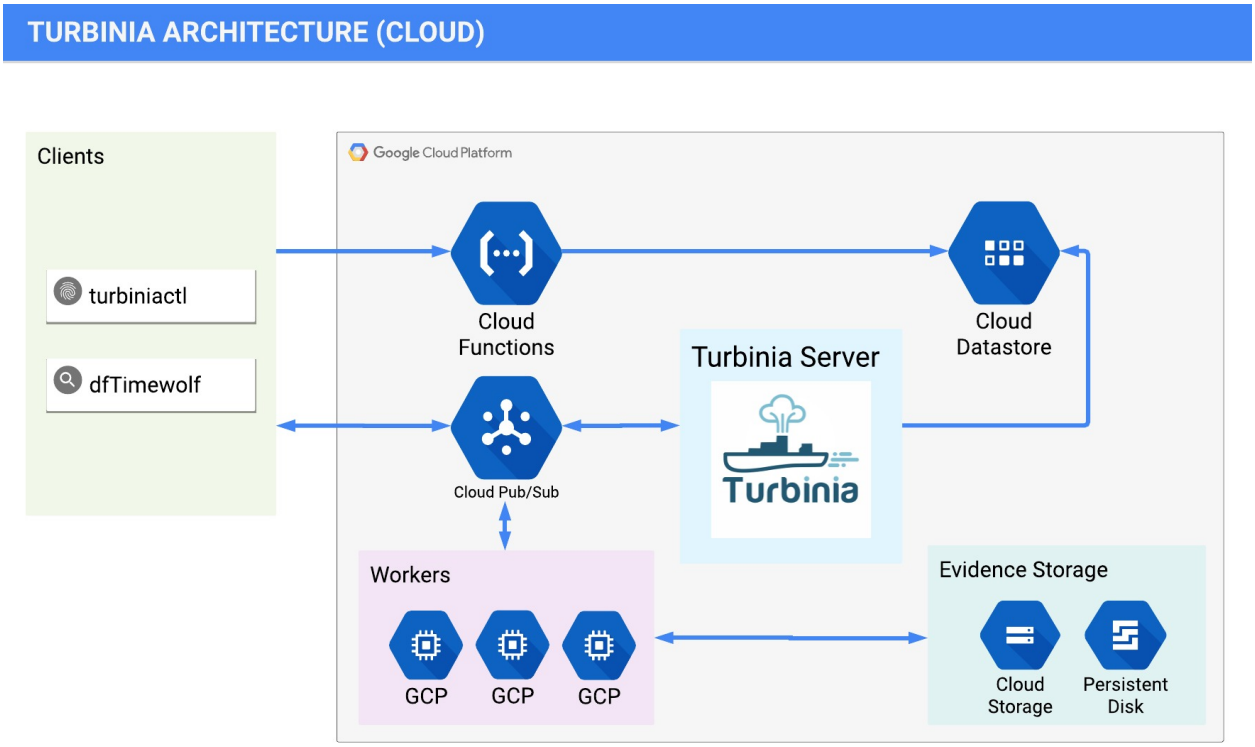
1.4.1 General

This page contains some of the details on the internals of how Turbinia works.

1.4.2 Architecture

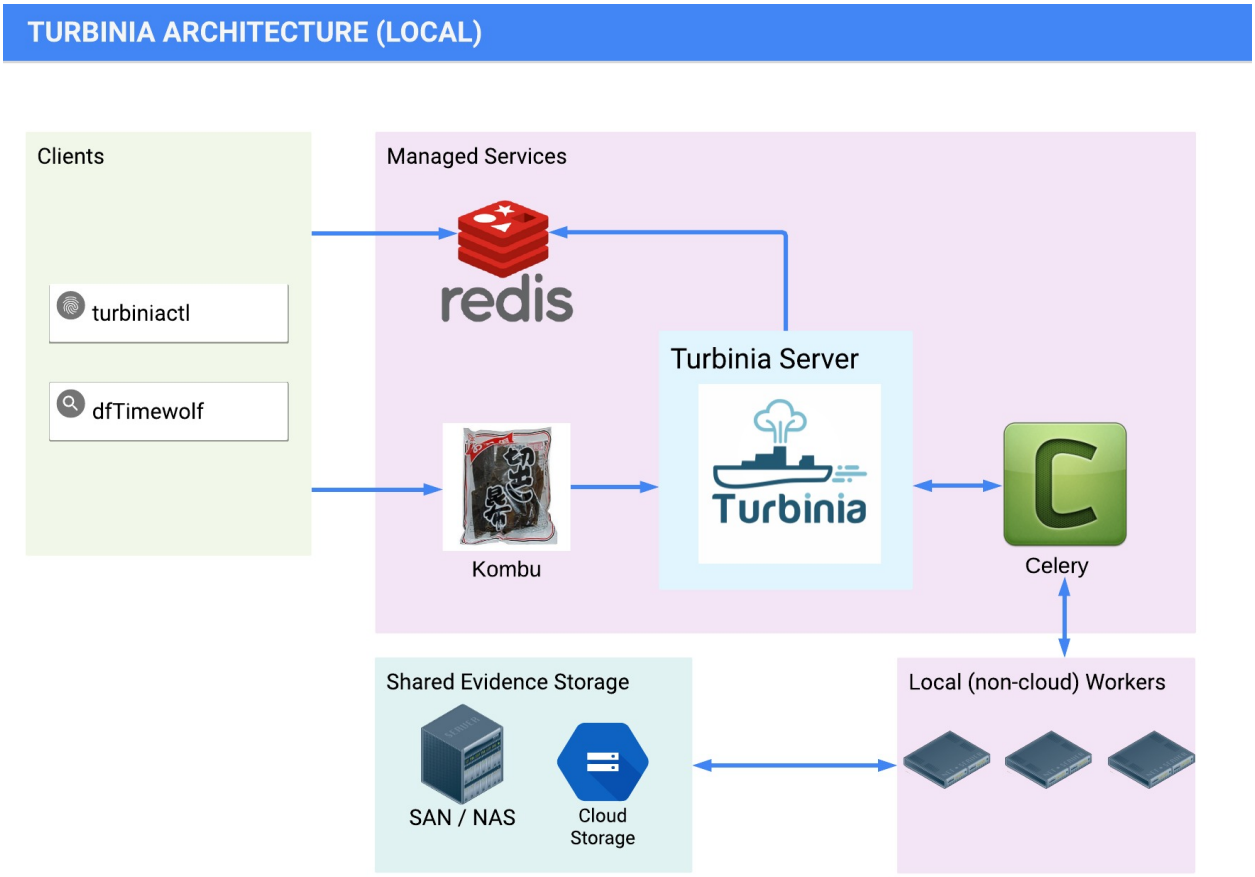
Turbinia can run its components completely in the Cloud, on local machines, or as a hybrid of both. When running in hybrid mode, Turbinia uses Cloud services, but the workers and the Turbinia server run locally. Running locally or in hybrid mode requires shared storage to be accessible by all Workers.

Cloud Architecture



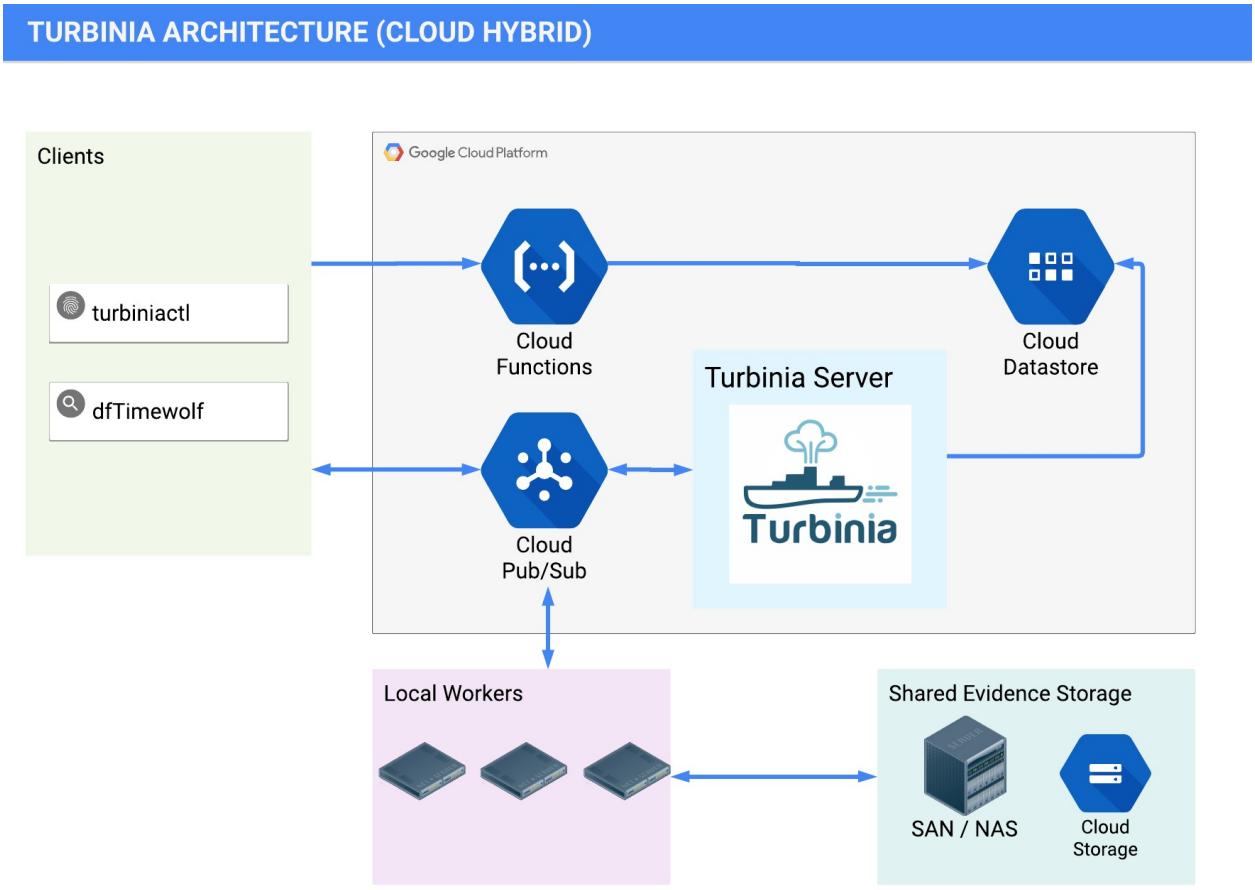
turbinia-architecture-cloud

Local Architecture



turbinia-architecture-local.jpg

Hybrid Architecture



turbinia-architecture-hybrid

1.4.3 Tasks

A Task is the smallest discrete unit of schedulable work. Tasks are scheduled and executed on remote Workers. A Task can generate more Evidence that is returned back to the Turbinia server for possible further processing.

1.4.4 Jobs

A Job is a larger logical unit of work that creates one or more Tasks to process that work. A Job can either create a single Task, or it can break up that work and create multiple Tasks. Sometimes these terms are used interchangeably (though we mostly talk about things in terms of Tasks) because in most cases a Job will just create a single Task.

1.4.5 Workers

Workers are independent processes that run either in Cloud GCE instances or on local machines. They run continuously and wait for new Tasks to be scheduled. When a new Task is scheduled, a pseudo-random worker will pick up the Task and execute it. After the Task is complete (successfully or not), the Worker will return the status, any error logs and results.

1.4.6 Evidence

Evidence can be anything that Turbinia can process. Examples include disk images, cloud disks, Plaso files, strings files, etc. When you make a request to Turbinia only the metadata for the evidence is passed into the request, but it contains pointers to where the data is.

New Evidence

If you want to create a new Evidence type, they are simple Python objects in evidence.py. You can use object inheritance (e.g. an GoogleCloudDisk is a subclass of a RawDisk) if you have multiple related Evidence types.

Evidence Processors

Each Evidence object can have a pre- and post-processor that prepares Evidence prior to Task execution and cleans up afterwards. Evidence pre-processors run on the Worker node just prior to Task execution, and post-processors run after the task execution is complete. Processors can be used to do things like attach a Cloud Persistent Disk or mount a local disk. Because of the inheritance that comes from how Evidence objects are defined in code, these processors can be stacked together to run multiple Processors. An example of this is the `GoogleCloudDiskRawEmbedded` Evidence type which is a `GoogleCloudDisk` that contains a `RawDisk` image file. For this Evidence type, the Processor for the `GoogleCloudDisk` runs first to attach the disk to the worker, and then the Processor for the `RawDisk` runs to mount the "inner" raw disk image.

Copyable Evidence

Some types of Evidence can be marked as "copyable". What that means is that this kind of Evidence can be copied around as needed (either to make it available for a new Task to use it, or to copy it off of a Worker after the Task has completed). This is handled transparently by the Output Manager when it is configured. An example of this is the `PlasoFile` Evidence type. Right now the only storage that the Output Manager supports is Google Cloud Storage. If Evidence is not copyable (like a RawDisk) and not a Cloud Evidence type (as denoted by the `cloud_only` attribute), then you will need to have a shared disk that is available to all Workers.

1.4.7 Task Manager

The Task Manager runs in the server and acts as a broker between the clients and workers and handles management of Evidence, Jobs, Tasks and Workers. It will keep state on these objects while the processing request that generated them is still ongoing.

1.4.8 Task Manager Flow

Jobs are configured to process specific Evidence types, and if the Task Manager sees a new piece of Evidence (either from a new Turbinia request, or because another Task generated a new piece of Evidence), and there is a Job that is configured to run for that type of Evidence, then the Task Manager will create a new Job for it. The Job then generates one or more Tasks which will be immediately scheduled for execution.

Here is a breakdown of what happens during a typical processing request:

- The client sends a processing request to the server along with the Evidence to process
- The server creates new Jobs for each Job type that can process that Evidence type
- The Job generates one or more Tasks
- The Task Manager schedules the Tasks to be executed

- Workers from the pool pick up the Tasks for execution
  - Tasks read the Evidence to process from shared storage or copy it from cloud storage
  - The Task runner pre-processes the Evidence
  - The Task performs the actual processing and potentially generates new Evidence
  - The Task runner post-processes the Evidence
  - Any newly generated Evidence data will be saved to storage by the Output Manager
  - Tasks return results and any new Evidence to the Server
- The Task Manager records the Task results into Datastore/Redis
- The Task Manager picks up the newly created Evidence and restarts this processing loop

1.5 Turbinia local stack using Docker

Turbinia can be run locally without any Cloud components using Docker. It will use Redis, Celery and local disk to store data and perform message broker functionality.

1.5.1 Caveats

rawdisk: As Turbinia uses the loop device to mount different types of evidence (eg raw disks) the host operating system should support the loop device. Linux is currently the only OS that supports the processing of raw disks.

googleclouddisk: Turbinia as local stack cannot currently process Google Cloud disks.

1.5.2 Running

Step 1

Checkout the Turbinia source code. If you intend to start developing please fork the repository on github first and check out your own forked instance.

```
$ git clone https://github.com/google/turbinia.git
$ cd turbinia
```

Step 2

Generate configuration file using sed with default local stack values to the `./conf` folder. This folder will be mapped by docker compose into the containers.

```
$ mkdir ./conf
$ sed -f docker/local/local-config.sed turbinia/config/turbinia_config_tmpl.py > conf/
↪turbinia.conf
```

Step 3

Let's bring up the local Turbinia stack

```
$ docker-compose -f ./docker/local/docker-compose.yml up
```

Redis, a Turbinia server and worker should now be running on your local system and a local persistent 'evidence' folder will have been created containing the Turbinia log file and processing output. Note: Redis will store it's data in a volume that is mapped to `./redis-data/`. You can adjust this in the docker-compose.yml configuration.

Step 4

Let's process evidence to test your setup, eg a Chrome Browser history file.

```
$ curl https://raw.githubusercontent.com/obsidianforensics/hindsight/master/tests/
↪fixtures/profiles/60/History > History
$ tar -vzcf ./evidence/history.tgz History
```

This command runs the turbinia client, turbiniactl, within the turbinia server docker container and generates a processing request.

```
$ docker exec -ti turbinia-server turbiniactl compresseddirectory -l /evidence/
↪history.tgz
```

This will create a task in Turbinia to process the evidence file. A request ID will be returned and we can query the status with below command.

```
$ docker exec -ti turbinia-server turbiniactl -a status -r {REQUEST_ID}
```

There will be server and worker output displayed both on the docker-compose terminal as well as in the `./evidence` folder.

1.6 Using Docker for Job execution

1.6.1 Overview

Turbinia supports the use of Docker by allowing a Task to execute its command through a Docker Container instead. For example, when a Task for a PlasoJob is passed to a Worker, the Task will execute the command `log2timeline.py <ARGS>` through the Container and pass back the associated data to the Worker for further processing. The benefit being that it eliminates having to install the required dependencies and/or external programs for a Job on the Worker's host machine. Please note that Turbinia does not provide the Docker Images necessary for each Job and they will need to either be created or pulled from a container registry.

1.6.2 Enabling Docker usage

In order to enable this feature, please take the following steps.

1. Install the Docker daemon on the Worker's host machine. Please visit the Docker website for the Installation Guide.
2. In the `.turbiniarc` configuration file, set the `DOCKER_ENABLED` flag to `True` to enable the usage of Docker.

3. Review the `DEPENDENCIES` flag in the `.turbiniarc` configuration file and identify which Job you would like to execute a Docker container for. Once identified, replace the value for `docker_image` with the `image_id` of the Docker image.
4. Save the `.turbiniarc` configuration file then restart all Workers for the changes to take into effect.
5. When the Workers start, they will perform dependency checks to ensure that the binaries required by the Job are installed in the Container, and if that check passes, it will execute those in the configured Docker Container.
6. If you no longer would like to use the Docker image, set the `docker_image` value back to `None`.

1.6.3 Example using Plaso

The following section provides an example of the steps mentioned above for the Plaso Job by using the Docker CLI to retrieve the required information.

1. Retrieve the latest Plaso Docker image either locally or through a preconfigured registry containing the image.
   - `docker pull log2timeline/plaso`
2. Identify the `image_id` for the retrieved image.
   - `docker image ls`

   Then copy the value listed under the column `IMAGE ID`.

```
REPOSITORY                                      TAG                 IMAGE ID
↪       CREATED             SIZE
log2timeline/plaso                              latest              9c22665bff50
↪       4 days ago          314MB
```

3. Open up the `.turbiniarc` configuration file then set the attribute `DOCKER_ENABLED` to `True`.
4. Identify the `DEPENDENICES` attribute and look for the Job `PlasoJob`, then replace the `docker_image` value with the identified `IMAGE ID`.

```
{
    'job': 'PlasoJob'
    'programs': ['log2timeline.py'],
    'docker_image': '9c22665bff50'
}
```

5. Save the configuration file, then restart the turbinia Worker.
   - `sudo systemctl restart turbinia@psqworker.service`
6. If the dependency check succeeds, once a Worker receives a Docker configured Task, the Task will execute its external command through the Docker Container instead and pass the associated data back to the Worker for further processing.

1.7 Operational Details

1.7.1 Google Cloud Platform (GCP) Processing Details

Turbinia can read Evidence from either cloud Persistent Disks, or from other Evidence types saved as GCS objects. Turbinia can also write output to GCS. Note that you can operate multiple Turbinia instances within the same GCP Project as long as you're careful to make sure your config options (Pub/Sub topics/subscriptions, output paths, instance name, etc) don't overlap.

Persistent Disks

Persistent disks are the default when processing disks that come from the Cloud. The account you run Turbinia as must have access to the persistent disks that you want to process, and those disks must also be in the same zone as the Turbinia workers. If you process GoogleCloudDisk Evidence with Turbinia, the worker node will attach the disk automatically before it runs its tasks. If you already have access to a persistent disk in a separate project, Turbinia can copy this into the project where Turbinia is being run.

Processing non-cloud disks in GCP

If you have raw disk images from physical machines or elsewhere that you want to process in the cloud, the best option is to create a new Cloud Persistent Disk, and then copy the image into the filesystem of the new disk. Then you can use the GoogleCloudDiskRawEmbedded Evidence type.

Another option is to convert the raw image to a cloud image, and then create a Persistent Disk from that and process it as the GoogleCloudDisk Evidence type, but this is not generally recommended as it requires zero-padding the disk to a GB boundary which can change the hash of the disk, and isn't considered forensically sound.

One last option is to copy the image into GCS and process them directly from there, but the GoogleCloudDiskRawEmbeded option is generally recommended because this method requires setting up GCS FUSE, and this is less stable than using Persistent Disks. If you do choose this option you will need to configure all of your worker nodes to mounting your GCS bucket at a common path. Once your GCS bucket is mounted, you can process these images as the 'rawdisk' Evidence type.

Stackdriver Logging

Stackdriver Logging can be enabled within Turbinia, which would allow for all Turbinia logs to be centralized into the Stackdriver Logging console.

In order to enable this feature, please set the `STACKDRIVER_LOGGING` config variable to `True` within the `.turbiniarc` configuration file as illustrated below.

```
# Set this to True if you would like to enable Google Cloud Stackdriver Logging.
STACKDRIVER_LOGGING = True
```

Stackdriver Error Reporting

Stackdriver Error Reporting can be enabled within Turbinia, which would allow for full Traceback logging and alerting within the GCP Error Reporting console. Please note that Error Reporting will only alert on failures of a `Task` running on a `Worker`.

In order to enable this feature, please set the `STACKDRIVER_TRACEBACK` config variable to `True` within the `.turbiniarc` configuration file as illustrated below.

```
# Set this to True if you would like to enable Google Cloud Error Reporting.
STACKDRIVER_TRACEBACK = True
```

Prometheus instrumentation

The Turbinia worker and server expose metrics based on Prometheus. The implementation exposes a port so monitoring systems can poll the server and worker to fetch running metrics. The configuration (listening port and address) are defined within the `.turbiniarc` configuration file as illustrated below.

```
# Prometheus listen address and port
PROMETHEUS_ADDR = '0.0.0.0'
PROMETHEUS_PORT = 8000
```

By default it will listen on all interfaces on port 8000.

1.7.2 General Notes

- Turbinia currently assumes that Evidence is equally available to all worker nodes (e.g. through locally mapped storage, or through attachable persistent Google Cloud Disks, etc).

1.8 Recipes

1.8.1 Introduction

Recipes are a way to create pre-defined configurations for what Jobs/Tasks to run and how to run them along with various parameters that Tasks can use to change their runtime behavior for a given processing request. They can contain a number of "global" variables that affect the overall processing and also have per-Task variables that are specific to each Task.

1.8.2 Using Recipes

Recipes can be specified by name when sending a processing request to Turbinia. The name of the recipe is the filename that contains the recipe, which should work with or without specifying the `.yaml` extension.

```
turbiniactl --recipe triage googleclouddisk -d diskname-to-process
```

Note: This currently requires that the `RECIPE_FILE_DIR` configuration variable is set in the config file that you are using and is pointing to a valid directory containing the recipe files. Alternately you can also specify a recipe file directly by referencing the file path:

```
turbiniactl --recipe_path ./recipes/triage.yaml googleclouddisk -d diskname-to-process
```

1.8.3 Writing new Recipes

Recipes are `.yaml` files that are read and validated by the client and passed to the server along with the processing request data. There are no required sections and they can contain a `globals` section and zero or more Task sections. Each Task section must contain a `task:` key that references the relevant Task that the section applies to. Other keys in either the `globals` or Task sections must match the pre-defined keys for those sections. Here is a snapshot of the pre-defined variables allowed in the `globals` section along with the defaults:

```
    'debug_tasks': False,
    'jobs_allowlist': [],
    'jobs_denylist': [],
    'yara_rules': '',
    'filter_patterns': [],
    'sketch_id': None
```

These generally correlate with similarly named command line flags. The current full list can be found here. Each Task specifies the available recipe keys in a `TASK_CONFIG` attribute for the Task object (e.g. here is the `TASK_CONFIG` for the Plaso Task).

Here is a real sample of the `all` Recipe including the description in a comment:

```
# This recipe will run all Jobs with all configuration options turned on for in
# depth "kitchen-sink" processing of everything (e.g. all VSS stores and all
# partitions).  This may take a long time to complete.

globals:
  jobs_allowlist:
    - BinaryExtractorJob
    - BulkExtractorJob
    - FsstatJob
    - GrepJob
    - HadoopAnalysisJob
    - HindsightJob
    - HTTPAccessLogExtractionJob
    - HTTPAccessLogAnalysisJob
    - JenkinsAnalysisJob
    - JupyterExtractionJob
    - JupyterAnalysisJob
    - LinuxAccountAnalysisJob
    - PartitionEnumerationJob
    - PlasoJob
    - PsortJob
    - RedisAnalysisJob
    - RedisExtractionJob
    - SSHDAnalysisJob
    - SSHDExtractionJob
    - StringsJob
    - TomcatExtractionJob
    - TomcatAnalysisJob
    - WindowsAccountAnalysisJob

plaso_base:
  task: 'PlasoTask'
  status_view: 'none'
  hashers: 'all'
  partition: 'all'
  vss_stores: 'all'
```

For adding additional configuration options to a given Task, please see the *recipes configuration section* in the developing new Tasks documentation.

1.9 Debugging

1.9.1 Logs

There are a few different logs that may be useful when debugging:

- Server log: By default this is on the server in `OUTPUT_DIR/turbinia-server.log`
- Worker logs: By default this is on the workers in `OUTPUT_DIR/turbinia-worker.log`

- Task logs: These are logs that are generated by a Task running in the Worker. If you have GCS enabled, then these logs are saved there, and in either case, the paths can be determined with `turbiniactl -a status` (note that -a is required to show all data associated with the tasks).
- Task execution logs. These are the logs that are generated by binaries executed from a Task. For example, Plaso generates its own log file, and that gets saved by the Task. As with the Task logs, these can be stored in GCS and retrieved with `gsutil`.
- If you have `STACKDRIVER_LOGGING` and/or `STACKDRIVER_TRACEBACK` enabled in the config, the logs and/or traceback exceptions will also be logged into stackdriver in same project that Turbinia is configured to run in.

1.9.2 Finding previous request data

By default `turbiniactl` will make a processing request and immediately return. It will print out the request ID when it makes the request, and you can use this to see the current status of the request with `turbiniactl status -r <request id>`. If you specify the `-w` flag (`turbiniactl -w status -r <request id>`), the client will wait for all Tasks to complete and then return. Killing the client will not affect the processing, and you can resume waiting for the task results with the same command.

If you do not have your request ID, you can list all recent requests like this:

```
$ turbiniactl status -i

# Turbinia report for Requests made within 7 days
* 2 requests were made within this timeframe.

## Request ID: 5198949e3fdb8dk569de1268000f97d8
* Last Update: 2021-02-10T08:39:27.446000Z
* Requester: turbiniauser
* Task Count: 441

## Request ID: 69de1268000f97d85198949e3fdb8dk5
* Last Update: 2021-01-12T07:09:03.123000Z
* Requester: turbiniauser
* Task Count: 43
```

1.9.3 Task Debugging

Task Failures

The following is an example of debugging a task failure:

Running turbiniactl, we see a failure of the PsortTask (`-d1` specifies history for the last day):

```
$ ./turbiniactl status -d1

Retrieved 2 Task results:
2017-12-06T15:09:15.013Z PsortTask Failed: Execution failed with status 1
2017-12-06T15:08:49.616Z PlasoTask Successful: Completed successfully in 0:00:14.
↪178579 on aaronp.dev
```

We can specify `-a` to get all info we have about that Task, and this will include the request Id, Task Id, and the other logs associated with the task.

```
$ ./turbiniactl -a status -d1

Retrieved 2 Task results:
2017-12-06T15:09:15.013Z request: None task: 1f8c4b321f444614bde296b0a00bb91f␣
↪PsortTask Failed: Execution failed with status 1
        gs://my-turbinia-bucket/output/1512601742-1f8c4b321f444614bde296b0a00bb91f-
↪PsortTask/worker-log.txt
2017-12-06T15:08:49.616Z request: b94ad420cf5a483fb3fc4409d5fc6dcd task:␣
↪020fe728a2b64dc7ba1da2656e1cc454 PlasoTask Successful: Completed successfully in␣
↪0:00:14.178579 on aaronp.dev
        gs://my-turbinia-bucket/output/1512601704-020fe728a2b64dc7ba1da2656e1cc454-
↪PlasoTask/020fe728a2b64dc7ba1da2656e1cc454.log
        gs://my-turbinia-bucket/output/1512601704-020fe728a2b64dc7ba1da2656e1cc454-
↪PlasoTask/worker-log.txt
```

Note that there may be local paths that show up in the `turbiniactl status` output, and these are local to the Worker that executed that Task.

Using the saved paths above we can display the output with `gsutil`.

```
$ gsutil cat gs://my-turbinia-bucket/output/1512601742-
↪1f8c4b321f444614bde296b0a00bb91f-PsortTask/worker-log.txt

Running psort as [psort.py --status_view none --logfile /var/tmp/1512601742-
↪1f8c4b321f444614bde296b0a00bb91f-PsortTask/1f8c4b321f444614bde296b0a00bb91f.log -w /
↪var/tmp/1512601742-1f8c4b321f444614bde296b0a00bb91f-PsortTask/
↪1f8c4b321f444614bde296b0a00bb91f.csv /var/tmp/1512601730-
↪b04f479ef9094954968474431aa30e8a-PsortTask/b04f479ef9094954968474431aa30e8a.csv]
Execution failed with status 1
```

Full Task report data

To see the report output along with the request status you can specify `-R` similar to: `turbiniactl status -R -r <request id>`. By default this output will be filtered to only show high priority Task report output (as determined by the Task at runtime) in order to filter out uninteresting report info. You can specify `-p <prio num>` to set what priority you want to show in the output report. To see all report output you can specify `-p 100`. To also show all saved files you can specify `-a`. Putting these together will show all output and can be quite a large amount of data: `turbiniactl -a status -r <request id> -R -p 100`.

Determining Task status

If you want to get a view of the total running Tasks, you can specify `turbiniactl status -w` to see what Tasks are running or queued on each of the Workers. If you want to see all of the completed Tasks, you can also specify `-a` to include them: `turbiniactl -a status -w`.

Writing debug logs for binary dependencies

Some Turbinia tasks execute binaries that can also write their own debug logs as part of execution. This can be expensive so it is turned off by default. You can specify `-T` during the request to enable these logs temporarily per-request (e.g. `turbiniactl -T googleclouddisk -d disk-to-process`). Alternately you can set `DEBUG_TASKS = True` in the config file for the Worker.

Disappearing Tasks

If the Turbinia server is scheduling Tasks, but you're not seeing the Tasks or their output for some reason, the most common scenario is that either a second Turbinia server or a second set of Workers is running and operating on the same PubSub queues. Make sure that you have unique values per each Turbinia instance for the following config variables: `PSQ_TOPIC`, `INSTANCE_ID`, `PUBSUB_TOPIC` and `KOMBU_CHANNEL` (if you are using a local Turbinia installation type.

1.9.4 Common Errors

- If you encounter a cycle of errors on the Turbinia server when decoding specific malformed pubsub messages, and the failure happens prior to the server acknowledging the pubsub message, you may need to manually clear the pubsub queue. If you are sure there aren't other outstanding requests, you can just pull and auto-ack the existing messages, otherwise you may need a variation of this command to selectively ack messages:

```
gcloud beta pubsub subscriptions pull --auto-ack $my-pubsub-subscription-name
```

1.10 FAQ

1.10.1 Where do I specify configuration options?

Configuration can either go into `~/.turbiniarc` or `/etc/turbinia/turbinia.conf`

1.10.2 Where are the configuration options documented?

The configuration options are documented in turbinia_config_tmpl.py

1.10.3 How can I write new Tasks?

New Task development documentation can be *found here*

1.10.4 How can I debug problems with Turbinia?

Information on debugging and other common errors can be found here

1.10.5 Where are the files listed in the turbiniactl status output?

Files with local paths listed in the output for `turbiniactl status` are local to the Workers that ran that Task. Files with paths starting with gs:// are in the Google Cloud Storage bucket (as specified by `GCS_OUTPUT_PATH` in the config). Only Evidence types with the `copyable` property will actually be copied into Cloud Storage.

CHAPTER 2

Developer documentation

2.1 Contributing

2.1.1 Before you contribute

We love contributions! Read this page (including the small print at the end).

Before we can use your code, you must sign the Google Individual Contributor License Agreement (CLA), which you can do online. The CLA is necessary mainly because you own the copyright to your changes, even after your contribution becomes part of our codebase, so we need your permission to use and distribute your code. We also need to be sure of various other things—for instance that you'll tell us if you know that your code infringes on other people's patents. You don't have to sign the CLA until after you've submitted your code for review and a member has approved it, but you must do it before we can put your code into our codebase. Before you start working on a larger contribution, you should get in touch with us first through the issue tracker with your idea so that we can help out and possibly guide you. Coordinating up front makes it much easier to avoid frustration later on.

We use the github fork and pull review process to review all contributions. First, fork the Turbinia repository by following the github instructions. Then check out your personal fork:

```
$ git clone https://github.com/<username>/turbinia.git
```

Add an upstream remote so you can easily keep up to date with the main repository:

```
$ git remote add upstream https://github.com/google/turbinia.git
```

To update your local repo from the main:

```
$ git pull upstream master
```

Please follow the Style Guide when making your changes, and also make sure to use the project's pylintrc and yapf config file. Once you're ready for review make sure the tests pass:

```
$ pip install -e .[dev]
$ pip install -r dfvfs_requirements.txt
$ python ./run_tests.py
```

NOTE: If you are developing in a hybrid/local setup, you need to set the `PROMETHEUS_PORT` and `PROMETHEUS_ADDR` to `None` in your config file in order to run Turbinia.

Commit your changes to your personal fork and then use the GitHub Web UI to create and send the pull request. We'll review and merge the change.

2.1.2 Code review

All submissions, including submissions by project members, require review. To keep the code base maintainable and readable all code is developed using a similar coding style. It ensures:

The code should be easy to maintain and understand. As a developer you'll sometimes find yourself thinking hmm, what is the code supposed to do here. It is important that you should be able to come back to code 5 months later and still quickly understand what it supposed to be doing. Also for other people that want to contribute it is necessary that they need to be able to quickly understand the code. Be that said, quick-and-dirty solutions might work in the short term, but we'll ban them from the code base to gain better long term quality. With the code review process we ensure that at least two eyes looked over the code in hopes of finding potential bugs or errors (before they become bugs and errors). This also improves the overall code quality and makes sure that every developer knows to (largely) expect the same coding style.

2.1.3 Style guide

We primarily follow the Google Python Style Guide. Various Turbinia specific additions/variations are:

- Using two spaces instead of four
- Quote strings as ' or """ instead of "
- Textual strings should be Unicode strings so please include `from __future__ import unicode_literals` in new python files.
- Use the format() function instead of the %-way of formatting strings.
- Use positional or parameter format specifiers with typing e.g. '{0:s}' or '{text:s}' instead of '{0}', '{}' or '{:s}'. If we ever want to have language specific output strings we don't need to change the entire codebase. It also makes is easier in determining what type every parameter is expected to be.
- Use "cls" as the name of the class variable in preference of "klass"
- When catching exceptions use "as exception:" not some alternative form like "as error:" or "as details:"
- Use textual pylint overrides e.g. "# pylint: disable=no-self-argument" instead of "# pylint: disable=E0213". For a list of overrides see: http://docs.pylint.org/features.html

2.1.4 The small print

Contributions made by corporations are covered by a different agreement than the one above, the Software Grant and Corporate Contributor License Agreement.

2.2 Developing new Turbinia Tasks and Evidence types

2.2.1 It's easy!

Creating new Tasks for Turbinia is fairly easy, and if your Task is simple (like just executing an external command) it should only take a few lines of real code along with a bit of boiler-plate code, and a few extra lines to connect things together.

2.2.2 Before you start

- Check out the *How it Works* page to see how the different components work within Turbinia.
- Make sure to follow the Turbinia developer contribution guide.

2.2.3 Task code

Task Setup

The Worker which runs the tasks handles the following things before you even get to the `run()` method where most of our code will go:

- Running any pre- or post-processors that need to run to prepare the Evidence.
- If the Evidence is file-based, the pre-processor will add the path to the processed evidence in `evidence.local_path`.
- Setting up temporary directories (available as `self.output_dir` and `self.tmp_dir`).
- Preparing a TurbiniaResult object to save results into.

Task execution

To see a relatively simple example of the code required for a new Task, see this pull request. This simply executes the strings binary on Disk-based Evidence types.

Here is the bulk of the actual Task code for the Ascii Strings Task:

```
    # Create the new Evidence object that will be generated by this Task.
    output_evidence = TextFile()
    # Create a path that we can write the new file to.
    base_name = os.path.basename(evidence.local_path)
    output_file_path = os.path.join(
        self.output_dir, '{0:s}.ascii'.format(base_name))
    # Add the output path to the evidence so we can automatically save it
    # later.
    output_evidence.source_path = output_file_path

    # Generate the command we want to run.
    cmd = 'strings -a -t d {0:s} > {1:s}'.format(
        evidence.local_path, output_file_path)
    # Add a log line to the result that will be returned.
    result.log('Running strings as [{0:s}]'.format(cmd))
    # Actually execute the binary
    self.execute(
        cmd, result, new_evidence=[output_evidence], close=True, shell=True)
```

This is mostly self explanatory from the comments, but the line that needs a little more explaining is this one:

```
    self.execute(
        cmd, result, new_evidence=[output_evidence], close=True, shell=True)
```

This will:

- Run the command as specified
- Set the output evidence to be saved
- Save the stdout and stderr in the results object specified
- Close the Result in preparation for Task completion

Task Finalization and Saving Results

Before a Task completes and returns, the Result object must be "closed" which finalizes the results in preparation for them to be returned to the server. Closing a Result does a few things like set Task stats, save all of the output files, and run the post-processor to free up the Evidence (e.g. unmount disks, etc). In the above example of `self.execute()`, `close=True` is set, which will tells the method to handle closing the results. If you have other external commands that you want to run and save the output from, you should not close the results until after these are all complete (i.e. don't set `close=True` in `self.execute()` in this case). If you are not calling the `execute()` method and implicitly closing the results that way, you'll need to close them similar to this:

```
result.close(self, success=True, status='My message about the Task status')
```

One important parameter that was not set in this example call of `self.close()` is `save_files`. It takes a list of file paths that you want to save (no need to add the files you linked to the Evidence earlier, it will save those automatically). This is used for non-Evidence files that you want to save (for example log files).

If you want to write files from your Task, you should do this relative to the `self.output_dir`. If you have temporary files you want to write, you can write these to `self.tmp_dir`. These directories are unique for the given Task execution.

The `run()` method should return the `result` object which will be serialized and returned to the server along with the associated Evidence that may have been created. The new Evidence created and included in the results will be checked by the Task Manager to see if there are other Jobs/Tasks that should be scheduled to process it.

Pre/Post-Processing

Each Task can set the Evidence state that is required (e.g. mounted, attached, etc) prior to execution by setting the state in `Task.REQUIRED_STATES`. Each Evidence object can specify which states it supports in the `Evidence.POSSIBLE_STATES` list attribute for that Task (e.g. see the `GoogleCloudDisk` possible states here). These states are set up by the pre-processors and then after the Task is completed, the post-processor will tear down this state (e.g. unmount or detach, etc). For more details on the different states and how the pre/post-processors will set these up at runtime, see the `Evidence.preprocess()` docstrings.

Evidence Paths

As mentioned above, the pre-processor that runs before the Task is executed will set the path `evidence.local_path` to point to the local Evidence. If the Task generates any new Evidence objects, you must set the `.source_path` attribute for that object before you add it to the results. The `.source_path` is the original path the Evidence is created with and the `.local_path` is the path to access the Evidence after any pre-processors are

run (e.g. the path it was mounted on if it was mounted, etc). See the docstrings for these attributes in the Evidence object for more details on the differences, but in summary, Tasks should use `.local_path` to process the incomming Evidence and `.source_path` for newly created Evidence.

Recipe configuration

Tasks can also specify a set of variables that can be configured and set through *recipes*. This allows users to predefine set configurations for the runtime behavior of Tasks along with which Jobs/Tasks should be run. Each Task has a `TASK_CONFIG` dictionary set at the object level that defines each of the variables that can be used along with the default values that the Task will use when the recipe does not specify that variable, or there is no recipe used. See the Plaso Task `TASK_CONFIG` as an example. Tasks can access these variables by referencing the dictionary at `self.task_config`.

2.2.4 Boilerplate and Glue

The only two interesting bits for the Job definition in `turbinia/jobs/strings.py` are this one that sets the allowable input and output Evidence types for the Task (so the Task Manager knows what kinds of Tasks to schedule):

```
evidence_input = [RawDisk, GoogleCloudDisk, GoogleCloudDiskRawEmbedded]
evidence_output = [TextFile]
```

And this one, which just sets up the Tasks for both Task types (Ascii and Unicode):

```
  tasks = [StringsAsciiTask() for _ in evidence]
  tasks.extend([StringsUniTask() for _ in evidence])
  return tasks
```

In this case we have two separate Tasks that we are executing for the Job, but it's possible that there could be more or less depending on how much you want to split it up. Then you just need to add a reference to the new job in `turbinia/jobs/__init__.py`.

2.2.5 Reporting

Tasks can return report data in Markdown format by adding it as a string to `result.report_data`. If high priority findings are found, you can change `result.report_priority`. Priorites are 0 - 100, and the highest priority is 0. This will affect the ordering in the report, and if the priority is a value less than what is set with `--priority_filter` (i.e. a higher priority), then the full report data will be printed out when `--full_report` is specified.

Tasks can use the helper methods in turbinia.lib.text_formatter to help format the text with formatting like bold and code. Note that when setting headings in a task report, do not use heading1 through heading3 because these are used in other sections of the report, but you can use heading4 and above.

2.2.6 Tips

- If possible, set a meaningful `status` message that summarizes the Task execution or output. This can be done by either by setting the `status` parameter when calling `result.close()`, or by explicitly setting the `result.status` attribute. This is the line of output that shows up for each task when running `turbiniactl status`. If a Task has a low `report_priority`, then the full report data will not show up in the `turbiniactl status`, and so the status may be the only place that Task info will bubble up in the output by default, so setting it to something useful can be important.

- If your Task executes an external command that can generate a log file, it's helpful to specify the appropriate flags to generate this and then automatically save it by setting the `log_files` parameter when calling `self.execute()`. Additionally, if there are flags that control the verbosity of this log file, it's helpful to check the `config.DEBUG_TASKS` config parameter and log accordingly, and this way all tasks can generate debug output when this is configured.

2.2.7 Testing

There is a `TestTurbiniaTaskBase` object that task tests can sub-class for relatively easy testing of the basic run method. See the photorec test for a simple example. For a task test with reporting output see the sshd test as an example.

2.2.8 Notes

- The reason we separate out the strings processing into two separate Tasks is so we can do them in parallel and save on wall-time.
- One caveat about Task development is that it is possible to create a cycle in the Task Manager by generating Evidence types that your Task (or any of its parent's tasks) also listens to. Check out the Job and Evidence graph generator if you want to verify that there aren't any cycles in the graph.

2.3 Developing on a local Turbinia setup (no cloud required)

See *here* on how to setup the local Turbinia stack with Docker.

After you have the local stack up and running a usual development cycle would look like below.

2.3.1 Before you start

- Check out the *How it Works* page to see how the different components work within Turbinia.
- Make sure to follow the Turbinia developer contribution guide.

Step 1

Fork Turbinia on github and create a new feature branch to work on.

```
$ git clone https://github.com/[your-github-user-id]/turbinia.git
$ git checkout -b my-new-feature
$ cd turbinia
```

Step 2

Add some awesome new feature, maybe develop a new task?

Step 3

Rebuild Turbinia server and/or worker Docker images.

```
$ docker build -t turbinia-worker-dev -f docker/worker/Dockerfile .
```

Step 4

Change the `image:` location in the `docker/local/docker-compose.yml` file to point to your localy build image (eg turbinia-worker-dev).

Step 5

Let's bring up the local Turbinia stack

```
$ docker-compose -f ./docker/local/docker-compose.yml up
```

Step 6

Let's process evidence to test your setup, in this case a Chrome Browser history file but you will likely want to use specific evidence to test your new functionality.

```
$ curl https://raw.githubusercontent.com/obsidianforensics/hindsight/master/tests/
↪fixtures/profiles/60/History > History
$ tar -vzcf ./evidence/history.tgz History
$ docker exec -ti turbinia-server turbiniactl compresseddirectory -l /evidence/
↪history.tgz
```

This will create a task in Turbinia to process the evidence file. A task ID will be returned and we can query the status with below command.

```
$ docker exec -ti turbinia-server turbiniactl -a status -r␣
↪b998efb5dcb64949963d9c72ba143c1a
```

Step 7

Test and debug your new feature and repeat steps 1-7 until satisfied.

2.4 Developing with Visual Studio Code (no cloud required)

2.4.1 Introduction

This procedure will get you up and running with a Visual Studio Code environment for Turbinia development. The provided configuration files will create a development container containing all dependencies, pylint and yapf correctly setup and launch configurations for both client, server and workers. With this setup it is possible to initiate full Turbinia debug sessions including breakpoints, watches and stepping.

You can set Visual Studio Code up to run a local stack (using redis and celery) or use a hybrid GCP stack (using pubsub, datastore and cloud functions). We advice you to run a local stack if you don't need to debug or develop Turbinia GCP functionality.

2.4.2 Before you start

- Check out the *How it Works* page to see how the different components work within Turbinia.
- Make sure to follow the Turbinia developer contribution guide.

Step 1 - Install required software

Prepare your OS:

- Install Visual Studio Code and install the Remote Development extension pack.
- Install Docker on your operating system (eg Docker Desktop on OSX)

Step 2 - Fork Turbinia

Fork Turbinia on Github and create a new feature branch to work on.

```
$ git clone https://github.com/[your-github-user-id]/turbinia.git
$ git remote add upstream https://github.com/google/turbinia.git
$ git checkout -b my-new-feature
$ cd turbinia
```

Step 3 - Open in Visual Studio Code

Open the folder in vscode and choose to "Reopen in Container" when asked (vscode will see the `.devcontainer` folder in the turbinia cloned source tree). Vscode will build your Turbinia development container and that will take a couple of minutes.

Note: If vscode does not ask you to reopen in a container you need to verify you have installed the Remote Development extension!

Note: The instructions contain shell commands to execute, please execute those commands in the vscode terminal (which runs in the development container) and not in a terminal on your host!

Continue with Step 4 for a local Turbinia setup or Step 5 for a GCP hybrid setup.

Step 4 - Local Turbinia setup

The local turbinia setup will use redis and celery. Let's create the configuration file for this setup.

Note: This command needs to be executed in the vscode terminal!

```
$ sed -f ./docker/vscode/redis-config.sed ./turbinia/config/turbinia_config_tmpl.py >␣
↪~/.turbiniarc
```

Let's verify the installation in Step 6.

Step 5 - GCP hybrid Turbinia setup

Follow the 'GCP Setup' section here and setup Cloud Functions, a GCE bucket, Datastore and PubSub.

- Create a pubsub topic, eg 'turbinia-dev'
- Create a GCE storage bucket with a unique name

Create the Turbinia hybrid configuration file.

Note: This command needs to be executed in the vscode terminal!

```
$ sed -f ./docker/vscode/psq-config.sed ./turbinia/config/turbinia_config_tmpl.py > ~/
↪.turbiniarc
```

Edit the configuration file `~/.turbiniarc` and set below variables according to the GCP project you are using. Make sure all values are between quotes!

```
TURBINIA_PROJECT = '[your_gcp_project_name]'
TURBINIA_REGION = '[your_preferred_region]'  eg 'us-central1'
TURBINIA_ZONE = '[your_preferred_zone]'  eg 'us-central1-f'
PUBSUB_TOPIC = '[your_gcp_pubsub_topic_name]'  eg 'turbinia-dev'
BUCKET_NAME = '[your_gcp_bucket_name]'
```

Setup authentication for the GCP project.

Note: These commands need to be executed in the vscode terminal!

```
$ gcloud auth login
$ gcloud auth application-default login
```

Deploy the Google Cloud Functions

Note: This command needs to be executed in the vscode terminal!

```
$ PYTHONPATH=. python3 tools/gcf_init/deploy_gcf.py
```

Step 6 - Turbinia installation verification

Let's verify that the GCP hybrid setup is working before we start developing and debugging. We are going to start a server and worker in separate vscode terminals and create a Turbinia request in a third. Open up 3 vscode terminals and execute below commands.

Note: These commands need to be executed in the vscode terminal!

Terminal 1 - Start server

```
$ python3 turbinia/turbiniactl.py -S server
```

Terminal 2 - Start worker For a local setup

```
$ python3 turbinia/turbiniactl.py -S celeryworker
```

For a GCP hybrid setup

```
$ python3 turbinia/turbiniactl.py -S psqworker
```

Terminal 3 - Fetch and process some evidence

```
$ curl https://raw.githubusercontent.com/obsidianforensics/hindsight/master/tests/
↪fixtures/profiles/60/History > History
$ tar -vzcf /evidence/history.tgz History
$ python3 turbinia/turbiniactl.py compresseddirectory -l /evidence/history.tgz
$ python3 turbinia/turbiniactl.py -a status -r [request_id]
```

This should process the evidence and show output in each terminal for server and worker. Results will be stored in `/evidence` and in the GCS bucket.

Step 8 - Debugging example

When you are developing code you want to be able to step through your code and inspect variables. We can do this by running the different launch profiles provided. Visual Studio Code launch profiles are provided for server, celeryworker, psqworker and client requests.

As a small example we will change the version string of Turbinia.

- Edit `turbinia/__init__.py` and change around line 24 the version string from "unknown" to "iamrad" (The person who wrote this is old...).
- Set a breakpoint on the line you edited.
- Save the file.
- We want to hit this code path while running the server so we will start the 'Turbinia Server' launch profile.
- Click on the 'Run' icon on the left hand side (*not* the menu at the top!).
- Choose the 'Turbinia Server' profile.
- Hit the green play button to start debugging.
- The server will start and vscode will break when it hits your edited version string.
- Inspect the variables and step through your code at will.

It's important to understand that if you are developing and debugging more complex code paths that you will almost certainly run a combination of vscode terminal and vscode launch profile server/workers/client. You need to use the correct launch profile to hit the breakpoints depending on where you have set them.

2.5 Turbinia SRE Guide

2.5.1 Introduction

This document will cover topics to manage Turbinia infrastructure in cloud and includes Prometheus/Grafana monitoring stack how tos.

2.5.2 GCP

GCP management script: update-gcp-infra.sh

Preparation

The GCP stack is managed with the update-gcp-infra.sh management script. This script can be run from any workstation or cloud shell.

- Clone the Turbinia repo or download the tool directly.
- workstation only: Install the gcloud cli tool
- workstation only : Authenticate to your GCP project:
  - `$ gcloud auth application-default login`
- workstation only : set the correct GCP project:
  - `$ gcloud config set project {PROJECT_NAME}`

Stop and start the Turbinia stack

Turbinia uses Pub/Sub to distribute jobs and tasks to server and workers. We must first stop the server before stopping the workers. Before stopping the server we need to make sure the server is not handling any open tasks/requests anymore.

Stop infrastructure

- Check the server logs to see if it's waiting for any tasks to complete:
  - $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -c logs -s

---

NOTE

If you are not running from your workstation you can remove the `-s`.

- If no tasks are open, shutdown the Turbinia infrastructure. The management script will make sure this is done properly.
  - $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -c stop

---

NOTE

Workers are shut down asynchronously so it will take a while to show them as "STOPPED".

Start infrastructure

- The management script will make sure startup is done in the correct order:
  - $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -c start

---

NOTE

workers are started up asynchronously so it will take a while to show them as "RUNNING".

- Check the server and worker logs to see if they came back up as expected.
  - $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -c logs -s

---

NOTE

If you are not running from your workstation you can remove the `-s`.

Configuration update

The Turbinia configuration is base64 encoded as a metadata value (TURBINIA_CONF) attached to the GCE ContainerOS instances. This is written to disk at boot time and used by the server and worker docker containers.

Load the new configuration into the Turbinia stack with the following commands:

- Stop the stack (see *here*).
- $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -f [path-to-new-config-file] -c update-config
- Start the stack (see *here*).

Environment variable update

The Turbinia stack (including monitoring components) sets some configuration parameters through containerOS environment variables.

- Stop the stack (see *here*).
- $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -k [env-variable-name] -v [env-variable-value] -c update-config
- Start the stack (see *here*).

Docker image update

Turbinia runs as a docker image (separate image for server and worker) that are built and stored in the osdfir-registry project. When you are ready to release a new version you need to reload the Turbinia stack with that new docker image. Please make sure you also *update the configuration* if needed (depending on code changes).

Update to `latest`

This is the most common scenario. When releasing a new version of Turbinia in github, a production docker image will be built for both server and worker and tagged with the `latest` tag.

- Stop the stack (see *here*).
- Start the stack (see *here*).
  - When starting the stack again ContainerOS will poll the `osdfir-registry` and pull the `latest` image of worker and server.
  - Verify the running version by viewing the server and workers logs.

Update the docker tag

Some scenarios require deploying a specific version of Turbinia. Each docker image is tagged with the actual release tag from github (eg 20210606) and can be loaded.

- Stop the stack (see *here*).
- Configure a custom tag (instead of `latest`) for server and worker runners
  - Verify the required tag is actually successfully built and available in the registry for both server and worker.
  - $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -t [tag] -c change-image

- Start the stack (see *here*).
  - When starting the stack again ContainerOS will poll the osdfir-registry and pull the latest image of worker and server.
  - Verify the running version by viewing the server and workers logs.

Spin-up more Turbinia workers

Sometimes we need more capacity to process tasks and we want to spin-up new workers.

- Go to "VM instances" page under the Compute Engine tab in your cloud project.
- Filter instances on the current Turbinia instance ID.
  - Only use the numbers, leave out the 'turbinia-' part.
- Find the worker with the highest number at the end of the name (eg turbinia-worker-1aa1aaaa1a1a1111 **-4**)
- Open the detail page of that worker.
- Click the top button "Create Similar".
- Verify that the "Name" field correctly increased the last number in the name.
- Scroll down and click "Create".
- Verify the logs of the new worker for correct instantiation.
  - $ ./update-gcp-infra.sh -i {INSTANCEID} -z {ZONE} -c logs -s

---

NOTE

If you are not running from your workstation you can remove the `-s`.

2.5.3 Turbinia Metrics setup and management

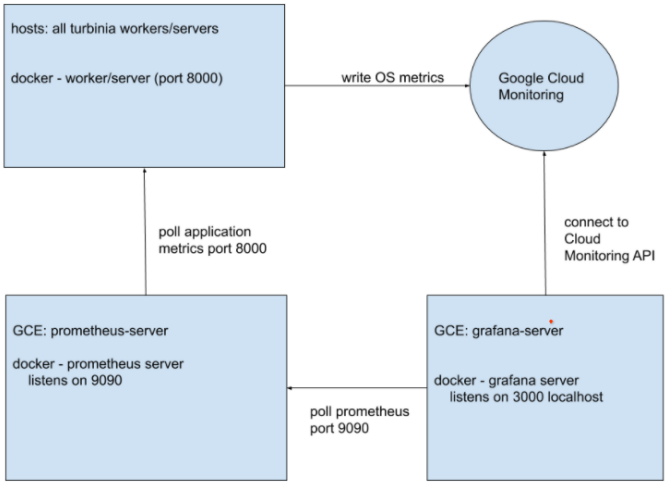
Monitoring configuration

In order to interact with the current Turbinia Worker(s) and Server to generate metrics, you can include the following in Turbinia configuration file:

```
# Prometheus monitoring config
PROMETHEUS_ENABLED = True
PROMETHEUS_ADDR = '0.0.0.0'
PROMETHEUS_PORT = 9200
```

GCP setup

The following is the same for both prod and dev cloud projects.



gcp_graph

Google Monitoring metrics are enabled on all container machines and provide the OS metrics.

- `google-monitoring-enabled` set to `true` in GCE configuration of container worker/server instances.
- All Turbinia machines are labeled with 'turbinia-prometheus' for auto-discovery by prometheus.

Turbinia server and workers are instrumented with prometheus code and expose application metrics.

- each server and worker listens on port 9100 on the private IP only for system metrics and port 9200 for application metrics.

Prometheus

- Prometheus will scrape the OS metrics and application metrics from each host and expose them on port 9090 for grafana
- Listens on port 9090
- Configuration: `/etc/prometheus/prometheus.yml`and *here*
- SSH:
  - `ssh -i ~/.ssh/google_compute_engine -L 11111:localhost:9090 prometheus-server-1aa1aaaa1a1a1111`
- Dashboard is on https://localhost:11111/
  - As the server has a self signed cert, please use 'thisisunsafe' to continue past the error message.

Prometheus configuration GCP

```
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is␣
↪every 1 minute.
  # scrape_timeout is set to the global default (10s).
  external_labels:
      environment: turbinia-gcp_node

scrape_configs:
  - job_name: 'turbinia-gcp'
```

(continues on next page)

(continued from previous page)

```
    gce_sd_configs:
        # The GCP Project
        - project: {PROJECT_NAME}
          zone: {ZONE}
          filter: labels.turbinia-prometheus=true
          refresh_interval: 120s
          port: 8000
```

The Turbinia hosts that need to be scraped have the label 'turbinia-prometheus' set by the Terraform scripts and are auto discovered by Prometheus.

Grafana

- Grafana pulls metrics from Google monitoring (OS metrics) and application metrics from Prometheus.
- Grafana displays dashboard for both the OS and Turbinia application metrics.
- Datasource 1: Google Monitoring of gcp project
- Datasource 2: `http://prometheus-server-1aa1aaaa1a1a1111:9090/`
- Listens on port 3000
- Current the username is **"admin"** and password is generated by terraform when it deploys Grafana.

WARNING: You are responsible for securly storing the Grafana admin password after terraform deployment is complete.

- SSH access
  - `ssh -i ~/.ssh/google_compute_engine -L 11111:localhost:3000 Grafana-server-1aa1aaaa1a1a1111`
- Dashboard is on https://localhost:11111/
  - As the server has a self signed cert, please use 'thisisunsafe' to continue past the error message

Importing a new dashboard to Grafana

- Login to *Grafana*
- Click the "+" sign on the left sidebar and then select "import".
- Then copy/paste the json file from the dashboard you want to import and click "Load".

Exporting a dashboard from Grafana

- Login to *Grafana*
- From the dashboard, select the "dashboard Setting" on the upper right corner
- Click on "JSON Model" and copy the contents of the textbox.
- In order to import this to another dashboard, follow *these steps*.

How to update prometheus config

To update the prometheus config file or add new rules to prometheus:

- SSH to the host VM as mentioned *here* .
- To change the config:
  - Change and save the contents of `/etc/prometheus/prometheus.yml`
  - Restart the prometheus container by running `docker restart {CONTAINER_ID}.`
    - To find the container_id run `docker ps`.
- To add new prometheus rules:
  - Create or update an existing rule file. Prometheus docs have great tips on how to write recording rules.
  - Update the config file `/etc/prometheus/prometheus.yml` and add the path to the rule file under `rule_files` section.
  - Copy the rule file to the prometheus container by running `docker cp prometheus.rules.yml {CONTAINER_ID}:/etc/prometheus/prometheus.rules.yml`
  - SSH to the prometheus container `docker exec -ti {CONTAINER_ID} sh` and check if the rule file is correct by running:
    - `promtool check rules rule_file_path`
  - Finally to apply the rules, you need to send a `SIGHUP` signal to prometheus process by running:
    - `killall -HUP prometheus`

For more details please visit Turbinia design documents.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search