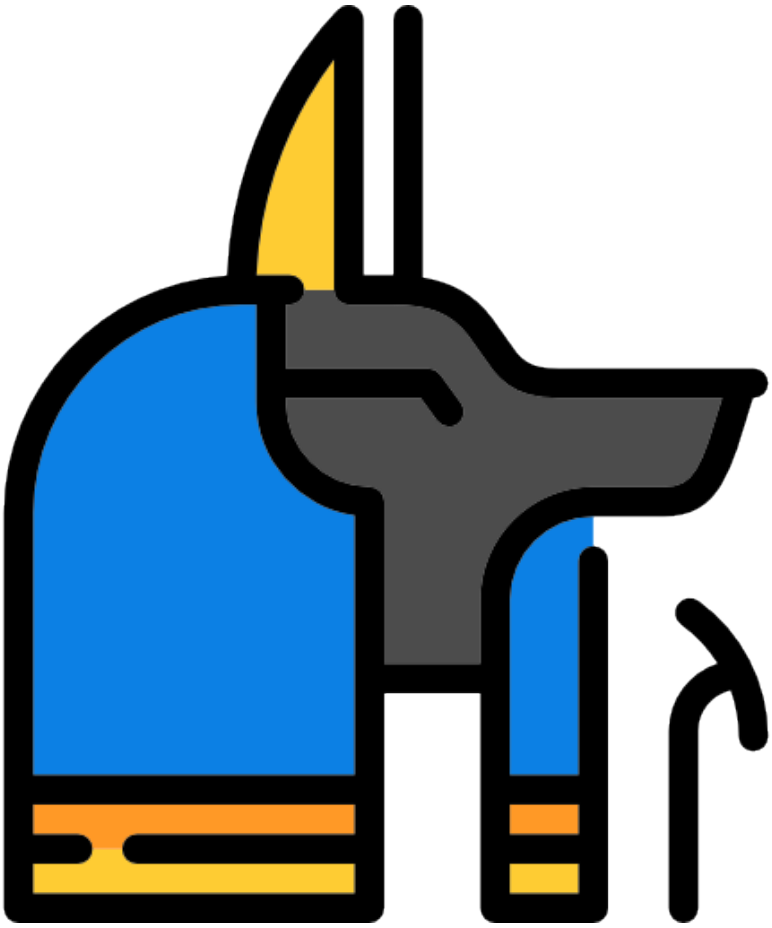
Anubis Design Doc



Author: John Cunniff

Version: v2.2.3

Contents

1 Overview

1.1 Elevator Pitch

At its core, Anubis is a tool to give Intro to OS students live feedback from their homework assignments while they are working on them. Instead of having students submit a patch file, through github classrooms each student will have their own private repo for every assignment. The way students then submit their work is simply by submitting before the deadline. Students can then push, and therefore submit as many times as they would like before the deadline.

When a student pushes to their assignment repo, a job is launched in the Anubis cluster. That job will build their repo, run tests on the results, and store the results in the datastore.

Students can then navigate to the anubis website, where they will sign in through NYU SSO. From there, they will be able to see all the current and past assignments, and submissions for their classes. They are able to view all relevant data from their build and tests for a given submission. There they can request a regrade, there by launching a new submission pipeline. While the submission still being processed, the frontend will poll the API for updates. In this, the frontend will be constantly updating while the submission is being processed, giving a live and interactive feel to the frontend. Once a submission is processed Anubis will show the students logs from their tests, and builds along with which tests passed and which failed.

New in version v2.2.0, there is now the Anubis Cloud IDE. Using some kubernetes magic, we are able to host theia servers for individual students. These are essentially VSCode instances that students can access in the browser. What makes these so powerful is that students can access a terminal and type commands right into a bash shell which will be run in the remote container. With this setup students have access to a fully insolated and prebuilt linux environment at a click of a button.

1.2 Motivations

The purpose of this paper is to both explain the design, and the features of the Anubis autograder.

2 Services

2.1 Traefik

For our edge router, we use traefik. Traefik will be what actually listens on the servers external ports. All external traffic will be routed through Traefik. Above all else, we will be using Traefik's routing features to handle the Ingress of requests.

Traefik lets us do some spicy and powerful stuff. We can route requests to different services based off predefined rules, and even change requests as they pass through. This makes it so that we can have both the static store and api on the same domain. The rules are set up such that every request that starts with a path of `/api` goes to the api service.

By leveraging these features of Traefik, we can make it appear that the services work different when being accessed externally. Namely, the basic authentication for certain paths (and therefore services).

> One thing to note here is that when being accessed from within the cluster, none of these rules apply as we would be connecting directly to services.

2.2 API

The API is the backbone of anubis. It is where all the heavy lifting is done. The service relies on both the elasticsearch and mariadb data stores to maintain state.

2.2.1 Zones The API is split into two distinct, and uniquely treated zones. There is a `public` and a `admin` zone. All endpoints for Anubis fall within one of these zones.

These zones are simply paths that are treated differently depending on where the request is external. Namely, for the admin zone external requests will require you to be marked as an admin. By adding this simple level of authentication, we can lock down a section of the more sensitive API to only those authenticated from the outside.

2.2.2 Responsibilities The Anubis API is responsible for handling most basic IO, and state managing that happens on the cluster. Some of this includes:

- Authenticating users
- Providing Class, Assignment, and Submission data to the frontend
- Handling github webhooks
- Handling reports from the submission pipeline cluster
- Handling regrade requests
- Initializing new IDE sessions

2.2.3 SSO Authentication To authenticate with the api, a token is required. The only way to get one of these tokens is through NYU Single Sign On. By doing this, we are outsourcing our authentication. This saves a whole lot of headaches while being arguably more secure than if we rolled our own.

In implementation, the frontend loads it will attempt to authenticate with the API. If there is a stale or broken token in the current cookies, the frontend will redirect users to the NYU login page. Given that they authenticate there, they will be redirected back to the API, where we will provide them with a token. From there, they will be logged into Anubis.

All of this is about 20 lines on our end. All that is necessary are some keys from NYU IT.

2.3 Submission Pipeline

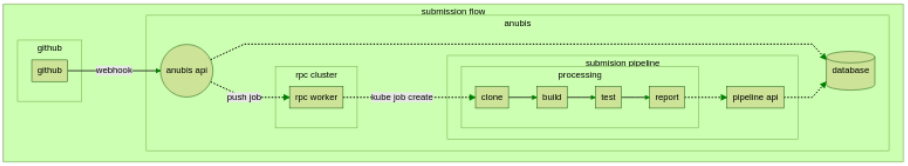
2.3.1 Kube Job A given submission pipeline is of the form of a Kubernetes Job. These jobs have some built in assurances. A job is configurable to continue to launch new containers if a Pod fails.

2.3.2 Submission State Reporting At each and every stage of a submission pipeline, the job will report to the api with a state update. This state is in the form of a string that describes what is currently happening at that moment. This data can then be passed along to a user that is watching their pipeline be processed live.

Each unique per-assignment pipeline is packaged in the form of a docker image.

> See Creating a new Assignment
>
> An error at any stage of the submissions pipeline will result in an error being reported to the API, and the container exiting.

2.3.3 Stages It is important to note that at each stage of the submission pipeline, we will be moving execution back and forth between two users. There will be the entrypoint program managing the container as user `anubis`. The `anubis` user will have much higher privileges than the `student` user. The `student` user will be used whenever executing student code. It will not have any level of access to anything from the `anubis` user.



Clone In this initial stage, we will pull the current repo down from github. After checking out the commit for the current submission, we will also delete the `.git` directory as it is not needed. Lastly we will `chown` the entire repo as `student:student`. This will then be the only place in the container that the student user can read or write to (other than /tmp of course).

Build At this stage we hand execution off to student code for the first time. We will be building the student code *as the* ***`student`*** *user*. The command for building the container will be specified by on a per-assignment basis. The stdout of the build will be captured by the `anubis` user.

Once built, a build report will be sent to the API, along with a state update for the submission. If the student is on the submission page, then they should be able to see the build info shortly thereafter.

Test Tests will be defined on a per-assignment basis. Again we are executing student code, as the student user.

After each test, we will return to the entrypoint program as the `anubis` user. We will then send a report of the last test before continuing to the next.

Once we reach the last test, we send off a separate notification to the API indicating the completion of the pipeline. It is at that point that the API marks the submission as processed.

2.4 Web Frontend

The frontend is designed to be a simple reflection of the backend data. Once authenticated, users will be able to see the classes they are a part of, current and past assignments, and all their submissions. With few exceptions, the frontend is a near one to one translation of the API's data models. Most pages will have a corresponding API endpoint. The data shown on that page will be in exactly the form of the API response.

The notable exceptions to this simplistic model would be the submission page, the regrade button, and the find-missing button.

The submission page will poll for new data if the submission is not marked as processed. As it sees new data, it will update what is displayed.

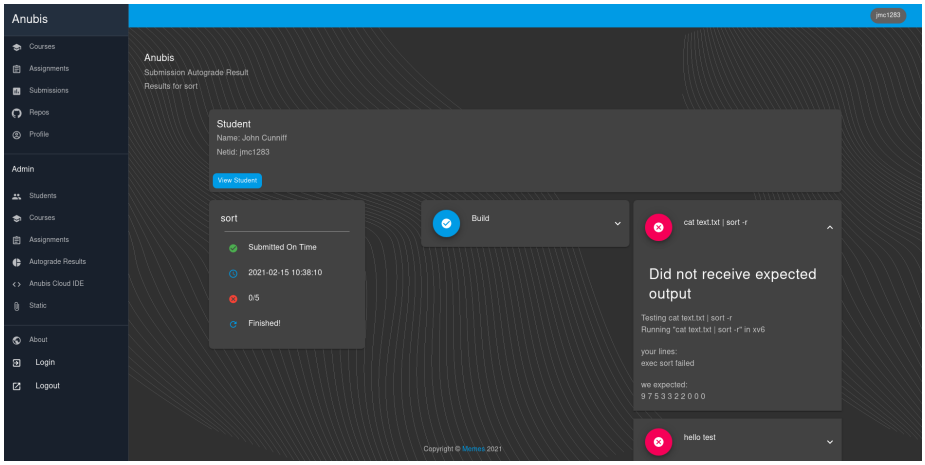
Located on the submission page, the purpose of the regrade button is to be a simple and easy way for users to request regrades. When clicked, the frontend will hit a special endpoint for requesting regrades. If successful, the submission will be re-enqueued in a submission pipeline.

On the submissions page, the find-missing button will trigger a server side update of the submission data. When the find-missing endpoint is hit, the API will use the graphql API for github to pull all the commits for all the known repos for that user. If it sees commits that were not previously seen (likely though github not delivering a webhook), then it will create and enqueue the new submissions.

2.4.1 Autograde Results Getting the autograde results is as easy as searching a name in the autograde results panel. You will be taken to a page like this where the calculated best submission is shown. The best submission is the most recent submission that passed the most tests.

The students see a reduced version of the panel showed to the TAs and professors. In the admin view, more data is displayed about the student. The students can see the status of the submission, and the build and test results.

> *Calculating the best submissions requires heavy IO with the database. For this, the autograde results are heavily cached. Results are updated hourly.*



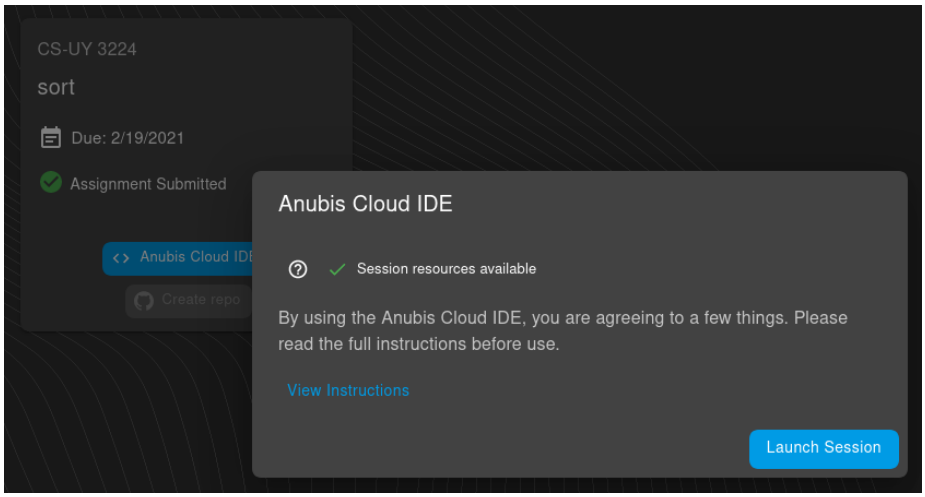
2.5 Anubis Cloud IDE

One of the more exciting new features is that of the cloud ide. Leveraging some magic that Kubernetes and containers give us, we can provide a fully insolated IDE environment for all ~130 concurrently.

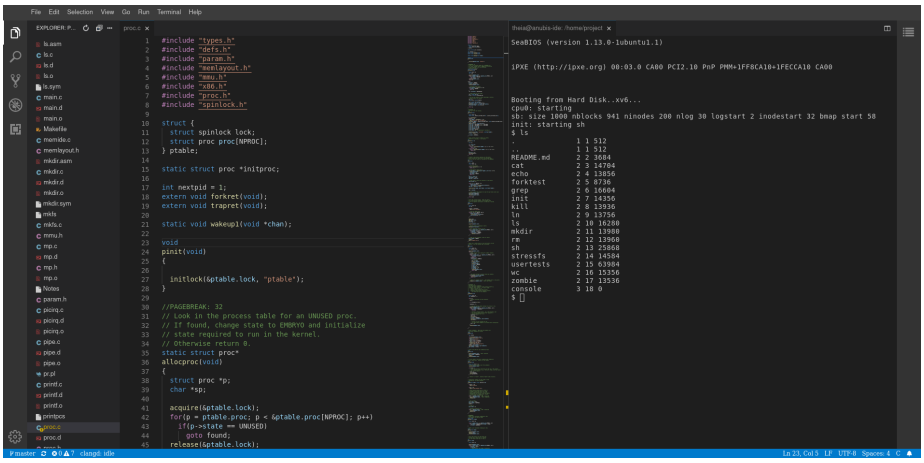
The Theia IDE is a basically a VSCode webserver. They have docker images for specific languages.

Something very special about theia is that we can actually build theia using a package.json. By doing this we are able to compile in only the plugins we actually need/use. This is very useful because we can bring the default theia-cpp image down from 4GiB to ~1.5GiB. This is still an annoyingly large image, but hey, it's not 4GiB.

2.5.1 IDE Frontend Once students have created their repo, they will be able to launch a theia session for a given assignment. At that time, we clone exactly what is in github into a theia pod.



Once their session pod has been allocated, then they can click the goto session button. It is at this point that their requests start to go through the theia proxy. Assuming all goes well with initializing the session, and starting the websocket connection, then students will have the following theia IDE.



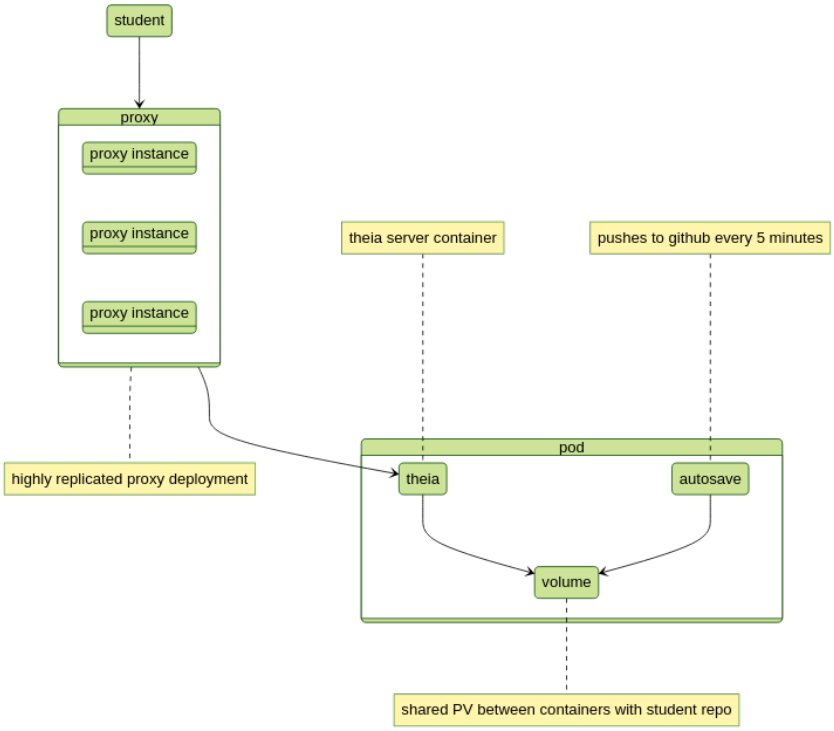
Something to point out here is that this looks, feels, and acts exactly as VSCode, but it is accessed over the internet. We can even load some VSCode plugins into the builds.

2.5.2 Theia Pod Design The pod design requires some distributed finesse. There are a couple of things about theia that make it so that we need to do some rather fancy things in Kubernetes to make the setup work.

Distributing and handling multiple theia severs and concurrent connections is the name of the game here. We need to be able to have a setup that scales well that is able to handle many users on the system at once.

The main thing that we need to handle is the fact that theia requires a websocket connection between the browser and theia server instance. When the pods are allocated, we note the ClusterIP in the database. Then when we need to initialize a client session, we use this saved ClusterIP to forward requests (both http and websockets) to the pod.

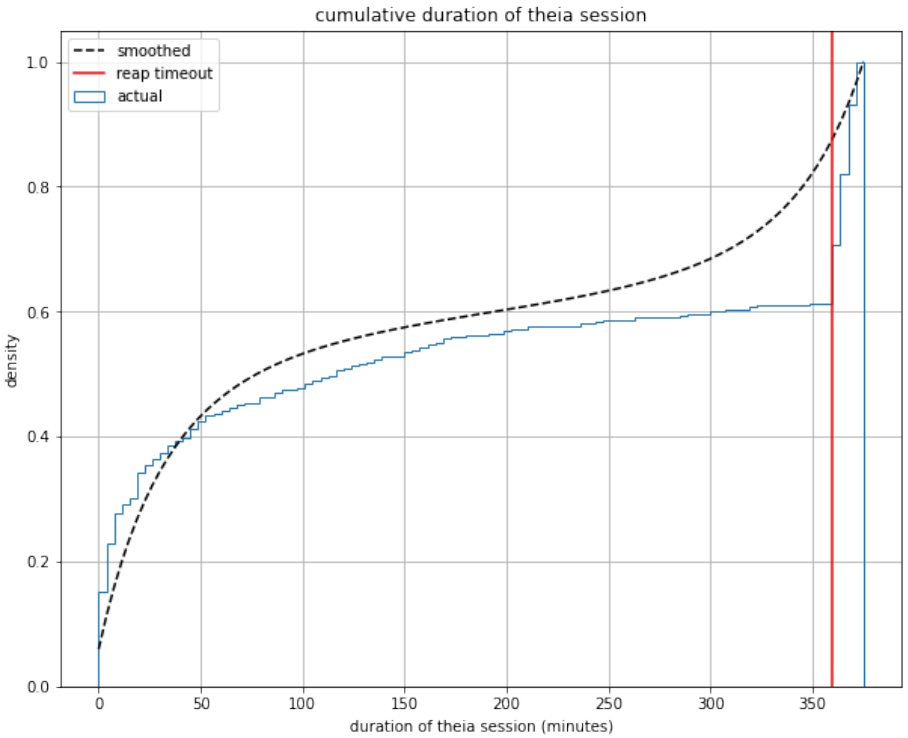
These pods are temporary. When the student is finished working (or after a timeout) we reclaim the resources by deleting the containers and data. Because of this, we needed some form of autosave. Saving is pushing to github. The issue we need to contend with is how do we have automatic commits and pushes to github without exposing a password or api token to the users. We handle this by having a second container whose only role is committing and pushing to github. The credentials live in that container, completely separate and inaccessible to the user who may be working in the other theia server container. These two containers are connected by a shared longhorn volume. This volume is relatively small (~50MiB). With this setup, we have autosave running in the background while being completely hidden from the user.



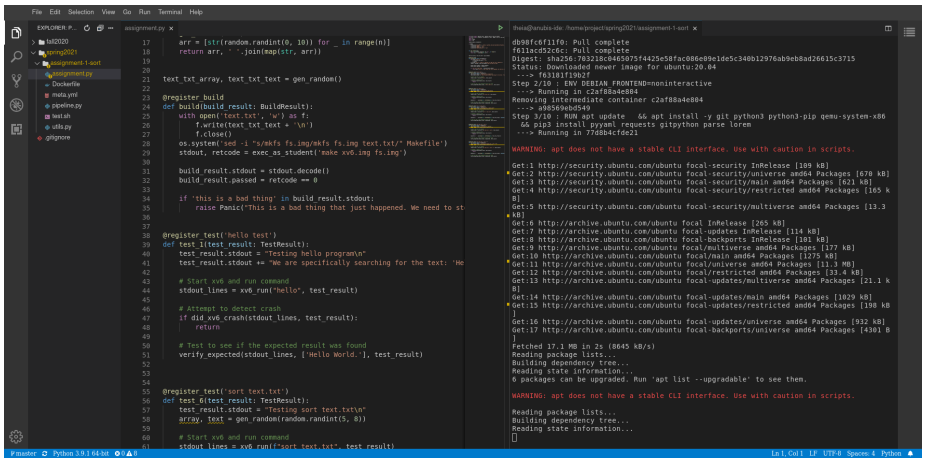
With these lightweight containerized theia servers, we are able to support significantly more concurrent users than if we had elected to implement a cloud vm solution. Because with containers, we do not need to virtualize hardware, the resources on the system for each user is significantly less. Given the resources that we have in the Space Cluster, we would be able to have maybe 20-30 concurrent users using cloud virtual machines. With our containerized theia, we can handle all ~130 students at the same time with room to breath.

2.5.3 Reclaiming Theia Resources The theia sessions are often forgotten about. Students often create an IDE server, work on it for a bit, then forget about it. Due to this, we have a time to live of 6 hours for each session. A cronjob runs every 5 minutes to look for and schedule delete for stale resources. We do provide a button in the anubis panel as a way for students to manually schedule their session for deletion. Even when we ask students to click this button when they are done, some still do not. In the following graph we can see an interesting experiment in student behavior. It shows a cumulative graph

showing the duration of a theia session for the final exam. Just about 40% of sessions hit the 6-hour timeout.



2.5.4 Management IDE Separate from the student xv6 IDE, there is also an admin build of theia that has some extra things. On the assignments page of the admin panel, any TA or professor can launch their own admin IDE. In this IDE, the assignment-tests repo is cloned instead of a student assignment repo. It has less networking restrictions, the anubis cli and even has docker running right in the pod. With this, TA's and professors can create, modify, and deploy assignments.



2.6 Datastores

2.6.1 Mariadb Anubis will use the OSIRIS Space Clusters existing MariaDB cluster. That cluster is made from bitnami's MariaDB chart. It runs with 3 read-only replication nodes, along with a main node that does read-write. The underlying MariaDB files are also backed up with a Longhorn persistent volume that has 3x replication in the cluster. That volume has daily snapshots. Redundancy is the name of the game here.

The precise data model that we will be using is a simple relational database. From the API, we will interface with Mariadb via SQLAlchemy.

Here is an example of a minimal mariadb deployment installed through the bitnami chart. Obviously for production we would want to change the password from anubis to something stronger.

```
# Create a minimal mariadb deployment in a mariadb namespace. On
# prod, the mariadb is in a separate namespace, so we do the same
# here.
echo 'Adding mariadb'
kubectl create namespace mariadb
helm install mariadb \
     --set 'auth.rootPassword=anubis' \
     --set 'volumePermissions.enabled=true' \
     --set 'auth.username=anubis' \
     --set 'auth.database=anubis' \
     --set 'auth.password=anubis' \
     --set 'replication.enabled=false' \
     --namespace mariadb \
     bitnami/mariadb
```

2.6.2 Elasticsearch Anubis uses elasticsearch as a search engine for logging, and event tracking. The elastic ecosystem has other tools like kibana and logstash for visualizing data, and logging. Other than the visualization features, elastic allows us to simply start throwing data at it without defining any meaningful shape. This allows us to just log events and data into elastic to be retrieved, and visualized later.

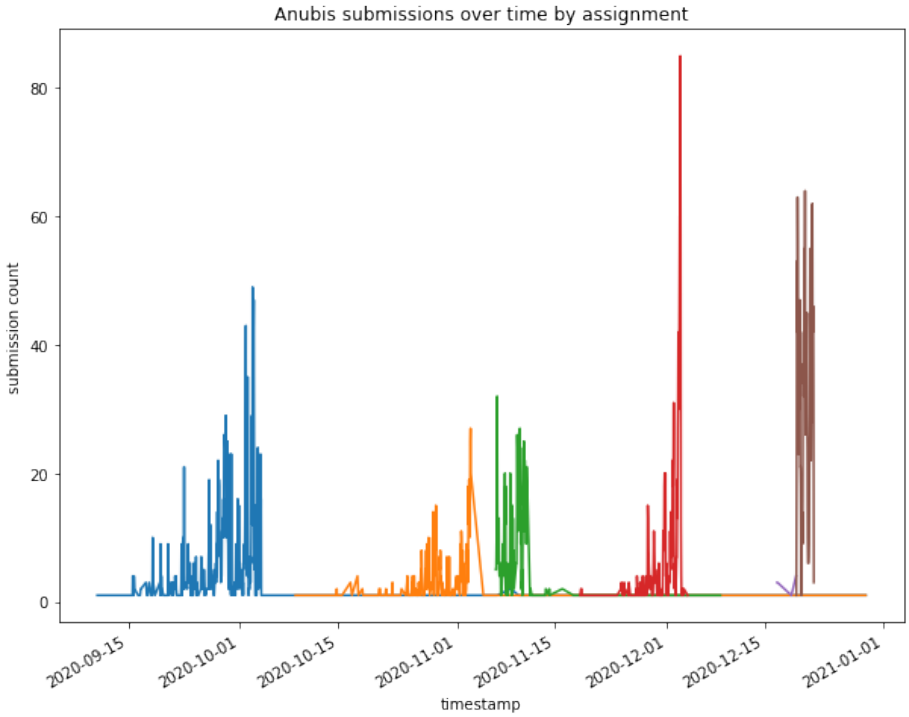
The Elasticsearch stack is pretty annoying to manage on its own. Installing through the bitnami chart is much easier.

```
# Install a minimal elasticsearch and kibana deployments
echo 'Adding elasticsearch + kibana'
kubectl create namespace anubis
helm install elasticsearch \
     --set name=elasticsearch \
     --set master.persistence.size=1Gi \
     --set data.persistence.size=1Gi \
     --set master.replicas=1 \
     --set coordinating.replicas=1 \
     --set data.replicas=1 \
     --set global.kibanaEnabled=true \
     --set fullnameOverride=elasticsearch \
     --set global.coordinating.name=coordinating \
     --namespace anubis \
     bitnami/elasticsearch
```

2.6.3 Kibana The Anubis autograder generates a lot of data. We have intense logging, and event tracking on every service. When something happens on the cluster, it will be indexed into elastic. Kibana is elastic's data visualization tool. It runs as a website that interfaces with the internal elasticsearch. Through kibana, we can stream live logs, view event data, and create meaningful visualizations.

The API sees when students start their homeworks (create their github repo), and when they are submitting (pushing to their repo). This data is indexed into elasticsearch, and visualized via kibana. In Anubis version one we were able to show graphs of when students were starting vs finishing their assignments. To no ones surprise, the majority of the class was starting very late, with a large influx of submissions in the few hours before each deadline. Furthermore, we can show how long it takes for a student to start their assignment to when they have their tests pass on average. We can also show which tests were causing students the most trouble.

Here is one such visualization of assignment submissions over time. The peaks are the few days before a due date.



This incredibly precise view into the actual data that can be generated on a platform such as Anubis is something that sets it apart from its competitors. We can show meaningful statistics to the professors and TAs on the platform about what went well with their assignment, and where students struggled.

2.6.4 Redis + RQWorker Redis has two purposes in Anubis. It is used by flask-caching to cache some endpoint and function results. For most all the functions that are heavy on the database, the results are cached for a specified period of time. The second purpose is as a rpc job broker. We use python-rq as it is super simple and easy to configure and set up. Using rq, we can enqueue functions to be run by a deployment of rpc-worker pods. Just about every time we need or want to do some work asynchronously, rq is used.

The redis setup needed for anubis is quite minimal. Again we would obviously want to change the password here in prod.

```
# Install a minimal redis deployment
echo 'Adding redis'
helm install redis \
     --set fullnameOverride=redis \
     --set password=anubis \
     --set cluster.enabled=false \
     --namespace anubis \
```

`bitnami/redis`

2.7 Logging

When it doubt, just log it

2.7.1 Logstash Logstash itself is a service that your applications can ship their logs to before being indexed into elasticsearch. Its purpose is to act as a natural buffer of log data. It is able to interface with elasticsearch, adjusting the speed of log ingestion as needed. In addition to acting as a buffer, it also enriches the data that it sees. For example, the logstash python client will not only ship the log message, but also the file that the log is coming from, along with which node the log is coming from.

This centralized, and persistent logging is indexed into elasticsearch, and accessed via kiaban. Anubis uses logstash on its API and submission pipeline.

3 Deployment

3.1 Kubernetes

The main goal with moving to Kubernetes from a simple single server docker-compose setup was scalability. We needed to scale our load horizontally. No longer was adding more RAM and CPU cores to the VM viable. With more users, we have a greater load. Kubernetes allows us to distribute this load across the servers in the clusters quite easily.

In moving to Kube, we are also now able to make guarantees about availability. In theory, even if sections of physical servers on the cluster go offline Anubis will still remain available. More on that later...

3.1.1 Helm Chart Anubis itself is packaged as a helm chart. Both mariadb and elasticsearch need to be setup separately. There are several options for how to deploy anubis that can be specified when running the deploy script at `kube/deploy.sh`. Notable options are deploying in debug mode, or restricting access to only the OSIRIS vpn. Options passed to the deploy script will be then passed to helm.

```
# Deploy with anubis only being accessable from the OSIRIS vpn
./kube/deploy.sh --set vpnOnly=true

# Deploy in debug mode
./kube/deploy.sh --set debug=true

# Set the number of replicas for the api service to a specific value
./kube/deploy.sh --set api.replicas=5

# Disable rolling updates
./kube/deploy.sh --set rollingUpdates=false
```

A full list of options can be found in the values.yaml file at `kube/values.yaml`.

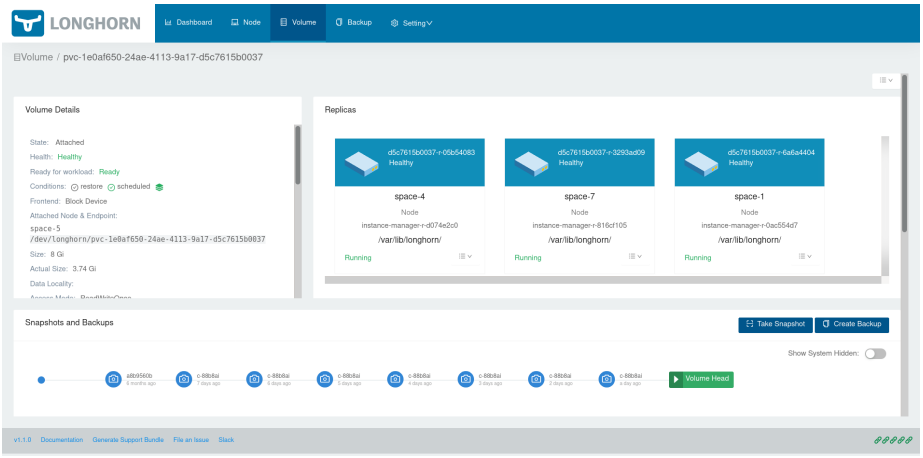
3.1.2 Rolling Updates In the decisions that were made when considering when expanding Anubis, availability was of the most important. Specifically we needed to move to a platform that would allow us to do *zero* downtime deploys. Kubernetes has very powerful rolling update features. We can bring up a new version of the Anubis API, verify that this new version is healthy, then bring down the old version. All the while, there will be no degradation in availability.

Of all the features of Kubernetes that Anubis leverages, none are quite as important as rolling updates. The Anubis API can be updated to a new version with *zero* downtime. This means we can live patch the API with new versions, with little to no degradation in service.

3.1.3 Longhorn The Kubernetes StorageClass that the Space Cluster supports is Longhorn. It allows us to have replicated data volumes with scheduled snapshots and backups.

All persistent data is stored on 3x replicated Longhorn StorageVolumes. Those volumes all have at least daily snapshots taken of them. At any given time, we can reset any stateful service to a previous snapshot from the last seven days.

For the things that are very important we have daily snapshots, and extra replication. Longhorn makes this as simple as checking some boxes. You can see here our mariadb master image with triple replication (two in Brooklyn one in Manhattan), with a 7 day snapshot.



3.1.4 The OSIRIS Space Cluster The OSIRIS Space Cluster is a k3s Kubernetes cluster that is managed by the OSIRIS Lab at NYU Tandon. It is primarily used for research and website hosting. It was designed by John Cunniff to be highly redundant, even in the case of a networking outage / natural disaster.

Its nodes are split between a datacenter in Manhattan and one in Brooklyn. A whole lot has to fail before the cluster gets in a state where it cannot heal itself.

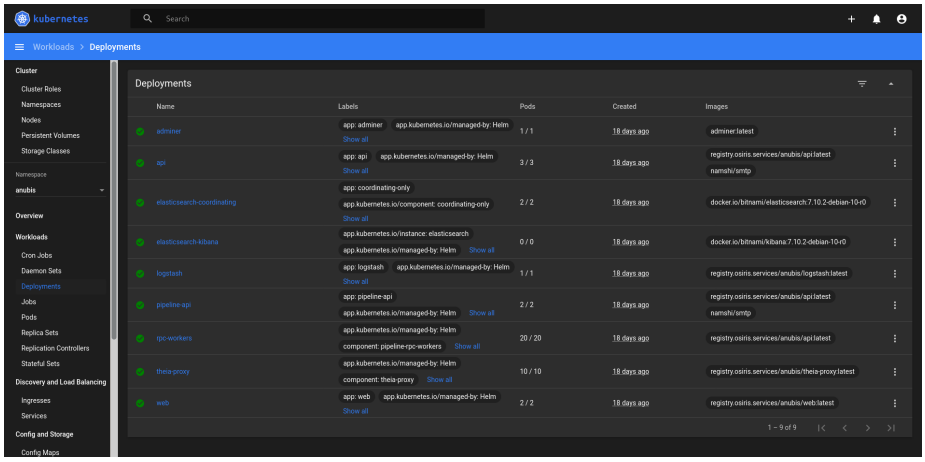
3.1.5 Nodes The cluster is comprised of servers that are evenly distributed between the South Data Center (SDC) in lower Manhattan and MetroTech Center (MTC) in Rogers Hall in Brooklyn. The nodes are connected via the internal OSIRIS network.

3.1.6 Networking There are multiple so called “Ingress Nodes” in both the MTC and SDC. These are nodes that have both public IP addresses and internal OSIRIS IP addresses. These nodes handle all inbound traffic to the cluster via Traefik.

Because there are multiple IP address for the space cluster, the main ingress DNS `space.osiris.services` will always resolve to all the current IP address. This guarantees greater availability, as if one IP is down for whatever reason another may be available.

The public IP addresses that are used are made up of both NYU, and POLY IP addresses. This design is quite purposeful. It will guarantee that even in the event of a network outage at Poly or NYU, the cluster will remain able to handle ingress.

3.1.7 Shared Services The space cluster has a few shared internal services. Namely, there is a shared replicated, and backed up MariaDB instance which Anubis will use a datastore. There is also an instance of Kubernetes-Dashboard running for viewing kubernetes resources. The Space Cluster also provides Longhorn for persistent data.



3.2 Github

3.2.1 Organization Each class that will need a github organization to put all its repos under. The only members of that organization should be the professor and TAs.

The organization should be set up to have push event webhooks to anubis. The endpoint the webhook should push to is `https://anubis.osiris.services/api/public/webhook/`. That endpoint will be hit when there is a push to a repo in that organization.

3.2.2 Classroom Github classroom is used to create, and distribute repos for assignments. There you can create assignments that will pull from a template repo of your choosing. A link will be generated that can be distributed to students. When clicked, that link will generate a new repo for the student within the class organization. As the student is not a part of the organization, they

will not be able to see any of the other student repos (given the assignment was made using a private repo).

The best place to put the template repo is within the class organization as a private repo.

One very important thing to note here is that in order to be able to create private assignment repo's, the classroom you create must be verified by github. This can take a few weeks, ad a professor needs to email github asking for permission from a .edu email providing their title and whatnot.

> Getting your github classroom / org approved will likely cause delays if not done at least a month before the semester starts.

4 CLI

4.1 Installing the CLI

The command line interface for anubis is packaged as a simple pip package. To install it, make sure you have python3 and pip installed, then from the anubis repo run `make cli`. This will run the pip install command necessary for installing the cli globally. If that was successful, then the `anubis` command will be installed.

4.2 CLI Usage

The main purposes of the CLI is for making it a bit easier for admins to make private API calls. In some cases it will handle the input and output of files.

```
jc@aion < master@61e1674 > : ~/nyu/os/assignment-tests/spring2021
[0] % anubis assignment init new-assignment
Creating assignment directory...
Initializing the assignment with sample data...

You now have an Anubis assignment initialized at new-assignment
cd into that directory and run the sync command to upload it to Anubis.

cd new-assignment
anubis assignment build --push
anubis assignment sync
```

Some use cases would be: - Adding, Updating and Listing assignment data - Adding, Updating and Listing class data - Packaging a new assignment tests, and sending its data off to the cluster

5 Assignments

5.1 Creating a new Assignment

Using the anubis cli, you can initialize a new assignment using `anubis assignment init <name of assignment>`

5.2 Writing Tests

All the files to build and run a complete anubis pipeline image will be dropped into the new directory.

```
new-assignment
|- assignment.py
|- Dockerfile
|- meta.yml
|- pipeline.py
|- test.sh
`- utils.py
```

The only thing you will ever need to edit is `assignment.py`. This is where you define your build and test code. Just like all the other cool libraries out there, the anubis pipeline works through hooking functions. Here is a minimal example of an assignment.py that will build and run a single simple test.

```
from utils import register_test, register_build, exec_as_student
from utils import TestResult, BuildResult, Panic, DEBUG, xv6_run, did_xv6_crash, verify_expe


@register_build
def build(build_result: BuildResult):
    stdout, retcode = exec_as_student('make xv6.img fs.img')

    build_result.stdout = stdout.decode()
    build_result.passed = retcode == 0


@register_test('test echo')
def test_1(test_result: TestResult):
    test_result.stdout = "Testing echo 123\n"

    # Start xv6 and run command
    stdout_lines = xv6_run("echo 123", test_result)

    # Run echo 123 as student user and capture output lines
    expected_raw, _ = exec_as_student('echo 123')
    expected = expected_raw.decode().strip().split('\n')
```

```
    # Attempt to detect crash
    if did_xv6_crash(stdout_lines, test_result):
        return

    # Test to see if the expected result was found
    verify_expected(stdout_lines, expected, test_result)
```

There are a couple functions to point out here. The `register_build` and `register_test` decorators are how you tell anubis about your build and test. The `exec_as_student` is how you should call any and all student code. It lowers the privileges way down so that even if the student pushes something malicious, they are still low privileged enough where they can not do much. It also adds timeouts to their commands. Boxing student code in like this is absolutely essential. Do not underestimate the creative and surprising ways students will find to break things.

5.3 Uploading Tests

Now you have your tests. That's great. The next thing you need to do is push the image to the docker registry and upload the assignment data to anubis. This is as simple as running two commands:

```
anubis assignment sync          # sends assignment metadata to anubis
anubis assignment build --push  # builds then pushes the assignment pipeline image to the re
```