Google Summer of Code

Proposal - Google Summer of Code 2024

Internet Yellow Pages Automatic Deployment

By

[redacted]

Table of Contents

About me

Name:	[redacted]
Timezone:	[redacted]
Email:	[redacted]
Course:	[redacted]
University:	[redacted]
Country:	[redacted]
GSoC Full Time:	[redacted]
Links:	[redacted]
Resume:	[redacted]

Why did I decide to work for IHR?

It is hard to imagine a world without the internet. We have grown accustomed to our lives where nearly everyone is online. The way we approach our finances, entertainment, and communication all rely on the internet's infrastructure. Yet, only forty years ago, the internet began as just a way for select United States universities and government researchers to communicate. While the origins of the internet are impressive, it does not compare to the vast network that billions of users rely on today.

I would love to contribute to IHR because of the crucial role it plays in monitoring this expansive network and its infrastructure. Because so many people and institutions around the world are on the internet, it is imperative that software engineers, policy-makers, network operators, and other stakeholders understand the state of the internet, which is made possible by the Internet Health Report. I recently took my university's Computer Networking course and was fascinated by how complex the internet is. Working with the IHR provides the unique opportunity for me to apply what I have been introduced to while continuing to learn in a hands-on environment. I know I will gain invaluable skills in crucial technologies like Python, Docker, and Neo4j as I collaborate with my mentors.

Project Title:
Internet Yellow Pages Automatic Deployment

A brief summary of the proposal: Abstract

The Internet Yellow Pages is a database containing information about internet resources including domain names, autonomous system numbers (ASNs), and subnets, which it receives from 18 organizations including Cisco, Cloudflare, and Stanford University. The data is stored using Neo4j, a graph database known for its flexibility and scalability.

It is important that the Internet Yellow Pages contain accurate data from all these sources and provide users with the data quickly and efficiently. However, IHR currently computes a new database of this form each week, and this deployment is not automated, which increases the time it takes to make the new data publicly available. Thus, the goal of this project is to design and implement an automated pipeline to deploy a new database to a remote server each week and to upload the database dump to a public repository. The solution must be maintainable and rigorously tested, with unit tests for each IYP dataset. This ensures that users receive reliable data with no downtime and that future developers can easily expand upon the pipeline.

These goals will be accomplished by completing the following four subparts: creating unit tests, creating a pipeline to automatically compute a database dump and push it to Github, integrating the tests into this pipeline, and extending the pipeline to also deploy a running instance of the database to a remote server.

How do I plan to achieve: Milestones

Subpart-I: Creating Unit Tests

Rigorous testing must be completed to ensure the integrity of the new database. In particular, unit tests must be implemented for each of the IYP datasets. While it may seem out of order that the deployment pipeline is not the first subpart, it is important that unit tests be written first. This way, we will be able to ensure that the deployment pipeline is working as expected incrementally for all the datasets as we are building it. If we create the pipeline first, then write tests once it is in a workable state, it would be much harder to know if the pipeline is actually working as expected during the development process.

Testing Libraries

Libraries can be imported to help create unit tests. A suitable testing library should have the following features:

1. Ability to create test fixtures
2. Ability to create specific test cases

Test fixtures represent the environment in which test cases are run. Fixtures prepare all the necessary ingredients for the test, so the test itself can be as simple as checking values against each other. For example, in our unit tests, fixtures will be used to load data from each IYP data set. Once this data is prepared and loaded by the fixture, we can run test cases on it.

The testing library then must provide an extensible and smooth process for creating test cases, so the validity of the data can actually be analyzed. The data in the newly computed database from the automated pipeline will be compared against the expected data. Two options that satisfy these requirements are unittest and pytest.

Unittest

Unittest is a unit testing framework that comes preinstalled with Python. It supports test automation, sharing of setup and shutdown code for tests, and aggregation of tests into collections. Unittest takes an object oriented approach, where each test is represented in a Python class that extends the `unittest.TestCase` base class, allowing us to create fixtures and test cases.

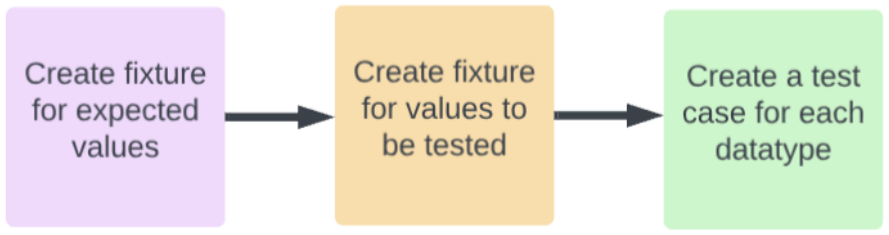
Pytest

Pytest is a third-party testing framework that focuses on creating small maintainable tests, which makes it more flexible than unittest. Its console output is also more readable and has colors. Pytest leverages the `assert` keyword to easily compare values against expected values. While Pytest focuses on simplicity, it also offers powerful tools to create more expansive tests if necessary. For example, it offers parameterization, so fixtures can be reused and be called on multiple different test cases that depend on the same setup.

Thus, we will use Pytest for this project because it is more lightweight and modular than unittest and appears to have more community support in recent years, which will help in the debugging process if questions arise.

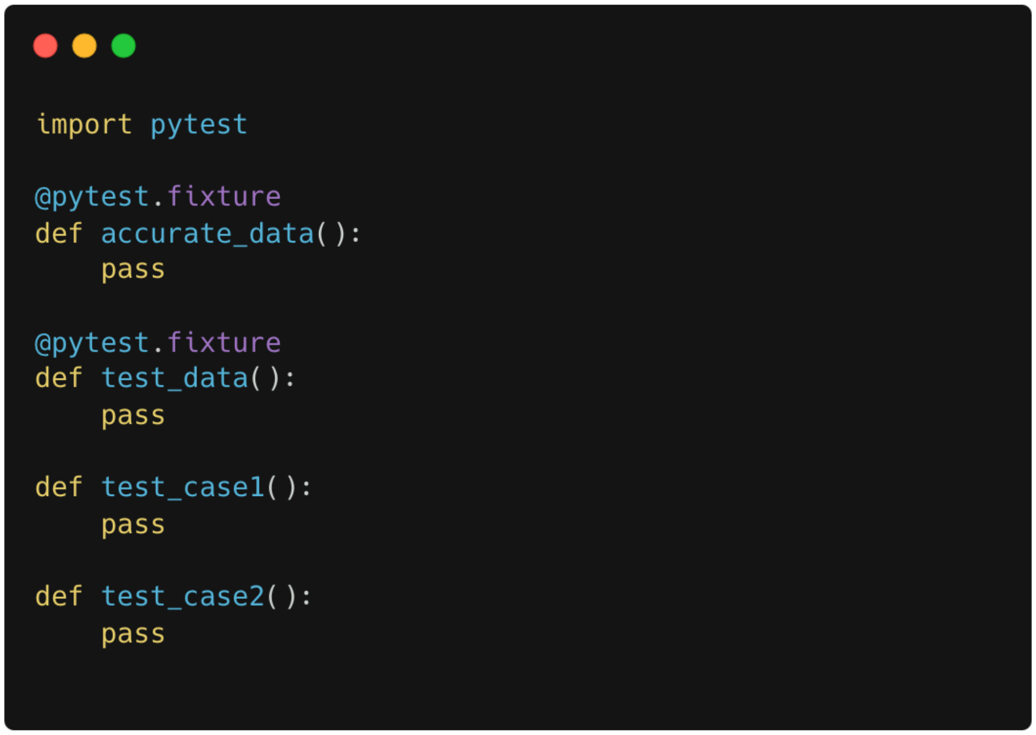
Testing IYP Datasets

The test suite will consist of tests on each of the datasets following this backbone testing structure:


Create fixture for expected values
Create fixture for values to be tested
Create a test case for each datatype

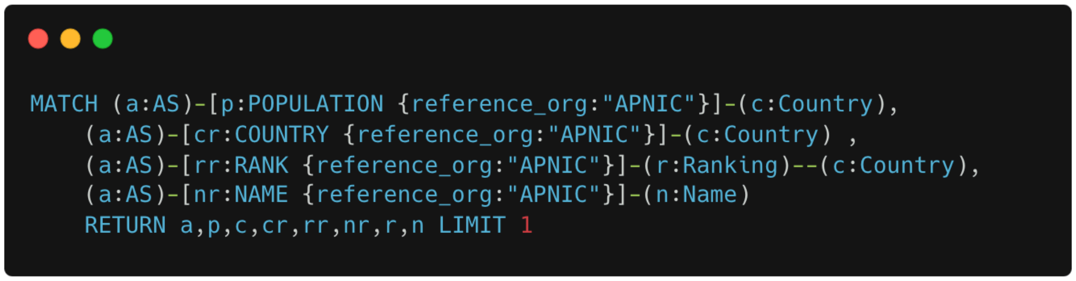
1. First, we must collect our baseline data that is accurate. This can be retrieved from the currently deployed database, which is manually created.
2. Once we have the accurate data, we can load the data that we want to check, which will be returned by another fixture.
3. Using the two previous fixtures, the data provided in each of them can be compared. An obstacle in this stage is that some of the data changes often, so the data in the current database will not be the same as the one that has been freshly calculated, making it difficult to determine if the new data is accurate. Thus, there must be a balance between testing enough data points while also ensuring that these data points can actually reflect the validity of the experiment data. An approach that offers a good tradeoff would be to identify data points or datasets that we know empirically are unlikely to change and then test the new data against those with a threshold to determine what should be considered passing. For example, we consider the new database to be accurate if 95% of the data that we expect to remain the same actually does remain the same.

Each dataset's unit testing Python file will follow this structure:



```python
import pytest

@pytest.fixture
def accurate_data():
    pass

@pytest.fixture
def test_data():
    pass

def test_case1():
    pass

def test_case2():
    pass
```

<u>APNIC Test Cases Example</u>
There will be a separate test case for each data type provided by the dataset that we have determined will not change often. For example, according to the IYP documentation, the Cypher query below shows an example of all the relationships added by the APNIC dataset:



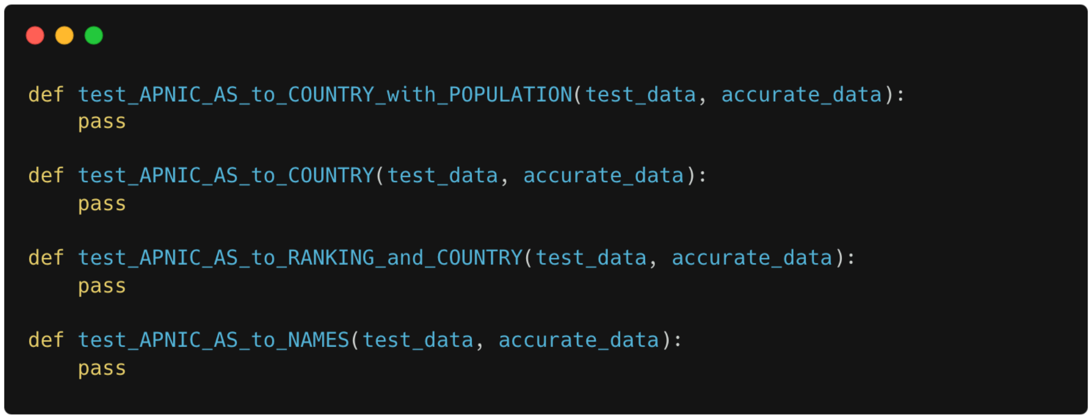
```
MATCH (a:AS)-[p:POPULATION {reference_org:"APNIC"}]-(c:Country),
    (a:AS)-[cr:COUNTRY {reference_org:"APNIC"}]-(c:Country) ,
    (a:AS)-[rr:RANK {reference_org:"APNIC"}]-(r:Ranking)--(c:Country),
    (a:AS)-[nr:NAME {reference_org:"APNIC"}]-(n:Name)
    RETURN a,p,c,cr,rr,nr,r,n LIMIT 1
```

This means that APNIC creates these relationships:

1. Connect AS to country nodes with a 'population' relationship representing the percentage of the country's population hosted by the AS.
2. Connect AS to country nodes, meaning that the AS serves people in that country.

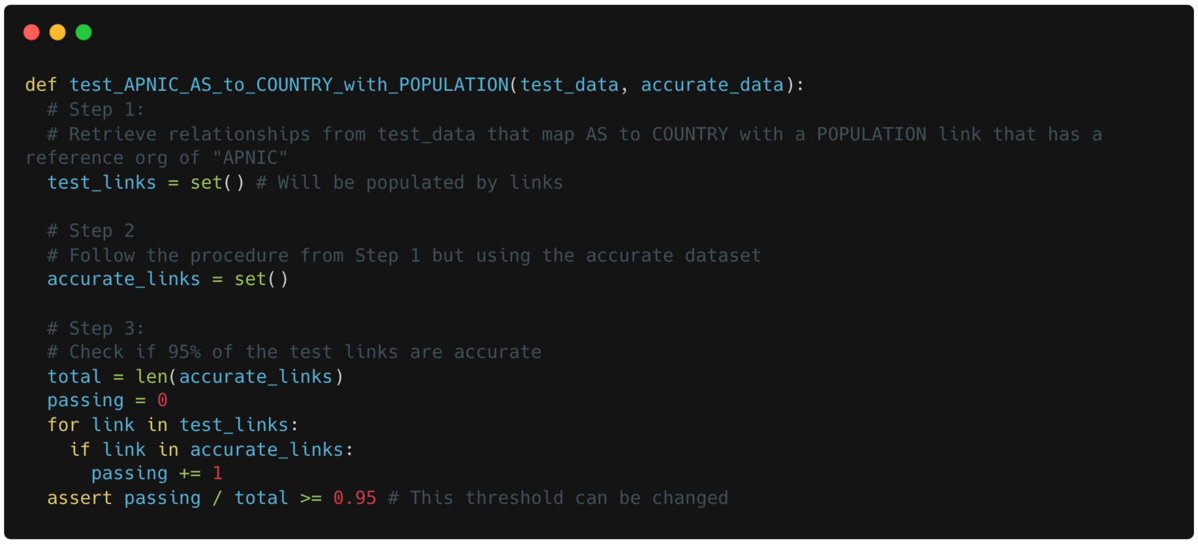
3. Connect ASes to ranking nodes which are also connected to a country. Meaning that an AS is ranked for a certain country in terms of population.
4. Connect AS to names nodes, providing the name of ranked ASes.

Then a test case needs to be written for each of these, which would look like this in Python:



```python
def test_APNIC_AS_to_COUNTRY_with_POPULATION(test_data, accurate_data):
    pass

def test_APNIC_AS_to_COUNTRY(test_data, accurate_data):
    pass

def test_APNIC_AS_to_RANKING_and_COUNTRY(test_data, accurate_data):
    pass

def test_APNIC_AS_to_NAMES(test_data, accurate_data):
    pass
```

Next, we can zoom into one of the specific test cases. The pseudo code below details how to test the first relationship. This tests if 95% of the links from the test data match up with all the links to the accurate data.



```python
def test_APNIC_AS_to_COUNTRY_with_POPULATION(test_data, accurate_data):
  # Step 1:
  # Retrieve relationships from test_data that map AS to COUNTRY with a POPULATION link that has a
reference org of "APNIC"
  test_links = set() # Will be populated by links

  # Step 2
  # Follow the procedure from Step 1 but using the accurate dataset
  accurate_links = set()

  # Step 3:
  # Check if 95% of the test links are accurate
  total = len(accurate_links)
  passing = 0
  for link in test_links:
    if link in accurate_links:
      passing += 1
  assert passing / total >= 0.95 # This threshold can be changed
```

Once such unit tests have been written for each dataset, it will streamline the process of creating the automated database pipeline since changes can be incrementally tested. These tests will also be used to check the database before it is deployed.

Subpart-II: Automatically pushing database dump to public repository

The next stage of the project is to design and implement a continuous pipeline to automate the process of computing and releasing the IYP database on a weekly basis. This can be achieved using Docker and Github Actions.

Tools

Docker

Docker allows code to be run in a controlled and replicable environment, which is exactly what is needed for this project. Our goal is to ensure that a stable database is deployed each week. This means that the environment that runs the code that automates this process must also be controlled. This stability is made possible through Docker containers.

Github Actions

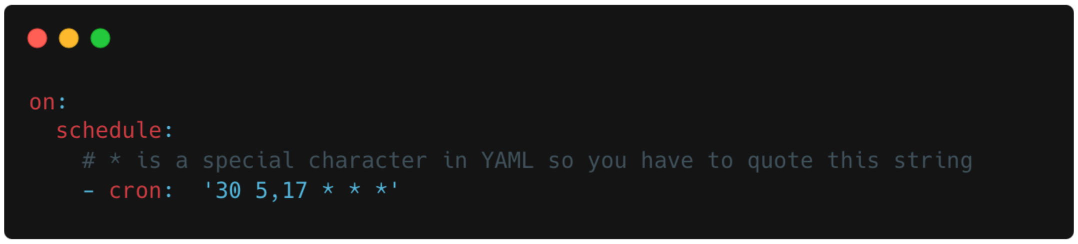
Github Actions works hand-in-hand with Docker. It allows containers to be automatically built, making it easy to create automatic software workflows, which is exactly what we want. An alternative to Github Actions is Jenkins, which is another open-source continuous deployment software. However, Github Actions is more inline with IYP's workflow. First of all, IYP already uses a public Github repository to store its code, which means that Github Actions are free. Furthermore, Github Actions is already used in IYP to run the pre-commit hook, which is used to inspect the code before it is committed. If something else like Jenkins were to be used, extra time would have to be dedicated towards setting up and maintaining a completely new and separate tool. Thus, Github Actions can be used with Docker and can be seamlessly applied in the database deployment.

Scheduling

The pipeline will consist of one Github Action workflow consisting of multiple jobs, the instructions of which will be specified in a YAML file contained in a `.github/workflows` directory. This is where Github is configured to look when trying to find Github Actions workflows.

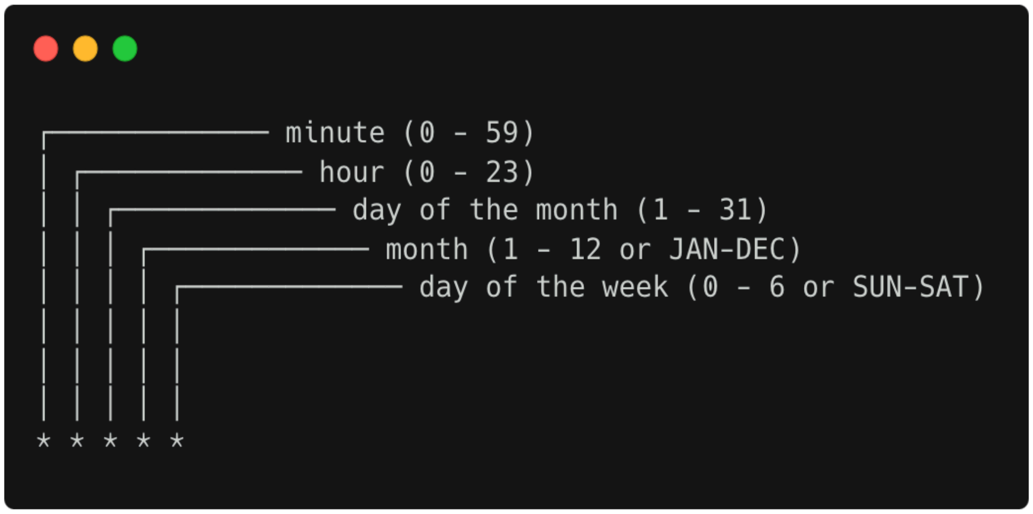
The first subgoal to creating this pipeline is to get code to run on a schedule, since we want to ultimately complete the same task every week. In our YAML file, we can include properties on how the action will be run. One such property that we can specify is `schedule`, which controls when the action is run.

The following code sample shows an example of the schedule part of an actions YAML file. This specifies that the action should be run according to `cron`.



```yaml
on:
  schedule:
    # * is a special character in YAML so you have to quote this string
    - cron:  '30 5,17 * * *'
```

`Cron` is a command-line utility that allows you to create jobs that run at regularly scheduled intervals and times. The details of each field you can provide to the command are outlined below. Note that in the example above, the cron job specifies that the workflow should be run everyday at 5:30 and 17:30 UTC.



```
┌───────────── minute (0 - 59)
│ ┌───────────── hour (0 - 23)
│ │ ┌───────────── day of the month (1 - 31)
│ │ │ ┌───────────── month (1 - 12 or JAN-DEC)
│ │ │ │ ┌───────────── day of the week (0 - 6 or SUN-SAT)
│ │ │ │ │
│ │ │ │ │
│ │ │ │ │
* * * * *
```

<u>Tasks</u>

Now that we have detailed the schedule, we must detail the tasks that are completed according to the schedule. Primarily, the automated pipeline must create and push an IYP database (to accomplish the goal of this subpart).

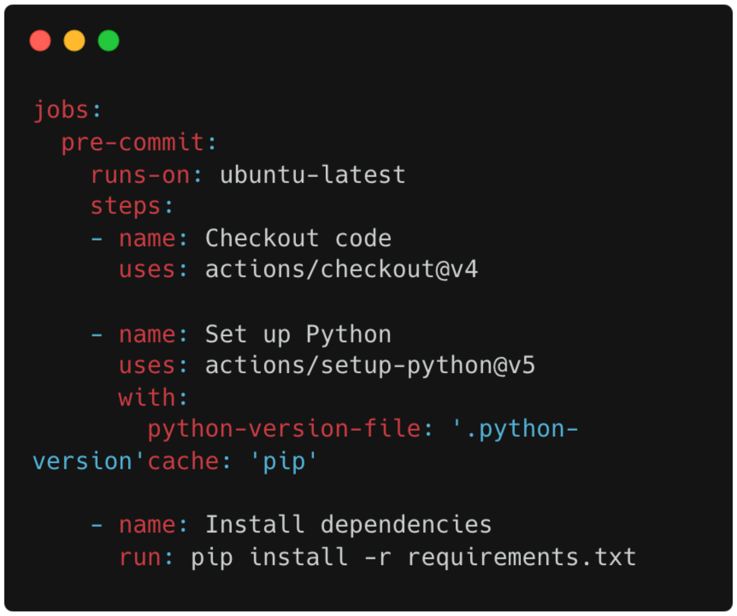
Automatically creating a new database
Currently, the database is created using the `create_db.py` script located in the top-level project, which must be run manually. The steps it takes to create the database are:

1. Load the configuration file `config.json`, which is used to determine settings for how the database should be created and contains details such as which sources should be used to collect data.
2. Create a docker container with version `5.16.0` of the Neo4j
3. Start the image created in step 2, so there is now a fresh instance of a Neo4j graph database that can be populated with new data.
4. Collect data from each source specified in the `config.json` file from Step 1 by running the data crawler that is associated with each source. Running the crawler will populate the database with the newly collected data.
5. Run post processing on the data.
6. Stop the container and dump the database.

Once the script is finished running, the dumped database is uploaded to `https://exp1.iijlab.net/wip/iyp/dumps/`, where users can download the dump from the desired week.

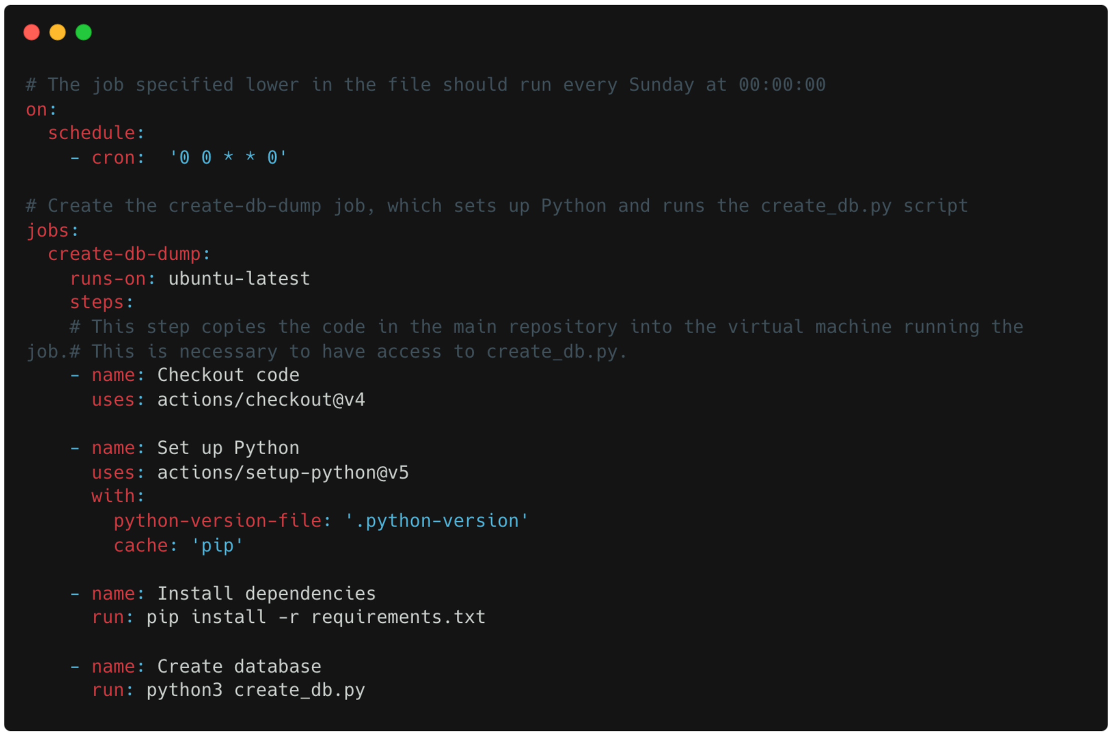
This script must be scheduled to run every week, so the database can be created automatically, which we can achieve by telling a Github Action to run the script. Github Actions workflows run in dedicated virtual machines, which satisfies the requirement that the scheduled code be run in an isolated and controlled environment. Since a new environment is created when the workflow starts, we can configure Python however we want. In fact, we will follow closely the approach that is already taken in the existing workflow, `pre-commit.yml`.

The following snippet is taken from `pre-commit.yml`. Notice how in the pre-commit job, the steps "Set up Python" and "Install dependencies" detail how Python and the environment in which the code is run should be configured. This approach ensures that the environment in which we run the script is isolated and can be recreated. It also ensures that if the dependencies of the project, which are listed in the `requirements.txt` that is used in the last line, change, then the build environment will automatically use the updated version of the requirements file.



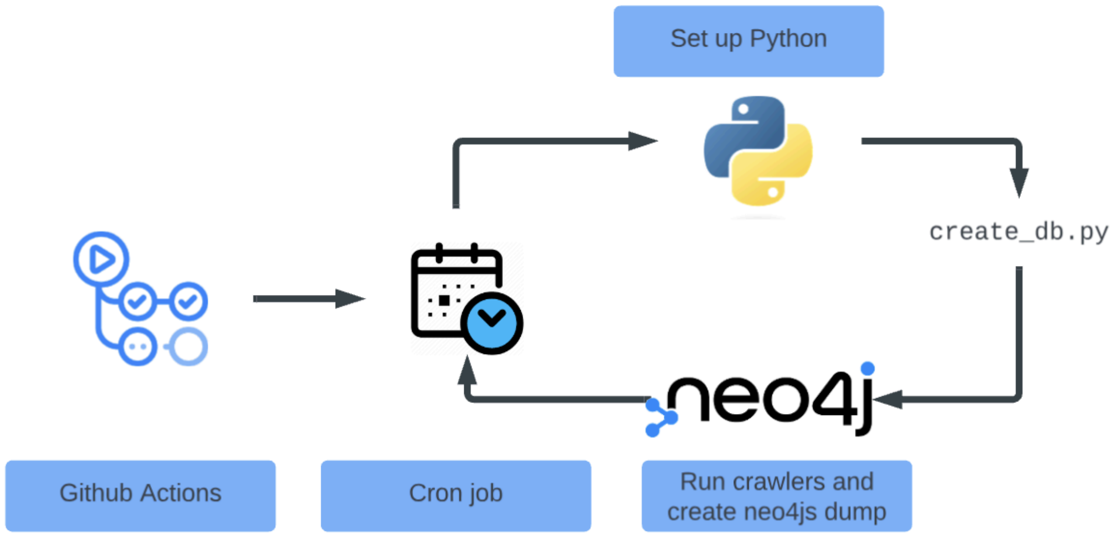
```yaml
jobs:
  pre-commit:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version-file: '.python-
version'cache: 'pip'

    - name: Install dependencies
      run: pip install -r requirements.txt
```

Once Python has been set up with the desired dependencies, `create_db.py` can be run to automatically create the database dump. The YAML file to that controls the pipeline can be expected to look like the following:



```
# The job specified lower in the file should run every Sunday at 00:00:00
on:
  schedule:
    - cron:  '0 0 * * 0'

# Create the create-db-dump job, which sets up Python and runs the create_db.py script
jobs:
  create-db-dump:
    runs-on: ubuntu-latest
    steps:
    # This step copies the code in the main repository into the virtual machine running the
job.# This is necessary to have access to create_db.py.
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version-file: '.python-version'
        cache: 'pip'

    - name: Install dependencies
      run: pip install -r requirements.txt

    - name: Create database
      run: python3 create_db.py
```

Overall, the database creation part of the pipeline will follow the following structure:

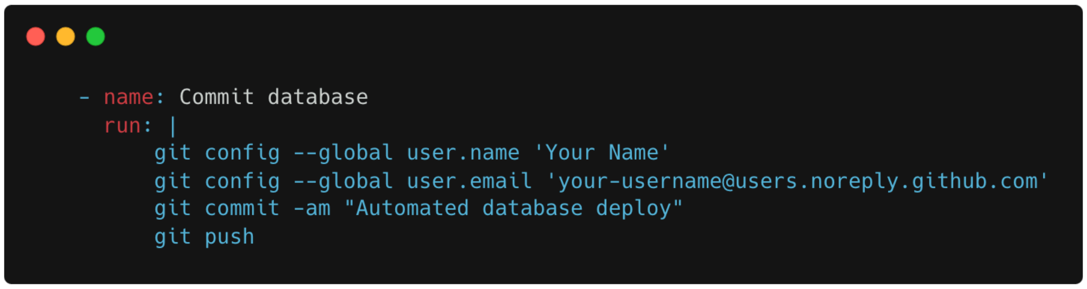


Automatically pushing the database

Once the database is created, it must be pushed to a public repository.

<u>Approach 1: Creating and pushing in one job</u>

This approach modifies the YAML file from above. After running `create_db.py` the dump is stored in the virtual machine currently running the workflow. The following step can then be added to the job to commit and push this dump to a public repository.



```
- name: Commit database
  run: |
      git config --global user.name 'Your Name'
      git config --global user.email 'your-username@users.noreply.github.com'
      git commit -am "Automated database deploy"
      git push
```

The commands under the “run” property can be modified to specify which account is used to push the new database.

One issue with this approach is that it puts both phases, creation and deployment, in one workflow job. Ideally, jobs with different goals should be separate. The next approach explores a solution to this problem

<u>Approach 2: Creating a separate job for pushing</u>
In order to make the pipeline organized and maintainable, the pushing phase can be separated into its own job in the workflow. However, since each job is executed in a separate environment, the deployment job does not have access to the dump file that was created by the other job by default.

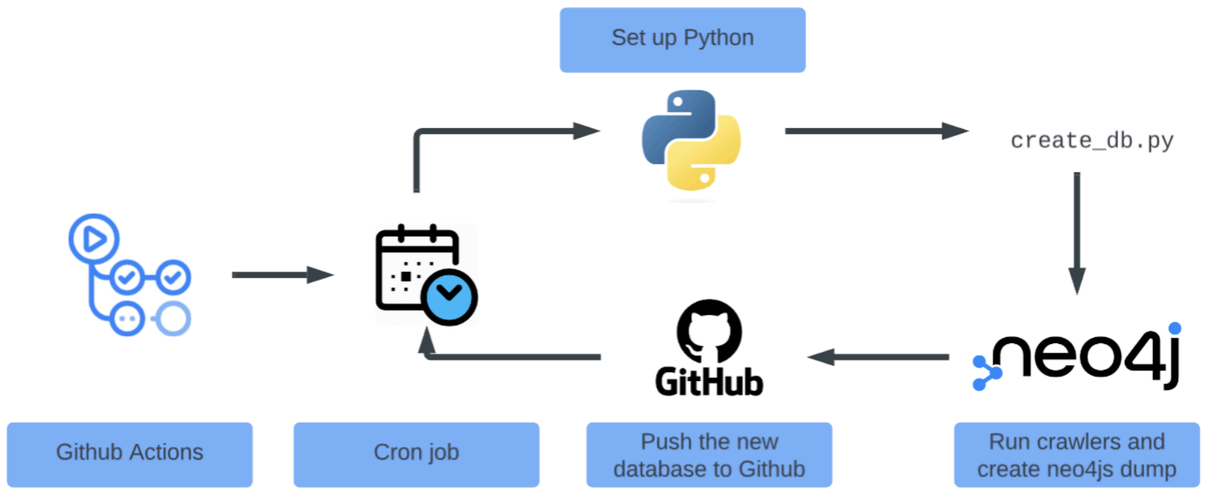
To solve this problem, we can leverage the cache action. The database creation job will add the dump to the cache. Once this is done, the deployment job can be run, where it will access the dump from the cache. With access to the dump, the deployment job can push the dump to a public repository. First, the creation job from above must be modified to cache the dump. This is done by adding a `uses: actions/cache@v2` property.

Additionally, the `needs` property of the pushing job must be set to `create-db-dump`. This property tells the pushing job to wait until the creation job is finished before it should run. Now, the pushing job can read from the cache to access the dump.

However, this approach is also flawed. Github enforces a 10 GB limit to the size of each repository's cache. This is especially problematic because dumps are typically 30 GB, with reaching sizes of over 60 GB.

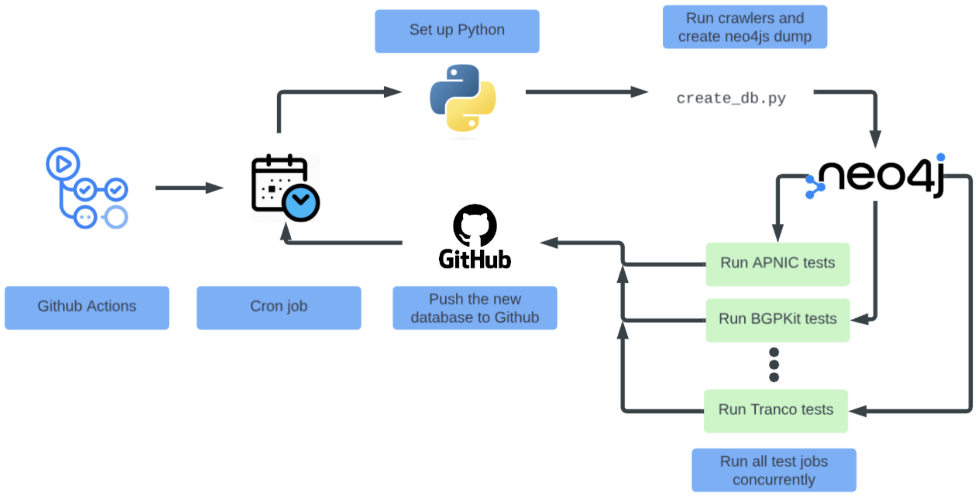
To get around this, it is also possible to download the dump directly as an artifact. This is Github Action's way of persisting files across entire actions, not just across jobs. Yet it takes over 30 minutes to download a large dump. This approach would add an extra upload sequence from the database creation job and a download from the database deployment job, which ultimately adds potentially over an hour of unnecessary overhead.

<u>Which approach is best?</u>
Overall, the first approach is more suitable. While it is slightly less organized by pushing both phases of the pipeline into a single job, this price is worth it because the other approach adds too much wasted time and overhead. We arrive at the following workflow:

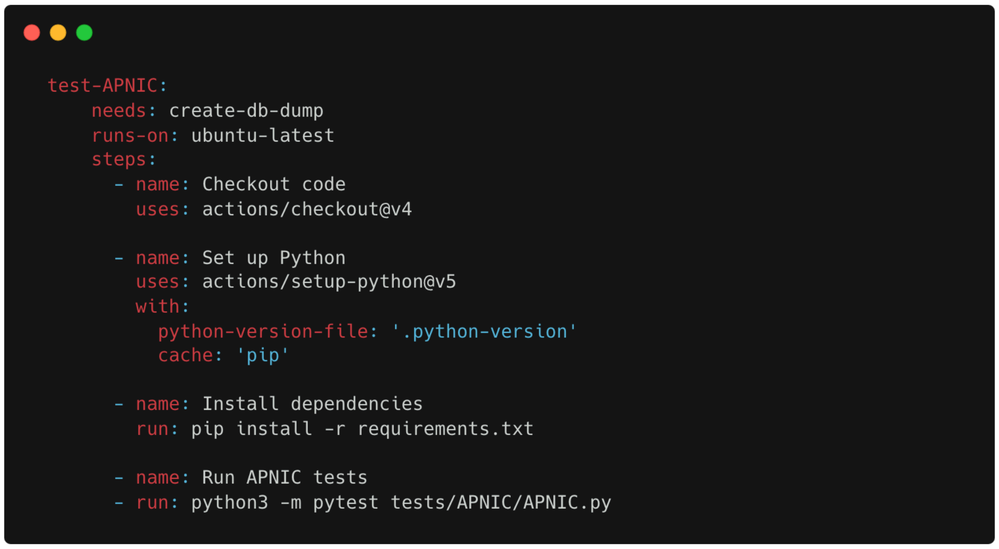

Set up Python
create_db.py
neo4j
GitHub
Github Actions
Cron job
Push the new database to Github
Run crawlers and create neo4js dump

Subpart-III: Combine the testing with the pipeline

Because the Python files were created for each dataset, we have a modular system for automatically running tests. A job can be created in the workflow for each of the tests. Each job will run the corresponding Python unit test file for the dataset. After the database has been created but before the database is deployed, the workflow will concurrently run each of the jobs. If all the jobs pass, then the database dump will be pushed to a public Github repository.



There will be jobs added to the workflow YAML file that look like the snippet below, which is an example for the APNIC dataset:


```
test-APNIC:
    needs: create-db-dump
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version-file: '.python-version'
          cache: 'pip'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run APNIC tests
      - run: python3 -m pytest tests/APNIC/APNIC.py
```

Note that Github Actions runs jobs concurrently, and each job has its own virtual machine. This means that for each test, Python must be configured again from scratch. The overhead of having to reconfigure Python is worth it because of the time that is saved by running the jobs concurrently.

Furthermore, the testing jobs should only be run after the database dump has been created, so we add `needs`, which specifies that the jobs should only run after the `create-db-dump` dump has successfully completed. Finally, once Python has been set up, pytest is run with the unit test file for the given dataset.

Subpart-IV: Deploying the database to a remote server

From the first three subparts, we have a maintainable and tested pipeline that automatically pushes a weekly database dump to a public Github repository. This allows end-users to have a regularly-updated source from which they can download the latest database dumps, but it does not provide a way for users to connect to the current database directly.

This will be accomplished using another Github Actions job with a self-hosted runner and Docker.

Self-hosted runner

By default, Github Actions executes jobs on virtual machines hosted by Github that have a preconfigured base environment, on top of which workflows can add more features. The `runs-on` property from the YAML snippets in the previous subsections tell the job to be run on a VM using `ubuntu-latest`.

However, to deploy our own database, the job that deploys the database should be executed on IHR's own virtual machine instead of Github's. This is made possible by a self-hosted runner, which is a system that the IHR deploys that can be used to manage actions from Github. The runner machine connects to Github using the Github Actions self-hosted runner application, which is open source.

Deploying the database to remote server

Similar to the testing suite jobs, the remote deployment job will run after `create-db-dump` and all of the tests have finished. The key difference is that the deployment job is run on another machine, which is specified using the `runs-on` property.

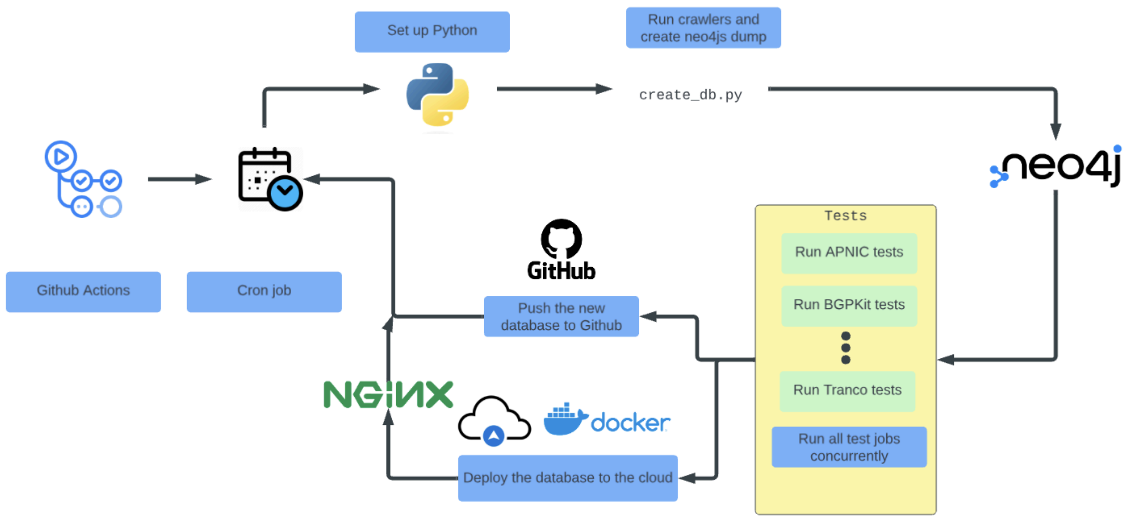
The self-hosted runner will be configured in the Github Actions page. The job will first checkout all the code, just like all the previous jobs. Then, it will have access to the `docker-compose.yaml` file. This file includes the instructions for how to create a docker image that runs an instance of an IYP database. As a result, creating a container based on this image is as simple as having the job run `docker compose —profile local up`, the behavior of which is already defined in the existing `docker-compose.yaml` file. This first creates a container that runs Neo4j admin, which creates an active graph database from the previously computed dump. Next, it creates a container that provides public read-only access to the aforementioned Neo4j database. We now ultimately have a remote Docker container running a version of the new database, which can be accessed by the public.

Deployment downtime
Note that there may be downtime between stopping the current container and starting the one running the newly calculated database. Nginx can be used to solve this issue. Nginx is a web server that can act as a reverse proxy and a load balancer. When users want to access a database, their requests will first go through nginx, which forwards this request to the desired container.

This approach allows us to have multiple containers and, in turn, database versions, running at the same time. To address the downtime issue, we can keep an instance of the old database running as the new instance is starting. Thus, there will be a point where the old database and new database are running simultaneously, so users never experience a period where they have access to no data at all. When the new database is ready, we can notify nginx to start forwarding requests to that database instead of the old one.

One of the IHR mentors, Romain, noted that nginx has the additional benefit of providing flexibility and access to database history by allowing different hostnames to route to past databases. For instance, we could have the following addresses:
- now.iyp.iijlab.net (current database)
- 1w.iyp.iijlab.net (one week old)
- 1m.iyp.iijlab.net (one month old)

Nginx could be configured to forward the user to the correct database depending on the hostname. Ultimately, we arrive at the following flowchart:


Set up Python
Run crawlers and create neo4js dump
create_db.py
neo4j
Github Actions
Cron job
GitHub
Push the new database to Github
Tests
Run APNIC tests
Run BGPKit tests
Run Tranco tests
Run all test jobs concurrently
NGINX
docker
Deploy the database to the cloud

Documenting it

I will make sure to clearly and concisely comment on all of the code that I write, so future engineers can quickly understand how everything works.

For the dataset tests, I will follow a similar structure to the crawlers directory, where each dataset has a corresponding `README.md` file that describes the data that is being retrieved and how the crawler works. For each dataset, I will also include a `README.md` describing what each test function and fixture does. At the top level of the tests directory, I will also include an overarching readme that describes the general structure and flow of the tests similar to how I have described it earlier in this document.
Similarly, I will also include a readme file in the `.github/workflows` directory that walks through the steps and jobs in the workflow.

Plan of Action: Timeline

May 1 - July 1

1. Go through Neo4j docs to become more familiar to graph databases
2. Improve understanding of Cypher syntax
3. Become more accustomed with the codebase
4. Experiment more with running Cypher queries in IYP browser
5. Create a mock CI/CD project to gain more experience with Docker and Github Actions.

   a. Create a basic Docker “Hello World” application with Redis that tracks page visits and implements continuous integration with Github.

July 1 - July 14

1. Team bonding
2. Gain deeper understanding on how the IYP website uses/interacts with the current database.
3. Deep dive into the datasets to understand what relationships each one creates.
4. Create unit tests and documentation for APNIC, BGPKit, and Bgp.tools. This will likely take longer than it will to write the other tests as I become accustomed to the test-writing process.

July 15 - July 31

1. Write unit test files and documentation for each of the other datasets
2. Begin setting up Github Action workflow for automatic deployment
3. Set up cron job that regularly runs `create_db.py`
4. Complete deployment pipeline excluding the testing phase
5. Add the testing phase job into the deployment pipeline.

August 1 - August 14

1. Set up a self-hosted runner on a remote server.
2. Assign the deployment job to the self-hosted runner.
3. Configure nginx to route to different running containers

Apart from the vital contributions: Extras

An additional feature would be to keep deployments of previous databases active. It will be simple to store prior dumps on Github because the repository containing the dumps will just be regularly expanded. Furthermore, in order to have remote deployments be backlogged, we can dynamically add new containers running Neo4j from different ports. Older versions will run on separate ports while the current database will use the default 7474 and 7687 ports. As stated earlier, nginx can then be used to route requests to the correct ports.

Open Source contributions and Projects

I was an undergraduate software engineering research assistant for the [redacted]. The [redacted] actively develops [redacted], a language and modeling environment to create agent-based simulations. In particular I worked on [redacted], a tool built into [redacted] that systematically varies the parameters that are used to run the user's simulations, so users can perform experiments on different combinations of inputs.

Organization	Contributions
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]

Expectations from Mentor

I am always looking to learn and improve upon my mistakes. I would like my mentors to be open about any mistakes that I make, so we can work together to make my project better. This gives me the opportunity to collaborate with them and learn from their experiences as professionals in this field. I would also love to learn about why they decided to work with the IHR and their journeys to where they are today.

Furthermore, I am eager to learn more about how my project ties into the bigger picture of IHR's goals, so I would like my mentors to teach me about how my aspects or subparts of my project connect with other ongoing projects that they are working on.

What are my other commitments?

My final exams for the Spring Quarter at school end on June 7. I have accepted an offer to [redacted] in [redacted], which starts on June 10. It is a ten-week internship divided into two phases. The first part is a bootcamp-style experience where I will learn from software engineers at [redacted]. During the second part, I will apply what I have learned to work with a [redacted] team on a real-world project.

Throughout high school and my first two years of university, I have learned to manage my time and find a balance with high workloads. So while it may seem like a lot of work to complete, I am confident in my abilities to deliver strong software solutions for both IHR and [redacted].

What do I plan after the GSoC period is over?

After the GSoC period is over, I want to continue working with IHR. I believe that their mission to provide data about the internet is extremely important in the age where everything relies on the internet, and I would like to contribute to their cause. I will be available to make modifications and additions to this project. Also, during the GSoC period, I hope to learn more about the other projects that the IHR is working on, which will prepare me to contribute to other projects in the future as well.

Who am I: About me

My name is [redacted], and I was born and raised in [redacted]. I am a current sophomore studying at [redacted], which is located in the suburbs north of [redacted]. I decided to major in computer science because I have always loved creating things from scratch. Gaining skills in CS will allow me to build software from the ground up that people around the world can use. I also love to challenge myself as I believe the best way to learn is to get out of my comfort zone and to learn from my mistakes, which is what motivated me to double major in mathematics. I love problem-solving, and majoring in a difficult field like math has allowed me to improve my skills while collaborating with and learning from my peers.

In my free time, I enjoy fitness and being active. I was captain of my high school wrestling team, where I helped lead my team to a first-place undefeated finish in the [redacted]. I am now currently on my school's club gymnastics team. I also enjoy being outdoors and love to go hiking.

The experience I possess: Places worked

Teaching

[redacted]

I was a teaching fellow at [redacted], which introduces students from underrepresented communities to computer science. During this six-week program, students learned how to code in Javascript and how to combine HTML, CSS, and Javascript to create dynamic websites with a Firebase backend. I helped answer questions during live lessons and led office hours sessions where I assisted students who needed extra help.

[redacted]

I was an undergraduate teaching assistant for [redacted]. This class is required for all computer science majors and introduces students to variable scope, control flow, and recursion using the functional programming language Racket.

Experience

[redacted]

See details in the "Open Source Contributions and Projects" section. This experience provided valuable skills in how to effectively use Git and Github in the open source development process.

[redacted]

As a software engineering intern at [redacted], I gained my first real-world experience in computer programming. I gained experience using React Native, Express.js, and AWS by creating new app screens and expanding their backend API.

Projects

[redacted]

Optimized gravity simulation time complexity from O(n2) to O(nlog(n)) using the Barnes-Hut Algorithm. Leveraged C++, compiler optimizations, and multithreading to support over 100,000 bodies, improving performance over prior version written in Python that was unable to perform calculations with only 5,000 bodies. This project gave me more experience with creating visual projects with SFML and with data structures and algorithms.

[redacted]

I developed a full-stack website that displays ASCII art generated from user-uploaded images. Created an API to handle user authentication and CRUD operations on posts. This project gave me more experience creating full-stack web applications.

Relevant Coursework

- [redacted]
  - Learned about basic computer science concepts and data structures in Java.
- [redacted]
- [redacted]
  - Learned about imperative programming and manual memory management with C, and object oriented programming in C++.
- [redacted]
- [redacted]
  - Learned about the hierarchy of abstractions and implementations that comprise a modern computer system.
  - Gained more experience in programming in C in the Unix environment and GDB debugging.
- [redacted]
  - Learned about key features of programming languages including closures and garbage collection by creating a series of interpreters in Racket.
- [redacted]
  - Learned about the 5-layer networking model.
  - Created an HTTP server with sockets
  - Created a TCP/IP stack
  - Implemented link state and distance vector routing algorithms

    - Created a [redacted] tool that collects information on websites and DNS servers. Went beyond the requirements of the project to leverage asyncio, to learn about coroutines and to speed up the program speed by over 5x.
- [redacted]
    - Introduced to Docker and AWS.
- [redacted]
- [redacted]
    - Introduction to graph theory: graphs, trees, matchings, planar graphs, and colorings.