**Workshop Guide: Automating Threat Hunting with Detection As Code**

*References used within this guide:*

[*Getting Started with Detection-as-Code and Chronicle Security Operations (Part 1 of 2)*](https://www.googlecloudcommunity.com/gc/Community-Blog/Getting-Started-with-Detection-as-Code-and-Chronicle-Security/ba-p/702154)

[*AI Runbooks for Google SecOps: Security Operations with Model Context Protocol*](https://www.googlecloudcommunity.com/gc/Community-Blog/AI-Runbooks-for-Google-SecOps-Security-Operations-with-Model/ba-p/906723)

[*Building an open ecosystem for AI-driven security with MCP*](https://www.googlecloudcommunity.com/gc/Community-Blog/Building-an-open-ecosystem-for-AI-driven-security-with-MCP/ba-p/898843)

Table of Contents

[**Prerequisites 3**](#_yq7twgfn0lq0)

[(if you’re using your local VS Code IDE) 3](#_fpwty5mt580j)

[**Setting up Detection-As-Code on GitHub 4**](#_4fh2zenkrf11)

[Setting up Github 4](#_608a9te820e7)

[Accessing Web based VS Code 6](#_3omej2wch8de)

[Setting up AI on VS Code 12](#_fywspash5ptn)

[Configure GitHub Actions & Variables/Secrets 19](#_gcte6y6n3n64)

[Testing the Github Actions - 1. Pull Latest Content 20](#_3nukcsep7g7t)

[Testing the Github Actions - 2. Creating a new rule 23](#_17tvc5d90tzx)

[Using Model Context Protocol (MCP) servers in your IDE to Automate Threat Hunting 28](#_ix5l2qqs1fnz)

[Example Prompts: 28](#_qy48zf5ce3hn)

[Prompt to automate threat hunting with a detection rule based on a report 29](#_xb9by5k7nu4h)

[**Appendix: Best Practices working with Git 30**](#_682q6lxbgotc)

[Commit early and often (and Small!) 30](#_qndrhm5ib0g6)

[Write Good Commit Messages 30](#_gn8vm14ys0ax)

[Stay Up-to-Date and Manage Conflicts 30](#_xpc3bzb2le4k)

[Keep Your Repository Clean 30](#_ap5knz3ey9od)

# 

# 

# **Prerequisites**

* A GitHub account

## *(if you’re using your local VS Code IDE)*

* [Visual Studio Code](https://code.visualstudio.com/)
* If using local VS Code:
  1. Installed in your local terminal:
     1. [Python](https://www.python.org/downloads/)
     2. [uv](https://docs.astral.sh/uv/getting-started/installation/)
     3. [gcloud CLI](https://cloud.google.com/sdk/docs/install)
     4. [jq](https://jqlang.org/download/)
     5. [Github cli](https://cli.github.com/)

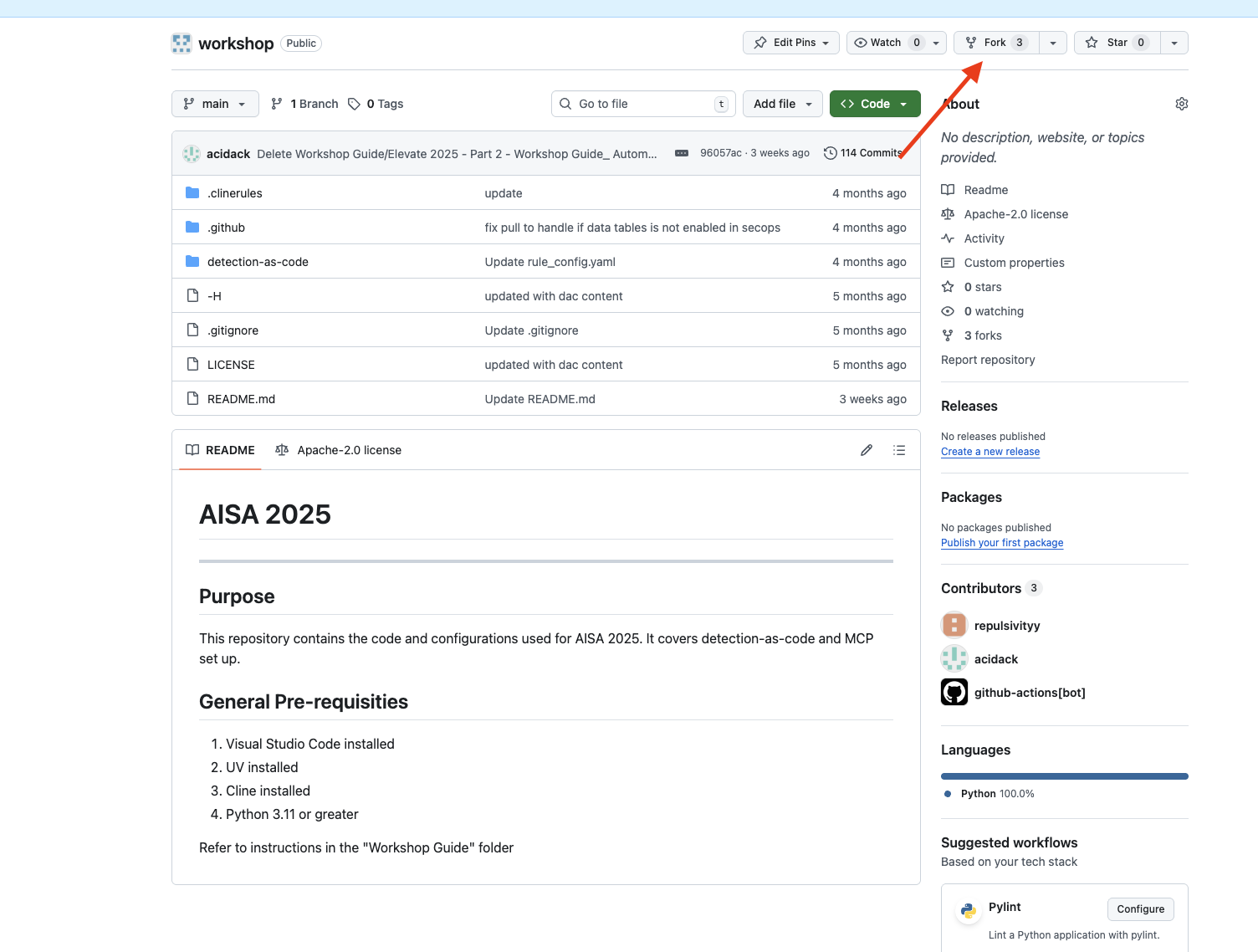
### 

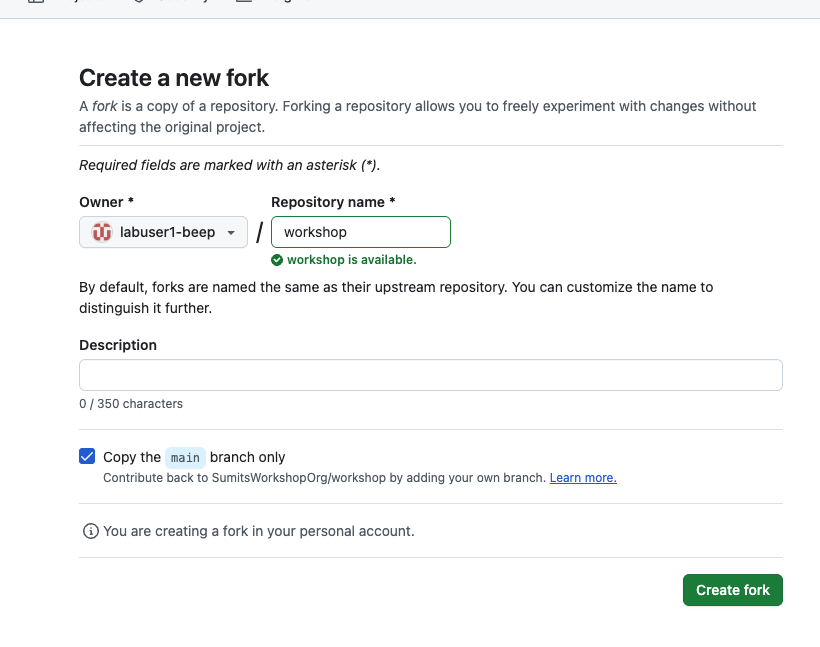
# 

# **Setting up Detection-As-Code on GitHub**

## **Setting up Github**

1. Set up (if you don’t have one) and log into a Github account
2. Next, back in github start by creating a fork of this repo <https://github.com/SumitsWorkshopOrg/workshop> into your own repo *e.g.* [*https://github.com/[YOUR\_REPO]/workshop*](https://github.com/%5BYOUR_REPO%5D/workshop)



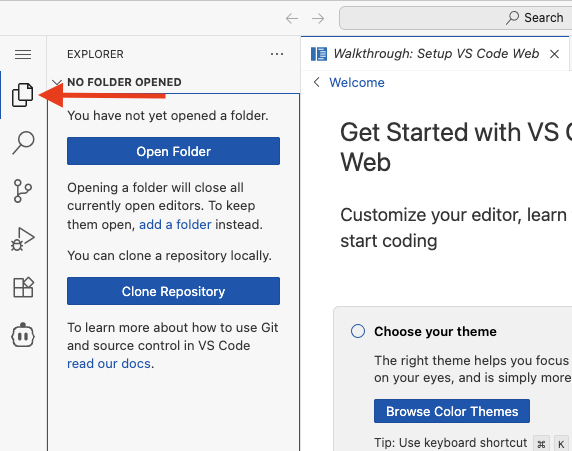


1. *At this point, organizers will permit access based on Github user accounts that have forked the repo above (please be patient till this step is completed)*

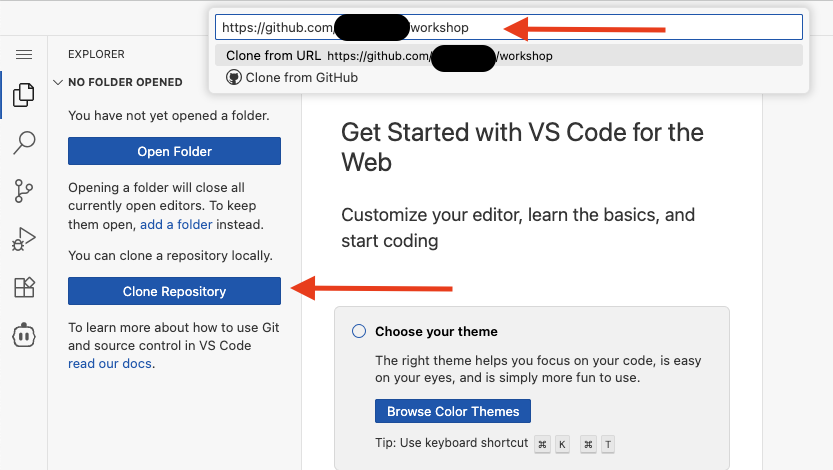
## 

## **Accessing Web based VS Code**

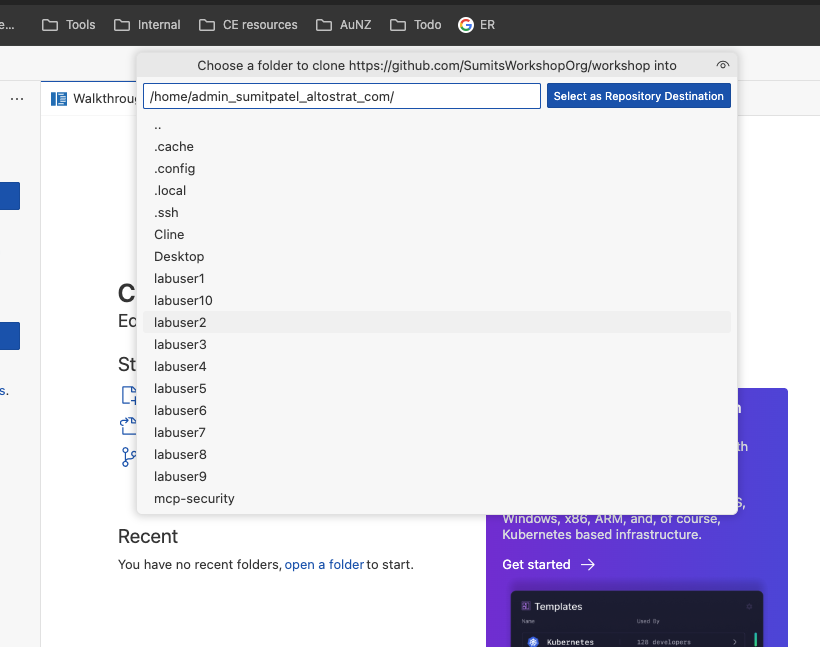
1. Open VS Code via <https://code.patesumi.dev/?ew=true>
2. *(Organizer to allocate) username / password*
3. Select the **Explorer** icon in the menu



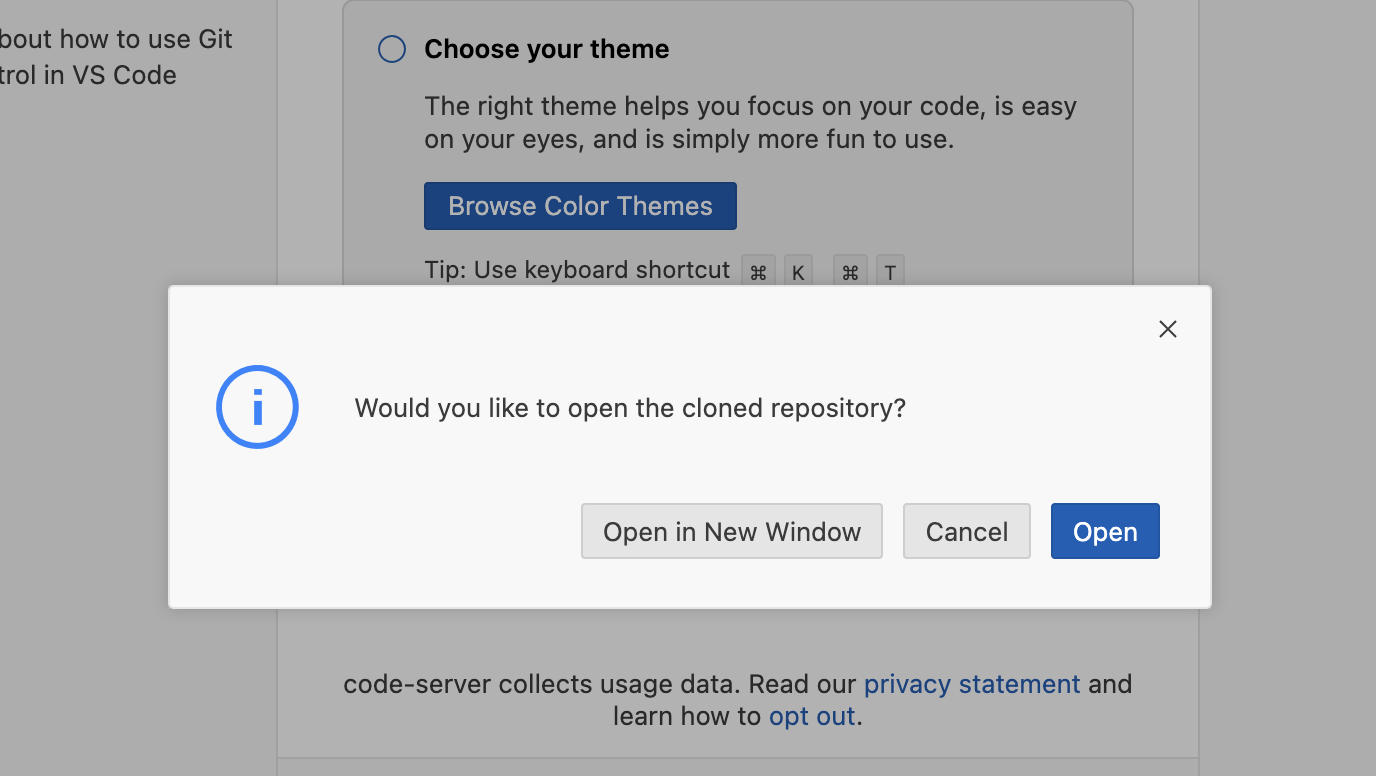
1. In VS Code, select “**Clone Repository**” of YOUR repo you just forked



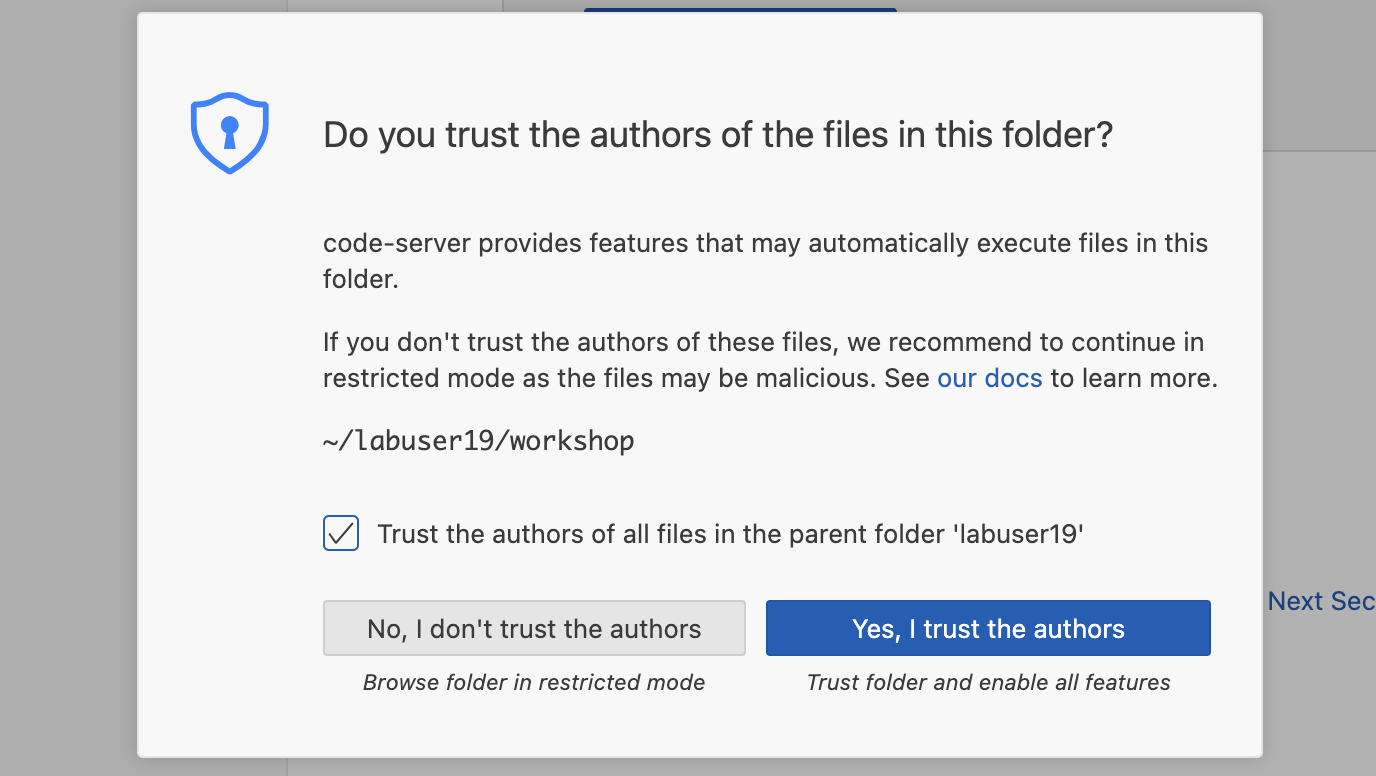
1. Choose a folder to clone to (use your allocated username folder name)



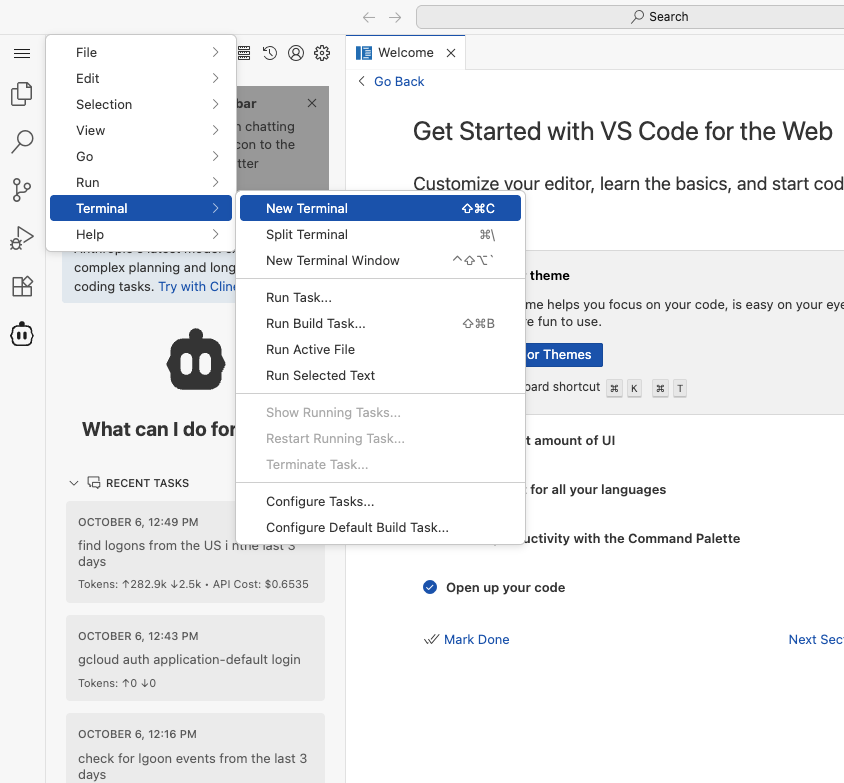
1. Choose “Open”



1. Choose **Yes, I trust the authors**



1. Enter your GitHub details so you can commit and push changes
2. Open New terminal - menu, **Terminal** - **New Terminal**

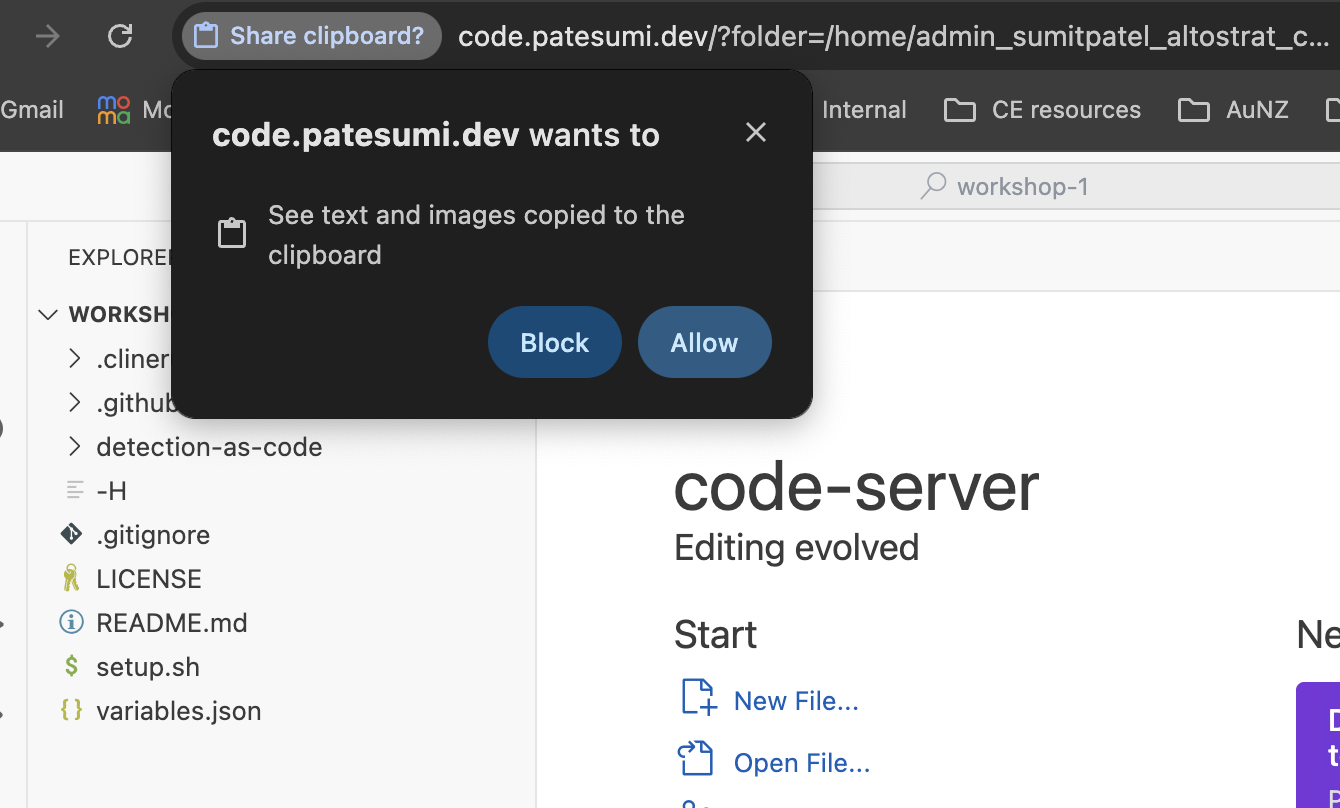


1. Use your name & email you used for GitHub.

Enter the following:

| git config --global user.name "Your Name"  git config --global user.email "youremail@example.com" |
| --- |

*Note: you may be asked to allow clipboard - choose Allow.*

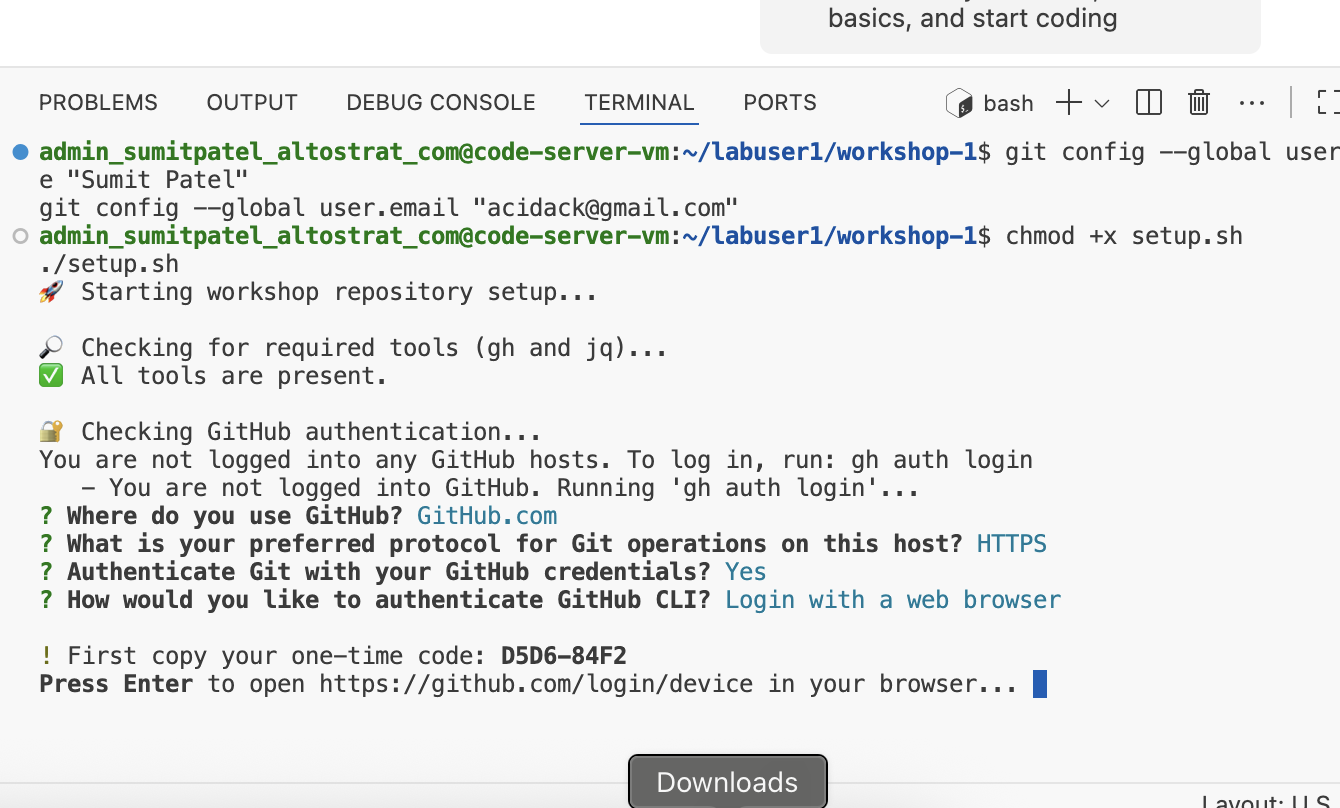


8. **Run the Setup Script:** This script will check for required tools and automatically set up all the necessary GitHub Actions variables in your forked repository.

1. In terminal enter:

| chmod +x setup.sh ./setup.sh |
| --- |

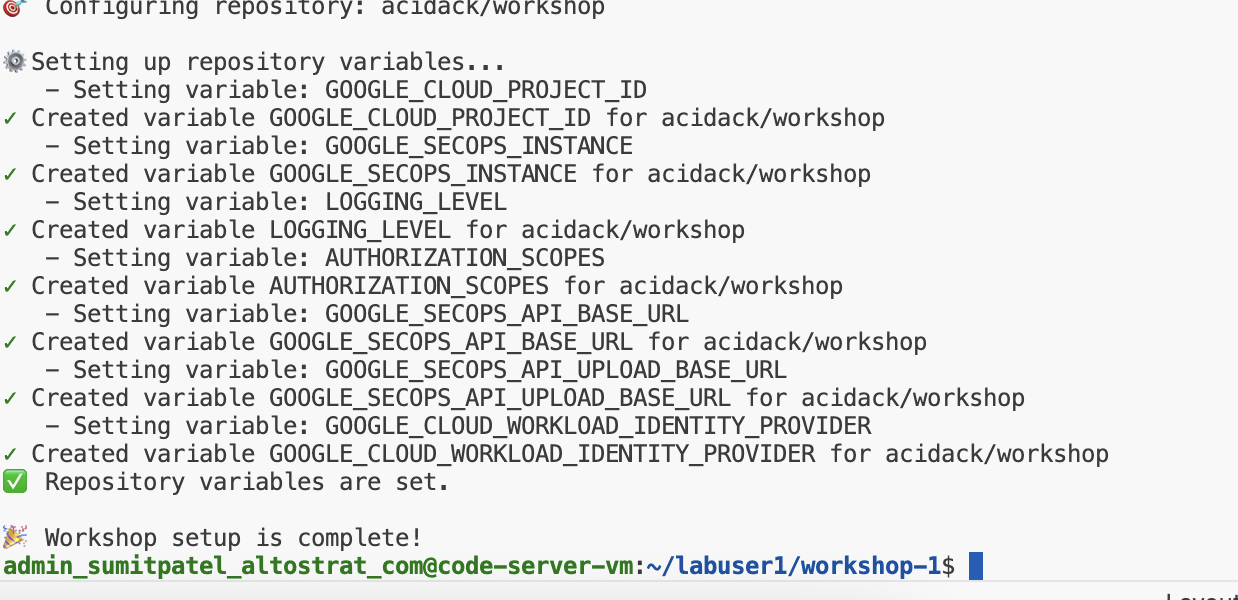
1. Authenticate when asked ([***Github.com***](http://github.com)***, HTTPS, Yes, Login with a browser***) :



1. **Authorize github** with the one-time code

### 

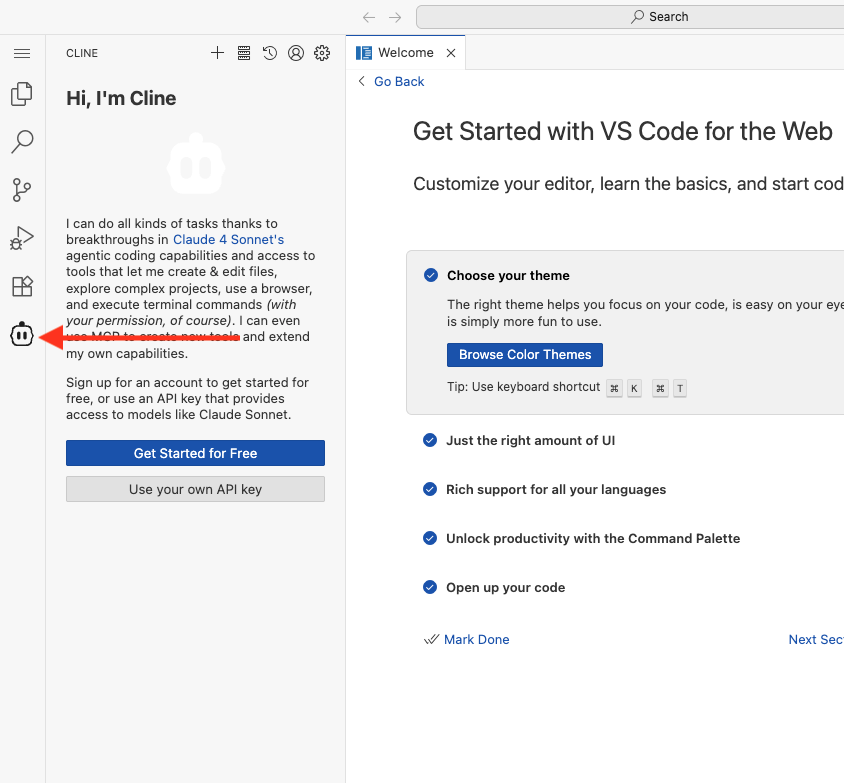
1. If successful it should have created the variables in your github repo



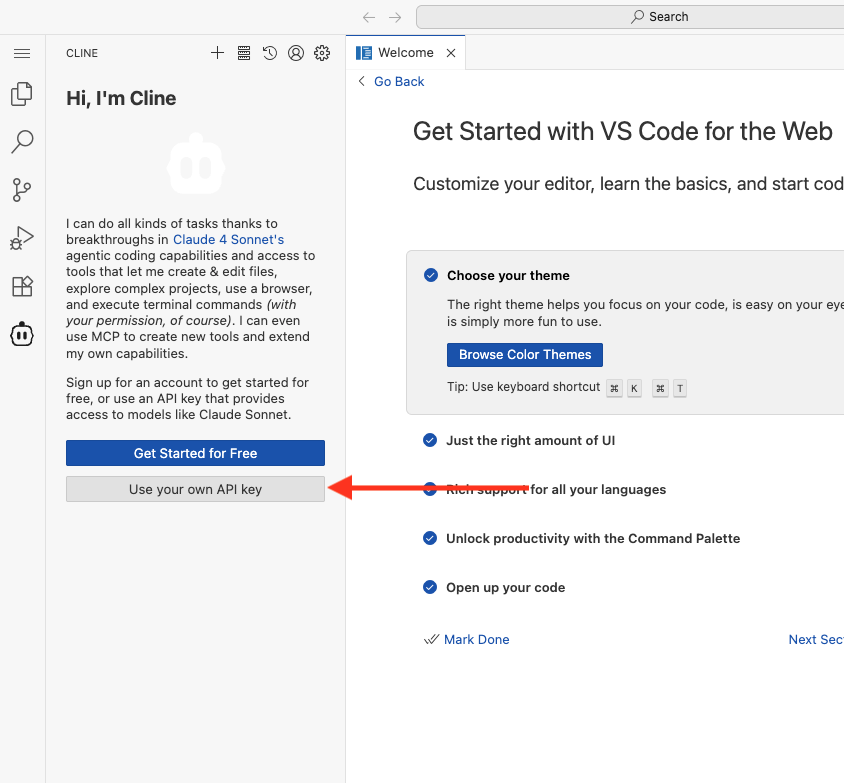
## 

## **Setting up AI on VS Code**

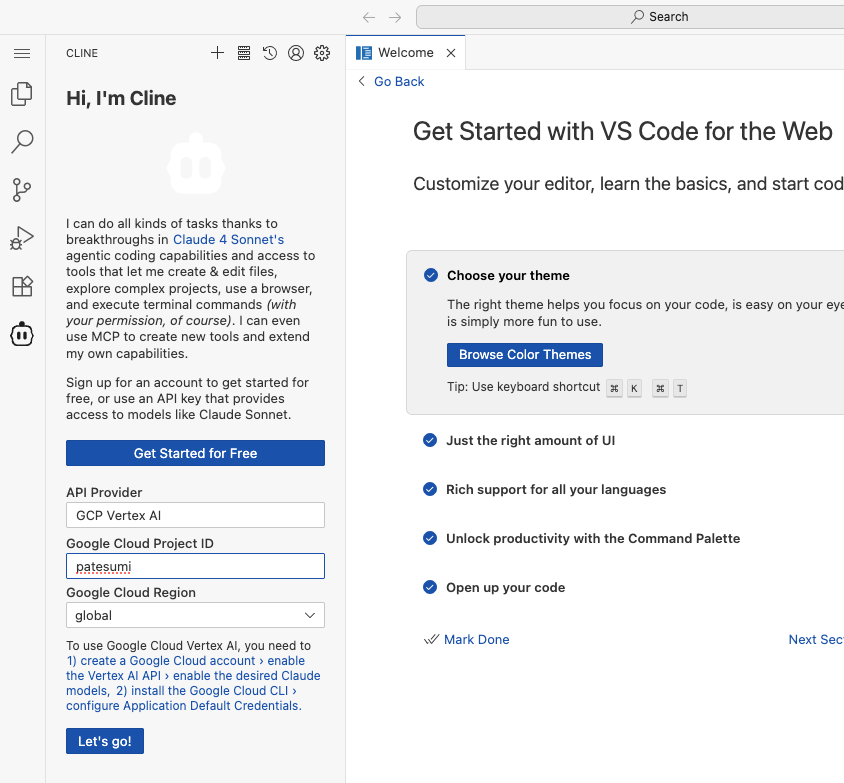
1. Select Cline extension from menu on left



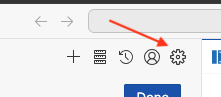
1. Select “**Use your own API key**”

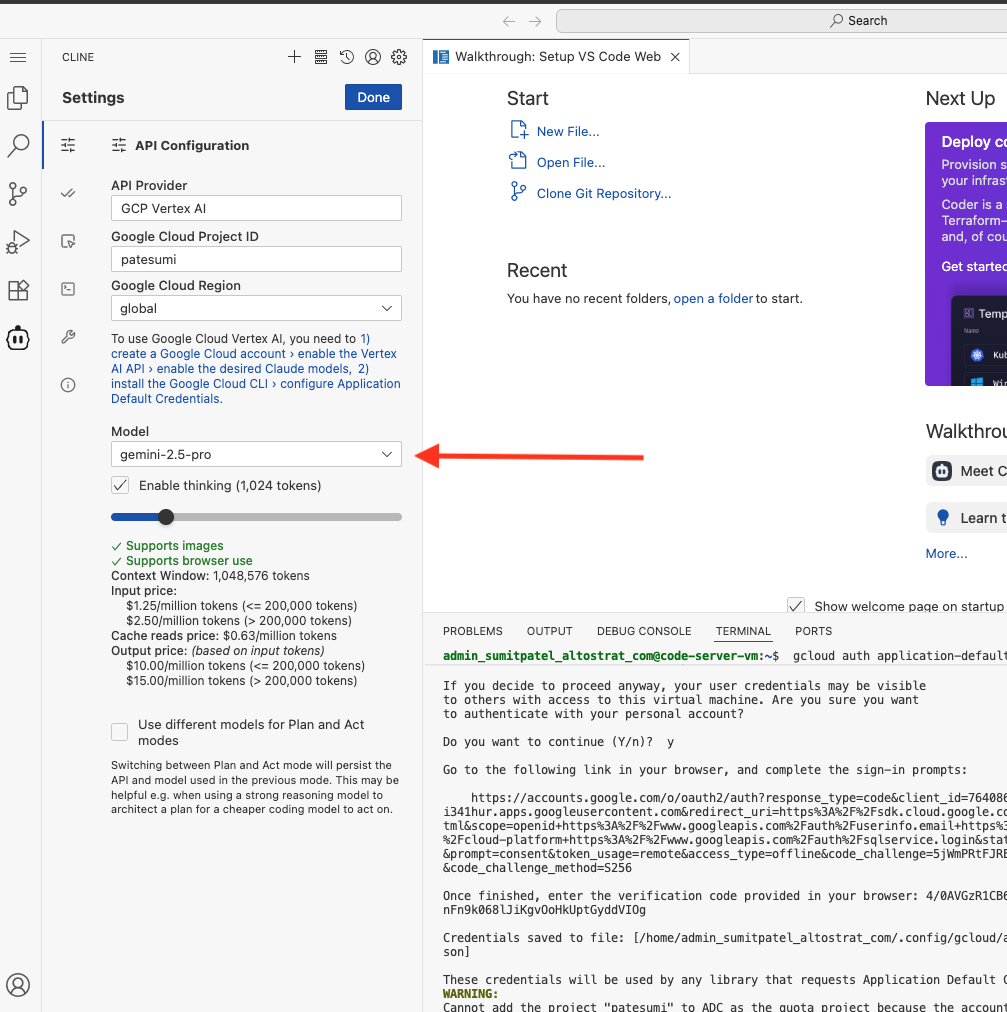


1. Select “**GCP Vertex AI**”, project ID: **patesumi**, region: **global**
2. Choose **“Lets go!”**



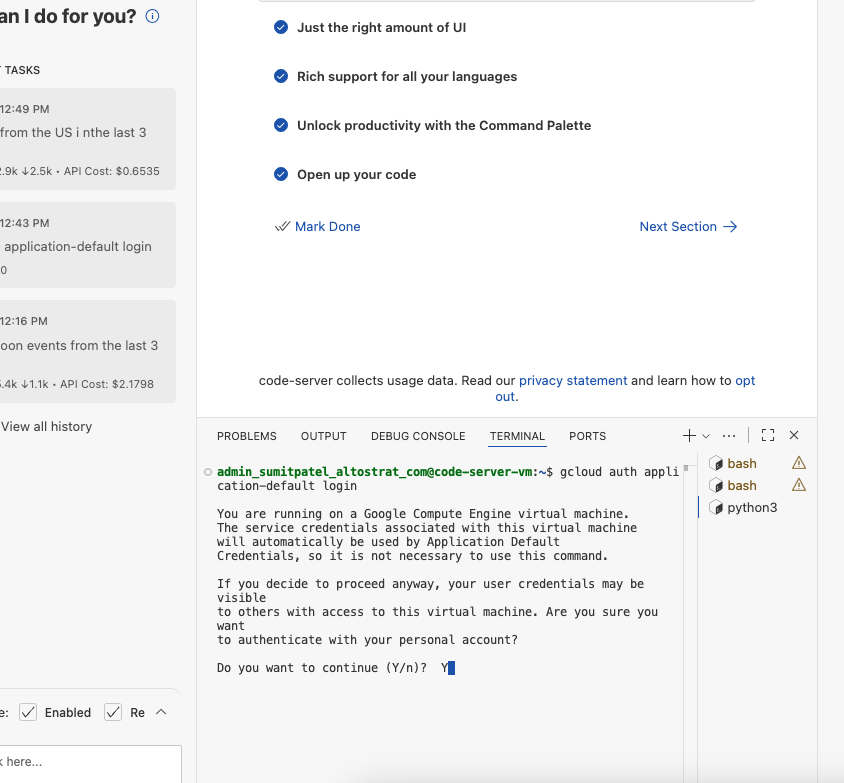
1. Next, select the Settings icon and select **gemini-2.5-pro** as the Model



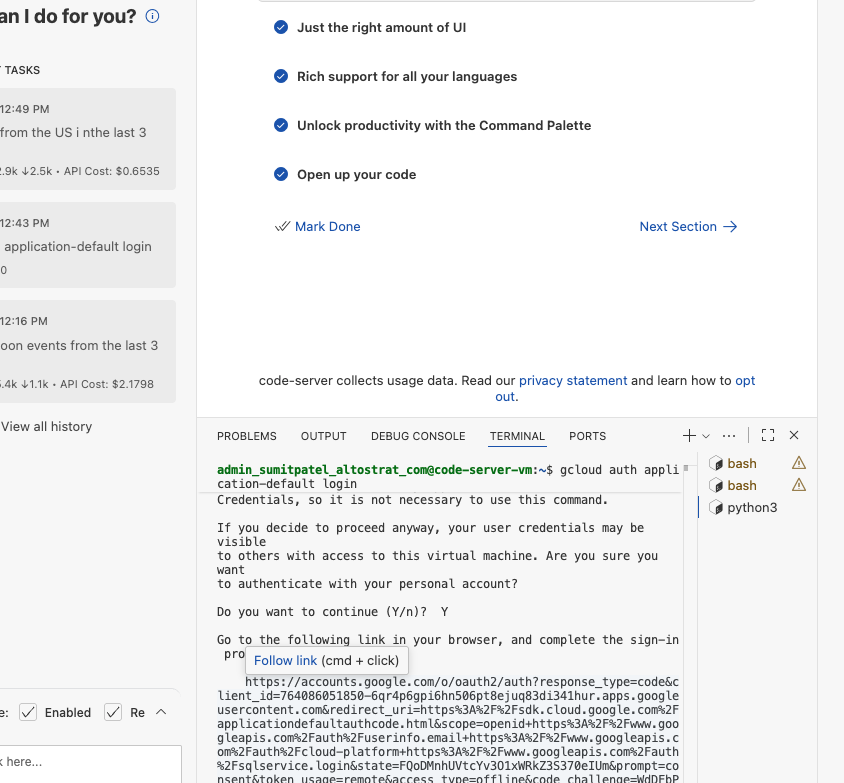


1. Go back to your Terminal. Type gcloud auth login

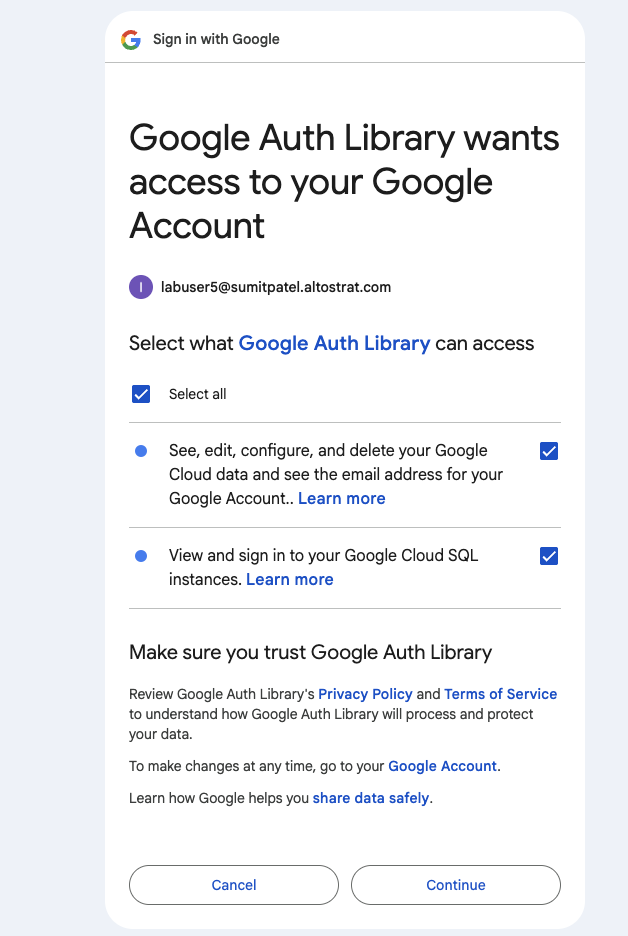
Hit enter, then type Y



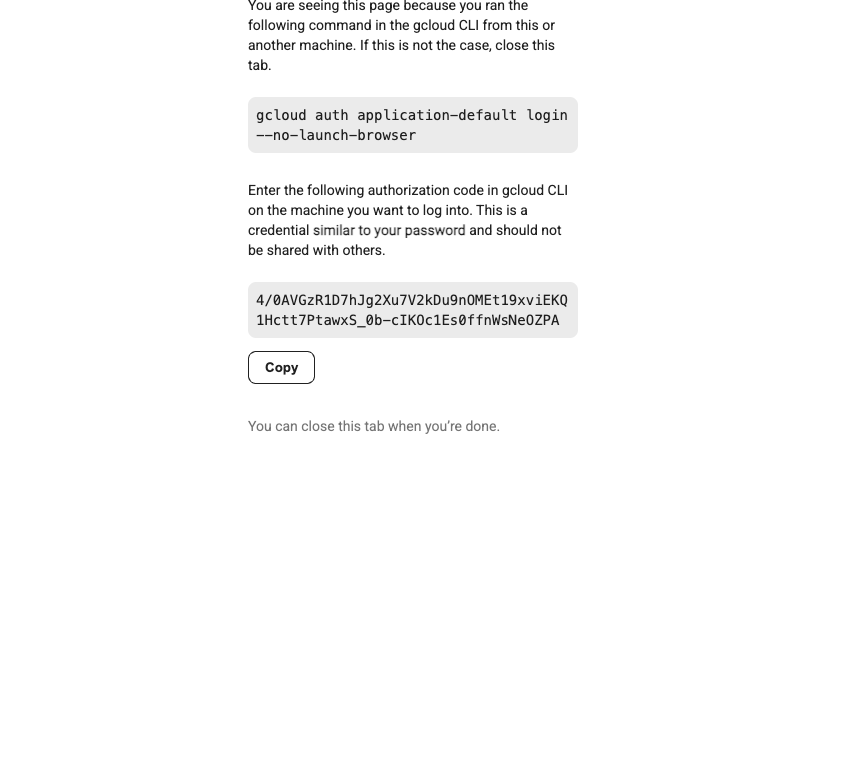
1. Open the web link



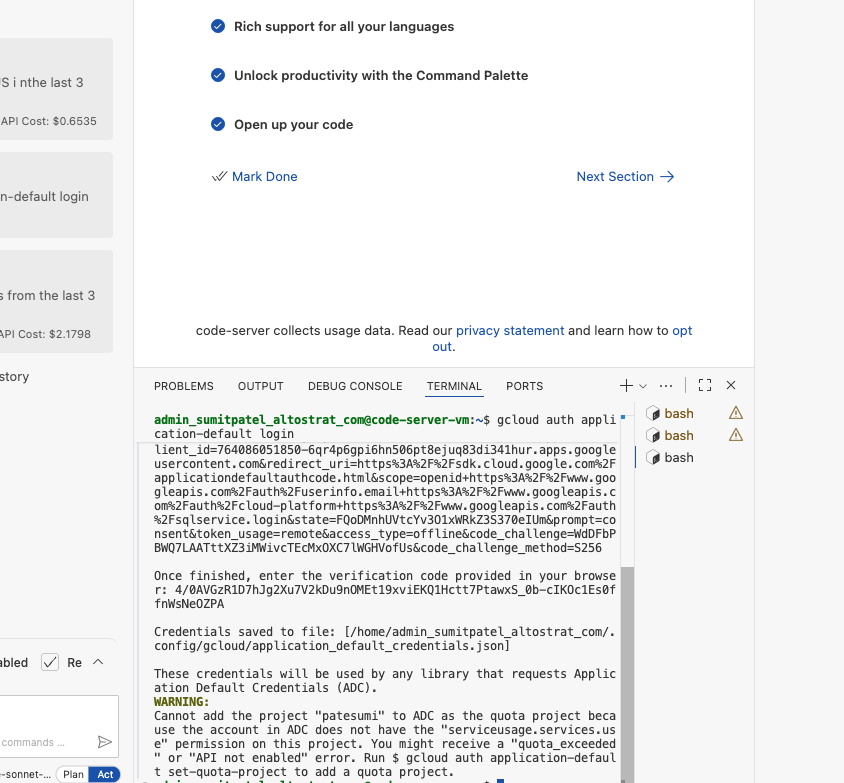
1. Sign in with your credentials from step 1, tick **Select all** and continue



1. Copy the unique code



1. Copy & paste it back in terminal

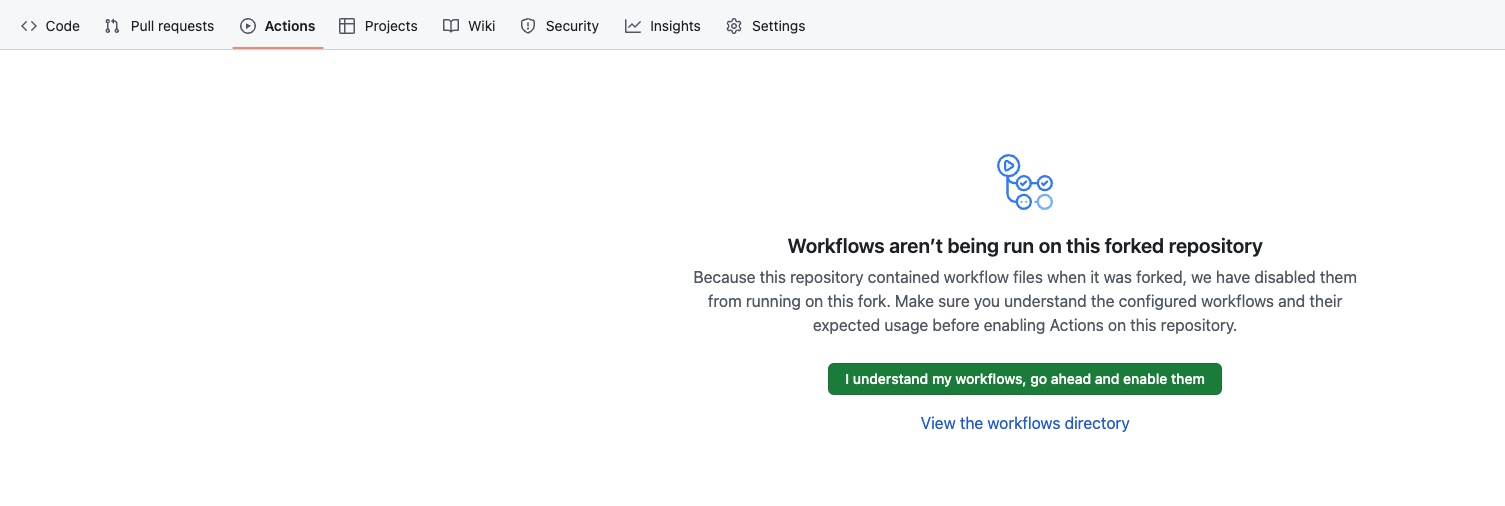


## 

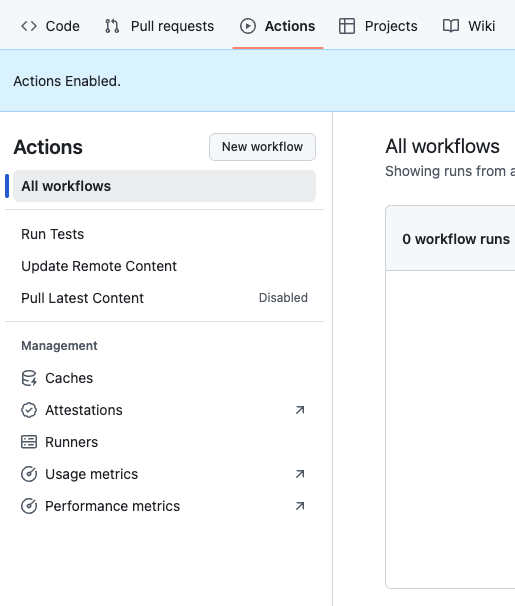
## 

### **Configure GitHub Actions & Variables/Secrets**

1. Go back into Github and choose **Actions**, and choose “**I understand my workflows, go ahead and enable them**”

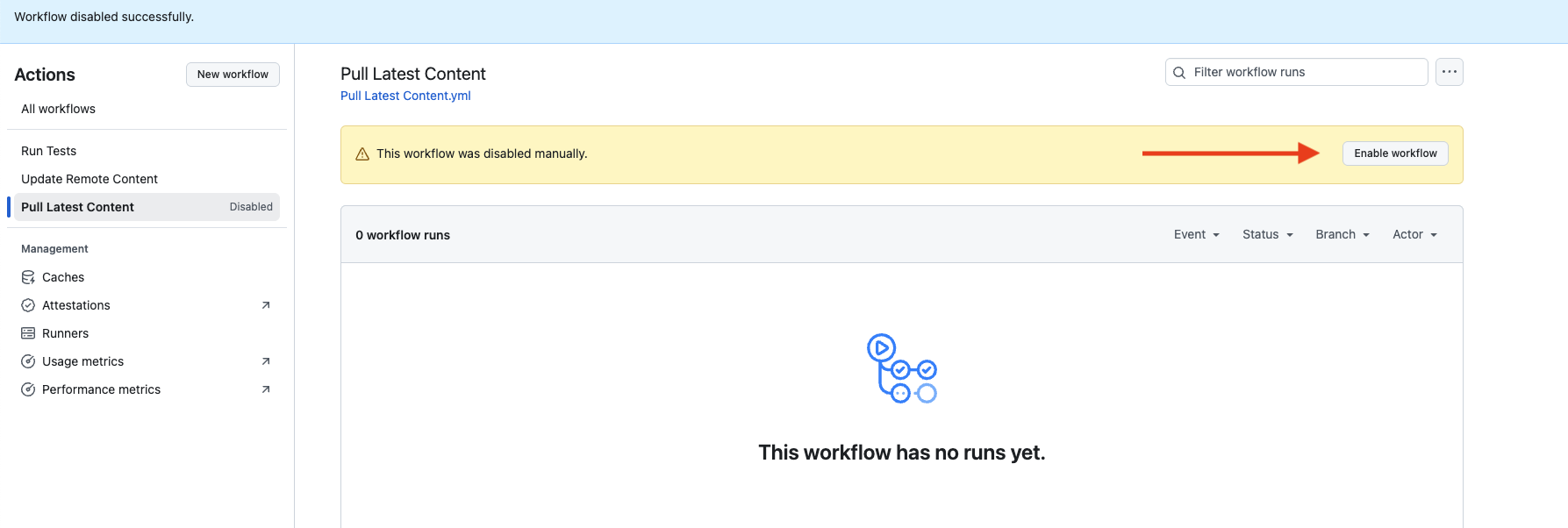


1. Confirm that the 3 Github actions are listed already as below



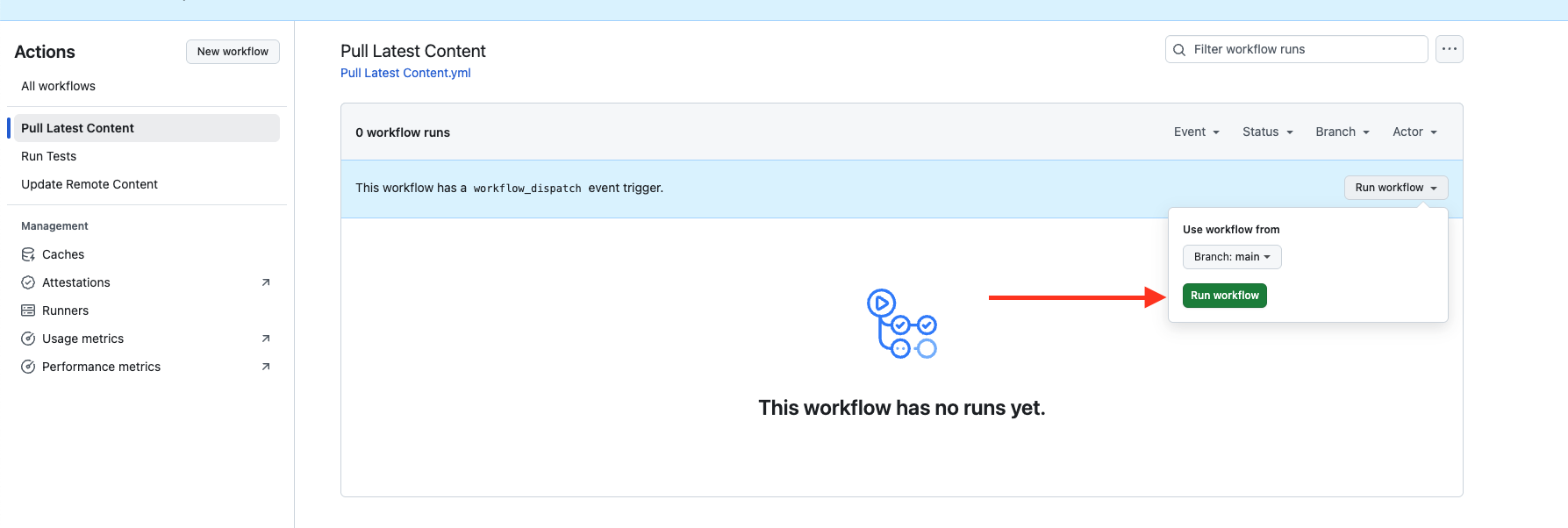
* Pull Latest Content
* Run Tests
* Update Remote Content

1. Click on **Pull Latest Content** action. Now select “Enable workflow”

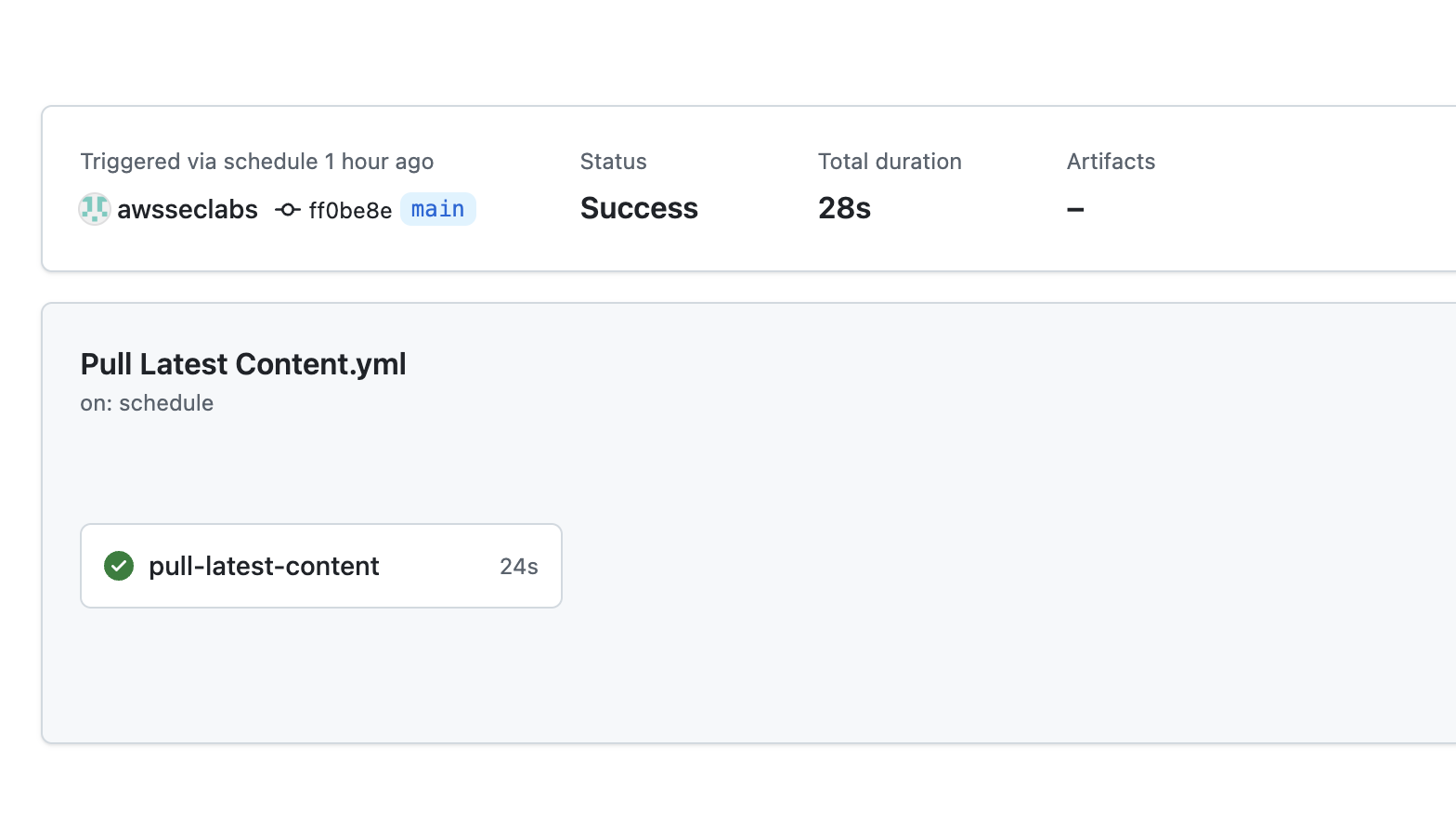
****

### **Testing the Github Actions - 1. Pull Latest Content**

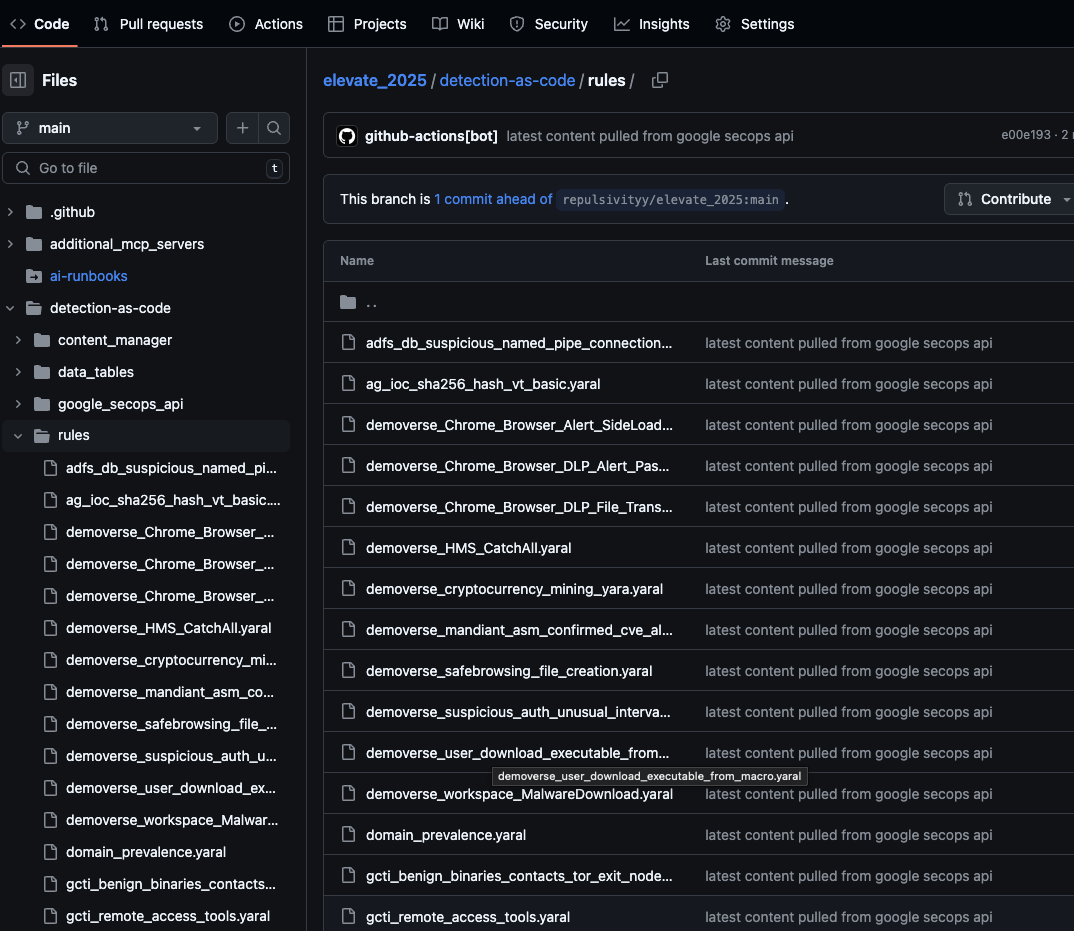
1. Select **Pull Latest Content** *(if not already selected)*
2. Lets now kick off the “**Pull Latest Content**” action to pull down rules / data tables / reference lists. Go into Actions, Choose “**Run workflow**”.



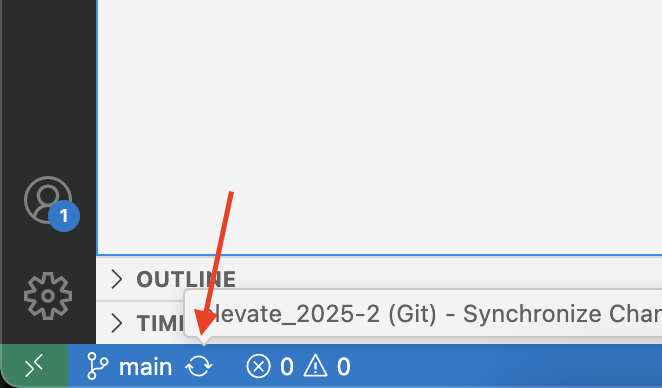
1. If you refresh the screen, it should show the “Pull Latest Content” action run. If successful it will turn green.



1. On completion of the above action, in your Github repo and in your local repo in VS Code (once it syncs), you should see all your rules



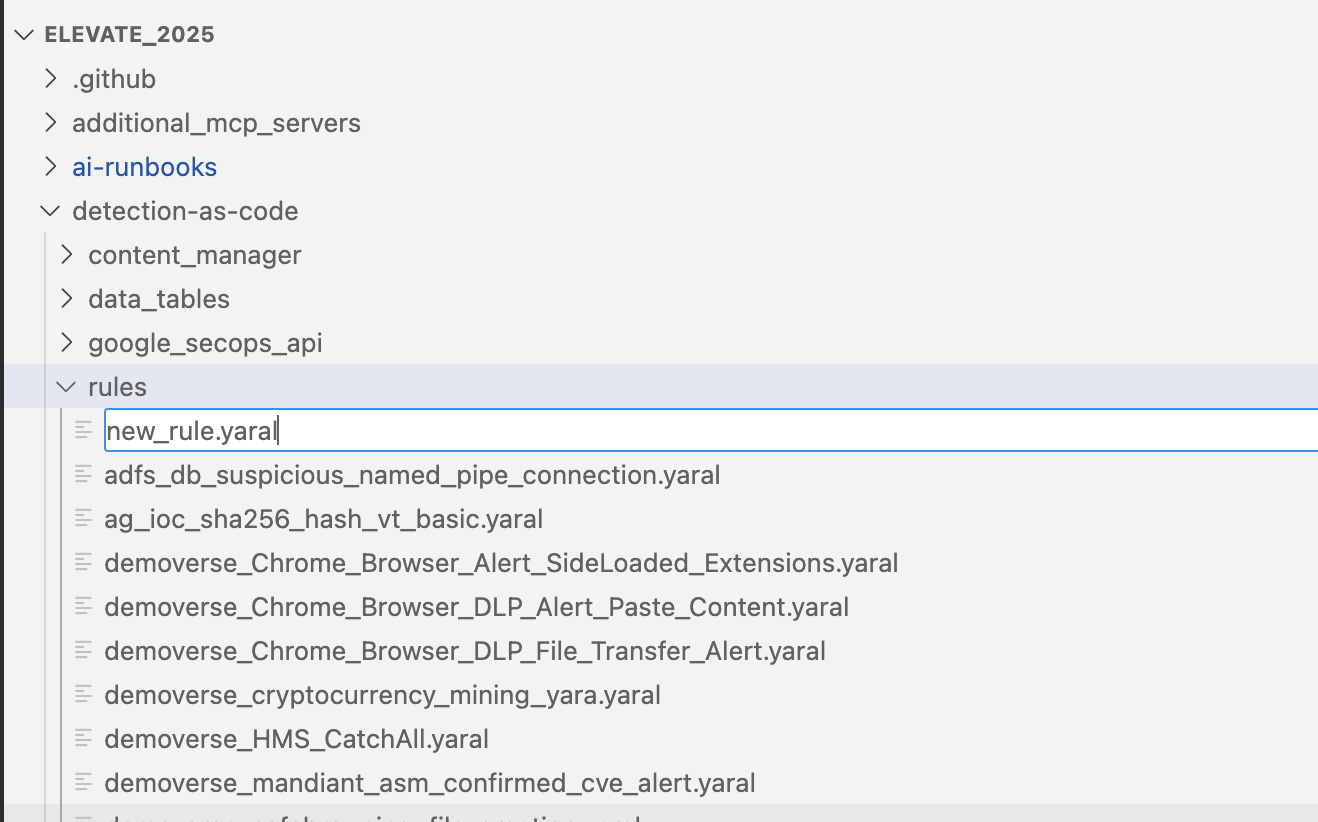
1. You can now try testing Detection as Code! Lets sync the content in VS Code with our repo as shown below:



1. If successful, you should see the rules within the detection-as-code -> rules folder.

### **Testing the Github Actions - 2. Creating a new rule**

1. Now let's create a new yara-l rule. Go into VS Code, look under detection-as-code - > rules folder. Create your new rule in this folder.



1. Use a file name of “**labuser**[#]**newrule**.yaral” (replace [#] with your allocated user number). This way we have a unique file name amongst all Workshop attendees.
2. In the content of the file, use the following (replace “[#]” with your allocated user number):

rule **labuser[#]newrule.yaral** {

// This rule matches single events. Rules can also match multiple events within

// some time window. For details about how to write a multi-event rule, see

// https://cloud.google.com/chronicle/docs/detection/yara-l-2-0-overview#single-event\_versus\_multi-event

meta:

// Allows for storage of arbitrary key-value pairs of rule details - who

// wrote it, what it detects on, version control, etc.

// The "author" and "severity" fields are special, as they are used as

// columns on the rules dashboard. If you'd like to be able to sort based on

// these fields on the dashboard, make sure to add them here.

// Severity value, by convention, should be "Low", "Medium" or "High"

author = "analyst123"

description = "8:00 AM local time"

severity = "Medium"

events:

$e.metadata.event\_type = "USER\_CREATION"

outcome:

// For a multi-event rule an aggregation function is required

// e.g., risk\_score = max(0)

// See https://cloud.google.com/chronicle/docs/detection/yara-l-2-0-overview#outcome\_conditionals\_example\_rule

$risk\_score = 0

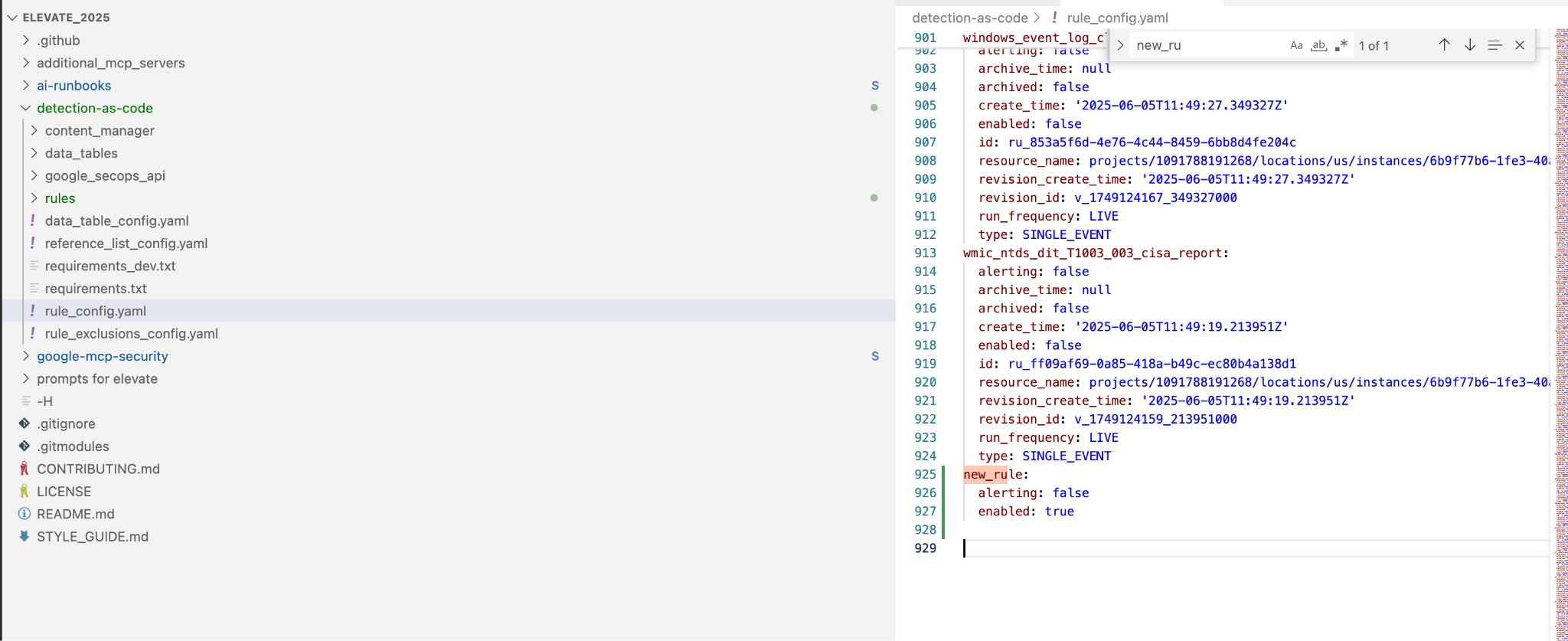
condition:

$e

}

Hit save on the file, after adding your new rule.

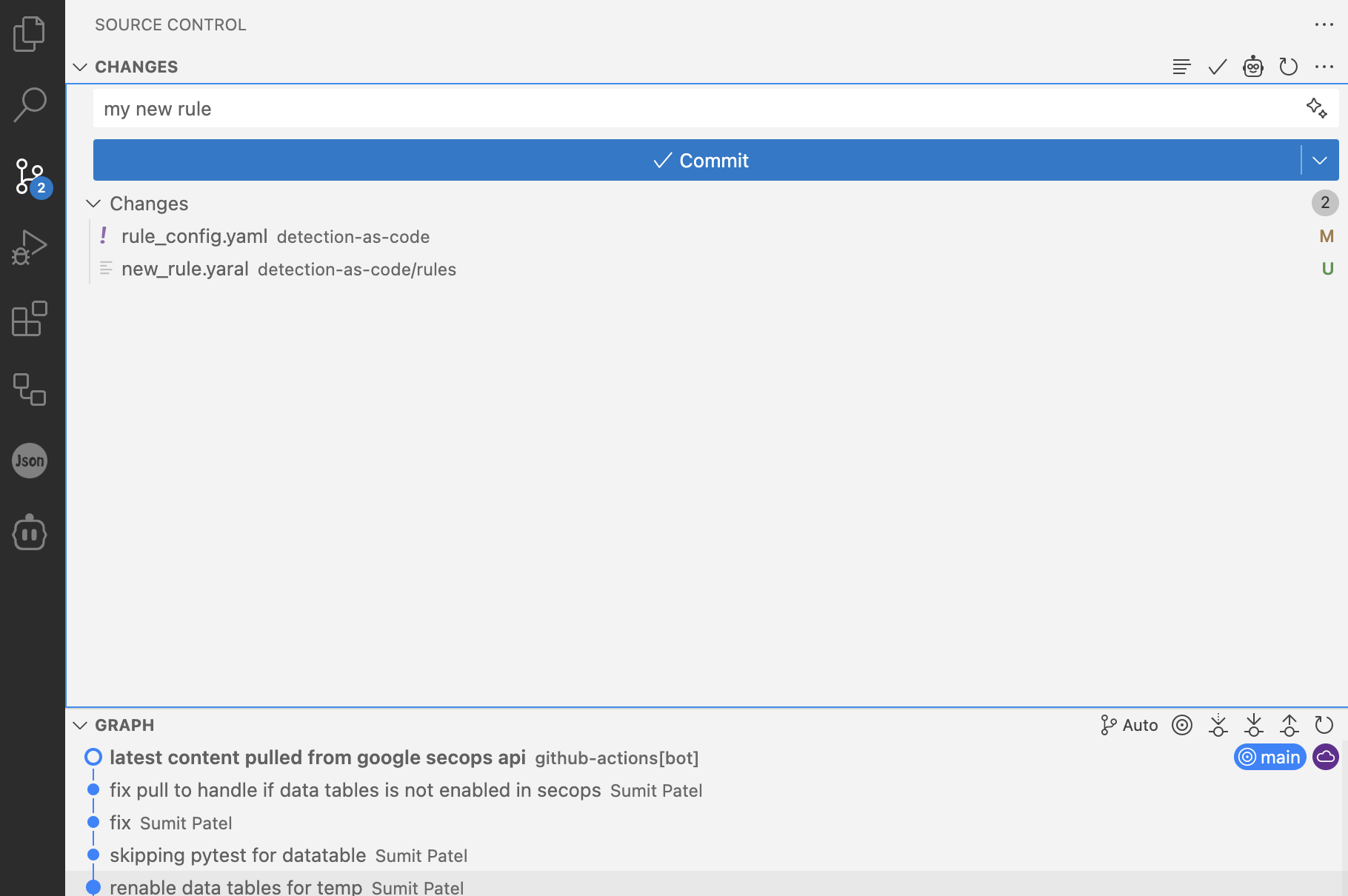
1. Now add an entry to detection-as-code -> rule\_config.yaml file and save.



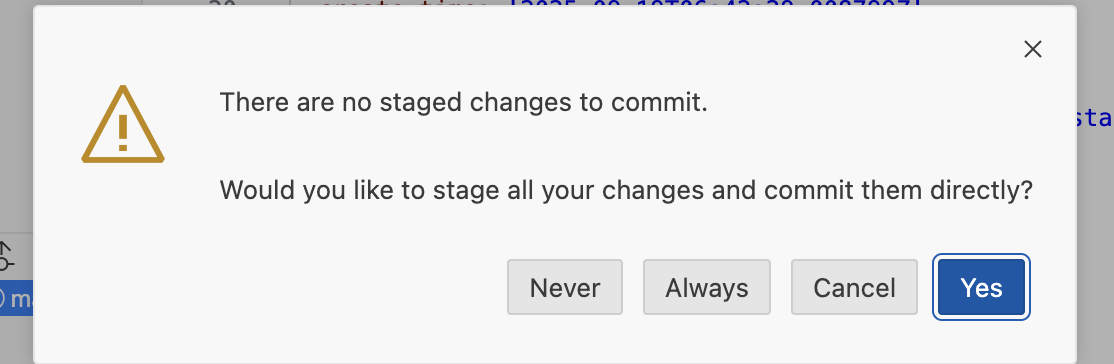
Example (replace “[#]” with your allocated user number):

| labuser[#]newrule:  alerting: false  enabled: true |
| --- |

1. Hit the “Source Control” menu item on the left, and add a comment e.g. “my new rule” and hit “**Commit**”.



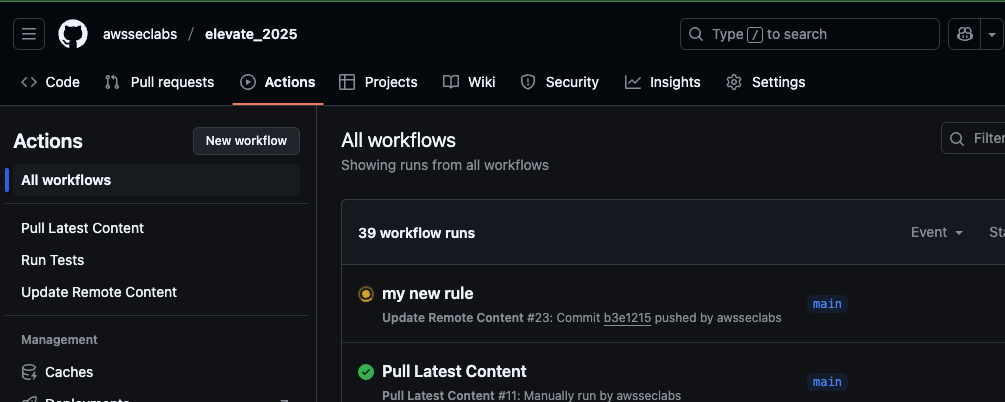
1. If it asks you stage changes, select Yes.



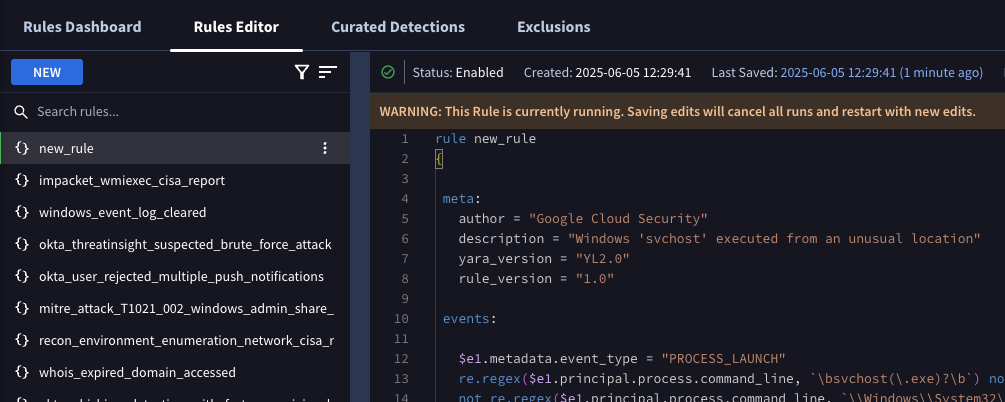
1. Let's now “**Sync changes**”.



1. Back in Github, it now should kick off a bunch of tests via GitHub Actions,



1. On successful completion, you should see your new rule in the SIEM - you can log in via <https://gglshy.backstory.chronicle.security/rulesEditor> & use the same credentials you used for vs code..

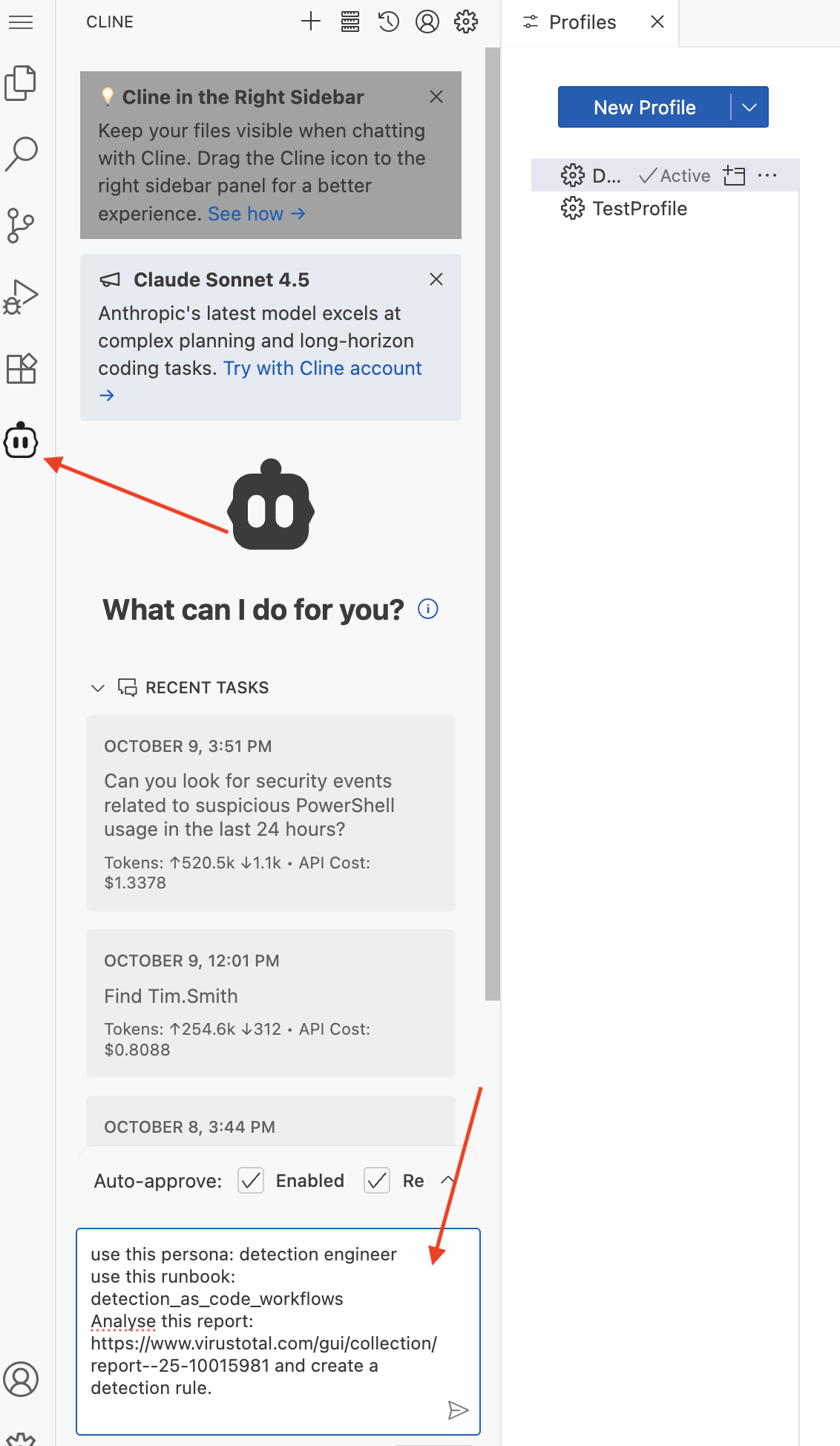


*Note: the rules\_config.yaml file should also now be updated with additional fields about our new yara-l rule e.g. its ID, creation time etc.*

# 

## **Using Model Context Protocol (MCP) servers in your IDE to Automate Threat Hunting**

1. Select the cline extension on the menu on the left
2. Enter your prompt and hit enter



### Example Prompts:

* Can you search for information about the Emotet malware family?

*The LLM will use the GTI server to search for and retrieve information about the Emotet malware family, including related IoCs, campaigns, and threat actor information.*

* Can you look for security events related to suspicious PowerShell usage in the last 24 hours?

*The LLM will use the Chronicle SecOps server to search for security events matching this description and present the findings.*

* Can you list open security cases and show me details about the highest priority one?

*The LLM will use the SecOps SOAR server to list open cases and provide details about the highest priority case.*

### Prompt to automate threat hunting with a detection rule based on a report

| **use this persona: detection engineer use this runbook: detection\_as\_code\_workflows Analyse this report: https:*//www.virustotal.com/gui/collection/report--25-10015981 and create a detection rule.*** |
| --- |

# 

# **Appendix: Best Practices working with Git**

### Commit early and often (and Small!)

* **Atomic Commit**s: Each commit should represent a single, logical change. If you're fixing two different bugs, make two separate commits. If you refactor code and add a new feature, do them in separate commits. This makes it much easier to understand, review, and revert changes if needed.
* **Frequent Commits**: Don't wait until you've completed a huge chunk of work. Commit small, testable increments. This reduces the risk of large, messy merge conflicts and allows you to easily track your progress.

### Write Good Commit Messages

* **Clear and Concise Subject Line:** The first line of your commit message should be a brief, imperative summary (around 50 characters). It should answer "What did this commit do?" (e.g., "Fix: Login button not responding").

### Stay Up-to-Date and Manage Conflicts

* **Pull Before You Push:** Always pull the latest changes from the remote repository (git pull or git fetch and then git rebase/git merge) before pushing your own changes. This helps you integrate upstream changes early and resolve conflicts locally.
* **Understand git merge vs. git rebase:**
  + **git merge:** Combines histories, creating a merge commit. This preserves the exact history of branches.
  + **git rebase:** Rewrites history by moving your commits "on top" of the target branch. This creates a clean, linear history, but **never rebase a public branch** that others are already working on, as it can cause significant issues for collaborators. Use it for your local feature branches before merging into main.
* **Resolve Conflicts Promptly:** When merge conflicts occur, don't panic. Git highlights the conflicting sections. Understand the changes from both sides and resolve them carefully.

### Keep Your Repository Clean

* **.gitignore File:** Use a .gitignore file to prevent untracked files (like compiled code, temporary files, IDE configurations, node\_modules, sensitive API keys, etc.) from being accidentally committed.
* **Clean Up Old Branches:** Once a feature branch is merged and no longer needed, delete it from both your local and remote repositories to keep the branch list clean.
* **Regular Maintenance:** Periodically review and clean up your commit history if necessary (e.g., using git rebase -i to squash small, related commits *on your local, unpushed branches*).