NCI CBIIT provides Enterprise GitHub services which is a complete developer platform to build, scale, and deliver secure software. GitHub Enterprise provides the following suite of products to support the entire software development lifecycle, increasing development velocity and improving code quality:

- Code Repository Management
- GitHub Communication Services
- GitHub Projects
- GitHub Pages
- GitHub Packages
- GitHub Actions
- GitHub Advanced Security
- Webhook based integrations
- GitHub Copilot

To simplify administration for all the stages in the software development lifecycle, CBIIT provides a single point of visibility and management through its GitHub Enterprise Account. Developers can store, and version control their source code in repositories leveraging GitHub Projects and Issues, to plan and track their work. Code changes can be reviewed, approved, or rejected with pull requests in addition to leveraging code security features to keep secrets and vulnerabilities out of the codebase. Finally, teams can automate software build, test, and deployment pipelines with GitHub Actions and host software packages with GitHub Packages.

Note: GitHub is not meant to be used as a hosting platform for hosting run time applications, and systems. GitHub should solely be used for software development management purposes only.

Source Code Management:

GitHub Source code management is used to track modifications to a source code repository. It tracks a running history of changes to a code base and helps resolve conflicts when merging updates from multiple contributors. Changes are reviewed, approved, or rejects using the pull requests capabilities provided by GitHub in addition teams are strongly encouraged to track code base changes using branching techniques against their main repository; this will ensure that only the changes that are reviewed and approved are merged into the main source control preventing conflicts and broken code from being promoted and deployed. For more information and best practices related to source code management, please refer to the following article: Source Code Management.

GitHub Communication Services:

GitHub provides built-in collaborative communication tools allowing development teams to interact closely with each other. Teams can create and participate in issues, pull requests, GitHub Discussions, and team discussions, depending on the type of conversation is needed. Below is a list of the various communication services provided by GitHub stating the purpose of each and usability use cases

- **GitHub Issues - Purpose:**

    - Are useful for discussing specific details of a project such as bug reports, planned improvements and feedback.
    - Are specific to a repository, and usually have a clear owner.
    - Are often referred to as GitHub's bug-tracking system.

    GitHub Issues – Scenarios:

    - Team wants to keep track of tasks, enhancements, and bugs.
    - Team wants to file a bug report.
    - Team wants to share feedback about a specific feature.
    - Team wants to ask a question about files in the repository.

- **GitHub Pull Requests – Purpose:**

    - Allows teams to review code of specific changes
    - Allows teams to propose specific changes.
    - Allows team members to comment directly on proposed changes suggested by others.
    - They are specific to a repository.

    GitHub Pull Requests – Scenarios:

    - A developer wants to fix code in a repository branch.
    - A developer wants to make changes to a repository.
    - A developer wants to make changes to fix an issue.
    - A developer wants to comment on changes suggested by others.

- **GitHub Discussions – Purpose:**

    - Are like a forum and are best used for open-form ideas and discussions where collaboration is important.
    - May span many repositories.
    - Provides a collaborative experience outside the codebase, allowing the brainstorming of ideas, and the creation of a community knowledge base.
    - often don’t have a clear owner.
    - often do not result in an actionable task.

    GitHub Discussions – Scenarios:

- A team member has a question that's not necessarily related to specific files in the repository.
- A team member wants to share news with my collaborators, or my team.
- A team wants to start or participate in an open-ended conversation.
- A team wants to make an announcement to the community of collaborators.

- **GitHub Team Discussions - Purpose:**

  - Can be started on a team's page for conversations that span across projects and don't belong in a specific issue or pull request. Instead of opening an issue in a repository to discuss an idea, A developer can include the entire team by having a conversation in a team discussion.
  - allow development teams to hold discussions relevant to their planning, analysis, design, user research and general project decision in one place.

GitHub Team Discussion – Scenarios:

- A team member has a question that's not necessarily related to specific files in the repository.
- A team member wants to share news with collaborators, or the team.
- Start or participate in an open-ended conversation.
- Make an announcement to the team.

Note: Team discussions are very similar to GitHub Discussions. It is recommended to use GitHub Discussions to collaborate with other communities on GitHub if needed. If the team and associated repositories are part of an organization and would like to initiate conversations within that organization or team within that organization, it is recommended to use team discussions.

GitHub Projects:

A project is an adaptable spreadsheet that integrates with team's issues and pull requests on GitHub to help plan and track work effectively. Teams can create and customize multiple views by filtering, sorting, grouping issues and pull requests, adding custom fields to track metadata specific to the team, and visualize work with configurable charts. Rather than enforcing a specific methodology, a project provides flexible features that are customizable to the team's needs and processes.

Projects are built from the issues and pull requests the team add, creating direct references between the project and the development work. Information is synced automatically to the project as changes are made, in addition to updating views and charts. This integration works both ways, so that when changing information for a pull request or issue in the project, the pull request or issue reflects that information. For example, changing an assignee in the project will

be reflected in the related issue. This integration can be taken even further where teams can group their project by assignee and make changes to issue assignment by dragging issues into the different groups.

Teams can use custom fields to add metadata to issues, pull requests, and draft issues and build a richer view of item attributes. Teams are not limited to the built-in metadata (assignee, milestone, labels, etc.) that currently exists for issues and pull requests. For example, teams can add the following metadata as custom fields:

- A date field to track given dates.
- A number field to track the complexity of a task.
- A single select field to track whether a task is Low, Medium, or High priority.
- A text field to add a quick note.
- An iteration field to plan work week-by-week, including support for breaks.

It is recommended to use **"Tasklists"** within your project to build hierarchies of issues, dividing issues into smaller subtasks, and creating new relationships between issues. These relationships are displayed on the issue, as well as the Tracked by and Tracks fields in projects. Teams can filter by issues which are tracked by another issue and can also group table views by the **"Tracked by"** field to show all parent issues with a list of their subtasks.

GitHub Pages

GitHub Pages is a static site hosting service that takes HTML, CSS, and JavaScript files straight from a repository on GitHub, optionally runs the files through a build process, and publishes a website.

There are three types of GitHub Pages sites: **project, user, and organization**. Project sites are connected to a specific project hosted on GitHub, such as a JavaScript library. User and Organization sites are connected to a specific account on GitHub.com.

To publish a user site, the team must create a repository owned by a GitHub account that's named **<username>.github.io**.

If needed, to publish an organization site, the team must create a repository owned by an organization that's named **<organization>.github.io.**

The source files for a project site are stored in the same repository as their project.

Only one user or organization site can be created for each account on GitHub. Project sites, whether owned by an organization or a personal account, are unlimited.

Note: GitHub Pages sites are publicly available on the internet, even if the repository for the site is private. It is recommended not to publish any sensitive data in the site's repository.

Teams can publish a site when changes are pushed to a specific branch or can write a GitHub Actions workflow to publish the site. If a team do not need any control over the build process for their site, it is recommended that the site gets published when changes are pushed to a specific branch. Teams can specify which branch and folder to use as the publishing source. The source branch can be any branch in the repository, and the source folder can either be the root of the repository (/) on the source branch or a /docs folder on the source branch. Whenever changes are pushed to the source branch, the changes in the source folder will be published to the GitHub Pages site.

If you want to use a build and do not want a dedicated branch to hold your compiled static files, it is recommended that you write a GitHub Actions workflow to publish your site. GitHub provides starter workflows for common publishing scenarios to help teams write their workflow.

GitHub Pages is **NOT** intended for or allowed to be used as a free web-hosting service to host transactional applications, or any other website that is primarily directed at either facilitating commercial transactions or providing commercial software as a service (SaaS). GitHub Pages sites shouldn't be used for sensitive transactions either.

GitHub Packages:

GitHub Packages is a platform for hosting and managing packages, including containers and other dependencies. GitHub Packages combines source code and packages in one place to provide integrated permissions management, so software development activities can be centralized on GitHub.

GitHub Packages can be integrated with GitHub APIs, GitHub Actions, and webhooks to create an end-to-end DevOps workflow that includes code, CI, and deployment solutions.

GitHub Packages offers different package registries for commonly used package managers, such as npm, Apache Maven, Gradle, and Docker. GitHub's Container registry is optimized for containers and supports Docker and OCI images.

Package permissions and visibility

The permissions for a package are either inherited from the repository where the package is hosted or can be defined for specific user or organization accounts. Some registries only support permissions inherited from a repository.

Visibility

Teams can publish packages in a public repository (public packages) to share with all of GitHub, or in a private repository (private packages) to share with collaborators or an organization.

Supported Clients & Formats

Language	Description	Package Format	Package Client
JavaScript	Node Package Manager	package.json	npm
Java	Apache Maven project management and comprehension tool	pom.xml	mvn
Java	Gradle build automation tool for Java	build.gradle or build.gradle.kts	gradle
N/A	Docker container management	Dockerfile	Docker

Authenticating to GitHub Packages

The team will need an access token to publish, install, and delete private, internal, and public packages.

Individual team members can use a personal access token (classic) to authenticate to GitHub Packages or the GitHub API. To authenticate to a GitHub Packages registry within a GitHub Actions workflow, the following options are available:

- GITHUB_TOKEN to publish packages associated with the workflow repository.
- a personal access token (classic) with at least packages:read scope to install packages associated with other private repositories (which GITHUB_TOKEN can't access).

Managing Packages

Packages are managed in the GitHub user interface or using the REST API. For certain registries, teams can use GraphQL* to delete a version of a private package.

Definition: GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data – GraphQL is a common query language for leveraging GitHub APIs.

When you use the GraphQL API to query and delete private packages, the developer must use the same personal access token (classic) used to authenticate to GitHub Packages.

Teams have also the option to configure webhooks which are discussed later in this document to subscribe to package-related events, such as when a package is published or updated.

NOTE: To keep better manage packages storage pertaining to private repositories within an organization, please refer to the caching strategies recommendations listed under the **Github Actions** section below.

GitHub Actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows development teams to automate their build, test, and deployment pipeline. Teams can create workflows that build and test every pull request against a given repository or deploy merged pull requests to subsequent environments.

Teams can configure a GitHub Actions *workflow* to be triggered when an *event* occurs in a repository, such as a pull request being opened, or an issue being created. The workflow contains one or more *jobs* which can run in sequential order or in parallel. Each job will run inside its own virtual machine *runner*, or inside a container, and has one or more *steps* that either run a script that you define or run an *action*, which is a reusable extension that can simplify your workflow.

In most cases, runners are hosted by GitHub and are automatically leveraged by GitHub Actions when an event is triggered. GitHub provides Linux, Windows, and macOS virtual machines to run workflows, or teams can host their own self-hosted runners, if need be, when access to GitHub hosted runners is not permitted.

Workflow

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to the repository and will run when triggered by an event in that repository, or they can be triggered manually, or at a defined schedule.

Workflows are defined in the **".github/workflows"** directory in a repository, and a repository can have multiple workflows, each of which can perform a different set of tasks. For example, a team can have one workflow to build, and test pull requests, another workflow to deploy the application every time a release is created, and still another workflow that adds a label every time someone opens a new issue. For additional details please refer to the following document: GitHub Workflows.

Events:

An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository. Teams can also trigger a workflow run on a schedule, or manually. For additional details please refer to the following document: GitHub Workflow Events.

Jobs:

A job is a set of *steps* in a workflow that execute on the same runner. Each step is either a shell script that will be executed, or an *action* that will be run. Steps are executed in order and are dependent on each other. Since each step is executed on the same runner, the team can share data from one step to another. For example, the workflow can have a step that builds the application followed by a step that tests the application that was built.

Teams can configure a job's dependencies with other jobs; by default, jobs have no dependencies and run in parallel with each other. When a job takes a dependency on another job, it will wait for the dependent job to complete before it can run. For example, a team may have multiple build jobs for different architectures that have no dependencies, and a packaging job that is dependent on those jobs. The build jobs will run in parallel, and when they have all completed successfully, the packaging job will run. For additional details please refer to the following document: Using Jobs.

Actions:

An action is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files. An action can pull your git repository from GitHub, set up the correct toolchain for your build environment, or set up the authentication to your cloud provider.

You can write your own actions, or you can find actions to use in your workflows in the GitHub Marketplace.

For additional details please refer to the following document – Creating Actions.

Runners:

A runner is a server that runs your workflows when they're triggered. Each runner can run a single job at a time. GitHub provides Ubuntu Linux, Microsoft Windows, and macOS runners to run workflows; each workflow run executes in a fresh, newly provisioned virtual machine hosted by GitHub." If you need a different operating system or require a specific hardware configuration, or if you have a use case for running long processes through GitHub actions against a private repository, or if your environment/organization prohibits access to the GitHub hosted runners, teams have the option to host their own runners. For more information about self-hosted runners, please refer to the following document: Hosting your Own Runners.

Storing Secrets in GitHub workflows

If the team's workflows use sensitive data, such as passwords, certificates, access keys, etc. It is recommended from as a best practice to save these in GitHub as *secrets* and then use them in workflows as environment variables. This means that teams will be able to create and share workflows without having to embed sensitive values directly in the workflow's YAML source. For more information regarding using GitHub Secrets, please refer to the following document: Encrypted Secrets.

GitHub Actions Utilization & Recommendations

CBIIT currently has 50,000 available actions minutes per month that are shared across organizations under the Enterprise account. These minutes are consumed with Actions pipelines build against **Private** repositories in each organization. Below are some recommendations to ensure utilization does not exceed the minutes limit threshold.

If your team's business requirement states that a single or multiple or most of the repos within a given org should be set to be **Private,** then the following should be considered:

- Build your pipeline(s) & thoroughly test and monitor its execution while noting the overall execution time irrespective of the pipeline trigger design (e.g., parallel execution on each commit, pull request, etc.)
- If the execution time exceeds an hour (60 minutes) for a given pipeline; then revise your pipeline design, refactor, and retest.
- If the execution time remain the same and it is accurate per design requirement (e.g., executing automated test cases as part of a pipeline execution or running CodeQL scans against the full code base) then these pipeline jobs are considered long running processes and could impact the overall shared pool of minutes if they are executed frequently on daily basis; therefore, the development team should consider the following options in this case:
    - Refactor your pipeline configuration from leveraging the GitHub Actions hosted runner to a self-hosted runner **as minutes won't be consumed leveraging self-hosted runners against private repositories.**
    - Validate from a business requirements standpoint whether this repository must remain **Private** and if not; then consider switching the repository visibility to **Public** as GitHub actions minutes are not consumed against Public Repos.

Key Take aways on utilization:

- Minutes are not consumed for Pipelines built on Public Repositories
- Minutes are not consumed for Pipelines leveraging self-hosted and managed runners.

Caching Strategies

This section lists some of the strategies (and example workflows if possible) which can be used to ...

- use an effective cache key and/or path
- solve some common use cases around saving and restoring caches
- leverage the step inputs and outputs more effectively

Choosing the right key

```
jobs:
  build:
    runs-on: ubuntu-latest
    - uses: actions/cache@v3
      with:
        key: ${{ some-metadata }}-cache
```

In your workflows, you can use different strategies to name your key depending on your use case so that the cache is scoped appropriately for your need. For example, you can have cache specific to OS, or based on the lockfile or commit SHA or even workflow run.

Updating cache for any change in the dependencies

One of the most common use case is to use hash for lockfile as key. This way, same cache will be restored for a lockfile until there's a change in dependencies listed in lockfile.

```
  - uses: actions/cache@v3
    with:
      path: |
        path/to/dependencies
        some/other/dependencies
      key: cache-${{ hashFiles('**/lockfiles') }}
```

Using restore keys to download the closest matching cache

If cache is not found matching the primary key, restore keys can be used to download the closest matching cache that was recently created. This ensures that the build/install step will need to additionally fetch just a handful of newer dependencies, and hence saving build time.

```
  - uses: actions/cache@v3
    with:
      path: |
        path/to/dependencies
        some/other/dependencies
      key: cache-npm-${{ hashFiles('**/lockfiles') }}
      restore-keys: |
        cache-npm-
```

The restore keys can be provided as a complete name, or a prefix, read more here on how a cache key is matched using restore keys.

Separate caches by Operating System

In case of workflows with matrix running for multiple Operating Systems, the caches can be stored separately for each of them. This can be used in combination with hashfiles in case multiple caches are being generated per OS.

```
 - uses: actions/cache@v3
   with:
     path: |
       path/to/dependencies
       some/other/dependencies
     key: ${{ runner.os }}-cache
```

Creating a short lived cache

Caches scoped to the particular workflow run id or run attempt can be stored and referred by using the run id/attempt. This is an effective way to have a short lived cache.

```
    key: cache-${{ github.run_id }}-${{ github.run_attempt }}
```

On similar lines, commit sha can be used to create a very specialized and short lived cache.

```
 - uses: actions/cache@v3
   with:
     path: |
       path/to/dependencies
       some/other/dependencies
     key: cache-${{ github.sha }}
```

Using multiple factors while forming a key depending on the need

Cache key can be formed by combination of more than one metadata, evaluated info.

```
 - uses: actions/cache@v3
   with:
     path: |
       path/to/dependencies
       some/other/dependencies
     key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}
```

The GitHub Context can be used to create keys using the workflows metadata.

Restoring Cache

Understanding how to choose path

While setting paths for caching dependencies it is important to give correct path depending on the hosted runner you are using or whether the action is running in a container job. Assigning different path for save and restore will result in cache miss.
Below are GiHub hosted runner specific paths one should take care of when writing a workflow which saves/restores caches across OS.

Ubuntu Paths

Home directory (~/) = /home/runner
${{ github.workspace }} = /home/runner/work/repo/repo
process.env['RUNNER_TEMP']=/home/runner/work/_temp
process.cwd() = /home/runner/work/repo/repo

Windows Paths

Home directory (~/) = C:\Users\runneradmin
${{ github.workspace }} = D:\a\repo\repo
process.env['RUNNER_TEMP'] = D:\a_temp
process.cwd() = D:\a\repo\repo

macOS Paths

Home directory (~/) = /Users/runner
${{ github.workspace }} = /Users/runner/work/repo/repo
process.env['RUNNER_TEMP'] = /Users/runner/work/_temp
process.cwd() = /Users/runner/work/repo/repo
Where:

cwd() = Current working directory where the repository code resides.
RUNNER_TEMP = Environment variable defined for temporary storage location.

Make cache read only / Reuse cache from centralized job

In case you are using a centralized job to create and save your cache that can be reused by other jobs in your repository, this action will take care of your restore only needs and make the cache read-only.

```
steps:
  - uses: actions/checkout@v3

  - uses: actions/cache/restore@v3
    id: cache
    with:
      path: path/to/dependencies
      key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}

  - name: Install Dependencies
    if: steps.cache.outputs.cache-hit != 'true'
    run: /install.sh

  - name: Build
    run: /build.sh

  - name: Publish package to public
    run: /publish.sh
```

Failing/Exiting the workflow if cache with exact key is not found

You can use the output of this action to exit the workflow on cache miss. This way you can restrict your workflow to only initiate the build when cache-hit occurs, in other words, cache with exact key is found.

```
steps:
  - uses: actions/checkout@v3

  - uses: actions/cache/restore@v3
    id: cache
    with:
      path: path/to/dependencies
      key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}

  - name: Check cache hit
    if: steps.cache.outputs.cache-hit != 'true'
    run: exit 1

  - name: Build
    run: /build.sh
```

Saving cache

Reusing primary key from restore cache as input to save action

If you want to avoid re-computing the cache key again in save action, the outputs from restore action can be used as input to the save action.

```
  - uses: actions/cache/restore@v3
    id: restore-cache
    with:
      path: |
        path/to/dependencies
        some/other/dependencies
      key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}
  .
  .
  .
  - uses: actions/cache/save@v3
    with:
      path: |
        path/to/dependencies
        some/other/dependencies
      key: ${{ steps.restore-cache.outputs.cache-primary-key }}
```

Re-evaluate cache key while saving cache

On the other hand, the key can also be explicitly re-computed while executing the save action. This helps in cases where the lockfiles are generated during the build.

Let's say we have a restore step that computes key at runtime

```
uses: actions/cache/restore@v3
```

```
id: restore-cache
with:
   key: cache-${{ hashFiles('**/lockfiles') }}
```

Case 1: Where an user would want to reuse the key as it is

```
uses: actions/cache/save@v3
with:
   key: ${{ steps.restore-cache.outputs.cache-primary-key }}
```

Case 2: Where the user would want to re-evaluate the key

```
uses: actions/cache/save@v3
with:
   key: npm-cache-${{hashfiles(package-lock.json)}}
```

Saving cache even if the build fails

There can be cases where a cache should be saved even if the build job fails. For example, a job can fail due to flaky tests but the caches can still be re-used. You can use actions/cache/save action to save the cache by using if: always() condition.
Similarly, actions/cache/save action can be conditionally used based on the output of the previous steps. This way you get more control on when to save the cache.

```
steps:
 - uses: actions/checkout@v3
 .
 . // restore if need be
 .
 - name: Build
   run: /build.sh
 - uses: actions/cache/save@v3
   if: always() // or any other condition to invoke the save action
   with:
     path: path/to/dependencies
     key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}
```

Saving cache once and reusing in multiple workflows

In case of multi-module projects, where the built artifact of one project needs to be reused in subsequent child modules, the need of rebuilding the parent module again and again with every build can be eliminated.
The actions/cache or actions/cache/save action can be used to build and save the parent module artifact once, and restored multiple times while building the child modules.

Step 1 - Build the parent module and save it

```
steps:
 - uses: actions/checkout@v3

 - name: Build
   run: ./build-parent-module.sh

 - uses: actions/cache/save@v3
   id: cache
```

```
  with:
    path: path/to/dependencies
    key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}
```

Step 2 - Restore the built artifact from cache using the same key and path

```
steps:
  - uses: actions/checkout@v3

  - uses: actions/cache/restore@v3
    id: cache
    with:
      path: path/to/dependencies
      key: ${{ runner.os }}-${{ hashFiles('**/lockfiles') }}

  - name: Install Dependencies
    if: steps.cache.outputs.cache-hit != 'true'
    run: ./install.sh

  - name: Build
    run: ./build-child-module.sh

  - name: Publish package to public
    run: ./publish.sh
```

GitHub Advanced Security

CBIIT has invested in GitHub Advanced Security which has many features that help teams improve and maintain the quality of their code. By default, some of these features are enabled such as dependency graph and Dependabot alerts. Other security features have been enabled for organizations under the enterprise account to run on all repositories. These features are listed in table below:

	Description	Enabled for Public Repos	Enabled for Private Repos
Code Scanning	This allows to search for potential security vulnerabilities and coding errors in code – For more information on setting up code scanning capabilities please refer to the following guide: Code Scanning.	Yes	Yes
Secret Scanning	This allows detection of secrets, for example keys and tokens, that have been checked into repositories. For more information on setting up secret scanning	Yes	Yes

	capabilities please refer to the following guide: Secret Scanning.		
Dependency Review	This provides the full impact of changes to dependencies and see details of any vulnerable versions before merging a pull request. For more information on setting up dependency review for a given repository please refer to the following guide: Dependency Review.	Yes	Yes

GitHub provides starter workflows for security features such as code scanning. Teams can leverage these suggested workflows to construct code scanning workflows, instead of starting from scratch. For more information regarding setting up starter workflows, please refer to the following document: Using Starter Workflows.

Note: The development teams and/or repository owners are responsible for setting up advanced security features against their repositories in addition to continuously reviewing their scanning results along with mitigating the reported findings – These activities should be embedded within the software development lifecycle.

GitHub Webhooks

Webhooks allow teams to build or set up integrations, such as OAuth Apps, which subscribe to certain events on GitHub.com. When one of those events is triggered, GitHub sends an HTTP POST payload to the webhook's configured URL. Webhooks can be used to update an external issue tracker, trigger CI builds, send a message to a collaboration platform such as MS Teams, or even deploy to a given environment.

Webhooks can be installed against a specific repository. Once installed, the webhook will be sent each time one or more subscribed events occurs.

Note: Up to 20 webhooks can be created for each event on each installation target (e.g., specific repository).

For additional information regarding setting up GitHub Webhooks, please refer to the following document: Creating Webhooks.

GitHub Copilot

GitHub Copilot is an AI pair programmer that offers autocomplete-style suggestions as developers write code. Individual developers can receive suggestions from GitHub Copilot either by starting to write the code, or by writing a natural language comment describing what

the code is intended to do. GitHub Copilot analyzes the context in the file being edited, as well as related files, and offers suggestions from within the text editor. GitHub Copilot is powered by OpenAI Codex, a new AI system created by OpenAI.

GitHub Copilot is trained on all languages that appear in public repositories. For each language, the quality of suggestions received may depend on the volume and diversity of training data for that language. For example, JavaScript is well-represented in public repositories and is one of GitHub Copilot's best supported languages. Languages with less representation in public repositories may produce fewer or less robust suggestions.

GitHub Copilot is available as an extension in Visual Studio Code, Visual Studio, Neovim and the JetBrains suite of IDEs

For more information on how to get started with GitHub Copilot, please refer to the following document: GitHub Copilot