How to get started with GitHub Enterprise Cloud

WRITTEN BY GITHUB WITH

Table of Contents

Introduction

What to expect from GitHub Enterprise Cloud

How to adopt GitHub Enterprise Cloud

- Step 1: Consider your structure
- Step 2: Choose a user model
- Step 3: Configure your settings
- Step 4: Manage billing
- Step 5: Set up your organization
- Step 6: Set up your repositories
- Step 7: Organize your teams
- Step 8: Assign users
- Step 9: Set up security measures
- Step 10: Manage integrations
- Step 11: Migrate

Authors:
Philip Holleran, Field CTO, GitHub
Kevin Alwell, Principal Engineer, GitHub

Introduction

This ebook illustrates the deep business value of GitHub Enterprise Cloud. Throughout the pages, we'll provide a detailed roadmap for executing a stress-free implementation—and cover the many features and tools that can help your organization scale securely.

What to expect from GitHub Enterprise Cloud

GitHub is home to 100+ million software developers. It's where builders collaborate on software from idea to production. Here's what GitHub Enterprise Cloud can offer your team:

The best-in-class developer experience

Developers know and love working on GitHub. Their comfort with our platform and its tools accelerates productivity and improves overall satisfaction, which leads to talent acquisition and higher retention. What's more, our fully integrated developer toolchain prevents unnecessary context switching, keeping developers in the flow longer. In fact, the latest Forrester report says that after using GitHub Enterprise and GitHub Advanced Security for three years, composite organizations see +22% productivity gains overall.

Best in class Developer Experience

Our enterprise customers derive great value from the open source community, including through well-maintained service provider integrations and data-driven services like Dependabot and GitHub Copilot. Plus, access to reusable components built by the community greatly impacts productivity and innovation.

Largest Connected Developer Community

Our enterprise customers derive great value from the open source community, including through well-maintained service provider integrations and data-driven services like Dependabot and GitHub Copilot. Plus, access to reusable components built by the community greatly impacts productivity and innovation.

Integrated, Developer Centric Application Security

Application security is native with GitHub through code, secret, and dependency scanning. This reduces friction and improves mean time to mediation.

Stable, Innovating partner for the long term

In 2018, GitHub became a Microsoft company. Since the acquisition, GitHub has hired thousands of engineers that continue to mature core services, resulting in deeper integration, greater value, and tool consolidation. GitHub continues to differentiate through services like GitHub Advanced Security, GitHub Copilot, and Codespaces.

Tons of capabilities (with more on the way)

From GitHub Actions, which automates build, test, and deployment pipelines, to Codespaces, which allows devs to spin up an environment in mere moments, GitHub Enterprise Cloud has numerous features that increase speed, reliability, and developer happiness. We recently launched GitHub Copilot, which uses the OpenAI Condex to suggest code and entire functions in real time right from the editor. Research has already found that GitHub Copilot helps developers code 30% faster, focus on solving bigger problems, stay in the flow longer, and feel more fulfilled with their work.

Cloud-first production

As a SaaS platform, GitHub Enterprise Cloud is the most operationally efficient deployment option for organizations operating at scale on GitHub. In addition to its ability to scale to support companies with more than 50,000 developers, the platform provides resilience and disaster recovery out of the box.

How to adopt GitHub Enterprise Cloud

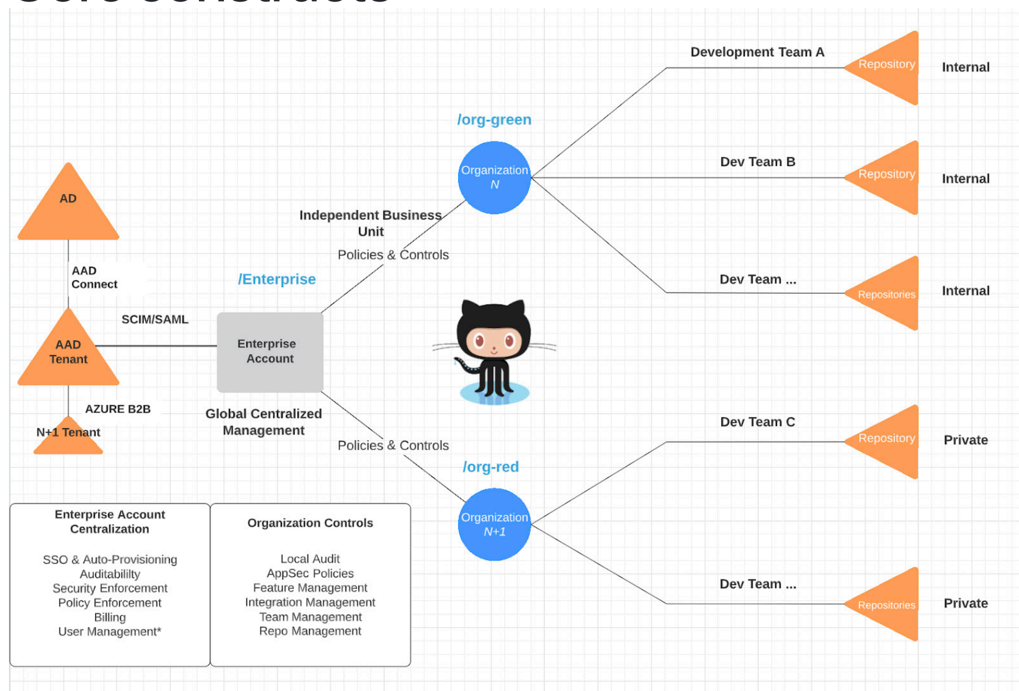
GitHub Enterprise Cloud provides a number of flexible configuration options, allowing each business to configure the platform to best meet their unique needs. While there is no single "best" way to implement GitHub Enterprise Cloud, there are some common implementation patterns and steps you should carefully consider.

Successful GitHub Enterprise Cloud implementations begin with establishing a single enterprise account for all internal work. This will govern policy across your GitHub organizations, users, and teams, and be your single portal for managing billing.

Step 1: Consider your structure

Organization structures are constantly evolving, especially in acquisitive- and/or technology- driven environments. Your GitHub architecture should reflect your intention to promote a collaborative developer experience that respects the complex and dynamic nature of your business. Below, we'll dive into the core constructs of a GitHub Enterprise Cloud account.

Core constructs



Enterprise account

The enterprise account is the highest level, logically isolated construct within GitHub Enterprise Cloud. It is what administrators use to manage how GitHub interfaces with internal business systems like your identity provider (IdP) and log management system. The enterprise account also provides the ability to define and enforce policies governing the use of GitHub resources and capabilities. If you're using GitHub Advanced Security, the enterprise account also receives AppSec insight derived from organizations operating within its confines.

Organization

Enterprises consist of one or more organizations to which users are added. Organizations are the "owners" of shared repositories, discussions, and projects. They let administrators set more granular policies on repositories belonging to the organization and user behavior within the organization. Policies not enforced at the enterprise level are distributed out to individual organizations.

Organizations also serve as a roll-up mechanism for reporting. Consumption-based services, such as GitHub Actions and Codespaces, are reported at both the repository and organization level. Spending limits on these services can be set on a per-organization basis. GitHub Advanced Security provides a summary of the security status of repositories in an organization.

Repository

Repositories are your application source code and the primary construct developers interface with on GitHub. In addition to source code, repositories are the way users access application security metrics, CI/CD workflows, and other day-to-day activities for developers.

There are three visibility types for repositories: public, internal, and private.

- Public repositories are visible to the world and are primarily used for open source content.
- Internal repositories are public to members of your enterprise account.
- Private repositories are only visible to admins and individuals or teams who have been given explicit access.

Team

GitHub teams group users of common projects or specialized skills. They are often the mechanism for providing role-based access to collections of repositories.

User

GitHub users are mapped to individual people. They can be added to organizations as members and repository teams as collaborators. Two user models exist: Standard user accounts and Enterprise Managed Users (EMUs).

Step 2: Choose a user model

When creating your enterprise account, you'll need to decide which user model (detailed below) works best for your company.

Ultimately, if you need full control of your developers' accounts and a firm separation between the open source community and your company's code, EMUs are for you. However, if your developers will be regularly building and maintaining open source code as a part of their daily work, the standard user model may be best.

Standard users

Standard users on GitHub are owned and operated by individuals, and are designed to follow that individual as they join, contribute, and leave organizations throughout their life. These users can actively participate in the OSS community.

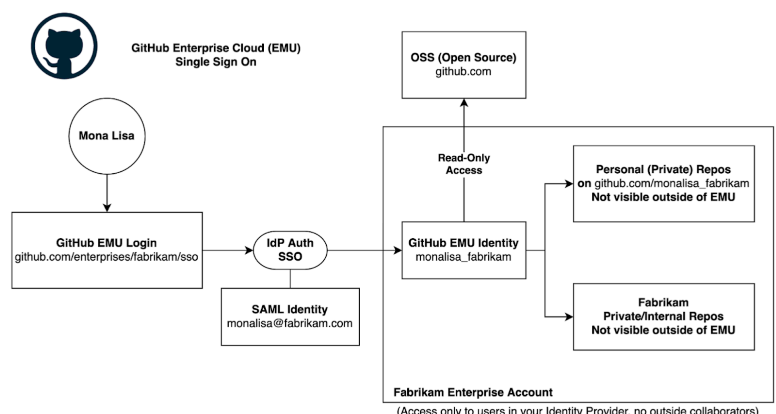
Standard GitHub users can also be invited to join organizations as contributors gated by SAML 20.0 SSO. SCIM governs the workplace identity association and authorizes or terminates access automatically.

EMUs

Enterprise Managed Users (EMUs) are company-owned user accounts on GitHub that operate exclusively within the confines of an enterprise account on proprietary source code. The lifecycle of these accounts is managed by your identity provider (IdP):

Azure Active Directory, Okta, or Ping Federate. You can read more about EMUs here.

In the workflow illustrated above, you can see the user Mona Lisa logging in through Enterprise SSO for an active user session. That GitHub EMU identity has read-only access to GitHub.com (OSS) and is subject to RBAC within your Enterprise account.



Step 3: Configure your settings

Once your enterprise is established it's time to configure settings and policies that apply to all development work. Configurations and policies not applied at the enterprise level are managed in each organization. This lets you manage your desired balance between centralized and distributed administration.

Authentication

The first setting to configure is authentication. A few minor differences exist between EMUs and standard users, so we'll cover each of them in its own section

Enterprise Managed Users

When using EMUs, a single Identity Provider (IdP) is configured at the enterprise level. Supported IdPs include Azure Active Directory (AAD) and Okta. If you need to support users from more than one IdP, you will need to configure either AAD or Okta to federate your various identity providers and expose a single store with which GitHub Enterprise Cloud will integrate.

EMUs and outside collaborators

Every user account in an EMU enterprise must be mapped to an identity in your IdP. Collaborators, consultants, and other outside parties will also require an identity in your IdP. However, these identities don't have to exist in your identity store (LDAP, AD). Azure AD includes the concept of a guest user and all supported IdPs provide support for multiple user stores through federation.

EMUs and conditional access policies

If you use Azure AD, you can configure GitHub to use OpenID Connect (OIDC) to manage authentication with your IdP. Doing so adds support for Azure AD's conditional access policies.

Standard users

When using standard users, you have the option to either configure a single SAML 2.0+ provider at the enterprise level or configure each organization with a SAML 2.0+ provider. Most customers using standard users opt for a single, enterprise-level IDP integration. However, if your business has several disparate identity systems, the organization-level configuration may work best. This multi-IdP approach is most commonly used in cases where a business operates as a set of very-loosely held portfolio companies, each with independent IT departments, tools, and processes.

Audit log

GitHub audits activity in your enterprise, including:

- Events triggered by activities in your enterprise, like configuration changes and CRUD events
- API access to resources, including those initiated by integrations
- Git events, such as pushing, pulling, or cloning of code

GitHub maintains most audited events in logs for a configurable period of up to six months. Git events are retained for seven days. Data from the logs is available to administrators within the GitHub UI, via the API, and via streaming.

You should configure audit log streaming within GitHub to ensure all data captured is held within your preferred log management system and retained in accordance with existing policies. GitHub currently supports native integration with Azure Events Hubs, Datadog, and Splunk, and has the ability to directly write to AWS S3, Azure Blob Storage, and Google Cloud Storage.

Some customers use this data for anomaly detection via tools like Defender for DevOps and the GitHub App for Splunk.

Policies

The policies pane in your GitHub Enterprise account enables the definition and enforcement of a variety of policies across your enterprise that are broken down by topic. Companies using EMUs should note that, unless explicitly stated, these policies apply over organizations but do not apply to activity in personal namespaces.

Base permissions represent the default set of permissions (if any) granted to all members of the enterprise. In most cases this will either be “no permission” or “no policy” in order to allow for limited access to particularly sensitive repositories. A number of other permissions are detailed in the documentation.

Below we’ll dive into the different topics.

Repository creation and management

Repository creation policies cover whether a user can create a repository in the enterprise and, if so, what its visibility can be. Repository creators are automatically granted administrative permissions. Each option comes with its own set of considerations:

- If a “members can create repositories” policy is set at the enterprise level, users are able to create repositories in organizations in the enterprise.
- More granular control is available by selecting “No policy, ” which allows each organization to set its own policy.

- Selecting “disabled” will prevent your users from creating repositories in the enterprise. Companies opting for this approach typically use an automated process to create repositories from standard configurations with the necessary associations to meet audit requirements. Some large customers prefer retroactive configurations applied post-create to ensure configuration standards.

Companies using EMUs may want to choose to enable “Block the creation of user namespace repositories” because enterprise policies do not currently apply to repositories in personal namespaces. Doing so is not advised, however, if your company makes heavy use of forking or chooses to leverage those personal spaces as a “sandbox” for developers.

GitHub Copilot

Policies in this section govern use of GitHub Copilot, GitHub’s AI pair programmer. GitHub Copilot is licensed separately from GitHub Enterprise Cloud. If your enterprise is using this tool, you can use the settings here to manage which organizations’ users have access, and allow or prevent suggestions from GitHub Copilot that match public code.

GitHub Actions

Actions can be enabled on all, some, or none of the organizations within the enterprise. If Actions are enabled, administrators have the ability to define which specific Actions are available to their workflows. Three options are available:

- Allow all Actions and reusable workflows: This will let users run workflows containing any GitHub Action from the marketplace or defined in any public repository. This is not appropriate for most companies, since it is very permissive. No review or approval is required to make use of any public Action.
- Allow enterprise Actions and reusable workflows: This setting is highly restrictive compared to the above setting, but it is still not appropriate for most companies because it requires all Actions to exist within your enterprise. That means developers won’t be able to use Actions directly from the GitHub Marketplace at all. You’ll also need to define a process to clone all desired Actions into your enterprise, train your teams to reference those internal Actions, and regularly check for and incorporate updates.
- Allow enterprise, select non-enterprise Actions, and reusable workflows: This is the sensible, manageable choice for most enterprises. Selecting this option presents an allow list that administrators can use to individually authorize GitHub-created Actions, Actions verified creators (authorship is verified by GitHub), and specific third-party Actions and reusable workflows.

When implementing the latter option, enterprises will need to create and maintain their own process for requesting access to GitHub Actions. Administrators will have the option to reference approved Actions in four ways:

- The simple owner/repo path to the Action's source code
- A specific branch of the Action, referenced as owner/repo@branch
- A specific tag/release of the Action, referenced as owner/repo@tag
- A specific commit SHA of the Action, referenced as owner/repo@SHA

Companies performing their own security reviews of third-party Actions should use the owner/repo@SHA syntax specific to the version of the Action approved, as commit SHAs are immutable, while code referenced by branch names and tags can change.

If your organization makes use of fork-and-pull workflows, you can define how Actions should respond to workflows initiated by pull requests into the parent repository. These settings are especially important if you want to maintain open source repositories in your enterprise or if you make use of outside collaborators.

Projects

Policies in this section govern use of GitHub Projects, GitHub's developer-centric,built-in project management tool. Administrators can use this section to optionally disable use of projects in one or more organizations and govern visibility changes of projects.

Organizations

This section allows administrators to enable or disable dependency insights, which include aggregated information about the use of OSS dependencies, their licenses, and any associated security vulnerabilities. This should remain enabled as part of a comprehensive strategy to understand OSS consumption, mitigate security concerns, and maintain license compliance.

Code security and analysis

This section includes policies governing use of GitHub's security features. Administrators can decide whether repository administrators should be allowed to enable or disable Dependabot alerts. Most customers should set this to "not allowed," which will keep Dependabot enabled across all their code.

Users of GitHub Advanced Security (our security upgrade) will find additional, similar policy controls in this section. If you are using GitHub Advanced Security, you'll likely want to keep all of these set to "not allowed," which will prevent individual repository owners from circumventing enterprise-wide security settings.

Supplemental controls

While the policies section of the enterprise account covers a wide variety of needs, your specific requirements may necessitate the enforcement of additional policy controls. The GitHub community has built a number of GitHub applications that extend our core platform to define and enforce unique policies. A few of the most common are listed below:

- Safe-settings: Safe-settings is an event-driven GitHub application that prevents configuration drift within your organization.
- Policy-bot: This application allows for the definition and enforcement of robust policies governing code review and approval of pull requests.
- GHAS compliance: GHAS compliance is a policy-as-code, event-driven GitHub Action that enables administrators to configure their risk threshold for security issues that are reported by GitHub code scanning, secret scanning, and Dependabot security.
- Jira: Easily connect one or more GitHub organizations to your Jira site and select specific repositories to bring your work together. Get updates and links about what's happening with your repositories in Jira issues for things like pull requests, commits, and branches. You can also use specific commands in your commit message to perform actions within Jira, like closing an issue, adding a comment, updating time tracking, and transitioning workflow state.

Step 4: Manage billing

The enterprise account is the central point for all billing within GitHub Enterprise Cloud. This includes all organizations that are part of your enterprise.

User licenses

GitHub Enterprise Cloud licenses are provisioned to your enterprise upon purchase. Users provisioned in your enterprise consume a license and can be added to any of your organizations. If you are using GitHub Advanced Security, those licenses can be purchased and managed the same way.

Consumption-based services

Use of services such as GitHub-hosted Actions runners, packages, and codespaces are billed per unit consumed, and consumption across all of your enterprise's organizations will be aggregated for invoicing. Customers who purchase GitHub Enterprise with a credit card are billed directly on a monthly basis.

Though charges are aggregated into a single invoice, a detailed, per-organization report of all consumption services is available. Plus, an enterprise-wide spending limit for each service can be set in your enterprise settings. In the near term, we plan to enable organization-level spending limits for more fine-grained control.

If you purchased GitHub Enterprise Cloud through a Microsoft Enterprise Agreement you can connect an Azure Subscription ID to your enterprise account and have all charges managed via your Azure invoicing. If you purchased GitHub directly on invoice, all consumption charges will be billed via invoice.

Some companies may need to implement an internal chargeback model for resources

consumed. During migration, we recommend building a service catalog that maps repositories to projects and teams. This catalog could be kept in sync if repository creation is externalized. Otherwise, repositories should have their cost center ID as a repository topic. If there is no topic, that repository cannot request to have consumption services enabled.

Insight into consumption services billing, including GitHub Advanced Security is provided at the organization and enterprise levels. In some cases, reports should be programmatically parsed for mapping aggregate consumption back to project teams.

GitHub Advanced Security

GitHub Advanced Security is licensed per seat for active committers to repositories using the features. Each license for GitHub Advanced Security specifies a maximum number of accounts, or seats, that can use these features. A committer is considered active if one of their commits has been pushed to the repository within the last 90 days, regardless of when it was originally authored.

Compliance reporting

GitHub provides our latest SOC reports, third-party attestations, and self-reported business operations procedures as self-serve documents within the enterprise administration page.

Support

Enterprise administrators can open and manage their GitHub support tickets through the Enterprise landing page.

Step 5: Set up your organization

Standard Organizations are shared accounts where your users collaborate across many projects at once, with sophisticated security and administrative features. Your enterprise will consist of one or more organizations.

The structure you apply to your organizations can greatly facilitate collaboration and discovery while reducing administrative burden, or it can create unnecessary silos and add administrative overhead. As such, it is important to give careful consideration to organization structure ahead of time.

Organization structure

Don't overly index your GitHub organizations to your current corporate structure.

When setting up your GitHub Enterprise instance, your immediate "least privilege" instinct may be to create an organization for every project or department at your company. This might seem like a good way to manage permissions and reduce

noise, but it's not the ideal strategy and should be avoided, as it doesn't allow for fluid organizational structures. Your GitHub configuration should gracefully handle corporate reorganizations without having to rename organizations, and handle all the resulting downstream work, such as re-pathing integrations and updating external links.

Such a structure also increases the team management burden and siloed conversations. Teams (discussed later in this document) are currently an organizational construct. This means a team defined in one organization cannot be @mentioned in another organization. If a team needs access to more than one org, the team must be created in each organization and mapped to the appropriate IdP group.

Finally, having an organization for every project increases the burden of managing integrations. GitHub Apps are installed in organizations, so enterprises with many organizations must install and manage each of their integrations in each of their organizations. Also, note that there is a limit of 100 GitHub App installations per organization.

We want you to get started with an architecture that helps your team work together seamlessly, creatively, and transparently without bogging you down in unnecessary overhead. Instead of creating many organizations and siloing users, we suggest using one or few organizations for shared ownership of repositories and making use of teams to segment users within those organizations.

Below we'll dive into three common models successfully adopted by GitHub Enterprise Cloud customers that effectively utilize as few organizations as prudent for their business.

Model 1: Single organization

In this model, a single organization is used for all or the vast majority of repositories. Many small- to medium-sized organizations (less than 5,000 developers) that operate as a single company effectively use this approach to manage their GitHub environment

Most users of this model set their organization's base permissions to "none." This requires all users to be explicitly added to all repositories in order to view them and/or propose changes. By itself, this approach creates silos, greatly limits visibility, and actively prevents Innersource. To counter this tendency we recommend customers who choose this model use teams to ensure visibility. For example, an "all members" team can be created and added to repositories open to collaboration. Some have gone as far as to have this type of team automatically added to all repositories by default, and establishing an exception process for when repositories must truly be kept on a need-to-know basis.

Exceptions

Slight variations on this model exist to handle extenuating circumstances. Some customers using this model maintain one or two "top secret" organizations for projects

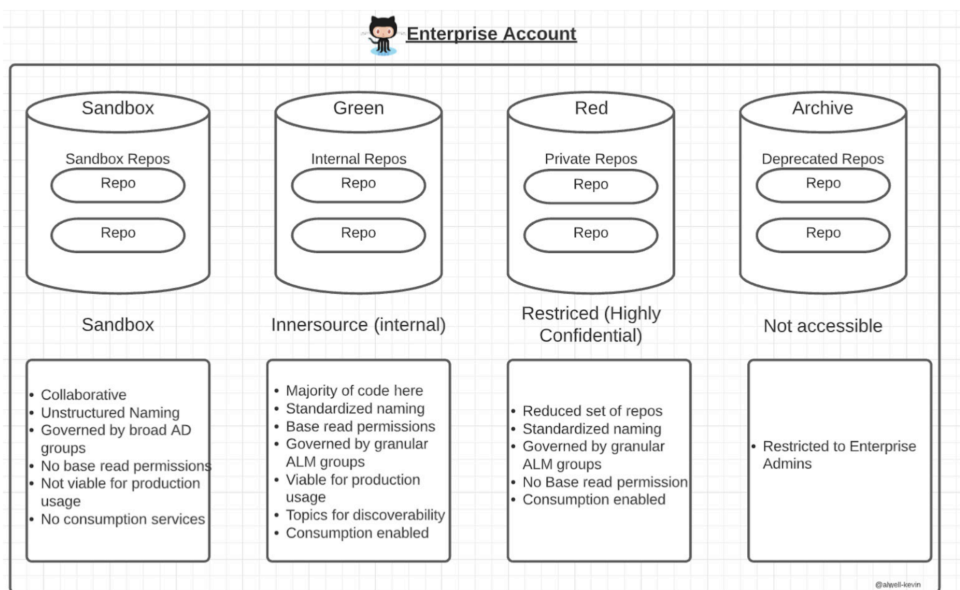
that must be kept completely separate from all other work, like projects for highly sensitive customers.

Another variation on this model uses a separate organization to segment repositories managed by teams in particular locations. Customers taking this approach tend to have concerns around intellectual property laws.

For organizations with 10s of thousands of developers, it may be practical to limit the number of developers in an organization to approximately 10,000.

Model 2: Red-green-sandbox-archive

The red-green-sandbox-archive model utilizes two primary organizations and a sandbox. We'll explain how each piece works below.



Green organization

The green organization serves as the primary collaboration space for developers operating within your enterprise account. About 90% of all repositories will reside here. Setting the base permissions to "write" promotes Innersource and empowers users to propose changes to repositories. Effective use of branch protections (discussed later in this document) on repositories in the organization ensures users can only propose changes, and any change proposed will need to go through the defined code review and approval process to be accepted.

Higher-level permissions to repositories are granted through teams. Each team can use their team's repository page to view the repositories to which they have been granted additional permissions. With most activity happening in the green organization,

the burden of managing teams and integrations is minimized.

Red organization

The red organization is for repositories that must be kept on a need-to-know basis. The default permissions of the organization are set to "none," which requires teams to

be explicitly added to the repositories to read them or propose changes.

A defined process should exist for creating repositories and teams in this organization. Doing so prevents unnecessary, self-imposed constraints.

Optionally, two more organizations may be added to this model:

Sandbox organization

A sandbox organization is a place in which any user can create a repository. This allows developers to experiment in a place that's more visible and more collaborative than personal repositories. Such an organization is especially important if you configure GitHub Enterprise Cloud to prevent developers from creating personal repositories.

A process for moving work from the sandbox into the green or red organizations should be defined to ensure experiments can transition into more formal management.

Archive organization

Then you'll want an archive organization to contain repositories that are no longer actively maintained. Any repository can be archived, making it read-only for all users. While these repositories can remain in their existing organizations, some companies prefer to transfer ownership of these archived repositories to a separate organization, leaving their other repositories for active work. Note that choosing to do this changes the owner/name syntax of the repos. This may make finding the repository in GitHub a bit more challenging if the user expects it to be in the original organization.

Model 3: Portfolio company

There are a few scenarios in which the red-green model may not adequately match your company's internal structures.

For example, very large companies (10s of thousands of employees) that are divided into relatively static business units, like a global bank consisting of retail banking, personal investments, capital markets, and insurance divisions.

Also in this camp are portfolio companies with operating units that function independently of one another, like a media company consisting of broadcast networks, production studios, streaming services, and interactive games companies. These companies may also experience regular mergers, acquisitions, and divestitures.

For a very large company we recommend mapping your GitHub Enterprise Cloud organization structure to your highest-level corporate division. These will generally be divisions one level below the CEO and be static in nature. (Reorganizations typically

happen within these divisions, leaving the top level structure alone.)

In a portfolio company, organizations should be mapped to each business entity. Organizations can be transferred from one enterprise to another, easing the burden of managing a merger or divestiture.

If possible, we recommend keeping a single organization per portfolio company or major business division. This is essentially multiple implementations of the single organization strategy described above. At most, we recommend implementing a minimal red-green structure for each major division. If you go down this path, consider implementing a single sandbox across all units and keeping archive repositories in the primary organization. This will help minimize the number of active organizations to manage.

A note on open source projects: Open source projects should be kept in a dedicated organization, separate from the corporate org(s). Otherwise you can run into security boundary issues, particularly when you use outside collaborators. If you are using an Enterprise Managed User environment, this may require having separate enterprise accounts given the constraints.

Step 6: Set up your repositories

TRepositories are the core construct within which developers build software. Controls are available to manage work associated with repositories, including issue tracking, managing proposed changes, and code reviews.

Use teams to grant permissions

Permissions can be granted to teams and individuals. When possible, enterprises should grant permissions solely to teams, whose membership can be synced with IdP groups. Automation can be created, using webhooks and the GitHub API, to revert any individual user permissions granted. GitHub's open source safe-settings app implements a form of this, enforcing permissions defined in a configuration file.

Base permissions

The earlier section on organizational models advocated for defaulting repository permissions to "write" when possible. That being said, GitHub provides three different types of default repository permissions:

- Read allows all organization members to see the repository, clone it, and participate in issues. If forking is allowed in your enterprise, users can fork the repository, too.
- Write allows all organization members the ability to push code to the repository. It does not necessarily let members push to the default branch or approve their own changes. Properly configured branch protections, discussed below, control how members can modify the code.

- Admin allows all organization members to administer the repository. This should not be used as a default given its overly permissive nature.

Repository roles

Base permission levels for repositories are fairly coarse-grained. GitHub provides an expanded set of repository roles, which can be explicitly granted to individuals or teams, including Read, Triage, Write, Maintain, and Admin.

When planning your GitHub Enterprise Cloud implementation you may find that you need even more granular permissions than those applied to the default set of repository roles. If that happens, you can create custom roles for your organization and apply more fine-grained permissions.

Branch protections

Permissions to Git repositories are, by design, coarse. Without a mitigating control in place, a user with “write” permissions to a repository can push commits to any branch, including default. To prevent this, teams should establish branch protection rules for their repositories.

There are a variety of protections available to help teams map their desired processes to their branching strategy, and include:

- Requiring a pull request and code review for all commits merged. Most teams will enable this on their default / `main` branch that they merge features into.
- Requiring automated review(s) to pass. External processes such as GItHub Actions (CI/CD) and Advanced Security (code scanning) can create status checks on pull requests. You can define which checks must successfully pass before a pull request can be merged.
- Requiring pull request approvals from different teams, defined in a CODEOWNERS file, based on what file(s) were changed.
- Restricting the user(s) who can push to the branch. This can help teams prevent people from pushing to branches managed by machine accounts as part of a branch-based release workflow. It can also be used to limit which users are allowed to merge a pull request, if mandated by policy.
- Requiring signed commits. This feature requires users’ commits to the repository by cryptographically signing, increasing confidence in their authorship.

Customers may want to require a base level of branch permissions on each repository in an organization. This is possible through the use of the open source safe-settings app (created by GitHub), or by using GitHub’s webhooks and APIs. Safe-settings will let you define repository settings, such as branch protections, as code in a single repository. The app can apply those policies to all existing and newly created repositories.

Those looking to enforce complex pull request approval workflows beyond that offered

by required reviews and CODEOWNERS should consider implementing the open source palantir/policy-bot application.

Topics

Repository topics are a useful mechanism for grouping like-repositories and promoting discovery. Use topics to tag repositories that are:

- Components of a larger system or process, like payment processing, mobile access, or data modeling.
- Written in the same language and/or framework, like Spring Boot, React, or Rust.
- Maintained by the same team.

Note: Topics are mutable by design.

Step 7: Organize your teams

Teams to manage permissions

Earlier in this document we advocated for using teams, instead of organizations, to manage repository permissions and visibility, facilitate conversation, and reduce management overhead. Teams used for permission management should be synced with an IdP group. This allows existing IdP processes and audit controls to be relied upon for managing access to code. Onboarding, offboarding, and access changes are all managed by the IdP.

Teams to facilitate communication

Permissions aren't the only use for teams. GitHub teams can be used to engage project teams (not explicitly defined in your IdP) and/or topic-based teams in conversations:

> <Graphic of a PR conversation>"Hey @octocorp/react, I'm having trouble figuring out why this component isn't updating along with the others. Could someone more familiar with this component give it a :eyes:? Thanks.

They can also be automatically assigned as reviewers to code using CODEOWNERS.

Example of a CODEOWNERS file

```
# This is a comment.
# Each line is a file pattern followed by one or more owners.

# These owners will be the default owners for everything in
# the repo. Unless a later match takes precedence,
# @global-owner1 and @global-owner2 will be requested for
# review when someone opens a pull request.
*       @global-owner1 @global-owner2

# Order is important; the last matching pattern takes the most
# precedence. When someone opens a pull request that only
# modifies JS files, only @js-owner and not the global
# owner(s) will be requested for a review.
*.js    @js-owner #This is an inline comment.

# You can also use email addresses if you prefer. They'll be
# used to look up users just like we do for commit author
# emails.
*.go docs@example.com

# Teams can be specified as code owners as well. Teams should
# be identified in the format @org/team-name. Teams must have
# explicit write access to the repository. In this example,
```

If your company does not have a simple, quick way to create and manage these groups in your IdP, organization members should be able to create and manage their own ad-hoc teams.

Team pages

Each team has its own page within an organization, which takes the form https://github.com/orgs/<orgName>/teams/<teamName>. Alternatively, you can navigate to a team's page from your organization's main page (https://github.com/orgs/<orgName>), click on teams, then click on the desired team.

Team repositories

Customers commonly want to easily see which teams maintain which projects, and members within teams appreciate a canonical list of the code their team maintains. To meet both of these needs, you can find a list of all repositories to which a team has been granted explicit permissions (beyond the organization's default) on each team's page in GitHub.

The team repositories page can be accessed by clicking on "Repositories" in the top bar, or directly accessing it at github.com/orgs/<myOrg>/teams/<myTeam>/repositories.

Step 8: Assign users

Additional considerations for Enterprise Managed Users

The earlier section on enterprise account configuration detailed the differences between standard users and EMUs and how to choose between the two. If you select Enterprise Managed Users, there are a few additional configuration options.

Personal namespaces

By default, users can create repositories in their namespace. The user who creates the repository is, by default, an administrator and can invite other users to collaborate on the repository. Team-based permissions are not available, as teams are owned by organizations, and personal namespaces are outside of any organization.

Some customers prefer to allow individual developers the freedom to experiment in their personal namespaces. Those who do should have a defined process for promoting experiments to organization-owned repositories.

Other customers prefer to give developers the ability to create experimental repositories within an organization, often called a sandbox, which we covered above. Keeping such repositories in an organization lets them apply organization-wide policies to those experiments and use teams to manage access. This behavior can be enforced by blocking the creation of user namespace repositories.

Restricted users for partners or consultants

Managing access for partners and consultants presents a unique challenge. Such users often need access to several repositories and to participate in team conversations. However, adding them as full members of the organization would grant them the default access given to all other organization members, which may be too broad.

For this reason, GitHub provides a restricted user type. Restricted users in an EMU environment operate as standard users, but they don't have access to internal repositories.

Step 9: Set up security measures

Establishing controls for authentication and authorization

Push protections

When you enable push protection, secret scanning proactively checks pushes for high-confidence secrets (those identified with a low false-positive rate). Secret scanning lists any secrets it detects so the author can review the secrets and remove them or, if needed, allow those secrets to be pushed.

In addition to push protections for secret scanning, GitHub has developed additional controls to limit pushes that contain invalid extensions (.mov, etc), non-compliant commit messaging, and file path length limits.

Code in or exfiltration

Some organizations apply broad network controls to limit developer access to GitHub.com in order to address the risk of data exfiltration. However, GitHub UI traffic can be managed through an enterprise session decorator (alpha). This indicates whether someone is accessing your code via an active EMU session and inherently prevents write access to public GitHub.com. If the decorator exists, traffic to GitHub is allowed because constraints are automatically applied. If the decorator does not exist, traffic is further evaluated. This doesn't satisfy all aspects of malicious exfiltration but it does provide another layer of control to prevent inadvertent publishing to public GitHub. Consider monitoring traffic to GitHub at the proxy or network gateway for this decorator.

Similar to Stack Overflow, Medium or miscellaneous technical posts, with unfettered access to GitHub.com, developers may have access to code with malicious intent. While access to open source code has a measurably positive impact on the quality and velocity of software delivery, organizations should manage its consumption thoughtfully.

There are layers of controls within most organizations that limit the blast radius and efficacy of that code including:

Segmentation within sensitive environments

Developers may execute malicious code, but if that code doesn't have access to sensitive data or environments, its blast radius is reduced. Typically, there are processes in place for managing code escalation between environments that should catch vulnerabilities prior to exploitation.

Automated SAST and SCA scans prior to deployment

In the code promotion process—from development through QA and into production—automated code scans should be enforced to provide feedback to the developer and their team as to whether the code being introduced is secure. GitHub Advanced Security (an upgrade) has features for this purpose, all of which are deeply integrated within the developer workflow for in-context remediation.

Private registry enforcement

Packages pulled from GitHub and promoted through initial stages of development are typically unavailable to build agents who only have access to organization-approved private registries, excluding public registries. Packages brought into an application that are unlisted on the application manifest are subject to standard SAST scans.

Network ingress and egress controls

Code that attempts to, for example, reach out to a malicious URI would be prevented at the network level through standard network egress controls. All inbound traffic from GitHub comes from known, cryptographically verified locations.

Personal Access Tokens (PAT) and SSH keys

Classic

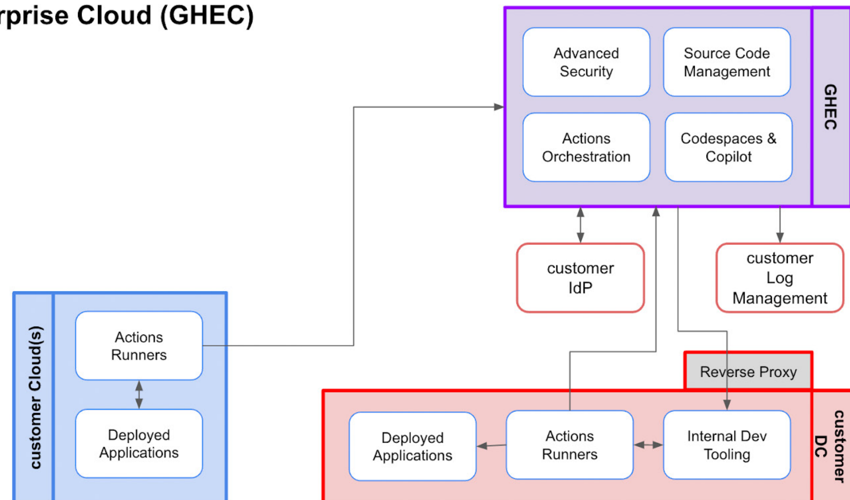
Personal access tokens are an alternative to using passwords for authentication to GitHub when using the GitHub API or the command line. They're intended to access GitHub resources on behalf of yourself. To access resources on behalf of an organization, or for long-lived integrations, you should use a GitHub app.

V2

Modern personal access tokens provide more fine-grained access to organization resources. In contrast to most classic tokens, they also have finite lifespans that are defined during credential creation.

MFA

For supported IdPs, MFA is typically handled within that environment. However, GitHub also provides native MFA for non-EMU organizations.



Step 10: Manage integrations

Companies are dutifully hesitant to create a porous network boundary for their self-hosted resources. As such, integrating a SaaS service like the GitHub platform requires a thoughtful, layered approach. The following integration architecture and design patterns ensure secure connectivity between GitHub Enterprise Cloud and your self-hosted resources.

Common integration architecture Webhooks

Webhooks allow integrations to subscribe to events on GitHub and receive notifications when those events occur. Integrations create webhooks, select the events they want for notification, provide a publicly routable HTTPS endpoint where GitHub can deliver a JSON payload, and optionally provide a secret value that GitHub can use to cryptographically sign the payload.

Webhook payloads are JSON delivered in an HTTPS POST.

Delivering webhooks to on premises resources

As part of your deployment of GitHub Enterprise Cloud you may need to have webhooks send notifications to applications hosted behind your firewall. The two most common needs are sending notifications to locally deployed CI/CD solutions, like Jenkins, and to locally deployed project management tooling, like Jira Server.

With proper network configuration, GitHub Enterprise Cloud can notify these systems by implementing a form of reverse proxy, which includes:

- A publicly routable FQDN to which GitHub can send the HTTPS post
- URL rewrite and port forwarding to the appropriate application

This pattern can be implemented using off-the shelf components from vendors like ngrok, configuration of existing web servers like NGINX, or a commercial product such as a WAF or API gateway.

Securing webhook deliveries to onpremises resources

When implementing a reverse proxy to handle webhook delivery, GitHub recommends the following practices:

- Only allow inbound HTTPS traffic on port 443 to your FQDN
- Only allow traffic from the IP ranges advertised in the hooks section of https://api.github.com/meta
- Terminate SSL at your FQDN and inspect the JSON payload for the presence of well-formatted JSON
- Ensure the solution(s) you are integrating with support and use webhook signatures

Copilot endpoints

The Copilot IDE integration(s) require connectivity back to GitHub.com for authentication and prompt sending/receiving. The following endpoints should be reachable by developers' machines, directly or through the corporate proxy:

https://github.com/login/device/code

https://github.com/login/oauth/access_token

https://api.github.com/user

https://api.github.com/copilot_internal/v2/token

https://api.github.com/copilot_internal/notification

https://default.exp-tas.com

https://copilot-telemetry.githubusercontent.com/telemetry

https://copilot-proxy.githubusercontent.com

API access

IP allow lists

Integrations with GitHub may need to call GitHub's APIs in order to properly function. If you have configured an IP allow list for your organization, the IP ranges of any integrations installed must be included. When setting up the IP allow list, you can optionally allow installed integrations to update it. If the author of the integration provides an allow list for their application, it will automatically be added to your allow list. If not, the allow list will need to be managed manually.

IP allow lists and Conditional Access Policies for EMUs

If you are using Enterprise Managed Users with OIDC SSO, GitHub will automatically use your IdP's conditional access policy (CAP) IP conditions to validate user interactions with GitHub. You will also need to ensure the IP ranges of all applications and integrations are added to your IdP's CAP.

IP allow lists, CAP, and Actions runner access

GitHub provides hosted Actions runners that you can use to perform your builds, deployments, static analysis scans, and any other desired automations. If you are using GitHub-hosted runners with four or more vCPUs, you can receive a static IP address for your runners. This IP address is unique to your enterprise and can be added to your IP allow list or IdP CAP to grant the runners access to your code.

Polling

If you are writing an integration or automation against the GitHub API, you should avoid API polling. To ensure the performance and reliability of our system for all, GitHub enforces a hard rate limit on API activity for individuals and a higher limit for applications. Regular polling of our API, especially in larger organizations, will hit the rate limit.

Rather than polling our API on a scheduled basis, you should instead use webhooks to be notified of events on GitHub. The integration will receive any webhook events to which it is subscribed, and can then make an authenticated request to GitHub should any action need to be taken.

Step 11: Migrate

Changing to a new system can seem like a daunting task. Luckily, GitHub provides a number of helpful tools to ease the technical aspects of migration, and services to help your organization establish a plan for successful migration with minimal impact to your teams.

Migrating step by step

Version control, CI/CD, and security tools and processes can get quite complex when you're migrating. Given the complexity, most companies focus first on migrating their source code to GitHub Enterprise Cloud, while supporting existing tooling such as planning, tracking, and CI/CD. Moving all code from disparate version control systems into GitHub lets developers quickly start realizing GitHub's benefits.

This migration is usually performed team by team or business unit by business unit. Once a team's code is in GitHub, and integrated with existing tooling and processes, those teams can begin migrating their security scanning process to GitHub Advanced Security's code scanning and CI/CD jobs to GitHub Actions.

Consider enabling GitHub Copilot early in the code migration process to deliver a quick and incredibly impactful win for your organization. GitHub Copilot not only makes developers more productive, but it measurably improves their satisfaction and reduces the friction of adopting new tools like GitHub Actions by creating templates automatically.

Migration tooling

GitHub provides tooling to help you easily migrate your repositories and CI/CD pipelines.

The GitHub Enterprise Importer (GEI) is a highly customizable API-first migration offering designed to help you move your enterprise to GitHub Enterprise Cloud. The GEI-CLI wraps the GEI APIs as a cross-platform console application to simplify customizing your migration experience. Most customers who use the CLI, abstract it for their developer communities by building a dashboard on top of the CLI. That abstraction can contain a series of validations for the migration, including checks on repo/file size and LFS usage. Contributions can be mapped back to original committers during the migration process.

The GitHub Actions Importer helps facilitate the migration of Azure DevOps, CircleCI, GitLab CI, Jenkins, and Travis CI pipelines to GitHub Actions.

Want to learn more? Visit github.com/enterprise or chat with a member of our sales team.