Technical Memo

ECMWF
European Centre for Medium-Range Weather Forecasts

Memo No 914

Impact of using GitHub Enterprise for Open Development

Tiago Quintino, Iain Russell,
Daniel Tipping, Dusan Figala

October 2023

Series: ECMWF Technical Memoranda

A full list of ECMWF Publications can be found on our web site under:

http://www.ecmwf.int/publications/

Contact: library@ecmwf.int

European Centre for Medium Range Weather Forecasts
Shinfield Park, Reading, Berkshire RG2 9AX, England

Contents

Executive Summary

ECMWF is looking for ways to work better with its Member States and the worldwide open-source community. ECMWF is also searching for ways to improve the automatic tools for its own software development process. The goals are to encourage collaboration, make development work more efficient, and increase robustness by reducing errors in software releases.

This report explains how ECMWF could use GitHub Enterprise to make it easier for people to collaborate on software projects, especially with partners external to ECMWF. It also explores how to use GitHub specific workflows (based on Github Actions) to build Continuous Integration and Deployment (CI/CD) even more automated at ECMWF. We show that ECMWF can safely use GitHub Enterprise to host its software online and that it can be linked to ECMWF's existing sign-in authentication system.

We demonstrate that using CI/CD pipelines integrated with GitHub are a big improvement over Bamboo, the current tool used for automating software tasks. GitHub Actions can handle ECMWF's complicated testing needs, like testing software on different systems and settings all at once. It's also faster, so tests can be run more often, reducing the chance of releasing software with errors. With GitHub, we demonstrated that ECMWF developers can achieve workflows that are hard or impossible with Bamboo. With the new CI/CD pipelines, we can also automatically distribute finished software to different online software repositories, making it easier to reach the wider community, and Member States in particular.

Importantly, we demonstrate that ECMWF's own developers can easily understand and change the CI/CD pipelines that were prototyped during this exploratory project. Therefore, we estimate that adoption will be quick and efficient, with good prospects of future maintainability.

The report outlines options for backing up code repositories and plans for redundancy under emergencies or failure to access Github.com. We also define and outline the training that ECMWF developers will need to adopt the Github workflows.

We also provide a cost estimate for moving ECMWF to GitHub Enterprise. The cost of these licenses is estimated at ~55K USD for the support of developers involved in Open Development (scenario 1), with an option costing ~98K USD for extension to all RD researchers and users of IFS (scenario 2, that includes external users that currently operate prepIFS). In addition, we estimate that an extension to support Github Copilot, an AI code development assistant based on GPT-4 would cost ~14k USD for 60 licenses. We estimate that the Github Copilot total cost for scenario 1 is ~82k USD and ~138k USD for scenario 2.

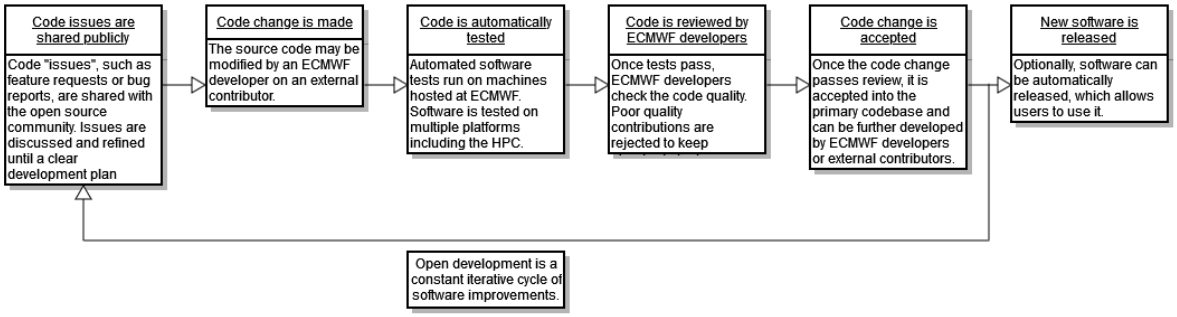
We conclude that using GitHub Enterprise offers ECMWF several benefits compared to the current system, and we suggest that switching to GitHub Enterprise would be a good investment.

We recommend the adoption of scenario 1 and, if financially possible and RD engagement, the adoption of full scenario 2.

Abstract

As part of its software strategy, ECMWF is exploring methods to make it easier to collaborate with its Member States and global community of software developers. Furthermore, ECMWF is exploring methods to improve the automation tooling available to its own software developers, with the dual aims of reducing developers' cognitive overhead and reducing the risk of software anomalies being released. This report examines how GitHub Enterprise could be used at ECMWF to encourage open development, including how to use GitHub Actions to improve ECMWF's software automation capabilities.

We demonstrate that GitHub Enterprise can be used to securely make the widely popular open platform github.com, the online home of ECMWF software. This gives ECMWF a mechanism to interact with the community of open-source developers more transparently, in particular provides a pathway for closer interaction with developers from its Member States.



We demonstrate that GitHub Actions solves multiple issues ECMWF developers have with Bamboo, the current tooling used for software automation at ECMWF. We demonstrate that GitHub Actions can be used effectively for ECMWF's complex software testing use-cases, including testing software multiple operating systems and compilers automatically in parallel, and including testing software automatically on the HPC. We demonstrate that automated testing on GitHub Actions is fast and so can be carried out with higher frequency than the existing Bamboo system, which we believe reduces the risk of software anomalies being released. We demonstrate that GitHub Actions has some capabilities requested by ECMWF developers which are either impossible or challenging to implement with the Bamboo system. We further demonstrate that GitHub Actions can be used to publish released software automatically to various repositories so it can be shared and reused by users and other developers. Importantly, we demonstrate that ECMWF developers can easily understand and extend the prototype GitHub Actions workflows so ongoing development may be carried out in-house.

The report presents options for source code backup strategies and disaster recovery. We present an estimate of costs to migrate ECMWF to GitHub Enterprise and a breakdown of training needed for ECMWF developers. We finalise be a list of recommendations to adopt GitHub Enterprise for open development.

1. Introduction

1.1. Scope

The scope of this report is limited to analysing the impact of moving the current software hosting from Bitbucket (part of the Atlassian suite) to GitHub Enterprise for all ECMWF software including production suites and configurations. This includes both software which will be public-facing and developed via open development policy, and private software not available to the public but managed within the same platform. An important part of this includes replacing the current Bamboo (another Atlassian tool) setup for testing and deployment with facilities offered by GitHub Enterprise.

Although the project scope initially did not include IFS, discussions and interviews with the IFS section seem to suggest that a similar approach could be adopted with IFS-specific tweaks in the workflow.

1.2. Introduction to Open Development

Open development is the process of developing open source software in public.

The open development philosophy revolves around transparency, collaboration, and inclusivity. It welcomes contributions from a diverse range of developers, values merit-based decision-making and empowers communities to collectively improve software. This approach fosters innovation, trust and a sense of shared ownership, resulting in high-quality, accessible and sustainable open-source projects.

Transparency

Open development emphasises transparency in every aspect of a project's lifecycle. The entire development process, including code changes, bug reports, discussions, is visible and accessible to everyone. This transparency builds trust within the community and ensures that project activities can be scrutinised and verified.

Member States will be able to follow and observe the full development process from the onset of new developments. As such, they are able to provide early feedback on the code.

Collaboration

GitHub facilitates a collaborative environment where developers from around the world can work together on software projects. It enables multiple contributors to work on code together, track changes and discuss improvements seamlessly. This philosophy encourages teamwork, enabling developers to combine their skills and ideas to create better software.

Due to openness, Member States will be able to access our developments and provide contributions back to ECMWF code alongside ECMWF developers.

Community involvement

Open-source projects thrive on involvement of a diverse community of developers, testers, designers and users with a variety of backgrounds and expertise. It's key to encourage active participation and to value the input of all community members.

GitHub is the de-facto home of open-source software on the internet. Adopting it at ECMWF is a step towards improving our interaction with the community.

Meritocracy

Evaluating contributions based on their quality and value ensures that the best ideas and contributions rise to the top.

We will profit from the expertise of our community, in particular our Member States, when they contribute back to our software.

1.3. Overview of GitHub

GitHub is the de-facto home of open-source software development on the internet. 87% of developers use GitHub[1] and 77% of organisations distribute open-source software using GitHub[2]. 413 million open-source contributions were made on GitHub in 2022[3].

GitHub can be used both by individuals and by organisations, with different features available at different pricing tiers.

GitHub can host both public and private software repositories.

GitHub has already been used at ECMWF for years to host some open source software, but this has historically always been alongside using Bitbucket in parallel for software development and limited to software packages that were public.

1.3.1. GitHub Enterprise

GitHub Enterprise is the paid version of GitHub for large organisations. It comes with many advanced features not available in GitHub's "Free" and "Team" tiers. Of these, features that are of particular interest for ECMWF are:

- Improved integration of GitHub Actions with internal ECMWF infrastructure.
- Extra features of GitHub Actions which make it easier to automatically build, test, and deploy software, such as environment protection rules used to configure different environments.
- SAML single sign-on (SSO) allowing integration with ECMWF user management.
- "Enterprise managed users" i.e. support for GitHub Enterprise-managed users for ECMWF, so ECMWF employees aren't required to use personal GitHub accounts.

[1] https://survey.stackoverflow.co/2022#technology-version-control
[2] https://openuk.uk/wp-content/uploads/2023/04/REPORT.pdf
[3] https://octoverse.github.com/

- Support for advanced auditing tools.

You can find the full list of features here: https://github.com/pricing.

1.3.2. GitHub Actions

GitHub Actions is the native tooling to run Continuous Integration (CI) and Continuous Delivery/Deployment (CD) on software which lives on GitHub. It is used to automatically build and test software packages in order to look for problems. GitHub Actions can also be used to automate various parts of the development cycle on GitHub.

Historically, ECMWF has used Atlassian Bamboo for this purpose, see section 1.4.2.

When using GitHub Actions, we define "workflows", which are version-controlled YAML files which live within each software repository. This allows developers to easily modify workflows and track their changes over time, in the same way they do for code changes.

1.4. Overview of the Atlassian stack

ECMWF currently uses the Atlassian stack for software development. Specifically, Bitbucket for version control, Bamboo for CI/CD, and Jira for issue management.

1.4.1. Bitbucket

ECMWF has a privately hosted Bitbucket instance which hosts most of our software repositories. The features of Bitbucket and GitHub are similar when it comes to version control and code contributions.

However, because our Bitbucket is private, it means we can't use the ECMWF Bitbucket for open development.

1.4.2. Bamboo

Bamboo is the Atlassian tool for automatically building, testing, and deploying software (CI/CD). Bamboo has been used at ECMWF for many years and suffers from limitations which cause pain both during initial configuration and during ongoing maintenance. Bamboo is inappropriate for ECMWF's cross platform testing needs because it was originally designed for testing Java code that is inherently cross-platform. Bamboo has limited expression for platform-specific workflows and although not impossible, it is laborious and error-prone to implement platform testing.

Unlike GitHub Actions workflows, the equivalent Bamboo "build plans" must be configured using a graphical user interface (GUI) and are not version controlled. This has been reported as causing frustration and lost time for ECMWF developers.

1.4.3. Jira

Jira is the Atlassian tool for issue management. Jira is used extensively across ECMWF, for everything from the Service Desk to software tickets.

1.5. Findings from Interviews with the ECMWF Development Section

In September 2022, we interviewed 34 developers and management in the Development Section to understand their needs for a Git-hosting platform and for CI/CD. The summary of findings is presented below.

1.5.1. ECMWF needs

1.5.1.1. Must have

- CI runners should integrate with ECMWF infrastructure including the HPC. We must be able to build and test code on our internal machines (including the HPC) as well as more typical platforms.
    - Motivation: ECMWF has complex operational requirements for its software. Our software is used on a wide range of operating systems and is built with a range of compilers - including the HPC. We must therefore build and test our software on the same (or at least similar) platforms before it is released to minimise the risk of software bugs and other anomalies reaching operations.

- Ability to run CI with a matrix of build options: different platforms, different compilers, and different configurations.
    - Motivation: as above, ECMWF's operational requirements require testing on multiple platforms, compilers, and configurations. We must be able to easily define a matrix of different build options for testing and the matrix must be easily reconfigured. This ensures maintenance costs are low whilst minimising the risk of software bugs, as explained in the previous point.
- Strong Jira integration. We must be able to link commits/pull requests (PRs) to Jira tickets.
    - Motivation: Jira is used extensively across ECMWF, for everything from the Service Desk to software tickets. Bitbucket has in-built integration with Jira because both Bitbucket and Jira are Atlassian tools. Links between Jira tickets and code changes are used for record-keeping and documentation. They serve as an important communication aide between ECMWF developers and the service desk. Therefore, we must have similar strong integration between GitHub Enterprise and Jira.

1.5.1.2. Should have

- The Bitbucket diff tool is widely liked and considered powerful by developers - we need the same thing from GitHub. GitHub does, in fact, have an identical diff tool.
    - Motivation: We don't want to take a backwards step by migrating to GitHub. ECMWF developers should be able to continue using tooling which they consider to be valuable and useful for their work.
- Easy access management.
    - Motivation: GitHub has had, for a few years, some limited use at ECMWF as some of our open source codebases are hosted there as well as on Bitbucket. However, giving ECMWF developers access to our codebases on GitHub has always been a manual task for admins: admins must manually give each ECMWF developer access to

ECMWF's organisation space on GitHub and also set permissions for each repo each developer needs access to. With GitHub Enterprise, we should aim to improve the system so access management involves less manual work.

- Ability to run tests which depend on multiple software packages. Specifically: the ability to run tests **using specific branches** of other packages.
  - Motivation: We need this so developers can see whether their code change has broken a downstream package (i.e. a package which depends on another package) before merging it.
- Fast feedback from CI.
  - Motivation 1: The software development loop involves changing code then testing it for correctness (CI). CI must be fast so it doesn't bottleneck the development loop.
  - Motivation 2: When CI is fast it can be run more frequently. More frequency CI makes it more likely that software bugs are caught early in the development cycle. ECMWF has a large "stack" of codebases with multiple layers of dependencies. Ideally, when a codebase changes upon which other codebases depend, the "downstream" codebases should be built and tested as well to check they still function correctly with the new dependent code. Currently, with Bamboo, this stack can be built and tested all together, but it takes 2+ hours to run and so can't be run on every code change. With faster CI, we could potentially test the stack of downstream codebases on every change to dependent code.
- Focused notifications from CI. They should be infrequent, informative, and targeted at the correct developer/team.
  - Motivation: ECMWF developers currently feel overloaded by notifications from Bamboo. Frequently, they are not read or simply switched off. This means important notifications can be missed.
- Visibility on the CI/CD pipeline and task flow.
  - Motivation: automated CI/CD pipelines are often complex and if part of the pipeline fails, it should be straightforward for a developer to find what failed and why. Visibility is currently a big challenge with Bamboo, because Bamboo build plans frequently trigger other Bamboo build plans, but Bamboo doesn't provide a central location where all build plans triggered by a single code change can be monitored. Developers have to gather information from multiple locations to understand and solve problems, which wastes time.
- Ability to download test outputs (e.g. images from regression tests).
  - Motivation: Some test outputs need to be checked by a human if the tests break.
- Ability to run CI/CD on Git tags.
  - Motivation: Git tags are commonly used when releasing a new software version. Releasing a new software version often involves extra automated workflows beyond the usual development cycle workflows, such as continuous delivery/deployment. With Bamboo, developers can't specify build plans to run when a Git tag is created, which makes it a challenge to run special workflows when a new software version is released.

1.5.1.3. Nice to have

- Ability to run CI locally (on developers' development machines) for fast feedback when developing the CI itself.
  - Motivation: the traditional CI development cycle is slow. For example, on Github: a developer changes a GitHub Actions workflow, then must commit the change, push their change to GitHub in order to trigger the workflow to test that it works, before iterating as needed. Slowness is caused because the remote cloud machines running the workflow have a set-up time for each workflow run, which adds overhead. The workflow run itself can be slow if the remote machine isn't as powerful as the

developer's own machine. Neither of these issues are big problems for *software* development because the overhead is small and remote machine resources can be increased for workflows which take a long time. However, for *workflow* development, where the workflow is likely to be repeatedly retriggered in a tight loop, the overhead can add up and cause slowdown.

1.5.2. Problems with the current Bitbucket/Bamboo and legacy GitHub setup

In order of most frequently reported:

1. Too much clicking to set up and modify Bamboo (because it must be configured using a GUI).
2. Bamboo build plans aren't clear: it's hard to know what has been triggered and why.
3. Managing developer access on GitHub is a pain because it's all manual.
4. Notification overload from Bamboo.
5. It's hard or impossible to build and test a package using a specific branch of a dependency package.
6. Tests are sometimes disabled or skipped because they take too long to run.
7. Bamboo logs are confusing to debug.
8. Bamboo build plans aren't version controlled so developers can't check what's changed after a build plan has been updated.
9. Sometimes developers have to wait 30-60+ mins for a Bamboo agent to become available to run their tests.
10. Can't build on Git tags with Bamboo.
11. Fussy Duck asks developers to sign the CLA for every single GitHub repo they contribute to, rather than once for all ECMWF repos.

These findings highlight significant technical frustrations with using Bitbucket and Bamboo.

1.5.3. Additional concerns

Other relevant information surfaced during these interviews related to software testing and software development at ECMWF which may impact open development:

1. Currently, synchronised releases of software induce much manual work. Adopting workflows like GitHub Actions would be conducive to reducing it.
2. Developers often rely on localised testing because testing on Bamboo is too slow. Adopting fast turnaround testing workflows based on GitHub Actions would encourage the adoption of a common testing practice.
3. Existing Atlassian-based CI only tests integrations into develop branches. GitHub Actions would allow early testing across feature branches, thus enabling fast error detection.

1.6. Project Goals

Based on our understanding of ECMWF's needs, our goal for this project was to answer the following questions:

1. How can we better encourage open source contributions to ECMWF software packages?
2. Is Github Enterprise a good fit for the needs of ECMWF open software development?

3. Can ECMWF reliably use GitHub Enterprise for its operational needs and as a platform to collaborate with its Member States?
4. Can we use GitHub Actions to meet ECMWF's complex needs for building and testing software in multiple environments (including on the HPC)?
5. Can we use GitHub Actions to improve the developer experience compared to Bamboo?
6. Can we use GitHub Actions to deliver and deploy ECMWF software?
7. Can we ensure it's straightforward for ECMWF developers to maintain and expand a CI/CD system built on GitHub Actions?
8. Can we integrate GitHub Enterprise with Jira?
9. Can we integrate GitHub Enterprise with ECMWF SSO?
10. What are the expected costs if ECMWF chooses to migrate to GitHub Enterprise (including GitHub Actions)?
11. What training would be required for ECMWF staff to make effective use of GitHub Enterprise?

1.7. Overview of report

In this report, we aim to demonstrate the various functionalities of GitHub Enterprise and GitHub Actions to show how they can be used to benefit ECMWF and its Member States. In section 2, we demonstrate how GitHub Actions can be hosted on machines at ECMWF in order to test on the platforms we care most about, including the HPC. Sections 3-5 demonstrate how GitHub Actions can be used for automated software testing, automated sharing of software libraries and compiled code, and automated deployment of web applications respectively. Section 6 explores how GitHub Actions workflows can be reused and extended by ECMWF developers. Sections 7-8 explore the integration of GitHub Enterprise with ECMWF SSO and with Jira respectively. In section 9 we explore options for disaster recovery and backup plans. Section 10 breaks down the expected costs of migrating to GitHub Enterprise. We offer recommendations in section 11. Annex 1 is a glossary of terms, and Annex 2 details the training required to make effective use of GitHub Enterprise.

2. Investigation of Self-hosted GitHub Actions Runners

When defining a GitHub Actions workflow to run CI/CD, we can choose the environment in which it runs. For example, the environment will always define an operating system.

GitHub Enterprise includes access to GitHub-hosted Actions runners. These runners are available in a few versions of Ubuntu and MacOS. They're typically not powerful machines, but more powerful machines are available at increased cost. We have minimal control over these machines.

By using self-hosted runners, we have much more control of our CI/CD environments. For example, we can define specific compilers and compiler versions to use. We can also better control the hardware resources, so we can use more powerful machines to speed up any compute-heavy workflows. A major benefit is that by self-hosting, we can have runners inside the ECMWF network, which allows us to run CI/CD on the HPC and to deliver compiled software to internal ECMWF repositories.

In this section, we explain how self-hosted GitHub Actions runners can be used for CI/CD at ECMWF.

2.1. Overview of self-hosted GitHub Actions runners at ECMWF

We prototyped using self-hosted GitHub Actions runners in multiple scenarios to assess their suitability.

There are 4 categories of self-hosted runner:

1. Platform builders run CI/CD on core ECMWF software packages. These are powerful machines which decreases our compilation time significantly vs GitHub-hosted runners.
2. MacOS builders run CI/CD on core ECMWF software packages. These self-hosted runners are more powerful and cheaper than GitHub-hosted MacOS runners.
3. HPC builders run CI on the HPC, for which we need to be inside the ECMWF network.
4. Web builders run CD for web apps, for which we need to be inside the ECMWF network.

Self-hosting GitHub Actions runners is necessary but not sufficient to show that GitHub Actions could be superior to Bitbucket for ECMWF's needs. ECMWF also has self-hosted Bitbucket runners with similar functionality to our GitHub-hosted ones. However, in order to unlock the benefits of GitHub Actions, we must first prove that it can meet ECMWF's requirement to be able to run CI/CD in multiple complex environments.

2.2. Provisioning and configuring self-hosted runners

All categories of self-hosted runner are automatically provisioned and configured with Terraform and Ansible, with the exception of the MacOS builders which must be manually provisioned. They are hosted on a private vSphere cluster at ECMWF owned by the development section, again with the exception of the MacOS builders which are hosted by MacStadium.

The tooling (Terraform, Ansible, vSphere) used to provision and configure our self-hosted runners is standard tooling at ECMWF. These tools are used when working with any ECMWF VM. We have used the same tooling to reduce the maintenance overhead of our self-hosted runners.

Full technical documentation is on Confluence: How to provision and configure self-hosted runners

2.3. Platform Builders

The platform builders run CI/CD on core ECMWF software packages. We provisioned the platform builders with 8 vCPUs and 32GB RAM to speed up compilation time and give developers fast feedback from their automated tests.

Platform builders are used to run automated tests every time a code change is made.

Currently we can test in the following environments:

- Ubuntu 22.04 with gnu compilers
- Debian 11.5.0 with gnu compilers

- Rocky Linux 8.6 with gnu and clang compilers
- CentOS 7.9 with gnu 7.3 and gnu 8 compilers
- Fedora 37.1 with gnu compilers

The available resources and environments are easily changed and automatically deployed. By using self-hosted runners, we can make use of environments which are simply unavailable to GitHub-hosted runners. We can also add extra value to ECMWF's Member States, e.g. the reason we have gcc 7.3 as one of our environments is because the Met Office HPC uses that compiler version. Now we can be sure ECMWF software will continue to function as expected on the Met Office HPC.

Using GitHub-hosted runners, the time to build and test ecCodes is 5 minutes. On the self-hosted runners, it takes 2.5 minutes, a 2x speed-up. We expect the speed-up to be even greater for more resource-intensive compilations.

2.4. MacOS Builders

The MacOS builders are much the same as the platform builders - they are fairly powerful machines used to run CI/CD on core ECMWF software packages.

They are unique in a few ways:

- MacOS runners are hosted by MacStadium, because we currently don't have any available MacOS cloud hardware at ECMWF. This means they're outside the ECMWF network.
- MacOS runners must be manually provisioned by logging into the ECMWF MacStadium account and requesting a new machine. The hostname and login credentials for this machine must then be manually added to the repo containing the Ansible configuration scripts before it can be configured.
- MacOS runners are more powerful than GitHub-hosted runners but are weaker than the platform builders. This means, unlike the platform builders, they're not suitable for running tests every time a code change is made, because the tests would take too long to run. We instead use the MacOS runners to run tests nightly. We find this to be a suitable compromise given MacOS is not used in operations at ECMWF.

2.5. HPC Builders

The HPC builders run CI/CD on core ECMWF software packages whenever a code change is made, in the same way as the platform builders.

Unlike the platform builders, the HPC builders are lightweight VMs which do not run CI/CD on their own hardware. Instead, they submit CI/CD jobs to the ECMWF HPC using Troika, a software tool developed by ECMWF for exactly this purpose. The logs and status from the HPC are monitored and reported back to GitHub so they can be used by developers to diagnose any test failures.

Currently we can test in the following environments:

- gnu 12.2.0 compiler
- gnu 8.5.0 compiler

- nvidia 22.11 compiler
- intel-2021.4.0 compiler

2.6. Web Builders

The web builders are used to deploy ECMWF web applications using kubernetes (k8s), which requires being inside the ECMWF network.

The VMs are lightweight but come pre-configured with all the libraries and dependencies we need to deploy using k8s.

2.7. Limitations of self-hosted runners

The main limitation of using self-hosted runners is that they're not automatically scalable. We have a fixed number of self-hosted runner instances, and if they're all busy running CI/CD, any new CI/CD requests will go into a queue.

With GitHub-hosted runners, we're free to run as many CI/CD workflows in parallel is we like.

With our current Bamboo setup, we have the same limitation as with self-hosted runners.

As explained, it's possible to automatically provision more self-hosted runners using our automated Terraform + Ansible scripts. In that sense we do have the ability to scale our runners, although we must trigger it manually.

The true scaling limit is set by the underlying hardware which makes up our dedicated vSphere cluster. Once that hardware reaches capacity, we would be forced to procure additional machines in order to continue scaling our GitHub Actions runners. We estimate we can scale up our runners by a factor of 3x on the current hardware before running into performance problems.

2.8. Security implications of self-hosted runners

2.8.1. Background

Open-source developers contribute code changes with a mechanism called a "pull request" (PR). This is a request for our developers to pull their code changes into our repo.

Before we accept the PR and merge the code changes, we want to test the code to make sure it works as intended. The most effective way to do so is to use our GitHub Actions CI, as we would for new code contributions from ECMWF developers.

There are two security concerns to running GitHub Actions on a PR from an external contributor:

1. Our self-hosted runners are inside the ECMWF network so we must make sure no malicious code is allowed to run on these machines.

2. Our GitHub Actions have access to "secrets" which we must not allow external contributors to maliciously access. e.g. secret access tokens, SSH keys, other login credentials.

We have developed a process for managing code contributions from open-source contributors to mitigate security risks whilst still allowing CI to run.

2.8.2. Public PR Approval Process

The below process allows us to use our self-hosted runners to run CI on a PR from a non-ECMWF developer:

1. When the PR is opened from a contributor, a GitHub Action automatically labels the PR with the "contributor" label.
2. An ECMWF developer sees that we have a new contributor PR and reviews the code change.
    1. The developer should be particularly suspicious of any changes to our GitHub Actions workflows. An open-source contributor should not usually need to change these. The developer must make sure the contributor is not trying to maliciously access our GitHub Actions secrets by e.g. sending their contents to a third party service.
    2. The developer must also review the code changes to ensure they aren't malicious.
3. If the ECMWF developer is confident the code change is not malicious, they assign the "approved-for-ci" label to the PR.
4. The GitHub Actions CI runs automatically on our self-hosted runners once the "approved-for-ci" label is added.
5. If the PR is updated with another code change, the "approved-for-ci" label is automatically removed by a GitHub Action. We must then repeat steps 2-4 as necessary.
6. Once the PR passes CI and is approved by ECMWF developers, it is merged into the codebase as normal.

2.8.3. Access controls for GitHub Labels

GitHub has 5 tiers of access to a repo, called "roles":

- Read: a contributor can read the contents of a repo. This allows public contributors to create a "fork" of the repo and submit code changes back using a PR.
- Triage: read access, plus the ability to manage issues and pull requests.
- Write: a contributor can write directly to the repo, as well as having triage access. This allows ECMWF developers to create new branches and push to the repo without creating a fork.
- Maintain: a maintainer can manage some repository settings, as well as having write access. Typically, we set up maintainers, so they are the only contributors allowed to merge code changes to the main branches. Maintainers are the ECMWF developers primarily responsible for the given repo.
- Admin: everything a maintainer can do, plus the ability to manage all repo settings, including adding collaborators.

By default, on GitHub, any contributor with "triage" or "write" access to a repo is allowed to add labels to a PR.

However, we might prefer to restrict label access to maintainers, since maintainers are best placed to judge whether or not a code change is malicious.

Restricting labels only to maintainers isn't possible using the standard GitHub roles. Instead we would need to create a custom role, which would have identical permissions to the GitHub-provided "write" role except without the "Add or remove a label" permission.

2.9. Conclusion

It's feasible to use self-hosted GitHub Actions runners for open development at ECMWF to meet ECMWF's complex software testing needs. Specifically, we've proven it's possible to test ECMWF software on a wide range of platforms, including the HPC, in a way which is more extensible and maintainable than the current Bamboo system.

There are two important caveats:

1. Self-hosted runners are limited by the resources we assign to them from the ECMWF infrastructure, which is the same limitation that Bamboo has;
2. There are critical security implications when using self-hosted runners for open development because it has the potential to open the door for malicious code execution. ECMWF developers engaging in open development will need to be trained on the correct process.

We recommend using self-hosted GitHub Actions runners for CI/CD with GitHub Actions at ECMWF in addition to the GitHub-hosted runners as appropriate for each project.

3. Continuous Integration with GitHub Actions

Continuous Integration (CI) is the process of automatically testing software changes before merging them into the main codebase.

The goal of a good CI system is to give developers comprehensive feedback as quickly as possible on whether or not their code change has introduced a fault to the codebase.

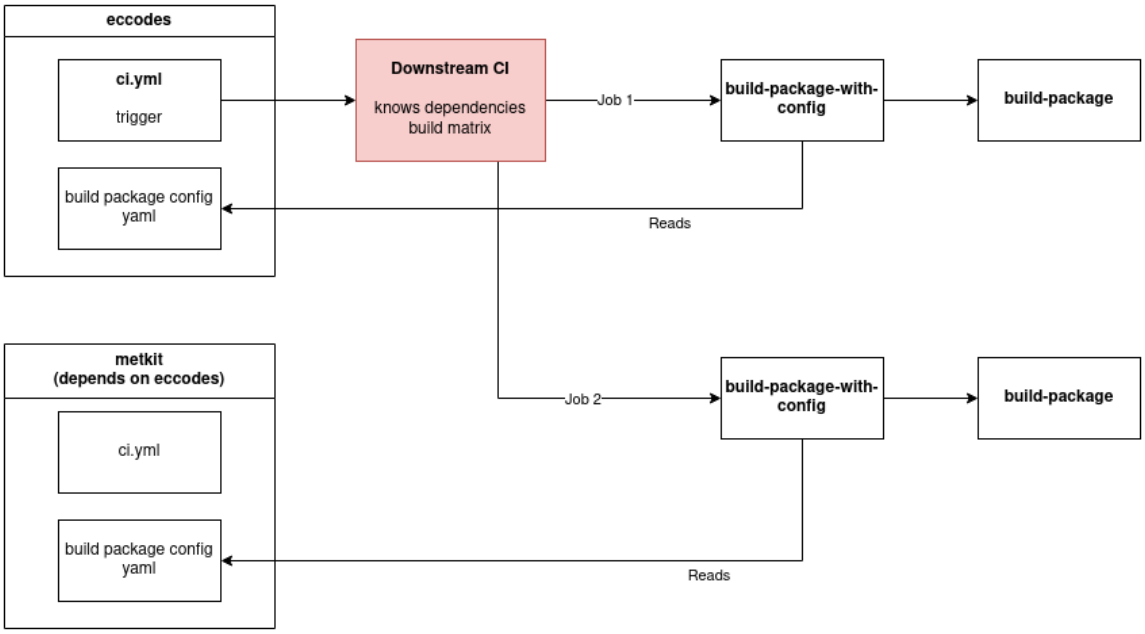
In this section, we explain how GitHub Actions can be used to automatically test ECMWF software on various platforms.

3.1. 3.1. Downstream CI

We prototyped a system called "downstream CI" which builds and tests all packages that depend on the one being changed (i.e. downstream packages) to give developers comprehensive quick feedback on whether or not their code change has introduced a fault in any part of the core ECMWF software stack. This goes beyond typical CI, which will usually only test the one package being directly changed and gives developers confidence that their work won't cause a breaking change in another ECMWF package.

However, comprehensive feedback loses value if it's not also delivered quickly, because developers have to stop and wait for the CI. We have implemented a few performance enhancements with the target of giving developers feedback within 10-15 minutes:

1. We only build downstream packages. e.g. for ecCodes, which lots of packages depend on, we must build and test most of the dependency tree. However, for metkit, which fewer packages depend on, we can build and test the dependency tree only from metkit onwards.
2. We use caching to make sure we don't repeatedly rebuild packages.



The above diagram shows the sequence in which Github actions and workflows are called within downstream CI.

1. eccodes/ci.yml workflow is triggered (the trigger event is defined here, and usually runs on pull request or push to the main branch)
    a. calls Downstream CI reusable workflow which contains the dependency tree for the whole software stack.
    b. Downstream CI is split into 2 workflows, one for self-hosted runners and one for HPC, therefore eccodes/ci.yml has to call each workflow separately.
2. Downstream CI triggers build for eccodes first, and then recursively for packages that depend on eccodes (in the above diagram only metkit).
    a. each job calls build-package-with-config, which is a wrapper around build-package. It reads the build configuration file for the current package and passes that to the build-package action.

3. Build-package action builds the package including its dependencies and executes tests. Each build is cached in GitHub cache and as an artifact (in the scenario above eccodes would be restored from cache during the metkit job)

3.1.1. Downstream CI Environments

Downstream CI is split into 4 environments:

- Platform builders public downstream CI
- HPC builders public downstream CI

- Platform builders private downstream CI
- HPC builders private downstream CI

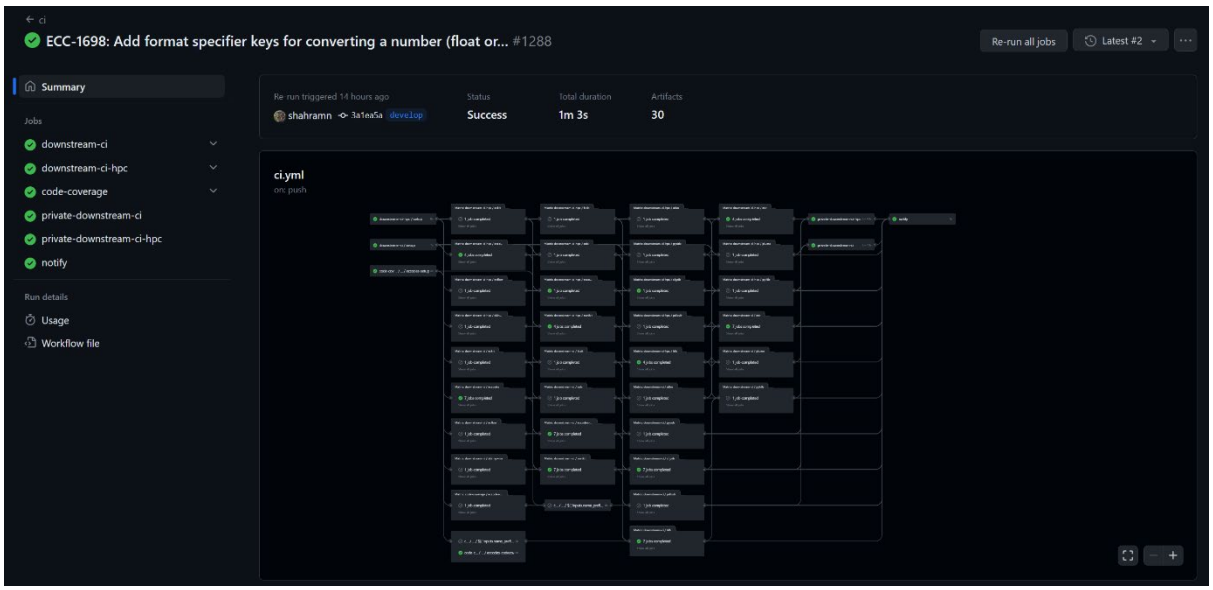
As explained in section 2, we use various self-hosted runners for CI. For downstream CI, we use the platform builders, which build and test software on various Linux distributions, and the HPC builders, which build and test software on the HPC.

We also have the concept of "public" and "private" downstream CI. The reason is that the dependency tree may contain both public and private repos. If we trigger downstream CI from a public repo, the build logs for all the packages in the full dependency tree will be available publicly via the GitHub Actions logs panel for the triggering repo. This is not acceptable if some repos in the dependency tree are private.

The solution is to have a separate job for private downstream-ci. The public triggering repo triggers the private workflow, which runs in a private repo so the logs are private. The private downstream CI is designed so that the status of the private build (pass/fail) is reported back to the public triggering repo, along with a link to the private logs so that ECMWF developers can investigate any problems.

The workflows which run on the platform builders vs HPC builders are independent of each other and so happen in parallel. The private downstream CI depends on the success of the public downstream CI, so they happen in series.

The below image shows a downstream-ci workflow run for ecCodes on GitHub. On the left you can see links to the logs for the 4 environments and in the centre you can see the full downstream-ci job tree. Jobs with green ticks were successful and jobs with grey slashes were skipped because they were unnecessary (i.e. because they were jobs to test a package which doesn't depend on ecCodes and so don't need to run when ecCodes is changed).



3.1.2. Extensibility of Downstream CI by ECMWF developers

Given the inherent complexity of the downstream CI system, it's important we demonstrate that ECMWF developers can easily understand and extend our prototype workflows so ongoing development may be carried out in-house. We ran an initial training workshop with a select group of three ECMWF developers, consisting of an hour-long presentation explaining the background, principles, and technical detail, followed by a 2-3 hour long practical session where each developer participating in the training session added their own software package to the downstream CI build tree.

Each developer was able to add a new package to the downstream CI build tree as a result of the workshop.

3.2. Nightly CI

Some codebases have longer-running tests which are untenable to run on every code change. We still want to run these tests frequently and automatically, so any software faults are noticed and fixed as quickly as possible. To prototype how to handle this case, we set up nightly CI for ecCodes. As the name suggests, we run automated CI every night at a specified time.

There are currently two categories of test which we run nightly:

- Extended tests. These are data-intensive tests which need longer to run than typical software tests.
- MacOS tests. Our self-hosted MacOS builders aren't as powerful as our platform builders or the HPC, so CI takes longer to run on them.

3.3. Build Matrix

A "build matrix" defines multiple environments (operating systems + compilers) on which we want to run CI. When running a workflow, the build and tests for each entry in the matrix is run in parallel.

On the platform builders we currently have 7 operating system + compiler combinations as defined in section 2.3.

On the HPC builders we currently have 4 compilers in the build matrix as defined in section 2.5.

3.4. CI for Quality Control

GitHub Actions can be used to test the quality of source code as well as the functionality of the code. Broadly, quality control falls into two categories:

1. Checking source code style
2. Checking the coverage of automated tests (i.e. what percentage of the codebase is tested by our automated tests)

For example, on Python codebases we use GitHub Actions to check the source code style by:

- Running `isort` to check that imported modules are in alphabetical order
- Running `black` to check that the source code has the correct formatting
- Running `flake8` to check for various source code style issues and bugs, such as syntax errors, unused variables, and attempted usage of undefined variables.

We use the tool "codecov" to check the coverage of our tests. When a code change is propose via a PR, GitHub Actions will automatically run codecov and leave a comment on the PR which tells the developers the change in code coverage which the PR will cause if it's merged. This is a useful review tool: if a contributor opens a PR which adds lots of new code but doesn't test it well, the reviewer will see the drop in code coverage and can request the contributor to add more tests. This helps us maintain high code quality even as we have more frequent open source contributions thanks to open development.

3.5. CI for mars-hpss

HPSS is a proprietary IBM software package which ECMWF uses to interact with MARS. We need to be able to test changes to MARS software which uses HPSS but it's essential that access to HPSS is restricted to a select few approved developers at ECMWF.

We prototyped using GitHub Actions to run CI for the mars-hpss repo to demonstrate:

1. We can securely download and install HPSS from a private Nexus repository
2. We can build and test ECMWF software which depends on HPSS
3. We can clean up all traces of HPSS from our self-hosted runners once the CI is finished.

The solution was simple using standard GitHub Actions features. We used GitHub Actions secrets to securely store an access token for the private Nexus repository. We implemented two cleanup steps at the end of the CI job which are always forced to run, even if earlier steps in the CI fail or have unexpected issues.

The full workflow can be found here: https://github.com/ecmwf/mars-hpss/blob/develop/.github/workflows/ci.yml

3.6. Conclusion

GitHub Actions can be used for all CI use-cases at ECMWF. GitHub Actions enables more powerful CI setups and improves the developer experience as compared to Bamboo.

Recommendations:

- ECMWF should use the Downstream CI strategy to automatically test downstream software packages on every code change to upstream packages in order to catch software issues early in the development cycle.
- ECMWF should run long-running CI nightly, instead of on every code change, to allow fast iterations of the development cycle whilst still getting the advantages of frequent automated testing.
- ECMWF should use GitHub Actions for automated quality control of its codebases.

- ECMWF can use GitHub Actions to securely build and test software which depends on restricted-access software packages such as mars-hpss, but due care should be taken to ensure any newly created GitHub Actions workflows clean up the restricted-access software package when it is finished. We recommend multiple experts review any new workflow using restricted-access software to ensure its usage is secure. We estimate this special case applies to a tiny fraction of ECMWF CI workflows.

4. Continuous Delivery with GitHub Actions

Continuous Delivery (CD) is the process of automatically publishing software packages to package repositories, or the module system on the HPC, so they can be deployed into operations.

In this section, we demonstrate how GitHub Actions can be used to deliver software to an assortment of delivery targets. To demonstrate this flexibility, we have explored 3 disparate targets: ECMWF Nexus, ECMWF Homebrew, and PyPI.

4.1. Nexus CD with GitHub Actions

Nexus is the ECMWF-hosted compiled software repository.

We built a reusable workflow to build and deliver ECMWF packages called `create-package`. The workflow will create DEBs/RPMs for cmake-based software packages on Debian 11, CentOS 7, and Rocky 8, then upload the binaries to ECMWF Nexus repositories for each platform. The workflow is configurable to allow developers to specify CPack options and to override upload repositories.

We demonstrated 2 types of Nexus CD:

1. CD when a release is created. When a Git tag is created in a codebase, packages are automatically delivered to stable repositories on Nexus.
2. Nightly CD. We deliver to the bleeding-edge Nexus repositories every night for some repos.

The reusable workflow is extensible to add new platforms if we need to in the future.

4.2. Homebrew CD with GitHub Actions

Homebrew is a package manager commonly used on MacOS, but which can also be used on Linux.

We overhauled the existing ECMWF Homebrew GitHub repository, updated the formulas (the instructions to build each ECMWF package), and created GitHub actions which build binaries using the formulas and then upload them to Nexus.

4.3. PyPI CD with GitHub Actions

PyPI is a public python package manager.

ECMWF software has been published to PyPI for many years already. We created a reusable workflow which makes it easier for ECMWF developers to publish their packages to PyPI, which you can find here: https://github.com/ecmwf-actions/reusable-workflows/blob/main/.github/workflows/cd-pypi.yml

4.4. Conclusion

We conclude that GitHub Actions can be used to deliver software to a plethora of repositories. This will allow ECMWF to select and use the most appropriate delivery mechanisms whilst being confident that GitHub Actions will be extensible to support it. There is no evidence to suggest GitHub Actions is poorly suited to delivering software to any particular repository type.

Recommendations:

- ECMWF should use GitHub Actions for all continuous delivery needs.

5. Continuous Deployment of Web Apps with GitHub Actions

Continuous Deployment (CD) is the process of automatically deploying software applications, typically web applications, so they can be used by users.

There are two types of web app deployment used at ECMWF. For the purposes of CD, both types of web app deployment are essentially the same. The critical thing is that the workflow runs on our self-hosted web runners, so that we are inside the ECMWF network and so can run Kubernetes (k8s) commands.

In this section, we demonstrate how GitHub Actions can be used to deploy web apps at ECMWF following the preferred deployment process of the web team.

5.1. Desired web app deployment process

ECMWF developers have a preferred deployment process for web apps, which we outline here. Web app deployment involves two core technologies: docker and kubernetes (k8s). Docker is used to build platform-independent "images" for the elements which together form full web app. Docker images are pushed to a "container registry" when built so they can be downloaded and used by other tools. K8s is used to download and deploy the docker images for a web app as a single, cohesive, scalable, and robust system.

ECMWF typically uses 3 deployment environments for web apps, named "prod", "test", and "dev". Prod is the production environment, i.e. the version of the app which is used by users. Test is the staging environment, where changes being considered for release to production can be tested together. Dev is the development environment, used by developers to test their features during development.

Typically, we have a 1:1 mapping between deployment environments and Git branches. Usually, we use main/master for the prod environment, a release branch or develop for the test environment, and develop or a feature branch for the dev environment. It's important for

the mapping to be configurable on a per-project basis so developers have the flexibility to configure their applications however is best.

Sometimes, but not always, a web application is split across multiple Git repos. For example, we may have a repo for the front end and a separate repo for the back end. In this situation, developers want to be able to do two things: 1) redeploy the application whenever either repo changes; 2) have full control over which branches of each repo are deployed to each environment.

The full deployment process is:

- A code change is made to one of the repos which makes up a web app on one of the branches which has been chosen to be deployed to one of the deployment environments.
- Tests are run automatically. If successful, docker images are built and pushed to a container registry. Typically, this is the ECMWF self-hosted container registry ECCR.
- A tool (kustomize or skaffold) is used to automatically generate k8s config.
- A tool (kubectl apply or helm) is used to apply the config to an ECMWF-hosted k8s cluster.
- A developer monitors the deployment to make sure it has been successful.

5.1.1. Requirements for an ideal web app CD system

Below is a list of requirements for the web app deployment process which we gathered from ECMWF developers.

- Developers must have visibility on what was deployed, when it was deployed, and what triggered the deployment.
    - We mostly have this with Bamboo, though visibility on the full deployment pipeline is a challenge and it's not always obvious what triggered a deployment. In the event of a deployment failure, it's not always obvious what went wrong.
- Developers want to be able to control which branches of the repos which make up a web app are deployed to each deployment environment.
    - This is not easily configurable with Bamboo.
- Developers want the deployment process for prod, test, and dev to be identical to reduce the risk of problems occurring during a production deployment.
    - We currently have this with Bamboo.
- Developers want to manually approve the final step in the production deployment after all k8s configuration has been generated by the automatic build process, so they can manually check the generated config before deploying. This is to reduce the risk of a problem occurring during a production deployment.
    - We currently have this with Bamboo.

In the following sections, we demonstrate these requirements can be met by GitHub Actions.

5.2. Web app CD prototypes: IFSHub and Polytope

5.2.1. IFSHub CD prototype

IFSHub is a complex web application designed to be made up of multiple other web applications. Currently there are two web applications: IFSHub itself and PrepIFS. These two web applications are developed across three GitHub repos, with an additional two GitHub repos used for deployment. A summary of each repo is below:

- ifshub-frontend contains the front end ReactJS code for both IFSHub and PrepIFS.
- ifshub-backend contains the back-end Python code for IFSHub.
- prepifs-backend contains the back-end Python code for PrepIFS.
- ifshub-kubernetes contains the k8s config for the elements within IFSHub and PrepIFS, as well as a script to generate the combined k8s config, called the "manifest".
- ifshub-deployment contains the state of which docker images are deployed in each deployment environment, a GitHub action to update the state and trigger the deployment, as well as GitHub Actions workflows to deploy IFSHub and PrepIFS.

Each development repo (ifshub-frontend, ifshub-backend, and prepifs-backend) contains its own docker image definitions and Github Actions Workflows to build and deliver its docker images. i.e. the knowledge for how to build each codebase lives alongside the relevant code. Each development repo also contains the configuration for which of its branches get deployed to each of the deployment environments.

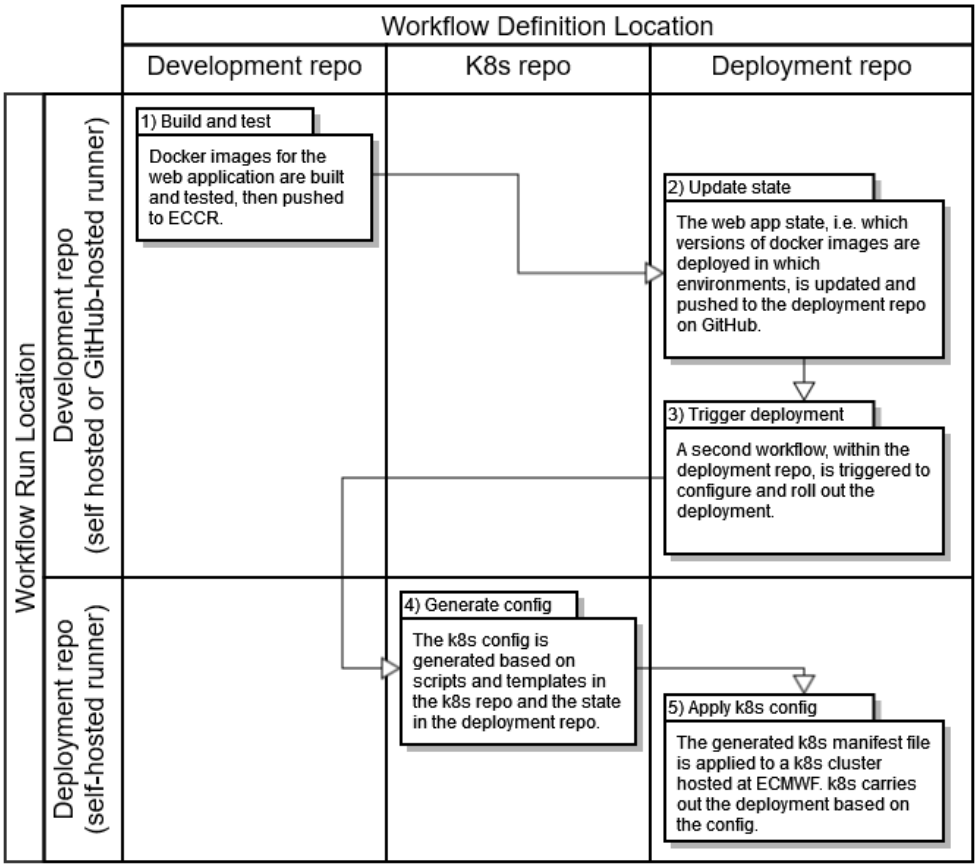
The deployment itself is done centrally, in the ifshub-deployment repo, to improve visibility on the deployment pipeline. Developers only have a single place to check to see all deployment logs. The deployment repo also contains the state for what is currently deployed in each environment, in the form of version-controlled text files. To understand the motivation for this decision, consider the case where IFSHub deployment is triggered by the ifshub-backend repo. The ifshub-backend workflow can tell ifshub-deployment which version of the backend docker image should be deployed, but how does ifshub-deployment know which version of the frontend to deploy alongside it? We solve this issue by storing the docker image names in version-controlled text files. The added advantage is that developers can always see exactly which versions of their dockerfiles are deployed, plus, thanks to version control, they can see the history of their deployments as well.

The deployment flow is described below and with a supporting diagram.

- A code change is made to a branch relevant for one of the deployment environments in one of the development repos (ifshub-frontend, ifshub-backend, or prepifs-backend).
- The code change triggers automatic tests, then automatically builds and pushes the docker image to ECCR within the development repo.
- The final step in the development repo workflow uses the GitHub Action defined in the deployment repo. This GitHub Action first updates the state of which docker images are deployed in each deployment environment by creating a git commit in the deployment repo, then triggers the deployment using a repository dispatch event from the development repo to the deployment repo.
- The triggered workflow runs in the deployment repo and does two things: 1) generates the k8s manifest using the script and config within ifshub-kubernetes, as well as its own state; then 2) deploys the manifest using `kubectl apply`.
    - If the triggered deployment is for the prod environment, the final step requires manual approval. Otherwise, every step in the deployment is identical regardless of deployment environment.

To understand this flow, it's important to understand the distinction between in which repo a workflow runs and in which repo the steps of said workflow are defined. The deployment flow is split into two parts: 1) the build, which happens in the development repo; and 2) the deployment, which happens in the deployment repo. However, the build workflow uses for

one of its steps an action defined in the deployment repo, whose job is to update the deployment state, and the deployment workflow uses scripts and config from the k8s repo.



5.2.2. Polytope CD prototype

The prototype CD for Polytope was more basic than for IFSHub. The goal was to prove that there are no barriers preventing us from deploying with skaffold + helm instead of kustomize. There were none. The prototype workflow can be found here: https://github.com/ecmwf/polytope-deployment/pull/3/files

5.3. Limitations of the current design

There are two key limitations of the current web app deployment prototypes built with GitHub Actions.

5.3.1. Lack of visibility on the full deployment pipeline

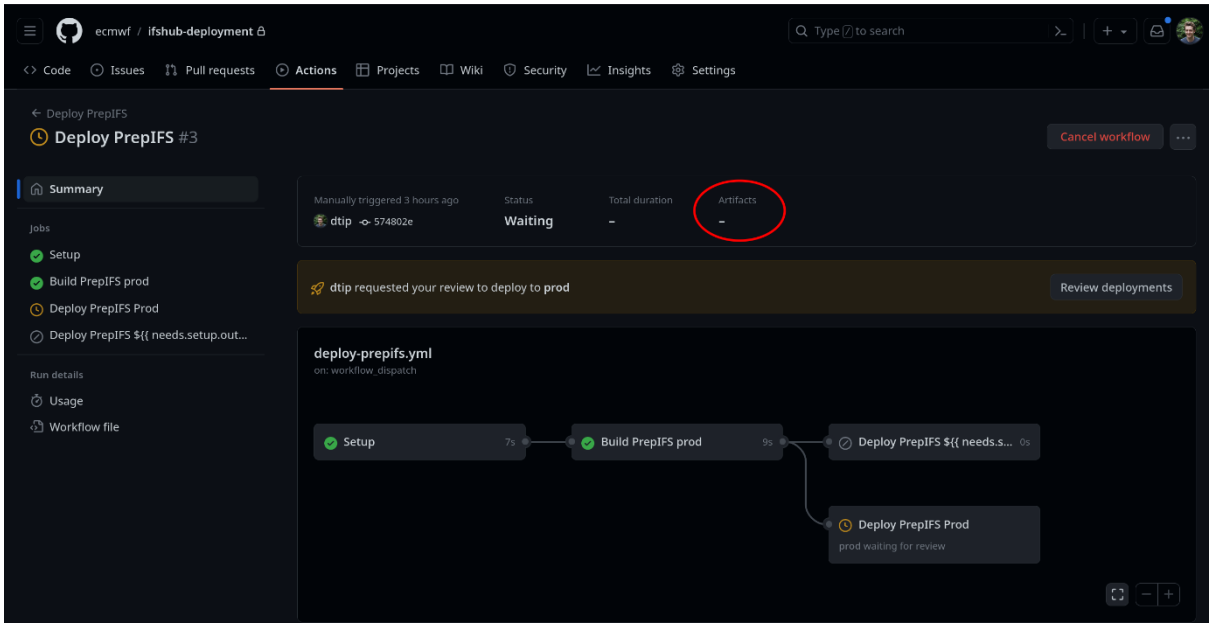
The full deployment pipeline is not visible in one place. The full pipeline is split across the development repos (e.g. ifshub-frontend, ifshub-backend, and prepifs-backend), which are responsible for building and delivering their docker images, and the deployment repo (e.g. ifshub-deployment) which is responsible for generating and applying the k8s config. This means, for IFSHub, the pipeline is split across four locations. We have the same limitation on

Bamboo.

5.3.2. Manual approval for prod deployment is not straightforward.

Requiring manual approval for the final step in the prod deployment process whilst also giving developers access to build artifacts from earlier steps in the deployment process is not straightforward with GitHub Actions. This is because of a known limitation with GitHub Actions, first reported over 3 years ago and unaddressed at time of writing, where build artifacts are only available in the GitHub UI and via the GitHub API after a workflow has completed. Therefore, if we pause a workflow at the final step to wait for manual approval, the developer who is supposed to approve the workflow can't access the build artifacts they're meant to be reviewing.

This is demonstrated by the below figure.



The figure shows the first two jobs in the workflow, "Setup" and "Build PrepIFS prod" have completed successfully. The build job creates and uploads two artifacts: the list of docker container images which will be deployed and the full k8s manifest file. The final job, which requires manual approval, is the "Deploy PrepIFS Prod" job. In order to review and approve the deployment, the developer must be able to check the two build artifacts from the build job. However, as highlighted by the red circle, these artifacts are not available because the workflow has not yet finished. This makes it impossible to review the deployment.

There are 3 possible workarounds:

1. Cancel the workflow, which will cause the artifacts to be available to download. Review the artifacts, then re-run the "failed" jobs in the workflow, which is the final deployment step. This is the simplest workaround, but constantly cancelling and re-triggering deployments is a manual overhead which may become tiresome for developers.

2. Upload the artifacts to an external storage service, such as AWS S3, so they can be reviewed by a developer. This is reasonably simple to set up, but adds maintenance overhead and cognitive overhead, because developers will have to leave GitHub to review deployments.
3. Split the workflow into two workflows, where the second workflow is triggered by the first. When the first workflow finishes, the artifacts will be available. The problem is: how do we know which artifacts were uploaded by the first workflow? The GitHub API requires an artifact ID to download artifacts, but the ID is not generated until the artifact is uploaded, so the first workflow is unable to pass it to the second. However, the first workflow can pass its workflow ID to the second. The second workflow can use the GitHub API to list the recently-uploaded artifacts, then use a combination of workflow ID + artifact name to uniquely identify the artifacts uploaded by the first workflow. This solution requires writing and maintaining custom code to get the artifacts from the GitHub API. An additional downside is it adds yet another piece to the deployment pipeline.

Ultimately, it's possible to get what we want from GitHub Actions, although it's disappointing that such basic functionality isn't natively provided.

5.4. Conclusion

GitHub Actions offers significant improvements over Bamboo for continuous deployment of web applications. There are currently a couple of limitations however we've demonstrated that it's possible to deploy both common types of web app at ECMWF: apps which use kustomize and apps which use skaffold + helm. We have demonstrated that GitHub Actions can be used to implement the preferred deployment process of ECMWF software developers, with the notable exception of the requirement for manual approval of the final step of a production deployment, which requires a workaround as described in section 5.3.2.

We believe there are no downsides to using GitHub Actions instead of Bamboo for web app CD at ECMWF, other than switching costs.

Moreover, we identified the following advantages of GitHub Actions over Bamboo:

- We have better control over which branches or tags are tested together.
- We have better control over which branches or tags are deployed to which environments.

6. Reusability of GitHub Actions workflows

Reusability of testing workflows is important so that common functionality can be shared between developers and across packages, thereby improving the speed of adoption and reducing the maintenance overhead.

There are two ways to reuse GitHub Actions workflows: "reusable workflows" and "composite actions". Generally, reusable workflows are used to quickly build basic reusability, whereas composite actions are used as building blocks for more complex or generic reusability.

GitHub Actions which have the most complex requirements can be defined as code using TypeScript. They can then be reused in the same way as a composite action. A typescript

action was created to build cmake-based ECMWF software packages as part of a prior project, which has been extended for use within downstream-ci.

Downstream CI, described in section 3.1, is a set of reusable workflows which use composite actions and a typescript action under the hood.

6.1. Reusable Workflows

You can make a standard GitHub workflow reusable to reuse existing functionality in another workflow. A reusable workflow is executed as a job in the caller workflow without the option to add other steps before or after in the context of a single job. The reusable workflow defines inputs which are the only way a caller can modify its behaviour.

Reusable workflows should be used when completely reusing the functionality of an existing workflow.

6.1.1. Advantages of Reusable Workflows

- Reusable workflows can define multiple jobs, the same as any other workflow. This makes them a good choice for creating reusable functionality which doesn't need to be extremely generic.
- Reusable workflows create rich logs in the GitHub UI, because every job and step is logged independently.

6.1.2. Limitations of Reusable Workflows

See the official GitHub documentation for the latest information:
https://docs.github.com/en/actions/using-workflows/reusing-workflows#limitations

- You can only nest reusable workflows to a depth of 4.
- You can call a maximum of 20 reusable workflows from a single workflow file.
- Environment variables aren't propagated from the caller workflow to the reusable workflow. They must be explicitly passed as inputs if you want to use them.

6.2. Composite Actions

Composite actions usually have just one function and are called in the context of a step, not as a whole job.

6.2.1. Advantages of Composite Actions

- You can call an unlimited number of composite actions within a workflow.
- Callers can set up the job environment in steps before calling the composite action and can continue working in the environment afterwards. This makes composite actions better for creating extremely generic re-usability.

6.2.2. Limitations of Composite Actions

- Composite Actions can't define multiple jobs.

- Logs are harder to explore than for reusable workflows, because the logs for the entire action are presented as the logs for a single step in the GitHub UI, even if the composite action itself contains multiple steps.
- You can only nest composite actions to a depth of 10.

6.3. TypeScript Actions

TypeScript actions are similar to composite actions in that they are called in the context of a step, not as a whole job.

6.3.1. Advantages of TypeScript Actions

- It's easier to create very complex functionality with TypeScript, using the GitHub-provided client library, rather than stringing together scripts in a YAML file as with reusable workflows and composite actions. Type checking provided by TypeScript reduces the likelihood of bugs and makes code easier to refactor.
- Callers can set up the job environment in steps before calling the composite action and can continue working in the environment afterwards, the same as with composite actions.

6.3.2. Limitations of TypeScript Actions

- TypeScript Actions are more expensive to maintain due to their complexity. If the underlying problem being solved is complex, this is a necessary cost.
- Unlike composite actions, TypeScript actions cannot call other actions.
- All the limitations of composite actions.

6.4. How to choose between a Reusable Workflow, a Composite Action, and a TypeScript Action

Reusable workflows are better for reusable functionality which doesn't need to be extremely generic or complex, or when we want to reuse a bundle of multiple jobs. For example:

- When deploying a web app to multiple environments (prod, test, dev), we want the deployment process to be identical for each environment but it needs to be configured differently. We have a reusable workflow which handles the deployment (builds docker images, runs tests, pushes docker images, generates kubernetes config, applies kubernetes config) which defines inputs for things like the names of the docker images and the kubernetes namespace in which to deploy. The caller workflow sets up the inputs for each environment, then calls the reusable workflow. You can see this in action here: https://github.com/ecmwf/ifshub-backend/blob/develop/.github/workflows/ci-cd.yml
- When creating DEBs/RPMs and delivering them to Nexus, we want to define a reusable matrix of supported platforms and deliver a binary to each of them. We use a reusable workflow so we can define a matrix and run the builds in parallel jobs: https://github.com/ecmwf-actions/reusable-workflows/blob/main/.github/workflows/create-package.yml

Composite actions are good for small units of functionality that need to be widely shared or for complex reusability. For example:

- Sending a Teams notification when a new GitHub Issue is opened. This is a small, self-contained unit of functionality which can be reused by any ECMWF repo as a step within a job, so we use a composite action: https://github.com/ecmwf-actions/notify-teams-issue
- Downstream CI is a complex system involving a huge number of jobs, which itself needs to reuse functionality from the codebases contained in the build tree. Reusable workflows aren't suitable because we hit the limitations on nesting depth and of how many reusable workflows, we can call from a single workflow file. Using composite actions lets us bypass these limits.

TypeScript actions should be used only for workflows with the most complex requirements. For example:

- Building a cmake-based package, including building dependencies as necessary, running tests, running code coverage, and uploading test artifacts: https://github.com/ecmwf-actions/build-package.

6.5. Using GitHub Actions for reusable automations beyond software testing

As mentioned, a point of frustration for Bamboo users is a lack of focused notifications. GitHub Actions is a generic automation tool, capable of being used to created automations beyond CI/CD, so we prototyped applying GitHub Actions to send ECMWF developers targeted, relevant notifications from GitHub to MS Teams. We sent notifications to Teams for 2 reasons: firstly because ECMWF developers use Teams daily as a core part of their work - it's important for notifications to live on a platform where developers already spend time; secondly because Teams channels give us a simple way to control who receives which notifications. We created a now Teams team, called "Software Packages", which contains one package per ECMWF GitHub repo. Developers can subscribe to each package for which they want to receive notifications.

We created a set of 3 GitHub Actions.

One is a workflow which notifies developers when their CI fails, then notifies again the first time the failing CI passes. Contrast this with a notification system which simply reports the status of each CI run. Most CI passes, which means developers are overloaded with notifications containing no useful information. Our custom workflow lets us share only the important information (when CI fails and when it stops failing) and has allowed us to design concise notification messages, containing all relevant information for developers to understand CI failures in a glance, with links back to GitHub for any developer who needs more detail.

The other two are workflows which notify developers when a new GitHub Issue or Pull Request is opened. This is important for open development, so that ECMWF developers are aware of code contributions and can be timely in their responses to contributions from members of the community.

6.6. Conclusion

GitHub Actions are widely reusable workflows. We have successfully demonstrated that GitHub Actions at ECMWF can be reused in different situations. We believe the reusability

of GitHub Actions makes it a good choice of tool with large advantages over Bamboo, which has poor reusability.

Recommendations:

- TypeScript actions should be reserved for only the most complex use-cases due to their development and maintenance complexity.
- Composite actions must be used for any workflows which require scalability, such as downstream-ci, due to nesting limitations of reusable workflows.
- Reusable workflows must be used for any workflow in which we want to re-use a build matrix to run jobs in parallel, as only reusable workflows can define entire jobs and their build matrices.

7. Integration of GitHub Enterprise with ECMWF Single Sign-On (SSO)

Integrating GitHub Enterprise with ECMWF SSO is necessary to increase security and avoid duplication of user administration by delegating authentication and authorisation to the ECMWF Identity Provider (IdP).

7.1. Options available for GitHub Enterprise access management

There are two options available for GitHub Enterprise access management:

1. SAML SSO: Developers use their personal GitHub accounts and link them to their ECMWF SAML identity.
2. Enterprise Managed Users: Developers are forced to use an ECMWF-created GitHub account to work on ECMWF software.

GitHub provides documentation on how to choose which is best: https://docs.github.com/en/enterprise-cloud@latest/admin/identity-and-access-management/managing-iam-for-your-enterprise/identifying-the-best-authentication-method-for-your-enterprise.

In summary, using Enterprise Managed Users imposes strong restrictions on what users can do, for example managed user accounts cannot create public repositories. They can also only contribute to repositories within the enterprise account. These two things combined make Enterprise Managed Users incompatible with open development.

7.2. Integration of GitHub Enterprise and ECMWF SSO

We connected GitHub Enterprise to Keycloak, the ECMWF IdP, for authorisation/authentication when a user tries to access ECMWF software on GitHub. This involved configuring GitHub Enterprise as a SAML SSO app in Keycloak, as with any other SAML SSO app.

The workflow for developers is:

1. A developer tries to access some private part of ECMWF GitHub. e.g. a private repo or a repo's settings.

2. GitHub redirects the user to ECMWF SSO.
3. The user signs in to ECMWF SSO
    1. If this is their first time signing in with ECMWF SSO from github.com, their ECMWF SAML identity will be linked to their GitHub account in the IdP.
    2. If the user has no ECMWF SAML identity, they are prevented from accessing private parts of the ECMWF GitHub Enterprise Account.
4. The user is now authenticated and can carry out any authorised actions on github.com.
5. Every few days, the user is prompted to re-login with ECMWF SSO to ensure they're still authenticated.

7.2.1. Using SSH keys with SSO

Developers frequently access code on GitHub using SSH (Secure SHell) keys, a more secure authentication method than username and password. The first time they link their GitHub account with ECMWF SSO, they must separately authenticate each SSH key configured on their GitHub account with ECMWF SSO if they want to be able to use that SSH key with ECMWF software.

7.2.2. Provisioning a user account

Provisioning a user account (when a developer joins ECMWF) requires a GitHub Enterprise Admin to add the user's personal GitHub account as a member of the relevant GitHub organisation(s). This can be done using the GitHub web interface or the API.

Once the user has access to one or more ECMWF GitHub organisations, the next time they try to access an ECMWF codebase, they'll be asked to sign in with ECMWF SSO. This will create a link between their GitHub account and their ECMWF SAML identity.

7.2.3. Deprovisioning a user account

Deprovisioning a user account (when a developer leaves ECMWF) involves an extra step beyond simply removing the user from ECMWF's IdP.

Once the user is removed from the IdP, they will no longer be able to sign in with ECMWF SSO and so they will no longer have access to private parts of the ECMWF GitHub Enterprise Account. However, their personal GitHub account will still show as a member of the GitHub Enterprise and will still use a GitHub Enterprise license until it is removed.

We created a script which uses the GitHub API to automatically remove GitHub accounts from the enterprise: https://github.com/ecmwf-actions/github-enterprise-utilities/blob/main/deprovision-user.sh.

This script can be run as a step in the existing ECMWF offboarding process to make sure we release our GitHub Enterprise licenses when a developer leaves.

7.3. Conclusion

We have demonstrated the feasibility of integrating GitHub Enterprise with ECMWF SSO and have been using ECMWF SSO for authentication for the best part of a year already as part of this project. There is no doubt we can use ECMWF SSO to authenticate ECMWF software developers before giving them access to ECMWF codebases.

The one caveat is that when a developer leaves ECMWF, they must be removed manually from GitHub Enterprise (using an already-created automated script) in order to free up the GitHub Enterprise license they were using.

A decision is needed on whether we should use Enterprise Managed Users or allow ECMWF developers to use their personal GitHub accounts.

8. Integration of GitHub Enterprise with Jira

Currently ECMWF uses Bitbucket, an Atlassian tool, for its git repository. Jira, ECMWF's ticketing system, is also an Atlassian tool, which means we can take advantage of tight links between Bitbucket and Jira. Specifically, when a code change is made which is related to a Jira ticket, a two-way link is automatically created between the ticket and the code change. This improves tracking, accountability, and documentation.

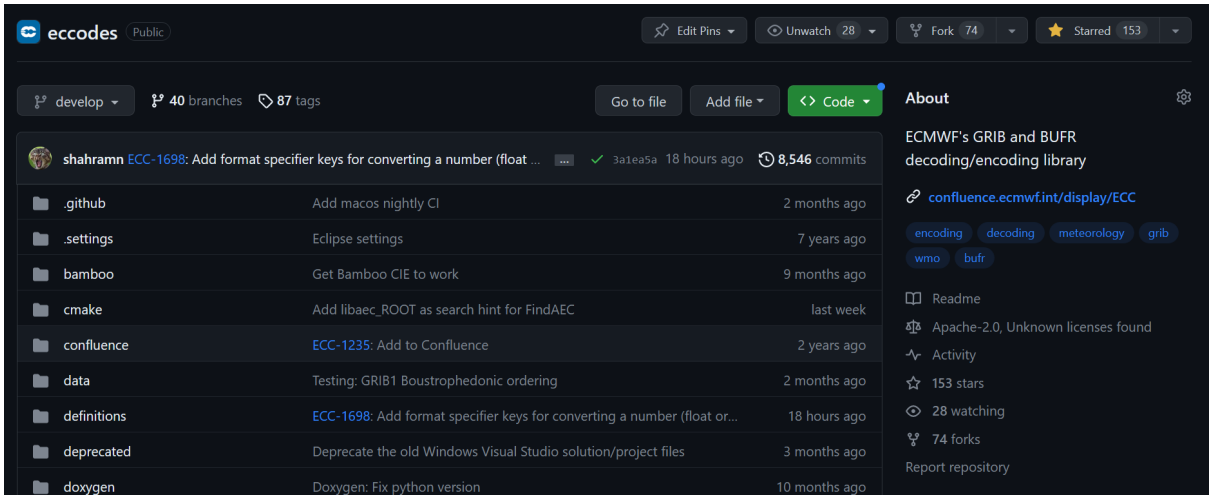
GitHub is outside the Atlassian ecosystem and so doesn't have the same features natively. We explored the options available to link GitHub to Jira in the same way and present an alternative developer ticket process which is more compatible with open development.

In this section we also explore the relationship between GitHub Issues, Jira tickets, and open development. We provide two options for ECMWF to consider in order to encourage open development without requiring a complete overhaul of existing processes.

8.1. Linking GitHub Enterprise with Jira

8.1.1. GitHub Autolinks

Autolinks are a GitHub feature which let us configure custom hyperlinks in the GitHub UI anywhere a defined pattern exists. We use this feature to enhance the GitHub UI so that anywhere a Jira ticket ID exists, it will be automatically transformed into a hyperlink to the same ticket in the ECMWF Jira. We demonstrated this functionality works as expected by setting it up for ecCodes, as shown by the below picture. Wherever Jira ticket IDs are mentioned in commit messages which appear on the GitHub UI, they have been automatically transformed into blue hyperlinks which open the relevant Jira ticket when clicked.



This is a one-way link from GitHub to Jira but is the first step towards creating a rich two-way link.

8.1.2. GitHub for Jira App

It's possible to install apps in both GitHub and Jira to enhance their functionality. Atlassian have created an app called "GitHub for Jira" which is freely available to install. The app creates a rich two-way link between GitHub and Jira, which means links between Jira tickets and code changes will be automatically created. When combined with GitHub Autolinks, the functionality between GitHub and Jira is the same as the functionality between BitBucket and Jira.

8.1.2.1. Limitations of GitHub for Jira

The official Atlassian-provided GitHub for Jira app is unfortunately only available on Atlassian Cloud, whereas ECMWF currently self-hosts the Atlassian stack.

8.1.2.2. Alternatives to GitHub for Jira

There are various third-party apps available for self-hosted Atlassian which implement the same functionality as the GitHub for Jira app. The downside is these apps are paid-for on an annual subscription. The costs are broken down in section 10.

8.2. Jira and Open Development

As discussed, the open-source software community lives primarily on GitHub. As a result, open-source developers expect to be able to contribute to open source packages using GitHub Issues, the GitHub-native ticketing system. Jira is a barrier which adds friction to the open development process.

However, at ECMWF Jira is used for far more than only software development tickets. Jira is well-ingrained at ECMWF and part of many mature processes which have been refined over a period of years. For example, the ECMWF user support and the service desk. It's untenable

to suggest a full-scale replacement of Jira with GitHub Issues for the sake of open development.

8.3. Possible Processes to Integrate GitHub Issues and Jira for Open Development

We present two process to integrate GitHub Issues with Jira tickets with the goal of encouraging open development. The first is to set up an automatic link between GitHub Issues and Jira tickets. The second is to define a partition between GitHub Issues and Jira, where only software development tickets move to GitHub Issues and everything else, including user support, stay on Jira.

In either case, when a user attempts to open a GitHub Issue, they are first prompted to visit ECMWF's Support Portal if their issue is not a software issue. We encourage developers to direct users to this portal if they open GitHub Issues which are actually user support issues.

8.3.1. Option 1: Automatic Link between GitHub Issues and Jira Tickets

We prototyped a system to automatically link GitHub Issues and Jira tickets with a one-way sync from GitHub to Jira. A new GitHub Issue creates a new Jira ticket, and any comments on the GitHub Issue are added as comments to the Jira issue. Therefore, Jira comments are private. ECMWF staff can create Jira tickets as normal for software development issues.

There is some overhead added for ECMWF developers with this system: to interact with open source contributors, they must remember to go to GitHub and comment on the GitHub Issue instead of the Jira ticket.

This system is also not perfect for open development because of the private Jira comments. The discussion should be public if we want to encourage open-source contributors to help solve the issue. On the other hand, it may be useful to have the ability to make private comments on a public issue in some situations.

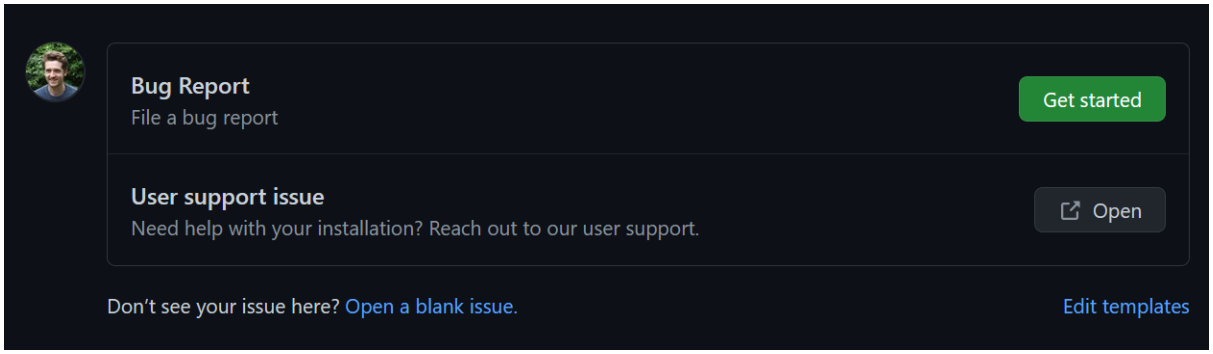
8.3.2. Option 2: Move Software Development tickets from Jira to GitHub Issues

This is the ideal solution from an open development perspective but may not be practical for the reasons outlined above: most of ECMWF continues to use Jira in the same way it currently does, but all software development tickets are opened on GitHub instead of Jira.

The link between GitHub and Jira, discussed in section 8.1, ensures that GitHub Issues can be linked to Jira tickets by, for example, commenting the name of a Jira ticket ID on a GitHub Issue.

This solution encourages open-source contributions by allowing contributors to open GitHub Issues, allowing contributors to interact with all open software development issues, and making all discussion of software development issues public. Ultimately this helps build a sense of community, rather than an "us vs them" mentality, and allows ECMWF developers to easily call on external collaborators for help by simply tagging them on GitHub.

Open-source contributors who try to open a GitHub Issue for support, rather than a bug report, will be directed to the Support Portal so that ECMWF developers aren't overwhelmed by user support requests. Note that only users with a GitHub account may open a GitHub Issue. The image below shows how this guidance looks for GitHub users.



The downside is this process would require training for ECMWF staff, both software developers and user support.

8.4. Conclusion

It's possible to get strong integration between GitHub and Jira, the same as what we have currently between Bitbucket and Jira, by making use of paid-for third party applications.

A significant decision is how ECMWF should choose to integrate GitHub Issues with Jira to maximise the benefits of open development without needing to overhaul existing processes. This is discussed is section 8.3. We recommend option 2: move software development tickets from Jira to GitHub Issues. This option is preferred as it is in the spirit of open development, although it will require training for both the service desk and software developers to come to a joint understanding that software development tickets will live on GitHub, whereas service support tickets will remain in Jira. This option has a clear separation between GitHub Issues and Jira tickets and all other existing Jira processes stay as they are. This option also avoids the complexity of needing a system to keep GitHub Issues and Jira tickets in sync.

Experience of running a limited trial with this GitHub Issues reporting system has shown that:

1. Having an Issue template leads to users providing higher-quality descriptions of their issues.

2. External users seem to understand the difference between software issues (GitHub) and service issues (Jira).

9. Disaster Recovery and Backups

When considering moving ECMWF code to be hosted on GitHub, we must consider what happens if GitHub is not reachable or responsive, and what we need to do if that happens in order for ECMWF to meet its operational requirements. Please note ECMWF operational systems depend indirectly on the version control service for deployment, configuration, and recovery activities.

In this section we analyse recent uptime of GitHub and ECMWF-hosted Bitbucket, and present 5 options for redundancy and backups.

9.1. Analysis of recent GitHub uptime

GitHub has been going down frequently so far in 2023. See https://www.githubstatus.com/history. It was particularly bad in April and May, with multiple incidents where pushing/pulling code was broken and where GitHub Actions were broken.

There were multiple issues with pushing/pulling code and GitHub Actions in June-September as well, including one major outage in June (primarily in North America), though more issues occurred outside of European normal working hours and so were lower impact for ECMWF developers.

Most issues are minor and don't affect the development cycle. However, a handful of issues in 2023, particularly in April and May, left GitHub in a broken-enough state that pushing and testing new code contributions was impossible.

Most GitHub issues are typically fixed by the GitHub team within an hour.

9.2. Option 1: Self-host GitHub Enterprise at ECMWF

We could set up GitHub Enterprise Server on ECMWF infrastructure. However, GitHub Enterprise Server is designed for private instances and so is not compatible with open development out of the box.

More information on how we'd need to configure self-hosted GitHub Enterprise for open development can be found here: Investigation: approaches to redundancy and backups for repos hosted on GitHub

Advantages

- We have full control over our hosting setup and disaster recovery plan. This is the most flexible option.
- We won't need to send our private code to Microsoft (Microsoft owns GitHub).

Disadvantages

- Open development is much more complicated with GitHub Enterprise Server
  - Our repositories will not appear on GitHub.com if we self-host. They'll appear on some domain which we control.
  - There's a feature called "GitHub Connect" which allows our server to access some features from github.com
    - One of these features is "unified search"
    - "unified search" allows us to include results from github.com public repos in the searches performed on our server. However, it doesn't work the other way round.

        - *"Users will never be able to search your GitHub Enterprise Server instance from GitHub.com, even if they have access to both environments."*
        - source: https://docs.github.com/en/enterprise-server@3.9/admin/configuration/configuring-github-connect/enabling-unified-search-for-your-enterprise#about-unified-search
    - The only solution would be to use GitHub Cloud and GitHub Enterprise Server and set up mirroring for our open source repos.
        - We would either have to have two-way syncronisation OR it would be one-way from github.com → our server and ECMWF developers would have to remember to make changes to public repos from github.com and private repos from our server.
        - Either option creates complexity and adds overhead.
- Significant cost (time and money) for initial setup and maintenance.

9.3. Option 2: Use GitHub Enterprise Cloud and Bitbucket as redundant read-only service.

We could use GitHub Enterprise Cloud as the primary source of our repositories, but set up automated synchronisation with Bitbucket (or another repository store) so we can still access our code when GitHub goes down.

Advantages

- We already use Bitbucket at ECMWF so there is minimal cost involved with setup and training.
- We already have GitHub Actions in place which let us synchronise repositories from GitHub to Bitbucket. The sync is one-way, so complexity is minimal.
- Our repositories would keep their primary home on GitHub.com, which is where open-source contributors expect them to be.
- We already have backups set up for Bitbucket: daily backups from the FS and database, plus the VM is backed up with Veem so we could recover it in theory. We've not specifically attempted a backup recovery in the past, but each time we upgrade or move Bitbucket from one system to another, the process is essentially the same as what would be done in a disaster recovery, so we can be confident that our processes work.
- Our Bitbucket is self-hosted which gives us extra control and protects us from Atlassian Cloud downtime.

Disadvantages

- If GitHub goes down and we make code changes on Bitbucket, they'd need to be manually pushed to GitHub once it's back online.
- Concern that if we go to Atlassian cloud, the problem may resurface.

9.4. Option 3: Use GitHub Enterprise Cloud and continuous backups.

We could use GitHub Enterprise Cloud and take continuous backups by automatically syncing GitHub with a self-hosted backup Git server.

There are some existing tools for backups (e.g. https://github.com/marketplace/backhub) but these take and restore backups to and from GitHub, so they will not work if GitHub is down. We'd likely need a custom solution.

The exact choice of Git server doesn't matter - we could use GitHub Enterprise self-hosted or a custom Git server.

Advantages

- Backups are essential for data loss. GitHub will have their own backups and disaster recovery in place, but it would be a good idea for us to back up our own repositories.
- We can get rid of Bitbucket entirely.
- Our repositories would keep their primary home on GitHub.com, which is where open-source contributors expect them to be.

Disadvantages

- We'd need to manually make sure any changes are pushed to GitHub once GitHub comes back online.

9.5. Option 4: Trust GitHub Enterprise Cloud

We could trust GitHub cloud and not set up any redundancy or backups.

Advantages

- This is the cheapest option: we don't have to spend time or money setting up or maintaining backups.

Disadvantages

- GitHub has been going down frequently so far in 2023 as outlined in section 9.1. If we trust them, we may not be able to access our source code in time to meet ECMWF's operational requirements.

9.6. Conclusion

GitHub has had some downtime issues over the past year and so some form of redundancy and backups is essential for ECMWF to meet its operational requirements.

In the transition phase we recommend option 2: use GitHub Enterprise cloud and Bitbucket as a redundant read-only service.

In the long term we recommend option 3: use GitHub Enterprise cloud with self-hosted Git server only for backup & fallback purposes.

10. Expected Costs

In this section we describe two cost scenarios:

1. We adopt GitHub Enterprise for the Forecast Department only and support mainly software services, which would entail providing licences for Forecast Department developers and collaborators. We estimate 270 licences which are 150 for the Forecast Department developers, 100 for DestinE, and 20 for CCI (Common Cloud Infrastructure) developers. Notably this excludes IFS researchers and developers.
2. We adopt GitHub Enterprise for all developments at ECMWF including IFS developers and provision for some external collaborators which require access to IFS. We estimate 500 licences of which 150 are for the Forecast Department, 100 for DestinE, 20 for CCI, and 230 for IFS.

10.1. Cost of GitHub Enterprise seats

GitHub Enterprise costs $19.25 per user per month.

In scenario 1, with 270 licences, GitHub offer an 11.5% discount on official prices.

In scenario 2, with 500 licences, GitHub offer a 15% discount on official prices.

Forecast annual cost:

Scenario 1 (270 licences)	Scenario 2 (500 licences)
$55,197.45	$98,175.00

10.2. Cost of CI runners

10.2.1. GitHub-hosted cloud runners

Included with GitHub Enterprise, we're given 50,000 GitHub Actions minutes per month on GitHub's cloud-hosted runners for private repos. Public repos have unlimited free GitHub Actions minutes for GitHub's cloud-hosted runners.

We have used an average of 1,600 minutes per month over the past 90 days, just over 3% of our quota. We do not anticipate needing to pay for extra GitHub-hosted runner minutes in the near- to medium-term.

Forecast annual cost: $0.

10.2.2. MacStadium-hosted runners

Github does not have a wide support for runners using the MacOS system that powers the ECMWF laptops mostly used by our researchers and analysts. We compensate that by using a separate hosting service, MacStadium, where we run our CI/CD pipelines.

We currently lease:

- 2x Mac Mini with Arm64 architecture, 8 core M2 CPU, 16GB RAM, 1TB SSD, 10G Ethernet => $209 + VAT each per month.

- 2x Mac Mini with intel architecture, 6 core i7 CPU, 32GB RAM, 512GB SSD, 10G Ethernet => $169 + VAT each per month.

for a total cost of $756 + VAT per month.

MacStadium offer an annual discount of 8%.

Annual cost: $8,346.24

10.3. Cost of GitHub Copilot

GitHub Copilot is a feature which uses AI to automatically generate code. Currently it uses a version of OpenAI GPT-4 optimised for software code generation and interpretation. GitHub Copilot is currently being tested by a small number of developers at ECMWF. The evaluation has shown remarkable capabilities and robust code generation. We believe this may significantly speed up productivity of developers by automating some of the tedious and repetitive parts of software development.

If we decide to proceed with GitHub Copilot, the cost is $19.00 per user per month.

Forecast annual cost for 120 licenses is $27,360.

Scenario 1 (60 licences)	Scenario 2 (120 licences)
$13,680	$27,360

10.4. GitHub for Jira App

As mentioned in section 8.1.2, we must pay for a third-party app to integrate GitHub and Jira because the free Atlassian-provided GitHub for Jira app is only available on Atlassian Cloud.

The options are:

- Git Integration for Jira, developed by GitKraken.
  - 3.6/4 with 324 ratings and 11k installs.
  - Works on Jira Server and Jira Data Center.
  - Annual costs: $1,025 up to 50 users, $1,775 up to 100 users, $3,075 up to 250 users, $4,415 up to 500 users.
  - Atlassian marketplace listing: https://marketplace.atlassian.com/apps/4984/git-integration-for-jira-github-gitlab-and-more?tab=pricing&hosting=datacenter

- Jigit, developed by Move Work Forward.
  - 4/4 with 25 rankings and 939 installs.
  - Works on Jira Server and Jira Data Center.
  - Annual costs: $50 up to 50 users, $400 up to 100 users, $1,000 up to 250 users, $2,000 up to 500 users.
  - Atlassian marketplace listing: https://marketplace.atlassian.com/apps/1217129/jigit-jira-github-gitlab-integration?tab=pricing&hosting=datacenter
- Exalate, developed by Exalata.
  - 3.3/4 with 106 rankings and 4.2k installs
  - Works on Jira Server and Jira Data Center.

- Annual costs: $2,300 up to 50 users, $2,645 up to 100 users, $3,965 up to 250 users, $4,600 up to 500 users.
- Atlassian marketplace listing: https://marketplace.atlassian.com/apps/1213645/exalate-jira-issue-sync-more?tab=pricing&hosting=datacenter

We recommend "Git Integration for Jira", despite it being significantly more expensive than Jigit, because it is the application and is the only application which is SOC2 certified. SOC2 is a security framework that specifies how organisations should protect customer data from unauthorised access, security incidents, and other vulnerabilities.

Forecast annual cost:

Scenario 1 (270 licences)	Scenario 2 (500 licences)
$4,415	$4,415

10.5. Conclusion

The expected annual cost for GitHub Enterprise and GitHub Actions is $81,638.69 for scenario 1 and $138,295.24 for scenario 2.

Costs are broken down as follows:

Service	Annual Cost Scenario 1	Annual Cost Scenario 2
GitHub Enterprise seats	$55,197.45	$98,175.00
GitHub-hosted cloud runners	$0.00	$0.00
MacStadium-hosted runners	$8,346.24	$8,346.24
GitHub Copilot	$13,680.00	$27,360.00
GitHub for Jira App	$4,415.00	$4,415.00
TOTAL	**$81,638.69**	**$138,295.24**

We believe this cost represents a good price for the value which ECMWF receives in exchange.

The minimum recommendation is scenario 1 to allow for a successful implementation of open development.

If budget allows, we further recommend scenario 2, where all ECMWF would adopt GitHub Enterprise for all developments including IFS. Based on preliminary conversations with the IFS Section, we believe they will likely benefit from migrating from Bamboo to GitHub Actions, similar to the benefits we have demonstrated in this report for the Development Section.

It is possible to start with scenario 1 and move to scenario 2 at a later date.

11. Recommendations

11.1. Main Recommendations

We recommend using GitHub Enterprise at ECMWF to allow a successful implementation of open development.

- The minimum recommendation is to use GitHub Enterprise for the purposes of Open Development only, meaning adoption of scenario 1, at the estimated cost of ~82k USD/year.
- However, given the benefits highlighted in this report, if budget allows, we further recommend all ECMWF adopting GitHub Enterprise for all developments including IFS, at the cost of ~138k USD/year.

11.2. Detailed Technical Recommendations

The below recommendations are summarised from the conclusions of each section of this report, mainly related to details for the technical implementation of the main scenario 1 or 2 recommendation.

- We recommend using self-hosted GitHub Actions runners for CI/CD with GitHub Actions at ECMWF in addition to GitHub-hosted runners as appropriate for each software project.
- Recommendations for Continuous Integration, Continuous Delivery, and Continuous Deployment:
    - ECMWF should use the Downstream CI strategy to automatically test downstream software packages on every code change to upstream packages in order to catch software issues early in the development cycle.
    - ECMWF should run long-running CI nightly, instead of on every code change, to allow fast iterations of the development cycle whilst still getting the advantages of frequent automated testing.
    - ECMWF should use GitHub Actions for automated quality control of its codebases.
    - ECMWF can use GitHub Actions to securely build and test software which depends on restricted-access software packages such as mars-hpss, but due care should be taken to ensure any newly created GitHub Actions workflows clean up the restricted-access software package when it is finished. We recommend multiple experts review any new workflow using restricted-access software to ensure its usage is secure. We estimate this special case applies to a tiny fraction of ECMWF CI workflows.
    - ECMWF should use GitHub Actions for all continuous delivery and continuous deployment needs.
- Recommendations on reusing GitHub Actions:
    - TypeScript actions should be reserved for only the most complex use-cases due to their development and maintenance complexity.
    - Composite actions must be used for any workflows which require scalability, such as downstream-ci, due to nesting limitations of reusable workflows.
    - Reusable workflows must be used for any workflow in which we want to re-use a build matrix to run jobs in parallel, as only reusable workflows can define entire jobs and their build matrices.
- We recommend integrating GitHub Enterprise with ECMWF SSO.
- We recommend integrating GitHub Enterprise with Jira using a third-party application.
- We recommend moving software development tickets from Jira to GitHub Issues.
- In the transition phase, we recommend using GitHub Enterprise cloud and Bitbucket as a redundant read-only service.

- In the long term, we recommend using GitHub Enterprise cloud with self-hosted Git server only for backup & fallback purposes.

11.3. Further decisions required.

- Section 7.3: A decision is needed on whether we should use Enterprise Managed Users or allow ECMWF developers to use their personal GitHub accounts.
- Section 8.3: A decision is needed on which working process option to use to integrate Jira and GitHub Enterprise.
- Section 8.4: A decision is needed on which third party application to use to integrate Jira and GitHub Enterprise. Details on options are presented in section 10.4.
- Section 9.6: A decision is needed on which disaster recovery option to implement.
- Section 10.5: A decision is needed on whether the whole of ECMWF, including the IFS Section, should adopt GitHub Enterprise for all developments.

12. Annex 1: Glossary of Terms

- Git: A software version control application which is the de-facto global standard, and which is used at ECMWF.
- GitHub: A web application which is used to store and interact with codebases version controlled by Git. Github.com is the de-facto home of open-source software development on the internet and is owned by Microsoft.
- GitHub Actions: A GitHub feature which allows developers to run automated scripts on a codebase based on various triggers. e.g. we can run automated tests whenever the code changes.
- BitBucket: A web application which is used to store and interact with codebases version controlled by Git and is owned by Atlassian. ECMWF currently uses a self-hosted BitBucket instance to host our Git repos.
- Bamboo: The BitBucket equivalent of GitHub Actions, currently used at ECMWF, and is owned by Atlassian.
- repo (repository): A codebase. In this document it should be assumed that "repo" refers specifically to a codebase version controlled by Git.
- workflow: GitHub Actions terminology for a file which defines one or more automated processes. e.g. automated software testing, automated quality control, or automated deployment.
- job: A workflow is broken down into jobs. Jobs automatically run in parallel if they're independent or run in series if they depend on each other.
- step: A job is broken down into steps. Steps are used to e.g. configure environment variables before running tests.
- build plan: Bamboo terminology equivalent to a GitHub Actions workflow.
- Continuous Integration (CI): the process of automatically running software tests before a code change is accepted into the codebase to check the change functions as expected.
- Continuous Delivery (CD): the process of automatically delivering library code or compiled binaries to a store or package manager so they can be shared with users and other developers. "CD" is typically used as a catch-all term to cover both Continuous Delivery and Continuous Deployment.
- Continuous Deployment (CD): the process of automatically deploying a software application, typically in the context of a web application being automatically deployed to the cloud. "CD" is typically used as a catch-all term to cover both Continuous Delivery and Continuous Deployment.

13. Annex 2: Training required to make effective use of GitHub Enterprise

Training is split into two categories:

1. Technical training teaches developers how to effectively use the tooling available on GitHub as part of their software development work.
2. Cultural training is about encouraging a culture of open development at ECMWF, our Member States, and the wider open-source community. Both are essential for ECMWF to get the most benefit from open development with GitHub Enterprise.

The topics necessary for technical and cultural training are already documented extensively on Confluence as part of the project which generated this report: Open Development on Github

13.1. 13.1. Technical training

For developers:

- Downstream CI.
  - Developers must understand the concepts behind downstream-ci.
  - Developers must know how to extend downstream-ci to add more ECMWF packages to the dependency tree.
  - We envision a half-day workshop to cover the relevant materials, supported by documentation on Confluence.
- Self-hosted runners & public PR approval flow.
  - All developers must understand the security concerns when using self-hosted runners outlined in section 2.8.
  - All developers must understand the label-based public PR approval flow outlined in section 2.8.2.
  - Developers selected to maintain our self-hosted runners must understand how to use and extend our automated provisioning & configuration scripts.
  - We envision a 1-hour workshop supported by documentation on Confluence.
- Using GitHub.
  - Developers should know how to reference a GitHub Issue from a commit, PR, or another Issue.
  - Developers must know how to use ECMWF SSO to access GitHub.
  - Developers must know how to authorise their SSH keys and tokens for use with ECMWF SSO.
  - Developers should know how to configure GitHub Autolinks
  - Training can be provided during onboarding and supported by documentation on Confluence.

For developers and user support, if we choose to move software development tickets to GitHub Issues:

- All parties must understand the boundary between Jira tickets and GitHub Issues.
- All parties should understand the links created between Jira tickets and GitHub (code changes, PRs, and Issues). The links are similar to the links created between Jira and Bitbucket so training on this point should be minimal.

- We envision a 2 hour workshop supported by documentation on Confluence.

For GitHub admins:

- Using GitHub.
  - Admins should know how to manage access to ECMWF GitHub repos and the difference in responsibilities between the "read", "write", "maintain", and "admin" roles.
  - Admins should know how to manage GitHub Actions secrets.
  - Admins should know how to work with GitHub personal access tokens.
  - Admins should be familiar with GitHub service accounts and their use-cases at ECMWF.
  - Training can be provided through in-person conversation and supported by documentation on Confluence.
- Onboarding new developers.
  - The onboarding process for GitHub Enterprise is outlined in section 12.3.
  - Training can be provided through documentation on Confluence.

13.2. 13.2. Cultural training

For developers:

- Developers should understand the benefits of open development, both for themselves and for ECMWF and its Member States.
- Developers should understand the importance of being responsive on GitHub Issues and PRs to encourage open-source contributions.
- Developers should be encouraged to direct GitHub Issues users to user support and close any irrelevant Issues.
- We envisage a 1 to 2 hours workshop supported by documentation on Confluence. Cultural training often takes longer to sink in than technical training, so team leads may need to reinforce and repeat the concepts over a period of weeks or months to encourage the shift towards open development.

13.3. 13.3. Extra steps during onboarding

We recommend taking some extra steps during the onboarding of a new ECMWF developer. These steps will ensure consistent standards when using GitHub and should reduce the burden on GitHub Enterprise admins.

For developers:

- Configure your GitHub account so it has your full name and a recognisable picture of you. This makes it easier for ECMWF teammates and external collaborators to identify you.
- Share your GitHub username with a GitHub Enterprise Admin.

For GitHub Enterprise admins:

- Get the new developer's GitHub username.
- Add the new developer to the correct GitHub organisations and teams.
- Configure the new developer to be maintainer of any relevant repositories.

13.4. 13.4. Extra steps during offboarding

For GitHub Enterprise admins:

- Run the offboarding script to automatically remove the departing developer's GitHub account from the enterprise to free up their license: https://github.com/ecmwf-actions/github-enterprise-utilities/blob/main/deprovision-user.sh

13.5. 13.5. Conclusion

GitHub is already a familiar environment for most software developers at ECMWF, and so necessary training is on the scale of hours to days rather than days to weeks.

We recommend training workshops for the more complex and detailed topics. Training should be supported by extensive documentation, which already exists on Confluence.