Architect Stack Best Practices:

- Projects
- Automations/Actions
- Containerization/Codespaces
- Security Lab

Discussion:

The starting point for context for each part of the challenge we're exploring within the OSS Architecture portion of this hackathon. Apply these best practices to those discussion thread and linked issues.

PM: Projects or Memex

A project is a customizable spreadsheet that integrates with your issues and pull requests on GitHub. You can customize the layout by filtering, sorting, and grouping your issues and PRs. You can also add custom fields to track metadata. Projects are flexible so that your team can work in the way that is best for them

Project Boards:

Hackathon Overall: Track progress across the goals of Amplify across skillsets
Architecture: Track progress across OSS Architecture Epics (big picture milestones), subtasks, and status

Status:

- New: needs to be triaged or has another issue blocking you to start
- Todo: can be worked on
- In Progress: Someone has assigned the issue too themselves/been assigned
- Help wanted: Someone assigned the issue but may be stuck
- Review: You've opened up a PR, depending on the result it will either get closed or move back to 'In Progress'

Tutorials:

- Getting Started
- Customizing Views
- Automation for workflows
- Best Practices

Automations: Actions & Workflows

Actions

Context:

This infrastructure automation tackles the common challenge of keeping developers' work on pace and not running ahead of operations capabilities, which is directly related to the ability to deploy software. Automation serves up another key feature, namely the creation of more broadly available selfservice in later stages of development, which of course leads to greater efficiency. Using an array of templates, GitHub allows users to automatically configure workflows in order to keep the status of project board cards in sync with associated issues and pull cards. Users can automate actions based on triggering events to eliminate some time consuming manual tasks in managing a project board. And users can copy a GitHub project board to reuse its customization for similar projects down the road.

Tutorials:

- How to build a CI/CD Pipeline with Actions

Actions Best Practices

- [ ] Unless an Action contains something really sensitive, consider making it public, so you can reuse it in all the repositories. Use GitHub Apps to use private actions in the organization.
- [ ] Create Javascript/Typescript actions when possible as they run faster than Docker actions. The latest `node` version supported on GitHub-hosted runners can be found by checking the details for each virtual environment
- [ ] Debug your actions locally using act
- [ ] Create tests for your actions decoupled from the GitHub actions framework using a CLI approach and the command pattern
- [ ] Create a meaningful `README` file with at least the following elements:
    - Description of the action
    - How to run it locally
    - What kind of token or permissions it needs to run

- What are the parameters that it accepts
- What are the outputs that it produces
- Contributing guidelines for internal or open source collaboration
- Which team or individuals are maintainers

Containerization/Codespaces:

Codespaces allows for customization on a per-project and per-branch basis with a `devcontainer.json` file. This configuration file determines the environment of every new codespace anyone creates for your repository by defining a development container that can include frameworks, tools, extensions, and port forwarding. A Dockerfile can also be used alongside the `devcontainer.json` file in the `.devcontainer` folder to define everything required to create a container image.

devcontainer.json

This file can be located in the root of the repository or in a folder called `.devcontainer`. If the file is located in the root of the repository, the filename must begin with a period: `.devcontainer.json`.

You can use your `devcontainer.json` to set default settings for the entire codespace environment, including the editor, but you can also set editor-specific settings for individual workspaces in a codespace in a file named `.vscode/settings.json`.

For information about the settings and properties that you can set in a `devcontainer.json`, see devcontainer.json reference in the Visual Studio Code documentation.

Dockerfile

A Dockerfile also lives in the `.devcontainer` folder.

You can add a Dockerfile to your project to define a container image and install software. In the Dockerfile, you can use `FROM` to specify the container image.

```
FROM mcr.microsoft.com/vscode/devcontainers/javascript-node:0-14

# ** [Optional] Uncomment this section to install additional packages. **
# USER root
#
# RUN apt-get update && export DEBIAN_FRONTEND=noninteractive \
#     && apt-get -y install --no-install-recommends <your-package-list-here>
#
# USER codespace
```

You can use the `RUN` instruction to install any software and `&&` to join commands.

Reference your Dockerfile in your `devcontainer.json` file by using the `dockerfile` property.

```
{
  ...
  "build": { "dockerfile": "Dockerfile" },
  ...
}
```

For more information on using a Dockerfile in a dev container, see Create a development container in the Visual Studio Code documentation.

Use Cases:

- It allows developers to code from the fastest machines. This ultimately increases the developer's efficiency and productivity, as the burden of local machine setup is removed from the process.
- Cloud development platforms are highly secured. Only authorized users can access GitHub Codespaces.
- Developers have the freedom to use different tech stacks, a variety of tools, and different versions of software without having to adjust the configuration of their local machines.
- Teams can access Codespaces from any device with browser-based coding. The experience is seamless across laptops, desktops, tablets, and more—improving portability.
- It's easy for developers to manage extensions and dependencies across multiple Codespaces and on a per-project basis alike.
- Codespaces runs consistently for every developer. So your team won't run into issues of the development environment working for one developer but not another.
- GitHub Codespaces runs directly in GitHub, in a matter of seconds. So any developer can start coding and collaborate on a project almost instantaneously.

How to implement:

Reference back-end
Configuration File (theme their codespace, starting linters etc)
Docker Dev Containerrs

- Container Registry within Actions workflow
- Docker images

AppSec: GitHub Security Lab

- CodeQL
- Token Scanning/Secrets
- Security Policy/Advisory

- Dependabot

CodeQL

Context:

GitHub experts, security researchers, and community contributors write and maintain the default CodeQL queries used for code scanning. The queries are regularly updated to improve analysis and reduce any false positive results. The queries are open source, so you can view and contribute to the queries in the `github/codeql` repository. For more information, see CodeQL on the GitHub Security Lab website. You can also write your own queries. For more information, see "About CodeQL queries" in the CodeQL documentation.

Enabling code scanning with CodeQL

Code scanning automatically scans your GitHub repository for vulnerabilities and errors. It's packaged as a GitHub Actions workflow and is very easy to enable for your repository. The configuration is stored in a YAML file in your .github subdirectory. In its default configuration, code scanning runs a suite of security checks implemented in CodeQL, which is a query language that lets you query code as though it were data. You can also use CodeQL to develop your own custom checks. The CodeQL queries used by code scanning are open source, so there are plenty of examples to learn from if you want to develop your own query.

By default, code scanning only runs a curated suite of the most accurate CodeQL queries, to avoid overwhelming developers with large numbers of alerts. The CodeQL repository contains many more queries that are not enabled by default. It's quite likely that there are queries that are not enabled that would find additional useful results on your project. You can add more queries by editing your code scanning configuration. We have done that in Exiv2 by adding this line to our YAML file:

```
config-file: .github/codeql/codeql-config.yml
```

The linked YAML file, `codeql-config.yml`, specifies the queries that we want to run.

Adding custom CodeQL queries

In the Exiv2 project, we have added some custom CodeQL queries, which will automatically detect variants of bugs that have caused security problems in the past. In this blog post, I am going to take a deep dive into the most sophisticated query that we have added so far: a query to detect unsafe uses of `std::vector::operator[]`. Although I have chosen the most complicated query as my main example, I don't want to give you the impression that CodeQL is always this hard. For example, one of the other custom queries is a very simple query for detecting signed shifts. Yet I think `unsafe_vector_access.ql` is interesting, primarily because it uses a few advanced CodeQL features, and secondly because it will allow me to highlight some of the trade-offs that are possible when you're designing a query for a single codebase, rather than trying to design a general purpose query.

Tutorials:

Adding Custom CodeQL

Best Practices You can run additional queries as part of your code scanning analysis. These queries must belong to a published CodeQL query pack (beta) or a QL pack in a repository. CodeQL packs (beta) provide the following benefits over traditional QL packs:

- When a CodeQL query pack (beta) is published to the GitHub Container registry, all the transitive dependencies required by the queries and a compilation cache are included in the package. This improves performance and ensures that running the queries in the pack gives identical results every time until you upgrade to a new version of the pack or the CLI.
- QL packs do not include transitive dependencies, so queries in the pack can depend only on the standard libraries (that is, the libraries referenced by an `import LANGUAGE` statement in your query), or libraries in the same QL pack as the query.

Token Scanning/ Secrets

Use encrypted secrets to store your deploy keys:

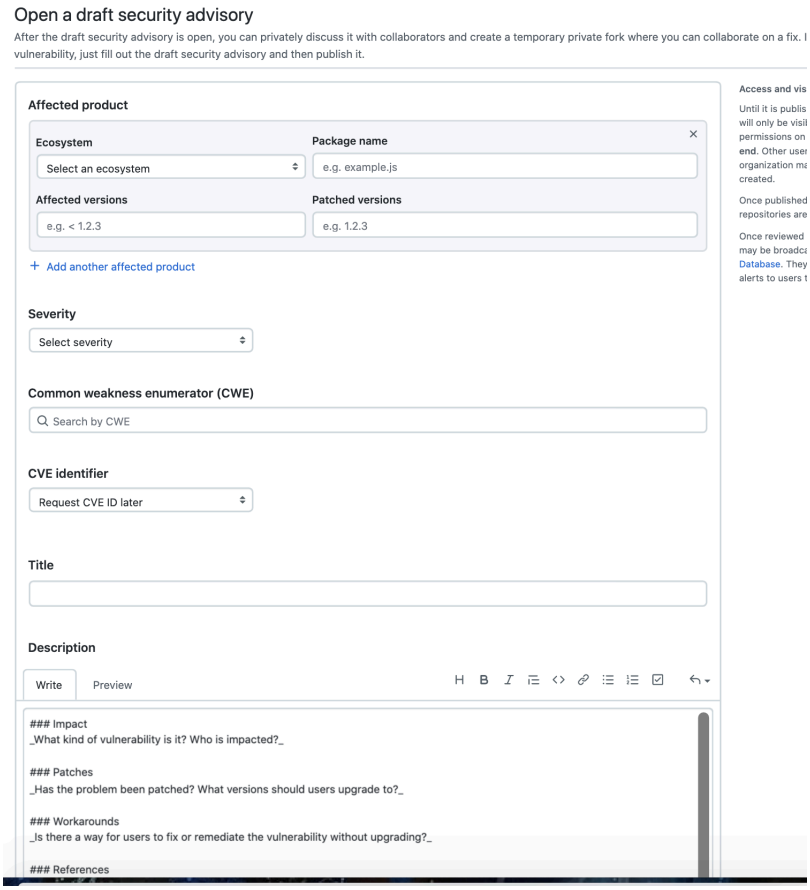
- For long secrets, store the encrypted file in the repository and keep the key as a secret
- Never add Personal Access Tokens (PAT), private certificate keys or deployment secrets to the repository without encryption. The history will remember them. If you add them by accident, revoke them and create new keys.

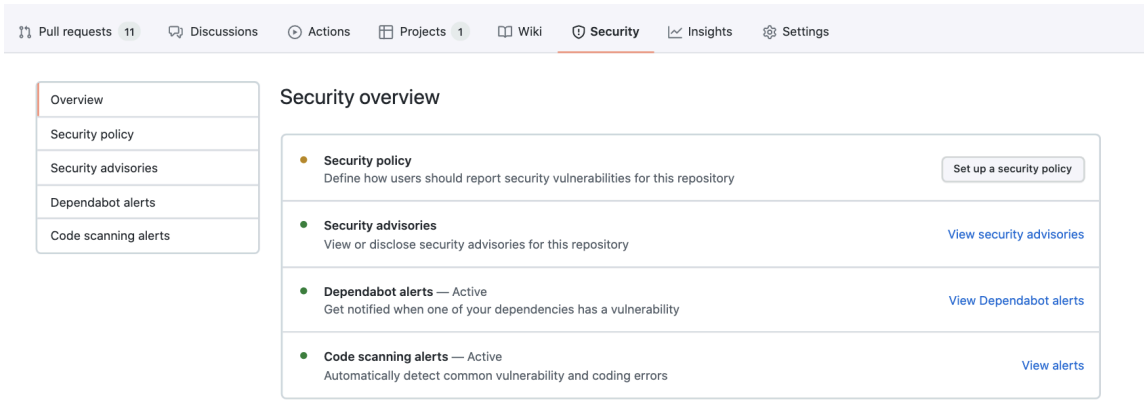
- Consider using a actions pinning in your workflows (or a tag if you trust the source; example) . Using master can be risky if it is not a stable version and might change the behavior without you noticing it. Read this article for more information.
- If you want additional security and policies around actions, add the setting in the org to allow only specific actions.
- Set the default workflow permissions to `contents:read` and require the `GITHUB_TOKEN` permissions to be set at the workflow level
- Use Actions environments in the workflows to manage your secrets. Secrets in environments can only be accessed while deploying to the environment.
- Share secrets at the organization level manage centrally the secrets of your organization

Security Policy and Advisory: how to report and disclose vulnerabilities

Security Lab brings together researchers and the open source ecosystem to amplicy discoveries and dedicate the community

Open up Security Lab | Creating a security advisory process





Dependabot: scan dependencies for updates around vulnerabilities