DevOps - Final Project Report

Christopher Elchik
Jake McDavitt
12/2/2025

Problem Statement + Motivation

We have an app called *Goober Detector*, an AI-enhanced SaaS application. Our problem is that for every code change, we originally had to manually build, test, and deploy the app, which was a lengthy process as updates were often needed. Additionally, we wanted other developers and product managers to be able to preview changes to test/give feedback, and our original method was for reviewers to manually check out the branch with these changes and build the app themselves, which was slow. We needed a streamlined approach to automate the process of building, testing and deploying, as well as facilitating code reviews for PR's.

Our solution was therefore to create/integrate an automated DevOps pipeline that automatically tests, lints, builds, and deploys our app as changes are made. One of the cornerstones of our pipeline will be that it will automatically deploy preview app instances upon each PR, so any developers assigned with reviewing the PR can visit the preview app instance for sanity checking rather than running the entire system locally. The next cornerstone will be the integration of Sonarqube, a code quality analysis tool, which will streamline quality checks to alleviate this task from developers/reviewers. Lastly, our solution integrates multiple security analysis tools, including dependency scanners as well as security gates from Sonarqube.

Our pipeline is event-driven (using pull requests) and not necessarily user-facing. Integrating this pipeline leaves a positive impact on our developers, who can now save time as our automated pipeline accelerates tasks that would otherwise be done manually. This positive impact would cascade onto our userbase, who would experience more frequent changes and thus be "comforted with discomfort" (Top 10 Adages slides).

Tagline from Proposal: *Our pipelines don't clog. They commit.*

Summarized Pipeline Features

- Self Hosted GitHub Actions Runner

Our pipeline uses a Self Hosted GitHub Actions Runner hosted off a durable Ubuntu 22.04LTS Image host from the NC State Virtual Computing Lab service. This runner runs all of our actions triggered from GitHub.

- Automated Unit Tests

Our pipeline automatically runs both backend and frontend unit tests, using pytest for backend and jest for frontend. The workflow fails if any test fails. This runs on our self-hosted GitHub Actions VCL instance.

- Automated Linting

Our pipeline runs a linter (pylint) that fails the workflow if the linter score is under 70%. This runs on our self-hosted GitHub Actions VCL instance.

- Dependency Scanners (security quality gate)

Our pipeline runs dependency scanners (pip-audit for backend, npm-audit for frontend), which scans all packages (and their dependencies) for known security vulnerabilities. The workflow will fail if any vulnerabilities are detected in the backend or frontend packages. This also runs on our self-hosted GitHub Actions VCL instance.

- Sonarqube Quality Gates

Hosted off a durable VCL we run an instance of Sonarqube Community Edition running off a Docker container with a postgres database it connects to so it can persist data. It's set up with a GitHub app installed on an organization where the repo exists, for users within the organization they are able to authenticate with their GitHub account to access the Sonarqube service. Each time a PR is made to a release branch, our actions script triggers 2 actions provided by the Sonarqube service. First the SonarSource/sonarqube-scan-action@v6, which uploads the last commit in the PR to the hosted Sonarqube service where it will run its analysis and generate issues and security hotspots it finds with the codebase. The user can access the service directly and look through what issues and security hotspots exist in the code and how to fix them, but it also provides the ability to set up a Quality Gate, which allows users to specify what constraints within the codebase can't be broken for a build to be considered successful. Then the script uses the second Sonarqube action, SonarSource/sonarqube-quality-gate-action@v1, where it will automatically fail the action if the Quality Gate returned is a failure.
The purpose of the Sonarqube service is that it provides issues and security hotspots that are more technically in depth than standard linters or static code analysis tools while providing suggestions for how to fix them & it provides Quality Gates where DevOps engineers can specify exactly what constraints must be met for a specific build and return if those constraints were met to the pipeline.

- Preview Apps ("lookasides")

For context, our pipeline deploys two types of instances: one production instance and unlimited preview app instances, which we also call "lookasides" throughout this project (this name was inspired by my summer internship, where they used this name to describe the same concept). Upon each PR to a release branch, using ansible scripts on self-hosted GitHub Actions, the pipeline dynamically deploys a lookaside by creating a docker container to host an app instance containing the PR's changes. As such, each lookaside is completely independent of production and other lookasides, having its own codebase/database. All instances are hosted on the same VCL instance, in separate docker containers (where nginx handles routing to each container). If a commit is pushed to the source branch while the PR is still open, the lookaside will automatically be rebuilt/redeployed to reflect these changes. Closing a PR automatically deletes the corresponding lookaside container to clean up space for more lookasides.

The novelty of lookaside instances is that other developers and product managers can conveniently view a fully deployed instance containing new changes before they reach production, so that feedback can be provided more efficiently.

- Nginx Reverse Proxy + Advanced Port Allocation System for Lookasides

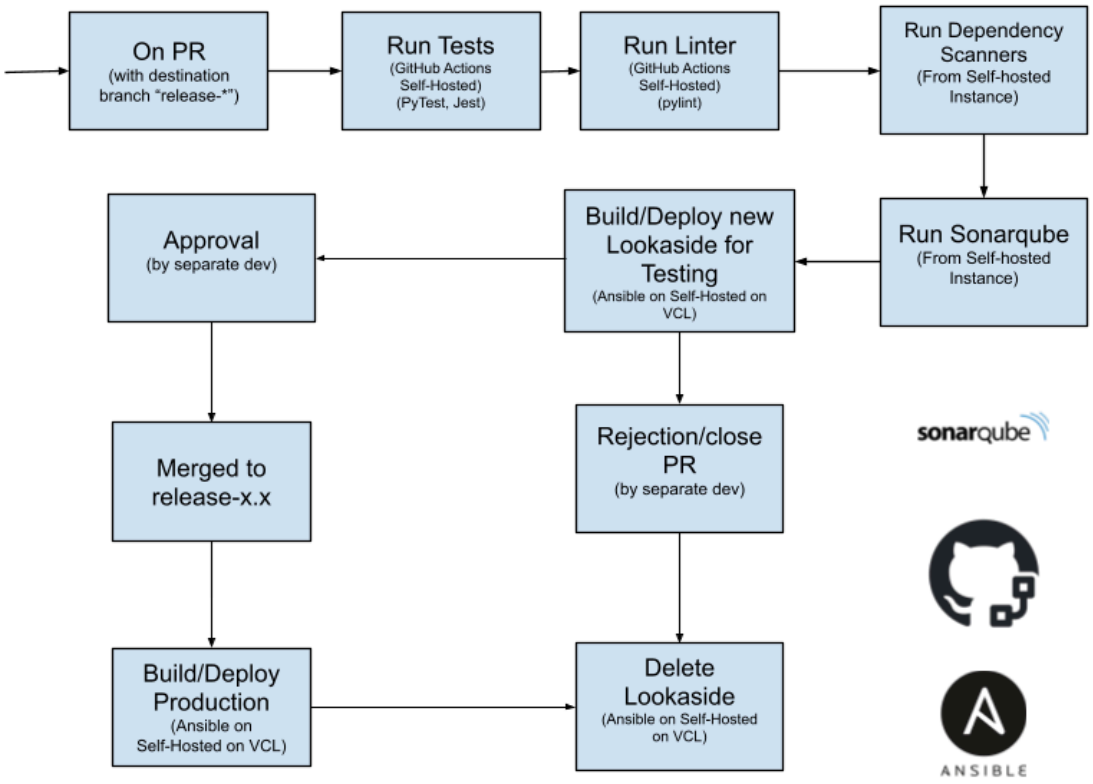
On our VCL instances, we were only given access to ports 3000, 443, and 80. Using port-forwarding with Docker, however, we can access the rest of the ports. The production container is assigned port 3000. Lookaside containers, upon being created, will be assigned to the first available port above 3000 (since there can be any number of lookasides). Note that within each virtualized container environment, apps run on port 3000. This system allows us to reuse ports after lookaside containers are destroyed.

URLs will follow the pattern “http://csc519-129-host.csc.ncsu.edu/” for production, and “http://csc519-129-host.csc.ncsu.edu/pr-[PR_ID]/” for lookaside instances. To route requests to their corresponding containers, we use nginx as a reverse proxy, which runs in a container on port 80.

- Branch Protection Rules on Release Branches

We added GitHub branch protection rules on branches following the name pattern “release-*”. These rules require a pull request with an approval, as well as the automated unit tests, linting, dependency scanners, and Sonarqube gates to pass in order to merge any changes into release branches.

Technical Approach



Our pipeline is set to run on a PR to any branch following the name pattern “release-*” (i.e., Release branches). Note that all scripts (actions + ansible) run on a GitHub Actions self-hosted runner (VCL).

The first gate involves running the unit tests, using pytest/jest, to make sure everything passes and there are no regressions in functionality. The next gate runs a linter, with a scoring threshold of 70% with pylint, to enforce code quality. The third gate runs dependency scanners, using pip-audit and npm-audit,

ensuring there are no known security vulnerabilities in our code's dependency tree. The fourth gate runs Sonarqube's code analysis suite, identifying both code quality issues and security vulnerabilities. If any of these first four gates fail, the workflow fails and nothing gets deployed, which is enforced through both branch protection rules and the workflow itself. If all code quality gates pass, an independent preview app instance (lookaside) gets deployed to http://csc519-129-host.csc.ncsu.edu/pr-[PR_ID]/, where reviewers can test out the new code/feature.

If the PR is approved, the code changes get merged to the release-x.x branch, and the main branch automatically gets updated to reflect the new release. Then, the production server is automatically rebuilt/redeployed to reflect the new changes on main. Lastly, the lookaside instance gets destroyed so that we can clean up space for future lookasides.

If the PR is rejected, no merges/changes will happen to the release branch or production server, and the corresponding lookaside instance gets destroyed. All deployments (and destruction of instances) happen through ansible scripts.

AI Usage

Generative AI was utilized in this project, mainly via Cursor and ChatGPT. Cursor is an AI-enhanced IDE that provides the codebase as context for LLM agents, which can either make direct file changes in "Agent" mode, or suggest edits/answer questions in "Ask" mode.

Cursor was used to create the *Goober Detector* app itself (i.e., html files and app.py), as we understood the project's emphasis was moreso on the deployment pipeline rather than the app itself. Unit tests were created with Cursor as well, since we focused more on integrating automated tests into the pipeline rather than the specific contents of the unit test cases. Cursor was also used to generate the jinja2 templates for the dynamically generated docker-compose files for each lookaside. Additionally, Cursor's "Ask" mode assisted in creating the nginx config files, since we were previously unfamiliar with how to use nginx as a reverse proxy for Docker containers. We also used Cursor to figure out how to make GitHub's Actions bot comment on PR's with workflow results.

ChatGPT was also used to help generate docker compose files, specifically the compose file that sets up the Sonarqube service that runs with a PostgresDB instance where it creates a container for the Sonarqube service and a container for the PostgresDB server and sets up the JDBC connection between the two containers.

Retrospective

- *What worked?*

Our division of labor ended up working out very well. We each took different parts of the pipeline (Chris took lookaside deployments + dependency scanning, while Jake took Sonarqube + Actions config), and this went smoothly since neither of our tasks impeded/depended on another person's tasks. As such, we were able to work at our own pace to create each step of the pipeline, with the final task being to string everything together in the end. Additionally, our communication was very clear and consistent, which was helpful since we kept each other up to date on the current state of the repository, leading to no conflicting ideas or merge conflicts.

- *What didn't work?*

In our proposal, we included Cursor's BugBot feature as an agentic, automated code review tool. We initially proposed this because it has a free tier, which we thought was readily available. However, this tool did not end up working out because BugBot is not meant for enterprise GitHub, and its usage on enterprise GitHub requires an administrator (of the entire enterprise; not just an organization) to whitelist its IP addresses. This was not feasible, so we had to ditch the idea and add the dependency scanners instead. Note that BugBot was already kind of extraneous to begin with since we already had 2 technical areas for the 2 of us.

- *What would we do differently?*

Next time, when deciding the features to add to our pipeline, we would conduct more thorough research so that if we use any 3rd party tools, we would make sure they actually work on enterprise GitHub. Additionally, we would make sure there is a free tier that is actually impactful to our project and not just some trivial item, which we learned while trying to integrate Cursor's BugBot. Also, we may look into using other cloud providers such as GCP or AWS so that we don't need to use NCSU's VPN.

Who Did What

Items done by Chris:

- Automated Linting
- Dynamic Lookaside Deployments
- Nginx Reverse Proxy Configuration
- Ansible Scripts
- Dependency Scanners

Items done by Jake:

- Sonarqube Integration
  - Quality Gate Control
- Self-hosted GitHub Actions Runner
- Automated Unit Tests

Security Extra Credit

We added a couple security features to our pipeline for the security extra credit. First, the dependency scanners were an extra item, and these scanners (pip-audit, npm-audit) scan the entire dependency trees for our frontend and backend for known vulnerabilities. If any vulnerabilities are found, the workflow fails, and the developer would then need to fix the dependency versions (or use alternatives) before any deployments are allowed to happen. Second, we use several tools in Sonarqube's suite of quality tools, but one of them involves making security checks in our code, which can identify common vulnerabilities, including CSRF, secure docker image usage, and more. If Sonarqube detects any highly severe vulnerabilities, the pipeline fails in order to enforce that the vulnerabilities get addressed.

Technical Commit Links (3 per person)

Chris Commits:

- Lookasides: e29e7f3
- Port allocation: 0503610
- Dependency Scanners: bc30801

Jake Commits:

- Actions Unit Tests: 6bd791d
- Sonarqube + Postgres Docker: d7737b2
- Quality Gate Action: 8ffe72e