Software Engineering Work Term Report

Optimizing the Continuous Integration/Continuous Delivery Pipeline

Yash Arora

Courses: CS 343 (Concurrency), CS 454 (Distributed Systems), CS 247 (Software Engineering Principles)

1. Context

Modern software development is fundamentally reliant on evolving quickly, transforming the needs of its users to a working solution as rapidly as possible. As the customers' requirements change, so must the product, for failure to adapt leads to dissatisfaction, and dissatisfied clients are sure to take their business elsewhere. There exist a variety of tools and frameworks that seek to tackle this problem, one of the most popular being the Continuous Integration/Continuous Delivery (CI/CD) Pipeline. CI/CD seeks to ensure that the core product is always in a deliverable state, that is, that at any point during development a complete, tested, viable version of the core product is available. When properly implemented, CI/CD processes result in rapid development, and can move stable code from the developer's environment to the end user in minutes. Yet if CI/CD is so effective in meeting the core need for software-oriented businesses to adapt quickly, why are these processes rarely implemented completely?

The truth is that applying a CI/CD pipeline to a codebase is relatively complex, particularly for projects that were not originally designed with continuity in mind. However, as CI/CD becomes the standard for product development, companies implement it without properly benefiting from its functionality. More specifically, there are a couple of core areas where these pipelines tend to break down, and it is up to the design decisions of an Engineer to address them. When this happens, it is easy to eliminate all the principles suggested by CI/CD to begin with, and the result is a product that requires additional resources, only to pretend to be ready for deployment at all stages of development. This, fundamentally, was the issue with Carbonite's implementation of CI/CD and was one of the core issues I was tasked with addressing.

2. The Problem

Summarized, the following are key problem areas that resulted from a poorly implemented pipeline.

1. Efficiency.
    a. Slow pipelines cause a variety of problems for developers, primarily in that they make even the slightest modifications take far too long to reach production environments. In many cases, software engineers on my team would wait several hours after committing code only to realize that it failed some existing test cases for trivial reasons. This would mean that we would have to stop working on our task and revisit the issue, wasting valuable development time.
    b. Inefficiency pertains not only to speed, but to resources used as well. Carbonite's Jenkins servers are hosted on Amazon Web Services' EC2 virtual machines, which means that increased use of the resource is billable. Though the metrics are confidential, the inefficiencies directly cost the company in a monetary manner.
    c. In the case of a need for an emergency hotfix, every additional minute between development and production is detrimental towards the company's product and

reputation. Though this never did occur during my time at Carbonite, it is crucial that we consider the worst-case scenario and prepare for it.

2. Dependence.
    a. In large-scale distributed systems such as the Carbonite Server Backup (CSB) platform, various repositories interact with each other, and can make the development process difficult and convoluted. For larger features, it is often the case that development occurs simultaneously, and affects various codebases. As a result, the existing CI/CD pipeline's reliance on other internal company repositories is a flaw, as it is often not possible to verify its functionality at any particular time.
    b. The tool is also reliant on third party infrastructure, including the Node (NPM) and NUnit (NuGet) packages that are used within the codebases. In the past, changes within these packages have led to system failures that have been difficult to diagnose, and it is crucial that the functionality is verified at all instances. However, neglecting to update to newer package versions is also an issue, as support for said packages may be terminated.

3. Constraints

3.1. Solution Functional Requirements

a) Allow developers to test the core functionality of their local code automatically and confidently merge it into a development branch.
b) Allow developers to completely test their code for all desired integration, as well as test all other sections of the codebase to ensure that no existing functionality has been changed.
c) Routinely and automatically merge successful development branch projects to the production branch.
d) Continuously ensure that the production branch is deployable at any stage of development.

3.2. Solution Functional Anti-Requirements

a) Allow significant untested code to reach the production branch.
b) Allow code that has failed any test cases to reach the production branch.
c) Allow any bugs to reach the production branch.

3.3. Solution Non-Functional Requirements

a) Efficient.
    i. The developer's personal branch pipeline (PR builds) should complete within 20 minutes.
    ii. The pipeline for the development branch (full builds) should complete within 1 hour.
b) Independent.
    i. The solution must ensure that it is not dependant on failures of other codebases, including those which are internal to the company, since their desired functionality cannot necessarily be verified at any particular time.

3.4. Solution Non-Functional Anti-Requirements

a) Expensive.
    i. This implies that the solution cannot change the existing pipeline provider, Jenkins, and must use the currently established infrastructure.
b) Slow.
    i. The pipeline must not require significant additional developer time, rather, it should be a tool that allows developers to integrate and deploy code more rapidly.

As is likely evident, the non-functional requirements here are far more important than their functional counterparts. This is because setting up a CI/CD pipeline itself is relatively trivial and solves a variety of problems itself. However, ensuring that the implemented design meets the non-functional requirements is less simple, as it requires careful analysis of each aspect of the pipeline. Further, as part of the design constrains the solution must be built upon existing infrastructure, meaning that it is in some cases impossible to choose between a variety of viable alternatives. Instead, my task involved creating this solution itself, and verifying that the requirements were met.

4. The Solution

The Jenkins CI/CD pipeline for Carbonite's Portal UI and API consist of the following stages as shown in the figure below. Much of the resource optimization design decisions pertained to testing, and as such the testing stage is shown broken down here.

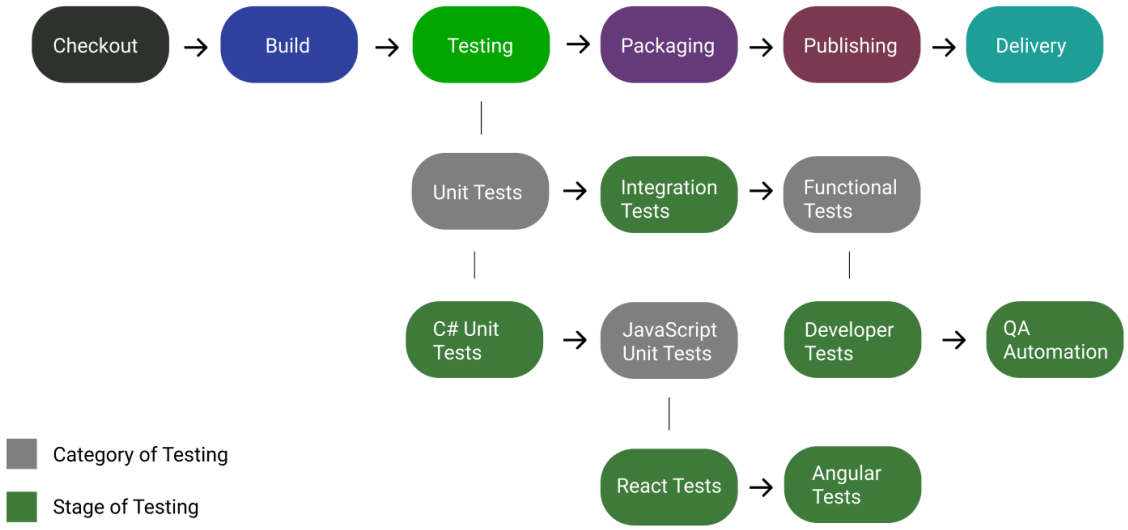

Figure 1: Complete Carbonite Server Backup CI/CD pipeline.

Evidently, the various stages of testing are the most resource-intensive, as they entail the core functionality of the pipeline. The figure below shows the pipeline without test categories, as it was being run before any optimizations were made. The values shown alongside each segment are representative of the average amount of time taken by said segment in minutes.

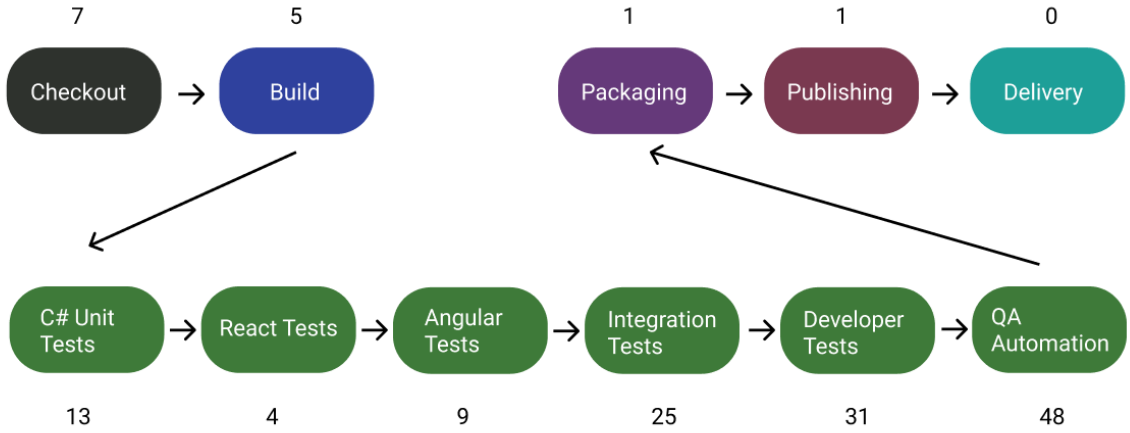

Figure 2: Pipeline with only tangible stages shown.

The stages of the pipeline related to testing, shown in green, were by far the longest. In total, the pipeline took an average of 144 minutes to run, or close to about 2.5 hours. This meant that every pull request resulted in a 2-3 hour waiting period during which it could not be merged into the development branch. Although developers could use this time to work on the next task, any errors in the PR would require the developer to have to revisit the previous task, losing the continuity in their work. To meet the both the functional and non-functional requirements, optimizations pertaining to parallelization, branch-type specification, and dependency caching were applied.

4.1. Parallelization

The most valuable optimization that was made pertained to parallelization. For context, this refers to running multiple processes concurrently rather than sequentially. As is visible in the figure below, the stages of testing were being run successively.

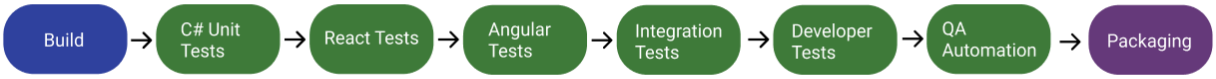

Figure 3: Pipeline with sequential testing stages.

Unfortunately, this is a severe mismanagement of resources, since the EC2 environments that were executing the tests were easily powerful enough to run several types of tests simultaneously. I suggested an alternate design which would group the test stages by category of testing, resulting in the diagram below.

Figure 4: Pipeline testing grouped by category.

With the test files separated by category, the different test stages within each category could be run in parallel. For example, while previously the Developer Tests would need to complete before QA Automation could run, they could now be executed at the same time. This could also be applied to the unit tests, where we could now simultaneously execute Angular, C#, and React tests. Below is a chart that represents this process visually.

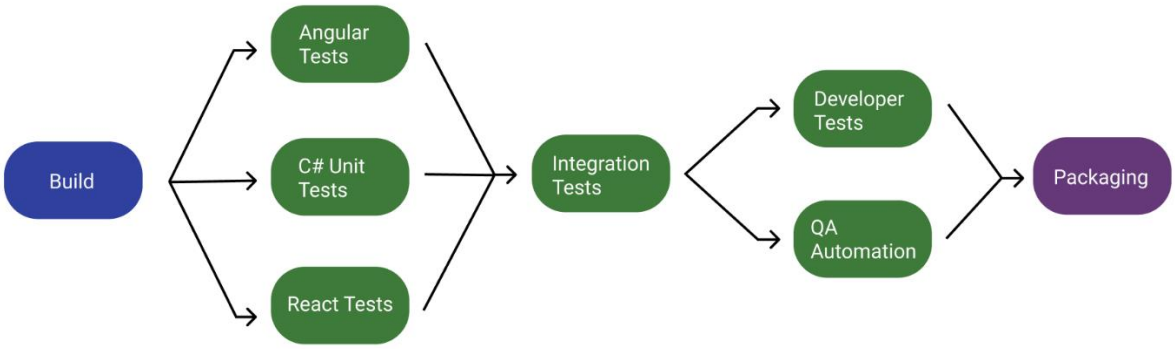

Figure 5: Pipeline with testing stages parallelized.

Parallelization of each category of testing resulted in a 30.56% saving of time used by the pipeline. While this optimization is good, it is still nowhere near the goal of 1 hour, much less 20 minutes. As seen in the diagram below, the integration and functional testing stages both communicated with the same testing database, which was why the test categories themselves still needed to be run sequentially. If this were ignored, SQL deadlock errors may have occurred which could corrupt data in the test database.

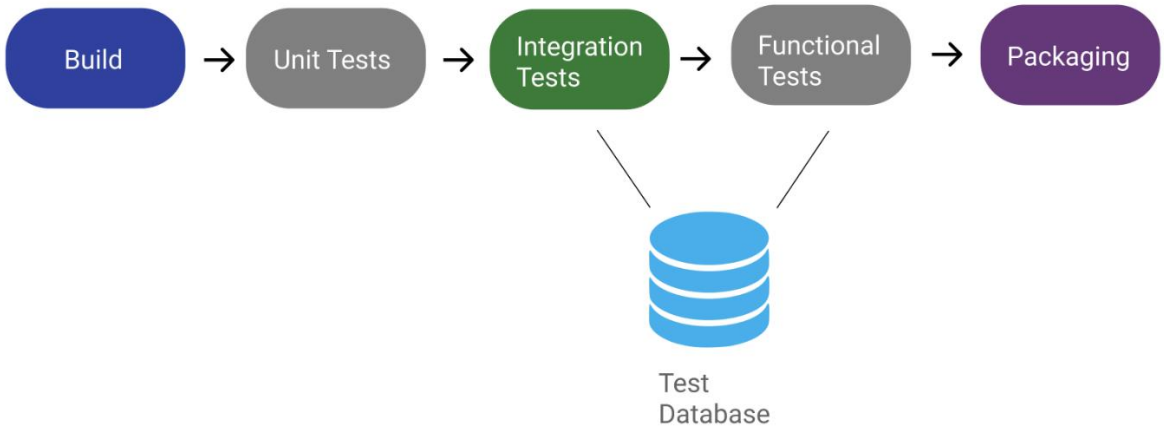

Figure 6: Diagram showing unified testing database.

I found that there was little reason for the integration and functional tests to communicate with the same database, as they tested completely different functionality. The design I proposed was to separate these databases entirely, so that the categories could be run in parallel.

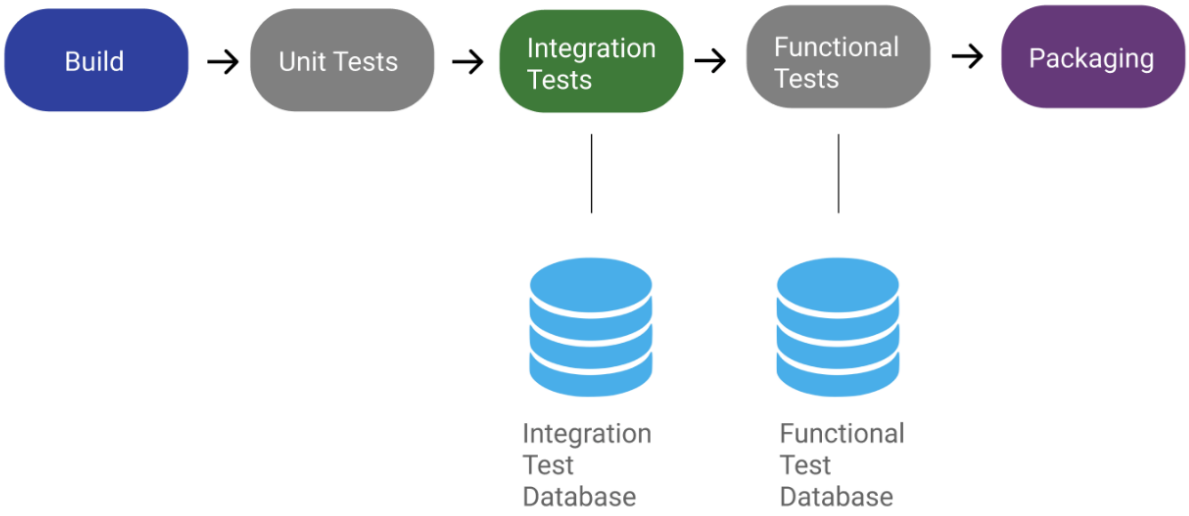

Figure 7: Diagram showing separate databases for integration and functional tests.

After ensuring that the separation of the databases did not break the testing environment, further parallelization was applied to the test categories themselves, as shown below.

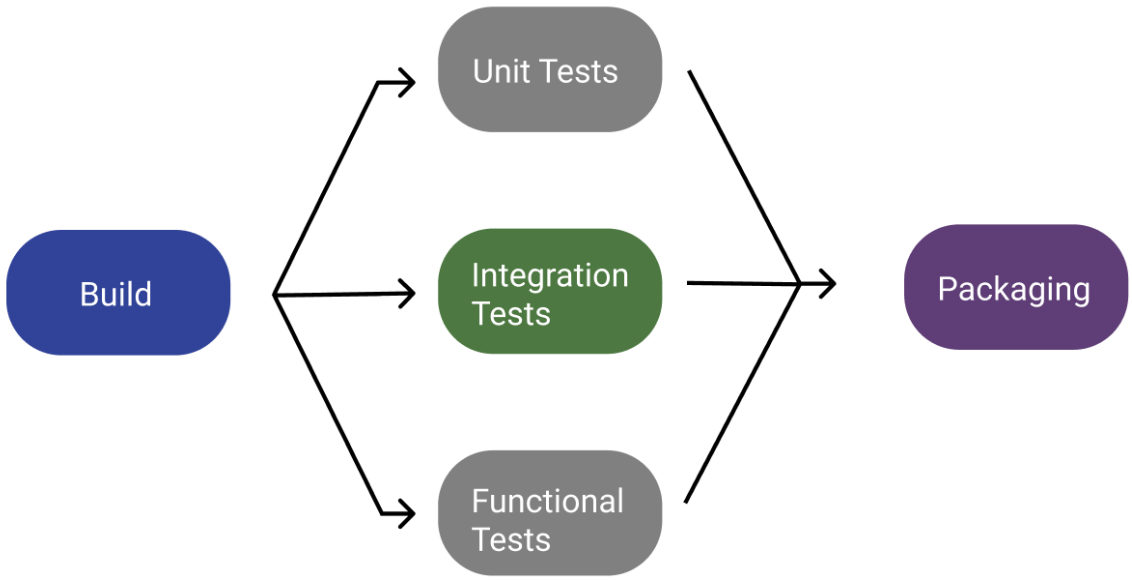

Figure 8: Pipeline with test categories parallelized.

Now, with both optimizations applied (parallelization of test stages within categories and parallelization of test categories themselves), all tests would execute in parallel and without database deadlock errors.

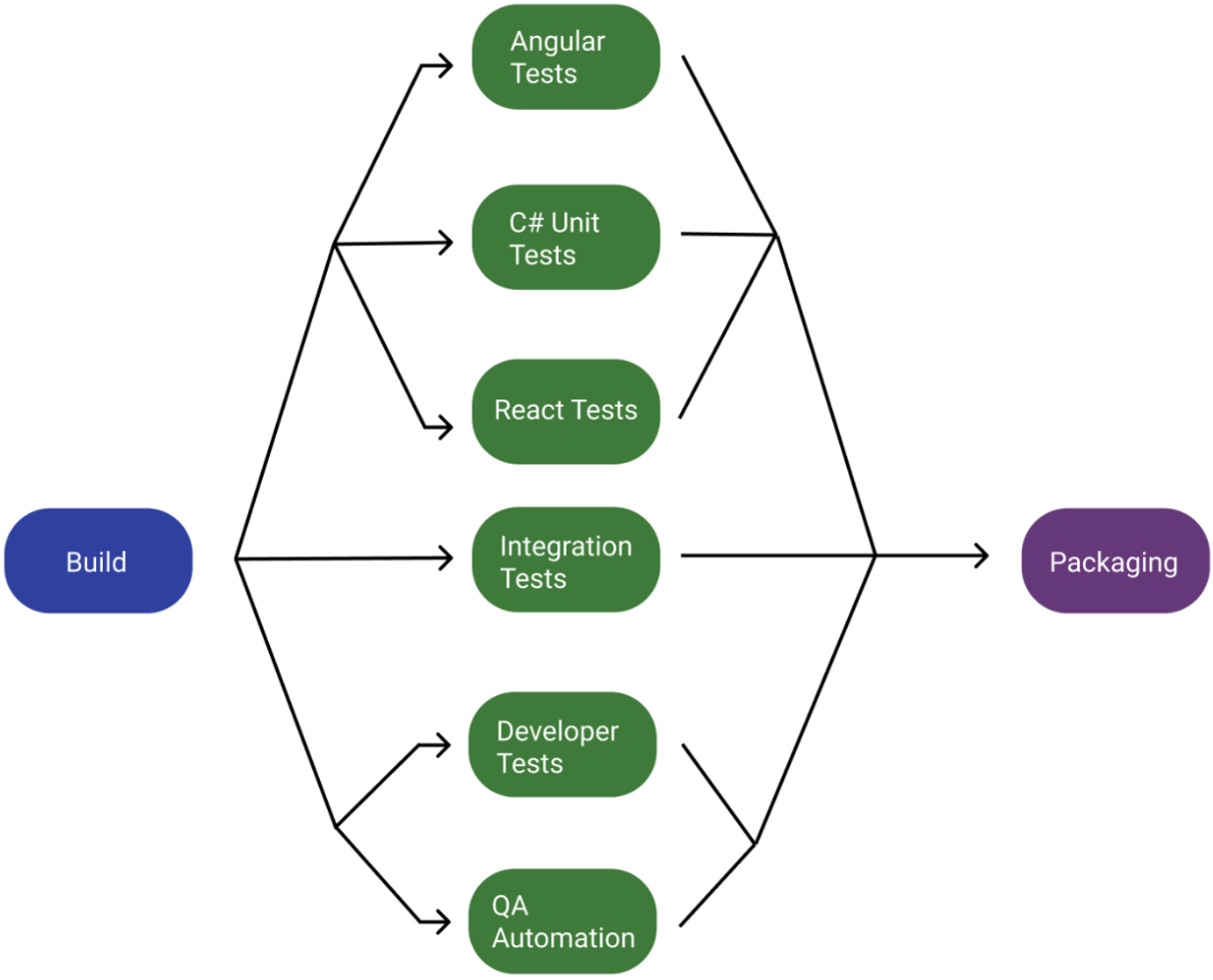

Figure 9: Pipeline with test stages and test categories parallelized.

Running test stages and categories concurrently rather than in series resulted in a reduction of time used by the CI/CD pipeline of 56.94%. This aspect of the solution begins to approach the functional and non-functional requirements, but changes are still needed before they are completely met.

4.2. Branch-Type Specification

One major oversight during the initial implementation of the CI/CD pipeline was that not all changes necessarily needed to be tested during every cycle. While it makes sense to verify all functionality when transitioning between the development branch and production environment, minor changes to the codebase do not warrant detailed testing. To use this information to further reduce the use of resources within the pipeline, I proposed that the pipeline should be aware of the branch-type and it to only run required stages. For simplicity, I chose to divide the builds into two categories – those which pertained to pull requests from developers' branches to the central development branch, and those which verified that the development branch was stable enough to be passed to production. After various conversations with Senior Build Engineers, it was recognized that the Integration Tests, Functional Tests,

Packaging, and Publishing stages were all unnecessary for PR builds.

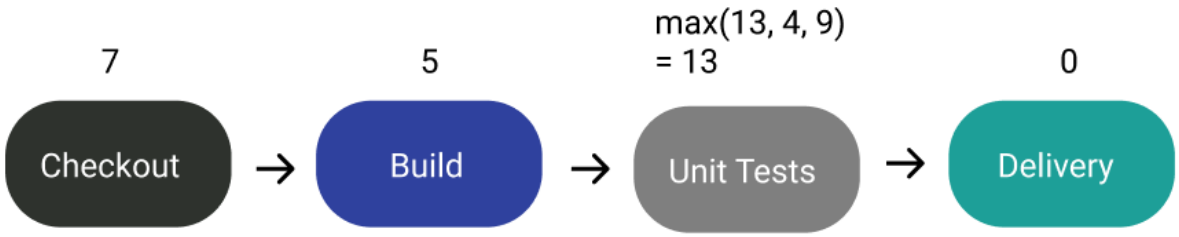

Figure 10: PR build pipeline with redundant stages removed.

As a result of this change, the PR build times were reduced from 62 minutes (post parallelization) to roughly 25 minutes, which was far closer to the 20-minute expectation as outlined in the non-functional requirements.

4.3. Dependency Caching

The final change that was made was in relation to caching the dependencies used during the pipeline. As mentioned earlier, the Carbonite Server Backup product is a relatively large-scale distributed system. The pipeline I was tasked with working on related to the CSB Portal UI and API, and as a result it was dependant on the functionality of the other components. The following figure shows a simplified view of these dependencies. For context, the "Installer" refers to an executable file that the user may run to install the backup software, and the "Agent" refers to a background service on the user's computer which performs a backup of the computer's files.

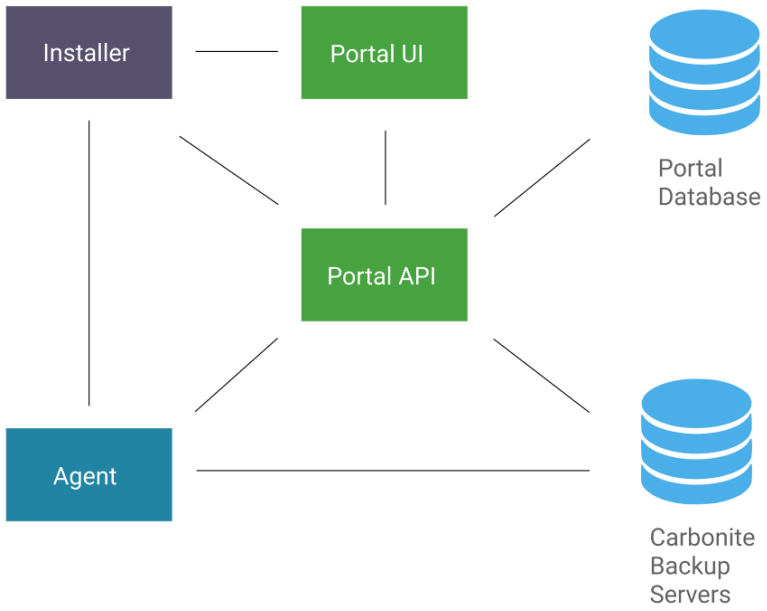

Figure 11: Carbonite Server Backup portal infrastructure.

These components were included as dependencies for the Portal's CI/CD pipeline and contributed to the time used by the checkout stage. This brought up the aforementioned issue with dependence since changes to the Agent or Installer should still be tested for proper integration with the portal. However, since the Agent and Installer codebases are not using a complete CI/CD cycle, failures within the dependencies' builds may appear as issues with the Portal when they are not. Having worked nearly exclusively with the Portal, I did not have the necessary insight about the other components in the distributed system. After conversing with Engineering Managers on several other teams we concluded that changes are pushed very infrequently to the production builds of both the Agent and Installer, and thus checking them out from version control during every Portal build is completely redundant.

I proposed that a Jenkins plugin may be used which caches both the Installer and Agent as dependencies and checks to see if there are changes to their production builds. If no changes are made (which encompasses the vast majority of cases), then the dependencies can be loaded from the cache. Otherwise, the pipeline would have to check them out from version control as done previously. Since this process was responsible for most of the time consumption of the checkout stage, implementing dependency caching reduced the stage to an average execution time of about 2 minutes. In addition, I found that caching was applicable to external dependencies as well, which further reduced the time to just 1 minute.

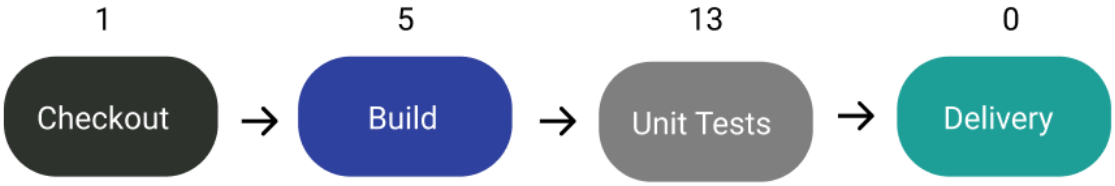

Figure 12: PR build time consumption in minutes by stage.

Figure 13: Full build time consumption in minutes by stage.

5. Conclusion

5.1. Impact

The result of the optimizations to the existing Jenkins CI/CD pipeline for the Carbonite Server Backup Portal was a system that allowed developers to quickly merge their code from their local branches to a central development branch. Additionally, the system continuously and automatically tested code from the development branch, pushing successful builds to production when finished. It ensured that no untested code, failed tests, or bugs would ever reach the production environment, and thus made sure that the production branch was deployable/deliverable to customers at all times.

Furthermore, the solution's design decisions were made particularly to increase performance of the system, using time as the primary metric. As previously stated, the original pipeline required an average of 144 minutes per cycle, whereas the new pipeline greatly reduced that number. More specifically, PR builds required an average of 19 minutes, and full builds required an average of 56 minutes post-optimization. This is representative of an 87% decrease in time used for PR builds, as well as a 61% reduction in time used for full builds. This is within the solution's non-functional expectations of 20 and 60 minutes respectively. By allocating resources more efficiently while using the same infrastructure as before, I was able to achieve the goal of guaranteeing that the solution was inexpensive to implement. Beyond time saved, the solution also ensured that it was not reflective of issues on other components of the distributed system or external dependencies. By caching only successful builds of the Agent and Installer (in addition to node modules, NuGet packages, etc.), the Portal's pipeline was solely reflective of the status of code within its own jurisdiction. Thus, by design, both the functional and non-functional requirements of the task were met. In an effort to exceed expectations, I also added a code coverage visualization dashboard which allowed developers to monitor their code to verify that it was being tested.

5.2. Benefits

The primary benefit of the optimizations to the CSB Portal CI/CD pipeline was that it saved valuable developer time. Rather than having to wait hours before being able to verify that code was tested, developers would have these results shown within a coffee break. It also added a sense of continuity to the work that was done, since tasks no longer needed to be revisited long after the code had been originally committed. Parallelization, branch-type specification, and dependency caching resulted in a solution that was fast, rigorous, inexpensive, and independent, exactly as desired.

5.3. Weaknesses

While I was happy with the solution that I was able to build, the truth is that it is still is not perfect for a distributed system. As a Software Engineer working on the CSB Portal, my solution did not address the same issues that were occurring with the Agent, Installer, and other components in the CSB network. The primary reason for this is that affecting all of the environments is a task beyond the scope of an internship and would likely require the introduction of some entirely new technologies. Having worked with infrastructure before, products such as CHEF offer the necessary DevOps tools to build pipelines for all of the components and allow them to communicate with each other in a unified manner. Going forward, it is worth exploring the implementation of a complete DevOps solution for all Carbonite tools.

5.4. Final thoughts

My original goal for my internship at Carbonite was to have a lasting positive impact on the product I was responsible for. I feel that given the task I was assigned, alongside the problem constraints I was working with, I created the most complete solution possible. While I recognize that there are definite areas of improvement that could be made to the infrastructure of the CSB suite, I believe I have had a permanent effect that developers at Carbonite will benefit from in the future. Finally, I would like to thank the Engineering team at Carbonite for their support, guidance, and friendliness during the past 4 months; it was a pleasure.

References

Jenkins JobCacher Plugin. (n.d.). Jenkin Plugins. Retrieved May 17, 2021, from https://plugins.jenkins.io/jobcacher/

Parallel Stages with Declarative Pipeline. (2017, September 25). Jenkins Documentation. https://www.jenkins.io/blog/2017/09/25/declarative-1/

Vasylyna, N. (2017, March 1). *Unit vs Integration vs Functional Testing*. QATest Blog. https://blog.qatestlab.com/2017/03/01/unit-integration-functional/