Using Continuous Deployment to Create a Serverless Rendering System

Nicholas Rivera, *UCLA*

Abstract

Continuous deployment is a set of software tools and techniques that lets developers release their changes to software frequently. Continuous deployment makes it easier to find bugs, get feedback on changes, and experiment with ideas. However, it also makes it easier to release bugs and create confusion that stems from a never-ending stream of updates. The goal of this project was to use the techniques of continuous deployment to create a serverless system that could automatically turn the content in R packages into a website. The focus was on making sure that the system was transparent to package authors so that they could focus on writing their packages without worrying about the details of the build system.

1. Introduction

R is a programming language used for statistical programming. Related R functions can be grouped together in a package. This package can have documentation for the functions and can have other documents called vignettes, which give a friendlier tutorial for the package. Vignettes are written in the R Markdown file format. R Markdown files are like normal markdown files with added features that make it easier to work with R code.

The goal of this project was to research and create a system to automatically generate a website from the documentation and vignettes of a group of R packages. The system needed to be flexible enough to let people add extra pages outside of the packages to the site, including a blog and library of examples. The website needed to be serverless, so no one person would need to maintain a server to host the website or a server to build the website. Also, the builds of the website needed to be fast, so that it would be easy to preview changes to the website. All these criteria make it necessary to work on a solution that uses the principles of continuous deployment.

2. Continuous Deployment

2.1 History

To create complex software, developers must rely on tools and techniques to manage complicated code bases. Continuous deployment is one set of techniques and tools that let developers frequently create changes to their software and release those changes into production [5, 6]. A continuous deployment setup means that developers can automatically release changes into production whenever they want, instead of having to wait for a rigid release window. Continuous deployment also entails that the barrier to releasing changes is low, which reduces the time that it takes to release those changes.

Continuous deployment stems from the principles of agile programming. The Agile Manifesto lays out these principles. As highlighted by Rodríguez, et al., one of these principles states, “Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.” Another principle states, “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software” [1]. Continuous deployment is a realization of agile programming principles because it lets developers deliver software to customers on a short timescale [6].

2.2 Benefits

The obvious benefit of continuous deployment is that it lets developers release software frequently because the barrier to releasing changes is low. This itself comes with many other benefits.

First, continuous deployment makes it easier to find and fix bugs. Bugs are found in production after software is released. Because it is easy to release changes into production with continuous deployment, there is only a small gap between finishing writing the code, releasing the change into production, and finding a bug in production. Fixing the bug is fast “because the developer just finished and can remember what he or she just did” [3]. Also, because changes are easy to release, code is deployed after making only a few changes at a time. This contrasts with waterfall development, where large batches of changes are released at once. This makes it easier to find bugs because only a few changes are put into production on each deploy, so it is easier to find the change that caused the bug [3].

This factor was helpful for me. Whenever someone found a bug on the website, it was easy to track down because I kept a historical copy of the website on each deploy. If a bug popped up, I could then go through the historical copies to find the deploy that caused the bug. Because I only released a few changes on each deploy, it was easy to find which change was the culprit. And because bugs were found soon after the deploy, I was spared from digging through months of historical copies.

A second benefit of continuous deployment is that it lets developers quickly get feedback on their changes. When there is only a small gap between making a change and

releasing it, the time between making the change and getting comments on the released version is also small [3, 6].

Quick feedback helped shape the design of the website. Whenever I made a visual change to the website and wanted feedback, I could quickly create a preview of the changes and share the link with others to get feedback. If I released a change to production that later needed to be improved, I could get feedback on that, make a quick change, and deploy to production again. This made it easier to experiment with changes to the build system and the look of the website. I never had to worry that I would not be able to tweak something in production if I did not like it later.

This leads to the third benefit, which is experimentation. Continuous deployment lets developers make small changes to their code, deploy them, and make sure that those changes are compatible with the rest of the software that is already in production. One way to do this is to release changes to production but hide the results of those changes under a configuration setting. Called dark launching, this prevents developers from having to keep a stale separate branch with that change until it is ready to merge into the master branch [5]. When it is time for users to see the feature, just deploy a change to a configuration file and the feature will show in production.

This played an important role in the development of the website. I released a main part of the website, the blog, but hid it until it was ready to turn on. I did not have to keep the blog's features on a separate branch. If I had to keep it separate, keeping the branch in sync with the master branch would have taken an unnecessary effort and would have discouraged experimenting with additional blog features.

2.3 Drawbacks

Being able to deploy software changes whenever can lead to problems. One problem is that it is easier to deploy bugs [2]. Because so many deployments happen so often, it becomes harder to do lengthy testing on each deployment [5]. This was a big obstacle on this project. On each deploy, I had to make sure to do cross-device testing on a desktop and phone and cross-browser testing on Edge, Firefox, and Chrome. However, because deployments happen so often, I did not have the resources to go through every page on the website and check for regressions. I could only test a handful of pages on each deploy.

Secondly, the rate of updates can be a problem. Continual deployments mean constant updates in production. It can be hard for users to keep track of this stream of updates, making the software harder to use [6]. To make it less confusing to deal with the stream of updates on the website, I kept a weekly note of everything that I changed over the past week. This log made it easier to explain what changes took place on each week. Keeping track of added features and features-in-progress on a Kanban board also helped.

2.4 Barriers to Implementation

Despite the many benefits of continuous deployment, there are many barriers that can prevent it from being used in an organization.

The first is company culture. Everyone involved on a project, not just the developers, needs to be informed about the continuous deployment process [2]. According to a case study done at Atlassian about the difficulties of implementing continuous deployment, "Both practitioners' literature [4,37] and the results of this study indicate that to successfully adopt the CD process in an organisation, a company wide effort must be made" [2]. This was the most important challenge to keep in mind for this project. Others in the research group had to be kept up to date on the build system's changes because they had to know how to preview pages and use the deployment system.

Secondly, creating automated testing is essential when setting up a continuous deployment system. Automation lets small groups of developers handle large projects by checking for regressions between deployments [5]. However, developers sometimes run manual tests after automation to check features that automation cannot catch, "especially those related to quality attributes such as performance and security" [3].

Automation was an important part of this project. I took up maintenance of a tool that checks for broken links on a website. I set up the tool to run automatically after each deployment to check if the released changes broke any pages. I would get a notification if the tool found anything, which let me fix any regressions quickly.

Finally, the last barrier is managing the configuration of development systems. Keeping a close watch on the configuration of the development and deployment environments can prevent problems when using continuous deployment. For example, the configuration of dependencies or server settings, if not maintained, can bring the build to a halt [5]. To avoid configuration problems, I used the build tool Travis to run our builds. Each build starts from a clean virtual machine image and reads in the settings from one main configuration file. This prevents problems where the project builds on one person's computer but does not work on another's. By having one build system that always starts from a clean state and uses a standard configuration file for everyone, I made

Figure 1: The process of rendering an R Markdown file.

sure that there were no configuration errors between builds.

3. Existing Software

The first steps in this project were to research what software already existed in the R community to build websites. Then, I needed to investigate how this software worked to see ways that I could extend it to accomplish the project's goals.

Writing any kind of document with R starts with an R Markdown file. R Markdown extends the markdown markup language. The difference is that R Markdown files let authors embed chunks of runnable R code into the document. This feature means that there are extra steps that must be taken to convert an R Markdown document to HTML that do not have to be taken with a regular markdown document. These steps are shown in Figure 1. First, an author writes the page in R Markdown. Then, a function in the rmarkdown R package is called on the document. This function calls knitr, which is another R package. knitr takes the R code in the document, runs it, and splices the output back into the document. This process is called knitting. Then rmarkdown calls pandoc to convert the document into the HTML format [4].

pkgdown is an R package that uses the above steps to create websites. Written by Hadley Wickham and Jay Hesselberth, they designed pkgdown to take the documentation files and vignettes in an R package and turn them into a website. Vignettes are written with the R Markdown file format, so pkgdown uses the steps above to convert them into website pages.

blogdown is another R package that can create websites. This package is meant for creating blogs using R Markdown files. Written by Yihui Xie, blogdown builds blog posts by following the same steps that pkgdown uses to build vignettes.

4. Project Goals

Unfortunately, there are limitations to pkgdown and blogdown that prevented them from being a simple solution for this project. pkgdown can only create a website from one package. If a developer creates two R packages, each needs its own separate website if the developer uses pkgdown. Also, pkgdown does not have the tools for a blog, so a developer cannot easily post updates about the R package on the website. blogdown has its own set of limitations. blogdown cannot combine its blog with the documentation for an R package. This forces the blog to live on its own blogdown website and the package documentation to live on its own pkgdown website.

Based on these limitations, the goal of this project was to create a system that could

1. Take a list of R packages and combine their documentation and vignettes together into one website. The website also needed to support blog posts and arbitrary pages that lived outside of the packages.

2. Rebuild the website daily so that updates to the packages and blog would appear automatically.

3. Run in a serverless environment.

4. Have as low an overhead as possible so that package authors can work on their packages without worrying about the internals of the website.

5. Build the infrastructure so that it scales when more pages and community contributions are added.

5. Architecture

5.1 The Serverless Build System

A simplified view of the build system's architecture is shown in Figure 2. Travis is the backbone of the build system. Travis is a service that gives me the ability to run builds of the website without having to manage my own build server, allowing the builds to be serverless. If someone makes an update to the website, for example, if someone updates a blog post or tweaks the CSS in the GitHub repository, the change will automatically trigger a build on Travis. Travis starts the build after it sees a push on the GitHub repository, so anyone can make edits to the website straight on the GitHub repository—no local development environment is needed. To build the website, Travis pulls in the website's files, including the HTML templates, CSS, JavaScript, and any other R Markdown website pages, from the website's GitHub repository. It also downloads a fresh copy of each R package that it needs to build. Travis runs the build, renders

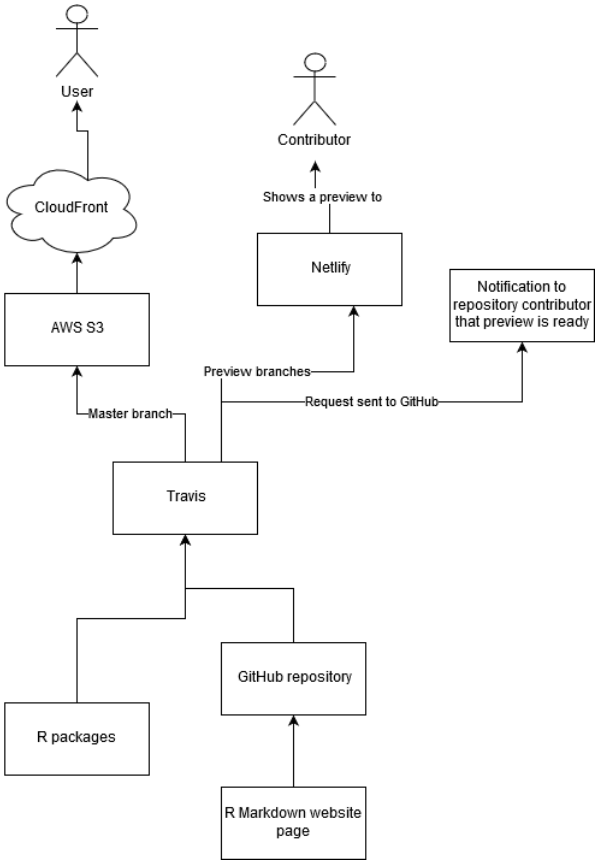

Figure 2: An overview of the entire build system.

the R Markdown files into HTML pages, and then, depending on whether the build started from the master branch or another branch, it will send them to one of two hosts.

5.2 The Serverless Hosts

Like the build system, there are no servers to manage for hosting the website. The production website is hosted on Amazon Web Services's Simple Storage Service (AWS S3). After a successful job on the master branch, Travis pushes the rendered HTML files into a bucket on S3, which overwrites any old files in the bucket. After a failed job on the master branch, Travis pushes no files to the bucket. Whatever old files are still there will show. Because each Travis job only builds a part of the website, a successful job will overwrite only a portion of the files in the bucket. This is a key part of the continuous deployment system: On a build failure, no changes are released into production. On a build success, the changes are automatically deployed.

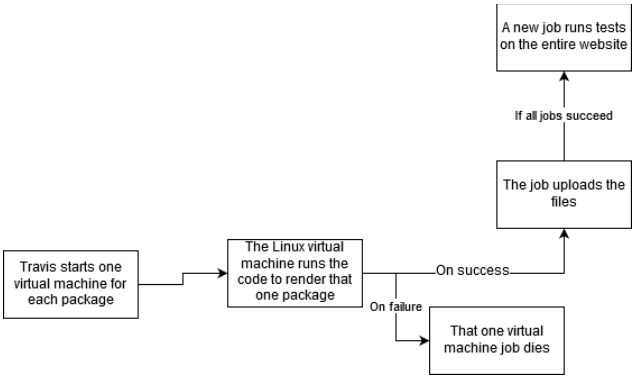

Figure 3: An overview of the internals of the Travis stage.

I used AWS CloudFront to cache the files at servers across the globe. Without CloudFront, when someone requests a file on the website, the request must travel across the globe to fetch the files from one server. With CloudFront, the website's files are distributed on multiple servers across the globe, which are called edge servers. The geographically closest server to the user will serve the file, making load times faster. The downside to CloudFront is that the cached files in these edge servers are only updated once a day from the main server by default, so I must send a request to invalidate the cache on each build.

I host previews on a service called Netlify. After a successful job on a non-master branch, for example, on a pull request, Travis will push whatever pages it built to Netlify. This lets contributors preview their changes. Netlify also keeps historical copies of each of these previews. This is useful for debugging because I can compare before and after versions of the website on each change.

After building a preview, I send a request to the GitHub API to add a notification to the pull request that triggered the preview build. This lets the contributor know that their preview is ready.

5.3 The Travis Build

When I request a build on Travis, Travis starts a Linux virtual machine from a clean Linux image. Each time that I start a new build, Travis starts from the same clean image. This ensures that the state at the start of each build is consistent.

The Travis build starts with the files in the website's GitHub repository. For example, say that I want to add a new blog post to the website. I create the R Markdown file, write my content, and then push it to GitHub.

Travis detects the push and starts a build (Figure 3). Builds on Travis start with a configuration file. This file dictates the behavior of the Travis build. Travis does not allow anyone to interact directly with the virtual machine. Instead, when the image starts up, it runs the scripts that I have in the configuration file and then exits.

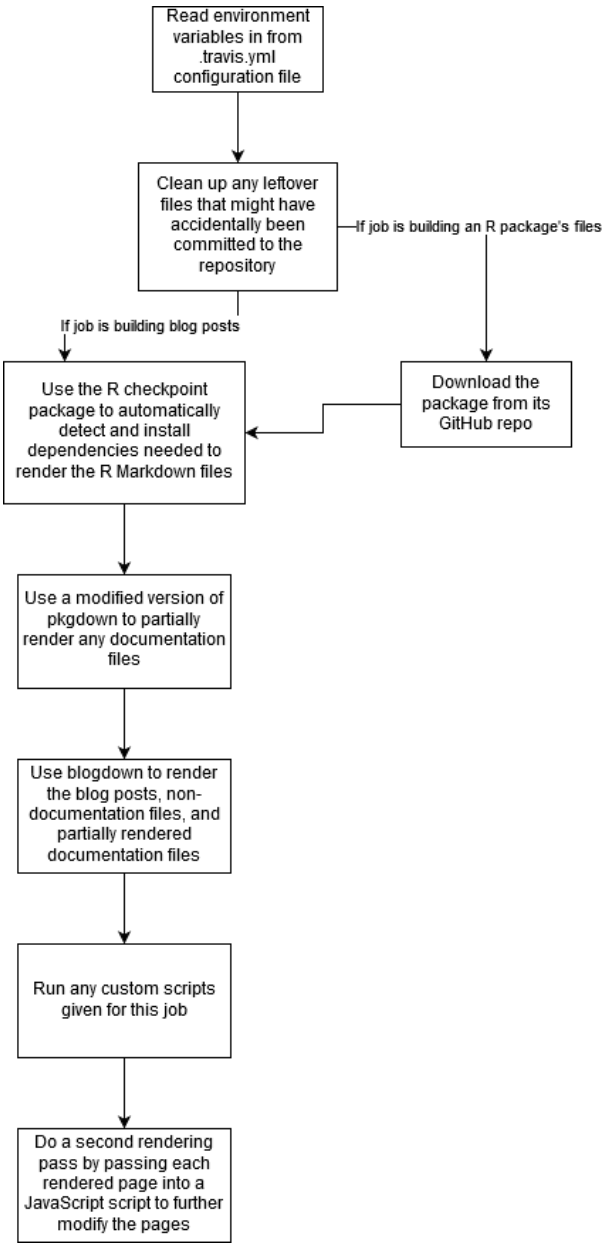

Figure 4: An overview of the static rendering process. Each job follows these same steps.

Each run of the virtual machine is called a job. Jobs can run in parallel, serially, or in any combination of the two. Each collection of jobs is called a build. When Travis detects a push on the GitHub repository, it starts up many jobs in parallel. Each job builds one R package's worth of documentation and vignettes. One job is also dedicated to building blog posts and non-package pages on the website.

In my blog post example, because I just committed my post, Travis scans the configuration file for the list of packages and starts one job for each package. One of those jobs runs and renders my files. At the same time, many other jobs render the other packages on the website. Once a job finishes, it uploads a preview to Netlify for me to see. Once the preview looks good, I can merge the branch into master, and the changes will be deployed into S3. Once all the jobs finish on the master branch, an automated test runs on the website to ensure that there are no broken links.

This separation of jobs is important. It ensures that if one job fails, the other jobs will succeed and update other parts of the website. This prevents one bad package from stopping the entire build of the website. Running jobs in parallel also speeds up website builds.

5.4 The Static Rendering Process

Each Travis job follows the same steps to render a package (Figure 4). The job starts by reading in environment variables from the Travis configuration file. The environment variables tell this job the details of the package that it needs to render.

Once the environment variables are in, a quick cleanup is done just in case someone committed temporary files to the repository. Next, as dictated by the environment variables, Travis downloads a fresh copy of the package that it will render. Next, an R package called checkpoint scans through any dependencies in the R Markdown files. This is important because it lets package and blog post authors use any dependencies that they want without having to manually tell the build process about them.

Next, a modified version of pkgdown runs to create the documentation pages. pkgdown pulls in custom HTML templates to create the layout for these pages. To render the documentation, pkgdown takes those templates and fills them with data from the documentation within the packages. pkgdown outputs HTML files that are not complete: They are missing the header of the page, menus, and other sidebars. Only the body is complete.

Next, those partially rendered pages, blog posts, and non-documentation pages are fed into blogdown. blogdown, like pkgdown, uses our HTML templates to lay out the site. It reads in data from the R Markdown files, converts the markdown formatting characters into HTML, and then copies that HTML into the correct spot in our templates. The templates are written in HTML with embedded snippets of Go. The snippets of Go allow logic to be embedded in the templates. This logic further manipulates the HTML that is copied into the templates by, for example, programmatically adding a summary under each heading or by automatically removing HTML tags from blocks of text.

This type of rendering is called static site generation. Sites like those created with WordPress are dynamically generated: When a user requests a page, data is fetched from a database, stuffed into the HTML templates, and then served to the user. Sites built with pkgdown and blogdown are statically generated: The pages are rendered using the templates before any user requests them, and then they are uploaded to a server. When a user requests a page, no communication with a database is necessary because the pages have already been created beforehand. This reduces security risks because there is no need for the user's request to talk to the database. Any database communication is done during the rendering process; the user only sees the final rendered pages. Static site generation is also faster because the pages are generated before the user requests them. No time is spent rendering the pages on a request, and all pages can be cached using a service like CloudFront. Dynamic sites cannot be cached like this.

Once all initial rendering is done, a second pass is done using JavaScript to modify the pages. This is necessary because complicated logic cannot be included inside the Go HTML templates. The JavaScript runs, reads any data that it needs from more YAML configuration files, and makes adjustments to the page's HTML. For example, these scripts add the table of contents to each page and the names of the package authors to the package homepages. After the second rendering pass is done, the render is finished.

6. Conclusion

Continuous deployment is not the solution for everyone. The benefits are great: Continuous deployment makes it easier to fix bugs, get quick feedback on changes, and experiment with new features. However, there are downsides, including making it easier to deploy bugs to production and the confusion that comes with constant updates to software. However, this project would not have survived without the benefits of continuous deployment. Continuous deployment made it possible for me to create a system where anyone, anywhere in the world can edit a website without running a local dev environment. Everyone is then able to see an exact preview of what the website will look like, and then those changes can be deployed, again from anywhere in the world.

With more time, there are improvements that I could have made. I could have decoupled some parts of the project, including the notification system for website previews, so that they could have been reused in other projects. I could have rewritten some JavaScript files using TypeScript to make them easier to maintain. And I could have streamlined the build system in some parts to make it easier to understand.

However, going into the future, I hope that this system makes it easier to maintain a website generated from a family of R packages.

References

[1] K. Beck *et al*., *The agile manifesto,* https://agilemanifesto.org.

[2] G. Claps, R. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," in *Information and Software Technolog,* vol. 57, pp. 21-31, 2015.

[3] M. Leppänen *et al*., "The highways and country roads to continuous deployment," in *IEEE Software*, vol. 32, no. 2, pp. 64-72, Mar.-Apr. 2015.

[4] R. Linacre, *Relationship between R Markdown, Knitr, Pandoc, and Bookdown,* https://stackoverflow.com/a/40563480.

[5] C. Parnin *et al*., "The Top 10 Adages in Continuous Deployment," in *IEEE Software*, vol. 34, no. 3, pp. 86-95, May-Jun. 2017.

[6] P. Rodríguez, *et al*., "Continuous deployment of software intensive products and services: A systematic mapping study," in *Journal of Systems and Software*, vol. 123, pp. 263-291**,** 2017.