Supplement to Shiny in Production

2019-01-14

Contents

Chapter 1

Shiny in Production Workshop

This document is full of supplemental resources and content from the Shiny in Production Workshop delievered at rstudio::conf 2019.

Figure 1.1: rstudio::conf 2019

Chapter 2

Introduction to Shiny in Production

Why are we here?

> Shiny applications are being deployed in high-value, customer-facing, and/or enterprise-wide scenarios. Unfortunately, they are often being done without the benefit of best practices. This workshop will help you and/or your IT colleagues who support your data scientists learn how to accelerate a successful Shiny application deployment in production scenarios.

Over the past year, software developers at RStudio have been working hard to dispel rumors that Shiny "isn't ready and can't run in production". They've built a bunch of cool new tools that are useful in preparing applications for production and understanding how to configure and scale them for optimized performance and user experience.

This workshop will cover all these new tools for shiny development as well as the equally important logistical pieces of a production story:

- What does production infrastructure and tooling look like for Shiny apps?
- How do we get Shiny apps from development into production?
- How are Shiny apps maintained production?

> When developers begin to think of infrastructure as part of their application, stability and performance become normative. - Jeff Geerling "Ansible for DevOps"

2.1 Workshop Objectives

Running Shiny in Production

- Is Shiny ready for production?
- What does it take to get there?
- What do you need to know?
- What tools are available?
- How do you develop a workflow?

Understand the importance of incremental changes and testing

- Version control

- Tests for package upgrades
- Use of separate environments for staging and production
- Incorporating automated testing into a development workflow: `shinytest`

Data product tradeoffs

- What are the advantages to using Shiny vs Plumber vs R Markdown
- What is the difference between a stateless Plumber API and a Shiny Session?

Development vs. Production environment considerations

- Defining a data model
  - Working with databases
- Tools for understanding application performance
  - `shinyloadtest`
  - `profvis`
- Tools for improving application performance
  - Plot caching
  - Synchronous vs asynchronous paradigms: `async`

Deployment architecture and tools

- Introduction to analytic infrastructure
- Configuration management
- Resources for scaling horizontally

2.2 Workshop Infrastructure

- RStudio Connect
- PostgreSQL
- RStudio Server Pro

Instructions for accessing the classroom environment are available in the workshop slide deck.

Chapter 3

Introduction to the Application

Every Application has an Origin Story

Data Scientists at RStudio University have discovered that there are trackable traits and behaviors students engage in that have been predictive of the desired 4-year graduation track.

They have built a shiny application that can be used by the very data-savvy advisors at this illustrious institution to identify students in need of guidance and show them the top behavioral factors driving individual predictions coming out of the model.

The POC was a smashing success - but now *our advisors actually want to use this thing for real.*

- We've developed a nice app
- We want to put it into production
- We want confidence that it will perform well in production, both now and in the future

3.1 Activity: Explore the Application

First: Open `app.R` *and Run the Application*

Discussion: Explore the Application

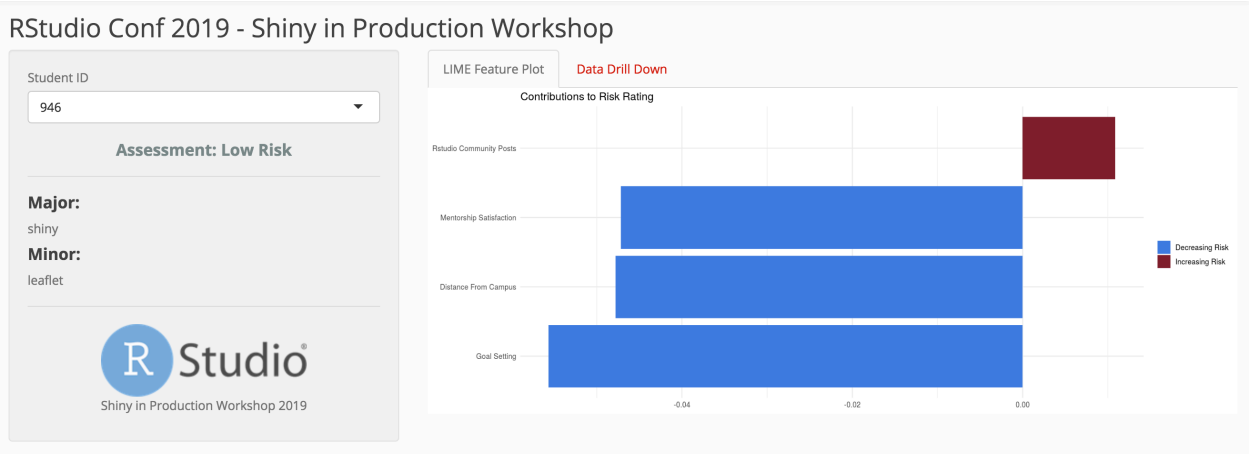


Figure 3.1: App Preview

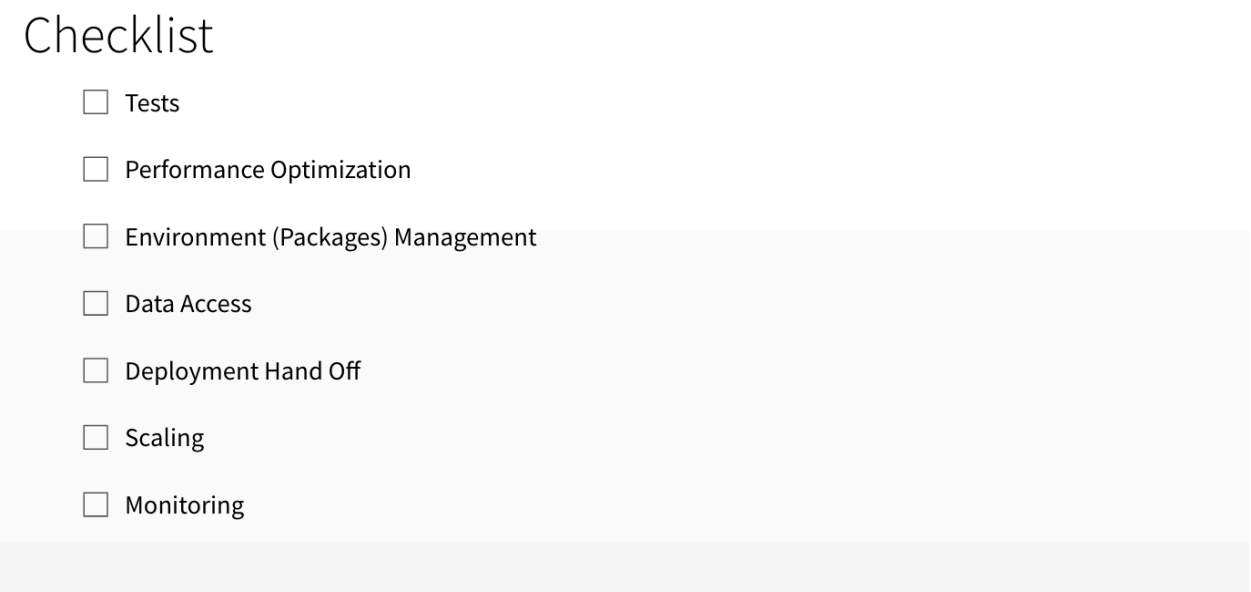


Figure 3.2: Ideas for Getting Started: Checklist

- Are there any parts of the app code that don't make sense?
- Is this application ready for production?
- How would you define "production"?
- What insights would be useful to have about the application before we try to deploy it?

Additional Discussion:

- What is your current process for taking applications into production?

Deliverable: Start a Plan

- Create a checklist
- Outline the steps you might take to put this application into production

Checklist for Taking Applications into Production

A high-level Checklist to build off of:

Chapter 4

Application Testing: `shinytest`

- You've developed a nice app
- You've put it in production
- You want to be confident that it will keep running in the future

Things that can change/break a Shiny application

- Modifying code
- Upgrading the `shiny` package
- Upgrading other packages
- Upgrading R
- External data source changes or fails

4.1 Testing Options

- Manual testing
  - time intensive
  - inconsistent
- Automated testing (hard)
  - web browser
  - simulated user interactions
  - tests for graphical elements

Enter: Snapshot-based testing Webinar Slide Deck

- Can be easier to create tests
- Can test an entire application
- More sensitive to spurious changes

Automated Testing for Shiny Apps

- Blog

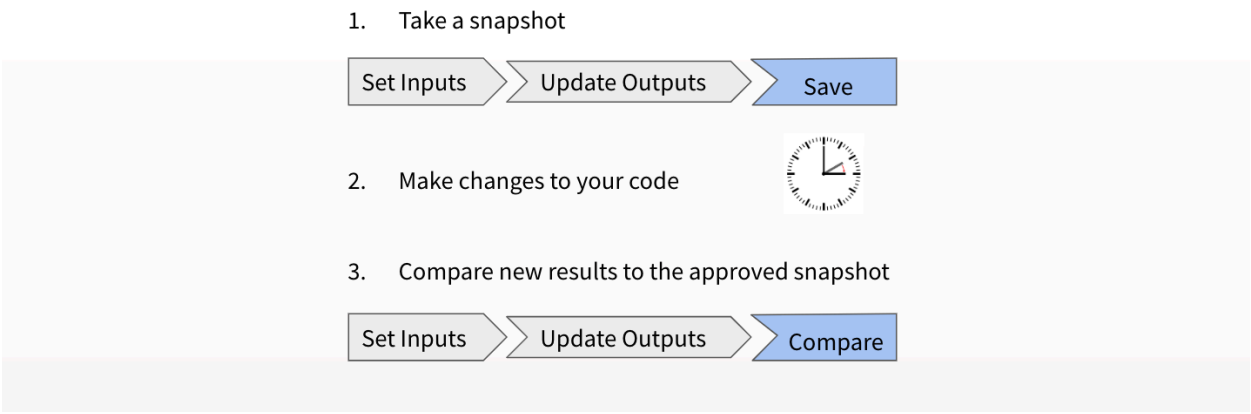


Figure 4.1: Snapshot Testing

- **Create a test:** Record user interactions with the app, saves them in a *test script*.
- **Make baseline (expected) snapshots:** Run the test script, which replays the interactions on a headless browser. This takes snapshots of application state along the way and saves them for later comparison.
- **Do your work:** Modify your app, modify your data, upgrade packages, upgrade R.
- **Re-run the test script and compare:** Run the test script again and take snapshots, then compare the new snapshots to the expected snapshots.

Figure 4.2: shinytest proceedure

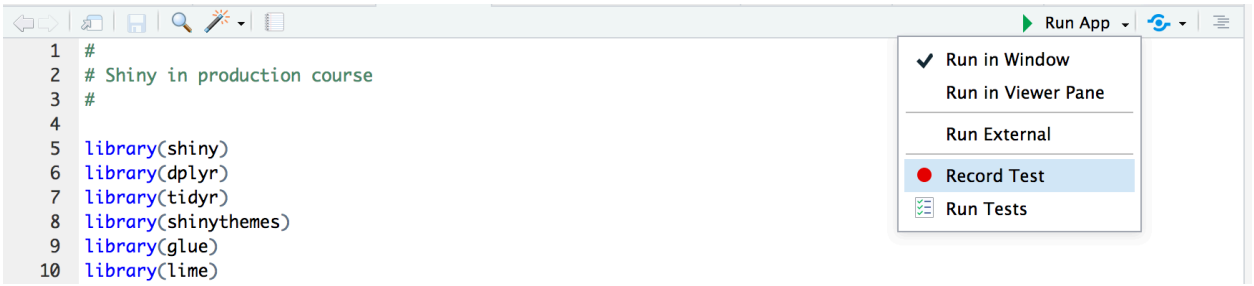


Figure 4.3: Record Test Button

`shinytest` is a package (available on CRAN) to perform automated testing for Shiny apps.

Basic `shinytest` proceedure:

Support for `shinytest` is available in RStudio v1.2 preview.

Installation

'install.packages("shinytest")

Note: When running `shinytest` for the first time, you may be prompted by the RStudio IDE or package warning messages to install some dependencies. `shinytest` requires a headless web browser (PhantomJS) to record and run tests.

- To install it, run shinytest::installDependencies()
- If it is installed, make sure the phantomjs executable can be found via the PATH variable.

Record Tests

- Run `recordTest()` to launch the app in a test recorder.
- Create the tests by interacting with the application - this will allow the recorder to snapshot the application state at various points.
- Quit the test recorder. This action will trigger the following events:
  - The test script will be saved as a .R file in a subdirectory of the application named `tests/`.
  - If you are running in the RStudio IDE, it will automatically open this file in the editor.
  - The test script will be run, and the snapshots will be saved in a subdirectory of the `tests/` directory.

To record tests from `R`, run the following:

```
library(shinytest)

recordTest("path/to/the/app")   #Replace with the correct path
```

To record tests from RStudio v1.2, when an application file (app.R, server.R, ui.R or global.R) is open in the editor, a button labeled *Run App* will appear at the top of the editor pane. Click on the small black triangle next to this button to reveal the menu of extended options.

This launches the Shiny application to be tested in a separate R process. We'll refer to this as the **target app**. At the same time, the current R process lauches a special Shiny application which displays the target app in an iframe along with some controls. We'll refer to this as the **recorder app**. You should see something like this:

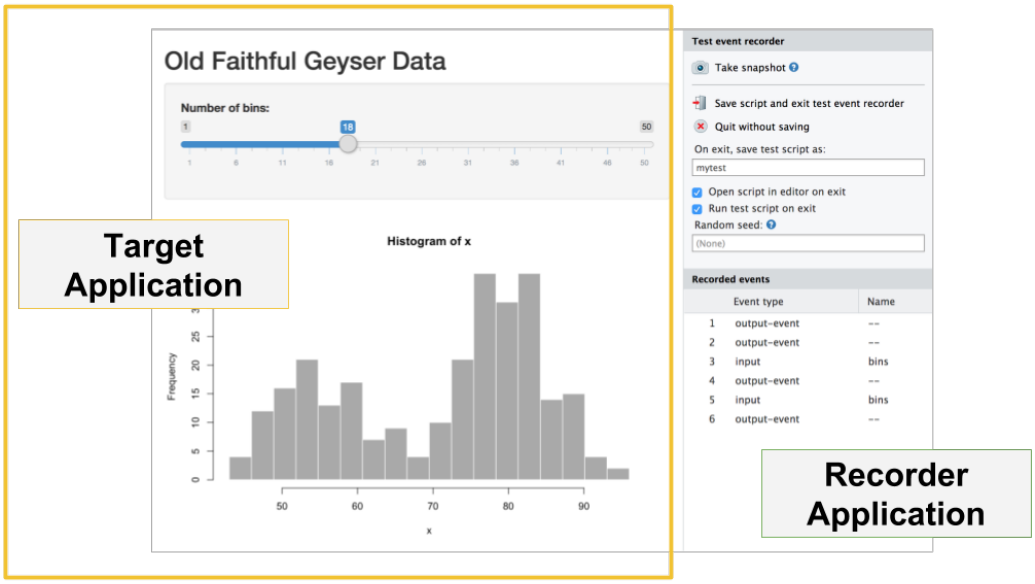


Figure 4.4: Target and Recorder App iframe

The panel on the right displays some controls for the test recorder, as well as a list of **recorded events**. As you interact with the target app, you will see those interactions appear in the recorded events list.

For testing a Shiny application, interacting with the inputs is only one part of the equation. It's also necessary to check that the application produces the correct outputs. This is accomplished by taking **snapshots** of the application's state.

To take a snapshot of the application's state, click the *Take snapshot* button on the recorder app. This will record all input values, output values, and exported values.

Running Tests

When you quit the test recorder, it will automatically run the test script. There are three separate components involved in running tests:

1. First is the **test driver**. This is the R process that coordiates the testing and controls the web browser. When working on creating tests interactively, this is the R process that you use.
2. Next is the **Shiny process**, also known as the **server**. This is the R process that runs the target Shiny application.
3. Finally, there is the **web browser**, also known as the **client**, which connects to the server. This is a headless web browser - one which renders the web page internally, but doesn't display the content to the screen (PhantomJS).

So, when you exit the test recorder, it will by default automatically run the test script and print something like this:

```
Saved test code to /path/to/app/tests/mytest.R
Running mytest.R
====== Comparing mytest ...
  No existing snapshots at mytest-expected/. This is a first run of tests.

Updating baseline snapshot at tests/mytest-expected
```

Figure 4.5: View Failed Tests

```
Renaming tests/mytest-current
      => tests/mytest-expected.
```

This is the result of running `testApp()`, which can also be manually run by providing the desired application and test like this:

```
testApp("exampleApp", "mytest")
```

The built-in integreation with RStudio v1.2 provides `Run Tests` as a drop down menu option in your Shiny app source file (see it located under the *Record Test* option in the screenshot above).

Subsequent Test Runs

After the initial test run, you can continue to run the tests to check for changes in application behavior.

If there are any differences between current and expected results, the test output will look something like this:

```
Running mytest.R
====== Comparing mytest ...
  Differences detected between mytest-current/ and mytest-expected/:

    Name         Status
    001.json  != Files differ
    001.png   != Files differ
Would you like to view the differences between expected and current results [y/n]?
```

To view failed tests in the RStudio IDE, go to the *Build tab* and make sure the *issues toggle* is selected:

For each test with different results, you can see the differences between the expected and current results.

Testing Code

The `shinytest` package was created for testing Shiny applications on the interaction-level. To test Shiny code and functions, we suggest using the `testthat` package.

Info GitHub

Learn about testing and how to setup test workflow and structure: R packages by Hadley Wickham

4.2 Activity: `shinytest`

First: Open app.R and use 'Record Test'

Discussion:

Understanding shinytest

- What does the recording file create?
- What challenges might our app pose to testing?
- Can you run the tests and get a success?
- How might you automate tests?

Deliverable: Try `shinytest`

- A set of tests that can catch unintentional changes to our app.

Chapter 5

Profiling: "The most important thing"

Profvis is a tool for helping you to understand how R spends its time. It provides a interactive graphical interface for visualizing data from `Rprof`, R's built-in tool for collecting profiling data.

- Profvis Home

To install profvis from CRAN: `install.packages("profvis")`

The RStudio IDE includes integrated support for profiling with profvis. These features are available in the current Preview Release of RStudio.

Profvis: Profiling tools for faster R code

- How do I make my R code faster?
- Why is my R code slow?

Each block in the flame graph represents a call to a function, or possibly multiple calls to the same function. The width of the block is proportional to the amount of time spent in that function. When a function calls another function, another block is added on top of it in the flame graph.

The profiling data has some limitations: some internal R functions don't show up in the flame graph, and it offers no insight into code that's implemented in languages other than R (e.g. C, C++, or Fortran).

Profvis is interactive. You can try the following:

- As you mouse over the flame graph, information about each block will show in the info box.
- Yellow flame graph blocks have corresponding lines of code on the left. (White blocks represent code where profvis doesn't have the source code – for example, in base R and in R packages. But see this FAQ if you want package code to show up in the code panel.) If you mouse over a yellow block, the corresponding line of code will be highlighted. Note that the highlighted line of code is where the yellow function is called from, not the content of that function.
- If you mouse over a line of code, all flame graph blocks that were called from that line will be highlighted.
- Click on a block or line of code to lock the current highlighting. Click on the background, or again on that same item to unlock the highlighting. Click on another item to lock on that item.
- Use the mouse scroll wheel or trackpad's scroll gesture to zoom in or out in the x direction.
- Click and drag on the flame graph to pan up, down, left, right.
- Double-click on the background to zoom the x axis to its original extent.

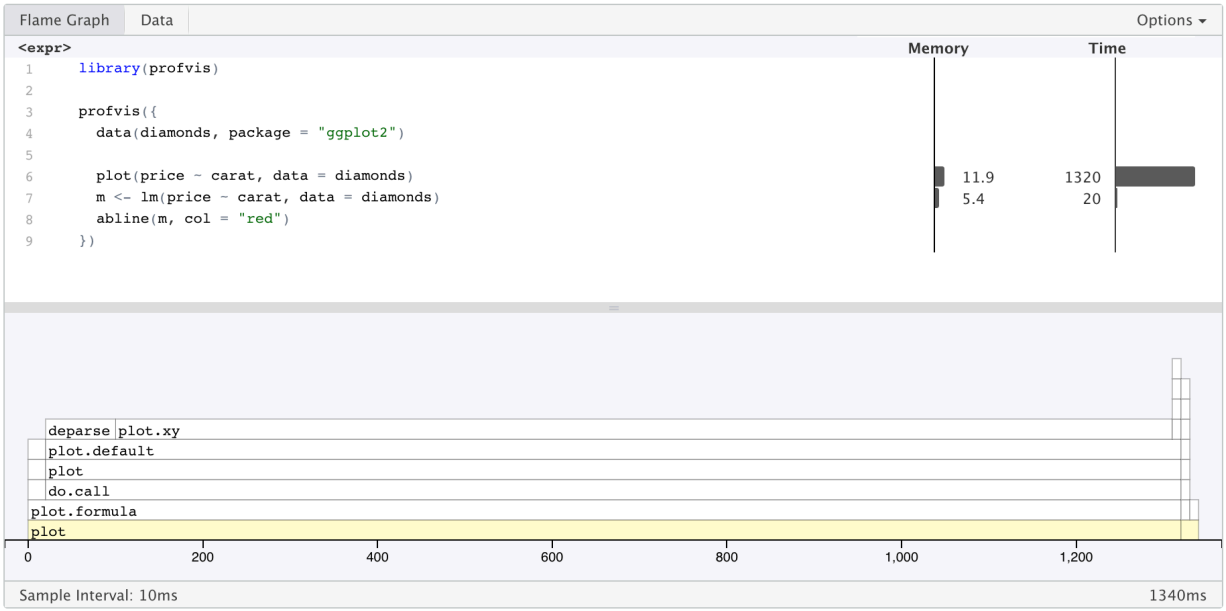


Figure 5.1: Profvis in Use

- Double-click on a flamegraph block to zoom the x axis the width of that block.

Things to remember:

- Understand how the sampling profiler works
- Understand the `profvis` interface
- Sometimes performance bottlenecks are counterintuitive!

5.1 Activity: Profiling

First: Create a profile of our app

Discussion:

Understanding app performance

- Where does our app spend most of its time?
- Do any parts of the profile surprise you?
- If you test the app again, do you get the same results?

Additional Group Discussion

- Is the call to `gt` slow?
- Why does `getStudentNum` take so much longer than `getStudentBin`? How could we speed this up?

Deliverable: Optimization Recommendations

- One recommendation for how we could speed up our app.

References and Resources:

- Make Shiny Fast by Alan Dipert
- Profvis Webinar Slides
- Shiny Dev Center Article: Profiling your Shiny app

Chapter 6

Deployment

6.1 RStudio Connect

> It doesn't matter how great your analysis is unless you can explain it to others: you need to communicate your results. - Grolemund & Wickham in *R For Data Science*

RStudio Connect connects you and the work you do in R with others as never before. Only RStudio Connect provides:

- "One button" deployment for any Shiny application, R Markdown document, or any static plot or graph to a single environment.
- The ability to manage and limit access to the work you've shared with others - and easily see the work they've shared with you.
- "Hands free" scheduling of updates to your documents and automatic email distribution.

RStudio Professional Quickstart (VirtualBox VM Download - Available Jan. 20)

- User Guide: Connecting
- User Guide: Publishing

6.2 Activity: Inital Deployment

First: Login to RStudio Connect

Discussion:

Pre-deployment Brainstorm

- What is our goal in deploying this code?
- What does our code depend on locally?
- What needs to deploy with our code?

Deliverable: Deployed App

Press the publish button in RStudio:

- Link to your Connect account

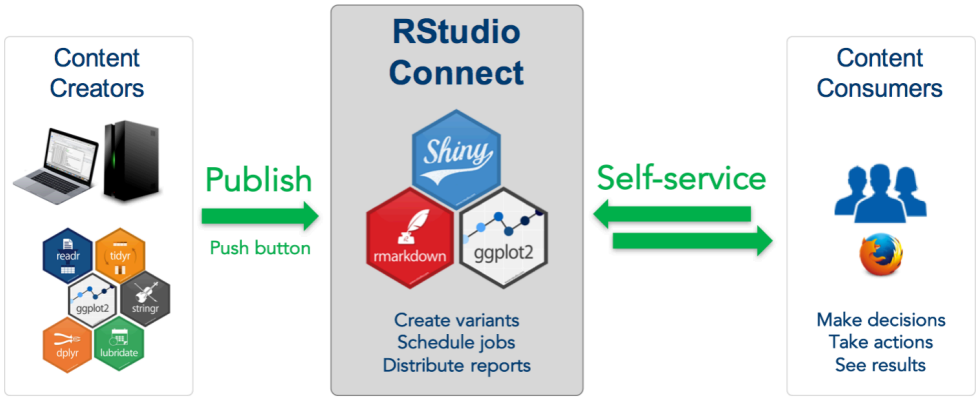


Figure 6.1: RStudio Connect

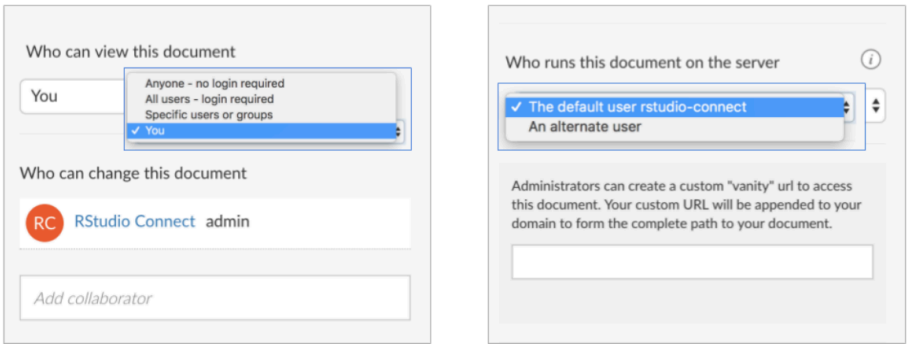


Figure 6.2: Access Settings Options

- Select the files to publish (do NOT use the default to publish all the files)
- Poke around at the results

Note: We are expecting an error!

6.3 Application Settings

6.3.1 Access

After publishing a piece of content to RStudio Connect, the Connect user interface will open to show the access panel.

There are three types of user interaction settings available to publishers in this panel:

- Viewer Access (who can see this content)
- Collaborator Access (who can change this content)
- Execution User (which server user will run the content)

The final setting for creating a vanity URL is reserved for administrators only.

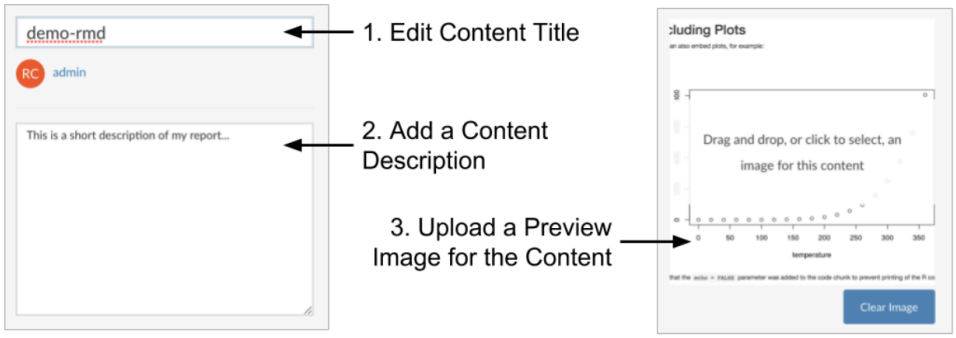


Figure 6.3: Metadata and Info Settings

6.3.2 Metadata

After your RStudio Connect content has been published, you can change its title, add a description, and upload a content image to make it easier for others to find your content. First, open the content and click on the "info" tab.

The top text field is the application title. This defaults to the application name if it was given. Other users will not be able to edit this field unless they are collaborators or administrators. The second, larger text area is the application description. You can describe the content in this field, or add other information that you feel is important. Other users will not be able to edit this field unless they are collaborators or administrators.

The area that displays a prompt to select an image for this content is the content image field. You can upload a jpeg, png, gif, or svg image that represents the content. Unlike title or description, users who cannot view the content (including administrators) will also be unable to view the image you have uploaded. This way, if you wish to upload a screenshot of the content, any information contained in that screenshot will not leak to users who would not otherwise be able to access it.

6.3.3 Logs

The logs panel allows you to view a history of jobs and snapshots as well as view and download the latest content logs. Content logs can be especially useful for debugging a failed deployment or unexpected behavior.

Figure 6.4: Application Logs

Chapter 7

Connecting to Data in Production

- Key Resource: db.rstudio.com

7.1 The `config` package

7.2 Environment Variables

When developing content for RStudio Connect, you should never place secrets (keys, tokens, passwords, etc.) in the code itself. Best practices dictate that this kind of sensitive information should be protected through the use of environment variables or another method of configuration such as the config package.

The Vars panel makes it easy to define environment variables which are then exposed to the processes executing your content. Note that there is no way to define environment variables prior to publishing content. If your content code relies on environment variables, publish it in an initial 'broken' state, then add the environment variables through this pane before sharing or testing the content.

Click on the Add Environment Variable button, then provide a name and value for your environment variable. For security reasons, once you add a variable, the value will be obscured and cannot be edited. You can always delete a variable and create a new one with the same name.

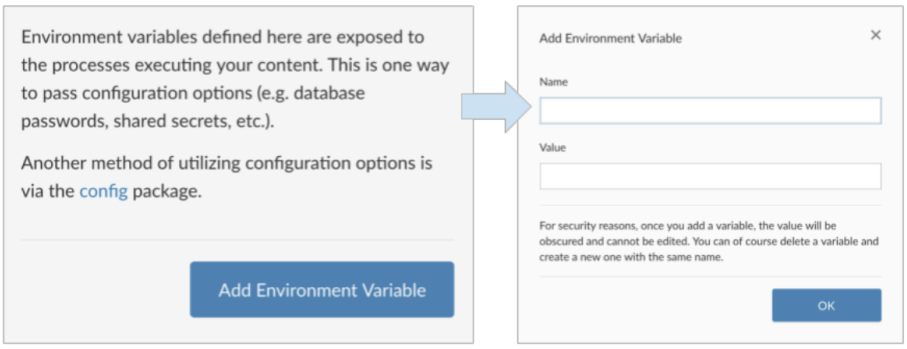


Figure 7.1: Environment Variables in RStudio Connect

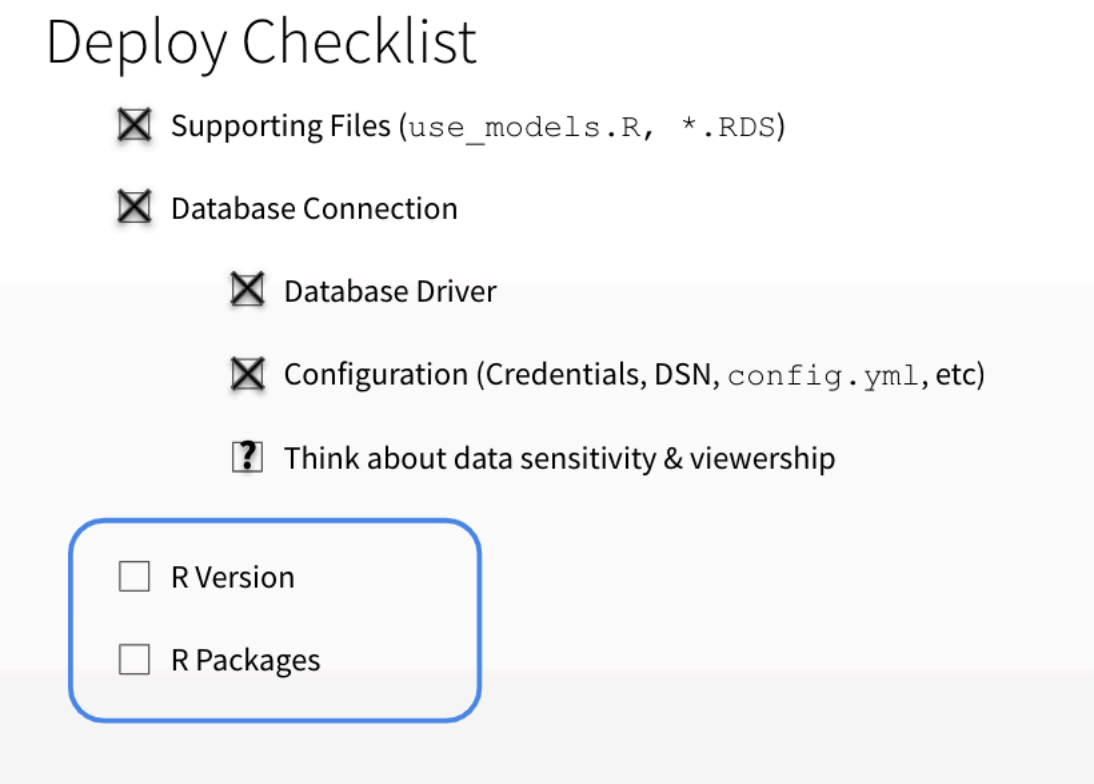


Figure 7.2: Database Checklist

7.3 Activity: Databases

First: Read This!

Discussion:

Data Management

- How does your organization connect to data?
- What data are we exposing to viewers? Is it appropriate for all viewers?
- What else could we manage with the `config` package?

Deliverable: Running, Deployed App

- Update config.yml and Redeploy
- Edit the Environment Variables in RStudio Connect

Revisit the Deployment Checklist

Chapter 8

Load Testing

Load testing helps developers and administrators estimate how many users their application can support. If an application requires tuning, load testing and load test result analysis can be used to identify performance bottlenecks and to guide changes to infrastructure, configuration, or code.

It's a common misconception that *"Shiny doesn't scale"*. In actuality, properly-architected Shiny applications can be scaled horizontally, a fact which Sean Lopp was recently able to demonstrate at rstudio::conf 2018. We used `shinycannon` to simulate 10,000 concurrent users interacting with an application deployed to AWS. You can see a recording of Sean's talk and the load test demonstration here: Scaling Shiny

To perform a load test you'll need two pieces of software: `shinyloadtest` and `shinycannon`.

- `shinyloadtest` is an R package used to generate recordings and analyze results. You should install it on your development machine. GitHub
- `shinycannon` is the command-line replay tool. You can install it on your development machine for testing, but for best results we recommend installing it on a server, and preferably not the one the application under test is also on. GitHub

See the Load Testing Quickstart Here.

8.1 Optimization Loop Methodology

- **Benchmark:** Use `shinyloadtest::record_session()` to record interaction, `shinycannon` to simulate users
- **Analyze:** Visualize and interpret the metrics
- **Recommend:** Propose ways for the capacity of the application to be increased
- **Optimize:** Implement recommendations and benchmark again. Repeat until satisfied

Workflow:

- Use 'shinyloadtest to record a session with a Shiny app
- Generate load with `shinycannon`
- Analyze metrics with `shinyloadtest`
- Make changes to the Shiny application
- Generate load and analyze again

Load Testing Shiny Applications

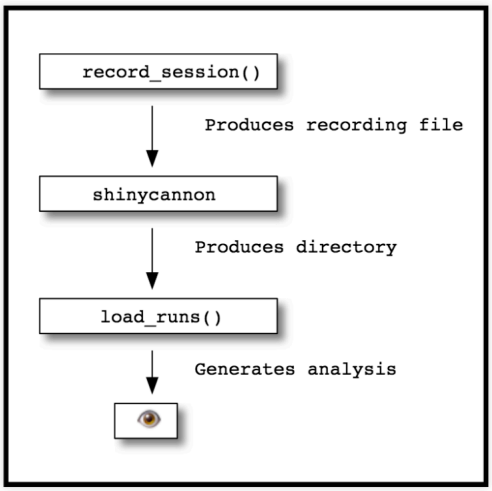


Figure 8.1: Load Test Workflow

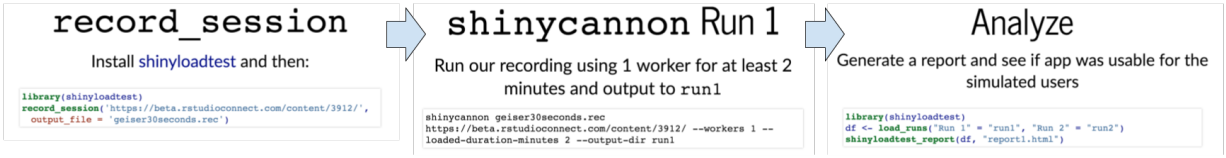


Figure 8.2: Load Test Steps

8.2 Activity: Load Testing

First: Open runloadtest.R,do the pre-run checklist

Discussion:

Preparing for the load test

- First, what is up with the pre-run checklist? Any idea why these steps are (currently) necessary?
- The `runloadtest.R` file is going to start with a baseline test of 1 and then a test of 25. Why is the baseline important?

Deliverable: Run the load test

- Follow the first set of commands in `runloadtest.R`
- How did the experience for 1 user compare to the experience for 25 users?

References and Resources:

Webinar:

- Load testing Shiny by Alan Dipert
- Webinar Slides

Vignettes:

- Analyzing Load Test Logs
- Case Study: Scaling an Application
- Limitations of `shinyloadtest`
- Load Testing Authenticated Apps

Chapter 9

Plot Caching

Blog: Shiny 1.2.0: Plot caching - November 18, 2018

The Shiny 1.2.0 package release introduced *Plot Caching*, an important new tool for improving performance and scalability in Shiny applications.

In a nutshell: The term "caching" means that when time-consuming operations are performed, we can save (cache) the results so that the next time that operation is requested, instead of re-running that calculation, we instead go fetch the previously cached result. When applied appropriately, this "fetch" operation should take less time that the original calculation and improve application performace (and user experience) overall.

Shiny's reactive expressions do some amount of caching by default, and you can use more explicit techniques to cache various data operations. Examples include use of the `memoise` package, or manually saving intermediate data frames to disk as CSV or RDS.

Plots are a very common and (potentially) expensive to compute type of output object in Shiny applications, which makes them a great candidate for caching. In theory, you could use `renderImage` to accomplish this, but because Shiny's `renderPlot` function contains a lot of complex infrastructure code, it's actually quite a difficult task.

Shiny v1.2.0 introduces a new function, `renderCachedPlot`, that makes it much easiter to add plot caching to your application.

9.1 When to use plot caching

A shiny app is a good candidate for plot caching if:

1. The app has plot outputs that are time-consuming to generate
2. These plots are a significant fraction of the total amount of time the app spends thinking
3. Most users are likely to request the same few plots

9.1.1 Using `renderCachedPlot`

The following example is a simple, but computationally expensive, plot output:

```
output$plot <- renderPlot({
  ggplot(diamonds, aes(carat, price, color = !!input$color_by)) +
    geom_point()
})
```

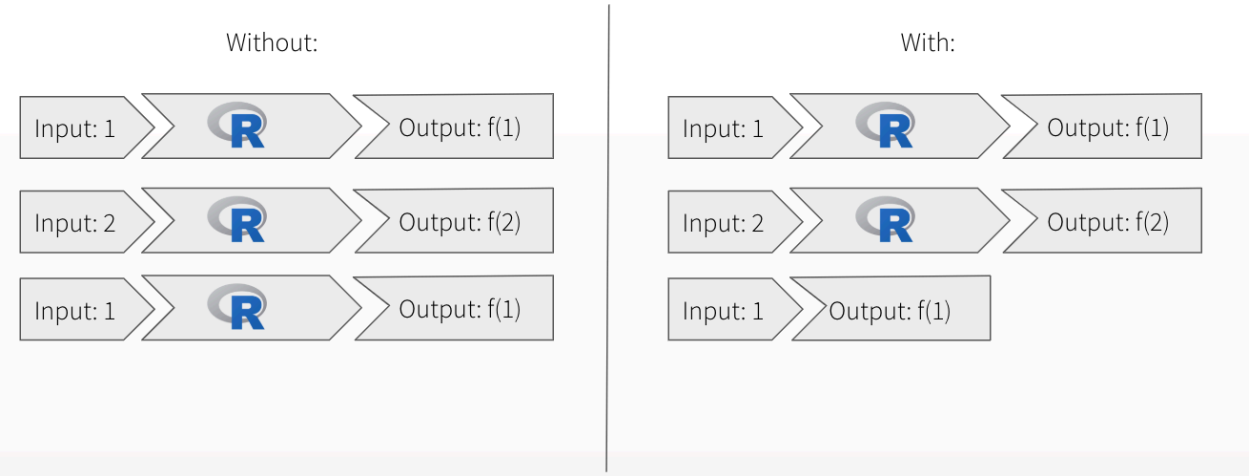


Figure 9.1: renderChachedPlot()

The `diamonds` dataset has 53,940 rows. This plot might take roughly 1580 milliseconds (1.58 seconds) to generate depending on the resources available. For a high traffic Shiny application in production, 1.58 seconds is likely slower than ideal.

Plot caching can be implemented in two steps:

1. Change `renderPlot` to `renderCachedPlot`
2. Provide a suitable `cacheKeyExpr`. This is an expression that Shiny will use to determine which invocations of `renderCachedPlot` should be considered equivalent to each other. In the example case, two plots with different `input$color_by` values can't be considered the "same" plot, so the `cacheKeyExpr` needs to be `input$color_by`

The example plot code would be updated like this:

```
output$plot <- renderCachedPlot({
  ggplot(diamonds, aes(carat, price, color = !!input$color_by)) +
    geom_point()
}, cacheKeyExpr = { input$color_by })
```

With these code changes, the first time a plot with a particular `input$color_by` is requested, it will take the normal amount of time. But the next time it is requested, it will be almost instant, as the previously rendered plot will be reused.

9.2 Activity: Plot cache benchmarking

First: Update your app code to use `renderCachedPlot`

Discussion: *caching + shinyloadtest*

- How will introducing `renderCachedPlot` affect our load test experience?

What is the performance comparison between tests?

Deliverable: Re-run Load Test

- Redeploy the version of the app with `renderCachedPlot`
- Re-run the load test and compare the outputs (continue to follow along with `runloadtest.R`)

9.3 Extended Topics

- Shiny Article: Plot Caching by Winston Chang

Plot Caching on RStudio Connect

Shiny can store cached plots in memory, on disk, or with another backend like Redis. There are also a number of options for limiting the size of the cache. Applications deployed on RStudio Connect should use a disk cache and specify a subdirectory of the application directory as the location for the cache. To do so, add this code to the top of your application:

```
library(shiny)
shinyOptions(cache = diskCache("./cache"))
```

This option ensures that cached plots will be saved and used across the multiple R processes that RStudio Connect runs in support of an application. In addition, this configuration results in the cache being deleted and reset when new versions of the application are deployed.

Chapter 10

Scaling

Application Scaling 101

10.1 RStudio Connect Performance Settings

Scaling and Performance Tuning in RStudio Connect

- Article Reference

RStudio Connect is built to scale content. Publishers and administrators have access to runtime settings to help tune and scale their applications and APIs. The primary concern for scaling content in RStudio Connect is understanding when and how R code will be executed on the server.

Content published on RStudio Connect can broadly fall into two categories:

1. Static / Batch Content

Static content is content a user can visit without requiring a running process. Examples include PDF documents, plots, and HTML files. HTML files can include interactive elements where the interactivity occurs on the client. RStudio Connect is able to update static content on a schedule.

2. Interactive Content

A powerful component of RStudio Connect is the ability to host data products that require backend processes during a user's visit. Examples include shiny applications and R Markdown documents with runtime::shiny, plumber APIs, and TensorFlow APIs. In the case of Shiny, an end user's browser (the client) is connected with an R process running on the server. When a user changes an input, the client sends a message to the server and a portion of R code is re-run. The server sends back the result as output. In the case of APIs, a client makes a request which is sent to a running process, and the results are sent back to the client.

For Interactive Content, RStudio Connect enables scaling through parameters that determine the content's scheduler.

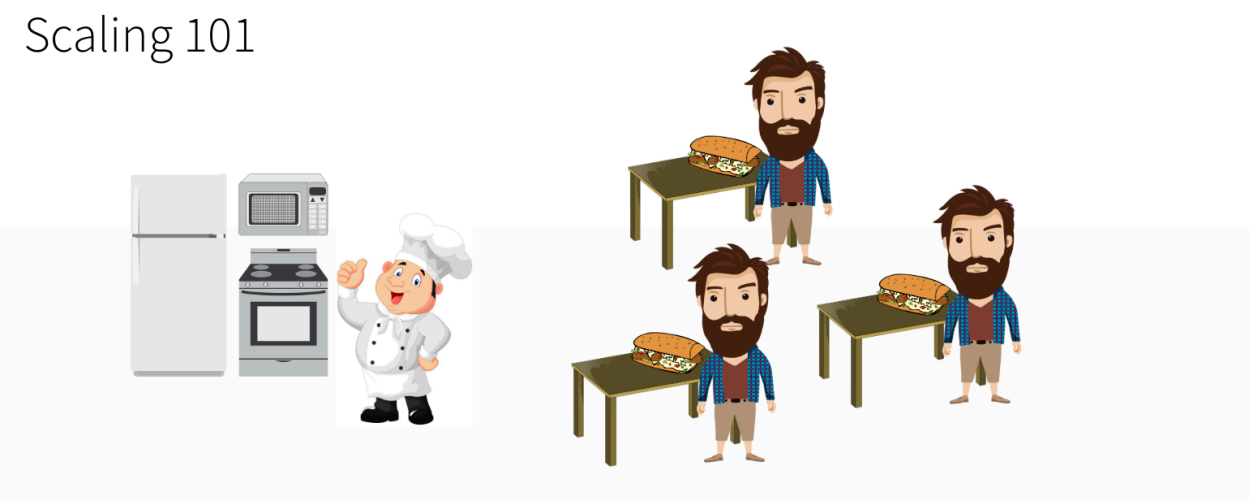


Figure 10.1: Kitchen Scaling

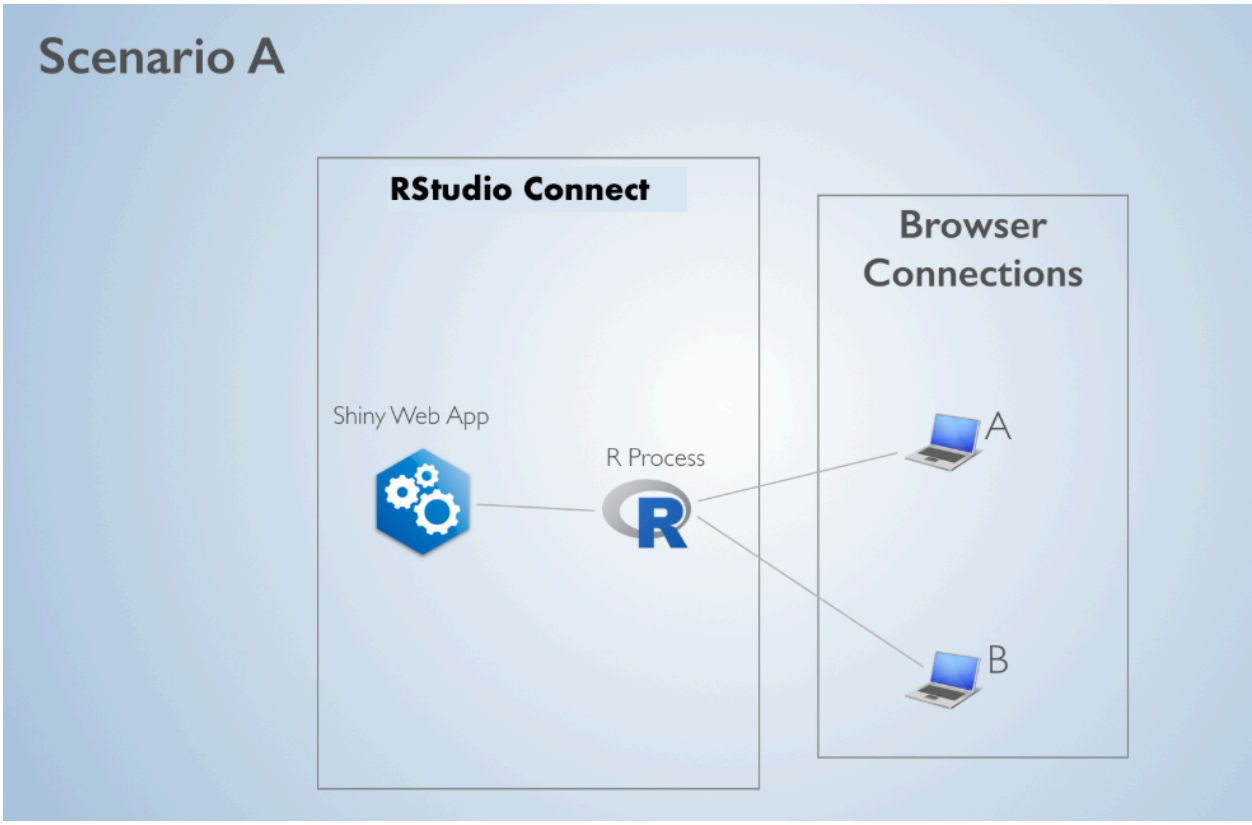


Figure 10.2: Scaling A

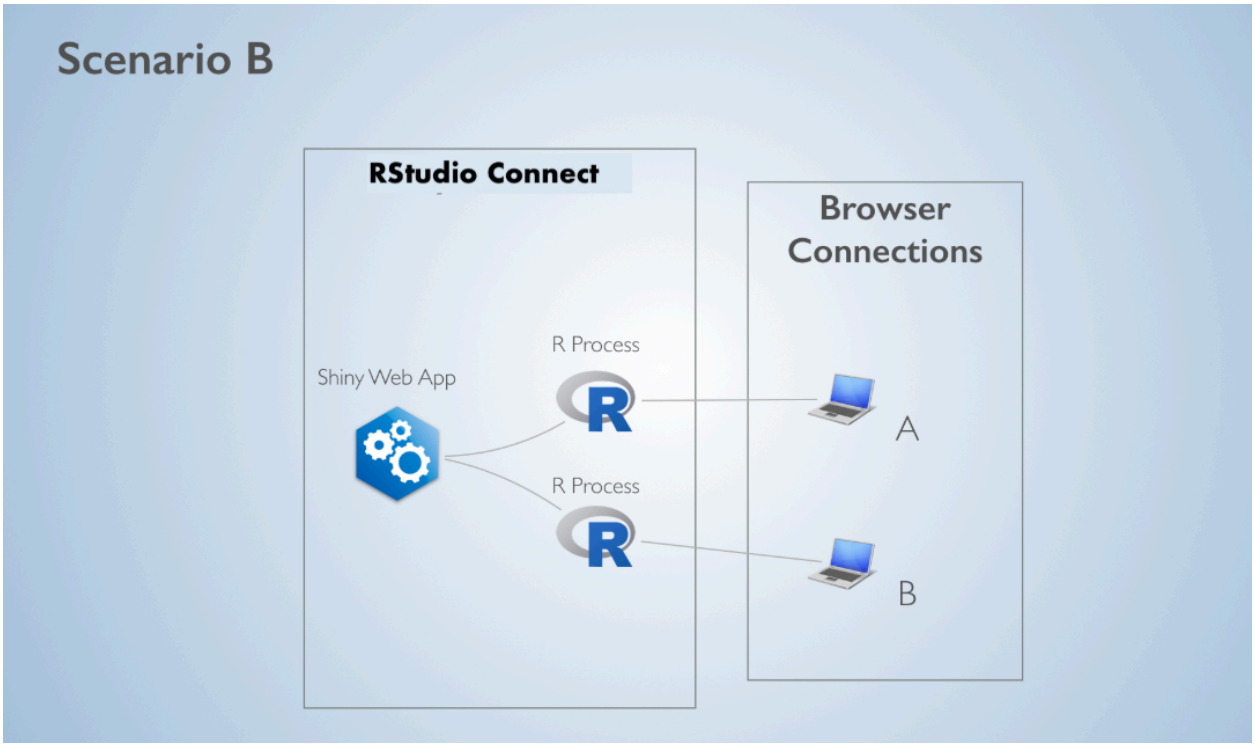


Figure 10.3: Scaling B

10.1.1 Content Scheduler

When a user requests content with Shiny components, RStudio Connect opens a channel between the user and an R process. Now, suppose a second user requests the same content. There are two potential scenarios:

In Scenario A, both users are linked to the same R process. Because R is single-threaded, if user A and user B both change an input and trigger R calculations, their requests will be handled sequentially. (User B will have to wait for user A's calculation to complete, and then for their own calculation to complete before they will see an updated output).

In Scenario B, Connect will link each user to their own R process. If user A and user B both change an input, their calculations will happen simultaneously.

Why wouldn't RStudio Connect always select Scenario B? The answer has to do with memory and initial load time. When 2 users are connected to the same R process, they get to share everything that is loaded outside of the server function.

To see this, consider when the different pieces of Shiny application code are executed:

This works because Shiny makes use of R's unique scoping rules (read more here). In Scenario B, all of the shiny code has to be re-run, including loading any globally available data. This means the memory usage is 2x what it would be in Scenario A. Additionally, spinning up an R process and executing all of the shiny code takes time. While the application is more responsive to both users after the web page is loaded, it will take longer for them to connect initially.

The scheduling parameters tell RStudio Connect to act somewhere in between Scenario A and Scenario B, to maximize the trade-off between app responsiveness and memory consumption/load time. The parameters also specify how long R processes and user connections should remain idle before timing out.

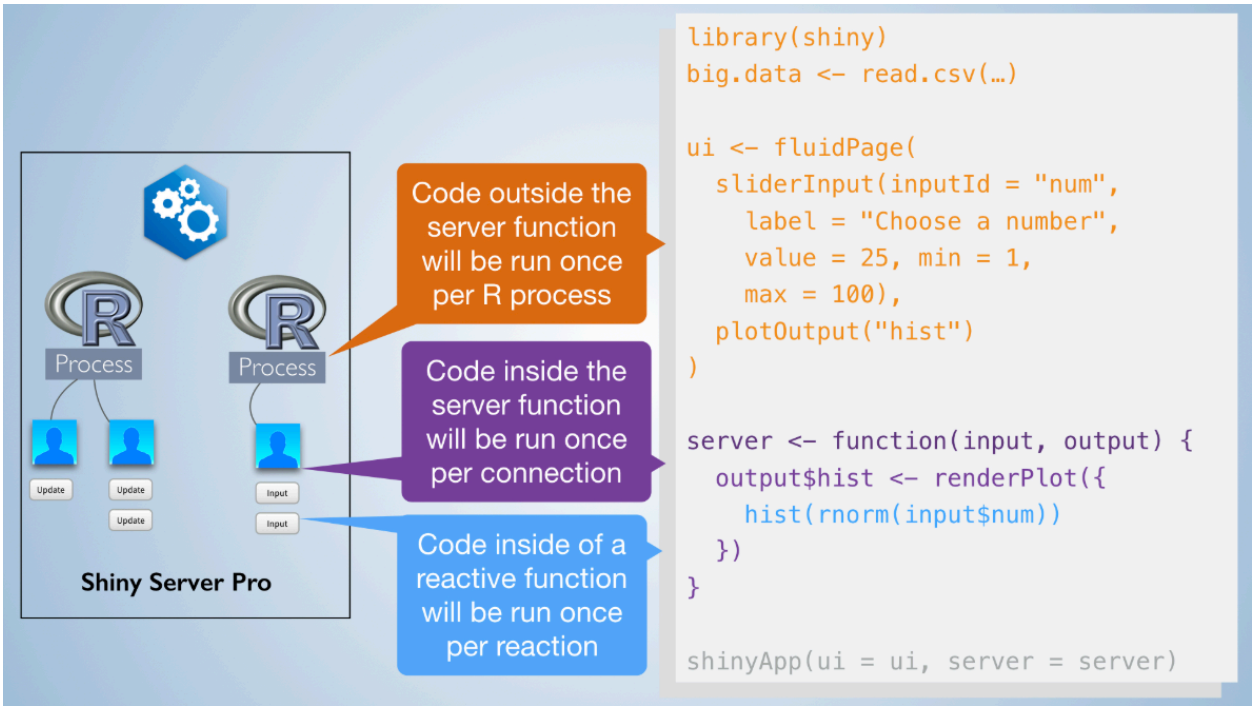


Figure 10.4: App Execution

10.1.2 Scheduling Parameters

Max processes - Determines the maximum number of processes that will be created. Max processes x Max connections per process = total number of concurrent connections. Default value is 3.

Pick a value that will support the expected number of concurrent users or requests.

Min processes - Determines the minimum number of processes that will always be running. For Shiny applications and plumber APIs, user requests only execute specific functions. R code outside of those functions can be run when the process starts before user requests are made, which can dramatically speed up response time.

Pick a number close to Max processes if your application or API pre-loads a large amount of data to be shared by every user. Pick a small number to minimize the amount of memory consumed.

Figure 10.5: Runtime Settings Options

Max connections per process - The maximum number of connections per process. Default value is 20.

Pick a small number if your content involves heavy computation. Pick a larger number if your content shares data between users. (e.g. Pick a large number if your content takes a long time to load, but after loading is very fast.)

Load factor - Determines how aggressively new processes will be created. A value close to 0 means new processes will be spun up aggressively to try and keep the number of connections per process small. A value close to 1 means the number of connections per process will be close to max connections. Default value is 0.5.

Pick a small number if your content loads quickly but involves expensive computation. Pick a number closer to 1 if your content loads slowly, but after loading is fast OR if you want to minimize the amount of memory.

10.2 Activity: Runtime Settings

First: Open the runtime settings for your app

Discussion:

- What do each of the settings mean?
- What changes to the settings might affect our app?

Deliverable: Re-run Load Test

Test your hypothesis by changing the runtime settings and then re-run the load test. Compare the results to the prior test (continue to follow along with `runloadtest.R`)

10.3 Activity: Admin Dashboard

First: Open the admin dashboard in RStudio Connect

Discussion:

Multi-Tenant Environment

- What are the tradeoffs for modifying the scaling settings?
- What else might you need to consider when sharing infrastructure?
- How does these concerns change over time?

Deliverable: Update checklist

Create one pre-deployment checklist step and one every-2-months checklist step based on your discussion.

10.4 Extended Topics

10.4.1 High Availability and Horizontal Scaling

Other considerations for widely accessed content are high availability and horizontal scaling. Both require content to be hosted by more than one server. RStudio Connect supports a cluster setup.

10.4.2 R Markdown Documents with runtime::shiny

R Markdown documents are a bit different. In essence, an Rmd with runtime::shiny specified places everything (data loading, UI creation, etc) inside of the server function. The implication is an Rmd with runtime::shiny will always consume more memory as users connect.

Prerendered Shiny Documents alleviate this problem.

Chapter 11

Alternatives to Shiny

11.1 Plumber

Could our student "scoring" be hosted outside of the Shiny app?

11.1.1 Intro to Plumber

Resources:

Plumber is an R package that converts your existing R code to a web API using a handful of special one-line comments.

> What are Web APIs? For some, APIs (Application Programming Interface) are things heard of but seldom seen. However, whether seen or unseen, APIs are part of everyday digital life. In fact, you've likely used a web API from within R, even if you didn't recognize it at the time! Several R packages are simply wrappers around popular web APIs, such as `tidycensus` and `gh`. Web APIs are a framework for sharing information across a network, most commonly through HTTP.

You can install the latest stable version from CRAN using the following command:

```
install.packages("plumber")
```

These comments allow plumber to make your R functions available as API endpoints. You can either prefix the comments with `#*` or `#'` but we recommend the former since `#'` will conflict with the Roxygen package.

Basic example:

1. Create a File: `plumber.R`

```
# plumber.R

#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg=""){
  list(msg = paste0("The message is: '", msg, "'"))
```

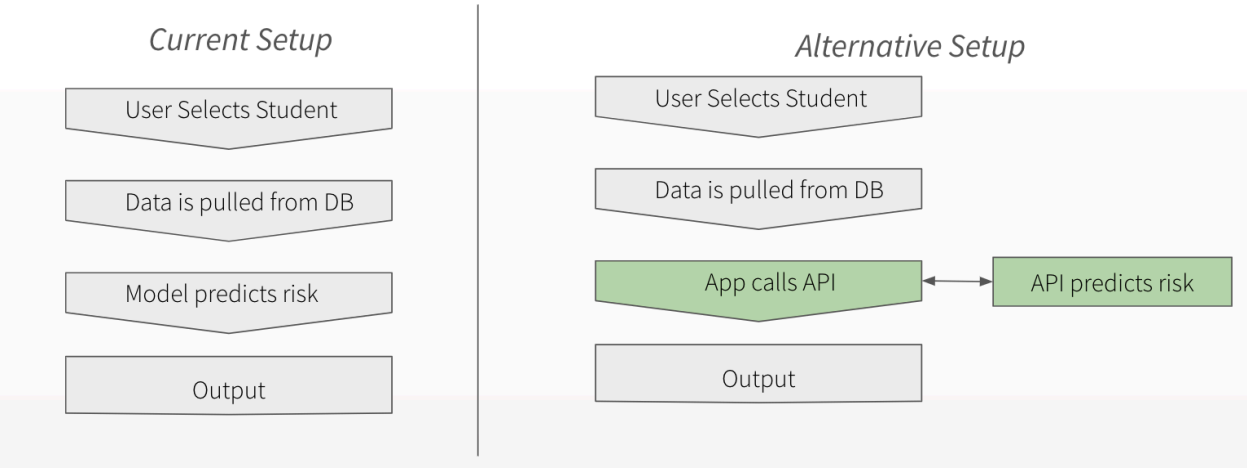


Figure 11.1: Plumber Scoring

```
}

#* Plot a histogram
#* @png
#* @get /plot
function(){
  rand <- rnorm(100)
  hist(rand)
}

#* Return the sum of two numbers
#* @param a The first number to add
#* @param b The second number to add
#* @post /sum
function(a, b){
  as.numeric(a) + as.numeric(b)
}
```

2. Serve the `plumber.R` file from the R console:

```
library(plumber)
r <- plumb("plumber.R")  # Where 'plumber.R' is the location of the file shown above
r$run(port=8000)
```

You can visit this URL using a browser or a terminal to run your R function and get the results. For instance http://localhost:8000/plot will show you a histogram, and http://localhost:8000/echo?msg=hello will echo back the 'hello' message you provided.

3. Hit the endpoints (example: use `curl` from mac/linux terminal)

```
$ curl "http://localhost:8000/echo"
 {"msg":["The message is: ''"]}

$ curl "http://localhost:8000/echo?msg=hello"
```

```
 {"msg":["The message is: 'hello'"]}

$ curl --data "a=4&b=3" "http://localhost:8000/sum"
 [7]
```

References and Resources:

- Reference: Plumber Docs
- Plumber Integration in RStudio 1.2
- RViews Blog: REST APIs and Plumber by James Blair
- Video: Turning your R code into an API by Jeff Allen
- Webinar: Plumbing APIs with Plumber by Jeff Allen

11.2 Activity: Plumber

First: Try creating your own plumber API in the IDE (New File)

Discussion:

Consider our new architecture

Does my R functionality need to be accessed by other systems?

- Why might a RESTful API be easier to scale than a Shiny app?
  - Requests are stateless, so it is easier for R processes to come and go.
- What benefits come from pulling the modeling code out of our app?
  - We can update the model independently of the app, e.g. if we wanted to retrain.

Deliverable:

- Add an item to our production checklist to consider plumber.
- Add 2 decision criteria to the checklist to decide when a plumber API would be useful.

11.3 R Markdown

R Markdown is an open-source R package that turns your analyses into high quality documents, reports, presentations and dashboards. An R Markdown document is written in markdown (an easy-to-write plain text format) and contains chunks of embedded R code. R Markdown documents are fully reproducible and support dozens of output formats including HTML, PDF, and Microsoft Word documents.

R Markdown files are designed to be used with the `rmarkdown` package. `rmarkdown` comes installed with the RStudio IDE, but you can acquire your own copy of `rmarkdown` from CRAN with the command:

```
install.packages("rmarkdown")
```

R Markdown reports rely on three frameworks:

1. `markdown` for formatted text
2. `knitr` for embedded R code

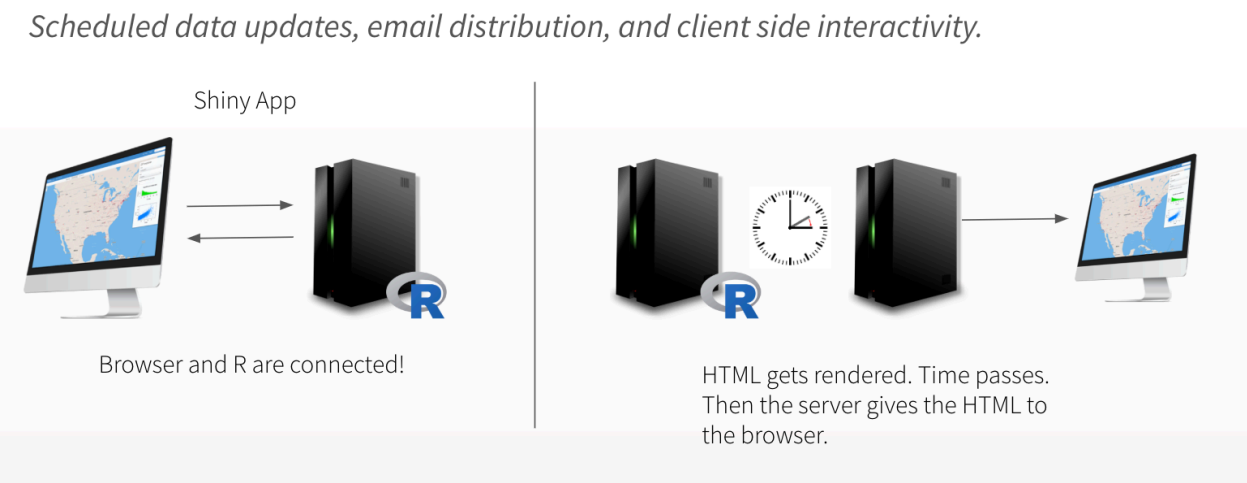


Figure 11.2: Publishing R Markdown

3. `YAML` for render parameters

To create an R Markdown report, open a plain text file and save it with the extension .Rmd. You can open a plain text file in your scripts editor by clicking File > New File > Text File in the RStudio toolbar.

Publishing R Markdown to RStudio Connect

- **Publishing Destination**

When publishing documents to RStudio Connect, you may encounter other deployment options depending on your content.

- RPubs documents are (1) always public, (2) always self-contained, and (3) and cannot contain any Shiny content.
- Choose "RStudio Connect" to publish to your RStudio Connect server.
- **Publish Source Code**

Publishing the document with source code means that your R Markdown file (.Rmd) will be deployed to RStudio Connect. This file is rendered (usually to HTML) on the server.

Publishing only the finished document means that the HTML file you rendered locally is deployed to RStudio Connect.

We recommend publishing your documents with source code, as it allows you to re-render the document with RStudio Connect (on a weekly schedule, for example). If the document cannot be rendered by RStudio Connect because of files or data sources that are unavailable on the server, choose "Publish finished document only" so others can view your work.

- **Document Selection**

It is possible to link together multiple R Markdown documents to make a multi-page document, so this is your chance to indicate that you've done this, and to publish all the documents at once. In most cases however you'll want to publish just the current document.

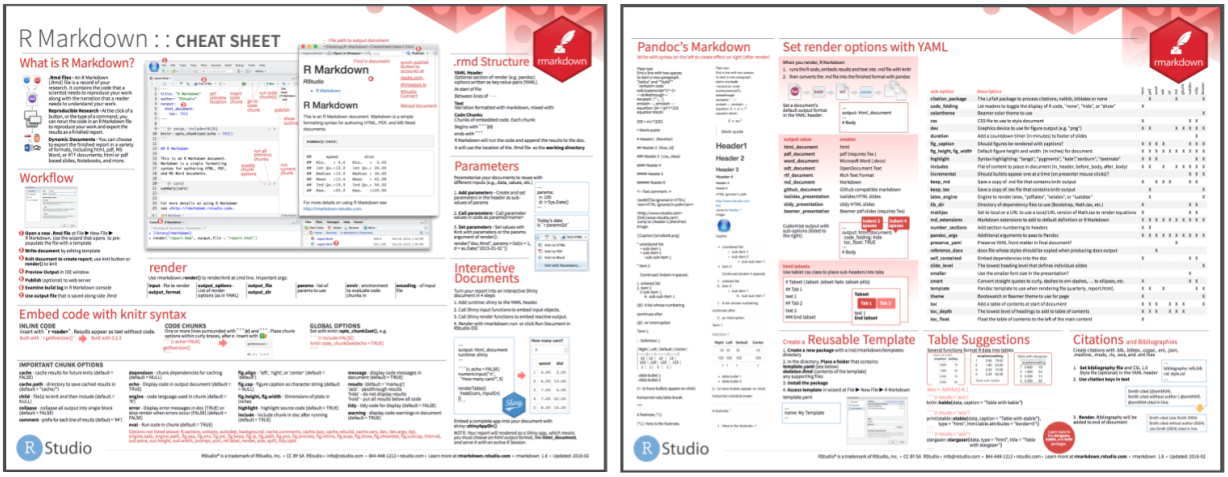


Figure 11.3: R Markdown Cheatsheet Preview

- RStudio Connect Publishing Guide
- R Markdown

References and Resources:

- R Markdown in the RStudio Connect User Guide
- R Markdown Reference Articles
- RViews Blog: Communicating results with R Markdown by Nathan Stephens
- RViews Blog: Enterprise Dashboards with R Markdown by Nathan Stephens

11.4 Activity: R Markdown

First: Create your own R markdown document (New File)

Discussion:

Consider R Markdown

- How does an R Markdown document scale?
- What types of interactivity can be added to R Markdown?

Deliverable:

- Create a decision matrix comparing Shiny to R Markdown

Alternatives *Criteria*	Shiny	R Markdown
# of Inputs	Users need multiple levels of dependent inputs to get an answer	All inputs can be specified up front
Update Frequency	Updates need to happen in real time	Updates occur on a regular basis
Output	Users come to the system to do exploratory Q&A, the outputs are the results	Users want a copy of the results or want results distributed to their email
Interactivity	Interactivity depends on lots of data	Interactivity can be based on data sent to the browser

Figure 11.4: Rmd Exercise

Chapter 12

DevOps Philosophy & Tooling

DevOps is a self-help philosophy for IT and software development teams that work together. It seeks to address a core problem: **The Fear of Taking Code into Production**

Software development and deployment often involves a separation of labor often referred to as a code deployment handoff. Managing handoffs can be painful and risky business when handled without care:

- Deployments/releases are high-risk events
- It takes a long time to deliver new features to customers
- There is no visibility into how code is deployed into production, therefore it is impossible to know what to optimize or fix in the system and/or people who seek to improve portions of workflow are likely not addressing true bottlenecks.
- Dev and Ops have seemingly conflicting goals and objectives

The Fix (ref: The DevOps Handbook) "The principles underpinning DevOps":

1. Accelerate the delivery of work from Development to Operations to customers
2. Commit to evolving ever safer systems of work through feedback
3. Promote a high-trust culture and embrace continual learning and experimentation (risk-taking) in daily work

> Even more important than daily work is the improvement of daily work –Mike Orzen, Lean IT

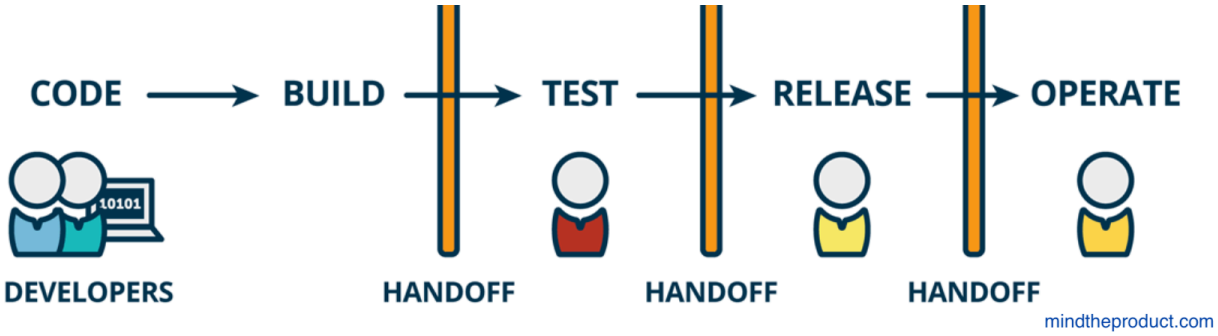


Figure 12.1: Deployment Handoff Cycle

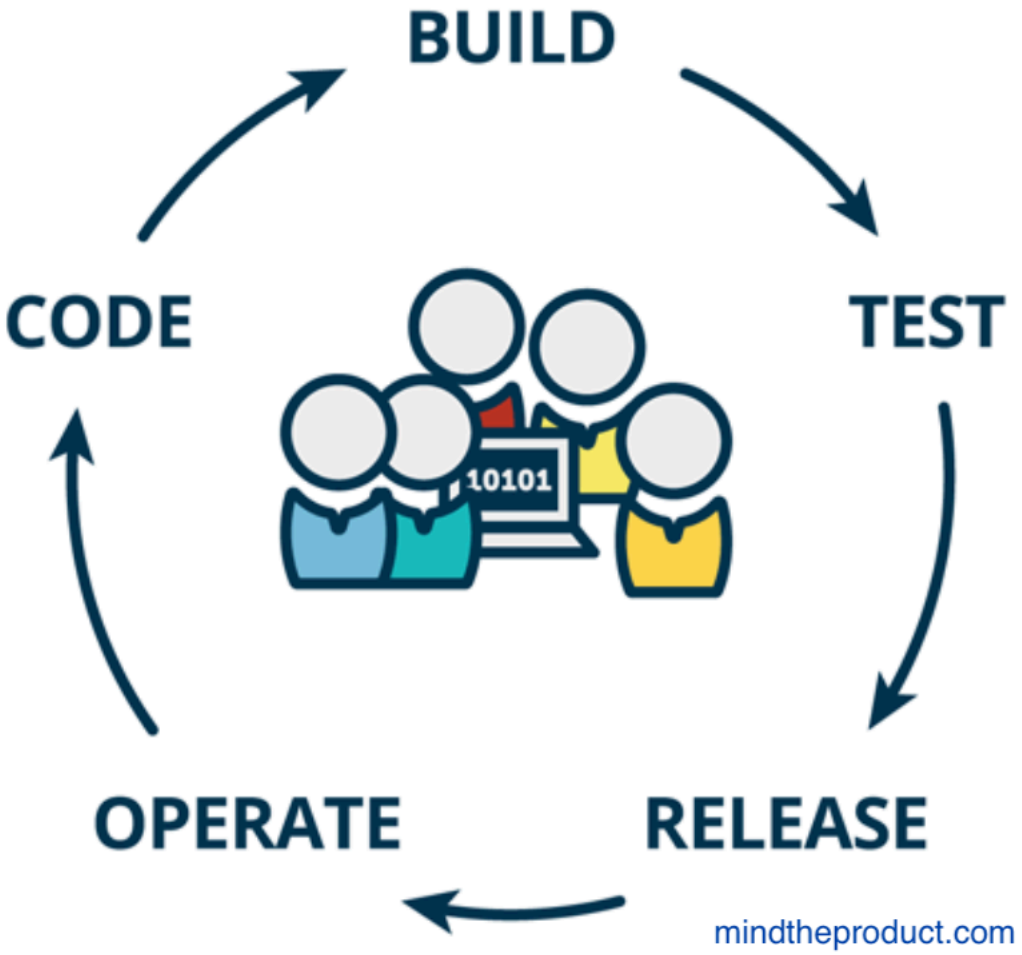


Figure 12.2: DevOps Cycle

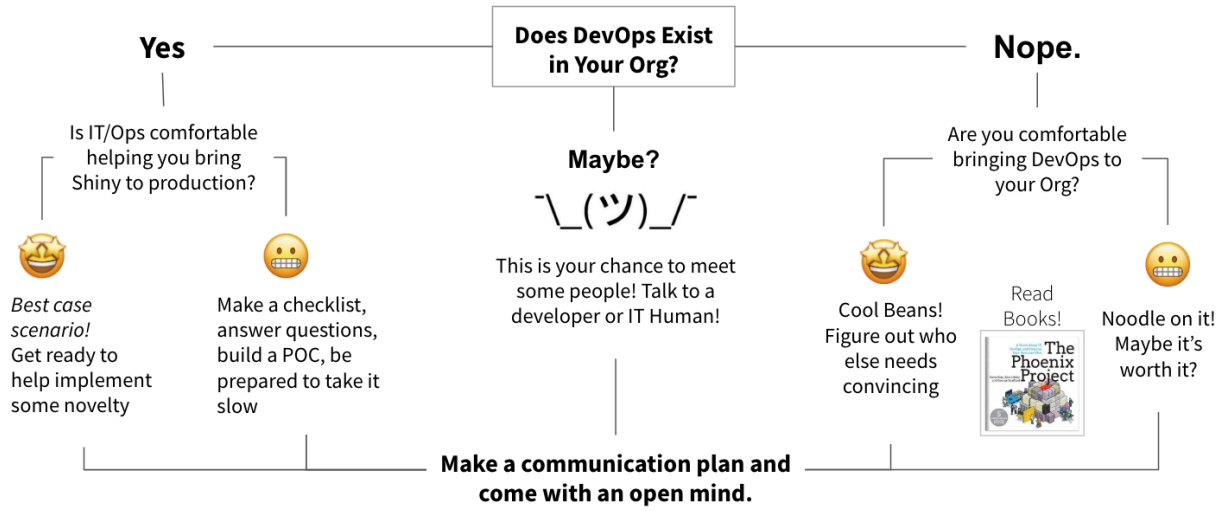


Figure 12.3: Delivering DevOps

12.1 Integrating Data Science and DevOps

12.1.1 How does data science fit in with the DevOps philosophy?

It's not *your job* to understand operations and systems administration. And it's not IT's job to understand R programming and shiny app development. But it takes shared goals and a little empathy from both sides to get less painful results.

We can leverage (steal) the DevOps philosophy and apply it to manage code deployment handoffs between data science and IT.

12.1.2 What do your dev, test and production environments look like?

- Do developers have access to production-like environments on their own workstations?
- Can these evnvironments be assembled on-demand?
  - Version Control
  - Infrastructure as Code

1. In what ways can analytic infrastructure for R change between development and production environments?
2. What processes or tools could help solve these issues?
3. Should development inform how a production environment is built, or should production inform how an application is developed?

12.1.3 Does code deployment feel like a high-risk operation?

Having developers focus on automating and optimizing the deployment process can lead to significant improvements in deployment flow.

- Smoke testing
- Build validation tools

- Environment consistency

Relate your Shiny deployment checklist to the pipeline an IT group might consider. Where is there risk? How can it be reduced?

Sample Ideas from The DevOps Handbook:

- Packaging code in ways suitable for deployment
- Creating pre-configured virtual machine images or containers
- Automating the deployment and configuration of middleware
- Copying packages or files onto production servers
- Restarting server, applications or services
- Generating configuration files from templates
- Running automated smoke tests to make sure the system is working and correctly configured
- Running testing procedures
- Scripting and automating database migrations

12.1.4 Can deployments be decoupled from releases?

- **Deployment** is any push of code to an environment (test, prod)
- **Release** is when that code (feature) is made available to users or customers

Deployment on demand and thoughtful release strategies allow more control (and more success) over the delivery of features to end users.

Environment-based Release Patterns:

- Blue-Green Deployment Pattern
- Canary and Cluster Immune System

Application-based Release Patterns:

- Feature Toggles
- Dark Launches

Which of these release patterns is most well-suited for Shiny Application development and deployment with the RStudio Connect publishing platform?

One Possibility: Feature Toggles (Dark Launch)

Shiny applications can access the username and groups of the current user through the session parameter of the shinyServer function.

Groups meta-data is populated when using password, or OAuth authentication. It is not populated with LDAP, PAM, or proxy authentication.

```
shinyServer(function(input, output, session) {
  output$username <- reactive({
    session$user
  })

  output$groups <- reactive({
    session$groups
  })
})
```

Your application could use this information to display customized messages or to enable functionality for a specific subset of users.

Chapter 13

Production Case Studies

Workshop Exercise: Case Study Discussions

First: Pick one case study

Discuss:

- What tools does your admin team use?
- How would you explain the goals of a shiny app to DevOps?

Deliverable:

- Whiteboard (in a medium of your choice) what an architecture might look like for your case study.
- Delineate who (and what tool) is responsible for code, R, R packages.

13.1 Case Study A: Dev/Test/Prod

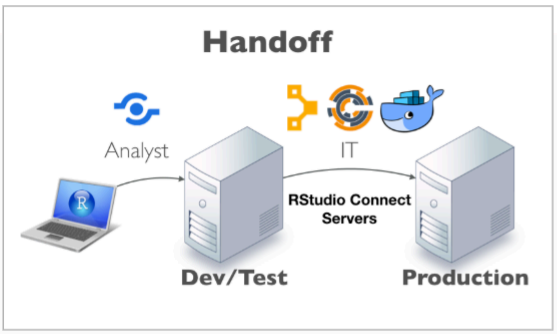
Before publishing an update to our app, we want to run user acceptance tests.

13.2 Case Study B: CI, Git, Chef

All of our production code must be deployed from Git.

13.3 Case Study C: Docker

We scale applications through Docker - how does Shiny fit in?

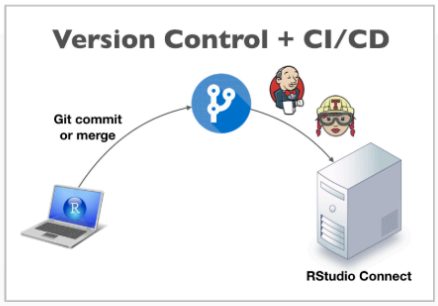


Code: Commited to Git by users after being deployed to Dev.

R: Multiple versions installed on server by admin's Ansible scripts.

R packages: Restored by RStudio Connect

Figure 13.1: Case A Considerations

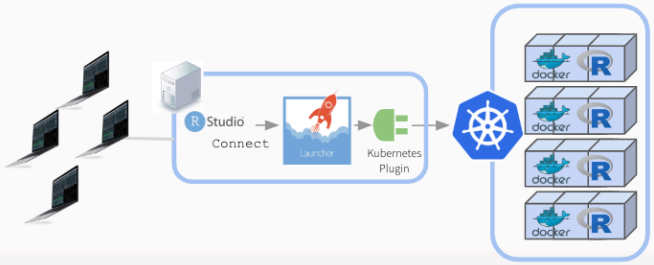


Code: Commited to Git by users. Deployed by Jenkins.

R: Multiple versions installed on server by admin's Ansible scripts.

R packages: Restored from a manifest in Git during Jenkins runtime.

Figure 13.2: Case B Considerations



Code: Cloned into GitHub as a step in the Dockerfile.

R: Inherited from a base image.

R packages: Installed as a step in the Dockerfile.

Figure 13.3: Case C Considerations

Chapter 14

Shiny Async

References and Resouces:

- Webinar - Scaling Shiny apps with asynchronous programming
- Webinar Slides
- Shiny Dev Center Article: Improving scalability with async programming