Guide

The R Developer's Guide to Databricks

databricks

Contents

Introduction

Welcome to the R Developer's Guide to Databricks, designed to assist R users and the system administrators who support them. For R users, our goals are twofold: first, to help you feel at home and make it clear how to do everything you normally do on Databricks. Second, to level up your skills and scale the work you're doing with the power of the platform. For admins, we aim to provide best practice recommendations for secure and cost-effective infrastructure management, while still being mindful of the preferences of many R users.

The content of the guide is organized systematically. We'll begin with the fundamental concepts and architecture of the Databricks Data Intelligence Platform, then bring those concepts to life by running R code in the Databricks workspace. Having gotten your hands dirty, we then go deep into how to set up your development environment — for the code editor in Databricks or IDEs like RStudio and VS Code, including a section on package management. At this point, you'll be properly oriented to Databricks and ready to learn how to scale your R code through Apache Spark™ and Databricks Workflows. This guide concludes with an Advanced Topics section, with details on Shiny.

It isn't our intention to replace Databricks documentation or rewrite the definitive book on R and Spark, so we'll reference existing resources wherever possible.

If you're looking for answers to specific questions, check the FAQ or search the pages. If your questions aren't answered, please raise an issue in GitHub.

Databricks Fundamentals

Databricks is on a mission to democratize access to data and AI. We accomplish this through a strategy of developing open source technology alongside a world-class commercial data platform. This began with Apache Spark, the in-memory cluster computing engine born in the Hadoop big data era. Apache Spark is *unifying* in that it can process data at petabyte scale, with APIs in SQL, Python, R and Scala. It supports machine learning on large datasets with Spark, near real-time streaming data pipelines and graph analytics. Spark is fast, scalable, flexible and *open source*.

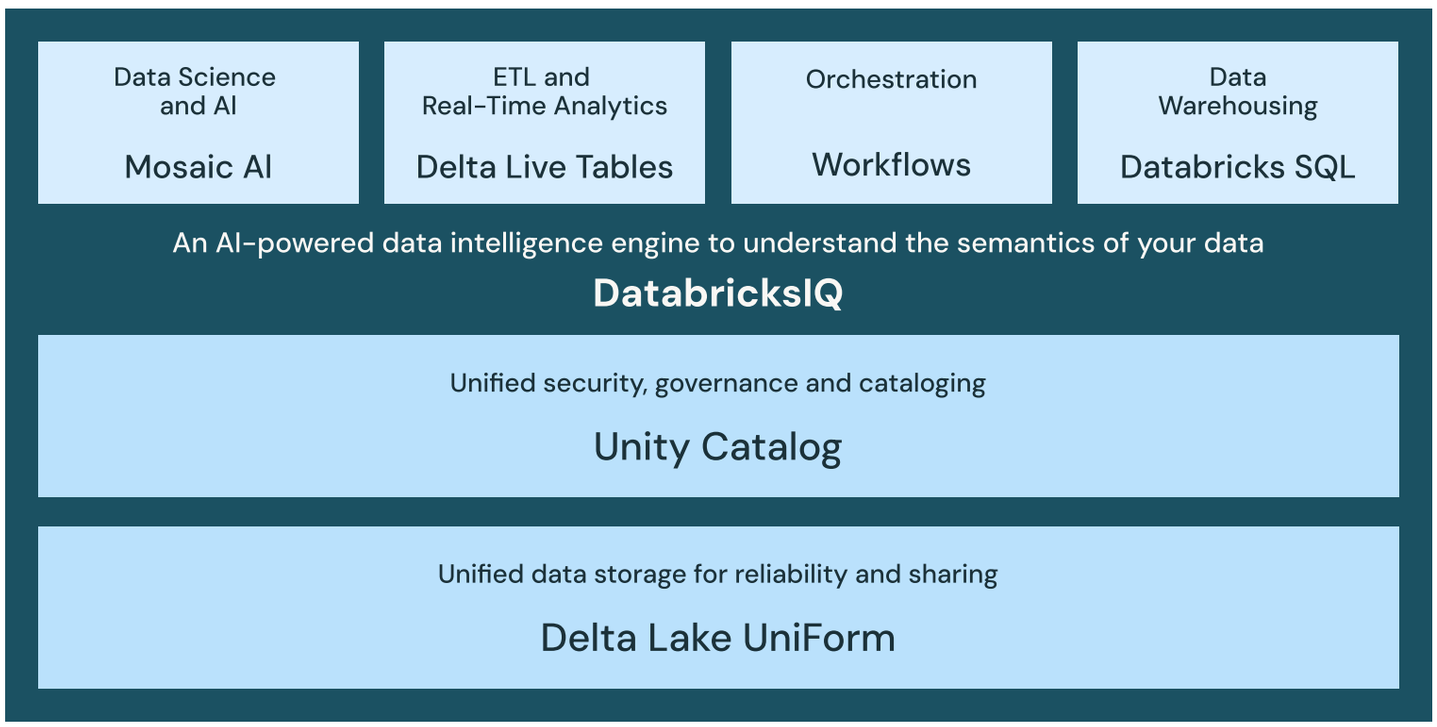
Another pillar of the Databricks Data Intelligence Platform is Delta Lake, an *open source* storage framework that *unifies* the data warehousing and data lake worlds — Databricks pioneered the term "data lakehouse," which is a portmanteau of these two terms. Delta Lake achieves unification by bringing capabilities that were traditionally only available in data warehouses to data lakes (e.g., ACID transactions). For the first time, structured data can now be managed alongside unstructured data with the same data quality and performance guarantees. Delta Lake is scalable too — it works extremely well with Apache Spark, so it reads and writes petabytes of data for breakfast.

MLflow is the third major *open source* project in the Databricks ecosystem, designed to help manage the entire machine learning model lifecycle. From model experimentation and selection to deployment and serving, MLflow helps individual data scientists stay organized while providing the essential governance framework for enterprise AI. Any arbitrary model can be managed in MLflow, and in this guide we'll explain how to make the most of R models with MLflow.

The final fundamental technology of Databricks is Unity Catalog, which is now also open source. Unity Catalog is currently the only open catalog for data and AI, unifying the governance model for tables, arbitrary files, functions and machine learning models. It aids teams and organizations in classifying and discovering data and AI assets and is an essential component to the emerging category of data intelligence. Check out the historic moment that Databricks co-founder and CTO Matei Zaharia open-sourced Unity Catalog onstage at Data & AI Summit 2024!

Each of the aforementioned technologies expand access to data and AI while simultaneously unifying the underlying open ecosystem. Apache Spark is the processing engine, Delta Lake the storage layer, MLflow helps get models into production and Unity Catalog governs across it all. The whole of these is greater than the sum of their parts, ensuring the right people get access to the right data and AI assets regardless of type or scale. This is what the Databricks Data Intelligence Platform is — open at its core, but far easier for admins to secure and data practitioners to use than if an organization attempted to do it themselves.

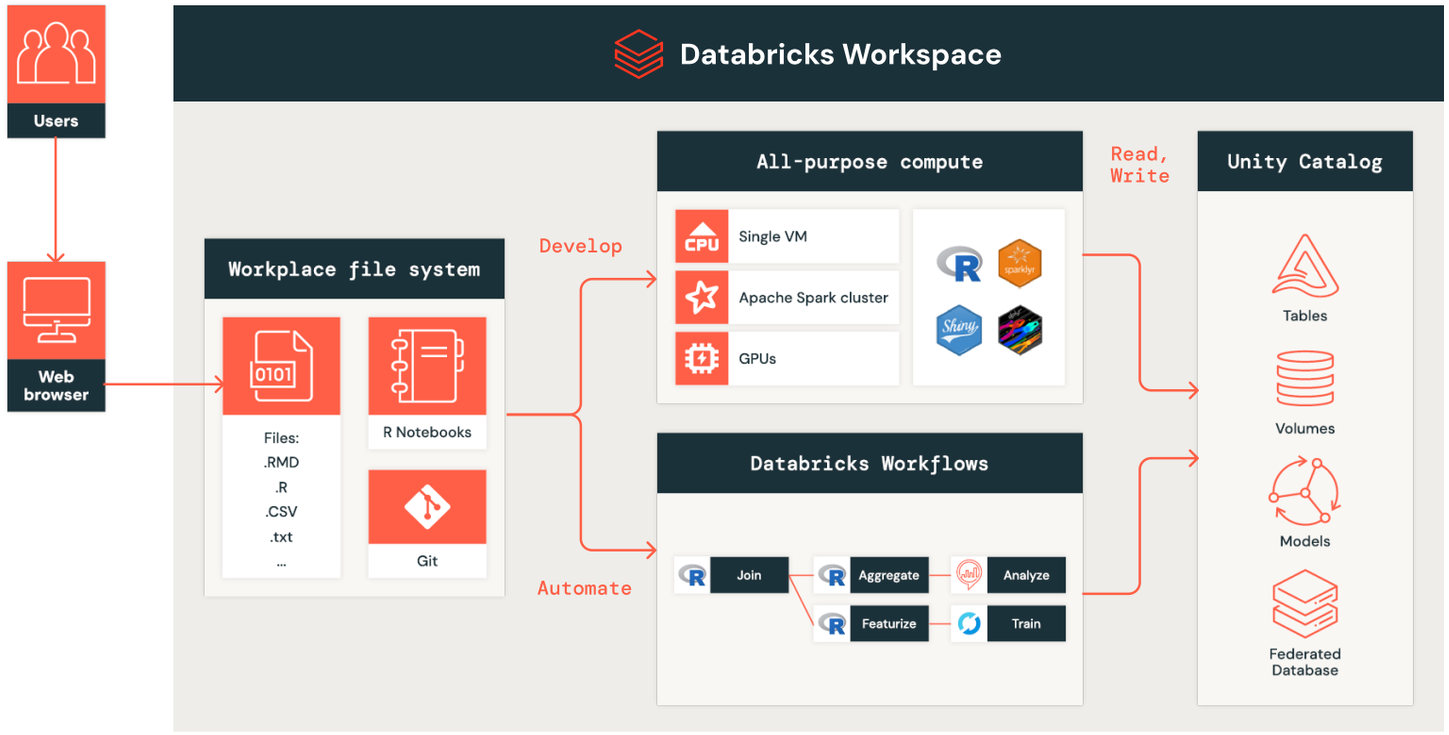
Databricks Data Intelligence Platform



We've covered the basics so far, but if you want to learn more about data intelligence, start with the Databricks documentation. Now let's turn our attention to how you interact with the Databricks Platform.

Workspace developer experience

Log in to the Databricks workspace from a web browser, which is the place to access all of the features and capabilities of the Databricks Platform.



In the workspace you'll find the workspace file system, where files and notebooks are managed. Folders in the workspace file system can be linked to version control systems like GitHub, allowing you to check out branches or commit code back to remote repositories. **The workspace file system is intended for files associated with a repository;** we'll talk about where to save files more generally later in this guide.

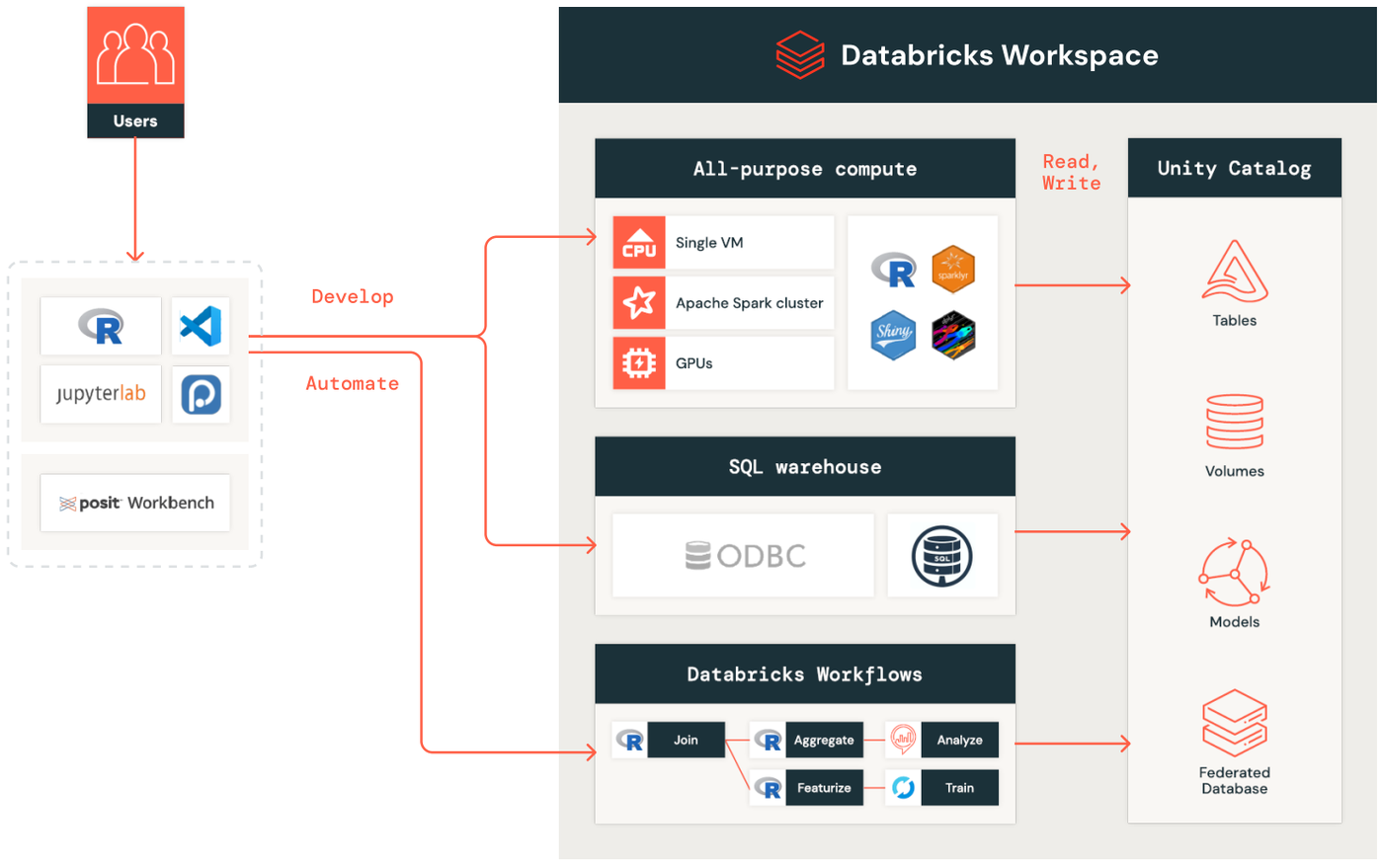
When you want to develop and run some code, several types of all-purpose compute are available. These are accessed by opening the Compute page from the left nav in the workspace, or directly from an open file or notebook. With the exception of serverless compute (which does not currently support R), you can install nearly any R package, custom or otherwise, on Databricks (see Package Management). **If you're just running R code, then single node is the right choice.** Think of single node as one virtual machine — a large desktop or workstation — where you get to choose the amount of RAM and CPU as needed. If you need to process larger amounts of data or do some sophisticated parallelized computations, Apache Spark clusters are the right choice. When you need to do some deep learning or generative AI, then GPU-enabled compute is the right choice. If you'd like to read more about the compute options or need some help making a decision, take a look at the compute configuration and computation management documentation.

Files and notebooks attached to all-purpose compute are able to read and write data in Unity Catalog (assuming you have permissions to do so). To explore Unity Catalog, open the Catalog page from the left nav or browse directly from the editor. Unity Catalog volumes are for storing any *files* (not tables) that aren't part of a code repository. A volume should be the long-term home for most RDS files, CSVs, video or image data, etc. Production-grade machine learning models are registered in Unity Catalog, though getting this to work with R models can be tricky. Your Unity Catalog admins may have also set up Lakehouse Federation, enabling you to access other databases more easily from Databricks.

Don't forget about automation. **Databricks Workflows is excellent for orchestrating and automating tasks, and it works well with R**. Workflows can be scheduled to run on a regular basis or triggered ad hoc, which is especially useful if you have a long-running task to execute but don't want to tie up your active R session. Running code on Databricks Workflows is also cheaper by default compared to all-purpose compute, making it a good way to maximize value. We believe this so strongly that we've included an entire section on the topic.

Using Databricks with IDEs

The main difference between using the workspace directly and using IDEs is the need to use APIs to remotely interact with Databricks. In both cases you'll connect and run code on Databricks compute resources, access data in Unity Catalog and automate with Databricks Workflows.



Note: Hosted RStudio on Databricks is deprecated, so we don't recommend building your architecture around it. In addition, this feature requires disabling auto-termination, forcing organizations to choose between leaving resources on continuously or finding ways to back up their users' work. For more details, see Guidance for working with IDEs.

There's a rich set of R and Databricks dev tools to interact with Databricks interactively or programmatically from IDEs. For processing data with Apache Spark, you can choose to use Databricks Connect through `sparklyr` or the Databricks ODBC driver through the `odbc` R package. There are also two R packages from Databricks Labs — `brickster` and the R SDK — that wrap the Databricks REST API in R. These packages provide utilities for programmatically interacting with Databricks (e.g., creating and running workflows). For projects that may need to operate in multiple workspaces (e.g., in a CI/CD process), Databricks Asset Bundles works with the Databricks CLI to simplify code deployment.

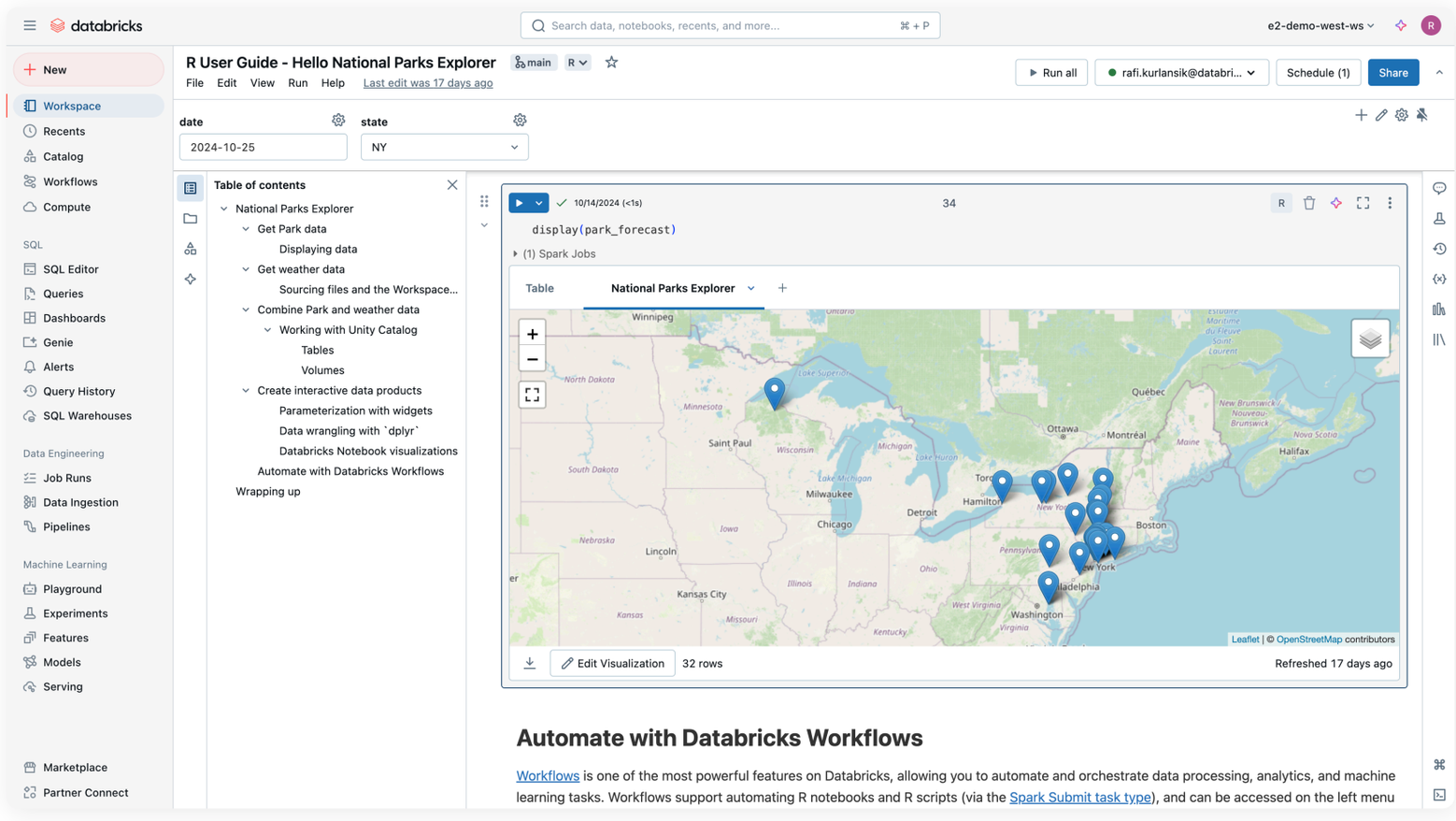
One notable change from the workspace developer experience diagram is the inclusion of Databricks SQL (DB SQL) warehouses. To understand SQL warehouses, it can be helpful to oversimplify and think of Databricks as "just another database." In the same way that you install and configure an ODBC driver for SQL Server or Oracle, then run queries using the `odbc` and `DBI` R packages, so too can you install the Databricks ODBC driver and run queries against a DB SQL warehouse. In RStudio, the connections pane will display tables in Databricks, allowing you to browse data in Unity Catalog. The ODBC driver can connect to DB SQL or all-purpose compute, but we recommend using DB SQL for better performance and lower cost.

Later in this guide we take a deeper look at these dev tools, their capabilities and when to use which ones. For now, know that these are the means to have interactive development sessions with Databricks from your R console.

Hello Databricks: A worked example

At this point you should have a basic understanding of Databricks fundamentals and entry points to work with the Databricks Platform. It's time to make these ideas more concrete and get your hands on the keyboard.

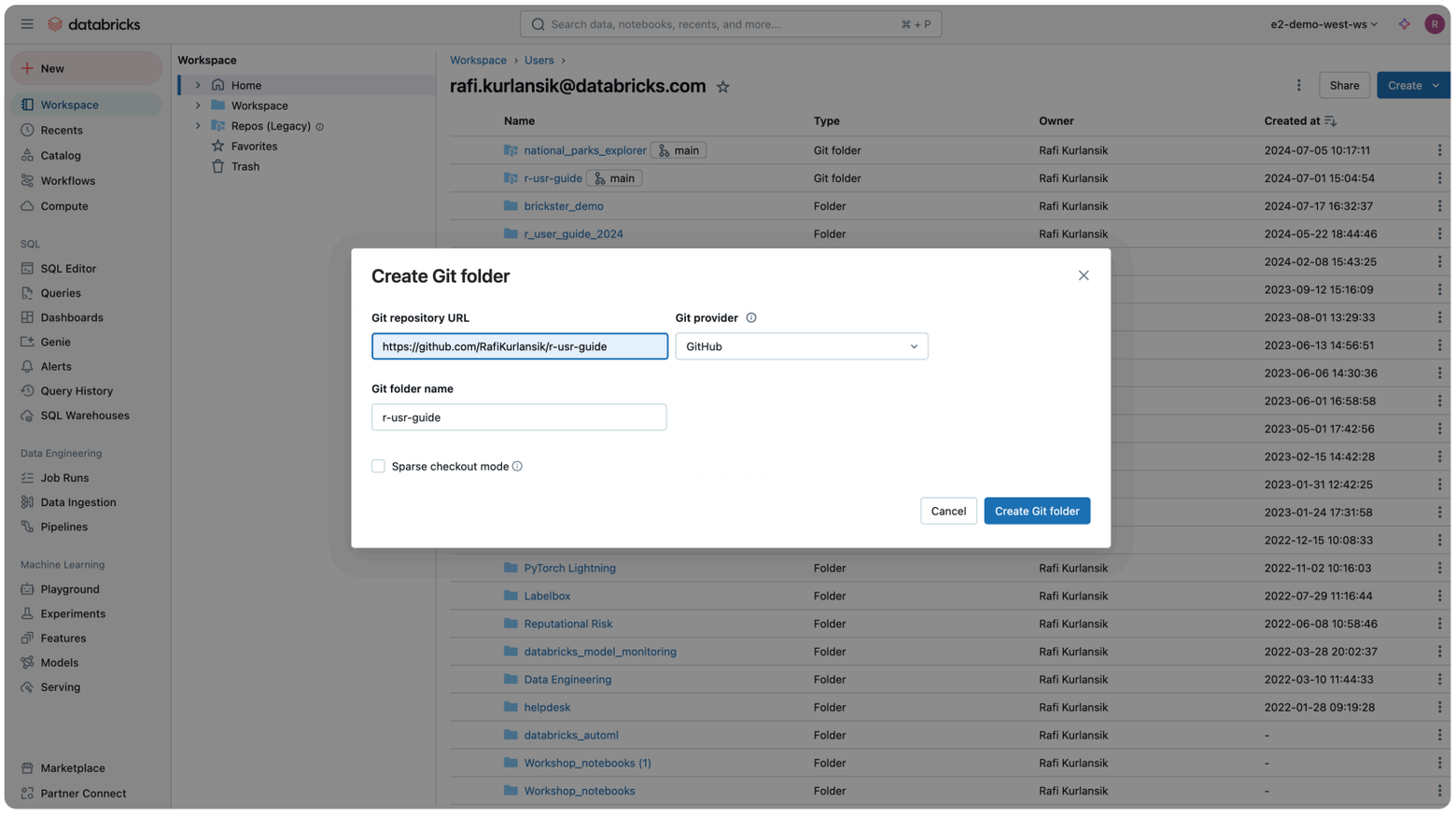
The following tutorial uses R to create an interactive visualization for planning a trip to National Parks in the U.S. Combine data from the National Parks Service API with weather forecasts from Open-Meteo.com to help make your decision.



Along the way, you'll learn how to use Databricks Notebooks, the workspace file system and Unity Catalog tables and volumes. By the end, you'll have an automated Databricks workflow to check the latest conditions in the parks.

HOW TO IMPORT THE NATIONAL PARKS EXPLORER CODE

The source files for this tutorial can be found at https://github.com/RafiKurlansik/r-usr-guide. We recommend importing the Git repository using Git folders.



If for some reason you can't use Git folders, then simply import the notebook and create a new file called get_weather_data.R and copy-paste the function into it.

ATTACH THE TUTORIAL NOTEBOOK TO PERSONAL COMPUTE

By default, all users should be able to create a small Personal Compute resource. After creating it, you can attach the *R User Guide - Hello National Parks Explorer* Notebook and start running code. Even if you have permissions to create other compute resources, the tutorial has been designed to work with Personal Compute so we recommend sticking with it.

> **Note:** R is not supported on standard or serverless compute as of July 2024.

WHAT IF I PREFER USING IDES?

If you're interested in a tutorial for using Databricks with an IDE, we recommend this one put together by the developers of `sparklyr` for Databricks Connect. We still highly encourage you to run through the tutorial in the workspace because it will make the concepts in this section much clearer and help you become a better Databricks developer.

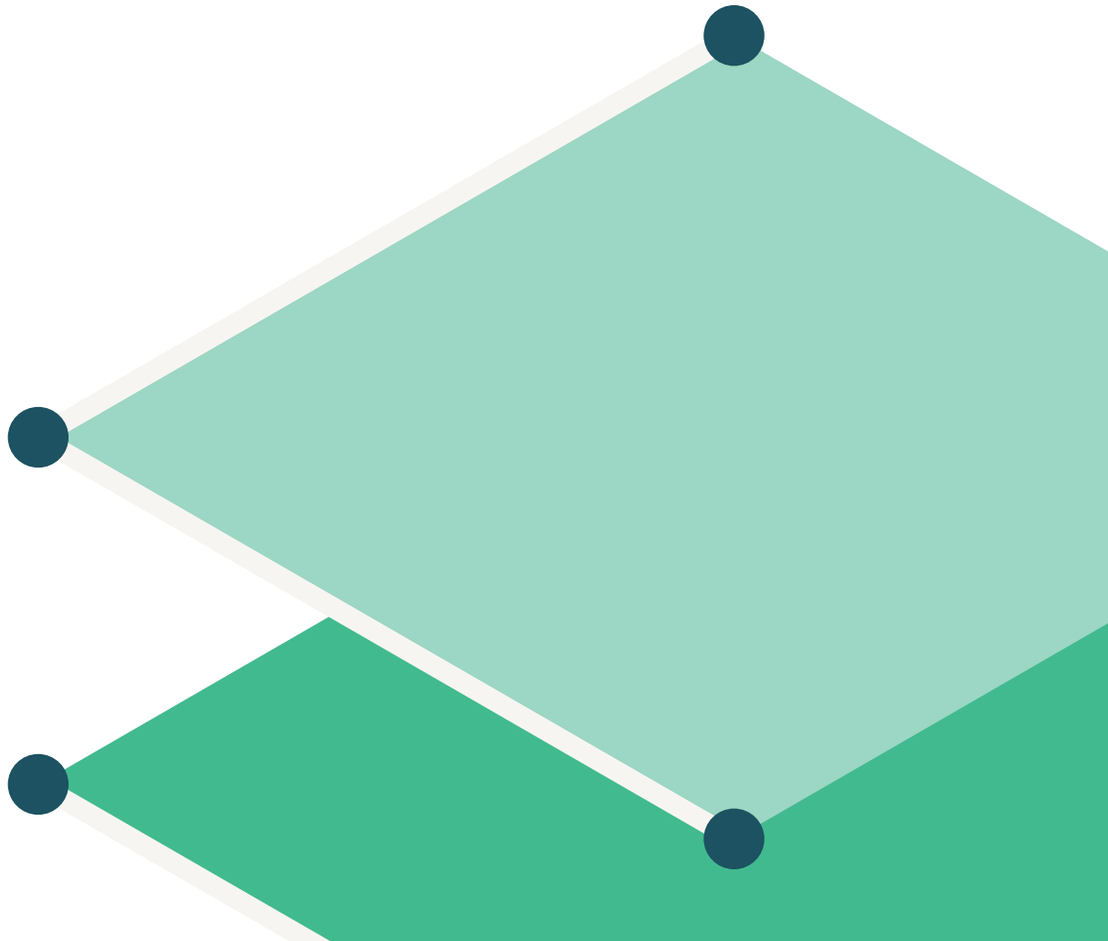
Setting Up Your Development Environment

The ingredients for a productive and joyful development environment include all of the quality-of-life features for writing code, as well as access to the right data and sufficient computing power to process it.

Choosing an editor

From a code authoring perspective, working with an IDE or the Databricks editor is largely a matter of preference. If you love RStudio, Positron or VS Code, you'll probably want to stick with them. On the other hand, if you prefer the convenience of a tight integration with the broader Databricks Data Intelligence Platform, or want access to all of its features, then the Databricks editor is the best choice.

R users generally expect the polish of RStudio, and as the table below illustrates, the code editor in Databricks delivers a comparable set of features.



	Feature	IDEs	Databricks editor with R
Code development and debugging	AI assistant / Ghost text	Yes	Yes
	Syntax / Error highlighting	Yes	Yes
	Code autocompletion	Yes	Yes
	Debugger	Yes	No
	Interactive console	Yes	Yes
	Environment explorer	Yes	No
	Quarto/R Markdown support	Yes	Limited
Visualization and data exploration	Integrated plot viewer	Yes	Yes
	Data / Variable explorer	Yes	Yes
Collaboration and project management	Version control integration	Yes	Yes
	Project import/export	Yes	Yes
	Real-time co-editing and commenting	No*	Yes
	Experiment tracking	No	Yes
Security and resource management	Scalability	Limited by local hardware (CPU, RAM)*	Dynamic, scalable cloud resources
	Setup and maintenance	Manual	Automated
	Security	User responsible**	Built-in cloud security and compliance features

*In the case of Posit Workbench, which offers co-editing and can be deployed on scalable infrastructure, the experience is more similar to Databricks. **Posit Workbench provides built-in security and compliance via SSO, console auditing, access logs and credential management.

If you haven't already done so, go ahead and read the workspace developer experience section and complete the National Parks Explorer tutorial. It'll give you a concrete sense of what it's like to work with R on Databricks.

Guidance for working with IDEs

If you prefer IDEs, it's important to understand which integrations with Databricks are available today and how they affect development and reproducibility.

UNDERSTANDING REMOTE EXECUTION

New users often get tripped up and think that by default, setting up a connection to Databricks means that all of their code is now running on bigger, faster hardware. As mentioned previously, *using IDEs with Databricks always entails a remote connection* to the platform that is facilitated through some API or protocol.

Figure: Tools like Databricks Asset Bundles allow users to interact with Databricks from IDEs via remote execution

In other words, **code always executes locally unless invoking REST APIs or Apache Spark Connect**. This means that Databricks can only offer more compute power if you use those APIs. If this sounds confusing, think of connections to Databricks like database connections — you *send* your query to the database, and the database *returns* results back to you. The nuances of how this affects development depends on which API or toolkit you're using to communicate with Databricks. These will be discussed in the toolkit section, but for now, just remember the key principle.

INTERACTIVE VS. BATCH EXECUTION

When working from an IDE, there are two modes by which you can execute code on Databricks: *interactive* and *batch*. Interactive execution is what most R users are familiar with — running a line of code from a cell in Quarto/R Markdown to the console and seeing the results instantly. Batch means the entire file you're editing is executed, similar to "Run all" in a notebook. Regardless of which you prefer, the tools available to R users for remote execution support both modes.

REPRODUCIBILITY

Working from an IDE also has implications for reproducible code. As a managed service, Databricks offers preconfigured software environments called runtimes, each with specific versions of R and many popular R packages. If you're working from a notebook, it's simple to reproduce your results when moving to automation or deploying to production. Simply choose the same runtime version and your code will behave exactly the same as it did when you were first writing it.

When working from an IDE, however, you're now responsible for ensuring compatibility between the local version of R, any packages and the runtime that will be used in automation or production. For example, if you're using the ODBC package with `dbplyr` and want to automate some scripts using Databricks Workflows, you'll need to install and configure the ODBC driver along with any other R packages as part of the setup for your script.

The release notes for each runtime contain the details you need to align local and remote environments, but if you want a more programmatic approach then tools like `renv`, the `brickster` package and the Databricks R SDK can be useful to help bridge this gap. Regardless of approach, when setting up your development environment we recommend being mindful of what your production environment looks like, then working backwards to align the two.

For more on this topic, see R development toolkit and Getting to production.

POSIT WORKBENCH

Developed by Posit PBC, Posit Workbench is a professional data science platform for R and Python developers. Posit partnered closely with Databricks to enhance Posit Workbench, and it's **our recommended enterprise solution for working with RStudio and Databricks**. Let's briefly discuss the biggest enhancements.

First, a new OAuth integration eliminates the need for users to manage personal access tokens. Users sign in to their Databricks workspace from the Posit Workbench UI, and their credentials are passed through to RStudio and VS Code sessions.

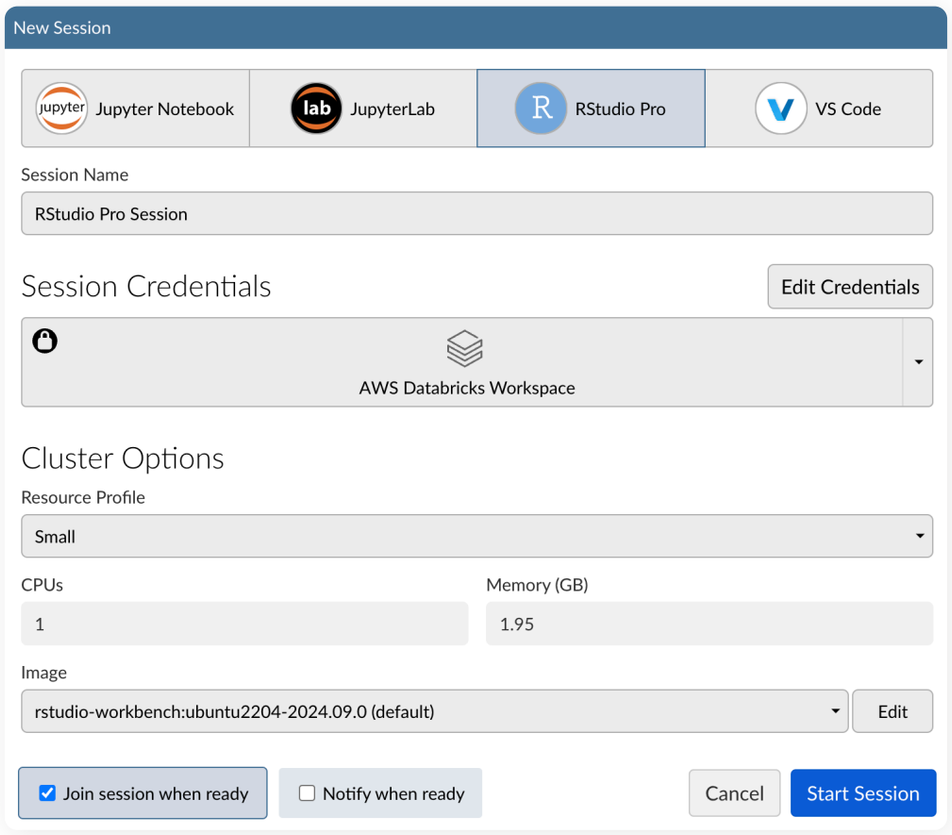


Figure: Signing in to a Databricks workspace with Posit Workbench using managed credentials

Not only is this *far* more secure than managing personal access tokens yourself, all of the R development toolkit packages come preconfigured and authenticated — you can start using them as soon as you sign in and launch a session. You can see these features and more in this live demo by Garrett Grolemund: Predicting Lending Rates with Databricks, tidymodels, and Posit Team.

In addition, RStudio has a new Databricks pane that lets users browse and manage compute options in their workspace and then connect from the UI.

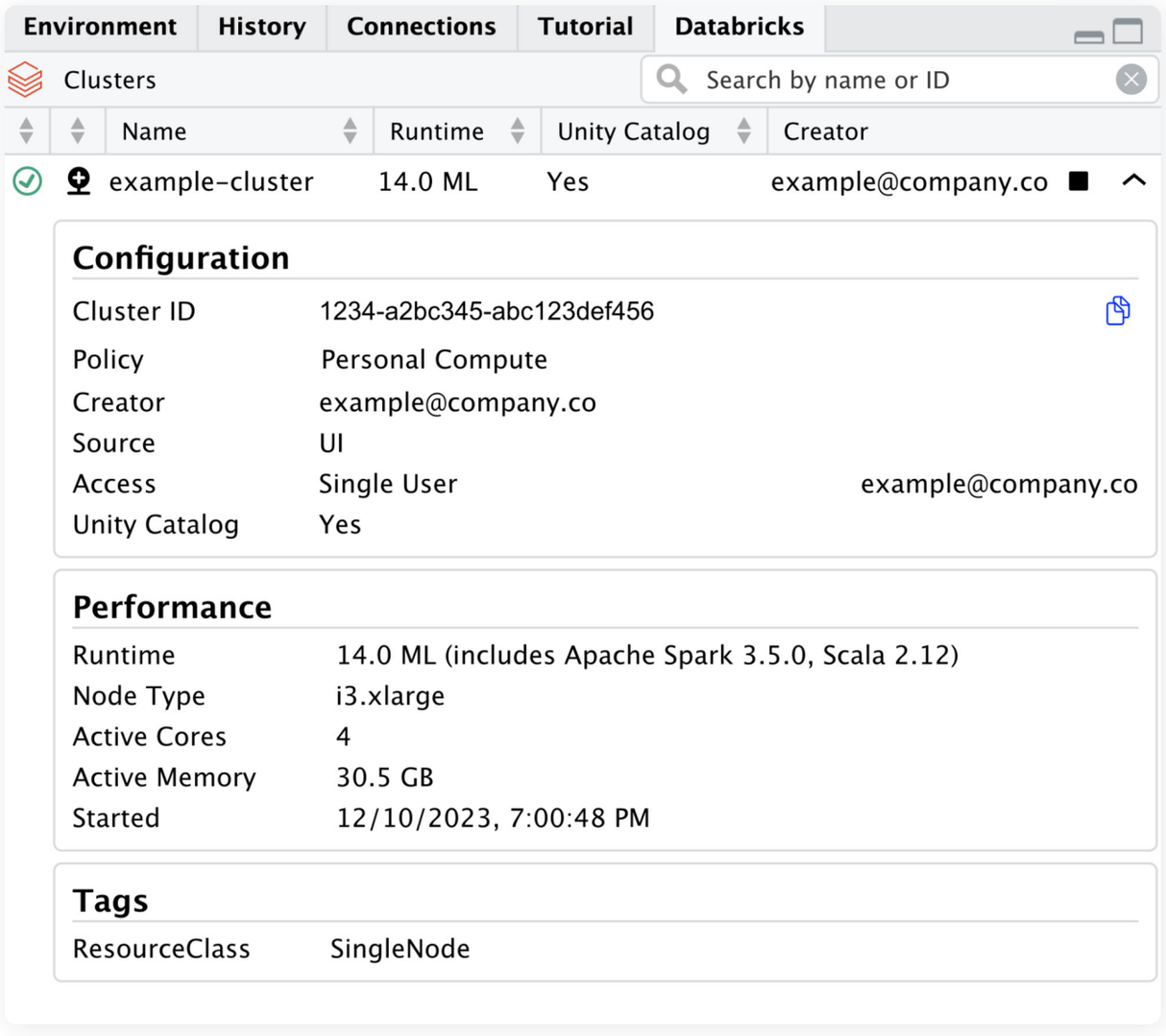


Figure: Browsing and connecting to remote compute resources from the RStudio IDE with Posit Workbench

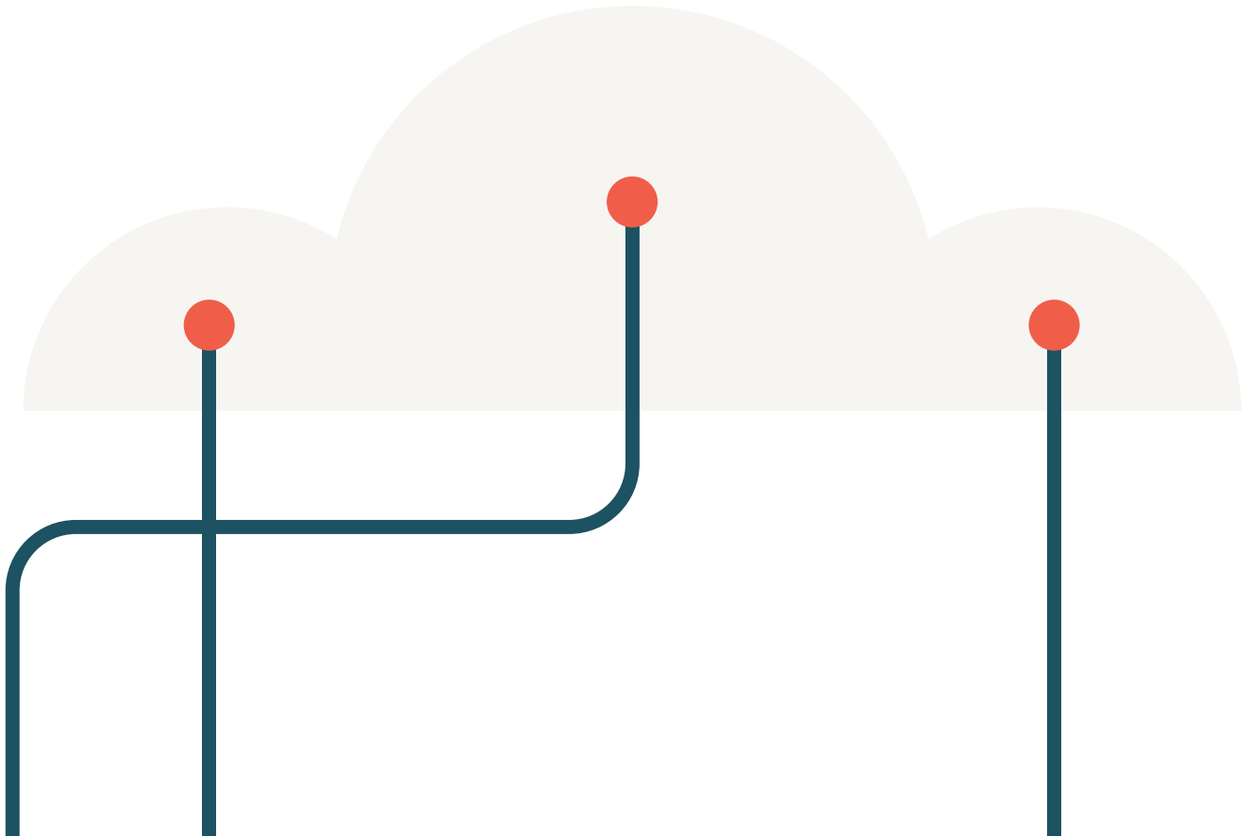
We believe users and administrators will love these features. To try it out, Posit offers a free trial.

OTHER IDES

When working with open source versions of RStudio, the Databricks pane isn't available, but the connections pane will still display tables in Unity Catalog when you make a connection with ODBC, Databricks Connect or the `brickster` package.

The Databricks extension for VS Code is supported by Databricks and includes functionality for running code interactively or batch, connecting to compute, OAuth and more. In fact, the Databricks pane in Posit Workbench was largely inspired by the VS Code extension. The extension works well with Posit Workbench–managed credentials and is available on OpenVSX and the VS Code Marketplace.

Positron is the latest data science–specific IDE developed by Posit. While still in public beta, it can be configured to use the Databricks extension for VS Code.



Compute resources and data access

If you aren't sure how much compute you need, or you're an admin who wants to ensure access to data in Unity Catalog for your R users, this section will set you on the right path.

SELECTING THE RIGHT COMPUTE FOR THE JOB

A constraint of working on a laptop or single VM in the cloud is that the compute is inflexible — you're stuck with the RAM and CPU available on your machine, and it can be difficult or impossible to swap for something more powerful. As discussed in the fundamentals section, Databricks completely eliminates these constraints by making it simple to configure and launch more powerful resources with a few clicks. Sometimes the variety of compute choices can be overwhelming, so let's summarize the use case for each.

	Use case	Processing capacity	Cost
Single node	Working locally in R on small or medium-sized datasets	<50 GB	**$-$$**
Apache Spark cluster	Big data processing, parallelizing arbitrary R code	TBs	**$-$$$**
GPUs	Deep learning, generative AI	High GBs to TBs	**$$$**
Serverless*	Big data processing	TBs	**$-$$$**

*Serverless is not currently available with R.

When it comes to processing capacity and cost, these are general rules of thumb to follow. In reality, the cost will depend on the exact task and the nature of the data you're working with. The compute configuration and computation management documentation have more details.

ENSURING ACCESS TO UNITY CATALOG

Not all configurations of compute resources on Databricks provide Unity Catalog access from R. To keep things simple, remember the following:

- **Dedicated** compute provides access to Unity Catalog from R in general. By default, dedicated compute is assigned to a single user, but a *group* of R users can be granted access to share the resource. Note that there are limitations.
- **Standard** compute provides access to Unity Catalog from R via Databricks Connect or ODBC. These are typically remote connections to Databricks from an IDE or Shiny app, and in these cases multiple R users can share standard compute resources.

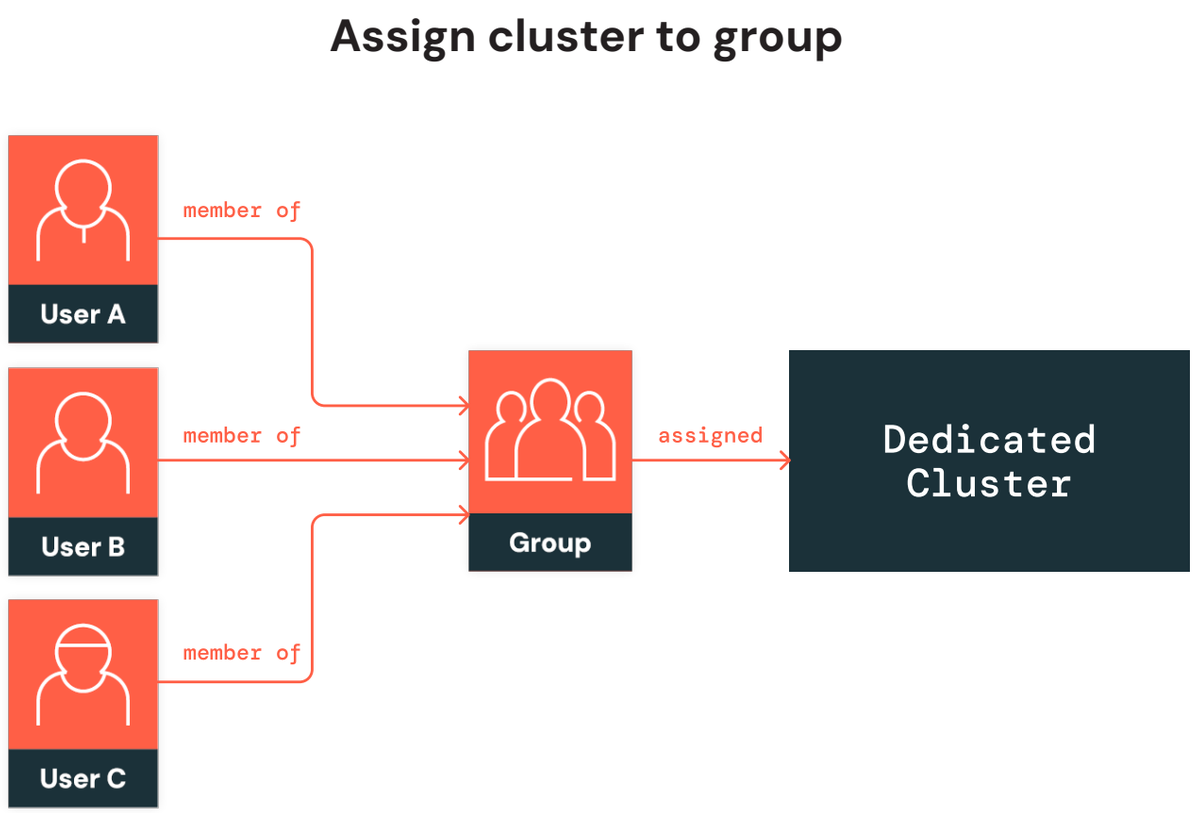


Figure: In the assign to group mode, users only need to be a member of the group the cluster is assigned to; their permissions will be downscoped to match the group

Note for administrators: We highly recommend using compute policies to set up guardrails for resource creation. Policies can be assigned to groups of R users, enabling them to create Unity Catalog–compatible compute for themselves while still controlling cost. This blog walks you through what a sensible policy might look like for data science teams, and it can be easily modified to include what R users need to share resources and access Unity Catalog.

R development toolkit

These tools are generally used for programmatically interacting with Databricks from an IDE, but they work within the Databricks workspace too. Before we review them, let's discuss authentication and the differences between officially supported tools and Databricks Labs projects.

AUTHENTICATION

To access Databricks remotely, you'll need credentials in the form of an API token. Almost all of the official dev tools for Databricks support OAuth to establish a secure connection to your Databricks workspace. OAuth works by prompting you to sign in, then storing a short-lived token on your machine. While you work, the OAuth client will continually refresh your credentials behind the scenes. If you stop working, the credential expires and you'll be prompted to sign in again the next time you use one of the dev tools.

We highly recommend using OAuth instead of personal access tokens (PATs), which are generally long-lived and easier to discover on your machine if you're hacked. **If you do use PATs, always use environment variables instead of putting credentials in code or plain text.** Here's a simple way to set the proper environment variables in RStudio with the `usethis` package.

```
1  usethis::edit_r_environ()
```

Then, in your `.renviron` file, set the following variables.

```
DATABRICKS_HOST = https://my-workspace.cloud.databricks.com
DATABRICKS_TOKEN = dapif9189unasdfuaod8f7o1f3n1l
```

Save the file, restart R, and you're ready to authenticate with the Databricks REST APIs through the various dev tools. To read more about authentication with Databricks, read the official documentation.

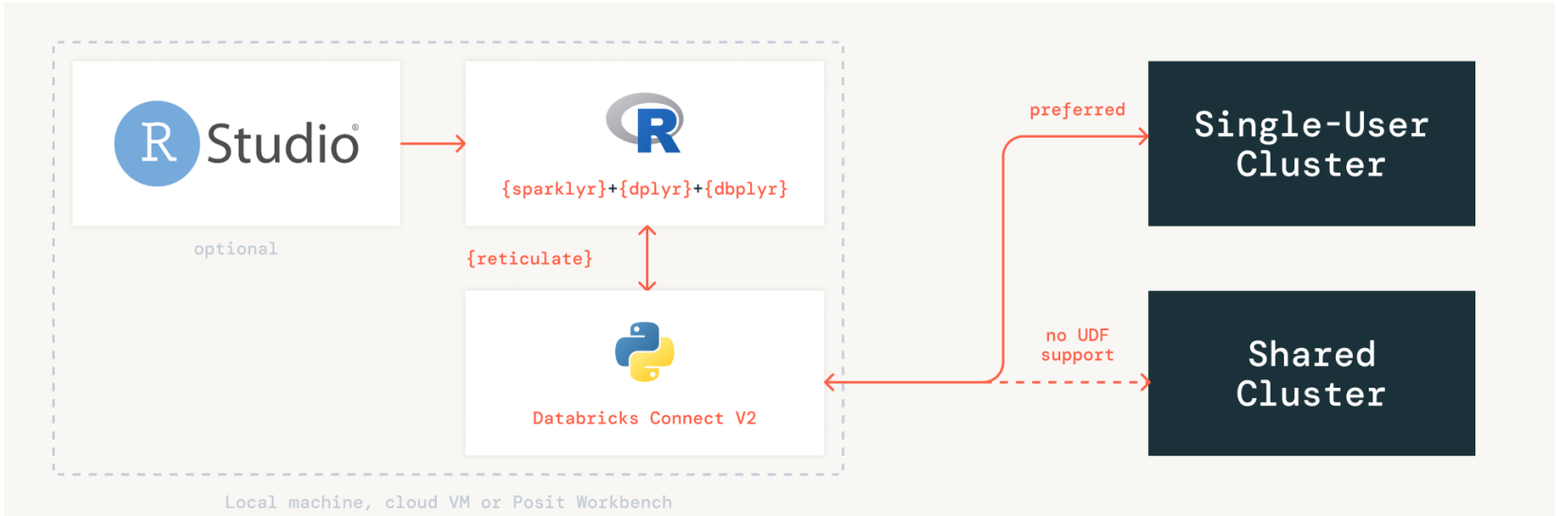
OFFICIALLY SUPPORTED TOOLS VS. DATABRICKS LABS PROJECTS

Officially supported tools are maintained by the Databricks Engineering team and are eligible for technical support. These include Databricks Connect, the ODBC driver, CLI and Databricks Asset Bundles. Databricks Labs projects are created and maintained by field engineers at Databricks to solve real customer challenges. Labs projects must meet certain standards for testing and maintenance, ensuring you don't wind up using software that is buggy or abandoned. Databricks Labs projects are not *officially* supported or part of any SLA, but we encourage you to use Databricks Labs projects and share feedback through GitHub.

With these distinctions in mind, let's discuss the dev tools themselves.

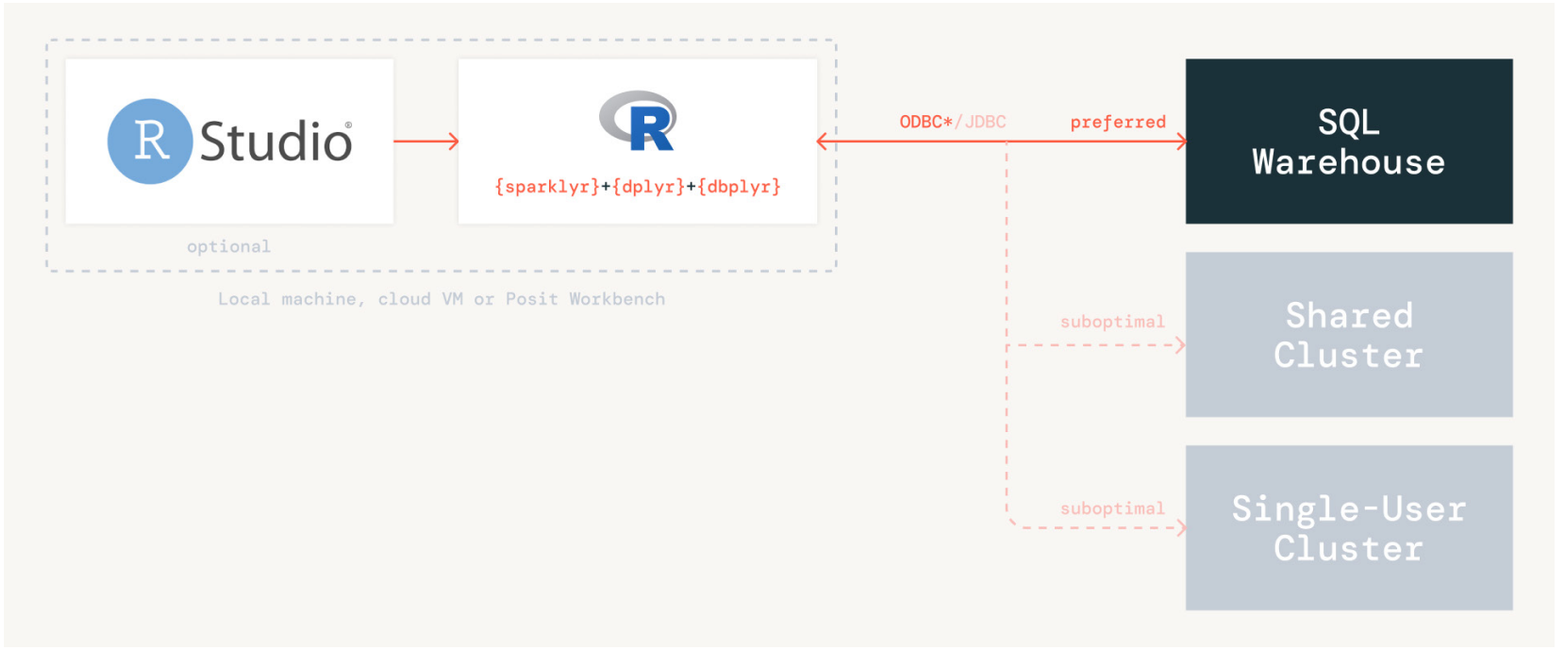
DATABRICKS CONNECT AND THE `odbc` PACKAGE

Starting with version 2.0, Databricks Connect allows you to access an Apache Spark cluster on Databricks remotely through a lightweight client library using the Spark Connect protocol. It supports a subset of Spark APIs, including DataFrames, SQL, machine learning and UDFs (user-defined functions). Databricks Connect is ideal for interactively exploring and transforming large datasets directly from your laptop, or creating interactive Spark-powered Shiny apps that operate independently from the Spark cluster.



You can learn more about the integration between Spark Connect and the `sparklyr` R package from the Databricks blog. The docs for `sparklyr` include details on how to use Databricks Connect, a tutorial and help for troubleshooting issues you might encounter. See this example of a Shiny app that uses `sparklyr` with Databricks Connect.

As mentioned in the Databricks Fundamentals section, the Databricks ODBC driver is a great choice for users that favor `dbplyr` or `DBI` and have experience working with database connections in R. As with Databricks Connect, the ODBC driver is a great way to give your Shiny app (or Quarto doc) a super-scalable and efficient back-end execution engine. Posit has added a special `databricks()` function to the `odbc` package, making it simple to create a connection to DB SQL warehouses or all-purpose compute.



This guide will help you get set up from RStudio Desktop or Posit Workbench. Posit Workbench customers can install the latest Databricks drivers and leverage Workbench-managed user credentials out of the box. We don't recommend using the Databricks JDBC driver with R due to comparatively poor performance and the risk of needing to spend time tinkering with `rJava`.

Both Databricks Connect and ODBC drivers allow R users to interactively work with data in Databricks. Due to their ability to leverage standard compute from a remote R session, they represent a more cost-effective solution for large teams of R users or mixed R and Python users. However, both Databricks Connect and ODBC support only a subset of Spark APIs. Meaning, R code that doesn't use `sparklyr` or the ODBC driver will execute on the same hardware as the active R session. (In the preceding figures, this would be the local machine, cloud VM or Posit Workbench.)

DATABRICKS CLI AND DATABRICKS ASSET BUNDLES

The Databricks CLI offers a comprehensive interface to the Databricks REST API and is essential for bash scripting automation. It's easy to install and get started, though if you aren't familiar with working from the terminal it can take a little getting used to.

The CLI works seamlessly with Databricks Asset Bundles (DABs), a capability which allows users to describe their Databricks projects in YAML format, manage deployments and run them in batch mode within a Databricks workspace. These YAML projects can be checked into version control and templated, helping R users follow software engineering best practices. To get an idea of what DABs look like with R, see the automation chapter of this guide.

R users might not need DABs or the CLI for simple projects. For larger or business-critical projects, they offer a lot of value and we highly recommend learning how to use them. Deploying workflows with their various configurations during testing and continuous integration/continuous deployment (CI/CD) processes is painful without DABs and the CLI. Therefore, **if you know you're going to take a project to production eventually, it's best to kick it off using DABs.**

If working with YAML is new to you, we recommend reviewing the following resources:

- Introduction to DABs at Data & AI Summit
- Self-guided, clickable demo
- Dustin Vannoy's introduction to DABs and advanced patterns

brickster

brickster is the R package built for R developers by an R developer. It wraps Databricks REST APIs that are of greatest interest to R users such as Databricks Workflows, file system operations and cluster management. In addition, it includes a rich set of utility functions and integrations with RStudio to streamline development and generally increase fidelity between your IDE and Databricks. It's well documented with vignettes for job automation and cluster management, and examples for each function. **Whether you're a rookie or a power user, if you're working with Databricks from an IDE, you owe it to yourself to give brickster a test drive.**

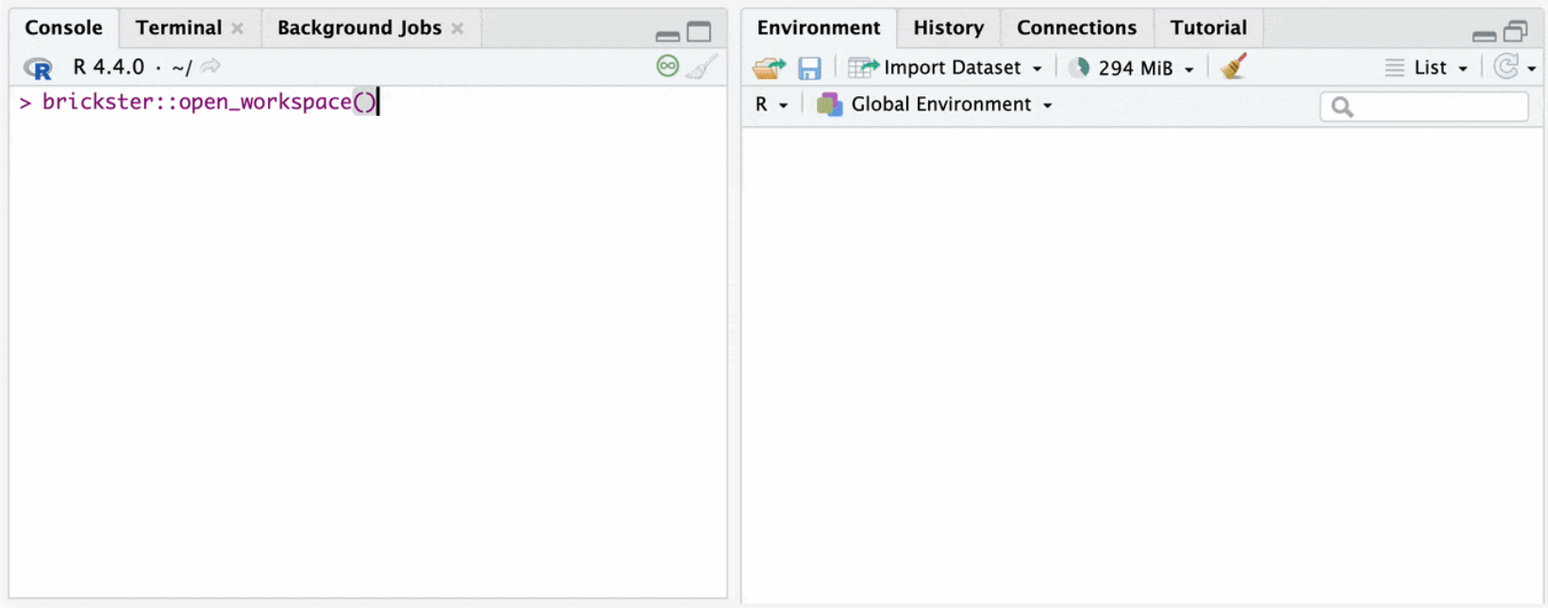
Let's spend some time reviewing the unique and immersive features in brickster.

Authentication

brickster supports the standard mechanisms for authenticating to a Databricks workspace and makes OAuth the default. This makes it simple to get started — after installing the package, use any of the functions and you'll be prompted to sign in to Databricks. See this article for more details on getting connected with brickster.

Workspace browser

A logical place to start a session with `brickster` is to connect to your Databricks workspace with the `open_workspace()` function. This will display relevant workspace objects in the RStudio Connections Pane: Unity Catalog, file system, compute.



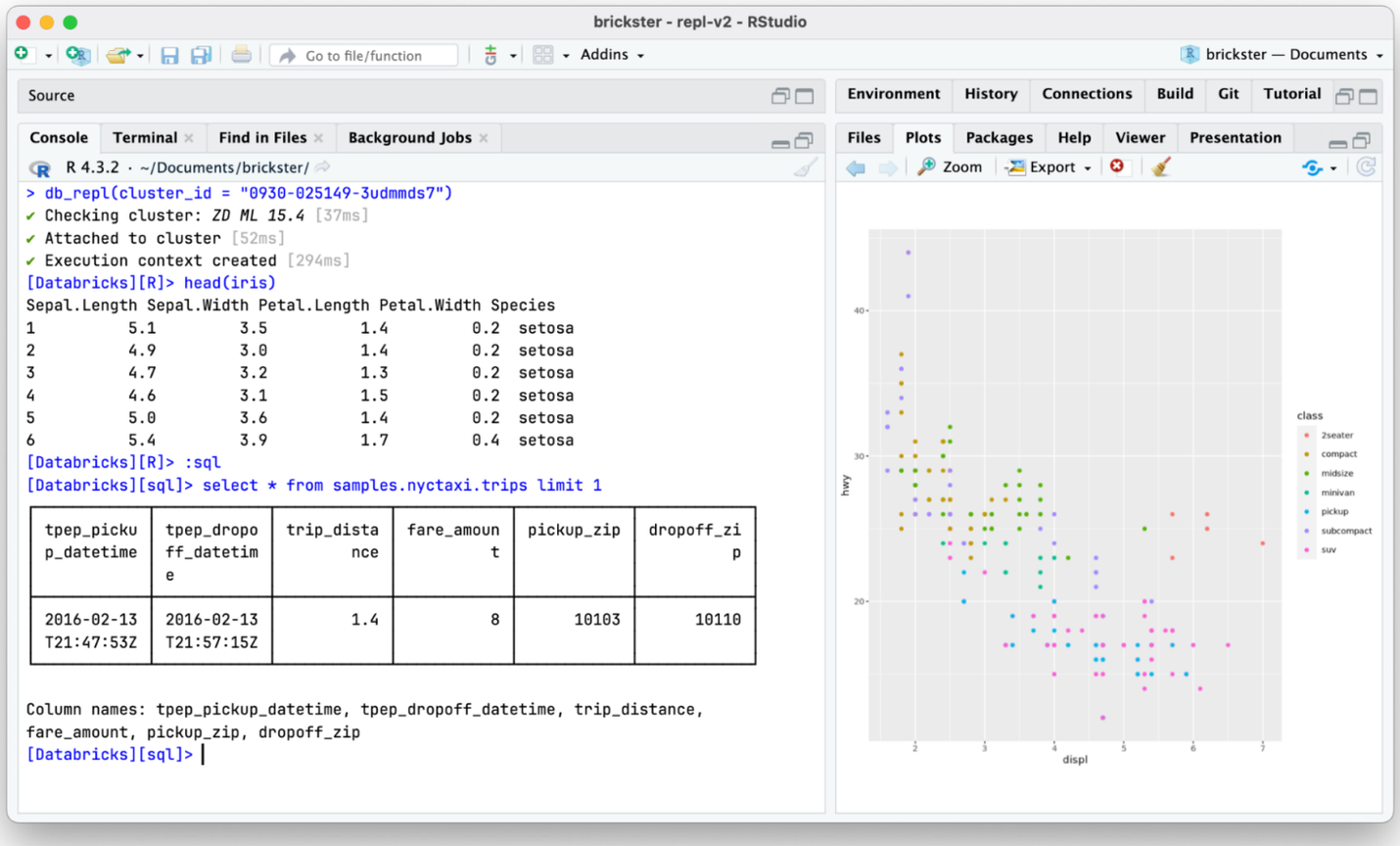
Among other things, this is very helpful for:

- Finding a table
- Getting a cluster ID
- Getting an MLflow Experiment or registered model ID

To close the connection, use `close_workspace()`.

REPL

For the most immersive developer experience, check out the `db_repl()` function in `brickster`. It creates a local REPL (read-eval-print loop) where every command executes remotely on Databricks in the language of your choice.



There are some limitations with `db_repl()`, namely that results aren't streamed back from Databricks compute, so installing packages or long-running computations won't show any progress.

DATABRICKS R SDK

The `databricks` R package is a wrapper around the Databricks REST API, and it's part of the larger ecosystem of Databricks SDKs that look the same, authenticate the same and work the same across Go, Python and Java. It provides an R interface for CRUD operations with various Databricks objects like folders, files and notebooks in the workspace, compute resources and workflows. The package has some nice features, like managing polling during long-running operations and pagination of large results.

Authentication is best handled by using Databricks CLI configuration profiles. If you're working in Posit Workbench, authentication is managed for you when you sign in to your Databricks workspace.

```
# Sample .Renviron file configured to authentication with the databricks package
DATABRICKS_HOST=https://my-workspace.cloud.databricks.com
auth_type=databricks-cli
```

If not using the CLI to authenticate, you can set the DATABRICKS_HOST and DATABRICKS_TOKEN environment variable in a `.Renviron` file.

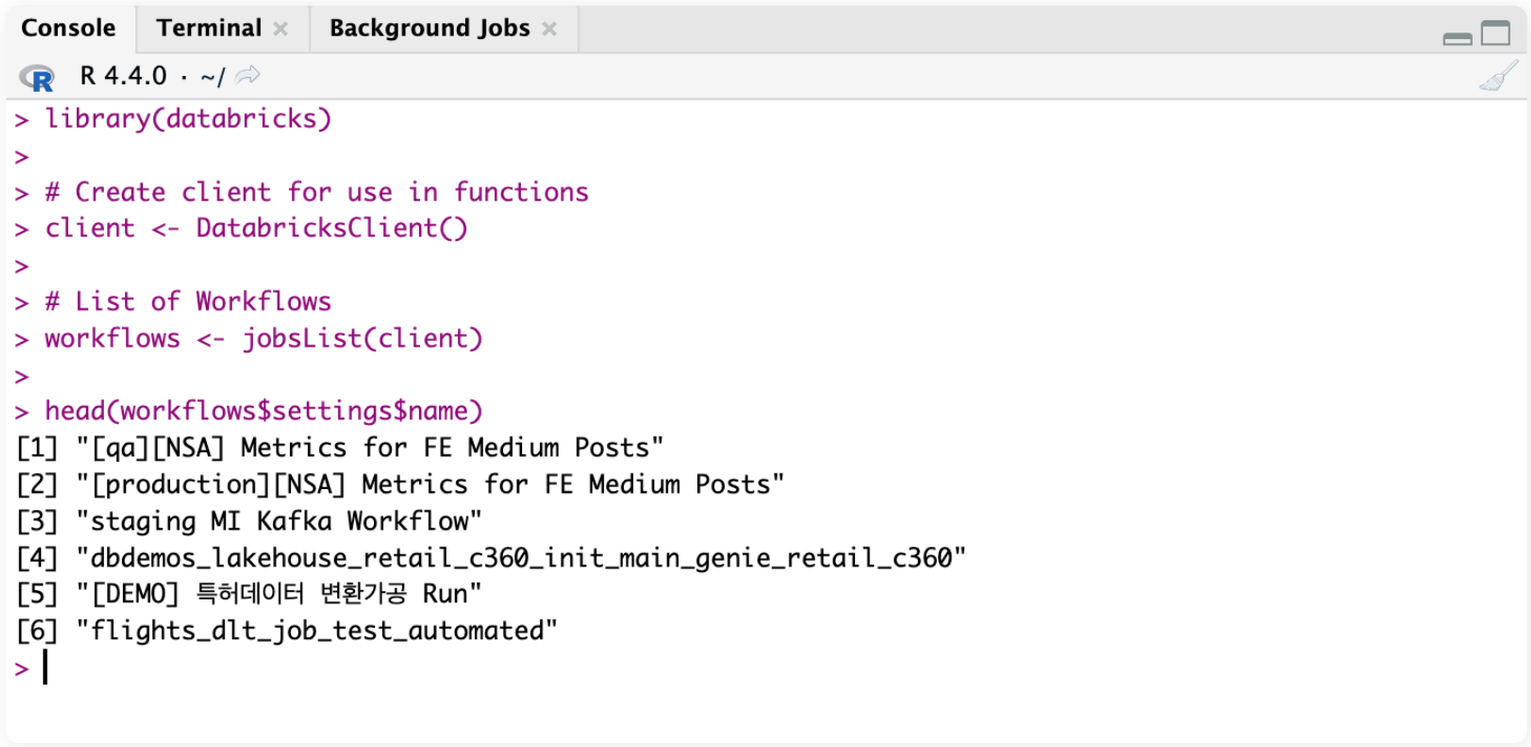
Once you've set the host and token, install directly from GitHub via the `remotes` package and load it into your R session.

```
# Install and load R SDK
remotes::install_github("databrickslabs/databricks-sdk-r")
library(databricks)
```

To work with any of the services in Databricks using the R SDK, the first step is to create a `DatabricksClient()`object.

```
client <- DatabricksClient()
```

The client picks up credentials stored as environment variables or in Databricks CLI profiles, making it a prerequisite to using *any* of the other functions in the R SDK. To use it, you simply pass it as an argument to the function you'd like to call.



In this example we use the *client* to call `jobsList()`, returning a DataFrame of details for every single workflow accessible to us in the Databricks workspace. In general, the R SDK will return API responses as DataFrames, with some of the columns being nested DataFrames themselves (see `settings` and `tags` in this example).

We recommend using the R SDK if you need complete API coverage, but the typical R user might find `brickster` easier to use and better suited to their development needs.

Package Management

Databricks supports a variety of options for installing and managing new, old and custom R packages. We'll begin by providing examples of the basic approaches, then progress into more advanced options.

Installing packages

At the most basic level, R package installations can be notebook-scoped or cluster-scoped. Cluster-scoped packages are part of the cluster configuration itself, becoming installed automatically upon restart and available to anyone who uses the cluster. Notebook-scoped packages are installed via `install.packages()` and are only available to users of the notebook. This allows for multiple versions of the same package to be used on the same cluster through multiple notebooks.

Regardless of which method you choose, packages will be available on each worker of the cluster. This is important when you want to perform user-defined functions (UDFs) with `SparkR` or `sparklyr`.

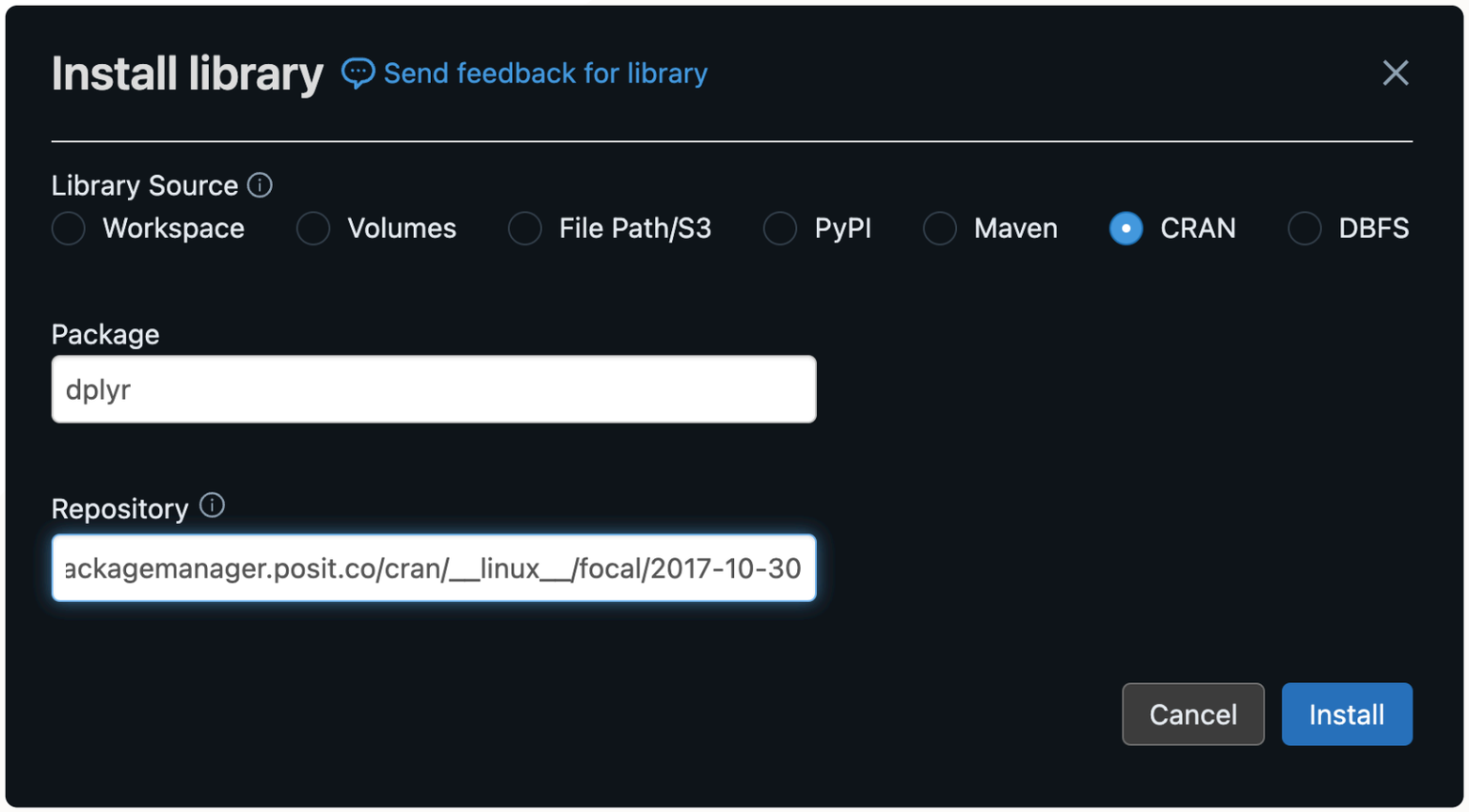
OLDER PACKAGE VERSIONS

The system environment for Databricks Runtime includes many popular R packages. These are typically the latest stable versions, but sometimes installing the latest version of a package can break your code. If you need to install an older version of a package, you have two options: the `devtools` package or a snapshot from Posit Public Package Manager.

From a notebook, you can use the `devtools` package.

```
devtools::install_version("dplyr", version = "0.7.4", repos = "https://packagemanager.posit.co/
cran/__linux__/jammy/latest")
```

To install an older package at the cluster scope, use a snapshot from Posit Public Package Manager (PPM). Posit archives CRAN (Comprehensive R Archive Network) packages on a regular basis, so packages pulled from a specific date will contain the version available at that time. You can access snapshots from the setup page in PPM. For version 0.7.4 of `dplyr`, you'd have to go back to the snapshot from 2017 and set that URL as the repository in the cluster UI:



CUSTOM PACKAGES

To install a custom package on Databricks, first build your package from the command line locally or using RStudio. Next, use the Databricks CLI to upload the file to a Unity Catalog volume:

```
databricks fs cp /my_dir/custom_package.tar.gz /Volumes/my_dir
```

Once you have the `tar.gz` file in a volume, you can install the package using `install.packages()`.

```
install.packages("/Volumes/my_dir/custom_package.tar.gz", type = "source", repos = NULL)
```

If the package source is located on GitHub or other version control systems, the `remotes` package will install directly from the repository:

```
remotes::install_github("tidyverse/dplyr")
```

SYSTEM DEPENDENCIES

Sometimes when installing a package, you might get an error message in your notebook or in the cluster UI that looks similar to this:

```
Warning in i.p(...) : installation of package <package_name> had non-zero exit status
```

This is usually due to missing dependencies, which you can try to remedy by passing an additional argument to `install.packages()`.

```
install.packages('packagename', dependencies = TRUE)
```

If you still see the `non-zero exit status` error, then you're probably missing an OS-level (Ubuntu) dependency.

For example, some visualization packages depend on GDAL, which isn't bundled as part of Databricks Runtime. When installing packages that use GDAL, like `sf`, you may see the `non-zero exit status` error along with a message like `gdal-config not found or not executable`.

In these cases, take the following steps.

1. Identify the system-level dependency by checking the stack trace
2. Look up how to install the dependency in Ubuntu
3. Use the web terminal to install the dependencies and troubleshoot the R package installation. If you can't use the web terminal, use `%sh` cells in Databricks Notebooks.

This will take some trial and error, but once you've figured out the right commands to install the R package with system dependencies, put the commands in an init script and add it to your compute configuration. This will ensure the package is installed properly on startup. Alternatively, you can run the commands in `%sh` cells.

Faster package loads

> " . . . you will be fastest if you avoid doing the work in the first place."
> — Dirk Eddelbuettel

You may have noticed that installing packages on the Databricks Platform can take a while. It could take minutes — or hours in extreme cases — to install the suite of packages your project requires. This is especially tedious if you need to do this every time a job runs, or each morning when your compute gets restarted.

WHAT IS SLOWING YOU DOWN?

The default behavior of `install.packages()` is to download package binaries for your operating system, if available. If binaries are unavailable, R will instead download the package source files from CRAN in `packageName.tar.gz` format. Binaries can be installed into your library directly, while source files need to be compiled first. By default Databricks installs packages from CRAN, which does not provide precompiled binaries for Linux. Given that Databricks compute uses Linux and is ephemeral by default with no persistent storage, packages must be recompiled and installed upon restart, leading to longer installation times than Windows or Mac users may be accustomed to.

Posit Public Package Manager saves the day here, containing Linux binaries for all packages on CRAN. There's a helpful wizard to get started. With this knowledge you can make installing R packages in Databricks significantly faster. As an added benefit, this approach also often limits the number of system dependencies required for package installation. There are multiple ways to solve this, each differing slightly, but fundamentally the same.

SETTING `repos` WITHIN A NOTEBOOK

The quickest method is to follow the wizard and adjust the `repos` option:

```
# Set the user agent string, otherwise pre-compiled binaries aren't used
# HTTPUserAgent is required when using R 3.6 or later
options(
 HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["platform"],
R.version["arch"], R.version["os"])),
  repos = "https://packagemanager.posit.co/cran/__linux__/jammy/latest"
)
```

This works well, but not all versions of Databricks Runtime use the same version of Ubuntu. It's easier to detect the Ubuntu release code name dynamically:

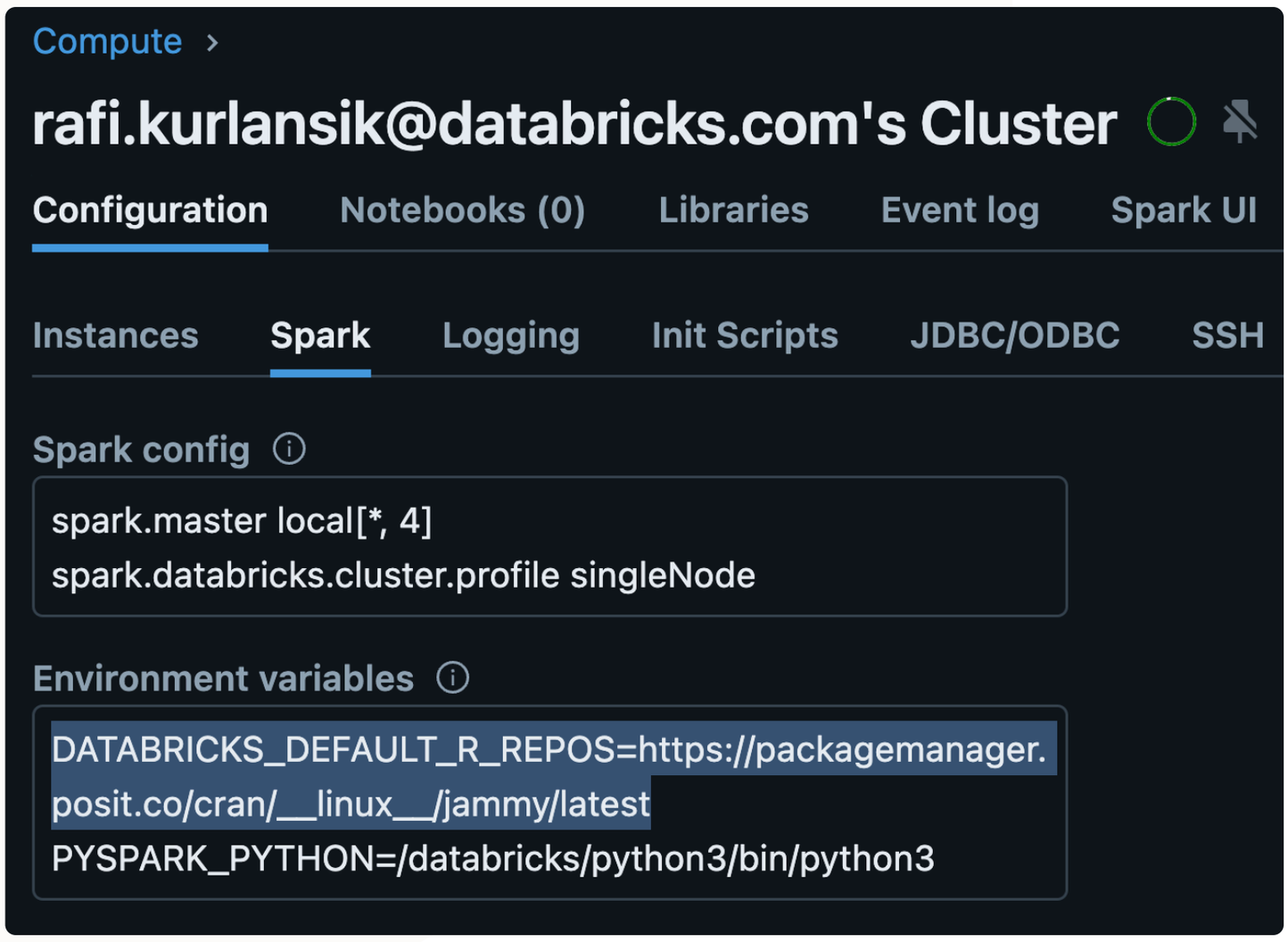
```
# Get Ubuntu release version
release <- system("lsb_release -c --short", intern = TRUE)

# Include in repos string
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["platform"],
R.version["arch"], R.version["os"])),
  repos = paste0("https://packagemanager.posit.co/cran/__linux__/", release, "/latest")
)
```

In this example, `system()` is used to run the command to retrieve the release code name. The downside of this method is that it requires every notebook to adjust the `repos` and `HTTPUserAgent` options.

ENVIRONMENT VARIABLES AND INIT SCRIPTS

Databricks compute resources allow specification of environment variables, and there is a specific variable — `DATABRICKS_DEFAULT_R_REPOS` — that can be set to adjust the default repository.



Unfortunately this isn't as dynamic as the first option. You'll still need to set `HTTPUserAgent` in `Rprofile.site` via an init script:

```
#!/bin/bash
# Append changes to Rprofile.site
cat <<EOF >> "/etc/R/Rprofile.site"
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["platform"],
R.version["arch"], R.version["os"]))
)

EOF
```

Note that due to how Databricks starts up the R shell for notebook sessions, it's not straightforward to adjust the `repos` option in an init script alone. `DATABRICKS_DEFAULT_R_REPOS` is referenced as part of the startup process after `Rprofile.site` is executed and will override any earlier attempt to adjust `repos`. Therefore, you'll need to use **both** the init script and the environment variable configuration.

Persisting packages

Before going any further, evaluate whether faster package loads can solve any installation pain points. It's an easier solution to set up and manage, but sometimes it's preferable not to install at all and persist the packages required, similar to how you'd use R locally. To achieve the same functionality on Databricks, we have to understand where packages get installed on Databricks compute.

All examples in this section use Unity Catalog volumes. DBFS can be used, but we don't recommend it.

`.libPaths()` ON DATABRICKS

When installing packages with `install.packages`, by default they'll be installed to the first element of `.libPaths()`, which returns the paths of "R library trees," directories where R packages reside. When you load a package, it'll be loaded from the first location it's found, as dictated by `.libPaths()`.

When working within a Databricks Notebook, `.libPaths()` will return six values by default. In order they are:

Path	Details
`/local_disk0/.ephemeral_nfs/envs/rEnv-<session-id>`	The first location is always a notebook-specific directory. This is what allows each notebook session to have different libraries installed.
`/databricks/spark/R/lib`	Only `{SparkR}` is found here
`/local_disk0/.ephemeral_nfs/cluster_libraries/r`	Cluster libraries: You could also install packages here explicitly to share amongst all users (e.g., `lib` parameter of `install.packages`)
`/usr/local/lib/R/site-library`	Packages built into Databricks Runtime
`/usr/lib/R/site-library`	Empty
`/usr/lib/R/library`	Base R packages

It's important to understand that the order defines the default behavior, as it's possible to add or remove values in `.libPaths()`. You'll almost certainly be adding values, because there's little reason to remove values.

PERSISTING A PACKAGE

When going down the route of persisting packages, to avoid making things messy, you should consider how they're organized and managed long term. Some practices you can consider include:

- Maintaining directories of packages per project, team or user
- Ensuring directories are specific to an R version (and potentially even a Databricks Runtime version)
- Coupling the use of persistence with `{renv}`

The recommended approach is to first install the libraries you want to persist on a cluster via a notebook. For example, to persist `{leaflet}` to a volume:

```
install.packages("leaflet")

# determine where the package was installed
pkg_location <- find.package("leaflet")

# move package to volume
new_pkg_location <- "/Volumes/<catalog>/<schema>/<volume>/my_packages"
file.copy(from = pkg_location, to = new_pkg_location, recursive = TRUE)
```

At this point the package is persisted, but if you restart the cluster or detach and reattach and try to load `{leaflet}`, it will fail to load.

The last step is to adjust `.libPaths()` to include the volume path, appending it to the existing paths:

```
# Adjust .libPaths, making new_pkg_location the first path
.libPaths(c(new_pkg_location, .libPaths()))
```

We recommend against making it the first value. See Adjusting .libPaths() to learn why.

ADJUSTING `.libPaths()`

Given that `.libPaths()` can return six values in a notebook, you might wonder if there's a "best" position to add your new volume path to. We don't recommend prepending `.libPaths()` with volume paths because packages will attempt to install to the first value and you can't directly install packages to a volume path (due to volumes being cloud storage and not a true file system). This is why the example for persisting a package copies to a volume *after* installation. That leaves a couple other options for where to add your path to `.libPaths()`.

A safe default is to add a path *after* the cluster libraries location (currently third). This will make it appear as if Databricks Runtime has been extended to include packages in the volume paths. Alternatively, you could add it after the first path and all users will still have the notebook-scoped package behavior by default. However, cluster libraries may not load if they appear in the earlier paths under a different version. It'll be up to you to decide what works best for you.

An example of adjusting `.libPaths()` looks like:

```
volume_pkgs <- "/Volumes/<catalog>/<schema>/<volume>/my_packages"
.libPaths(new = append(.libPaths(), volume_pkgs, after = 3))
```

Helpful functions

Here are some functions for copying packages and adjusting `.libPaths()` that may make your life easier.

```
copy_package <- function(name, destination) {
  package_loc <- find.package(name)
  file.copy(from = package_loc, to = destination, recursive = TRUE)
}

# e.g. move {ggplot2} to volume
copy_package("ggplot2", "/Volumes/<catalog>/<schema>/<volume>/my_packages")
```

```
add_lib_paths <- function(path, after, version = FALSE) {
  # Check if R version-specific path is needed
  if (version) {
    rver <- getRversion()
    lib_path <- file.path(path, rver)
  } else {
    lib_path <- file.path(path)
  }

  # Ensure the directory exists, create if not
  if (!file.exists(lib_path)) {
    dir.create(lib_path, recursive = TRUE)
  }

  lib_path <- normalizePath(lib_path, "/")

  # Inform the user about the primary package path
  message("primary package path is now ", lib_path)

  # Update the library paths with the new path
  .libPaths(new = append(.libPaths(), lib_path, after = after))

  # Return the library path
  lib_path
}
```

AVOIDING REPETITION

To avoid manually adjusting `.libPaths()` for every notebook, you can craft an init script or set environment variables, depending on the desired outcome.

> **Note:** In practice this interferes with how Databricks sets up the environment. Validate any changes thoroughly before rolling out to users.

Using an init script

This example init script appends to the existing `Renviron.site` file to ensure any settings defined as part of runtime are preserved. The last two lines of the script are setting `R_LIBS_SITE` and `R_LIBS_USER`. Changing these lines can give you granular control over order for anything after the first value of `.libPaths()`, as it's injected when the notebook session starts.

```
#!/bin/bash

# Define the variable 'volume_pkgs' with a path to the volume where 'my_packages' is located.
# <catalog>, <schema>, and <volume> are placeholders that should be replaced with actual values.
volume_pkgs=/Volumes/<catalog>/<schema>/<volume>/my_packages

# Append the R_LIBS_USER variable to the /etc/R/Renviron.site file
# This configures the R environment to include additional library paths for R packages.
cat <<EOF >> "/etc/R/Renviron.site"
R_LIBS_USER=%U:/databricks/spark/R/lib:/local_disk0/.ephemeral_nfs/cluster_libraries/r:$volume_pkgs
EOF

# The 'R_LIBS_USER' variable is set to include multiple directories for R libraries:
# - %U: User-specific library path.
# - /databricks/spark/R/lib: Path for Databricks Spark R libraries.
# - /local_disk0/.ephemeral_nfs/cluster_libraries/r: Path for cluster libraries.
# - $volume_pkgs: The path defined earlier for the 'my_packages' directory on the specified
volume.
```

Using environment variables

> **Note:** How Databricks Runtime defines and uses the R environment variables is something that may change and should be tested carefully, especially if upgrading runtime versions.

There are particular environment variables (`R_LIBS`, `R_LIBS_USER`, `R_LIBS_SITE`) that can be set to initialize the library search path (`.libPaths()`).

`R_LIBS` and `R_LIBS_USER` are defined as part of startup processes in Databricks Runtime and they'll be overridden if you set them from the cluster UI. It's easier to adjust them via an init script.

`R_LIBS_SITE` can be set via an environment variable but is referenced by `/etc/R/Renviron.site` and provides limited control over where the path will appear in the `.libPaths()` order. It'll appear fifth, after the packages included in Databricks Runtime, unless you're using an init script to alter `/etc/R/Renviron.site` directly.

Distributed Compute

Databricks can be used to write distributed computing applications with R in two senses: by using R packages that provide access to inherently distributed systems like Apache Spark and Delta Lake, or by distributing arbitrary R code across multiple CPUs and executing in parallel. We'll start with the former in order to fully grasp the latter.

Learning to scale with Databricks

Up to this point, we've emphasized that nearly all of the work you do in R can be done on Databricks. Now, it's time to take a step further into the Databricks ecosystem.

Apache Spark is *the* core engine of Databricks, and complementary open source projects like Delta Lake and Unity Catalog are designed to work together with it. **We recommend using Apache Spark to perform most of your daily data processing tasks** — feature engineering, ETL, exploratory data analysis — even if you're working with small to medium datasets. There are a few good reasons for this:

- **Scalability**: If the volume of data changes, you won't have to rewrite your code
- **Migration**: SQL and `dplyr` scripts are easily migrated to and from Spark
- **Skill development**: It'll make you a better R developer

To this end, let's begin with the two packages available for working with Spark in R: `sparklyr` and `SparkR`.

`sparklyr` vs. `SparkR`

R users find themselves in the unique position of having to choose between two APIs for Spark. **We recommend `sparklyr` over `SparkR`** due to its lower learning curve, better compatibility with Unity Catalog via Databricks Connect and Posit PBC's stewardship over the package. `SparkR` will also be deprecated with SparkR 4.0, so any new code you write is better off being written with `sparklyr`. However, if you have existing `SparkR` code and want to understand the differences between it and `sparklyr`, this section will be useful.

Stewardship

A key difference between the two packages lies in their origin and authorship. `SparkR` is the "official" package and is documented at spark.apache.org. Built by the Spark community and developers from Databricks, it looks and feels a lot like `PySpark` and adheres closely to the DataFrame API. For new R users, it's less approachable than packages they might be used to.

On the other hand, `sparklyr` originated from Posit PBC and is largely maintained by them. Its documentation is also hosted by Posit at spark.posit.co Given its origin, `sparklyr` is tightly integrated into the tidyverse, especially `dplyr`.

API differences

To understand the differences between APIs, let's read CSV files into Spark using both `sparklyr` and `SparkR` and compare the classes of each. In these examples we explicitly reference the package used for each function to avoid confusion.

```
## Read airlines dataset from 2008
airlinesDF <- SparkR::read.df("/databricks-datasets/asa/airlines/2008.csv",
                              source = "csv",
                              inferSchema = "true",
                              header = "true")

## Read airlines dataset from 2007
airlines_sdf <- sparklyr::spark_read_csv(sc, name = 'airlines',
                                         path = "/databricks-datasets/asa/airlines/2007.csv")

## Check the class of each loaded dataset
cat(c("Class of SparkR object:\n", class(airlinesDF), "\n\n"))
cat(c("Class of sparklyr object:\n", class(airlines_sdf)))


# output:
> Class of SparkR object:
>   SparkDataFrame
>
> Class of sparklyr object:
>  tbl_spark tbl_sql tbl_lazy tbl
```

Two distinct classes

Notice that `SparkR` and `sparklyr`, when used to read data, create objects that are two distinct classes. Now watch what happens when we run a `sparklyr` command on a SparkDataFrame and a `SparkR` command on a `tbl_spark`.

```
## Function from sparklyr on SparkR object
sparklyr::sdf_pivot(airlinesDF, DepDelay ~ UniqueCarrier)

# output:
> Error : Unable to retrieve a Spark DataFrame from object of class SparkDataFrame
```

```
## Function from SparkR on sparklyr object
SparkR::arrange(sparklyAirlines, "DepDelay")

# output:
> Error in (function (classes, fdef, mtable)  :
  unable to find an inherited method for function ‘arrange’ for signature ‘"tbl_spark",
"character"
```

Calling `SparkR` functions on `sparklyr` objects and vice versa doesn't work. Why not?

It doesn't work because `sparklyr` translates `dplyr` functions like `arrange()` into a SQL query plan that's used by Spark's SQL API. `SparkR` functions interact directly with the DataFrame API. This limits API interoperability and is one of the reasons why we don't recommend using both packages in a single script.

API interoperability

We recommend sticking with one package instead of mixing them in a code base. However, for the sake of learning, let's discuss the one way in which `SparkR` and `sparklyr` can talk to each other: Spark SQL. Recall that when we loaded the airlines data from 2007 into a `tbl_spark`, we specified the table name `airlines`. This table is registered with Spark SQL and can be referenced using the `sql()` function from `SparkR`. Executing SQL queries this way will return a Spark DataFrame:

```
## Use SparkR to query the 'airlines' table loaded into SparkSQL through sparklyr
top10delaysDF <- SparkR::sql("SELECT
                              UniqueCarrier,
                              DepDelay,
                              Origin
                              FROM
                              airlines
                              WHERE
                              DepDelay NOT LIKE 'NA'
                              ORDER BY DepDelay
                              DESC LIMIT 10")

## Check class of result
cat(c("Class of top10delaysDF: ", class(top10delaysDF), "\n\n"))

## Inspect the results
cat("Top 10 Airline Delays for 2007:\n")
head(top10delaysDF, 10)

# output:
> Class of top10delaysDF:  SparkDataFrame
>
> Top 10 Airline Delays for 2007:
>    UniqueCarrier DepDelay Origin
> 1             NW      999    EWR
> 2             AA      999    RNO
> 3             AA      999    PHL
> 4             MQ      998    RST
> 5             9E      997    SWF
> 6             AA      996    DFW
> 7             NW      996    DEN
> 8             MQ      995    IND
> 9             MQ      994    SJT
> 10            AA      993    MSY
```


From here on out, all examples will use `sparklyr`.

USING dplyr AND sparklyr

As mentioned previously, sparklyr is built adjacent to the broader tidyverse ecosystem, sharing a tight integration with dplyr. Most dplyr code is portable to sparklyr, though you need to understand a little bit of how the integration works to be productive. The following quick tutorial will help get you up to speed. See the official Databricks documentation for a longer version.

> **Note:** This section assumes you're using sparklyr in the Databricks Workspace. If you're using Databricks Connect to establish a remote connection with sparklyr, see this documentation and tutorial.

First load sparklyr and dplyr, then connect to Spark.

```
library(sparklyr)
library(dplyr)

## Connect to Spark
sc <- spark_connect(method = "databricks")
```

Now download some JSON data and read it into a Spark DataFrame with sparklyr::spark_read_json().

```
# Download data
system("wget https://raw.githubusercontent.com/prust/wikipedia-movie-data/master/movies.json -P /
dbfs/tmp/", ignore.stderr = TRUE)

# Read into Spark
jsonDF <-  spark_read_json(sc,
                           name = 'jsonTable',
                           path = "dbfs:/tmp/movies.json")

## Take a look at our DF
head(jsonDF)

> # Source: spark<?> [?? x 4]
>   cast       genres     title                                  year
>   <list>     <list>     <chr>                                 <dbl>
> 1 <list [0]> <list [0]> After Dark in Central Park             1900
> 2 <list [0]> <list [0]> Boarding School Girls' Pajama Parade   1900
> 3 <list [0]> <list [0]> Buffalo Bill's Wild West Parad         1900
> 4 <list [0]> <list [0]> Caught                                 1900
> 5 <list [0]> <list [0]> Clowns Spinning Hats                   1900
> 6 <list [0]> <list [2]> Capture of Boer Battery by British     1900
```

In this example, `jsonDF` is a Spark DataFrame, but the code would work just as well if it were an R DataFrame.

```
jsonDF |> group_by(year) |>
 count() |>
 arrange(desc(n))


> # Source:     spark<?> [?? x 2]
> # Groups:     year
> # Ordered by: desc(n)
>     year     n
>    <dbl> <dbl>
>  1  1919   634
>  2  1925   572
>  3  1936   504
>  4  1926   491
>  5  1924   480
>  6  1937   473
>  7  1943   465
>  8  1944   456
>  9  1935   446
> 10  1950   443
> # i more rows
```

SQL translation

This magic is possible because `sparklyr` uses SQL translation with `dplyr` to pass SQL statements to Spark. This can be expressed in SQL directly using `sparklyr::sdf_sql()` or `SparkR::sql()`.

```
sparklyr::sdf_sql(sc,
"SELECT year, COUNT(*) AS n
FROM jsonTable
GROUP BY year
ORDER BY n DESC"
)

SparkR::sql("SELECT year, COUNT(*) AS n
FROM jsonTable
GROUP BY year
ORDER BY n DESC"
)
```

If you have `dplyr` code and want to convert it to SQL, use `dbplyr::sql_render()` at the end of your command chain.

```
query <- dbplyr::sql_render(
       jsonDF |>
group_by(year) |>
       count() |>
arrange(desc(n))
)

print(query)

> <SQL> SELECT `year`, COUNT(*) AS `n`
> FROM `jsonTable`
> GROUP BY `year`
> ORDER BY `n` DESC
```

The query can then be directly passed to `sparklyr::sdf_sql()` and `SparkR::sql()`.

Using `dplyr::mutate()`

When you want to mutate data with `sparklyr`, you'll need to use Hive UDFs. Here's an example where we might normally use `lubridate`, but instead use Hive UDFs.

```
withDate <- jsonDF |>
              mutate(today = current_timestamp())

head(withDate)

> # Source: spark<?> [?? x 5]
>   cast       genres     title                            year today
>   <list>     <list>     <chr>                           <dbl> <dttm>
> 1 <list [0]> <list [0]> After Dark in Central Park      1900 2024-07-17 17:19:09
> 2 <list [0]> <list [0]> Boarding School Girls' Pajama…  1900 2024-07-17 17:19:09
> 3 <list [0]> <list [0]> Buffalo Bill's Wild West Parad  1900 2024-07-17 17:19:09
> 4 <list [0]> <list [0]> Caught                           1900 2024-07-17 17:19:09
> 5 <list [0]> <list [0]> Clowns Spinning Hats             1900 2024-07-17 17:19:09
> 6 <list [0]> <list [2]> Capture of Boer Battery by Br…  1900 2024-07-17 17:19:09
```

PRIMER FOR DELTA LAKE IN R

Delta Lake is arguably the technology that made lakehouse architecture possible. Use Delta Lake to manage the tables that you're working with in Databricks — for reads and writes, as well as updates, merges and deletes. When working with Delta Lake, you can always use SQL strings with `sparklyr::sdf_sql()`, but we'll show examples with `dplyr`, `dbplyr` and `sparklyr` where we can.

Writes

By default, tables written to Unity Catalog in Databricks will be in Delta Lake format.
With `spark_write_table()`:

```
# Using the jsonDF from the previous section
sparklyr::spark_write_table(
  x = jsonDF,
  name = "main.default.json_movie_table",
  mode = "overwrite"
)
```

Here we set mode to "`overwrite`". If you want to append new rows, switch it to "`append`".

Reads

To read data from Unity Catalog, use `dplyr::tbl() and dbplyr::in_catalog()`:

```
new_jsonDF <- dplyr::tbl(sc, dbplyr::in_catalog("main", "default", "json_movie_table")
```

Updates, merges and deletes

To make changes to existing tables, use `sparklyr::sdf_sql()`:

```
# Updates
sparklyr::sdf_sql(
      "UPDATE main.default.json_movie_table SET year = 2000 WHERE title = 'After Dark in Central
Park'"
)

# Merges
merge_df <- data.frame(title = c('Dune', 'Bespoke'), # data to merge
                       year = c(1967, 1900))
sdf_copy_to(sc, merge_df, name = 'merge_table")      # create temporary view in Spark SQL

sparklyr::sdf_sql(
      "MERGE INTO main.default.json_movie_table j
      USING merge_table as m
on d.year = m.year
       WHEN MATCHED THEN
  UPDATE SET *
WHEN NOT MATCHED
  THEN INSERT *"
)

# Deletes
sparklyr::sdf_sql(
      "DELETE FROM main.default.json_movie_table WHERE year = 1900"
)
```

Parallelizing arbitrary R code

One of R's greatest strengths is its ecosystem of over 20,000 open source packages, making the odds of finding a package to solve a specific problem good compared to other languages used in data science. A weakness, however, is R's scalability. Because R is single threaded by default and somewhat of a memory hog if you aren't careful, many R users seek ways to scale their code.

SCALING UP VS. OUT

What do R users do when their laptop processes a DataFrame in R too slowly or data won't fit in memory? Perhaps they downsample data or turn to packages like `doParallel` or `furrr` to parallelize R processes with the cores on their machine. If this fails, they might think of vertical scaling (scaling up) — getting a bigger machine with more cores and memory. This works to a point, but as data grows larger, this approach ultimately results in million dollar supercomputers. Good luck getting IT to provision one!

An alternative approach is to scale out horizontally, distributing or partitioning a large dataset across a cluster of cheap, commodity hardware. This paradigm is known as cluster computing, and lucky for Databricks users, Apache Spark is an *in-memory* cluster *computing engine*. Not only does Spark have a SQL and DataFrame interface, it supports execution of user-defined functions (UDFs) at nearly unlimited scale. This makes Spark powerful and flexible enough to tackle nearly any high-performance computing (HPC) workload.

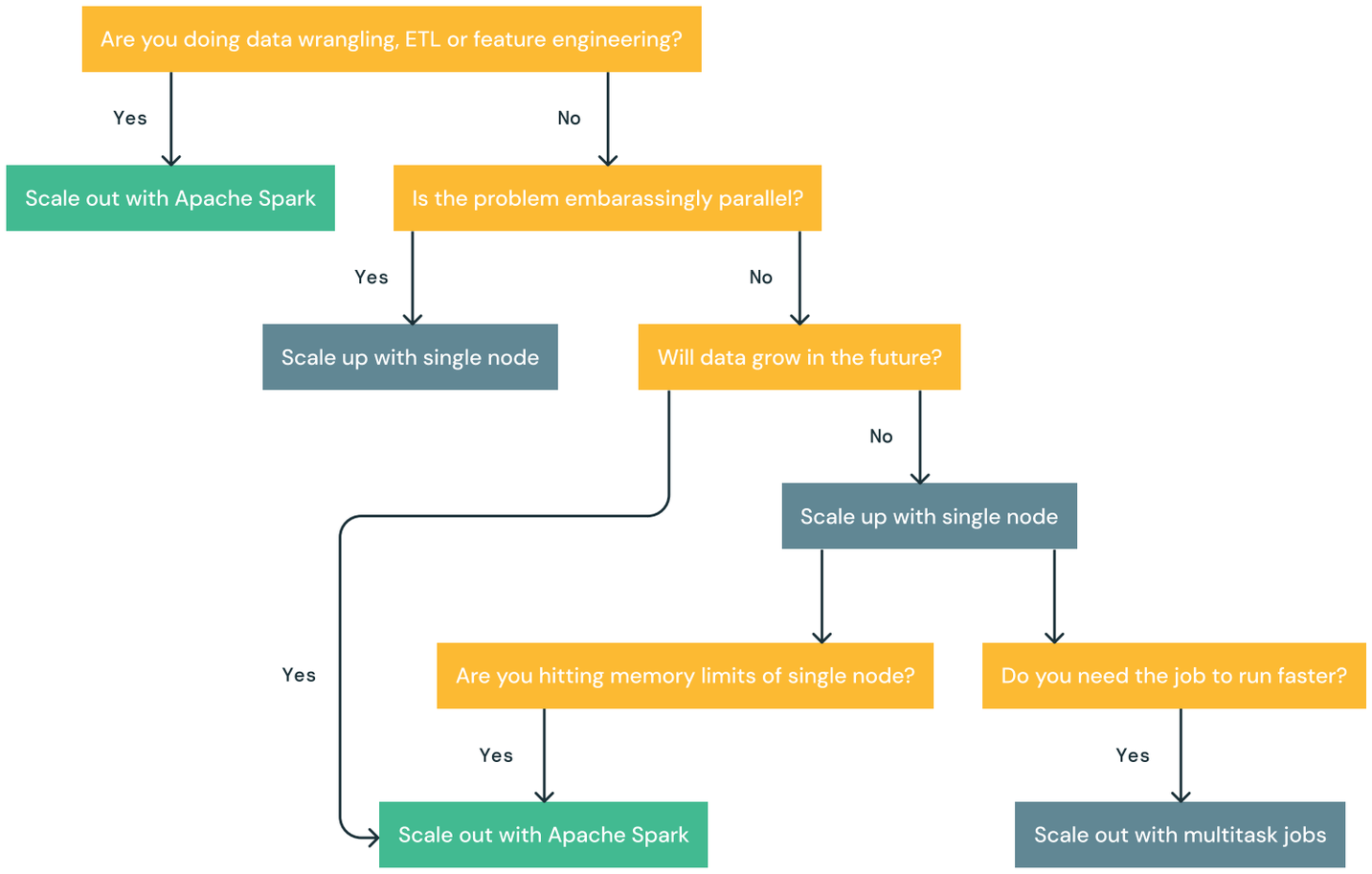
USE CASES

What would you do if you could scale your R code indefinitely? You might tackle some very difficult embarrassingly parallel problems:

- **Time series forecasting:** The demand of thousands of consumer products, the price of stocks in an index, the demand for electricity across nodes of a grid
- **Simulation:** Transportation schedules for fleets of vehicles or aircraft, stress testing portfolios, hypothesis testing in omics
- **Hyperparameter optimization:** Searching thousands of parameters in parallel to fit the best model
- **Inference:** Making billions of predictions with a model trained in R

Parallelizing R can make the impossible become possible. One customer was allotted 80 cores on their HPC system to run a vaccine search job, but the R code would've taken nearly a year to execute. With Databricks, they were able to scale up to 2000 cores and optimize the number of writes to disk, getting the job to complete in 2–3 days. In another case, the Minnesota Twins were able to simulate 300 billion pitches in a matter of days, not months.

If you're engaged in any kind of research, we strongly encourage you to consider what parallelism can make possible for you. If you aren't sure which approach to take, continue reading about when to scale up and when to scale out, using the decision tree below to guide you.



WHEN TO SCALE UP

If you already use R packages to parallelize your code and want access to more cores, or you just need more RAM, then scaling up with a larger single-node compute instance is a good place to start.

Quick migration from HPC systems

Migrating HPC workloads to Databricks is fairly simple. Install your R packages, load your data and run your code. All the popular packages for parallelizing R will work on Databricks, and you can easily provision a large single-node instance with 100+ cores. (We guarantee this is easier to get IT to agree to than a supercomputer.)

High-memory workloads

For problems that aren't easily parallelized, sometimes you just need more memory. For example, using an R package to fit a model on a very large dataset — you need all observations, and you can't split it up into a model per group. High-memory single-node compute instances can meet these requirements into the 10s of GBs, and some packages like `data.table` are optimized to work with large datasets.

A major advantage of using Databricks to scale up this way is that you can increase parallelism without pausing to first learn Apache Spark. Bear in mind though that Databricks compute is ephemeral by design, making it quite different from other HPC environments. If your code involves a lot of writing and reading files to disk, you'll need to update it to write to a Unity Catalog volume or copy the final files from disk to a volume. You'll need to reinstall packages every time you restart the single-node compute, forcing you to think more carefully about reproducibility.

WHEN TO SCALE OUT

Making the switch from scaling up to scaling out involves becoming familiar with user-defined functions (UDFs). Before getting deep into UDFs, here are the scenarios when scaling out is better than scaling up.

Long-running jobs

Losing your work due to errors or machines going down is one of the worst things that can happen with a long-running job. Apache Spark is fault tolerant by design — if a node goes down, another will take its place without interrupting or stopping the program. For *very* long-running applications it may make sense to build some checkpointing into it, but in general Spark will save you from the frustration of losing work.

Speeding up

If your job works but is bottlenecked by the number of cores available, switch to Spark. Spark can scale cores linearly by adding more nodes to the cluster. This lets you dial in the trade-off between execution speed, cost and time. **If you want or need to reduce the execution time of your code by an order of magnitude or two, Spark will probably get you there.** Of course, if a job is going to take a very long time, you may want to consider Spark anyway for its fault tolerance.

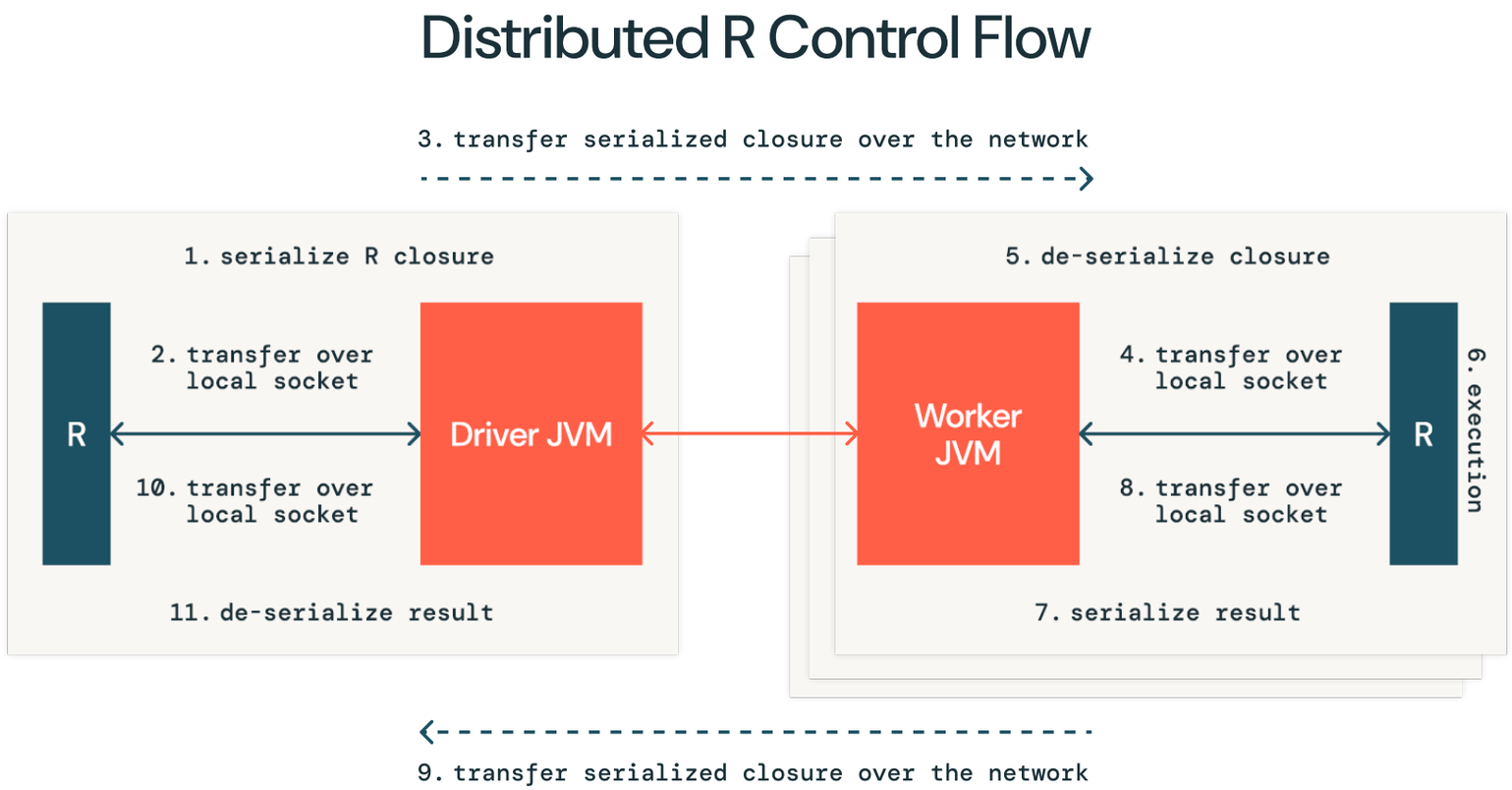
Bigger data

This is the obvious one. There's a limit to how much data R can comfortably process on a single machine, even when using `data.table` and parallelism. If you're working with 10s of GBs or consistently facing out-of-memory (OOM) errors, then it's time to switch to Spark. Spark will partition your data across multiple nodes and scale linearly into terabytes before you need to start being careful with what you're doing.

USER-DEFINED FUNCTIONS

Both `SparkR` and `sparklyr` support user-defined functions (UDFs) in R which allow you to scale out arbitrary R code across a cluster.

How do these functions work? The R process on the driver has to communicate with R processes on the worker nodes through a series of serialize/deserialize (ser/de) operations through the Java virtual machines (JVMs). To facilitate the performance of ser/de operations, Apache Arrow has been integrated with Spark and R. Arrow is widely used in the data and AI ecosystem these days, and it plays a critical role in making UDFs work well. We therefore highly recommend loading the `arrow` R package as part of any UDF work you plan on doing. With that being said, let's walk through the steps required to run arbitrary R code across a cluster.



There are a few important things to keep in mind with this control flow.

- There's overhead related to creating the R process and ser/de operations in each worker — UDFs will never be as fast as regular Spark code
- R processes on worker nodes are ephemeral. When the function being applied finishes execution, the process is shut down and all state is lost.
- As a result, you have to pass any contextual data and libraries along with your function to each worker to execute as expected

Since everything required for your UDF needs to be passed along with it, use notebook or cluster-scoped packages to ensure any dependencies for the UDF are available on each worker. This saves you time and gives you two options to reference a package within a UDF:

- Load the entire library — `library(broom)`
- Reference a specific function from the library namespace — `broom::tidy()`

Debugging UDFs can be hard enough, so we recommend using the second method to make it obvious which functions are being called at all times.

Distributed `apply()`

Between `sparklyr` and `SparkR` there are a number of options for how you can parallelize your R code, all of which are loosely modeled after the apply family of functions in R. An arbitrary R function can be applied to each *group* or each *partition* of a Spark DataFrame, or in the case of `SparkR::spark.lapply()`, to a list of elements in R. The following table summarizes all distributed apply functions.

Package	Function	Applied to	Input	Output
sparklyr	spark_apply	partition or group	tbl_spark	tbl_spark
SparkR	dapply	partition	Spark DataFrame	Spark DataFrame
SparkR	dapplyCollect	partition	Spark DataFrame	R data.frame
SparkR	gapply	group	Spark DataFrame	Spark DataFrame
SparkR	gapplyCollect	group	Spark DataFrame	R data.frame
SparkR	spark.lapply	list element	R list	R list

spark_apply()

Since we recommend sparklyr in general, we recommend learning spark_apply(). That's what we'll focus on in the examples below. Sometimes you might want to use SparkR::spark.lapply() due to its unique inputs and outputs, but the rest of the UDFs in SparkR can be replicated exactly in sparklyr.

sparklyr::spark_apply() takes a tbl_spark as input and must return a tbl_spark. By default it will execute the function against each partition of the data, but passing a column name to the group by argument will instead execute each group. spark_apply() will also distribute all of the contents of your local .libPaths() to each worker when you call it for the first time unless you set the packages parameter to FALSE. On the Databricks Platform, we recommend installing packages ahead of time and setting this parameter to FALSE.

Note: To get the best performance, we recommend:

1. Specifying the schema of the expected output to spark_apply. If you don't supply the schema, Spark will need to sample the output to infer it, which penalizes speed.
2. Loading the arrow package in R for better serialization/deserialization speed. It's available as part of DBR 14.3.

In the following examples we'll train a model on each group of `mtcars` with a distinct `cyl` value. This will be a simple linear model with `mpg` the dependent variable and all other variables (except `cyl`) independent. Furthermore, we use the `broom` package, available in Databricks Runtime, to tidy up the output. The results will be a `tbl_spark` with different coefficients for each group.

```
library(arrow)

# Connect to Spark
sc <- sparklyr::spark_connect(method  = "databricks")

# Push mtcars dataset to Spark
mtcars_sdf <- sparklyr::sdf_copy_to(sc, mtcars, overwrite = TRUE)

# Output schema
schema <- list(cyl = "double",
               term = "string",
               estimate = "double",
               std_error = "double",
               statistic = "double",
               p_value = "double")

## Fit a linear model on each group of data
results_sdf <- sparklyr::spark_apply(mtcars_sdf,
                                     group_by = "cyl",
                                     function(e){
                                       # 'e' is a data.frame containing all rows for each distinct
cyl
                                       tidymod <- broom::tidy(lm(mpg ~ ., data = e[, -2]))
                                       tidymod
                                     },
                                      # Specify schema
                                      columns = schema,
                                      # Do not copy packages to each worker
                                      packages = FALSE)

df <- sparklyr::collect(results_sdf)
```

DEBUGGING

Inevitably you'll need to debug UDFs while you craft them. Follow these tips to make this as painless as possible.

Start small

It's best to start by simply counting rows in each group or partition of data, making sure every row is flowing through the UDF. Then take a subset of input data and slowly introduce additional logic and the results that you return until you get a working prototype. Finally, slowly scale up the execution by adding more rows until you've submitted all of them.

Look at logs

When you hit errors, open up the `logs`. If the error is with Spark, you'll see the stack trace in the notebook. If the error is with your R code inside the UDF, this may not be shown in the notebook. You'll need to open up the Spark UI and check `stderr` from the worker logs to see the stack trace from the R process.

Monitor metrics

Once you have an error-free UDF, you can monitor the execution using compute metrics. These metrics contain detailed information on cluster utilization and can provide clues to where bottlenecks lie (e.g., indicating when CPUs are idle or swap memory is being used).

Additional resources

By this point you should know enough to start working with Spark, Delta and R, including how to scale arbitrary R code up and out. However, we're just scratching the surface. To continue your journey toward mastering Databricks, we recommend bookmarking and reading the following resources.

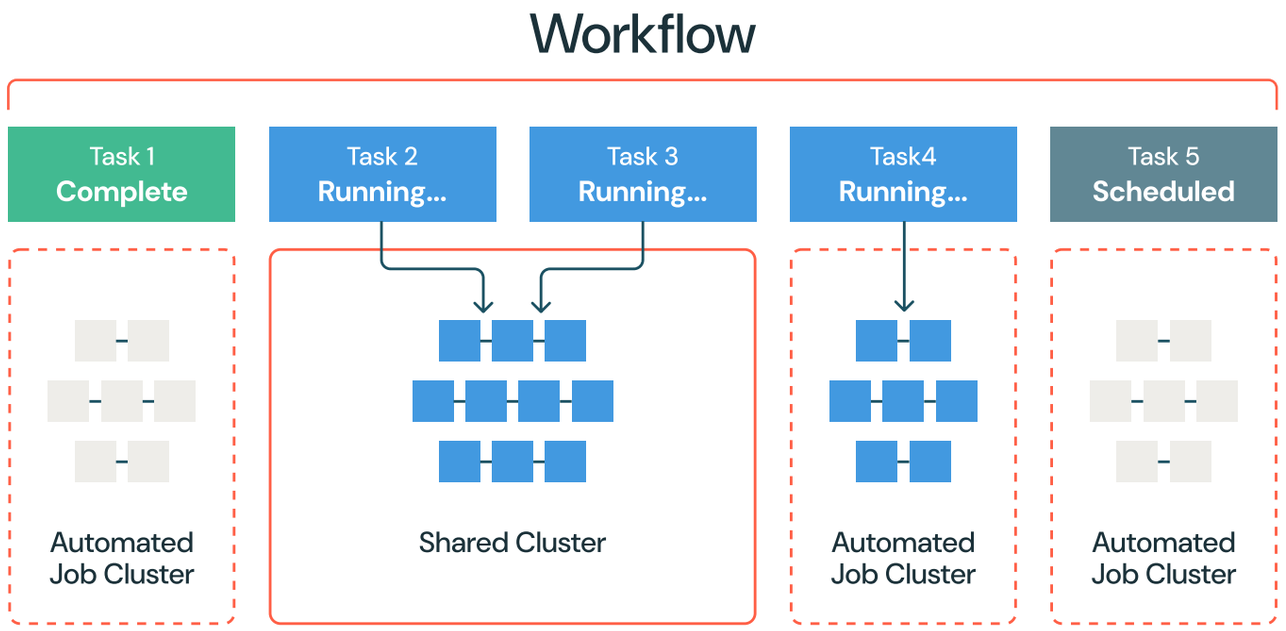
Spark and R

- The R in Spark — The definitive guide to R and Spark written by the authors of `sparklyr`
- `sparklyr` official documentation — A handy function reference with tutorials
- SparkR official documentation — An essential resource if you plan on using SparkR
- Collecting large results in `sparklyr` — Read this blog post to save yourself frustration
- Databricks Knowledge Base — Contains some useful troubleshooting tips

User-defined functions

- The R in Spark - Distributed R — The best resource for detailed information about how `spark_apply` works
- How the Minnesota Twins scaled pitch scenario analysis: Part I, Part II — A real-world use case with deep detail on debugging and overcoming bottlenecks with UDFs

Automation



Databricks Workflows is a fully managed orchestration service that enables users to automate a variety of tasks in their Databricks workspace. A workflow may be composed of a single task or a complex set of tasks with dependencies. Workflows include monitoring and debugging capabilities and are used by thousands of Databricks customers for business-critical workloads.

Automating with Databricks Workflows offers several compelling advantages for R users.

1. By offloading long-running tasks, you can significantly boost productivity and continue working interactively without interruption
2. Scheduling regular jobs for feature engineering, reporting or model training and inference keeps data products fresh and updated
3. You can run the same job with different parameters concurrently, optimizing resource utilization and scalability
4. Jobs compute is billed at a lower rate, making Databricks Workflows a more economical choice for extensive computations

For a more comprehensive look at Databricks Workflows, including best practices, we recommend reading Avnish Jain's blog posts (part 1, part 2).

Automating workflows from the UI

NOTEBOOKS VS. R SCRIPTS

Workflows can be created in a Databricks workspace by navigating to the Workflows page, or when working in a notebook, clicking the Schedule button. This is fairly straightforward when your code is in a notebook, but what about scheduling an R script? There's a Python script task type but no R script task type.

To run an R script as a workflow, there are a few options. First, you can import the R script into the workspace and Databricks will automatically convert it to a single-cell notebook for you. If you're using Git folders and want scripts in your repo to show up as notebooks in Databricks, then you'll need to change the file extension to `.r` (not `.R`) and add `# Databricks notebook source` as a comment to the first line of the script.

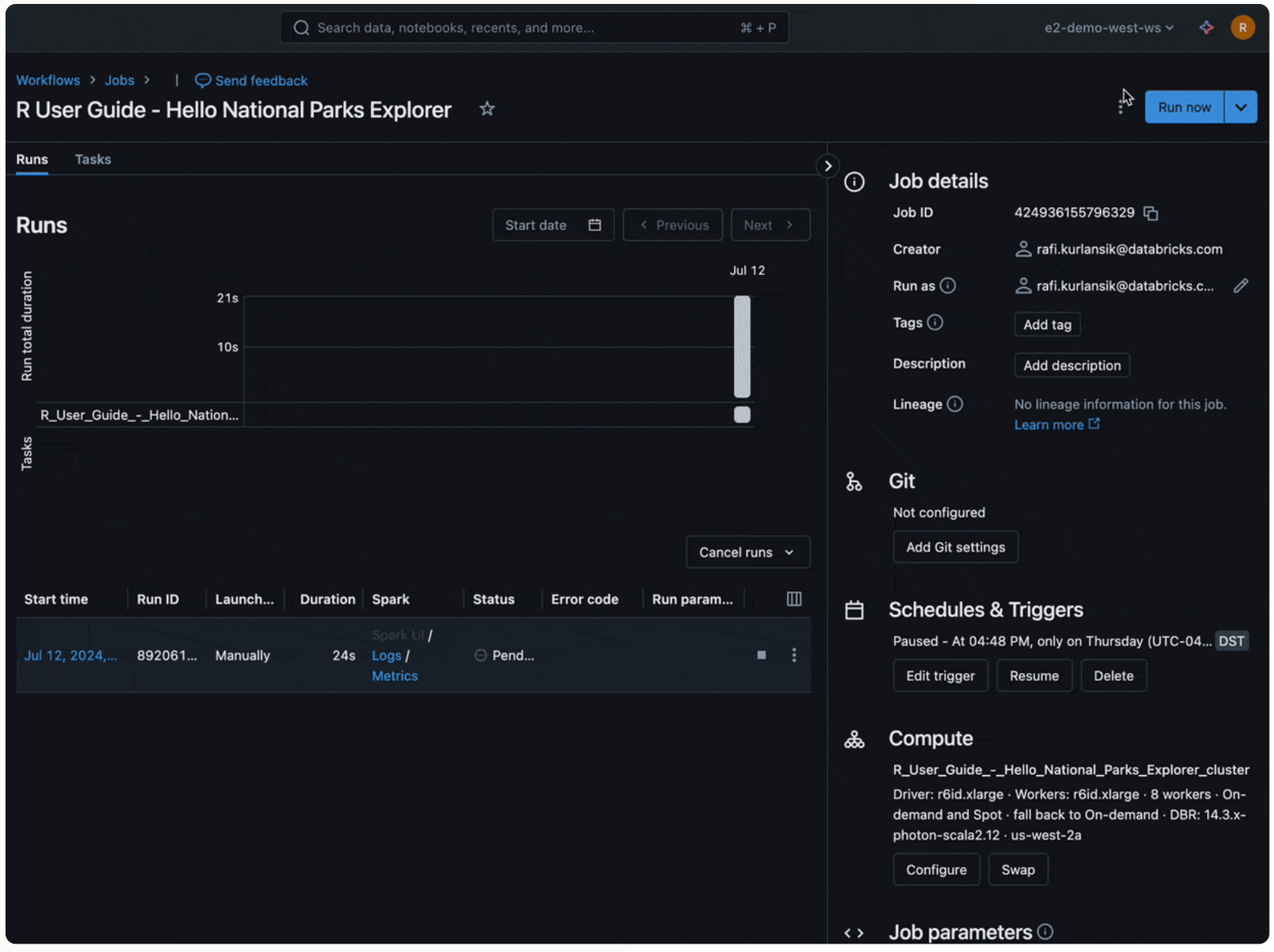
If you *don't* want to run your code as a notebook and prefer to run it as a `.R` file, then you'll need to use the Spark Submit task type in Databricks Workflows. Note that Spark Submit has several limitations and won't work with files in Git folders.

Automating workflows programmatically

DEFINING DATABRICKS ASSET BUNDLES

For projects that are going to be deployed into production environments, or ones with complex multitask workflows, we recommend taking the time to define them as Databricks Asset Bundles (DABs). They'll be much easier to maintain in the long run.

The fields in a DAB map 1:1 to the Databricks REST API, and we recommend authoring them in an IDE that supports YAML language servers. These language servers provide syntax checks and autocomplete, which will save you lots of debugging time. A shortcut for creating the YAML is to configure the workflow in the UI, then copy the YAML and save it to a file.



You'll still need to add the bundle name and targets to the YAML, but copy-pasting this way fills in the vast majority of the fields for you. The Databricks CLI can generate bundle configuration YAML from existing workflows too, with the `databricks bundle generate` command.

If we were to take our National Parks Explorer tutorial and DAB-ify it, we'd put the following in a `databricks.yml` file in the root directory of our project. This bundle configures a job named `R User Guide - Hello National Parks Explorer` to run every Thursday at 16:48:02 EST on a new cluster with Databricks Runtime 14.3, with single-user data access mode and email notifications on failure.

```
bundle:
  name: national_parks_explorer
resources:
  jobs:
    R_User_Guide_Hello_National_Parks_Explorer:
      name: R User Guide - Hello National Parks Explorer
      email_notifications:
        on_failure:
          - rafi.kurlansik@databricks.com
      schedule:
        quartz_cron_expression: 2 48 16 ? * Thu
        timezone_id: America/New_York
      tasks:
        - task_key: R_User_Guide_-_Hello_National_Parks_Explorer
          notebook_task:
            notebook_path: /Users/rafi.kurlansik@databricks.com/r_user_guide_2024/R User
              Guide - Hello National Parks Explorer
            base_parameters:
              date: 2024-05-25
              state: NJ
          new_cluster:
            spark_version: 14.3.x-scala2.12
            data_security_mode: SINGLE_USER
            num_workers: 8

targets:
  development:
    workspace:
      host: https://my-workspace.cloud.databricks.com
```

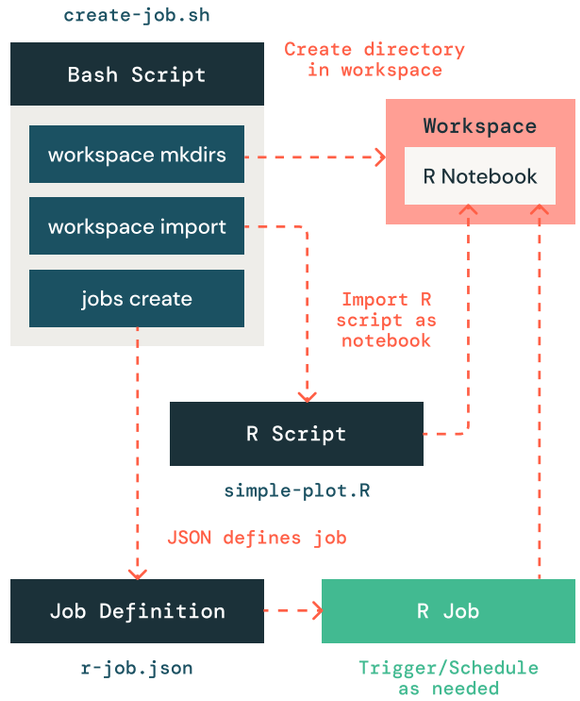
To deploy and run with the CLI, we execute the following two commands from the same root directory:

```
databricks bundle deploy national_parks_explorer
databricks bundle run national_parks_explorer
```

If you're on the fence about using DABs and are considering using the CLI directly — don't! It's much more complex and brittle to chain together the necessary commands to deploy code and other assets this way. You'll need to maintain your workflow configurations in a JSON file (which is arguably more difficult than YAML) and script all of the uploads and workspace object creation yourself. For example, imagine we have `simple-plot.R`, an R script that plots the `diamonds` dataset. These are the commands to create a workflow for the script using the CLI.

```
!/bin/bash
# create folder to upload script into
databricks workspace mkdirs /Shared/r-cli-demo/
# upload script to DBFS
databricks workspace import \
simple-plot.R \
/Shared/r-cli-demo/simple-plot \
--language 'R' \
--overwrite
# create job (notebook type)
databricks jobs create --json-file r-job.json --version 2.1
```

The interaction of the CLI with the workspace and JSON file can be visualized as follows.



Using `databricks bundle deploy` and defining the complexity of your workflow in one or more YAML files is the simpler and recommended way to automate Databricks Workflows. Stick to using the CLI for file uploads, secret management and other lightweight, ad hoc tasks.

`brickster`

`brickster` provides full coverage of the Databricks Workflows REST APIs and includes a vignette on workflow management. Before working through the example below, see the R development toolkit section to learn how to get authenticated. Then follow these three steps to launch your first workflow with `brickster`.

1. Import your code into the Databricks workspace
2. Create a new workflow with a notebook task
3. Launch the workflow

Importing code to the workspace

To get the best experience with Databricks Workflows, we recommend importing your R scripts into the Databricks workspace as a notebook. Let's create a `simple-notebook.r` file with a basic analysis of the `mtcars` dataset using the `tidyverse` package. You could replace this with any other file you want to test with.

First, install **brickster**.

```
# Install brickster
# With brickster
remotes::install_github("databrickslabs/brickster")
```

Next, save some sample code to the local disk.

```
# Save some R code to a local file
code <- "
library(tidyverse)

# Convert to tibble
mtcars_tb <- rownames_to_column(mtcars, var = 'car') %>%
  as_tibble()

# Data wrangling
mtcars_final <- mtcars_tb %>%
  filter(am == 1) %>%
  select(car, mpg, cyl, wt, am) %>%
  rename(cyclinder = cyl,
         weight = wt,
         transmission = am) %>%
  arrange(cyclinder, desc(mpg))

mtcars_final"

# Write .R file locally
temp_dir <- tempdir()
local_file <- file.path(temp_dir, "simple-notebook.r")
writeLines(text = code, con = local_file)
```

Now specify the path to where you want this file to be in the Databricks workspace and use `db_workspace_import()` to upload it. This will return an object ID.

```
# import to workspace
workspace_nb_path <- "/Users/rafi.kurlansik@databricks.com/brickster_demo/mtcars_analysis"

library(brickster)

db_workspace_import(
  path = workspace_nb_path,
  file = local_file,
  format = "SOURCE",
  language = "R",
  overwrite = TRUE
)
```

Create a new workflow.

```
# define a job task
simple_task <- job_task(
  task_key = "mtcars_analysis",
  description = "wrangling mtcars dataset",

  # specify a cluster for the job
  new_cluster = new_cluster(
    spark_version = "14.3.x-scala2.12",
    driver_node_type_id = "i3.xlarge",
    node_type_id = "i3.xlarge",
    num_workers = 0,
    cloud_attr = aws_attributes(ebs_volume_size = 32)
  ),
  # this task will be a notebook
  task = notebook_task(notebook_path = workspace_nb_path)
)

# create job with simple task
simple_task_job <- db_jobs_create(
  name = "first Workflow with brickster",
  tasks = job_tasks(simple_task),
  # 9am every day, paused currently
  schedule = cron_schedule(
    quartz_cron_expression = "0 0 9 * * ?",
    pause_status = "PAUSED"
  )
)
```

Run the workflow.

```
# Kick off the job using the job ID
job_run <- db_jobs_run_now(job_id = simple_task_job$job_id)
```

Your workflow should now be running. You can access the run URL in the Databricks workspace by digging into the run info.

```
run_info <- db_jobs_runs_get(job_run$run_id)

run_url <- run_info$tasks[[1]]$run_page_url

print(run_url)
```

DATABRICKS R SDK

Before working through this example, see the R development toolkit section to learn how to get authenticated. Similar to `brickster`, you'll launch a workflow in three steps:

1. Import your code into the Databricks workspace
2. Create a new workflow with a notebook task
3. Launch the workflow

Let's work through the same steps as before, but this time using the Databricks R SDK.

Importing code to the workspace

First, install the SDK.

```
# Install and load R SDK
remotes::install_github("databrickslabs/databricks-sdk-r")
library(databricks)
```

Now save some code to the local disk.

```
# Save some R code to a local file
code <- "
library(tidyverse)

# Convert to tibble
mtcars_tb <- rownames_to_column(mtcars, var = 'car') %>%
  as_tibble()

# Data wrangling
mtcars_final <- mtcars_tb %>%
  filter(am == 1) %>%
  select(car, mpg, cyl, wt, am) %>%
  rename(cyclinder = cyl,
         weight = wt,
         transmission = am) %>%
  arrange(cyclinder, desc(mpg))

mtcars_final"

# Write .R file locally
temp_dir <- tempdir()
local_file <- file.path(temp_dir, "simple-notebook.r")
writeLines(text = code, con = local_file)
```

Before you import `simple-notebook.r`, it needs to be base64 encoded. This is a single line of code using the `base64enc` package.

```
library(base64enc)
my_r_file <- base64encode(base::charToRaw(readChar(path, file.info(path)$size)))
```

To upload this file to the Databricks Platform, we'll use `workspaceImport()` from the R SDK.

```
# Specify where we want to import to
path <- "/Users/rafi.kurlansik@databricks.com/r-sdk-demo/mtcars_analysis"

# Import the file as a notebook
workspaceImport(client,
    path = path,
    content = my_r_file,
    format = "SOURCE",
    language = "R",
    overwrite = TRUE)
```

Creating the workflow

In the R SDK, new workflows are created using `jobsCreate()`. Tasks and their configurations are defined in a *list* (or list of lists), and we begin by defining a notebook task pointing to the code imported into the workspace.

```
# Specify the location of our notebook in the Workspace
path <- "/Users/rafi.kurlansik@databricks.com/r-sdk-demo/mtcars_analysis"

# Create notebook task
notebook_task <- list(
  notebook_path = path,
  source = "WORKSPACE"
)
```

To specify the compute resources for this simple R script, we set the number of workers to zero and choose an `i3.xlarge` node type. To browse available node types for your Databricks workspace, use `clustersListNodeType()` from the SDK. This command will return available types based on cloud (AWS, GCP, Azure) and region, including specs like total RAM and CPUs. Remember to pass values from the `node_type_id` column when working with parameters in the R SDK.

```r
# Compute configuration
single_node_config <- list(
  node_type_id = "i3.xlarge",
  num_workers = 0,
  spark_version = "14.2.x-cpu-ml-scala2.12"
)
```

Next, we declare any package dependencies for this workflow following the libraries data structure from the REST API. This data structure boils down to a list of lists that can be constructed simply using `lapply`:

```r
# Vector of package names
packages <- c("tidyverse", "broom")

# Repository URL
repo_url <- "https://packagemanager.posit.co/cran/__linux__/focal/latest"

# Use lapply to create the nested list structure
package_list <- lapply(packages, function(pkg) {
  list(cran = list(package = pkg, repo = repo_url))
})
```

To create the workflow, combine the notebook task, compute config and package dependencies into — you guessed it — a list that we pass to `jobsCreate()`.

```r
# Putting it all together
mtcars_analysis_task <- list(
  notebook_task = notebook_task, # task type
  task_key = "mtcars_analysis",  # name of the task
  new_cluster = single_node,     # compute config
  libraries = package_list       # dependencies
)

# Create job
response <- jobsCreate(
  client,
  name = "my_first_workflow",
  tasks = mtcars_analysis_task
)
```

Run this code and voilà, your workflow appears in the Databricks UI.

Launching the workflow

Triggering the newly created workflow is simple. Grab the `job_id` from the `jobsCreate()` API response and pass it to the `jobsRunNow()` function.

```
job_id <- response$job_id
run <- jobsRunNow(client = DatabricksClient(), job_id = job_id)
```

Checking results

The API response will include a `run_id` that can be passed to `jobsGetRun()`, which will return information about its status, including a URL to the Databricks Workflows UI for more details.

```
run_details <- jobsGetRun(client, run_id = run$run_id)
run_url <- run_details$url
print(run_url)
```

Advanced Topics

Moving beyond the foundations of R on Databricks, this section will give you the insight and tools to be successful with more complex techniques.

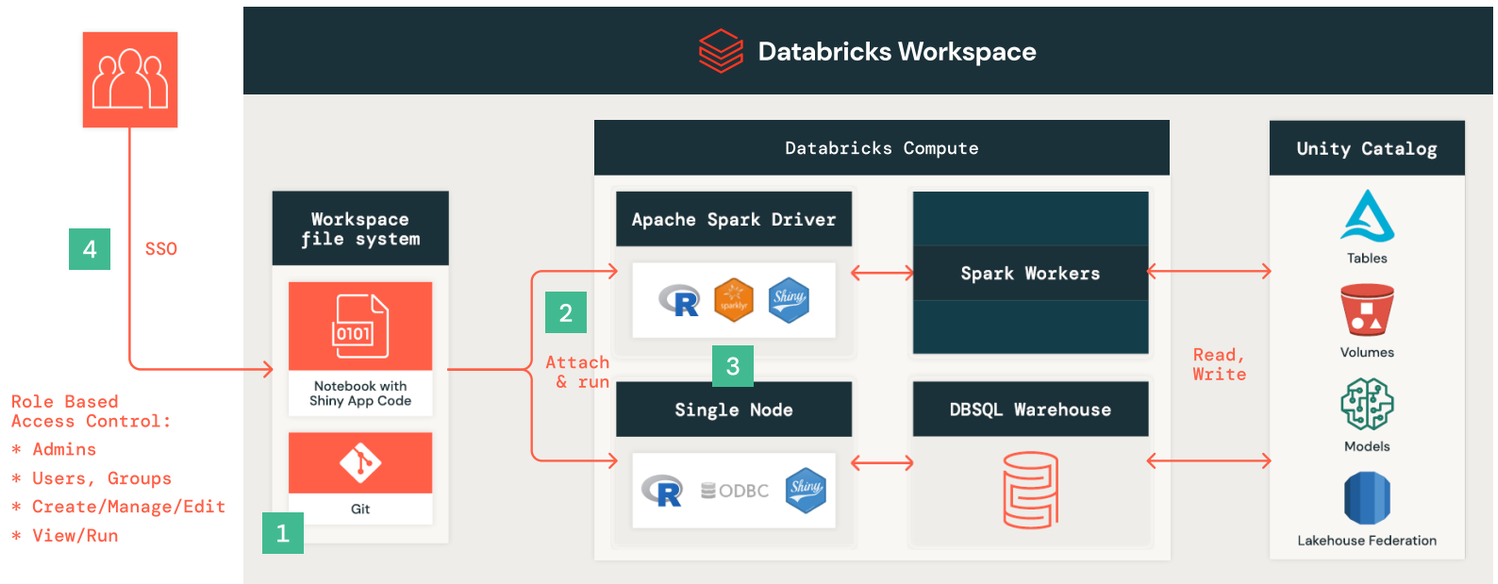
Shiny

Before we get into specifics about Shiny and the Databricks Platform, we highly recommend you review Mastering Shiny, the best resource for developing scalable and robust Shiny applications. This eBook presents all concepts with clear examples and covers the use of modules, testing and performance.

Note: We recommend hosting Shiny apps on the Databricks Platform for lightweight, internal use cases that do not have high concurrency or public internet access requirements. In those cases, or if you have many Shiny apps and could use help managing them over time, we recommend Posit Connect as the best enterprise solution for Shiny (and other data apps).

HOW SHINY WORKS ON DATABRICKS

On Databricks, Shiny apps are deployed via Databricks Notebooks. Assuming a deployed app, here is the high-level architecture.



To run a Shiny app in a Databricks workspace

1. Import your Shiny code into Databricks.
   While you can run Shiny using files, the launching of Shiny (e.g., `shiny::runApp()`) must be triggered from a Databricks Notebook cell.
2. Attach the notebook to Databricks compute and run it to launch the app.
   Launching this way generates a URL which can be shared with external users. Note that this URL is contingent on the specified cluster and port, so an alias would require a custom solution.
3. Use the single node with ODBC to DB SQL warehouse pattern instead of relying on a Spark-based connection like `sparklyr` or `SparkR`.
   For Shiny, we recommend this for the best performance, stability and access to Unity Catalog.
4. Users sign in to their Databricks workspace via SSO after following the URL.
   Any users of the Shiny app must have access to the notebook and be granted Can Attach permission to compute.

Running a Shiny application on the Databricks Platform is akin to operating it within a local RStudio session, as it doesn't employ a dedicated Shiny server. Authentication for the application is managed through the Databricks workspace, necessitating that users possess Can Attach privileges to access it.

Integration is facilitated by the proximity of the Spark session, making it easy to incorporate Spark's capabilities into the application. However, this setup is not intended for public hosting, and it includes a limitation with WebSocket time-outs, which typically occur at around 10 minutes.

HANDLING CONCURRENCY

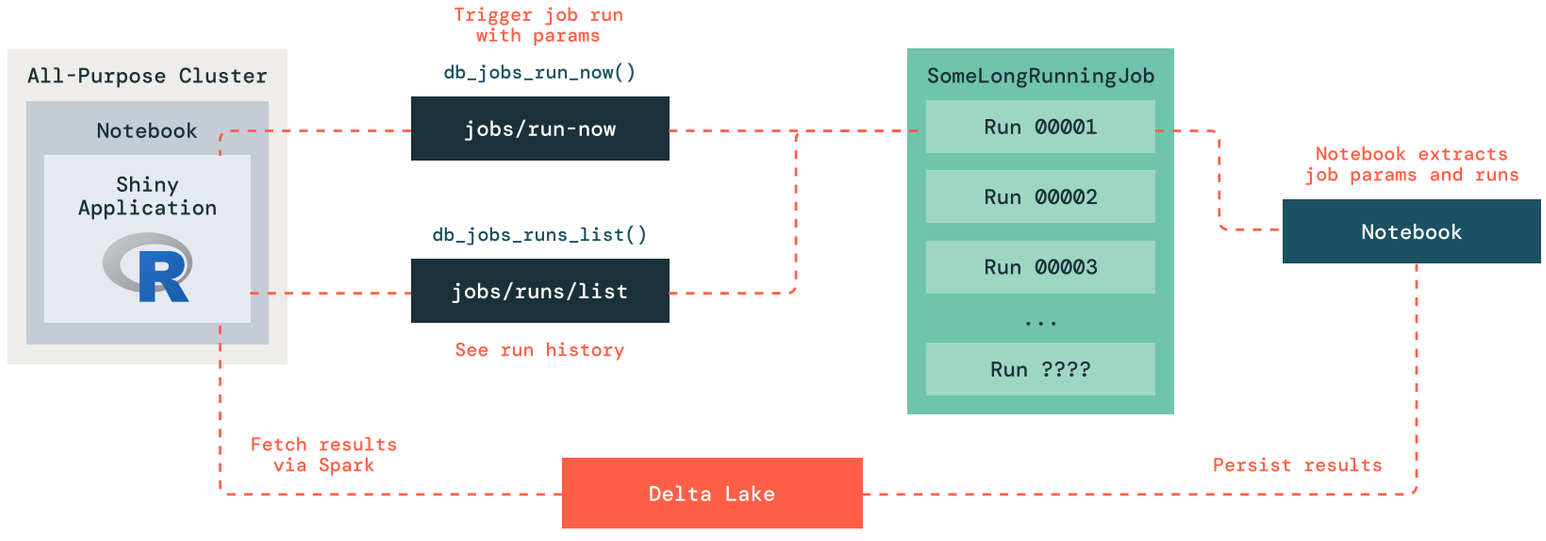
Promises

To implement asynchronous programming in R, you can utilize the `{promises}` package. This package offers a vignette that provides guidance on how to incorporate asynchronous code effectively.

Transitioning an existing application to asynchronous operations can be a significant endeavor. It's not as straightforward as simply enabling a feature; rather, it often involves extensive modifications across various layers of server code. For large and disorganized applications, it may be more practical to consider a complete rewrite rather than attempting to retrofit async functionality. This approach ensures better organization and integration of asynchronous principles throughout the application.

Background tasks and persistence

To enhance the power and flexibility of applications, the `{brickster}` package can be utilized to manage parameterized jobs in any language. Additionally, `{callr}` allows for the spawning of R threads or processes as needed, offering a potentially more straightforward approach compared to using `{promises}` for handling asynchronous operations.



The architecture diagram illustrates how a Shiny application can use the `{brickster}` package to manage and execute long-running jobs within a Databricks environment.

1. **All-purpose cluster:** The Shiny application, along with a supporting notebook, is deployed within an all-purpose cluster. This setup allows the application to leverage the computational resources and data handling capabilities of the cluster.
2. **Triggering jobs:** When the Shiny application needs to start a job, it uses the `db_jobs_run_now()` function from `{brickster}` to trigger a job run with specific parameters. This function communicates with the Databricks job API (`jobs/run-now`) to initiate the job.
3. **Managing jobs:** The jobs are managed and monitored through a sequence of runs, identified by unique run IDs (Run 00001, Run 00002, etc.). The history and status of these runs can be queried using the `db_jobs_runs_list()` function, which interacts with the `jobs/runs/list` endpoint to provide a list of all job runs.
4. **Executing notebooks:** The long-running job is defined within a notebook, which extracts the necessary parameters and executes the job. This notebook is part of the job definition and is crucial for the job's execution.

5. **Data persistence:** Upon completion, the results of the job are persisted in Delta Lake. Delta Lake provides a robust and scalable storage solution for large data outputs. The notebook handling the job ensures that the results are written back to this storage system.
6. **Result retrieval:** The Shiny application can fetch the results from Delta Lake via Spark, enabling efficient data retrieval and integration within the application interface.

When do to what?

Here are some questions you should ask yourself about the app you're building.

Question	If yes, consider using . . .
Are there operations which take a long time?	More than 30 seconds? Use background tasks or `{brickster}` to run a Databricks job and persist results. Under 30 seconds? Use `{promises}`.
Do I need to manage state over time?	Persisting results or metadata in Unity Catalog via ODBC, Spark or Volumes
Are there components of my application that are reused together often (e.g., many plots with same/similar reactive inputs)?	Modularize your application
Do you need to process large amounts of data on demand?	Use ODBC with a DB SQL warehouse. This yields the best price/performance without putting pressure on the application compute.
Do you have any of the considerations listed in this table in your application?	If not, the app should be simple, quick to run and work great as is. If not, revisit the questions with more scrutiny.

FAQ

1. **My libraries are taking too long to install. Why is it so slow? How can I speed it up?**
   Compute on Databricks is ephemeral by design, and Databricks Runtime is built on Ubuntu. Therefore, whenever compute resources are restarted, packages downloaded from CRAN must be recompiled and reinstalled. See Faster package loads for more information and techniques to speed up installation.
2. **My package is failing to install. Why is it failing?**
   This is usually caused by missing dependencies. See System dependencies for more details.
3. **Why can't I run R on standard compute?**
   For technical reasons related to CPU process isolation, R isn't supported with notebooks on standard compute. R users can share compute resources by using `sparklyr` with DB Connect V2, ODBC or assigning dedicated compute to a group. See Compute resources and data access for more information.
4. **Can I host Shiny apps on Databricks?**
   Databricks supports hosting Shiny apps by launching the app from within a notebook, which works well for lightweight deployments. We recommend Posit Connect for the best experience hosting Shiny apps (see the Shiny section for more details).
5. **Do I have to use notebooks with Databricks?**
   No, you can author code in files in the workspace, or you can use your IDE. See Choosing an editor for more.
6. **Do I have to run my R workflow/job as a notebook?**
   No, you can execute R scripts with Databricks Workflows without needing to convert to a notebook, though there are some limitations (see Notebooks vs. R scripts).
7. **How do I access Unity Catalog with R on shared compute**?
   R users can share resources by using remote connections to standard compute like `sparklyr` with DB Connect V2 and ODBC, or by assigning dedicated compute to a group. See Compute resources and data access for more information.
8. **How can I use RStudio/Positron with Databricks?**
   We recommend Posit Workbench as the best enterprise solution for using RStudio with Databricks. See Guidance for working with IDEs for more details.

9. **Does Model Serving support R models? Can I serve R models via REST API?**
   Model Serving doesn't support R models. If you want to use Model Serving, we recommend rewriting your models in Python. To serve R models via REST API, we recommend Posit Connect.
10. **How can I upgrade the version of R in Databricks?**
    See this article.
11. **How do I upgrade the version of a package in Databricks Runtime?**
    You can upgrade a package version in Databricks Runtime using any of the default installation methods. See Package management for more.
12. **What's the difference between SparkR and sparklyr? Which one should I use?**
    `SparkR` is more PySpark-y, while `sparklyr` is more dplyr-y. We recommend using `sparklyr` because it's easier to pick up and it supports Databricks Connect V2. See sparklyr vs. SparkR for more.
13. **Will my R code run faster in Databricks?**
    If you're importing R code that doesn't use Spark and running it as is, don't expect to see any performance improvements. If you're working with large datasets or parallelizing arbitrary R code, then Databricks can potentially accelerate your code by orders of magnitude. See Distributed Compute for more.

About Databricks

Databricks is the data and AI company. More than 10,000 organizations worldwide — including Block, Comcast, Condé Nast, Rivian, Shell and over 60% of the Fortune 500 — rely on the Databricks Data Intelligence Platform to take control of their data and put it to work with AI. Databricks is headquartered in San Francisco, with offices around the globe, and was founded by the original creators of Lakehouse, Apache Spark™, Delta Lake and MLflow. To learn more, follow Databricks on LinkedIn, X and Facebook.

Get started with a free trial of Databricks and start building data applications today.

START YOUR FREE TRIAL

databricks