Pip Technical Guidelines

R.Andres Castenada Ronak Shah

2025-12-04

Table of contents

Preface

Welcome to the **PIP Technical Guidelines**, a living technical resource for the Poverty and Inequality Platform (PIP) team at the World Bank.

This book serves as a centralized knowledge base for the technical infrastructure, development workflows, and best practices that support our data science and statistical work. As the PIP ecosystem continues to evolve—spanning R packages, Stata analysis, version control, automation, and AI-assisted coding—this guide provides practical, field-tested solutions to common technical challenges our team encounters.

Purpose

The primary objectives of this book are to:

1. **Document technical workflows** specific to PIP projects, including environment setup, development tools, and deployment processes
2. **Standardize best practices** across R, Stata, GitHub, VS Code, and other tools used in our daily work
3. **Accelerate onboarding** by providing new team members with clear, step-by-step instructions for configuring their development environment
4. **Promote consistency** in how we write, test, document, and maintain code across different projects and programming languages
5. **Preserve institutional knowledge** that would otherwise exist only in individual team members' setups or undocumented workflows

Scope

This book focuses on **technical infrastructure and development practices**, not statistical methodologies or economic theory. You'll find guidance on:

- Setting up R, RStudio, and Positron with unified configurations
- Configuring VS Code for Stata development with automated workflows
- Integrating Git and GitHub into daily work with proper authentication and branching strategies

- Implementing GitHub Actions for continuous integration and automated deployment
- Using GitHub Copilot responsibly and effectively, with mandatory protocols for code quality
- Optimizing performance on World Bank laptops and virtual machines
- Managing code coverage, testing frameworks, and documentation standards

Audience

This guide is written for PIP team members; data scientists, economists, and developers working with R, Stata, and related tools. While many sections assume familiarity with programming concepts, we provide step-by-step instructions designed to work in the World Bank's computing environment.

A Living Document

This book is continuously evolving. As we adopt new tools, refine existing workflows, or solve novel technical challenges, we add new chapters and update existing content. Each chapter represents accumulated team knowledge, battle-tested in real projects, and refined through practical experience.

We welcome contributions, corrections, and suggestions from all team members as we build this resource together.

Part I

R Setup and efficiency

1 Setting Up R, RStudio, and Positron to Use the Same Configuration Files

A unified and robust configuration setup for consistent R behavior across IDEs

1.1 Intro

When working with R, **your environment files** (`.Renviron` and `.Rprofile`) control how R behaves on startup—what packages it loads, what paths it uses, and how it connects to external resources (like CRAN mirrors or proxies).
However, depending on which IDE you use (RStudio, Positron, VS Code, or command-line R), these files might not always point to the same location.

A clean, unified setup ensures: - Reproducible environments across IDEs.
- Consistent library paths (`.libPaths()` output).
- Fewer surprises when running automated scripts, R Markdown documents, or package builds.
- Fewer "it works on my RStudio but not on Positron" headaches.

This guide describes the **recommended setup** for aligning your R environment across IDEs, including RStudio and Positron.

1.2 Step 1. Choose a canonical R configuration directory

Decide on a single folder that will store your personal R configuration files.
We recommend creating a dedicated folder at the root of your working directory, for example:

```
C:\WBG\R
```

Inside that folder, you'll have:

C:\WBG\R.Renviron
C:\WBG\R.Rprofile

1.3 Step 2. Create (or update) your configuration files

1.3.1 .Renviron

Create a text file named `.Renviron` at `C:\WBG\R\.Renviron` and add your preferred settings:

```
# C:/WBG/R/.Renviron
R_LIBS=C:/WBG/R/r-packages/%v
R_USER=C:/WBG/R
R_LIBS_USER=C:/WBG/R/r-packages/%v
RENV_PATHS_LIBRARY_ROOT=C:/WBG/R/r-packages/.renv/library
LANG=en_US.UTF-8
TZ=UTC
RENVIRON_LOADED_FROM=C:/WBG/R/.Renviron
```

This file defines **environment variables** that R reads at startup — for example, custom library locations, locale settings, or proxy credentials.

1.3.2 .Rprofile

Create a text file named `.Rprofile` at `C:\WBG\R\.Rprofile` and add the R code that should run automatically at startup:

```
# C:/WBG/R/.Rprofile
local({
  r <- getOption("repos")
  r["CRAN"] <- "https://cran.rstudio.com/"
  options(repos = r)
})

# Optional: print which file was loaded (for debugging)
cat(sprintf("[.Rprofile loaded from: %s]\n", normalizePath(sys.frame(1)$ofile, mustWork = FA
```

1.4 Step 3. Tell Windows and R where to find these files

The most reliable method is to set **persistent user environment variables** using `setx`, which works even in secure or constrained PowerShell environments (like corporate devices).

Open **PowerShell** or **Command Prompt** or **Terminal in Vscode** (no admin rights needed) and run:

```
setx R_ENVIRON_USER "C:\WBG\R\.Renviron"
setx R_PROFILE_USER "C:\WBG\R\.Rprofile"
setx HOME "C:\WBG\R"
setx R_USER "C:\WBG\R"
```

These commands instruct R to always look for `.Renviron` and `.Rprofile` in that directory, no matter which IDE you open.

Then **close and reopen** RStudio, Positron, or any terminal using R.

1.5 Step 4. Verify the configuration

After restarting R (in any IDE), run the following to confirm the setup:

```
path.expand("~")
normalizePath("~/.Renviron", mustWork = FALSE)
normalizePath("~/.Rprofile", mustWork = FALSE)
Sys.getenv(c("R_ENVIRON_USER", "R_PROFILE_USER", "R_LIBS_USER", "LANG", "TZ"))
```

Expected results:

- `~` resolves to `C:/WBG/R`
- `.Renviron` and `.Rprofile` point to the same files in that folder
- Your environment variables (`R_LIBS_USER`, etc.) are correctly loaded

---

1.6 Step 5. Optional cleanup

If you previously had `.Rprofile` files in other locations (like your `Documents` or `OneDrive` folder), remove or rename them by adding a `.bak` at the end (e.g., `.Rprofile.bak`). R automatically runs the **first** `.Rprofile` it finds, so duplicate files can cause confusion or run twice.

By following this setup, your R environment becomes predictable, portable, and IDE-agnostic — a professional-grade configuration for serious R development.

Part II

VSCode Setup

2 SetUp Vscode to work with Stata

Steps to configure VScode for Stata development.

2.1 Install VScode and Stata Extensions

1. Open "Company Portal" app on your computer.
2. Search for VS Code and install it.(Make sure Git Credential Manager is selected)
3. Search for GitForWindows and install it.
4. Open VS Code → Extensions (left sidebar), and install:
    - **Stata Enhanced** (syntax highlighting).
    - **Git Graph** (visualize branches).
    - **GitLens** (blame/insight).
    - **GitHub Pull Requests and Issues**.
    - **GitHub Copilot** (optional).
    - **GitHub Copilot Chat** (optional).

2.2 Configure `rundo` and `rundolines`

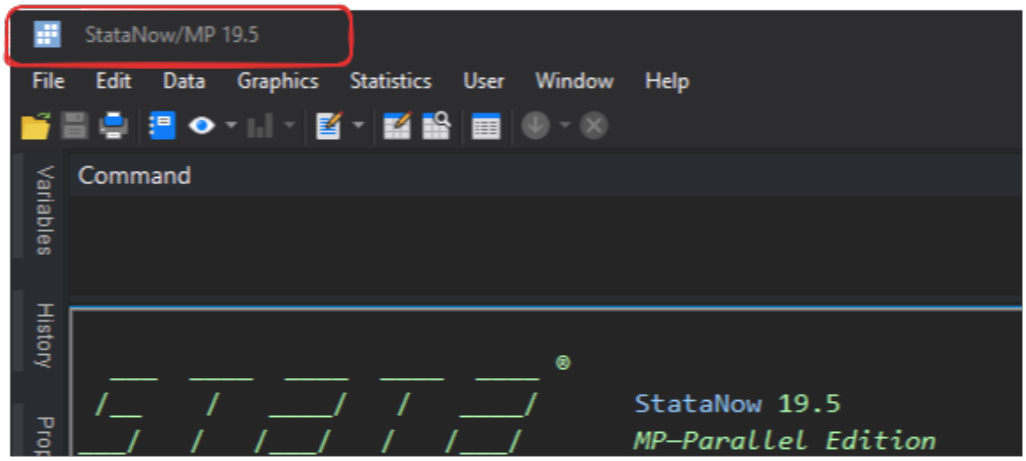
All the information in this section is based on Friedrich Huebler's article. If you need more detail, see his blog.

1. For convenience, we also mirror the files here: https://github.com/GPID-WB/Pip-Technical-Guidelines/raw/refs/heads/main/_data/rundofiles.zip
2. Extract the ZIP into `C:\ado\personal`. You'll see six files; you only need to modify two of them:

- `rundo.ini`
- `rundolines.ini`

3. Open **`rundo.ini`** and **`rundolines.ini`** and set:

   - `statapath` → full path to your Stata EXE, e.g.: `statapath = "C:\Program Files\Stata19\StataMP-64.exe"`
   - `statawin` → **exact** Stata window title, e.g.: `statawin  = "StataNow/MP 19.5"`



   - `statacmd` → shortcut that focuses Stata's **Command** window (default is `^1` = Ctrl+1).

4. The two INIs should be **identical** except for the filename. Keep both updated.

2.2.1 Recommended `.ini` tuning (for snappier runs)

These delays are safe and faster than the defaults:

```
[Delays]
clippause = 60   ; ms after copying selection to clipboard (40-80 is a good range)
winpause  = 120  ; ms between window ops (80-140 is a good range)
keypause  = 0    ; ms between keystrokes to Stata
```

If you ever get a "nothing happened" run, bump **winpause** up by ~20–40 ms.

Checklist

- Make sure the **window title** matches exactly (minor version changes matter).
- Keep **Stata open** before your first send (cold attaches are slower).

2.3 Set up User Tasks (global, once) to call `rundolines` / `rundo`

We use **User Tasks** so you don't have to repeat this per workspace.

1. Open **Command Palette** (`Ctrl+Shift+P`) → type: **Tasks: Open User Tasks**.
2. Paste this JSON (adjust paths to match your machine). NOTE:

- Use double backslashes \\ in paths.
- If `"tasks"` already exists, just add the two task objects inside the array.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Stata: Run selection/line (rundolines)",
      "type": "shell",
      "command": "\"C:\\ado\\personal\\rundolines.exe\"",
      "presentation": {
        "focus": false,
        "panel": "dedicated",
        "showReuseMessage": false
      }
    },
    {
      "label": "Stata: Do current file (rundo)",
      "type": "shell",
      "command": "\"C:\\ado\\personal\\rundo.exe\"",
      "args": ["\"${file}\""],
      "presentation": {
        "focus": false,
        "panel": "dedicated",
        "showReuseMessage": false
      }
    }
  ]
}
```

- **rundolines** sends the current **selection** (or current line if nothing is selected).
- **rundo** runs the **entire file** (saves first).

2.4 Bind your keyboard shortcuts (global, once)

1. Open **Command Palette** (`Ctrl+Shift+P`) → type: **Keyboard Shortcuts** → select: **Open Keyboard Shortcuts (JSON)**.
2. Add (or adapt keys if you already use F9/F10 or if you don't want to use F9/F10):
3. Houbler suggests not use any shortcut with the key `Ctrl` as it may get "stuck".

```
[
  {
    "key": "f9",
    "command": "workbench.action.tasks.runTask",
    "args": "Stata: Run selection/line (rundolines)",
    "when": "editorTextFocus && (resourceExtname =~ /\\.(do|ado|mata)$/ || (resourceScheme ==
  },
  {
    "key": "f10",
    "command": "workbench.action.tasks.runTask",
    "args": "Stata: Do current file (rundo)",
    "when": "editorTextFocus && (resourceExtname =~ /\\.(do|ado|mata)$/ || (resourceScheme ==
  }
]
```

- The **when** clauses limit these keys to Stata files.
- If you already mapped F9/F10 to something else, either change those or pick different keys (e.g., `f8` / `f9`).

  Tip: If you still have Code Runner installed, remove any bindings that clash (search "Run Code" in Keyboard Shortcuts and delete its keybindings). You can also disable/uninstall Code Runner entirely—you don't need it.

2.5 (Optional) Running Mata files directly

Two clean options; pick one.

2.5.1 A) Wrapper `.do` (recommended for standalone `.mata` files)

Create `.vscode/_run_mata.do` in your repo (or anywhere stable):

```
// _run_mata.do
args mfile
mata:
mata clear
mata do "`mfile'"
end
```

Add a **User Task** to invoke it with the current file:

```
{
  "label": "Stata: Mata do current file",
  "type": "shell",
  "command": "\"C:\\Program Files\\Stata19\\StataMP-64.exe\"",
  "args": ["/e", "do", "\"${workspaceFolder}\\.vscode\\_run_mata.do\"", "\"${file}\""]
}
```

And a keybinding for `.mata` files:

```
{
  "key": "f8",
  "command": "workbench.action.tasks.runTask",
  "args": "Stata: Mata do current file",
  "when": "editorTextFocus && (resourceExtname =~ /\\.(mata)$/)"
}
```

2.5.2 B) Mata blocks inside a `.do`

Place your Mata code inside a `mata: … end` block in a `.do`, then use **F9** (selection) or **F10** (whole file).

2.6 Quick sanity test (30 seconds)

1. Open a `.do`. Type and **select**:

```
display "hello from VS Code"
```

2. Press **F9** → Stata should execute the selection.
3. Press **F10** → Stata should run the entire file.

If nothing happens, check:

- **Tasks names** match your keybinding `args` exactly.
- You edited **User Tasks** (not workspace `.vscode/tasks.json`).
- No conflicting keybinding is intercepting F9/F10.

2.7 Troubleshooting & Performance

- **First run is slower** (1–3 s) due to task runner warm-up and first attach. Subsequent runs are often **sub-second** with tuned INIs.
- If sends are flaky, raise `winpause` slightly (e.g., from 120 → 140 ms).
- Verify `statawin` matches the **exact** window title (minor versions matter).
- Keep Stata open; cold-launches add seconds.
- Avoid network paths for ado/working dirs when possible.
- Consider AV exclusions for the Stata EXE, `rundolines.exe`, `rundo.exe`, and your ado/workspace.

3 SetUp Git & GitHub in VS Code

Practical steps to integrate Git and GitHub in VS Code using first-party and helper extensions.

3.1 Prerequisites

- **Git for Windows** installed (includes Git Credential Manager).
- **VS Code** installed.
- Make sure you're added to the World Bank GitHub organization using eservices/ request.

3.2 Install VScode and Stata Extensions

1. Open "Company Portal" app on your computer.
2. Search for VS Code and install it.
3. Search for GitForWindows and install it. (Make sure Git Credential Manager is selected)
4. Open VS Code → Extensions (left sidebar), and install:
    - **Stata Enhanced** (syntax highlighting).
    - **Git Graph** (visualize branches).
    - **GitLens** (blame/insight).
    - **GitHub Pull Requests and Issues**.
    - **GitHub Copilot** (optional).
    - **GitHub Copilot Chat** (optional).

3.3 First-time Git configuration (one-time)

Open **Terminal** in VS Code (Ctrl+‘) and run:

```
# Identify yourself in commits
git config --global user.name  "Your Name"
git config --global user.email "your_email@domain.com"

# Use main as default branch for new repos
git config --global init.defaultBranch main

# Line endings: recommended on Windows
git config --global core.autocrlf true  # checkout CRLF, commit LF

# Improve diffs for common texty files
git config --global diff.renameLimit 99999
git config --global fetch.prune true
```

3.4 Authenticate to GitHub

Go to your profile on VScode and login to GitHub.

Trying to **clone/push** over HTTPS will trigger a secure web flow:

1. Use a GitHub **personal access token (PAT)** or organization SSO as prompted.
2. Git Credential Manager securely stores/refreshes credentials.

3.5 Clone or publish a repository in VS Code

3.5.1 Clone an existing repository

- Command Palette → **Git: Clone** → paste repo URL (HTTPS or SSH).
- Choose a local folder → VS Code offers to open the folder.

3.5.2 Publish (create a new GitHub repo from a local folder)

1. Open your project folder in VS Code.
2. Source Control view → **Initialize Repository**.
3. Commit initial files.
4. Click **Publish to GitHub** (or run Command Palette → **Publish to GitHub**).
    - Choose a name, visibility (private/public), and org/user.

3.6 Power-ups

3.6.1 GitLens highlights

- **Inline blame** (who/when changed this line)
- **File & line history** with rich diffs
- **Visualize & compare branches**; open **Repositories** view
- **Open PR associated with a line/commit**
- **Code authorship heatmaps** and **commit search** by message, author, time

3.6.2 Git Graph highlights

- **Graph view** of all branches/tags
- **Drag & drop merges/rebases** (careful: follow team policy)
- Quick **reset**, **cherry-pick**, **revert** from context menu

3.6.3 Copilot & Copilot Chat

In order to access Copilot and Copilot Chat, You need to be added to the World Bank GitHub organization using eservices/ request.

Make sure you read the Getting Started Guide with GIthub Copilot.

Also, make sure you understand and follow the instructions in the General Github Copilot Usage Policy and guide.

- Copilot inline suggestions: code, tests, small refactors
- Copilot Chat for:
    - "Explain this diff" / "Write a PR description"
    - "Generate a `.gitignore` for Stata + VS Code"

– “Draft a release note from merged PRs since vX.Y”

3.7 Team conventions (recommended)

- **Branching**: feature branches from `main`; short, descriptive names.
- **Commits**: small, frequent, imperative subject line; reference issues (`#123`).
- **PRs**: clear description (what/why), checklist, reviewers; keep PRs small.
- **Merging**: prefer **squash** to keep history clean (unless you need merge commits).
- **CI**: make PR green before merge; don’t push directly to `main`.

3.8 Quick reference (commands)

```
# Start a repo
git init
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin <ssh-or-https-url>
git push -u origin main

# Typical feature flow
git checkout -b feature/my-change
# ...edit...
git add -A
git commit -m "Implement my change"
git push -u origin feature/my-change
# open PR in VS Code (GitHub PRs extension)

# Sync main
git checkout main
git pull --ff-only
```

3.9 Suggested VS Code settings (optional)

Create `.vscode/settings.json` in your repo to nudge consistent behavior:

```
{
  "git.enableSmartCommit": true,
  "git.confirmSync": false,
  "git.autofetch": true,
  "git.openRepositoryInParentFolders": "always",
  "files.eol": "\n",
  "editor.renderWhitespace": "boundary",
  "editor.rulers": [100],
  "diffEditor.ignoreTrimWhitespace": false
}
```

Part III

Copilot and AI Tools

4 Mandatory Protocols for Using GitHub Copilot in Technical Work

this document outlines mandatory protocols and recommended guidelines for using GitHub Copilot in technical work, ensuring productivity while maintaining code quality and team skills.

Purpose and Principles

GitHub Copilot is a powerful assistant that can increase productivity, generate boilerplate code quickly, and support experimentation. However, Copilot is not a substitute for engineering judgment, critical thinking, or expertise in R/Stata. This document establishes **Protocols** (mandatory actions) that must be followed and demonstrated during reviews.

These protocols have been drafted with the following principles in mind:

1. Improve productivity.
2. Ensure robustness, maintainability, and security of code.
3. Strengthen—not replace—team members' coding skills.
4. Prevent dependency creep, hallucinations, inefficiencies, and silent errors.

> **!** Important
>
> Every team member **must** follow these protocols when using Copilot for technical work. Team leads will verify compliance during code reviews.

Protocol 1 — Documenting Your Work With Copilot

Objective: We want every piece of Copilot-assisted work to be transparent, easy to review, and easy to understand—not only for you, but for anyone who touches the code later. Think of this as keeping a clear trail of "why we did what we did," without drowning anyone in hundreds of lines of chat history.

Start Every Copilot-Assisted Task With a Quick Declaration

Whenever you're about to use Copilot for a particular task (especially the ones formally assigned to you), like rewriting a function, adding a feature, building tests, or refactoring, let Copilot know you're starting a task.

You can use this short message (literally copy and paste; just change the `TASK_NAME`):

Prompt

I'm starting Task: `TASK_NAME`.
Please keep a concise running summary of our interaction, including:

- the major prompts I give
- the important decisions we make
- any dependencies added or removed
- assumptions or limitations we identify

At the end I'll ask you to produce a markdown summary.

Why do this? Because Copilot won't track your whole conversation. This little habit guarantees that your final summary is clean and complete.

Keep the Summary Updated as You Go (But Keep It Light)

As you work, Copilot may try several ideas, produce alternatives, or suggest changes. When something important happens, simply say things like:

- "Add this to the summary: we switched to `data.table` for speed."

- "Add this: the first approach was rejected because it created too many dependencies."
- "Record that we decided to remove `purrr` and use `collapse` instead."

You don't need to log every micro-step—just the meaningful ones.

This keeps your future self (and your teammates) very happy.

When You Finish, Ask Copilot for a Clean Markdown Summary

Once the task is done, ask Copilot:

Prompt

Now generate a markdown summary of this task, including:

- what the task was about
- the key prompts I used
- the major decisions or explanations you gave
- dependencies that were added, removed, or questioned
- any limitations or follow-up steps
- the final version of the function or script
- a short checklist showing how I followed the protocol

Save the file under `copilot_logs/TASK_NAME.md`

If you're working in an R package, make sure to include the `copilot_logs` folder in your `.Rbuildignore` file to prevent it from being included in the package build. You can do it by executing `usethis::use_build_ignore("copilot_logs")` or by manually adding the line `^copilot_logs/$` to the `.Rbuildignore` file.

Now, make sure to include it in your Pull Request.

This takes Copilot a few seconds and keeps our PRs tidy, transparent, and genuinely *reviewable*.

Protocol 2 — Making the Code Understandable

Objective: Once Copilot produces code, we need to ensure that it is understandable; not just to you, but to any teammate who might maintain it later. This protocol focuses on explanations, comments, and documentation, so the logic and intent of the code are always clear.

Our goal here is simple: **If Copilot wrote something, then Copilot should also help us understand it, both inside and outside the code.**

Ask Copilot to Explain the Code Step-by-Step

Right after Copilot generates a function, script, or refactor, ask:

> **Prompt**
>
> Explain this code step-by-step. Describe the purpose of each major block. List all assumptions you're making. Identify any cases where this code might break.

You're not expected to decipher the code alone. This step forces Copilot to surface the hidden logic behind its decisions, and gives you a clear foundation for understanding what's going on.

Include this explanation in your task summary.

Ask Copilot to Add In-Code Comments Explaining Both the "What" and the "Why"

Once the general explanation is clear, you must ask Copilot:

> **Prompt**
>
> Add clear comments directly inside the code. Explain:
>
> - what each major block does
> - why this approach is being used
> - why this logic or pattern is appropriate here

These comments are essential because:

- They ensure the logic is transparent.
- They help future maintainers (including you).

- They reduce the cognitive load when reviewing complex Copilot output.

We want readable, human-language explanations—not machine translations of the code.

Ask Copilot to Document All R Functions Using Roxygen2

If the output includes R functions, you must also request:

> **Prompt**
>
> Add proper Roxygen2 documentation for all R functions. Include:
>
> - `@title`
> - `@description`
> - `@param`
> - `@return`
> - `@examples` (if useful)
> - `@import` / `@importFrom` (only when truly necessary)

This step ensures:

- The function interfaces are clear.
- The expected inputs/outputs are known.
- Reviewers can understand intent without reading the body.
- Packages remain maintainable.

We want **professional-grade documentation** from the start, not as an afterthought.

Ask Copilot to Provide a Plain-Language Overview of the Logic

This is different from comments or Roxygen2 documentation.

You should ask:

> **Prompt**
>
> Write a short, plain-language explanation of how this code works. Pretend you're explaining it to a teammate seeing it for the first time.

This summary helps:

- New team members ramp up quickly.
- Reviewers understand context without digging.
- Future maintainers get immediate clarity.

This explanation goes into the task summary (alongside the step-by-step explanation).

Protocol 3 — Testing and Edge Cases

Objective: No Copilot-generated code should be accepted without being tested against realistic and edge-case scenarios. This protocol ensures that Copilot helps you design and implement tests that actually exercise the code.

Ask Copilot to Build a Validation Checklist

Before moving on, request:

> **Prompt**
>
> Create a validation checklist for this code. Include:
>
> - expected inputs and outputs
> - potential failure points
> - required dependencies
> - edge cases to test
> - assumptions this code relies on

This checklist goes into your summary and becomes a quick-reference tool for reviewers.

Ask Copilot to Generate Unit Tests and Edge Cases

No Copilot-generated code is considered "validated" until Copilot helps you test it.

For R:

Prompt

Generate `testthat` unit tests for this code. Cover:

- normal cases
- edge cases
- wrong input types
- missing values
- extreme or unusual data scenarios

For Stata:

Prompt

Generate Stata tests (do-file assertions or checks) for this code. Cover:

- realistic data
- corner cases
- unexpected inputs

These tests must be included in your PR.

(Optional but Recommended) Ask Copilot to Suggest Performance-Sensitive Tests

When working with large data or performance-critical code, you can also ask:

Prompt

Suggest performance-sensitive tests for this code. Identify:

- operations that may be slow on large data
- places where copies of large objects might be made
- scenarios that stress memory usage

This helps you spot performance issues early rather than discovering them in production.

Protocol 4 — Dependencies, Risks, and Final Validation Bundle

Objective: Before Copilot-assisted code reaches the repository, we want to be confident that it uses dependencies responsibly, avoids obvious risks, and has a complete "validation bundle" that reviewers can use to assess it quickly and fairly.

Ask Copilot to Run a Dependency and Risk Scan

If the code adds or modifies dependencies, request:

Prompt

Explain why each dependency is needed. Suggest ways to remove or replace unnecessary dependencies. Check for conflicts with `data.table`, `collapse`, or `rlang`. Identify any security or stability concerns (e.g., unsafe I/O, exposed paths, or risky patterns).

This helps prevent dependency creep and ensures the code fits our ecosystem and security expectations. Of course, this is only needed when you just started developing the code or when dependencies were changed.

Ask Copilot to Critique Its Own Code

Believe it or not, this might be one of the most powerful steps.

Prompt Copilot with:

Prompt

Review your own code as if you were performing a formal code review. Identify:

- inefficiencies
- unnecessary complexity
- risky assumptions
- unclear logic
- places where `data.table`, `collapse`, or `rlang` might be more appropriate
- opportunities to simplify or remove over-engineered patterns

Copilot is surprisingly good at pointing out its own flaws when prompted this way. Use its critique to improve the final version.

You can adapt the mention of `data.table`, `collapse`, and `rlang` to match the standards of your package or project. This are the ones that I (Andrés) use the most.

Package Everything Into a "Validation Bundle"

Before closing your task, ask Copilot:

Prompt

Add all validation materials to the task summary, including:

- step-by-step explanation of the code
- in-code comments added
- Roxygen2 documentation (for R functions)
- validation checklist
- plain-language explanation
- unit tests and edge cases
- dependency and risk analysis
- a brief summary of your self-critique and improvements made

This bundle provides everything reviewers need. It also ensures that anyone who maintains the code later won't be flying blind.

Why These Protocols Matter

The goal isn't to slow you down. It's to make sure that Copilot helps you, without replacing your thinking or lowering our standards.

With these protocols:

- You don't have to manually inspect every line in isolation.
- You don't have to fully understand the code immediately—but you always have the material to understand it.
- You don't have to memorize anything; explanations, comments, docs, and tests are all there.

Copilot explains, documents, critiques, and tests. You stay in control, make the decisions, and remain the author of the code.

That's the balance we're aiming for.

(EXTRA) Prompt Files Are Optional Tools, Not Documentation

Copilot has something called **Prompt Files**, which lets you save reusable instructions like:

- "Use `data.table`, not `dplyr`."
- "Follow `collapse` idioms."
- "Write tests using `testthat`."

These are great for giving Copilot consistent context, but they *don't* replace the documentation required above.

Use them if they help you, but they're optional.

5 Suggested Guidelines for Using GitHub Copilot in Technical Work

Blah

Purpose and Principles

blah blah blah

This is my “note box” with Tachyons.

Part IV

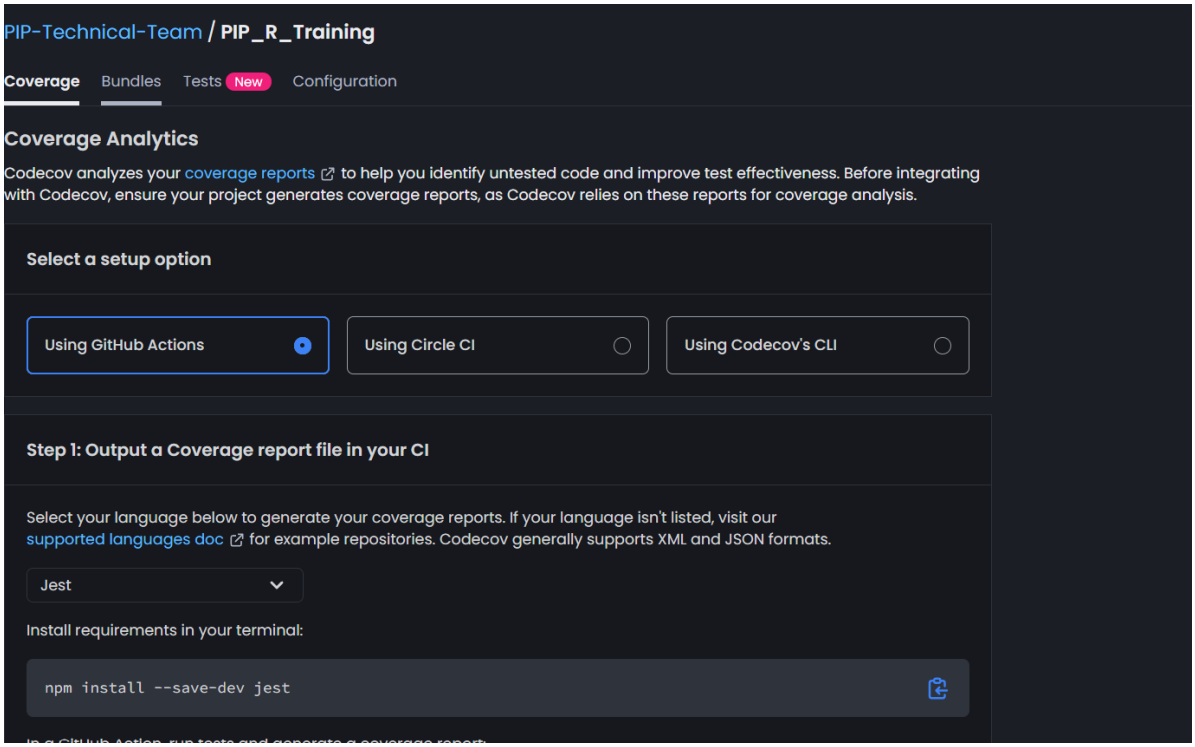
MISC Tools and Tips

6 Add code coverage badge to your GitHub Repository.

In this article we will learn how to add code coverage badge to your GitHub repository.

6.1 Codecov.io

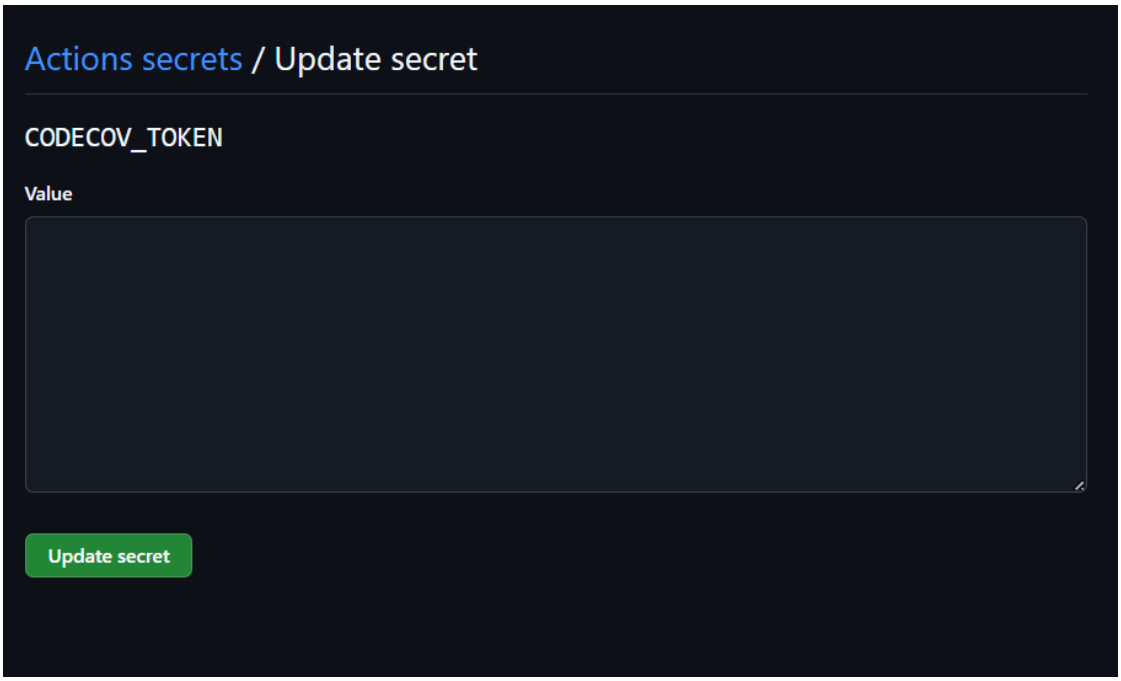
- Create an account at https://about.codecov.io/ , sign up with your GitHub account if you don't have an account already. **Codecov** is a popular tool for measuring and visualizing **code coverage** in software projects. It integrates with GitHub, GitLab, Bitbucket, and other CI/CD systems to provide insights into how much of your code is tested by your test suite.
- You can sync your private Github repositories on codecov platform to get started. If you want to add code coverage badge to a repository which is part of an organization (like PIP-Technical-Team, GPID-WB etc) then you need to be an admin of that organization. Admin role is needed because to sync the communication between Codecov.io with GitHub we need to generate a token which can only be done by admins.
- Once your repo is synced with codecov and you can see it there click on Configure to start the process. As an example it should give you this screen



- If you scroll below it will ask you to generate a repository secret, click on that to get a unique token for your repository and copy it.
- You can ignore rest of the steps mentioned on that page since those are very generic language agnostic steps and since we want to setup this for R packages, we have a better option which I will share below.

6.2 GitHub

- Now, moving to GitHub go to your repository. Click on Settings -> Secrets and Variables -> Actions -> Repository Secrets add the new token with name `CODECOV_TOKEN` and copy the token value which was generated in the previous step.



- Next, we are going to setup GitHub Action to run and calculate code coverage after every push. The calculated coverage report would be uploaded on codecov.io and would be visible on their dashboard.
- Additionally, I also added a possibility to run `R CMD CHECK` after every push. `R CMD check` is a tool that runs a series of automated checks on an R package to ensure it's correctly structured, documented, and error-free. It helps catch issues in code, tests, and documentation before sharing or submitting to CRAN. So it is like an additional validation that we have on our code.
- The new workflow file looks like below

```yaml
name: R-CMD-check and Codecov

on:
  push:
    branches: [master]
  pull_request:
    branches: [master]

jobs:
  R-CMD-check:
    runs-on: ubuntu-latest

    steps:
```

```
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up R
        uses: r-lib/actions/setup-r@v2

      - name: Set up pandoc
        uses: r-lib/actions/setup-pandoc@v2

      - name: Install dependencies
        run: |
          install.packages(c("remotes", "rcmdcheck", "covr"))
          remotes::install_deps(dependencies = TRUE)
        shell: Rscript {0}

      - name: Run R CMD check
        run: |
          rcmdcheck::rcmdcheck(args = "--no-manual", error_on = "warning")
        shell: Rscript {0}

      - name: Run test coverage
        run: |
          covr::codecov()
        shell: Rscript {0}
        env:
          CODECOV_TOKEN: ${{ secrets.CODECOV_TOKEN }}
```

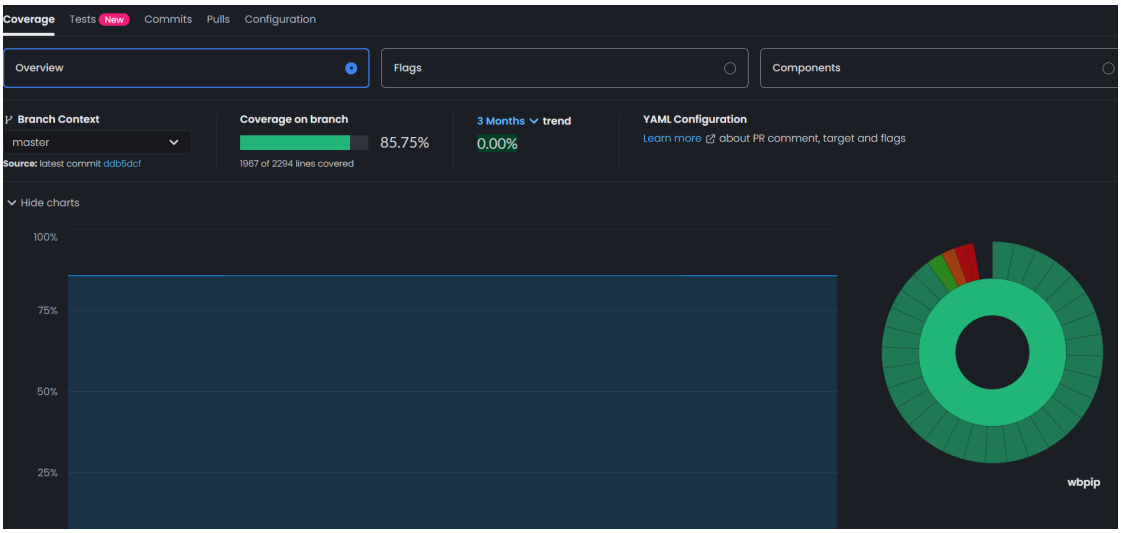
- This file is self explanatory but briefly, it checks out the repository that we want to run our action on, sets up R to run `R CMD CHECK` and finally generate code coverage report and upload it to codecov.io .
- One tip that I can share is to check if this workflow file works on your local branch before running on `master` branch. To do that you should temporarily enable the workflow file to run on your local branch. This can be done as below -

```
on:
  push:
    branches: [master, your-branch]
```

  where `your-branch` is the name of the local branch that you want to run the workflow for. Once you have verified that everything works as expected in the local branch, you can remove `your-branch` from the list again.

- Once the workflow runs successfully the dashboard on codecov.io should be updated and

you should see something like this



- Every time a push or PR is made to `master` the dashboard will be updated with latest data.

7 Setting up Github Actions for Auto Deployment of Quarto book

7.1 Introduction

One of the best parts of using Quarto for websites, blogs, or reports is how easily it integrates with GitHub Pages. With a simple GitHub Actions workflow, you can automatically render and publish your site every time you update your repository. In this post we are going to learn how we have enabled auto deployment for this quarto book.

7.2 Workflow

This is the workflow that we are using in Github Actions . Let's look at it one by one.

```yaml
on:
  workflow_dispatch:
  push:
    branches: main

name: Quarto Publish

jobs:
  build-deploy:
    runs-on: ubuntu-latest
    permissions:
      contents: write
    steps:
      - name: Check out repository
        uses: actions/checkout@v4

      - name: Set up Quarto
        uses: quarto-dev/quarto-actions/setup@v2
        env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

```
        with:
          tinytex: true
      - name: Render and Publish
        uses: quarto-dev/quarto-actions/publish@v2
        with:
          target: gh-pages
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

7.2.1 Triggering the Workflow

```
on:
  workflow_dispatch:
  push:
    branches: main
```

This tells GitHub Actions when to run the workflow. There are two triggers here:

- **`push to main`** – Any time you commit or merge changes into the `main` branch, the workflow runs
- **`workflow_dispatch`** – Allows you to manually trigger the workflow from the GitHub Actions tab in your repository. This is useful when you want to force a rebuild and republish without committing new changes.

7.2.2 Naming the workflow

```
name: Quarto Publish
```

This gives the workflow a friendly name that will appear in the Actions tab.

7.2.3 Defining the job

```
jobs:
  build-deploy:
    runs-on: ubuntu-latest
    permissions:
      contents: write
```

Here we're defining a single job called `build-deploy`.

- **`runs-on: ubuntu-latest`** – The job will run inside an Ubuntu-based virtual machine provided by GitHub.
- **`permissions: contents: write`** – The workflow needs permission to write to the repository (required for publishing to the `gh-pages` branch).

7.2.4 The Steps

7.2.4.1 1. Check out the repository

```yaml
- name: Check out repository
  uses: actions/checkout@v4
```

This makes your repository's files available in the workflow environment so Quarto can render your project.

7.2.4.2 2. Set up Quarto

```yaml
- name: Set up Quarto
  uses: quarto-dev/quarto-actions/setup@v2
  env:
    GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  with:
    tinytex: true
```

This installs Quarto in the workflow environment. The `tinytex: true` option ensures LaTeX support is available for rendering PDFs. The `GH_TOKEN` is github token repository secret that is added in Repo settings -> Secrets and Variables -> Actions . It is used for authentication when publishing.

7.2.4.3 3. Render and Publish

```yaml
- name: Render and Publish
  uses: quarto-dev/quarto-actions/publish@v2
  with:
    target: gh-pages
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

This step does two things:

1. **Renders** your Quarto project (turns `.qmd` files into HTML, PDF, or other output formats).
2. **Publishes** the output to the `gh-pages` branch, which GitHub Pages uses to serve your site. The `target: gh-pages` option ensures everything is pushed to the right branch.

7.3 Don't ignore the `.gitignore` file

Make sure that your `.gitignore` file excludes `_book`, `_site` folders. These are the folders where Quarto renders HTML/PDF files when testing them locally. These files should not be tracked since Github Actions will build them with our auto deployment process.

7.4 Conclusion :

With this workflow in place, the Quarto book will automatically rebuild and deploy whenever a push is made to the `main` branch or whenever we manually trigger the workflow. No more running commands locally or remembering to push generated files.

This is a clean, reproducible, and automated way to publish your Quarto projects using GitHub Pages. As a side not `usethis` package has a lot of good utility functions that helps you to set up similar workflow. You may explore using them. A good starting point is `usethis::use_github_action("render-quarto")`.

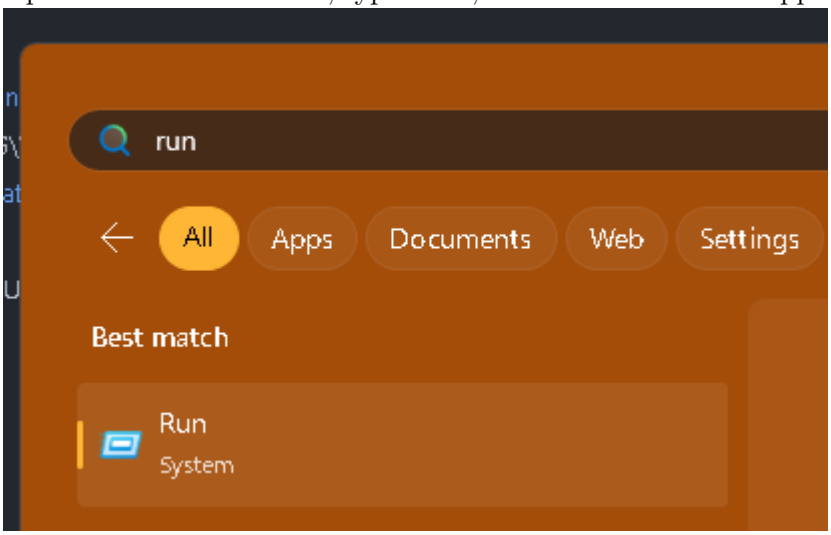
For reference the Quarto book is published here.

8 Improve Efficiency of a WB Laptop

There are a few things you can do to improve the efficiency of a WB laptop. These tips won't make your laptop *fast*, but they will help optimize performance and make it feel more responsive.

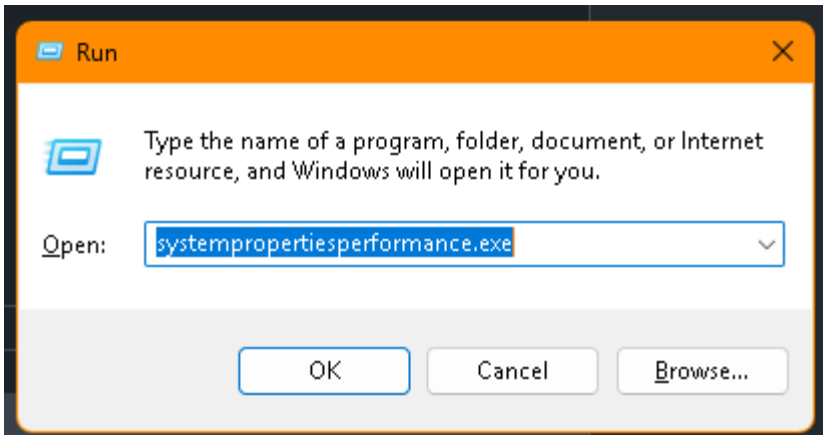
8.1 Modify System Properties for Performance

1. Open the Windows menu, type **Run**, and click on the *Run* app.



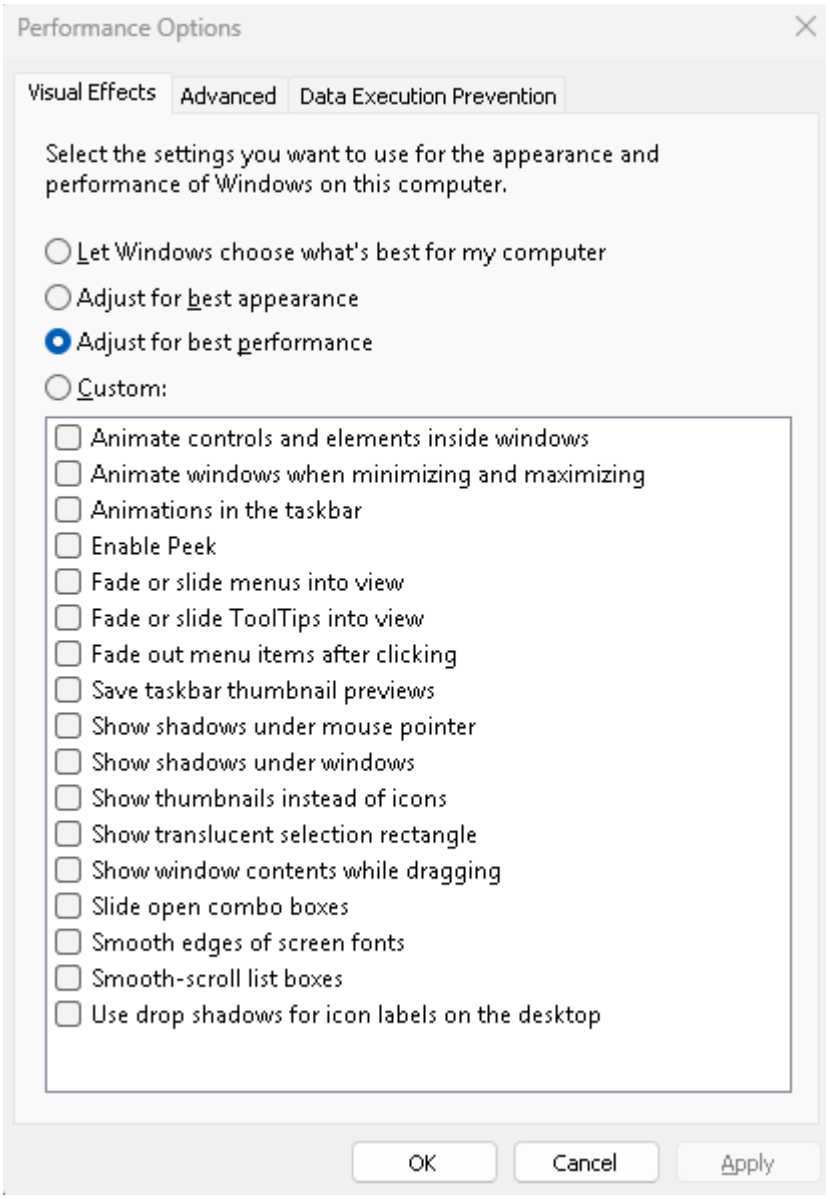
2. In the Run app, type the following command and hit **Enter**:

```
systempropertiesperformance.exe
```



3. In the *Performance Options* window, select the **Visual Effects** tab.

   - Click on **Adjust for best performance**.
   - Make sure all the checkboxes are unchecked.
   - Click **Apply** and then **OK**.

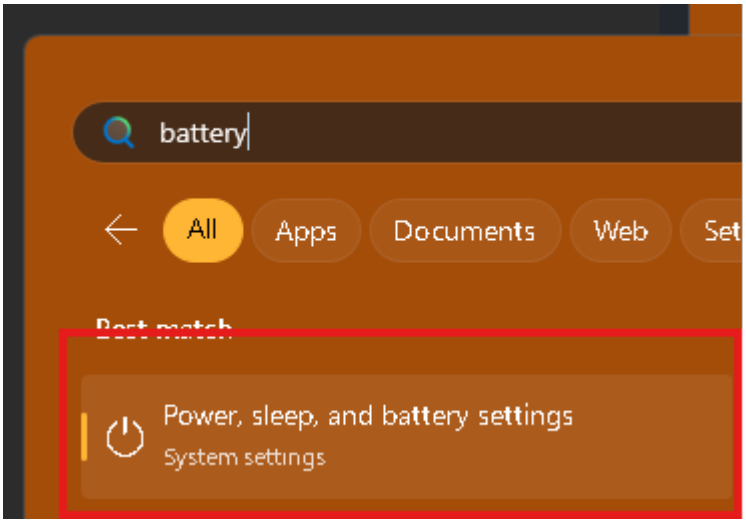


Warning

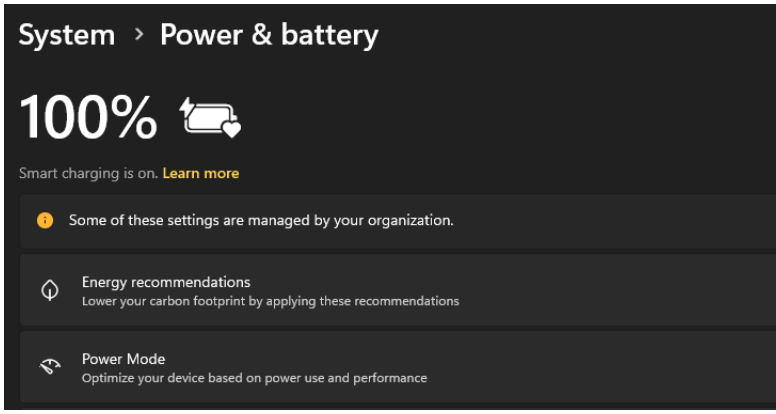
Note: This may affect the appearance of your system, since many visual effects will be disabled.

8.2 Battery Settings

1. Open **Battery Settings**.



2. Set the battery mode to **Best performance**.

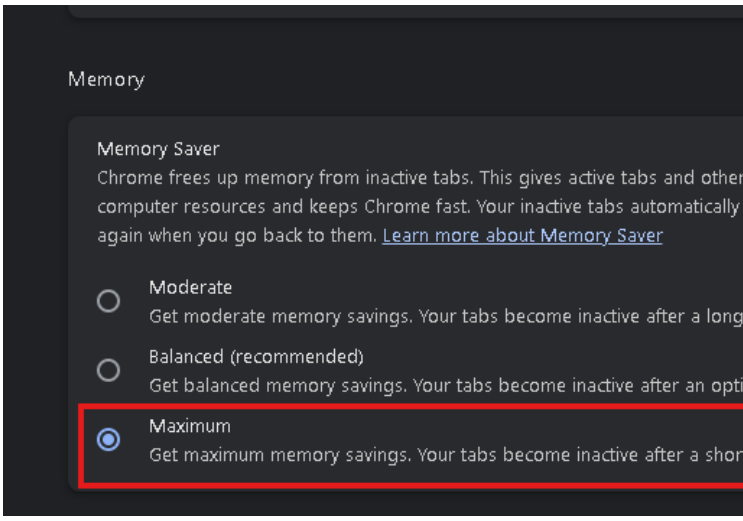


8.3 Google Chrome Settings

1. In Google Chrome, type the following in the address bar:

```
chrome://settings/performance
```

2. Under the **Memory** section, select **Maximum**.



9 How to interact with the Virtual Machine (VM) that hosts PIP

This guide explains how to connect to, inspect, and maintain the Linux Virtual Machines (VMs) hosting the PIP (Poverty & Inequality Platform) APIs. It covers user access, Docker container management, data storage, debugging, and safe cleanup procedures. This document is written for users with limited Linux/Docker experience who need to operate, inspect, or troubleshoot the PIP infrastructure safely.

9.1 Access and Session Management

9.1.1 Login to the VM

1. Go to PrivX.
2. Use **SSO login** (World Bank Authenticator App).
3. In the **Connections** tab, select the VM you want:
   - **Development:** `Linux-wzlxdpip01.worldbank.org`
   - **QA:** `Linux-wzlxqpip01.worldbank.org`
   - **Production:** `Linux-wzlxppip01.worldbank.org`
4. Once connected, switch to the `srvpip` user:

```
sudo su - srvpip
```

Shortcut	Action	Notes

9.1.2 Copy, Paste, and Keyboard Shortcuts

Shortcut	Action	Notes
Ctrl + C	Stop the current running process	e.g. stop an infinite loop or hanging R process
Ctrl + D	Log out of the current shell	Ends your session as `srvpip`
Shift + Ctrl + V	Paste from clipboard	Works in most PrivX terminals
Shift + Insert	Paste (alternative)	Often easier in remote shells

Notice that **Ctrl + C** does **not** copy text in terminal. You only need to highlight with your mouse the section you want to copy. Once you stopped highlighting, the text is copied to your clipboard automatically.

9.2 Understanding the Environment

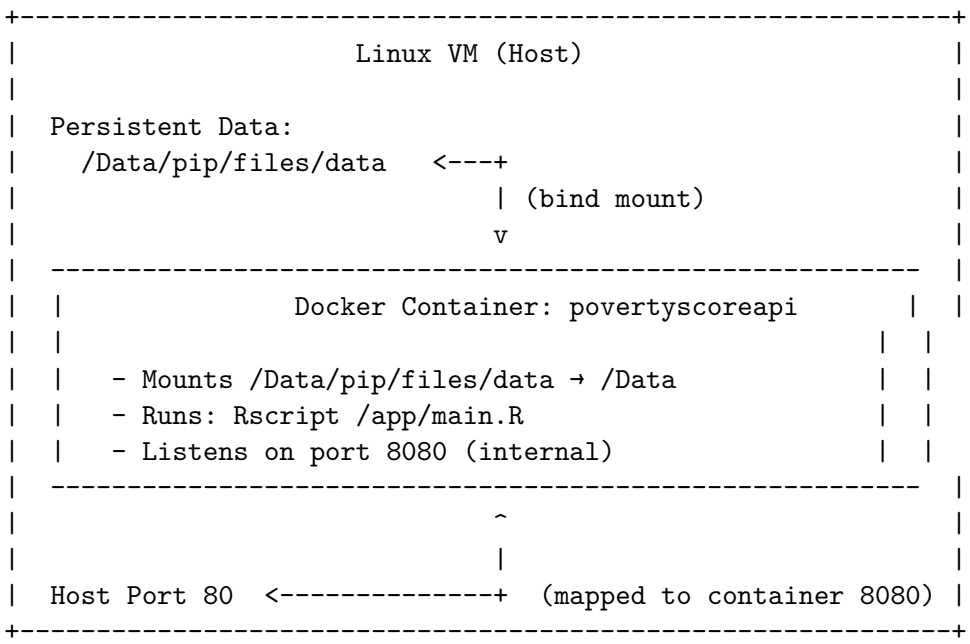
9.2.1 Users and Permissions

- **Your account:** initial login via PrivX (not root)
- **`srvpip` user:** used to manage Docker containers
- **`root` user:** required for system-level commands; use only with `sudo`

Common prefixes:

- `docker` → run Docker as normal user (often fails without permissions)
- `sudo docker` → correct way when managing containers
- `sudo su - srvpip` → switch to service account context

9.2.2 Architecture Overview



Legend: - Data written to /Data inside the container is actually stored on the host at /Data/pip/files/data. - The API runs inside the container on port 8080, but is accessible from outside the VM on port 80.

9.3 Essential Docker Commands (Cheat Sheet)

Task	Command	Notes
List all containers	`sudo docker ps -a`	Shows running & stopped
View logs	`sudo docker logs -f povertyscoreapi`	Add `--tail 200` to see last lines
Enter container shell	`sudo docker exec -it povertyscoreapi /bin/bash`	For interactive debugging

Task	Command	Notes
Check mounted volumes	`sudo docker inspect povertyscoreapi | grep Mounts -A 5`	See `/Data` source
Delete stopped containers	`sudo docker container prune`	Cleans unused ones
Delete old images	`sudo docker image prune -a`	Be cautious — removes all unused
Check Docker service status	`sudo systemctl status docker`	Confirms Docker is active

9.4 Managing Data

9.4.1 Data on the VM (Persistent)

- Main data location:

```
cd /Data/pip/files/data
```

- Inspect space usage:

```
df -h | grep Data
du -h --max-depth=1 | sort -h
```

- Preview files safely:

```
ls -lah
```

- View folder sizes:

```
du -sh *
```

9.4.1.1 Find large or old files

```
find /Data/pip/files/data -type f -mtime +365 | head -n 20
```

9.4.2 Data Inside Containers (Ephemeral)

Check where containers mount volumes:

```
sudo docker inspect povertyscoreapi | grep Mounts -A 5
```

You'll see:

```
"Mounts": [
  {
    "Type": "bind",
    "Source": "/Data/pip/files/data",
    "Destination": "/Data"
  }
]
```

That means:

- Inside container → `/Data`
- On host → `/Data/pip/files/data`

9.4.2.1 Access via container shell:

```
sudo docker exec -it povertyscoreapi ls -lh /Data
```

9.4.3 Safe Deletion and Cleanup

Preview first:

```
ls -d /Data/pip/files/data/project_*
```

Delete specific folders:

```
rm -rf /Data/pip/files/data/folder1 /Data/pip/files/data/folder2
```

Delete all contents but keep parent folder:

```
rm -rf /Data/pip/files/data/*
```

Move to trash instead of deleting:

```
mkdir -p /Data/pip/files/trash
mv /Data/pip/files/data/folder1 /Data/pip/files/trash/
```

Caution: `rm -rf` is irreversible — always check with `ls` first.

9.5 Diagnosing API or Container Issues

9.5.1 Check Container State

```
sudo docker ps -a
```

Get detailed exit info:

```
sudo docker inspect povertyscoreapi --format='ExitCode={{.State.ExitCode}} OOMKilled={{.State
```

- `ExitCode=0` → normal exit
- `ExitCode=137` → killed (likely out of memory)

9.5.2 View Logs

```
sudo docker logs povertyscoreapi | head -n 20   # startup logs
sudo docker logs --tail 200 povertyscoreapi     # recent logs
```

9.5.3 Reproduce Interactively

Start an interactive debug session:

```
sudo docker run --rm -it \
  --name povertyscoreapi-debug \
  -p 8080 \
  -v /Data/pip/files/data:/Data \
  itsesippscoreregistryprod.azurecr.io/povertyscoreapi:latest \
  /bin/bash
```

Inside:

```
Rscript /app/main.R
```

Press `Ctrl + C` to stop and `exit` to leave.

9.5.4 Restart or Rebuild Containers

```
sudo docker restart povertyscoreapi
```

Or rebuild completely:

```
sudo docker stop povertyscoreapi
sudo docker rm povertyscoreapi
sudo docker pull itsesippscoreregistryprod.azurecr.io/povertyscoreapi:latest
sudo docker run -d --name povertyscoreapi \
  -p 80:8080 \
  -v /Data/pip/files/data:/Data \
  itsesippscoreregistryprod.azurecr.io/povertyscoreapi:latest
```

9.6 Testing API Endpoints

9.6.1 From Inside the Container

```
curl http://localhost:80/api/v1/health-check
```

9.6.2 Loop through common endpoints

```
for ep in health-check pkgs-version data-signature gh-hash; do
  echo "---- Testing $ep ----"
  curl -s http://localhost:80/api/v1/$ep
  echo
done
```

9.6.3 From Outside (VM host or browser)

```
curl http://wzlxdpip01.worldbank.org/api/v1/health-check
```

9.7 Troubleshooting Common Errors

Symptom	Likely Cause	Fix
`curl: (7) Failed connect`	Container not running or wrong port	`sudo docker ps -a`
`Exited (137)`	Out of memory (OOMKilled)	Increase VM memory or limit API load
`authentication required`	Not logged in to Azure Container Registry	`sudo az acr login --name itsesippscoreregistryprod`
`permission denied` on `/Data`	Wrong user ownership	`sudo chown -R srvpip /Data`
No logs shown	Container failed before startup	Run `sudo docker inspect povertyscoreapi`

Symptom	Likely Cause	Fix
Port 8080 works but 80 doesn't	Port not mapped	Run container with `-p 80:8080`

9.8 Understanding Ports (80 vs 8080)

- Inside Docker, the API runs on **8080**:

```
pipapi::start_api(port = 8080, host = "0.0.0.0")
```

- Outside Docker (host or browser), you can map any port using:

```
-p 80:8080
```

  This means:

  `host_port:container_port`

  So users access:

  `http://wzlxdpip01.worldbank.org/api/v1/...`

  even though internally it listens on 8080.

9.9 Monitoring Resource Usage

Watch real-time CPU and memory:

```
sudo docker stats povertyscoreapi
```

Or globally:

```
top -u srvpip
```

9.10 Appendix

9.10.1 Check Disk Usage Quickly

```
df -h | grep Data
```

9.10.2 Check Docker Service Logs

```
sudo journalctl -u docker --since "2025-09-23 17:00"
```

9.10.3 Check R version inside container

```
sudo docker exec -it povertyscoreapi Rscript -e "R.version.string"
```

References