

Pip Technical Guidelines

R.Andres Castenada Ronak Shah

2025-08-03

Table of contents

Preface	5
1 Introduction	6
2 Add code coverage badge to your GitHub Repository.	7
2.1 Codecov.io	7
2.2 GitHub	8
3 Setting up Github Actions for Auto Deployment of Quarto book	12
3.1 Introduction	12
3.2 Workflow	12
3.2.1 Triggering the Workflow	13
3.2.2 Naming the workflow	13
3.2.3 Defining the job	13
3.2.4 The Steps	14
3.3 Don't ignore the <code>.gitignore</code> file	15
3.4 Conclusion :	15
4 Improve Efficiency of a WB Laptop	16
4.1 Modify System Properties for Performance	16
4.2 Battery Settings	19
4.3 Google Chrome Settings	19
5 Setting Up R, RStudio, and Positron to Use the Same Configuration Files	21
5.1 Intro	21
5.2 Step 1. Choose a canonical R configuration directory	21
5.3 Step 2. Create (or update) your configuration files	22
5.3.1 <code>.Renviron</code>	22
5.3.2 <code>.Rprofile</code>	22
5.4 Step 3. Tell Windows and R where to find these files	23
5.5 Step 4. Verify the configuration	23
5.6 Step 5. Optional cleanup	24
6 How to interact with the Virtual Machine (VM) that hosts PIP	25
6.1 Access and Session Management	25
6.1.1 Login to the VM	25

6.1.2	Copy, Paste, and Keyboard Shortcuts	26
6.2	Understanding the Environment	26
6.2.1	Users and Permissions	26
6.2.2	Architecture Overview	27
6.3	Essential Docker Commands (Cheat Sheet)	27
6.4	Managing Data	28
6.4.1	Data on the VM (Persistent)	28
6.4.2	Data Inside Containers (Ephemeral)	29
6.4.3	Safe Deletion and Cleanup	29
6.5	Diagnosing API or Container Issues	30
6.5.1	Check Container State	30
6.5.2	View Logs	30
6.5.3	Reproduce Interactively	31
6.5.4	Restart or Rebuild Containers	31
6.6	Testing API Endpoints	32
6.6.1	From Inside the Container	32
6.6.2	Loop through common endpoints	32
6.6.3	From Outside (VM host or browser)	32
6.7	Troubleshooting Common Errors	32
6.8	Understanding Ports (80 vs 8080)	33
6.9	Monitoring Resource Usage	33
6.10	Appendix	34
6.10.1	Check Disk Usage Quickly	34
6.10.2	Check Docker Service Logs	34
6.10.3	Check R version inside container	34
7	SetUp Vscode to work with Stata	35
7.1	Install VScode and Stata Extensions	35
7.2	Configure <code>rundo</code> and <code>rundolines</code>	35
7.2.1	Recommended <code>.ini</code> tuning (for snappier runs)	36
7.3	Set up User Tasks (global, once) to call <code>rundolines</code> / <code>rundo</code>	37
7.4	Bind your keyboard shortcuts (global, once)	38
7.5	(Optional) Running Mata files directly	38
7.5.1	A) Wrapper <code>.do</code> (recommended for standalone <code>.mata</code> files)	39
7.5.2	B) Mata blocks inside a <code>.do</code>	39
7.6	Quick sanity test (30 seconds)	39
7.7	Troubleshooting & Performance	40
8	SetUp Git & GitHub in VS Code	41
8.1	Prerequisites	41
8.2	Install VScode and Stata Extensions	41
8.3	First-time Git configuration (one-time)	42
8.4	Authenticate to GitHub	42

8.5	Clone or publish a repository in VS Code	42
8.5.1	Clone an existing repository	42
8.5.2	Publish (create a new GitHub repo from a local folder)	43
8.6	Power-ups	43
8.6.1	GitLens highlights	43
8.6.2	Git Graph highlights	43
8.6.3	Copilot & Copilot Chat	43
8.7	Team conventions (recommended)	44
8.8	Quick reference (commands)	44
8.9	Suggested VS Code settings (optional)	44
9	Summary	46
References		47

Preface

This is a Quarto book. In this book we intend to write technical information about PIP projects.

1 Introduction

The purpose of this book is to gather all the technical knowledge specific to PIP in one place.

2 Add code coverage badge to your GitHub Repository.

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

2.1 Codecov.io

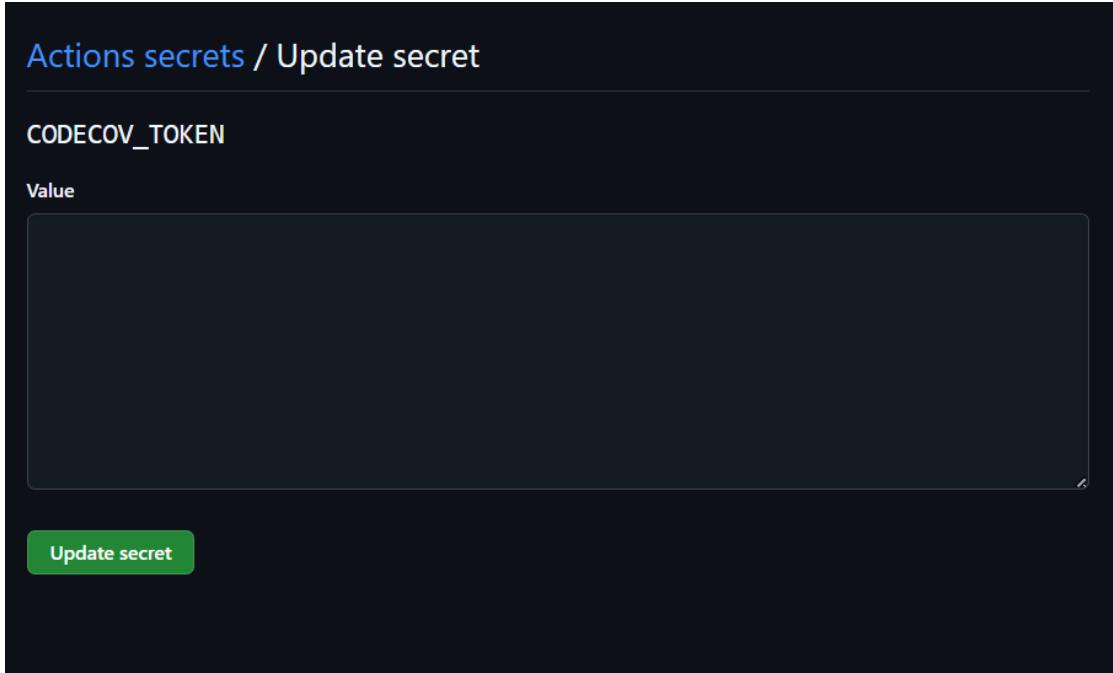
- Create an account at <https://about.codecov.io/> , sign up with your GitHub account if you don't have an account already. **Codecov** is a popular tool for measuring and visualizing **code coverage** in software projects. It integrates with GitHub, GitLab, Bitbucket, and other CI/CD systems to provide 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

The screenshot shows the Codecov Coverage Analytics setup page for the repository 'PIP-Technical-Team / PIP_R_Training'. The top navigation bar includes 'Coverage' (selected), 'Bundles', 'Tests' (New), and 'Configuration'. The main section is titled 'Coverage Analytics' with a note: 'Codecov analyzes your coverage reports to help you identify untested code and improve test effectiveness. Before integrating with Codecov, ensure your project generates coverage reports, as Codecov relies on these reports for coverage analysis.' Below this, a section titled 'Select a setup option' contains three radio buttons: 'Using GitHub Actions' (selected), 'Using Circle CI', and 'Using Codecov's CLI'. A step-by-step guide follows: 'Step 1: Output a Coverage report file in your CI'. It asks to select a language (Jest) and provides a terminal command: 'npm install --save-dev jest'. A copy icon is available for the command.

- 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.

2.2 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

```
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 -

```

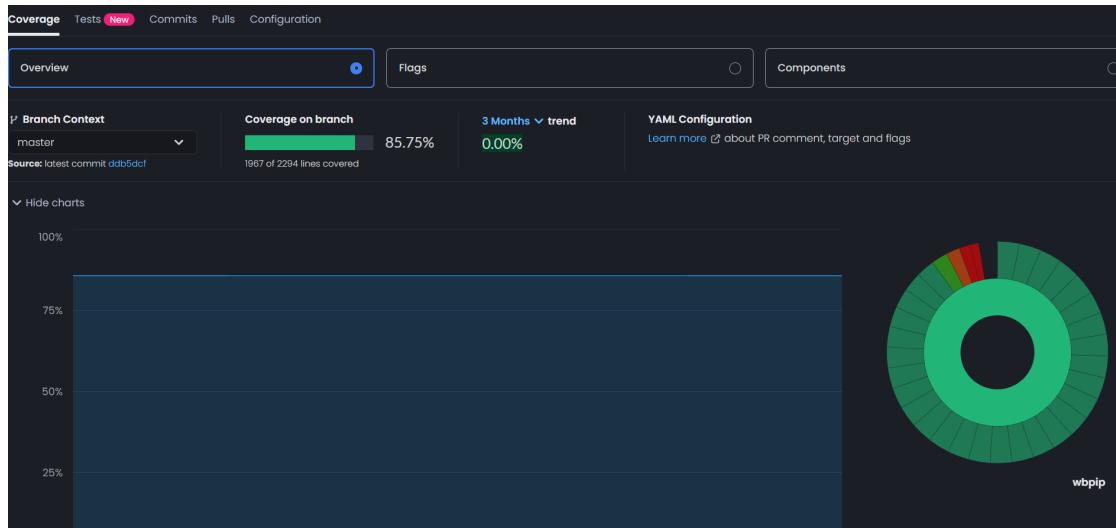
on:
  push:
    branches: [master, your-branch]

```

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.

3 Setting up Github Actions for Auto Deployment of Quarto book

3.1 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.

3.2 Workflow

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

```
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 }}
```

3.2.1 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.

3.2.2 Naming the workflow

```
name: Quarto Publish
```

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

3.2.3 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).

3.2.4 The Steps

3.2.4.1 1. Check out the repository

```
- 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.

3.2.4.2 2. Set up Quarto

```
- 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.

3.2.4.3 3. Render and Publish

```
- 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.

3.3 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.

3.4 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 note `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")`.

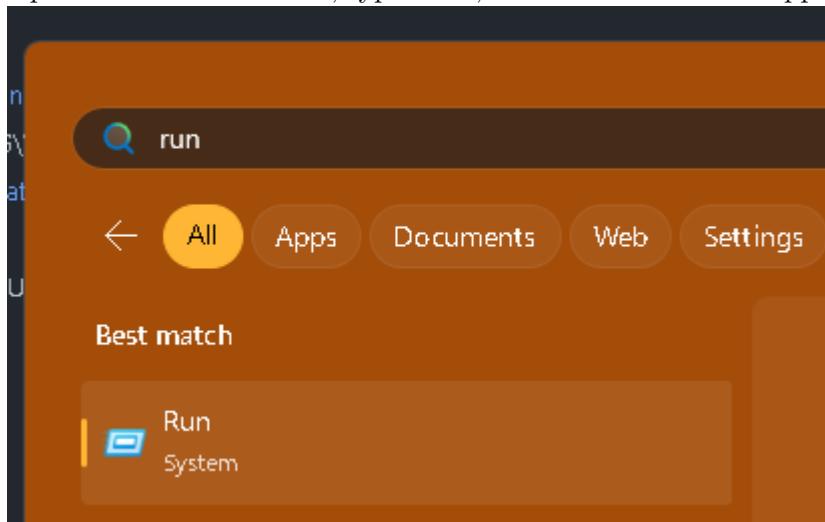
For reference the Quarto book is published [here](#).

4 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.

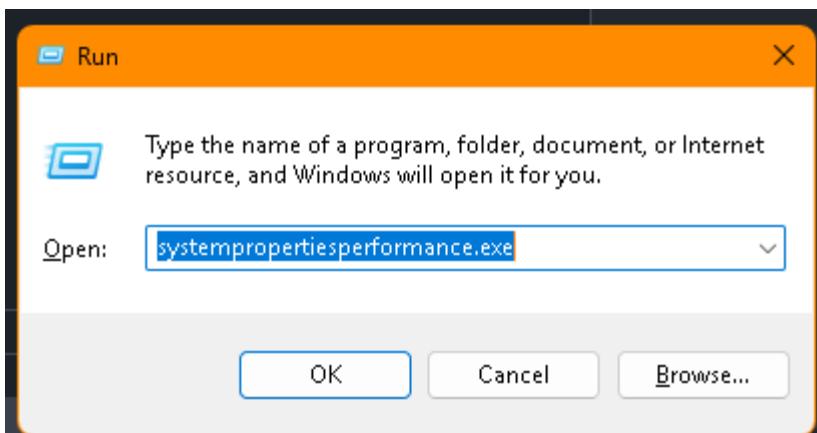
4.1 Modify System Properties for Performance

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



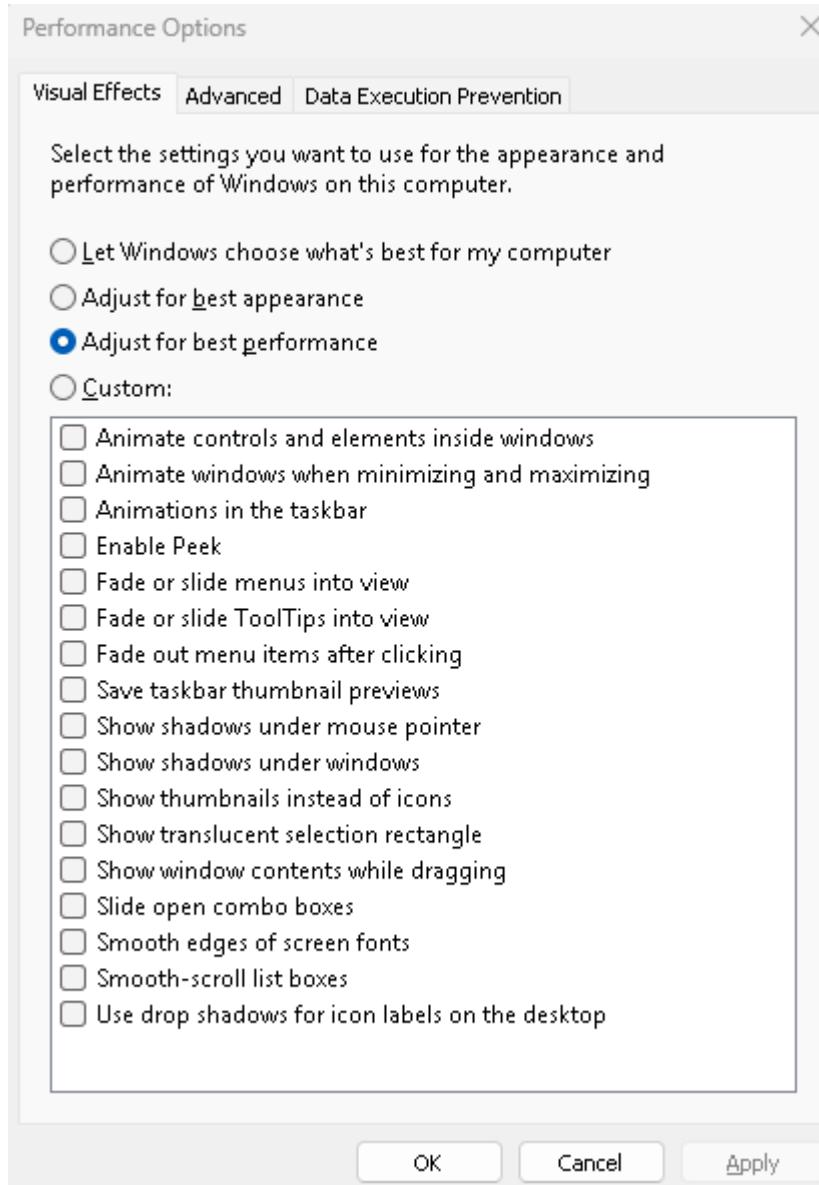
2. In the Run app, type the following command and hit **Enter**:

```
systempropertiesperformance.exe
```



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

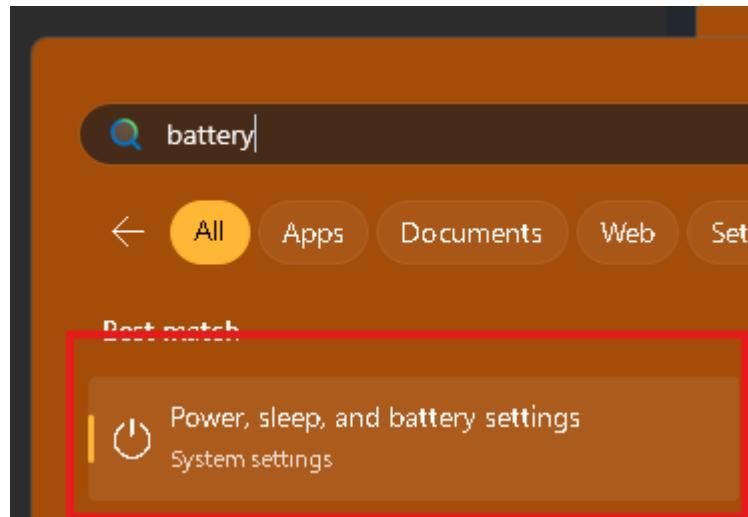
- Click on **Adjust for best performance**.
- Make sure all the checkboxes are unchecked.
- Click **Apply** and then **OK**.



⚠ Warning

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

4.2 Battery Settings



1. Open **Battery Settings**.

System > Power & battery

100%

Smart charging is on. [Learn more](#)

Some of these settings are managed by your organization.

Energy recommendations
Lower your carbon footprint by applying these recommendations

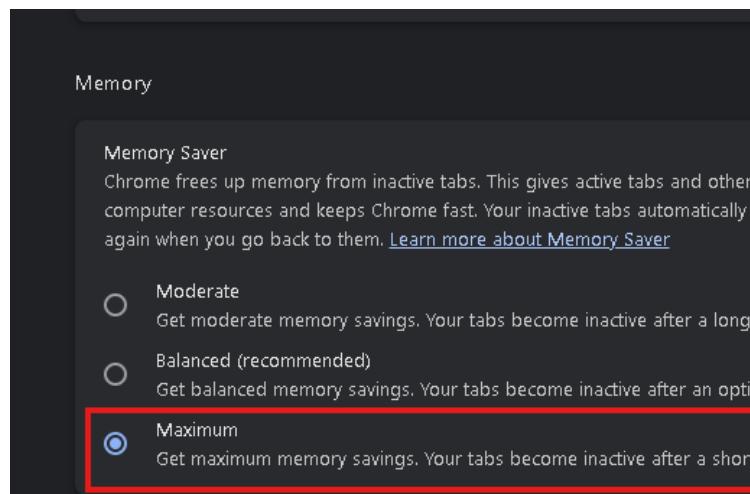
Power Mode
Optimize your device based on power use and performance

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

4.3 Google Chrome Settings

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

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



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

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

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

5.1 Intro

When working with R, **your environment files** (`.Renvironment` 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.

5.2 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
```

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

5.3.1 .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.

5.3.2 .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 = FALSE)))
```

5.4 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\.Renvironment"
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 `.Renvironment` and `.Rprofile` in that directory, no matter which IDE you open.

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

5.5 Step 4. Verify the configuration

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

```
path.expand("~")
normalizePath("~/Renvironment", 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`
 - `.Renvironment` and `.Rprofile` point to the same files in that folder
 - Your environment variables (`R_LIBS_USER`, etc.) are correctly loaded
-

5.6 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.

6 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.

6.1 Access and Session Management

6.1.1 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-wzlxqip01.worldbank.org`
 - **Production:** `Linux-wzlxppip01.worldbank.org`
4. Once connected, switch to the `srvpip` user:

```
sudo su - srvpip
```

Shortcut	Action	Notes
----------	--------	-------

6.1.2 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 <code>srvpip</code>
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.

6.2 Understanding the Environment

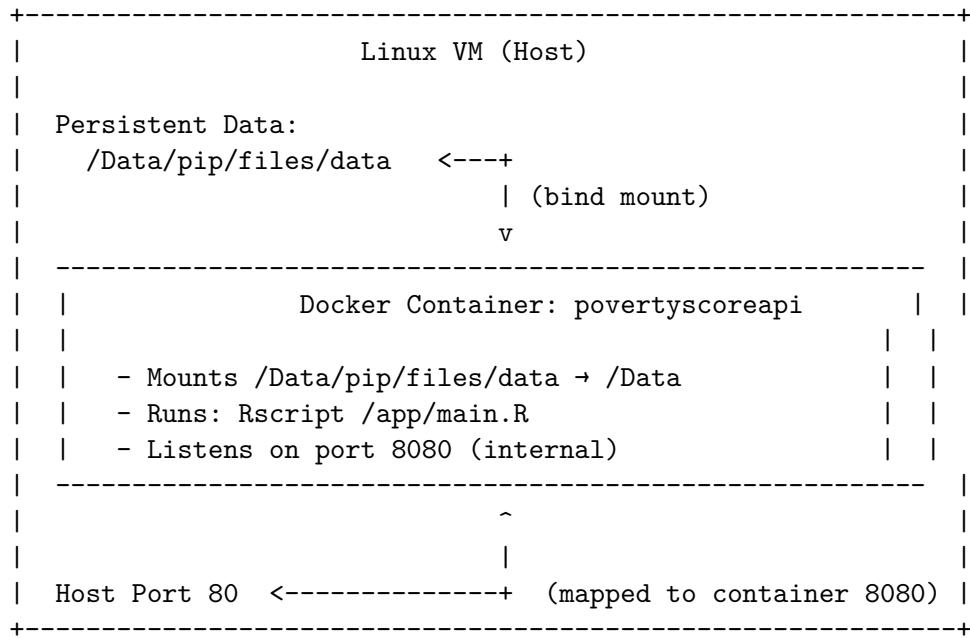
6.2.1 Users and Permissions

- **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

6.2.2 Architecture Overview



6.3 Essential Docker Commands (Cheat Sheet)

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

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

6.4 Managing Data

6.4.1 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 *
```

6.4.1.1 Find large or old files

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

6.4.2 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

6.4.2.1 Access via container shell:

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

6.4.3 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.

6.5 Diagnosing API or Container Issues

6.5.1 Check Container State

```
sudo docker ps -a
```

Get detailed exit info:

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

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

6.5.2 View Logs

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

6.5.3 Reproduce Interactively

Start an interactive debug session:

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

Inside:

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

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

6.5.4 Restart or Rebuild Containers

```
sudo docker restart povertycoreapi
```

Or rebuild completely:

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

6.6 Testing API Endpoints

6.6.1 From Inside the Container

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

6.6.2 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
```

6.6.3 From Outside (VM host or browser)

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

6.7 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 permission denied on /Data	Not logged in to Azure Container Registry Wrong user ownership	sudo az acr login --name itsesippsscoreregistryprod sudo chown -R svrpip /Data
No logs shown	Container failed before startup	Run sudo docker inspect povertyscoreapi

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

6.8 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.

6.9 Monitoring Resource Usage

Watch real-time CPU and memory:

```
sudo docker stats povertyscoreapi
```

Or globally:

```
top -u svppip
```

6.10 Appendix

6.10.1 Check Disk Usage Quickly

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

6.10.2 Check Docker Service Logs

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

6.10.3 Check R version inside container

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

7 SetUp Vscode to work with Stata

Steps to configure VScode for Stata development.

7.1 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).
-

7.2 Configure rundo and rundolines

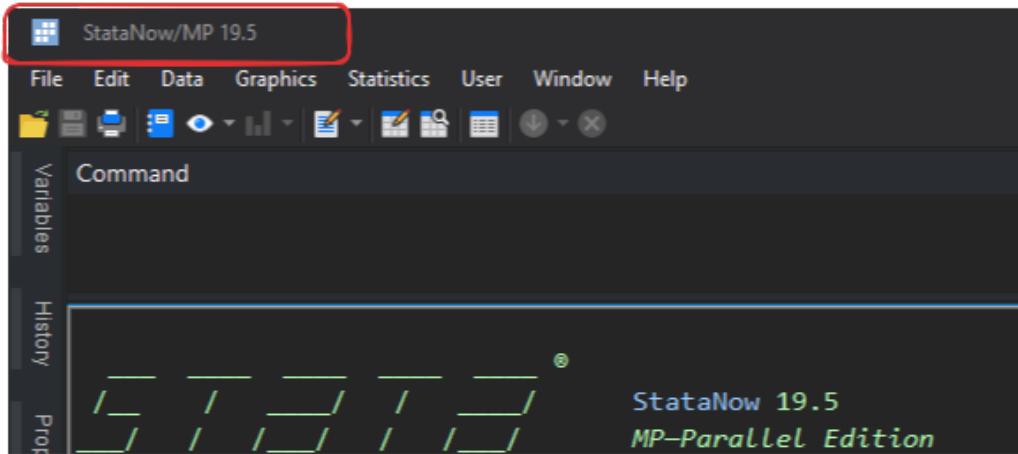
All the information in this section is based on Friedrich Huebler’s [article](#). If you need more detail, see his blog.

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.

7.2.1 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).

7.3 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).

7.4 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.

7.5 (Optional) Running Mata files directly

Two clean options; pick one.

7.5.1 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", "\"${workspaceFolder}\\._run_mata.do\"", "\"${file}\""]
}
```

And a keybinding for `.mata` files:

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

7.5.2 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).

7.6 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.
-

7.7 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.

8 SetUp Git & GitHub in VS Code

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

8.1 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.
-

8.2 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).
-

8.3 First-time Git configuration (one-time)

Open **Terminal** in VS Code (Ctrl+`)

```
# 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
```

8.4 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.

8.5 Clone or publish a repository in VS Code

8.5.1 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.

8.5.2 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.

8.6 Power-ups

8.6.1 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

8.6.2 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

8.6.3 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”

8.7 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`.

8.8 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
```

8.9 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  
}
```

9 Summary

References