STACY: RELIABLE STATA EXECUTION WITH EXIT CODES AND LOCKFILES

Jan Fasnacht
University of Chicago
jfasnacht@uchicago.edu

March 2026

ABSTRACT

Stata's defaults leave two things implicit that reliable workflows require to be explicit: execution reports success even on failure; dependencies are global and unversioned. These defaults break composition—the ability to integrate Stata with build systems, replication workflows, and mixed-language pipelines. `stacy` implements proper exit codes, and a manifest and lockfile for dependencies. With these, Stata projects can be automated, versioned, and reproduced.

Keywords: stacy · reproducibility · workflow · dependencies · exit codes · Stata

1 Introduction

Stata projects need to compose: with build systems that expect exit codes, with environments that must be reconstructed, with pipelines that mix languages. But Stata leaves two things implicit that composition requires to be explicit: the outcome—whether execution succeeded—and the environment—what packages the project needs.

The outcome is implicit. Stata's batch mode returns exit code 0 even when scripts fail. Build systems, schedulers, and downstream scripts cannot detect failure—they proceed as if nothing went wrong.

The environment is implicit. User-written packages install to a global path—no manifest, no lockfile, no isolation between projects. Partial solutions exist—`require` can check versions at runtime, `dependencies` can freeze and restore ado files—but no integrated workflow declares, installs, and locks packages from a single project file. Each `ssc install` retrieves whatever version exists at that moment; a collaborator installing later may get a different version.

`stacy` makes both explicit. For outcomes: `stacy run` parses Stata's logs and returns proper exit codes. For environments: a manifest (`stacy.toml`), a lockfile with checksums (`stacy.lock`), and cached installation with runtime isolation. A project that signals failure can be automated reliably. Build systems, schedulers, and CI pipelines detect Stata failures like they detect failures in any other tool. Cluster jobs halt immediately on failure instead of silently producing garbage. Mixed-language pipelines stop when Stata fails, so downstream steps never run on stale data. Collaborators can work from the same locked environment rather than debugging "it worked on my machine."

Package managers in other languages—npm, Poetry, Cargo—have long provided lockfiles and proper exit codes. Existing Stata tools address parts of this problem—`statacons` (Guiteras et al., 2023) provides build automation with proper exit codes, `require` (Cor-

reia and Seay, 2024) validates versions, `repkit` (Bjärkefur et al., 2025) checks execution determinism—but none combines lockfiles, package isolation, and exit codes in a single tool. Section 6 compares them in detail. `stacy` integrates these pieces into a unified workflow. Installation is described in Section 7.2.

2 Two Gaps in Stata

2.1 Execution observability

Consider a pipeline where Stata prepares data that Python then uses for machine learning. The pipeline runs `stata-mp -b do prep.do`, checks the exit code, and if zero, proceeds to the Python step.

But Stata *always* returns zero:

```
$ stata-mp -b do prep.do
$ echo $?
0
```

The prep script may have failed. Stata still exits with code 0. The pipeline proceeds. Python runs on stale or missing data. The failure propagates silently.

This problem is not limited to mixed-language pipelines. Consider a researcher running a multi-step analysis on a secure data server. The workflow submits five do-files sequentially via a shell script. The second script contains an error, but Stata returns exit code 0 for each step. All five scripts run to completion. The researcher returns the next day to find that steps three through five produced results based on an incomplete intermediate dataset. Three days of compute time—and the associated data access window—are wasted. With proper exit codes, the shell script would have stopped at step two.

This violates a fundamental convention that Unix-based tools rely on: programs signal success with exit code 0 and failure with nonzero codes (StataCorp, 2025). Build tools, continuous integration (CI) systems, and automation tools all depend on this convention. Stata does not follow it (Guiteras et al., 2023). The common workaround is a shell script that parses Stata's log for error patterns—the Gentzkow–Shapiro lab template (Gentzkow and Shapiro, 2014, 2024) is a widely used example—but such scripts are fragile, handling neither captured errors nor nested do-files correctly.

2.2 Environment determinism

Consider a replication package submitted to a journal. The code ran perfectly on the author's machine with `reghdfe` version 5.7.3. Six months later, the reviewer runs the same code. SSC has updated `reghdfe` to version 6.0, which changes the default options. The results differ. The replication fails. Reproducibility audits find that a significant share of economics research cannot be reproduced from provided code (Herbert et al., 2024; Vilhuber, 2020); while the causes are varied, uncontrolled package versions are a contributing factor.

This is not user error. Stata provides no standard mechanism to declare, install, and lock a project's package dependencies. Packages install globally, so different projects cannot use different versions.

Modern package managers solve this problem: Python has `poetry.lock`, R has `renv.lock`, JavaScript has `package-lock.json`. Each records exact versions and enables reproducible installation. Manual workarounds exist for Stata—Newson (2022) describes porting packages between machines via zip files—but these are ad hoc solutions that do not scale.[1]

[1] An important ecosystem constraint shapes any solution: SSC does not maintain version history. Once a package is updated, the previous version is no longer available. Exact version pinning—the kind npm or Poetry provide—is not possible for SSC packages. Section 4.3 discusses how `stacy` works within this limitation.

2.3 Design overview

`stacy` provides two commands that address these gaps directly. `stacy run` (Section 3) wraps Stata's batch mode with log-based error detection, proper exit codes, parallel execution, timeouts, and structured output. `stacy add` and `stacy install` (Section 4) manage dependencies through a manifest, a lockfile with checksums, and a global package cache. When both are used together, the runner constructs an isolated package search path from the lockfile before launching Stata. Figure 1 illustrates the overall workflow.

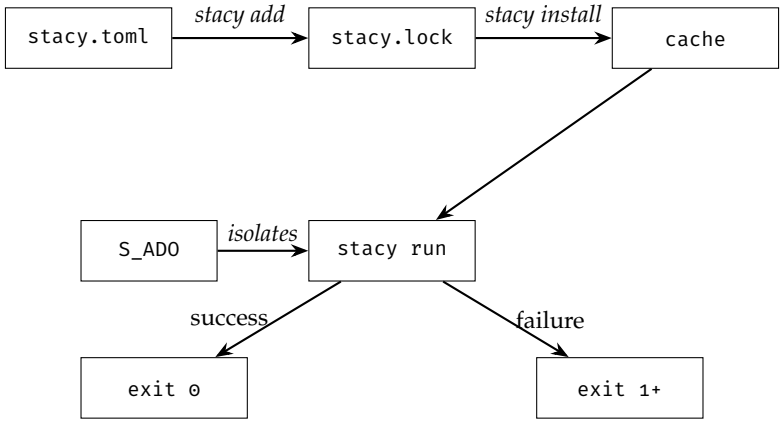

Figure 1: stacy workflow: environment management (top) records dependencies in a lockfile and downloads packages to a global cache; execution (bottom) constructs an isolated search path from the cache and returns proper exit codes.

Two principles guide the design. First, easy to adopt. Every command has a Stata wrapper, so users can work entirely from the Stata console with `help stacy` working as expected. No external tools, configuration files, or workflow changes are required to get started.

Second, reliable. Exit codes are derived from log parsing backed by more than 250 test cases covering nested do-files, captured errors, and abnormal termination. Dependencies are locked with checksums for integrity verification.

Every `stacy` command is available both from the Stata console (examples prefixed with `.`) and from the terminal (prefixed with `$`). In Stata, options use standard syntax (`stacy install, frozen`); on the command line, they use flags (`stacy install --frozen`).

3 Execution

3.1 Core command: stacy run

```
stacy run script [scripts...] [, code(string) directory(string)
    timeout(integer) quiet verbose]
```

Executes Stata scripts with proper exit codes. Multiple scripts run sequentially by default, stopping on first failure. The `code` option executes inline Stata code; `directory` sets the working directory; `timeout` kills the Stata process after a specified duration, useful for convergence loops or long-running jobs on shared clusters. From the command line, `stacy run --parallel` runs scripts concurrently (this requires spawning multiple Stata processes, so it is available only from the command line).

When a lockfile is present, `stacy run` also isolates the package search path (Section 3.2).

The Stata wrapper stores results in `r()` for programmatic use:

```
. stacy run analysis.do
```

```
Running analysis.do...
Completed in 12.3s

. return list

scalars:
    r(success) = 1
    r(exit_code) = 0
    r(duration_secs) = 12.3
    r(error_count) = 0

macros:
    r(log_file) : "/logs/analysis.log"
    r(script) : "analysis.do"
    r(source) : "file"
```

Table 4 documents all stored results. On the command line, exit codes signal success or failure directly:

```
$ stacy run analysis.do
Running analysis.do...
Completed in 12.3s

$ echo $?
0
```

When errors occur, `stacy` detects them and returns nonzero. The error message includes a human-readable description and a link to the relevant manual page—on a remote server, this means the user can diagnose the failure from the error output alone, without downloading and searching a log file:

```
$ stacy run broken.do
FAIL  broken.do  (0.8s)

   Error: r(199) - unrecognized command

   See: https://www.stata.com/manuals/perror.pdf#r199

$ echo $?
2
```

For remote or long-running jobs, `stacy run --verbose` streams Stata's log output in real time, so users can monitor progress over SSH without waiting for completion. On the command line, `stacy run --json` returns structured JSON output that monitoring scripts and automation tools can parse directly, rather than requiring downstream log processing.

Table 1 summarizes the options for `stacy run`.

3.2 Runtime isolation

When a lockfile is present, `stacy run` constructs an isolated package search path before launching Stata. It is the runner—not the lockfile—that builds this isolation: `stacy run` reads the lockfile, resolves cached package paths, and sets the `S_ADO` environment variable. Stata reads `S_ADO` at startup to determine where it looks for user-written commands (StataCorp, 2025). Two modes are available:

- **Strict mode** (default): `S_ADO` is set to the cached package paths followed by `BASE`:

Table 1: stacy run options

Option	Description
allow-global	Include global package paths in search path
cd	Change to script's parent directory (command line only)
code(*string*)	Execute inline Stata code instead of a script file
directory(*string*)	Set working directory for execution
json	Output structured JSON results (command line only)
parallel	Run scripts concurrently (command line only)
profile	Include execution performance metrics
quiet	Suppress output
timeout(*integer*)	Kill Stata after specified seconds
trace(*integer*)	Enable execution tracing at given depth
verbose	Stream Stata's log output in real time

```
S_ADO = "<cache>/stacy/packages/estout/3.31;
         <cache>/stacy/packages/reghdfe/6.12.3;BASE"
```

Only locked packages and Stata's built-in commands are visible. Undeclared packages—including personal utility ado files in `PERSONAL`—are hidden, ensuring that only declared dependencies are available and the script runs identically elsewhere.

- **Allow-global mode** (`allow-global`): the standard search path is appended after `BASE`:

```
S_ADO = "<cache>/stacy/packages/estout/3.31;
         <cache>/stacy/packages/reghdfe/6.12.3;
         BASE;SITE;PERSONAL;PLUS;OLDPLACE"
```

Locked packages take precedence, but globally installed packages are also available. This is useful during development or migration, when not all dependencies have been added to the lockfile yet.

where `<cache>` is the platform's cache directory (`~/.cache` on Linux, `~/Library/Caches` on macOS, `%LOCALAPPDATA%` on Windows); path separators adapt accordingly. Package cache paths are sorted alphabetically by package name for deterministic output. Mata libraries (`.mlib` files) are discovered through the same search path, so packages that include Mata code work without additional configuration. Compiled plugins (`.plugin`) and Java or Python integrations are not affected by `S_ADO`.

`stacy run` also passes the `-q` (quiet) flag to Stata, which skips execution of the user's `profile.do`. This prevents `adopath` modifications in `profile.do` from overriding `stacy`'s isolation, and ensures that execution does not depend on machine-specific startup configuration. The tradeoff is that startup preferences set in `profile.do` (such as `set` options or global macros) are also not applied; scripts that depend on these must set them explicitly. Users who need personal utility ado files during development should use `allow-global`.

Users do not need to modify their do-files or manually configure `adopath`.

3.3 Error detection

`stacy` parses the Stata log to detect errors. Error detection works by parsing the log from the end. The parser locates the *last* `end of do-file` marker—which corresponds to the outermost do-file in nested execution—and then scans the lines after that marker for a return code pattern (`r(`*code*`);`). If found, the script failed with that code; if not, the script suc-

ceeded. This design naturally handles `capture`: captured errors do not propagate past `end of do-file`, so they never appear after the final marker.

Several edge cases merit discussion. When a do-file calls another do-file, each produces its own `end of do-file` marker. The parser uses the *last* marker (found via reverse search), which corresponds to the outermost script. If an inner do-file errors and the error propagates, the return code appears after this outermost marker; if the error is captured, it does not. `capture noisily` is handled correctly: although the error is displayed, no `r(N);` line appears after the final `end of do-file` marker when the error is captured, so the parser correctly reports success. User-written output that resembles error patterns—such as `display "r(199);"`—likewise appears before the marker as part of Stata's regular output and is ignored. The parser only matches the strict pattern `r(N);` appearing after the final marker. Stata's explicit `exit` and `error` commands produce the same `r(N);` pattern in the log and are detected identically.

If Stata terminates abnormally—for example, killed by the operating system or a timeout—the log lacks a footer. The parser detects the missing `end of do-file` marker and returns an error (exit code 5), so abnormal termination is never mistaken for success. When `set trace on` is active, trace output can contain patterns resembling return codes, but these appear within the body of the log before the final marker and are not matched.

The test suite includes more than 250 test cases covering error detection (including nested do-files, false positives from display output, and incomplete logs), signal handling, S_ADO construction, timeout enforcement, and CLI integration. Table 2 summarizes the parser's behavior on key scenarios.

Table 2: Log parser behavior on edge cases

Scenario	Parser behavior	Exit code
Uncaptured error (`r(601)`)	Detected after final marker	3
Captured error (`capture reg ...`)	No r-code after marker—success	0
Nested do-file error (`r(199)`)	Propagates to outermost marker	2
Abnormal termination (no footer)	Missing marker detected	5
`display "r(199);"`	Output before marker—ignored	0
`set trace on` output	Trace before marker—ignored	0

To provide descriptive error messages, `stacy` queries the user's Stata installation for return code descriptions rather than shipping a static list. This means error messages stay accurate across Stata versions without requiring `stacy` updates. Where a direct description is unavailable, `stacy` falls back to range-based categories (e.g., codes 100–199 are syntax errors, 600–699 are file I/O errors).

Stata's internal return code (the number in `r(N)`) is preserved in the stored results (`r(exit_code)` when called from Stata, JSON output when called from the shell). The *shell* exit code is a normalized category derived from the r-code range (Table 3). This mapping is many-to-one by design: it compresses Stata's hundreds of return codes into a small, stable set that build tools can branch on. Exit code 1 is the residual category for r-codes not covered by the specific ranges. This mapping is documented and treated as a stable interface, so build systems and CI tools can depend on it.

3.4 Stored results

The Stata wrappers store results in `r()` for programmatic use:

4 Environment Management

Table 3: Exit code categories

Code	Meaning	Stata r-code range
0	Success	—
1	General Stata error	all other r-codes
2	Syntax error	r(100)–r(199)
3	File/data error	r(600)–r(699)
4	Memory error	r(900)–r(999)
5	Internal stacy error	—
6	Statistical error	r(400)–r(499)
10	System/environment error	r(800)–r(899)

Table 4: Stored results from stacy run

Result	Description
`r(success)`	Whether script succeeded (1/0)
`r(exit_code)`	Exit code (0 = success)
`r(duration_secs)`	Execution time in seconds
`r(error_count)`	Number of errors detected
`r(log_file)`	Path to log file
`r(script)`	Path to script
`r(source)`	Execution source: `file` or `inline`

4.1 Core commands

`stacy` provides project-level dependency management through two artifacts: a manifest declaring what packages the project needs, and a lockfile recording exactly what versions are installed. Packages themselves are stored in a global cache (`~/.cache/stacy/packages/`); at runtime, `stacy run` constructs an `S_ADO` path from the lockfile to point Stata at the cached packages (Section 3.2). Three commands manage the environment: `stacy init` creates a project, `stacy add` adds packages, and `stacy install` downloads packages to the cache from a lockfile.

```
stacy init [path] [, force]
```

Creates a new project with an `stacy.toml` manifest. The `force` option overwrites existing files.

```
. stacy init my-analysis
Created my-analysis/stacy.toml
```

```
stacy add packages [, source(string) dev test]
```

Adds packages to the manifest and lockfile. Packages install from SSC by default; use `source(github:user/repo)` for GitHub, `source(net:`*`url`*`)` for network sources, or `source(local:`*`path`*`)` for local directories. The `dev` and `test` options mark packages as development or test dependencies, respectively.

```
. stacy add estout reghdfe
Adding estout from SSC...
```

```
Adding reghdfe from SSC...
Updated stacy.toml
Updated stacy.lock
Installed 2 packages
```

For GitHub sources, the option syntax differs between interfaces:

```
. stacy add ftools, source(github:sergiocorreia/ftools)
Adding ftools from GitHub...
Updated stacy.toml
Updated stacy.lock
Installed 1 package
```

From the command line, the equivalent is:

```
$ stacy add ftools --source github:sergiocorreia/ftools
```

```
stacy install [package] [, from(string)]
```

Without arguments, installs all packages from the lockfile. With a package name, installs that specific package. The `from` option specifies the source. The `with` option includes optional dependency groups (`dev`, `test`), and `frozen` requires an exact lockfile match.

```
. stacy install
Installing estout 3.31 [cached]
Installing reghdfe 6.12.3 [cached]
Installing ftools 2.49.1 [downloading]
Installed 3 packages
```

4.2 The manifest (stacy.toml)

The manifest declares project metadata, dependencies, and scripts:

```
[project]
name = "my-analysis"

[packages.dependencies]
estout = "ssc"
reghdfe = "ssc"
ftools = "github:sergiocorreia/ftools"

[scripts]
clean = "src/01_clean.do"
analyze = ["src/02_analyze.do", "src/03_tables.do"]
all = ["clean", "analyze"]
```

The `[project]` section contains metadata. The `[packages.dependencies]` section lists dependencies: the value `"ssc"` installs from SSC, while a `"github:user/repo"` value installs from GitHub. Additional source types include `"net:`*`url`*`"` for network-hosted packages and `"local:`*`path`*`"` for packages in local directories. The `[scripts]` section defines named tasks; `stacy task` executes them. In the example above, `stacy task all` runs `clean` and then `analyze` in sequence.

The result is a self-contained project structure:

```
my-project/
  stacy.toml       <- manifest (what you want)
  stacy.lock       <- lockfile (what you have, exactly)
  src/
    analysis.do
```

Packages are not stored in the project directory. They live in the global cache and are made available to Stata at runtime via `S_ADO`.

4.3 The lockfile (stacy.lock)

The lockfile records the exact state of installed packages. It uses TOML format for human readability and version control friendliness:

```
# Auto-generated by stacy - do not edit manually
# Use `stacy install` to manage packages

version = "1"
stacy_version = "1.1.0"

[packages.estout]
version = "3.31"
checksum = "sha256:a1b2c3d4e5f6..."
group = "production"

[packages.estout.source]
type = "SSC"
name = "estout"

[packages.ftools]
version = "2.49.1"
checksum = "sha256:1a2b3c4d5e6f..."
group = "production"

[packages.ftools.source]
type = "GitHub"
repo = "sergiocorreia/ftools"
tag = "v2.49.1"

[packages.reghdfe]
version = "6.12.3"
checksum = "sha256:f6e5d4c3b2a1..."
group = "production"

[packages.reghdfe.source]
type = "SSC"
name = "reghdfe"
```

Each package entry records the exact version, source, SHA-256 checksum for integrity verification, and dependency group. Packages are sorted alphabetically for deterministic output. When `stacy install` runs, it compares checksums to verify integrity. If a mismatch is detected—for example because SSC has updated a package—installation fails with a clear error.

Because SSC does not maintain version history, `stacy` uses checksums as a substitute for version pinning. When a package is first added, `stacy` computes a SHA-256 hash of each file and combines them into a single order-independent checksum stored in the lockfile. If SSC later updates a package, the checksum will not match and `stacy install` will fail with a clear error, alerting the user to the change. The original version is no longer available from SSC at this point—the lockfile detects drift but cannot restore the prior version

on its own. Two practical mitigations exist: first, `stacy` caches downloaded packages in `~/.cache/stacy/packages/`, so reinstallation on the same machine (or any machine sharing the cache) succeeds even after SSC updates the package. The `ssc2` command (Vilhuber, 2023) offers an additional mitigation: it installs packages from a daily-mirrored SSC archive with dated snapshots, allowing installation of packages as they existed on a specific date. To accept the new version, users run `stacy lock` to regenerate the lockfile.

For GitHub sources, the situation is stronger: `stacy` pins to a specific tag (e.g., `v2.49.1`), and tagged releases are immutable. This provides true version pinning comparable to what npm or Poetry offer.

4.4 Additional commands

Table 5 lists additional commands that span environment management and execution.

Table 5: Additional stacy commands

Command	Description
stacy task	Run named tasks defined in `[scripts]`
stacy lock	Regenerate lockfile; `--check` verifies sync
stacy remove	Remove packages from project
stacy explain	Look up Stata error codes
stacy doctor	System diagnostics

5 Workflow Examples

These examples use terminal syntax because they integrate with shell-based tools; for interactive use, see the Stata console examples in Sections 3 and 4.

5.1 Batch computing

Researchers who run Stata in batch mode on remote servers, clusters, or secure data environments benefit most directly from proper exit codes. A simple shell script chains `stacy run` calls:

```
#!/bin/bash
set -e  # stop on first error
stacy install --frozen
stacy run src/01_clean.do
stacy run src/02_merge.do
stacy run src/03_analyze.do
stacy run src/04_tables.do
```

With `set -e`, the shell stops at the first nonzero exit code. Without `stacy`, every step would appear to succeed regardless of errors, and the researcher would discover the problem only by inspecting output files or logs—potentially days later.

5.2 Build system integration

A Makefile (Stallman et al., 2020) uses `stacy run` as the Stata executor:

```
STATA := stacy run

results/tables.tex: src/03_tables.do data/analysis.dta
    $(STATA) $<

data/analysis.dta: src/02_analyze.do data/clean.dta
    $(STATA) $<

data/clean.dta: src/01_clean.do data/raw.dta
    $(STATA) $<
```

Make detects failures because `stacy run` returns nonzero exit codes. Only outdated targets rebuild.

Snakemake (Mölder et al., 2021) works similarly:

```
rule prep_data:
    input: "src/prep.do", "data/raw.dta"
    output: "data/clean.dta"
    shell: "stacy run {input[0]}"

rule train_model:
    input: "src/train.py", "data/clean.dta"
    output: "models/model.pkl"
    shell: "python {input[0]}"
```

If Stata fails, the pipeline stops. Python never runs on stale data.

5.3 Parallel execution

For independent scripts, `stacy run –parallel` executes them concurrently:

```
$ stacy run --parallel src/table1.do src/table2.do src/figure1.do
Running 3 scripts in parallel...

  [1/3] src/table1.do   [ok]  8.2s
  [2/3] src/figure1.do  [ok] 12.4s
  [3/3] src/table2.do   [ok] 15.1s

Completed 3 scripts in 15.1s (vs 35.7s sequential)
```

Each script runs in its own Stata instance. If any script fails, the others continue, and `stacy` reports all failures at the end with a nonzero exit code.

5.4 Continuous integration

A GitHub Actions workflow runs the analysis on every push:

```
jobs:
  replicate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: stacy install --frozen
      - run: stacy task all
```

The `-frozen` flag ensures the lockfile matches the manifest exactly, failing the build if they diverge. For an additional check, `stacy lock -check` verifies lockfile integrity without installing anything.

The workflow fails if any script fails. Errors surface immediately, not months later during review.

5.5 Replication package

A reviewer receives a replication package. They clone the repository, install dependencies, and run the analysis:

```
$ git clone https://github.com/author/replication.git
$ cd replication
$ stacy install
Installing estout 3.31...
Installing reghdfe 6.12.3...
Installing ftools 2.49.1...
Installed 3 packages

$ stacy task all
Running clean...
  src/01_clean.do [ok] 4.2s
Running analyze...
  src/02_analyze.do [ok] 18.7s
  src/03_tables.do [ok] 3.1s

Completed 3 scripts in 26.0s
```

The lockfile verifies that installed packages match the recorded checksums. The exit codes provide visibility into success or failure.

6 Related Tools

Best practices for reproducible research emphasize automation, dependency management, and self-contained projects (Gentzkow and Shapiro, 2014; Wilson et al., 2017; Christensen et al., 2019), and journals increasingly require runnable replication packages (American Economic Association, 2023). Several tools address aspects of these goals for Stata; that multiple tools tackle overlapping subsets reflects both the breadth of the problem and the absence of built-in infrastructure in Stata for package versioning or error signaling. Table 6 summarizes how `stacy` relates to them.

Table 6: Comparison of Stata workflow tools

	`stacy`	`statacons`	`require`	`repkit`	`dependencies`	`project`
Lockfile with checksums	✓					
Isolated adopath	✓			✓		
Proper exit codes	✓	✓				
Version validation	✓		✓			
Package freeze/restore					✓	
DAG-based builds		✓				
Run-to-run determinism				✓		
GitHub package support	✓					
Stata console interface	✓	✓	✓	✓	✓	✓

`statacons` (Guiteras et al., 2023) provides a full build system based on SCons, with dependency tracking and incremental rebuilds. It detects Stata errors via log parsing and propa-

gates them as SCons build failures. `statacons` and `stacy` both address Stata's silent-failure problem, but in different ways: `statacons` embeds error detection inside its build system, while `stacy` provides OS-level exit codes as a standalone primitive that works with any build tool (Make, Snakemake, etc.) or without one. The tools are complementary: `stacy` for package management, `statacons` for build automation.

`require` (Correia and Seay, 2024) validates that installed packages meet version requirements via assertions inside `.do` files, can install missing packages when given the `install` option, and can read constraints from a requirements file. Both `require` and `stacy` perform version checking, and the tools overlap in *detecting* version mismatches. They differ in where and when they act: `require` validates at runtime inside do-files; `stacy` validates at the project boundary before Stata starts, using checksums and a lockfile artifact. The two are complementary layers: `stacy` ensures that `require`'s runtime checks pass.

`repkit` (Bjärkefur et al., 2025) is a toolkit with several commands. Its `reprun` command checks execution determinism by running a do-file twice and comparing Stata's internal state (RNG state, sort order, data checksums) after each line; its `repado` command provides adopath isolation by redirecting `PLUS` to a project-specific directory and stripping other paths. `stacy` and `repkit` both isolate the ado search path, but differ in mechanism: `repado` requires a team member to manually populate the project directory, while `stacy` installs from a lockfile with checksums. `repkit` checks whether execution is deterministic; `stacy` ensures that the environment is deterministic.

`dependencies` (Goldemberg, 2021) freezes and restores packages within Stata. Its `freeze` command zips installed ado files into an archive; `unfreeze` extracts them and prepends the directory to `adopath`. This provides a portable snapshot of packages but without checksums, version metadata, or source tracking. Both tools aim to make package state reproducible across machines; they differ in mechanism: `dependencies` captures the state as a zip archive, `stacy` records it in a lockfile with checksums and reinstalls from sources.

`project` (Picard, 2013) is a pure-Stata project management tool that tracks dependencies between do-files and their input/output files, skips unchanged scripts, and manages working directories—a do-file build system comparable to Make. It does not manage packages or modify the ado search path. `stacy` and `project` are complementary: `project` orchestrates which scripts run; `stacy` manages what packages are available and whether scripts succeed.

Haghish (2020) describes workflows for hosting and distributing Stata packages on GitHub. `stacy` builds on this by supporting GitHub repositories as a package source alongside SSC and network installations, adding lockfile tracking and checksum verification on top of the hosting workflow Haghish describes.

Several of these tools address parts of the problem `stacy` solves, and some overlaps are genuine. What `stacy` integrates into a single workflow is: (1) a lockfile as a first-class, version-controlled artifact—not runtime assertions or interactive commands; (2) combined package management and execution in one tool; (3) isolated execution via `S_ADO` without requiring do-file modifications; (4) stable, documented exit-code categories that any build tool can depend on; and (5) SSC, GitHub, and network sources under one interface.

7 Discussion

7.1 Limitations

`stacy` requires installing an external binary. The Stata wrappers mitigate this—users can work entirely from the Stata console—but it remains an additional tool to set up.

`stacy` manages Stata packages only. It does not manage data files, the Stata version, or external tools like Python or R. Docker provides complete environment reproducibility—OS, Stata version, packages, external tools—and `stacy` does not claim to replace it. However, `stacy` is lightweight and Stata-native: it requires no Docker knowledge, no container infras-

tructure, and no Stata license management within containers. For users who need full-stack reproducibility, `stacy` manages Stata packages inside or outside containers.

Unlike npm or Cargo, `stacy` does not resolve transitive dependencies automatically. This is a principled design decision, not a missing feature: SSC packages do not declare what they depend on, and the archive provides no dependency metadata. Automatic transitive resolution is therefore impossible without a curated dependency graph that does not exist. If package A requires package B, the user must `stacy add` both. If B is omitted, the script will fail at runtime—in strict mode, B will not be found; in allow-global mode, a globally installed version may be used silently, undermining reproducibility. GitHub sources could in principle include dependency metadata (e.g., in a `stata.toc` or project manifest), but no convention for this exists in the Stata ecosystem.

7.2 Installation

`stacy` supports macOS, Linux, and Windows, and is distributed as a single binary with no runtime dependencies. Two installation methods are available:

```
. net install stacy, from("https://stacy.janfasnacht.com/stata")
. stacy setup
```

```
$ curl -fsSL https://stacy.janfasnacht.com/install.sh | sh
```

The first installs the Stata wrappers via `net install`, then `stacy setup` downloads the compiled binary for the user's platform. The second installs everything from the terminal. Both place the binary in `~/.local/bin` (macOS/Linux) or `%LOCALAPPDATA%\stacy` (Windows). In environments where neither method is available, the binary can be downloaded directly from the GitHub releases page.

For Stata Journal distribution, the Stata wrappers (`.ado` and `.sthlp` files) are distributed through SJ archives as usual; the compiled binary is installed separately. This follows the precedent of `statacons` (Guiteras et al., 2023), which depends on an external runtime (Python and SCons) installed outside Stata's package system.

7.3 Documentation and development

Built-in help is available via `stacy --help` and `help stacy` from the Stata console. The project follows semantic versioning: interfaces and contracts are stable from version 1.0. The source code, issue tracker, and contribution guidelines are available in the open source repository.

8 Conclusion

`stacy` provides two primitives that Stata's defaults leave implicit: proper exit codes for observable execution, and a lockfile with checksums for environment determinism. Together, these let Stata projects compose with build systems, continuous integration, and replication workflows.

Programs and supplemental materials

`stacy` is open source under the MIT license, available at `https://github.com/janfasnacht/stacy`. It is written in Rust. The project website at `https://stacy.janfasnacht.com` provides a getting started guide, a complete command reference, and workflow examples. The Stata wrappers (`.ado` and `.sthlp` files) are available as supplemental materials.

References

American Economic Association. Data and code availability policy. `https://www.aeaweb.org/journals/data/data-code-policy`, 2023. Accessed: 2024-01-15.

K. Bjärkefur, B. Daniels, L. E. San Martin, and A. Singh. repkit: Tools for reproducible coding. *Stata Journal*, 25(4):699–718, 2025.

G. Christensen, J. Freese, and E. Miguel. *Transparent and Reproducible Social Science Research*. University of California Press, 2019.

S. Correia and M. P. Seay. require: Package dependencies for reproducible research. *Stata Journal*, 24(4):599–613, 2024.

M. Gentzkow and J. M. Shapiro. Code and data for the social sciences: A practitioner's guide. University of Chicago mimeo, 2014. URL `https://web.stanford.edu/~gentzkow/research/CodeAndData.pdf`.

M. Gentzkow and J. M. Shapiro. Gentzkowlabtemplate: A reproducible framework for multi-tool research projects. GitHub repository, 2024. URL `https://github.com/gentzkowlab/GentzkowLabTemplate`.

D. Goldemberg. dependencies: Stata module to manage package dependencies. Statistical Software Components, Boston College, 2021.

R. Guiteras, A. Kim, B. Quistorff, and C. Shumway. statacons: An scons-based build tool for stata. *Stata Journal*, 23(1):148–196, 2023.

E. F. Haghish. Developing, maintaining, and hosting stata statistical software on github. *Stata Journal*, 20(4):931–951, 2020.

S. Herbert, H. Kingi, F. Stanchi, and L. Vilhuber. Reproduce to validate: A comprehensive study on the reproducibility of economics research. *Canadian Journal of Economics*, 57(3): 961–988, 2024.

F. Mölder et al. Sustainable data analysis with snakemake. *F1000Research*, 10:33, 2021.

R. B. Newson. Stata tip 147: Porting downloaded packages between machines. *Stata Journal*, 22(4):996–997, 2022.

R. Picard. project: Stata module providing a set of tools to build and manage a stata project. Statistical Software Components S457685, Boston College, 2013. URL `https://ideas.repec.org/c/boc/bocode/s457685.html`.

R. M. Stallman, R. McGrath, and P. D. Smith. *GNU Make Manual*. Free Software Foundation, 2020.

StataCorp. *Stata 19 Programming Reference Manual*. College Station, TX, 2025.

L. Vilhuber. Reproducibility and replicability in economics. *Harvard Data Science Review*, 2 (4), 2020.

L. Vilhuber. ssc2: Stata module for installation of ssc snapshot packages. GitHub repository, 2023. URL `https://github.com/labordynamicsinstitute/stata-ssc2`.

G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal. Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6):e1005510, 2017.

About the Author

Jan Fasnacht is a Predoctoral Research Professional at the Becker Friedman Institute's Political Economy Initiative, University of Chicago. His research focuses on political economy, economic history, and development.