Automate macOS Course

Flavio Copes

Updated August 3, 2026

Contents

Automate repeatable Mac work with Shortcuts, shell commands, AppleScript, Finder actions, launchd jobs, logs, and safe failure handling.

What you'll learn: Build a documented Mac automation that accepts clear input, runs without hidden interaction, reports failures, and is safe to run twice.

Choose the automation

Define the repeated task, select the smallest suitable Mac automation tool, and design input, output, and rollback.

Describe the repeated task

Turn a vague wish to automate into a trigger, input, transformation, output, success check, and failure boundary.

Every automation in this course starts the same way: with a task you already do by hand. Before you open Shortcuts or write a script, describe that task precisely. Most automations fail here, in the description, not in the code.

Start with one task you already perform. Write down six things about it:

- **Trigger**: what starts the work. A hotkey, a Finder selection, a schedule, a new file appearing in a folder.
- **Input**: the exact data the task needs. File paths, text, a folder.
- **Actions**: the ordered steps, the way you would explain them to a colleague.
- **Output**: what exists afterwards that did not exist before.
- **Success check**: the evidence that proves it worked.
- **Failure boundary**: what must never happen.

Here is a real one, written as a small task card:

```
trigger: I press a hotkey after taking a screenshot
input: the newest PNG on the Desktop
actions: rename to acme-2026-08-03.png, move to ~/Projects/acme/media
output: the renamed file in the media folder
success: file exists at the destination, Desktop copy is gone
never: overwrite an existing file, touch anything that is not a PNG
```

Compare two candidate descriptions. ”When I finish a screenshot, rename it with the project and date, move it into the project media folder, then reveal the new file.” This is testable: you can run it and check every claim. ”Organize my Mac” is not. It has no trigger, no defined input, and no way to know when it succeeded.

The failure boundary deserves extra care. Record what must never happen, such as replacing an existing file or processing a folder outside the chosen project. The first time your automation meets unexpected input, that list is the difference between a harmless no-op and lost work.

One more exercise before you build anything: perform the task manually one last time, and narrate every step out loud. You will notice small decisions you make without thinking, like skipping certain files or picking a different name when one is already taken. Each of those decisions becomes an explicit rule in the card, or it becomes a bug later.

Choose the smallest tool

Select Shortcuts, a shell script, AppleScript, or launchd according to the boundary the task actually crosses.

macOS gives you several automation tools, and they overlap. The mistake I see most often is reaching for the most powerful tool available. Pick the smallest one that crosses the boundary your task actually crosses.

The four main options

Use **Shortcuts** when the workflow connects supported app actions and benefits from Finder, Share Sheet, or keyboard triggers. It is the friendliest place to glue app features together, and the only option here that also runs on iPhone and iPad.

Use a **shell script** for files, text, commands, and reliable non-interactive work. If the task is "take these files, transform them, put them there", a script in `~/bin` is usually the whole answer.

Use **AppleScript** when a scriptable application exposes the operation through Apple events. That is the sanctioned way to ask Finder, Mail, or Notes to do something app-specific from the outside.

Use `launchd` when the question is *when*, not *what*: start a command at login, on a schedule, or when a watched path changes. launchd starts things. It does not transform data.

A quick decision test

Ask what boundary the task crosses:

```
only files and text             -> shell script
needs actions inside an app     -> AppleScript, or a Shortcuts action
needs a Finder or hotkey trigger -> Shortcuts wrapping the script
needs to run unattended         -> launchd starting the script
```

These compose. A common shape in this course: a shell script does the work, a Shortcut triggers it from Finder, and a LaunchAgent runs the same script on a schedule. Each tool stays in its lane.

The tool to avoid

Do not begin with UI clicking. UI automation simulates a user moving through windows and menus, so it is fragile and requires powerful Accessibility permission. It breaks when a button moves, a label changes, or an unexpected dialog appears. Reach for it last, after every other interface has failed you. A later lesson covers how to contain it when you truly have no choice.

One warning sign that you picked wrong: friction. If your Shortcut has become one giant "Run Shell Script" action, the task wanted a script. If your script is mostly `osascript` calls, the task wanted AppleScript from the start.

Design input and output

Give an automation explicit data types, paths, exit behavior, and output so it can run from more than one trigger.

An automation that silently reads the current Finder selection is hard to test. You cannot run it from a script, you cannot write a fixture for it, and it behaves differently depending on which window happens to be frontmost. Prefer explicit input such as file paths, text, or a project directory.

The same goes for output. An automation that "just does things" leaves the next step nothing to work with. Print the paths you created on stdout, keep messages on stderr, and exit non-zero on failure. That is the whole interface.

For a shell command, write the contract before the implementation:

```
input: one existing image path
output: new JPG path on stdout
failure: non-zero exit with message on stderr
```

```
side effect: original remains unchanged
```

Then implement exactly that, and nothing more:

```
#!/bin/zsh
if [[ ! -f "$1" ]]; then
  echo "input file not found: $1" >&2
  exit 1
fi
output="${1%.*}.jpg"
sips -s format jpeg "$1" --out "$output" >/dev/null
echo "$output"
```

`sips` is the built-in macOS image tool, so this runs on any Mac. Verify the contract from both directions:

```
./to-jpg.sh ~/Desktop/sample.png
# /Users/flavio/Desktop/sample.jpg
echo $?
# 0

./to-jpg.sh missing.png
# input file not found: missing.png
echo $?
# 1
```

Because data goes to stdout and complaints go to stderr, another program can capture the result with `output=$(./to-jpg.sh "$file")` without accidentally parsing error text as a path. That is what makes the automation trigger-independent: a Shortcut, a scheduled job, and your own terminal all call the same contract and read the same answer.

Shortcuts can follow the same contract with accepted input types and **Stop and Output**. Declare which types the shortcut receives, decide what happens when no input arrives, and end with an explicit output instead of relying on whatever the last action happened to return.

The common failure here is mixing the streams. If your script echoes progress messages to stdout, the caller receives ”processing sample.png” glued to the real path, and the next step fails on a file that does not exist. Everything that is not the answer belongs on stderr.

Build a safe dry run

Preview every planned file or application action before allowing the automation to change real state.

A **dry run** shows what an automation would do, without doing it. Add a dry-run mode before the first destructive operation, while the script is young. It should print the exact source, destination, application command, or deletion it would perform.

The mechanics are one flag and one branch:

```
#!/bin/zsh
dry_run=0
if [[ "$1" == "--dry-run" ]]; then
  dry_run=1
  shift
```

```
fi

if [[ $dry_run == 1 ]]; then
  printf "move %q -> %q\n" "$source" "$destination"
else
  mv -- "$source" "$destination"
fi
```

The `%q` format matters: it quotes each path the way the shell would need it, so a filename with spaces or a stray quote shows up honestly in the preview instead of looking like two separate files.

Run the preview against real input before every first live run:

```
./sort-screenshots --dry-run
# move Screenshot\ 2026-08-03.png -> /Users/flavio/Projects/acme/media/acme-2026-08-03.png
```

Read each printed line as a question: is this the file I meant, going where I meant? If any line surprises you, you just found a bug for free.

Test duplicates, spaces, missing input, and an unexpected directory. A preview that hides edge cases is not a safety control. If the dry run reports three files and the live run touches forty, the preview lied, and it gave you false confidence at the worst possible moment.

That is the one real failure mode of this pattern: the two paths drift apart. Someone adds a deletion to the live branch and forgets the preview branch. Keep the decision logic shared and branch at the last possible moment, at the single line that mutates state. The example above does exactly that: both branches use the same `$source` and `$destination`, and only the final action differs.

My advice is to make a new automation default to dry-run mode and require an explicit `--run` flag for its first weeks. You will be surprised how often the preview catches something you were sure was fine.

Check your understanding: choose the automation

Check the tools, decisions, safety boundaries, and failure handling from choose the automation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Connect apps and files

Use Shortcuts, Finder Quick Actions, open, clipboard commands, and file metadata to connect desktop work.

Build a Shortcut with a contract

Create a Shortcuts workflow that accepts files, transforms them, and returns output without depending on hidden prompts.

Time to build this module's real workflow. Create a shortcut that accepts image files as input. Add actions to convert them to JPEG and save copies in a chosen test folder.

Open Shortcuts, create a new shortcut, and wire it like this:

```
receives: Images (from Quick Actions and the share sheet)
if there is no input: Stop and Respond ("no images received")
actions:
  1. Convert Image   -> to JPEG
  2. Save File       -> to ~/Pictures/prepared/, "Ask Where to Save" off
  3. Stop and Output -> the saved files
```

Two of those settings are the contract, and both live in the input header at the top of the editor.

Set the accepted input type to images only. Everything else gets rejected at the door instead of failing halfway through the actions.

Then define what happens when no input arrives. **Stop and Respond** is the honest choice for automation: it fails loudly instead of opening a file picker. A shortcut that quietly waits for a human defeats the point of automating.

Finish with **Stop and Output** so another process can receive the result. Without it the shortcut still works when you click it, but it returns nothing useful to a caller — and in the next lesson we call it from the terminal.

Now test it. Use disposable images first. Keep the originals and choose a destination that cannot overwrite them. Run it with one image, then run it with the same image again: the second run reveals whether your Save File settings duplicate, overwrite, or rename on collision.

Verify each run two ways: the destination folder contains the JPEG copies you expected, and the output preview at the bottom of the editor shows the files the shortcut returned. If the destination is somewhere protected, the first run also triggers a permission prompt — approve it once, deliberately.

Rename the shortcut only after the workflow passes the same input twice. The name is part of the interface: scripts will call it by that exact string, so renaming it later is a breaking change for everything built on top.

Run Shortcuts from Terminal

Pass files to a Shortcut with the shortcuts command and capture its output without turning file paths into plain text.

macOS ships a command-line tool that turns every shortcut into something scripts can call. The `shortcuts` command runs a named shortcut. Use the input-path option when the input is a file rather than text from a pipe.

First, confirm the exact name you will call:

```
shortcuts list
# Prepare Screenshot
# Resize for Blog
```

Run a shortcut and write its output:

```
shortcuts run "Prepare Screenshot" \
  --input-path "$HOME/Desktop/sample.png" \
  --output-path "$HOME/Desktop/output"
```

Quote names and paths. Shortcut names contain spaces almost by definition, and an unquoted name becomes two arguments and a ”shortcut not found” error.

Check the result the way you would check any command:

```
echo $?
# 0
ls "$HOME/Desktop/output"
# sample.jpg
```

A zero exit status means the shortcut finished. A non-zero one means it failed or was cancelled, which is exactly what a calling script needs to stop a pipeline. And the `--output-path` flag only receives something because the shortcut ends with **Stop and Output** — the contract from the previous lesson is what makes this bridge work.

The failure mode specific to this bridge is hidden interaction. Design command-line shortcuts without alerts or selection dialogs, because hidden interaction can leave automation waiting forever. When a scheduled script runs the shortcut, there is nobody in front of the dialog. The run does not error. It just hangs, holding its lock and its schedule slot, until you notice days later. If you take one rule from this lesson, take that one.

Expect a one-time permission prompt too: the first time your terminal runs a given shortcut, macOS may ask whether to allow it. Run the command interactively once and approve it before wiring it into anything unattended, otherwise the first scheduled run blocks on a question nobody sees.

Open files, URLs, and applications

Use the macOS open command to hand an item to Launch Services or select a specific application deliberately.

The `open` command connects shell work to Mac applications. It can open a file with its default app, open a URL, or choose an application. Behind it sits **Launch Services**, the same system that decides what happens when you double-click something in Finder.

Examples:

```
open report.pdf
open https://localhost:4321/
open -a "Visual Studio Code" project
```

The first line opens the PDF in whatever app owns PDFs, Preview on most Macs. The second opens your dev server in the default browser. The third overrides the default: `-a` opens the item with a specific application, here a project folder handed to an editor.

Two more variants earn their place in automations:

```
open .
open -R ~/Projects/acme/media/acme-2026-08-03.png
```

`open .` shows the current directory in Finder, the fastest bridge from a terminal session to the GUI. `open -R` reveals the file in a Finder window and selects it, which is the polite way for an automation to finish: the user sees exactly what was produced, already selected.

Verification is direct: the right app comes to the front with the right content. For scripting, check the exit status too:

```
open missing.pdf
# The file /Users/flavio/missing.pdf does not exist.
echo $?
```

```
# 1
```

A non-zero exit lets your script stop instead of carrying on without the document it was supposed to show.

Opening is a user-visible side effect. Validate every path and URL first, especially when input comes from another program or downloaded data. `open` launches whatever it is given: a URL pointing somewhere hostile, a file whose extension hides what it really is. In an automation, an unexpected `open` is your machine acting without you. Allow only the schemes you expect, like `https:` and paths inside your project, and refuse everything else before the command runs.

Build a Finder Quick Action

Expose a tested file workflow in Finder while preserving explicit input and predictable output.

A **Quick Action** puts your automation in Finder's right-click menu, next to the built-in ones. It is the difference between a workflow you occasionally run and a workflow you actually use.

Take the image shortcut from earlier in this module. In its details pane, configure the Shortcut to appear in **Quick Actions** and accept only the file types it understands. Finder supplies the selected items as input.

```
details pane -> Use as Quick Action -> Finder: on
receives: Images (from Quick Actions)
if there is no input: Stop and Respond
```

Restricting the input type does real work here. With images only, the action stays available for a selected PNG and out of the way for a PDF, instead of appearing everywhere and failing after the user picks it.

Inside the workflow, treat the selection as a batch, because Finder sends everything that was selected. For each file, derive a new destination and check whether it already exists. Return the created files so Finder or the next action can reveal them.

To use it, select one or more images in Finder, right-click, and look under **Quick Actions**. The first run on files in a protected folder such as Desktop or Downloads can trigger a permission prompt. That is expected — approve it once and rerun.

Then test like you mean it: one file, several files, a filename with spaces, and an unsupported item. The several-files case is where Quick Actions differ most from terminal runs, and where duplicate-name collisions surface. Watch the destination folder while you run each case, and count the results.

Keep the transformation logic usable from Terminal so the Finder trigger stays thin. The Quick Action should collect the selection and hand it over, nothing more. When the logic lives in a script or shared shortcut, you can test it with fixtures from the shell — and the same logic can later run on a schedule, unchanged.

The failure mode: building the whole workflow inside the Quick Action. It works, but every test now requires a hand-made Finder selection, and none of it is reusable anywhere else.

Check your understanding: connect apps and files

Check the tools, decisions, safety boundaries, and failure handling from connect apps and files.

Answer each question before continuing. Read the explanation after every answer, including the

ones you get right.

Take this interactive quiz online

Script applications

Inspect scripting dictionaries, send Apple events, combine AppleScript with shell commands, and avoid brittle UI scripting.

Inspect an application dictionary

Read the commands, classes, properties, and terminology an application deliberately exposes to AppleScript.

Scriptable Mac apps publish a **scripting dictionary**: the commands, classes, and properties they deliberately expose to AppleScript. A dictionary is the application's automation contract. Before scripting any app, read its contract.

Open Script Editor and choose **File → Open Dictionary**. Select an application you use, then find one command and the object it accepts. The dictionary browser groups terminology into *suites*, each containing commands (the verbs) and classes (the nouns, with their properties).

Finder exposes files and folders. Other apps may expose documents, windows, tabs, or no useful commands at all. An empty or trivial dictionary is an answer too: it tells you AppleScript is the wrong tool for that app before you spend an evening finding out the hard way.

You can also dump a dictionary from the terminal:

```
sdef /System/Library/CoreServices/Finder.app | head -30
```

`sdef` prints the raw XML that Script Editor renders. It is useful when you want to search a large dictionary with `grep` instead of scrolling.

Once you find a promising property, test it read-only from Script Editor:

```
tell application "Finder" to get name of startup disk
```

The result pane shows `"Macintosh HD"`, or whatever your disk is called. A small read like this verifies three things at once: the terminology is right, the app responds, and you have permission to talk to it.

Do not guess terminology from the interface label. The menu may say one thing while the dictionary names the object something else entirely, and AppleScript rejects the words the UI taught you with an unhelpful error. Use the dictionary and test read-only properties before sending commands that change documents.

The realistic failure: a property exists in the dictionary but the app implements it badly. Dictionaries are contracts, and some apps honor theirs loosely, especially older ones. When a documented command misbehaves, test it in isolation in Script Editor before assuming the bug is in your script.

Send a small Apple event

Write and run a minimal AppleScript command against Finder before combining application scripting with shell automation.

An **Apple event** is a structured message one process sends to another: give me this property,

perform this command. AppleScript is the language most people use to send them — every `tell application` block compiles into Apple events under the hood.

Start with a read-only request. Ask Finder for the path of the front window:

```
tell application "Finder"
  POSIX path of (target of front window as alias)
end tell
```

Run it in Script Editor and inspect the result. With a Finder window open on Downloads, the result pane shows:

```
"/Users/flavio/Downloads/"
```

The first run does something else too: macOS asks whether Script Editor may control Finder. That is a **TCC permission prompt**, and it appears once per pair of controlling app and target app. Approve it and the grant appears under System Settings → Privacy & Security → Automation. Deny it and every future event to Finder fails with error `-1743`, "Not authorized to send Apple events", until you change it there.

Now break the script on purpose. Close every Finder window and run it again. It errors, because there is no front window to ask about. If Finder has no open window, the script needs an explicit fallback:

```
tell application "Finder"
  if (count of windows) is 0 then return POSIX path of (desktop as alias)
  POSIX path of (target of front window as alias)
end tell
```

AppleScript errors are part of the contract, not messages to ignore. The scripts that survive are the ones that decide in advance what happens when the assumption behind a request turns out false.

Keep events small while you learn an app: one property read, one result checked. A `tell` block that chains five operations tells you nothing useful when it fails in the middle. Once each event is verified on its own, combining them with shell automation later is assembly, not archaeology.

Call AppleScript with osascript

Invoke a small AppleScript from the shell, capture its result, and keep data separate from script source.

`osascript` runs AppleScript and returns its result to the shell. It is the bridge that lets a zsh script use everything the previous two lessons covered.

For a one-liner, pass the source with `-e`:

```
osascript -e 'tell application "Finder" to get name of startup disk'
# Macintosh HD
```

Single quotes around the whole expression, double quotes inside for AppleScript strings. That division of labor works for one line. Beyond that, quoting becomes the main problem, so keep longer scripts in files.

File-based scripts can also receive arguments. Anything you pass after the filename arrives in an `on run argv` handler inside the script, so the shell side supplies the data and the AppleScript side stays generic.

Save the front-window script from the previous lesson as `scripts/front-finder-folder.applescript`, then run it and capture the result:

```
folder=$(osascript scripts/front-finder-folder.applescript) || exit
printf "folder=%s\n" "$folder"
```

Two details in those lines carry the weight. Command substitution captures the AppleScript result as a plain string. And `|| exit` lets a non-zero AppleScript exit stop the shell workflow: when the script throws, `osascript` prints the error to stderr and exits non-zero, so your workflow fails at the right line instead of continuing with an empty variable.

Treat the result as untrusted input. Quote it, validate the expected path, and only then use it:

```
if [[ ! -d "$folder" ]]; then
  echo "not a directory: $folder" >&2
  exit 1
fi
```

The permission model follows the caller. When Terminal runs `osascript`, it is Terminal that needs Automation permission for the target app, and the TCC prompt names Terminal. The same script inside a scheduled job runs with no one to show a prompt to, which is a common reason automation that worked during testing fails silently later. Error `-1743` in stderr is the signature of that failure. Test the script in the same context it will really run in, not only from your interactive shell.

Avoid UI scripting when possible

Recognize when Accessibility-driven clicks are brittle and replace them with files, URLs, dictionaries, Shortcuts actions, or APIs.

UI scripting searches windows and controls, then simulates user actions. Through System Events, AppleScript can click any button and choose any menu item. That sounds like the universal automation tool, right up until you have to maintain one of these scripts.

Here is what it looks like:

```
tell application "System Events"
  tell process "Notes"
    set frontmost to true
    click menu item "New Note" of menu "File" of menu bar 1
  end tell
end tell
```

Nothing in that script expresses intent. It encodes where things happen to be: a menu named ”File”, an item named ”New Note”, a process that must be frontmost when the click lands. Labels, timing, focus, localization, and layout changes can break it. A macOS update that renames a menu item, a slow launch that makes the click arrive early, a French system where the menu is ”Fichier” — all fatal, all silent until they strike.

It also demands the broadest permission on the system. System Events needs **Accessibility** access (System Settings → Privacy & Security → Accessibility), and a tool holding that grant can drive any application, not just the one you meant.

So before enabling Accessibility, check for a Shortcuts action, AppleScript dictionary command, URL scheme, command-line interface, or documented API. These interfaces express intent instead of screen position. Work down the list in order: the app's dictionary first, then its Shortcuts actions,

then its documentation. For Notes, the dictionary command `make new note` does what the click script above does, in one line that survives redesigns and localization.

If UI scripting is unavoidable, contain it. Narrow permission to one trusted tool, verify the active application before sending events, add timeouts, and stop on an unexpected window. A UI script that keeps clicking after a surprise dialog appears is typing into the wrong app with your permissions.

Treat every UI script as debt with a review date. Note the app version it was built against, and retest it after each update of that app or of macOS.

Check your understanding: script applications

Check the tools, decisions, safety boundaries, and failure handling from script applications.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Schedule with launchd

Build and operate a per-user LaunchAgent with explicit paths, environment, output, and lifecycle controls.

Choose a LaunchAgent or LaunchDaemon

Run user automation in the user session and reserve system-wide daemons for work that genuinely needs system scope.

launchd runs jobs in two contexts, and choosing the wrong one produces confusing failures. A **LaunchAgent** runs for a logged-in user. It can work with that user's files and session: the home folder, the user Keychain, the GUI. A **LaunchDaemon** runs in the system context and must not depend on a graphical user session. Daemons exist for machine-wide services that run before, or without, anyone logging in.

Location decides which one you get:

```
~/Library/LaunchAgents/     agents for this user only
/Library/LaunchAgents/      agents for every user who logs in
/Library/LaunchDaemons/     system daemons, run as root by default
```

The `/System/Library` equivalents belong to macOS itself. Leave those alone.

Personal automation belongs in `~/Library/LaunchAgents`. Sorting screenshots, tidying project folders, backing up your own data: all of it is user-scoped work on user-scoped files.

The two contexts also live in different launchd **domains**, which matters for every command later in this module:

```
launchctl print gui/$(id -u) | head    # your user's GUI domain
sudo launchctl print system | head     # the system domain
```

Here is the trap this lesson exists to prevent. Your agent fails with a permission error, and you notice that a daemon running as root would make the error disappear. Do not turn it into a root daemon to work around a path or permission mistake. That trade swaps a visible error for a job with maximum privilege that cannot see your login session, has no access to your user Keychain,

and may run while nobody is logged in to notice it misbehaving. One wrong path in a root job can damage files no user-level bug could touch.

Decide on paper first. Write down the required user, files, network access, and trigger. Choose the narrowest context that satisfies them. If the answer involves your home directory or anything tied to your session, it is an agent — and the fix for the permission error is fixing the permission.

Write a minimal LaunchAgent

Create a valid property list with a unique label, explicit program arguments, and one deliberate trigger.

A launchd job is described by a property list: an XML file that says what to run and when. Here is a complete, minimal LaunchAgent:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.flaviocopes.screenshot-sorter</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/flavio/bin/sort-screenshots</string>
  </array>
  <key>StartInterval</key>
  <integer>900</integer>
</dict>
</plist>
```

Save it as `~/Library/LaunchAgents/com.flaviocopes.screenshot-sorter.plist`. Matching the filename to the label is a convention worth keeping: six months from now, the filename is how you will find the job.

Three keys, three decisions.

`Label` is the job's identity in your user's launchd domain. Use a reverse-domain label, unique on the machine. Every `launchctl` command you run later refers to it.

`ProgramArguments` lists the executable and its arguments, one `<string>` per element. Use an absolute executable path, because launchd resolves nothing for you. And do not put shell operators inside `ProgramArguments`; launchd starts the executable directly, so a `>` or `&&` in there arrives as a literal argument to your program, not as redirection. If the job needs shell features, put them inside the script.

`StartInterval` is the trigger: run every 900 seconds. Other triggers exist — `StartCalendarInterval` for clock times, `WatchPaths` for reacting to file changes, `RunAtLoad` for login — but keep the first job small. Add only one trigger, then prove when and why it runs before adding more conditions.

Validate the complete file with `plutil -lint`:

```
plutil -lint ~/Library/LaunchAgents/com.flaviocopes.screenshot-sorter.plist
```

```
# .../com.flaviocopes.screenshot-sorter.plist: OK
```

A malformed plist is the most common first failure, and launchd's own complaint about it is unhelpfully generic. `plutil` points at the exact line. Nothing runs yet, though: writing the file registers nothing with launchd. Loading it comes in a later lesson, after we pin down the job's environment.

Control environment and output

Give a scheduled job absolute paths, a working directory, and dedicated output files instead of depending on Terminal configuration.

A LaunchAgent does not inherit your interactive `.zshrc` environment. Commands that work in Terminal can fail because PATH, working directory, or variables differ. This single fact explains most "works when I run it, fails on the schedule" reports.

The PATH a launchd job receives is short:

```
/usr/bin:/bin:/usr/sbin:/sbin
```

No `/opt/homebrew/bin`, no `~/bin`. A script that calls `jq` or anything else Homebrew-installed dies with `command not found`. Call tools by absolute path (`/opt/homebrew/bin/jq`), or set PATH explicitly at the top of the script, where it lives in version control next to the code that depends on it.

The working directory is not your project folder either. Configure `WorkingDirectory`, `StandardOutPath`, and `StandardErrorPath` when useful:

```
<key>WorkingDirectory</key>
<string>/Users/flavio</string>
<key>StandardOutPath</key>
<string>/Users/flavio/Library/Logs/screenshot-sorter.log</string>
<key>StandardErrorPath</key>
<string>/Users/flavio/Library/Logs/screenshot-sorter.err.log</string>
```

The two output paths are your only window into a job that runs with no terminal attached: launchd appends everything the job prints to those files. One caveat — launchd does not build the directory tree for you. Create parent directories before loading the job, or the output silently goes nowhere.

When a job misbehaves and you suspect the environment, make the job show you what it sees:

```
/usr/bin/env > /Users/flavio/Library/Logs/sorter-env.txt
```

Put that line at the top of the script, run the job once, and compare the dump with `env` in your terminal. The difference is usually the whole answer.

Keep secrets out of the property list and logs. A plist is a plain file on disk, and log files get zipped into bug reports. Retrieve secrets at runtime through a protected mechanism — the user Keychain via `security find-generic-password` works for agents, since they run in your session — and avoid shell tracing: `set -x` prints every expanded command, secrets included, straight into the error log.

Load, test, and remove the job

Validate a LaunchAgent, start it in the user domain, inspect state and logs, then remove it cleanly.

The plist exists and the environment is pinned down. Now hand the job to launchd, watch it run,

and — just as deliberately — take it back out.

Validate the property list before loading it:

```
plutil -lint ~/Library/LaunchAgents/com.flaviocopes.screenshot-sorter.plist
# OK
```

Load it into your user's GUI domain with `bootstrap`, force an immediate run with `kickstart`, and inspect it with `print`. Use the current user domain and label when inspecting the job:

```
launchctl bootstrap gui/$(id -u) ~/Library/LaunchAgents/com.flaviocopes.screenshot-sorter.pli
launchctl kickstart -k gui/$(id -u)/com.flaviocopes.screenshot-sorter
launchctl print gui/$(id -u)/com.flaviocopes.screenshot-sorter
```

`kickstart -k` runs the job now instead of waiting up to fifteen minutes for `StartInterval`, which makes testing bearable. The `print` output is dense, but three parts matter: the state, the last exit code, and the program it resolved. For a quicker glance, `launchctl list | grep screenshot` shows the PID (or `-` when idle) and the last exit status. Then confirm the job did its actual work: check the log files and the destination folder.

If `bootstrap` answers `Bootstrap failed: 5: Input/output error`, the job is usually already loaded — boot it out first — or the plist has a problem `plutil` cannot see, like a program path that is not executable.

These are the `launchctl` subcommands documented by the installed macOS release. Older tutorials use `load` and `unload`; they still work but are the legacy interface. When in doubt, trust `man launchctl` on your machine over a blog post.

Removal is `bootout`:

```
launchctl bootout gui/$(id -u)/com.flaviocopes.screenshot-sorter
```

Record the exact removal step in the runbook while it is fresh. After removal, confirm the label no longer appears in `launchctl list` and the process has stopped. Keep the property list until the uninstall check passes, so you can reload and retest if removal reveals a problem.

Check your understanding: schedule with launchd

Check the tools, decisions, safety boundaries, and failure handling from schedule with launchd.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate the automation

Add idempotence, locking, logs, notifications, tests, recovery, and a complete automation runbook.

Make the work idempotent

Design an automation so running it twice reaches the same correct state without duplicating or corrupting work.

An automation is **idempotent** when running it twice reaches the same correct state as running it once. This matters more for automations than for hand work, because automations rerun: schedules

fire again, a Quick Action gets clicked twice, launchd restarts a job that died mid-write.

An idempotent file workflow checks whether the desired output already exists and whether it matches the input it processed. It does not append the date again on every retry, turning one screenshot into `acme-2026-08-03-2026-08-03.png` on the second pass.

Three techniques do most of the work: stable destination names, temporary files, and an atomic final move.

```
name="acme-$(date +%F).png"
destination="$HOME/Projects/acme/media/$name"

if [[ -e "$destination" ]]; then
  echo "already processed: $name"
  exit 0
fi

tmp=$(mktemp "$destination.XXXXXX")
cp "$HOME/Desktop/screenshot.png" "$tmp"
mv "$tmp" "$destination"
```

The destination name is stable: computed from the input, not from ”now, plus a counter”. Running twice computes the same name, hits the existence check, and exits cleanly.

The temporary file plus `mv` is the atomic part. Within one volume, `mv` is a rename: any other process sees either no file or the complete file, never a half-copied one. If the job dies during `cp`, the destination never existed, and the next run starts over safely. Record processed input only after the output is complete — here, the `mv` itself is that record, because the destination's existence is the marker.

Verify by doing what reality will do to you: run the same fixture twice.

```
./sort-screenshots && ./sort-screenshots
# moved acme-2026-08-03.png
# already processed: acme-2026-08-03.png
```

The second run should report no change or the same verified result, not another copy.

The failure mode to hunt for is a marker written before the work finishes. If the script logs ”done”, or records the input as processed, and then crashes before the `mv`, every future run skips a file that was never actually produced. Markers come last.

Prevent overlapping runs

Use a lock or launchd lifecycle behavior so two triggers cannot modify the same files simultaneously.

Schedules, Finder actions, and manual retries can overlap. You click the Quick Action while the interval job is mid-run, or a slow run is still working when the next interval fires. Two instances may choose the same destination before either finishes writing, and the result is duplicated or corrupted output that neither run can explain on its own.

launchd gives you one guarantee for free: it does not start a second instance of a label while the first is still running. But that only covers launchd's own copies. The same script run by hand, or triggered through a Quick Action, is a separate process launchd knows nothing about. For that you need a **lock**.

macOS does not ship a `flock` command, so the portable pattern is `mkdir`, which is atomic: it either creates the directory or fails, with nothing in between.

```
lock="$HOME/.local/state/screenshot-sorter.lock"

if ! mkdir "$lock" 2>/dev/null; then
  echo "another run holds the lock, exiting" >&2
  exit 0
fi
echo $$ > "$lock/pid"
trap 'rm -rf "$lock"' EXIT
```

Create a lock atomically, store the owning process information, and remove it on normal exit. The `trap` releases it even when the script fails partway through.

Verify the lock by simulating the collision:

```
./sort-screenshots & ./sort-screenshots
# another run holds the lock, exiting
```

Then handle the crash case. A hard kill can leave the directory behind, and every future run refuses to start. Treat stale-lock recovery as a separate verified path: read `$lock/pid`, check whether that process is still alive with `kill -0`, and only then conclude the lock is stale. Do not delete locks blindly at startup; that reintroduces the exact race the lock exists to prevent.

Also decide whether new work should wait, exit, or replace an older run. The example exits, which suits an interval job: the next scheduled run picks up whatever is left. A lock without a documented policy only changes the failure.

Log and notify without leaking

Record start, result, duration, and actionable errors while keeping file contents, credentials, and private paths out of notifications.

A scheduled automation runs with nobody watching. Logs are how you find out next week whether it worked — and logs are also where automations leak private data. The goal is recording enough, and no more.

Logs should identify the automation, run, input category, result, and duration. That is one short structured line, not a transcript:

```
logger -t screenshot-sorter "run=20260803T1715 files=3 skipped=1 status=ok duration=2s"
```

`logger` writes a small event into the macOS unified log, where it gets timestamps and retention for free. Verify it landed:

```
log show --predicate 'eventMessage CONTAINS "screenshot-sorter"' --last 15m
```

For higher-volume output, a dedicated file under `~/Library/Logs` with restrictive permissions works too — that is what `StandardOutPath` and `StandardErrorPath` already give a LaunchAgent. Either way, write errors to standard error so launchd can keep them separate from routine output. When something breaks, you read the small error file, not a thousand lines of success.

Now the leaking part. Do not paste tokens, clipboard content, document text, or full private paths into logs. The unified log is readable by other tools and admins, log files get zipped into support bundles, and clipboard content might be whatever a password manager copied last. Log `files=3`,

not three filenames with a client's name in them. Retain enough evidence to diagnose the boundary, not the user's data.

Notifications follow a stricter rule than logs: send a notification only when the user can act on it.

```
osascript -e 'display notification "3 screenshots need review" with title "Screenshot sorter"
```

A "run succeeded" banner every fifteen minutes trains you to dismiss the automation, and the day it fails you will dismiss that too. Notify on the failure that needs a decision. Log the successes.

The realistic failure mode is debugging pressure. Something breaks, you add `set -x` or start echoing variables, and a token lands in the error log permanently. Add that detail temporarily, behind the dry-run flag, and strip it before the job goes back on the schedule.

Finish the automation runbook

Test normal, duplicate, missing-input, permission, and interruption paths, then document installation, verification, pause, and removal.

Build one final automation from this course: accept selected screenshots, create safe project copies, and run either from Finder or a LaunchAgent. Every module contributed a piece — the task card, the contract, the shortcut trigger, the script, the plist, the lock, the log line. What remains is proving the whole thing and writing it down.

Test the paths that will really happen

Test valid input, duplicates, spaces, missing files, denied permission, two simultaneous runs, and interruption before the final move. Each of those replays a lesson from this course against the finished automation, and each has a defined correct answer if you wrote the contract: duplicates skip, denied permission fails loudly, an interrupted run leaves no partial file behind.

Preserve fixtures and expected results:

```
fixtures/
  one-file/    sample.png           -> expect: 1 copy created
  duplicate/   sample.png, run x2   -> expect: "already processed"
  spaces/      Screen Shot 3.png    -> expect: 1 copy, name intact
  wrong-type/  notes.txt            -> expect: rejected at input
```

Kept fixtures turn "I tested it once in August" into a suite you rerun after every macOS update and every edit to the script.

Write the runbook

Document prerequisites, permissions, configuration, logs, dry run, manual invocation, scheduled invocation, disablement, removal, and recovery. In practice that is one short line or section each:

```
permissions: Terminal -> Automation -> Finder (approved 2026-08-03)
logs:        ~/Library/Logs/screenshot-sorter.log
dry run:     ~/bin/sort-screenshots --dry-run
manual run:  ~/bin/sort-screenshots
disable:     launchctl bootout gui/$UID/com.flaviocopes.screenshot-sorter
recovery:    remove ~/.local/state/screenshot-sorter.lock if the pid is dead
```

The measure of done: another person should be able to stop it safely. Not use it — stop it, at two

in the morning, without calling you. If the disable line works when pasted by someone who has never seen the project, the runbook is real.

Verify the runbook the way you verified the code: follow it literally, on a clean user account or after a reboot. Every step you perform from memory instead of from the page is a step missing from the page.

Check your understanding: operate the automation

Check the tools, decisions, safety boundaries, and failure handling from operate the automation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/automate-macos/.