Setting Up a Comprehensive Research Backup System on macOS

A three-tier strategy combining Time Machine, automated Git commits, and cloud synchronisation

Ronald 'Ryy' G. Thomas

2026-02-11

Table of contents

Figure 1: A layered backup strategy protects research data against hardware failure, accidental deletion, and Git corruption.

Building confidence through redundancy.

1 Introduction

I didn't really know how precarious my research workflow was until a colleague lost several months of analytical work to a single hard-drive failure. The data, the scripts, the manuscript drafts – all gone. That event prompted me to take a hard look at my own setup, and what I found was unsettling: dozens of repositories with uncommitted changes, no automated push schedule, and a Time Machine backup that had silently stopped running weeks earlier.

The problem is not that individual backup tools are difficult to configure. Time Machine, Git, and cloud services such as Google Drive or Dropbox each work well in isolation. The difficulty lies in weaving them together into a system that runs automatically, covers every layer of risk, and does not depend on the researcher remembering to press a button.

This post documents the three-tier backup system I built for my own macOS research environment. It is not a theoretical guide – it is the actual configuration I run daily across 300+ Git repositories. I hope it saves someone from learning the same lesson the hard way.

More formally, this post documents the backup layer of the Workflow Construct described in post 52. Post 52 names backup as a load-bearing layer with the principle 'two tiers or it is not backup'; the configuration described here implements three (GitHub for source-of-truth, an external SSD via Time Machine for local cold storage, and rclone-mediated cloud mirrors for off-site warm storage), each addressing a distinct failure mode that the others cannot.

1.1 Motivations

- I was running `git push` manually and forgetting for days at a time, leaving important work vulnerable to local disk failure.
- A colleague's data loss demonstrated that a single backup layer is never sufficient for active research projects.
- I needed an automated solution that could handle hundreds of repositories without manual intervention for each one.
- Cloud synchronisation alone does not provide version history – I wanted real Git commits with meaningful timestamps.
- I wanted the system to distinguish my own repositories from collaborator forks, archive directories, and backup folders to avoid polluting other people's remotes.

1.2 Objectives

1. Configure Time Machine as a system-wide safety net for files that live outside Git.
2. Write a minimal backup script that commits and pushes all dirty Git repositories automatically.
3. Extend the minimal script into a production-grade tool with logging, error handling, user filtering, and verbose output.
4. Schedule the full script to run every 15 minutes using `cron`, creating a hands-free backup cycle.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

2 Prerequisites and Setup

This guide assumes a macOS environment with the following tools available:

- **macOS** 12 (Monterey) or later
- **Homebrew Bash** (`/opt/homebrew/bin/bash`) – macOS ships with Bash 3.2, but the script uses Bash 4+ features
- **Git** 2.30 or later, configured with SSH keys for GitHub
- **A USB external drive** (1TB recommended) for Time Machine
- **A cloud sync service** (Google Drive, Dropbox, or iCloud) for the third tier

The research directory structure assumed throughout is `~/prj/`, containing all Git repositories. Adjust paths as needed for your own layout.

3 What is a Three-Tier Backup System?

A three-tier backup system layers three independent protection mechanisms so that no single point of failure can result in data loss. Think of it like redundant safety nets at different heights: if one fails, the next catches you.

The three tiers in this system are:

1. **Automated Git commits and pushes** (every 15 minutes) – protects against uncommitted work and local corruption by pushing changes to a remote server.
2. **Cloud synchronisation** (real-time via Google Drive or Dropbox) – provides continuous file-level replication across devices, useful for non-Git files and immediate access from other machines.
3. **Time Machine backups** (hourly, system-wide) – captures the entire filesystem including system settings, application data, and any files not covered by the other two tiers.

Each tier compensates for a weakness in the others. Git does not capture large binary files well; cloud sync does not preserve commit history; Time Machine does not push data off-site. Together, they cover the full risk surface.

4 Getting Started: Configuring Time Machine

Time Machine provides system-wide backup protection and serves as the safety net for everything beyond Git repositories. The setup is straightforward but benefits from a few deliberate configuration choices.

4.1 Connect and Format the USB Drive

1. Connect your USB drive to the MacBook.
2. When prompted, **do not** use it for Time Machine yet – we will configure it properly first.
3. Open **Disk Utility** (Applications > Utilities > Disk Utility).
4. Select the USB drive from the sidebar.
5. Click **Erase**.
6. Choose format: **APFS** (recommended for modern Macs) or **Mac OS Extended (Journaled)**.
7. Name it something recognisable, such as "Research Backup".
8. Click **Erase**.

4.2 Configure Time Machine

1. Open **System Preferences** > **Time Machine**.
2. Click **Select Backup Disk**.
3. Choose your USB drive.
4. Click **Use Disk**.
5. If prompted about encryption, choose **Encrypt Backup** for security.

4.3 Customise Exclusions

1. Click **Options** in Time Machine preferences.
2. Add folders you want to exclude (Downloads, Trash, virtual machines, and similar high-churn directories).
3. **Do not exclude `~/prj`** – we want this backed up as a secondary layer behind Git.
4. Enable "Back up while on battery power" if you work unplugged frequently.

Time Machine will now back up the entire system (including `~/prj`) every hour when the USB drive is connected.

4.4 The Minimal Backup Script

Before building the full production script, it helps to see the core logic distilled to its essence. This minimal version does three things: find every Git repository under `~/prj`, check whether it has uncommitted changes, and push those changes to the remote.

```
#!/opt/homebrew/bin/bash

find "$HOME/prj" -name ".git" -type d \
    | while read git_dir; do
    cd "$(dirname "$git_dir")" || continue
    [[ -n $(git status --porcelain) ]] || continue
    git add -A
    git commit -m \
```

```
            "Auto-backup: $(date '+%Y-%m-%d %H:%M:%S')"
        git push origin main 2>/dev/null \
            || git push origin master 2>/dev/null
    done
```

This works, but it lacks error handling, logging, user filtering, and any mechanism for diagnosing failures. The full script addresses each of these gaps.

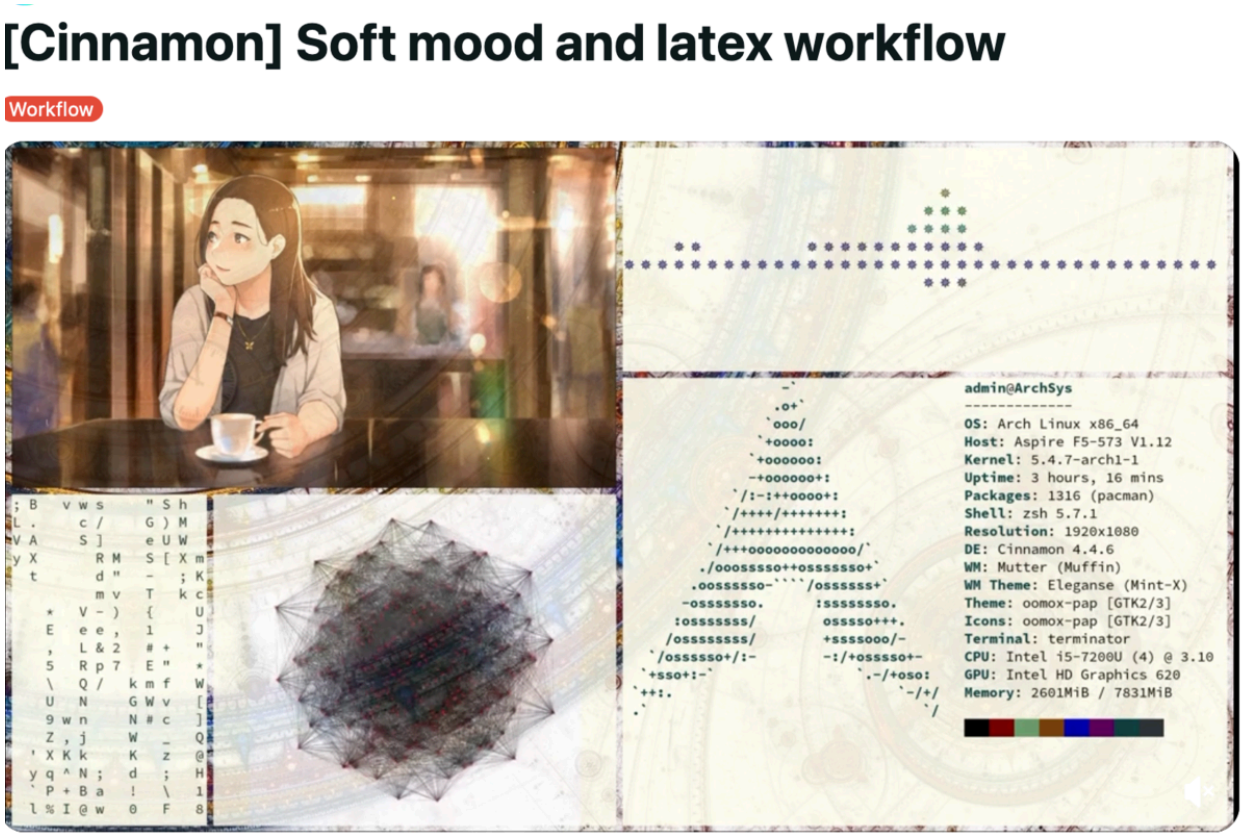

Figure 2: Terminal windows and backup logs – the machinery behind automated research protection.

5 Deeper Analysis: The Full Backup Script

The production script extends the minimal version with a comprehensive set of features. Rather than presenting the full script as a single monolithic block, I will walk through it in logical segments.

5.1 Configuration and Argument Parsing

The script begins by declaring configurable paths and parsing command-line flags. The verbose mode is essential for debugging; in normal cron operation, output goes only to the log file.

```
#!/opt/homebrew/bin/bash

RESEARCH_DIR="$HOME/prj/"
LOG_FILE="$HOME/Library/Logs/research_backup.log"
MAX_LOG_SIZE=10485760  # 10MB
VERBOSE=false

while [[ $# -gt 0 ]]; do
    case $1 in
        -v|--verbose)
            VERBOSE=true
            shift
            ;;
        -h|--help)
            echo "Usage: $0 [-v|--verbose]" \
                 "[-h|--help]"
            echo "  -v, --verbose" \
                 "Enable verbose output"
            echo "  -h, --help" \
                 "Show this help message"
            exit 0
            ;;
        *)
            echo "Unknown option: $1"
            echo "Use -h or --help for usage"
            exit 1
            ;;
    esac
done
```

5.2 Logging Infrastructure

Log rotation prevents the log file from growing without bound. The `log_message` function writes every event to disk and optionally echoes colour-coded output to the console when verbose mode is active.

```
mkdir -p "$(dirname "$LOG_FILE")"

if [[ -f "$LOG_FILE" \
      && $(stat -f%z "$LOG_FILE") \
         -gt $MAX_LOG_SIZE ]]; then
    mv "$LOG_FILE" "${LOG_FILE}.old"
    if [[ "$VERBOSE" == true ]]; then
        echo "INFO: Rotated log file" \
             "(exceeded ${MAX_LOG_SIZE} bytes)"
    fi
```

```
fi

log_message() {
    local level="$1"
    local message="$2"
    local timestamp
    timestamp=$(date '+%Y-%m-%d %H:%M:%S')
    local log_entry
    log_entry="$timestamp: [$level] $message"

    echo "$log_entry" >> "$LOG_FILE"

    if [[ "$VERBOSE" == true ]]; then
        case "$level" in
            ERROR)
                echo -e \
                    "\033[31m$log_entry\033[0m"
                ;;
            WARNING)
                echo -e \
                    "\033[33m$log_entry\033[0m"
                ;;
            SUCCESS)
                echo -e \
                    "\033[32m$log_entry\033[0m"
                ;;
            INFO)
                echo -e \
                    "\033[34m$log_entry\033[0m"
                ;;
            *)
                echo "$log_entry"
                ;;
        esac
    fi
}
```

5.3 Repository Validation Functions

Four helper functions handle the filtering logic. `check_remote` verifies that the repository has an `origin` remote. `check_user_association` ensures that only repositories belonging to the "rgt47" account are processed, preventing the script from pushing to collaborator remotes. `should_exclude_directory` skips archive and backup folders. `get_current_branch` and `branch_exists_on_remote` support intelligent push behaviour.

```
check_remote() {
    local repo_dir="$1"
    cd "$repo_dir" || return 1
    local remote_url
    remote_url=$(git remote get-url origin \
        2>/dev/null)
    [[ -n "$remote_url" ]]
}

check_user_association() {
    local repo_dir="$1"
    cd "$repo_dir" || return 1

    local remote_url
    remote_url=$(git remote get-url origin \
        2>/dev/null)
    if [[ "$remote_url" == *"rgt47"* ]]; then
        return 0
    fi

    local git_user git_email
    git_user=$(git config user.name 2>/dev/null)
    git_email=$(git config user.email 2>/dev/null)

    if [[ "$git_user" == *"rgt47"* ]] \
       || [[ "$git_email" == *"rgt47"* ]]; then
        return 0
    fi

    if [[ -z "$git_user" ]]; then
        git_user=$(git config --global \
            user.name 2>/dev/null)
    fi
    if [[ -z "$git_email" ]]; then
        git_email=$(git config --global \
            user.email 2>/dev/null)
    fi

    if [[ "$git_user" == *"rgt47"* ]] \
       || [[ "$git_email" == *"rgt47"* ]]; then
        return 0
    fi

    return 1
}
```

```
should_exclude_directory() {
    local repo_name="$1"
    local repo_path="$2"

    local lower_name lower_path
    lower_name=$(echo "$repo_name" \
        | tr '[:upper:]' '[:lower:]')
    lower_path=$(echo "$repo_path" \
        | tr '[:upper:]' '[:lower:]')

    if [[ "$lower_name" == *"archive"* ]] \
       || [[ "$lower_name" == *"backup"* ]]; then
        return 0
    fi

    if [[ "$lower_path" == *"archive"* ]] \
       || [[ "$lower_path" == *"backup"* ]]; then
        return 0
    fi

    return 1
}

get_current_branch() {
    git symbolic-ref --short HEAD 2>/dev/null \
        || git rev-parse --short HEAD 2>/dev/null
}

branch_exists_on_remote() {
    local branch="$1"
    git ls-remote --heads origin "$branch" \
        2>/dev/null | grep -q "$branch"
}
```

5.4 Main Loop: Discovery and Processing

The main loop uses `find` with null-delimited output to safely handle repository paths that contain spaces. Each repository passes through a series of checks before any Git operations are attempted.

```
log_message "INFO" \
    "Starting research backup scan" \
    " with verbose=$VERBOSE"

if [[ ! -d "$RESEARCH_DIR" ]]; then
    log_message "ERROR" \
```

```
        "Research directory $RESEARCH_DIR" \
        " does not exist"
    exit 1
fi

log_message "INFO" \
    "Scanning: $RESEARCH_DIR"

repo_count=0
backup_count=0
error_count=0
warning_count=0
skipped_count=0
excluded_count=0

while IFS= read -r -d '' git_dir; do
    repo_dir=$(dirname "$git_dir")
    repo_name=$(basename "$repo_dir")
    relative_path="${repo_dir#$RESEARCH_DIR}"

    if should_exclude_directory \
       "$repo_name" "$relative_path"; then
        log_message "INFO" \
            "Excluding (archive/backup):" \
            " $relative_path"
        ((excluded_count++))
        continue
    fi

    log_message "INFO" \
        "Processing: $relative_path"

    if ! cd "$repo_dir"; then
        log_message "ERROR" \
            "Cannot access: $repo_dir"
        ((error_count++))
        continue
    fi

    ((repo_count++))

    if ! git rev-parse --git-dir \
         >/dev/null 2>&1; then
        log_message "ERROR" \
            "Not a valid git repo:" \
            " $relative_path"
```

```
    ((error_count++))
    continue
fi

if ! check_user_association "$repo_dir"; then
    log_message "INFO" \
        "Skipping (not rgt47):" \
        " $relative_path"
    ((skipped_count++))
    continue
fi

log_message "INFO" \
    "$relative_path associated with rgt47"

if ! check_remote "$repo_dir"; then
    log_message "WARNING" \
        "No remote configured:" \
        " $relative_path"
    ((warning_count++))
    ((skipped_count++))
    continue
fi

current_branch=$(get_current_branch)
if [[ -z "$current_branch" ]]; then
    log_message "ERROR" \
        "Cannot determine branch:" \
        " $relative_path"
    ((error_count++))
    continue
fi

log_message "INFO" \
    "$relative_path on branch:" \
    " $current_branch"

git_status=$(git status --porcelain \
    2>/dev/null)

if [[ -z "$git_status" ]]; then
    log_message "INFO" \
        "$relative_path is clean"
    continue
fi
```

```
untracked=$(echo "$git_status" \
    | grep -c "^??" || echo 0)
modified=$(echo "$git_status" \
    | grep -c "^ M" || echo 0)
added=$(echo "$git_status" \
    | grep -c "^A " || echo 0)
deleted=$(echo "$git_status" \
    | grep -c "^D " || echo 0)

log_message "INFO" \
    "$relative_path: $untracked new," \
    " $modified modified, $added added," \
    " $deleted deleted"

if ! git add -A 2>/dev/null; then
    log_message "ERROR" \
        "Failed to stage: $relative_path"
    ((error_count++))
    continue
fi

log_message "INFO" \
    "Staged changes: $relative_path"

commit_message="Auto-backup:" \
    " $(date '+%Y-%m-%d %H:%M:%S')"

if git commit -m "$commit_message" \
   >/dev/null 2>&1; then
    log_message "SUCCESS" \
        "Committed: $relative_path"

    if ! branch_exists_on_remote \
         "$current_branch"; then
        log_message "WARNING" \
            "'$current_branch' not on" \
            " remote: $relative_path"

        if git push --set-upstream origin \
           "$current_branch" 2>/dev/null
        then
            log_message "SUCCESS" \
                "Created and pushed" \
                " '$current_branch':" \
                " $relative_path"
            ((backup_count++))
        else
```

```
                log_message "ERROR" \
                    "Failed to push new" \
                    " branch: $relative_path"
                ((error_count++))
            fi
        else
            if git push origin \
               "$current_branch" 2>/dev/null
            then
                log_message "SUCCESS" \
                    "Pushed '$current_branch':" \
                    " $relative_path"
                ((backup_count++))
            else
                log_message "ERROR" \
                    "Push failed:" \
                    " $relative_path" \
                    " (check network/auth)"
                ((error_count++))
            fi
        fi
    else
        if git diff --cached --quiet; then
            log_message "INFO" \
                "No changes to commit:" \
                " $relative_path"
        else
            log_message "ERROR" \
                "Commit failed:" \
                " $relative_path"
            ((error_count++))
        fi
    fi

done < <(find "$RESEARCH_DIR" \
    -name ".git" -type d -print0)
```

5.5 Summary Report

After processing every repository, the script logs aggregate statistics and, in verbose mode, prints a human-readable summary to the console.

```
log_message "INFO" "Backup scan complete"
log_message "INFO" \
    "Summary: $repo_count processed," \
```

```
    " $backup_count backed up"
log_message "INFO" \
    "Excluded: $excluded_count," \
    " Skipped: $skipped_count," \
    " Errors: $error_count," \
    " Warnings: $warning_count"

if [[ "$VERBOSE" == true ]]; then
    echo ""
    echo "=== BACKUP SUMMARY ==="
    echo "Repositories found:" \
         "$((repo_count + excluded_count" \
         " + skipped_count))"
    echo "Excluded: $excluded_count" \
         "(archive/backup)"
    echo "Skipped: $skipped_count (not rgt47)"
    echo "Processed: $repo_count"
    echo "Backed up: $backup_count"
    echo "Errors: $error_count"
    echo "Warnings: $warning_count"
    echo ""
    echo "Log file: $LOG_FILE"

    if [[ $error_count -gt 0 ]]; then
        echo ""
        echo "WARNING: There were errors" \
             "during backup. Check the log" \
             "file for details."
        exit 1
    elif [[ $warning_count -gt 0 ]]; then
        echo ""
        echo "NOTE: Backup completed with" \
             "warnings. Check the log file" \
             "for details."
    else
        echo ""
        echo "Backup completed successfully."
    fi
fi

exit 0
```

5.6 Things to Watch Out For

1. **Homebrew Bash path.** macOS ships with Bash 3.2, which lacks features the script depends on. Ensure the shebang points to `/opt/homebrew/bin/bash` (Apple Silicon) or

`/usr/local/bin/bash` (Intel). I spent an afternoon debugging failures that turned out to be Bash version issues.

2. **SSH key agent.** Cron jobs do not inherit your shell environment. If your Git remotes use SSH, the cron job may fail silently because `ssh-agent` is not available. Add `eval "$(ssh-agent -s)"` to the script or use macOS Keychain integration.
3. **Detached HEAD states.** Repositories in a detached HEAD state (common after checking out a tag or specific commit) will cause `get_current_branch` to return a short hash instead of a branch name. The script handles this gracefully, but the push may fail if no matching remote branch exists.
4. **Large binary files.** Git is not designed for large binaries. If your `~/prj` directory contains datasets larger than 50MB, consider adding them to `.gitignore` and relying on Time Machine and cloud sync for those files instead.
5. **Race conditions.** If you are actively editing files while the script runs, it is possible for `git add -A` to stage a file that is then modified before the commit. The script checks for empty commits, but rapid file changes can occasionally cause unexpected behaviour.

5.7 Scheduling with Cron

The final step is to make the script run automatically. A cron job set to 15-minute intervals provides a good balance between backup frequency and system resource usage.

5.7.1 Open the Crontab Editor

```
crontab -e
```

5.7.2 Add the Cron Entry

```
*/15 * * * * /Users/$(whoami)/scripts/backup-research.sh
```

5.7.3 Save and Exit

- **nano:** `Ctrl + X`, then `Y`, then `Enter`
- **vim:** `Esc`, then `:wq`, then `Enter`

5.7.4 Verify and Test

```
crontab -l
```

Wait 15 minutes, then confirm execution by inspecting the log:

```
tail -20 ~/Library/Logs/research_backup.log
```

Figure 3: The reassurance of seeing a clean backup log after a long day of research work.

6 What Did We Learn?

6.1 Lessons Learnt

Conceptual Understanding:

- A single backup tier is never sufficient for active research. Each tier covers a different failure mode, and the combination provides substantially more protection than any tier alone.
- Automated Git commits at regular intervals transform version control from a deliberate act into a continuous safety net, capturing work that would otherwise be lost to forgetfulness.
- User-association filtering is essential when a research directory contains repositories from multiple collaborators – pushing to someone else's remote is a serious workflow disruption.
- Time Machine complements Git by capturing system state, application settings, and binary files that do not belong in version control.

Technical Skills:

- Bash `find` with `-print0` and `read -d ''` safely handles directory names containing spaces and special characters, which is common in research project naming.

- Log rotation using file size checks (`stat -f%z`) prevents unbounded log growth in long-running automated scripts.
- Colour-coded terminal output via ANSI escape codes significantly improves the readability of verbose diagnostic output during debugging.
- The `--set-upstream` flag on `git push` allows the script to handle newly created branches without manual intervention.

Gotchas and Pitfalls:

- Cron does not source `.bashrc` or `.zshrc`, so environment variables, SSH keys, and PATH modifications are not available unless explicitly set within the script or the crontab.
- The macOS default Bash (3.2) does not support process substitution with `< <(...)` reliably – always use Homebrew Bash 4+ for scripts that depend on modern features.
- `git status --porcelain` can produce false positives in directories with `.gitignore` conflicts, so the script verifies that staged changes actually exist before committing.
- Repositories with no commits (freshly initialised) will fail on branch detection – the script logs an error and moves on rather than halting the entire backup run.

6.2 Limitations

- The script only pushes to the `origin` remote. If a repository uses multiple remotes (e.g., `upstream` and `origin`), only one receives the backup.
- Auto-generated commit messages ("Auto-backup: timestamp") lack descriptive content. This works for backup purposes but pollutes the Git history for active development branches.
- The user-association check relies on string matching against "rgt47" in remote URLs and Git configuration. This approach is brittle and would break if the username changes.
- Time Machine requires the USB drive to be physically connected. When travelling without the drive, this tier is inactive.
- Cloud synchronisation introduces a dependency on third-party services (Google, Dropbox) and does not provide cryptographic integrity guarantees comparable to Git's SHA-based object store.
- The cron interval of 15 minutes means up to 14 minutes of work could be lost between backup cycles in a worst-case scenario.

6.3 Opportunities for Improvement

1. Replace fixed commit messages with a brief summary of changed file names, providing more informative Git history.
2. Add Slack or email notifications when the error count exceeds a configurable threshold.
3. Implement a `--dry-run` flag that reports what would be committed and pushed without actually performing the operations.
4. Add support for multiple remotes, pushing to both `origin` and a secondary backup remote.
5. Migrate from cron to `launchd` for better macOS integration, including wake-from-sleep triggers and retry logic.
6. Add a configuration file (YAML or TOML) to replace hard-coded paths and thresholds, making the script portable across machines.

7 Wrapping Up

Building this three-tier backup system took an afternoon of focused configuration, and it has run reliably for months with minimal maintenance. The core insight is simple: no single backup mechanism covers every failure mode, so layering three complementary approaches creates genuine resilience.

What I learned personally is that automation changes the psychology of backups. Before this system, I worried about remembering to commit and push. Now I trust the process and focus on the research itself. The 15-minute cron cycle means that even my most absent-minded days result in committed, pushed, backed-up work.

For anyone building a similar system, my advice is to start with the minimal script, verify it works across all your repositories, and then layer on the logging and filtering features as needed. The full script may look imposing, but every feature was added to solve a specific problem I encountered in practice.

Main takeaways:

- Three independent backup tiers (Git + cloud + Time Machine) cover the full spectrum of failure modes from bit rot to hardware loss.
- The minimal script is 7 lines; the production script is around 370 lines – the difference is entirely error handling, logging, and filtering.
- Cron scheduling at 15-minute intervals provides a practical balance between protection and system resource usage.
- User-association filtering prevents the script from pushing to collaborator repositories, which is essential in shared research environments.

8 See Also

Related posts:

- Configure the Command Line for Data Science Development – the terminal and shell setup that complements this backup system.
- Creating a GitHub Dotfiles Repository for Configuration Management – organising and versioning your configuration files.

Key resources:

- Pro Git Book (free) – authoritative reference for Git concepts
- Apple Time Machine Documentation – official setup and troubleshooting guide
- Crontab Guru – interactive cron schedule expression editor
- Homebrew – macOS package manager for installing Bash 5 and other tools

9 Reproducibility

This post describes a Bash-based backup system rather than an R analysis pipeline. The complete script can be reproduced by copying the code segments above into a single file.

To assemble and test the full script:

```
nano ~/scripts/backup-research.sh
chmod +x ~/scripts/backup-research.sh
~/scripts/backup-research.sh --verbose
crontab -e
```

Project files:

- `analysis/report/index.qmd` – this blog post
- `analysis/media/images/` – hero and ambiance images

Environment:

- macOS 14+ (Sonoma)
- Bash 5.2 (via Homebrew)
- Git 2.43+

10 Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.