Archiving 400 GitHub Repos Locally

A three-phase approach to backing up, verifying, and safely deleting private GitHub repositories

Ronald 'Ryy' G. Thomas

2025-12-02

Table of contents

Figure 1: Git logo: the version control system at the foundation of every repository in this guide

Version control is the starting point; archiving is the next step.

1 Introduction

Relying on GitHub as the sole backup for hundreds of private repositories presents a significant risk. If an account were compromised, suspended, or if GitHub changed its pricing model, years of work could be lost. This concern becomes acute when counting 400+ private repositories on a single platform with no local copies.

The challenge is not backing up a single repository (a simple `git clone` handles that). The difficulty lies in doing it comprehensively for hundreds of repos while also preserving the metadata that lives

only on GitHub: issue threads, pull request discussions, release binaries, labels, milestones, and wiki pages. A plain `git clone` misses all of that.

This post documents a bash script tested against 400+ repositories. It follows a strict three-phase process: backup everything, verify every backup, then selectively delete only after typed confirmation.

More formally, this post documents the off-site mirror leaf of the Backup layer (Layer 12) of the Workflow Construct described in post 52. Post 52 positions GitHub as the canonical source-of-truth tier of the construct's two-tier-or-more backup principle; this post addresses the symmetric concern of mirroring GitHub itself, so that the source-of-truth tier survives the loss of the GitHub account or platform. The script is the operative artefact for the GitHub-mirror leaf of the backup family.

1.1 Motivations

The following considerations motivated this approach:

- Having 400 private repos on GitHub with no local backups creates a single point of failure for years of work.
- A regular `git clone` captures code history but misses issues, PRs, releases, and wiki content that can be more valuable than the code itself.
- A dry-run mode is essential for previewing every action before committing to anything destructive.
- Manual archiving of even a dozen repos is tedious and error-prone; automation is the only practical path.
- Selective preservation allows active or shared repos to remain on GitHub while dormant ones are archived and removed.
- Building the script provides an opportunity to understand `gh` CLI features that may be unfamiliar.

1.2 Objectives

1. Build a script that creates a full mirror, portable bundle, wiki clone, and metadata export for every private repository on a GitHub account.
2. Implement a verification phase that checks every bundle with `git bundle verify` before any deletion is permitted.
3. Add a selective preservation mechanism so active or shared repos remain on GitHub while dormant ones are archived and removed.
4. Include a dry-run mode that previews every action without making changes, so the script can be reviewed before real execution.

Corrections and alternative approaches are welcome.

Figure 2: A workspace ready for a focused archival session.

2 Prerequisites and Setup

Before running the script, you need three tools installed and authenticated:

```
gh auth status

git --version

# Check available disk space (400 repos can
# require 20-40 GB depending on history)
df -h ~
```

Background. This guide assumes familiarity with basic command-line usage and git concepts (cloning, remotes, branches). You do not need to be a bash expert; the script handles the complexity.

You also need a GitHub personal access token with `repo` and `delete_repo` scopes if you plan to use the deletion phase. The `gh auth login` flow can configure this interactively.

3 What is a GitHub Archive?

A GitHub archive, in this context, is a local copy of everything GitHub stores for a repository: not just the code and commit history, but also the metadata that lives only on GitHub's servers (issue threads, pull request discussions, release binaries, labels, milestones, and wiki pages).

A regular `git clone` captures the commit graph but misses all of that surrounding context. A **mirror clone** (`git clone --mirror`) captures every ref, including remote-tracking branches and tags. A **git bundle** packages that mirror into a single portable file. And the **GitHub API exports** capture the metadata that git itself does not track.

The script in this post combines all three approaches: mirror clone for completeness, bundle for portability, and API exports for metadata. The result is a self-contained backup directory per repository that can survive even if GitHub itself becomes unavailable.

4 Getting Started: The Three-Phase Approach

The archive script follows a strict three-phase process. Understanding this structure makes the full script easier to follow.

Phase 1: Backup Everything. For each repository, the script creates:

- A full git mirror with every branch and tag
- A portable bundle file for easy transfer
- Wiki content (if the repo has one)
- Metadata exports via the GitHub API (issues, PRs, releases, labels, milestones, workflows)
- Downloaded release assets (binaries, artifacts)

Phase 2: Verify Backups. Before any deletion, the script runs `git bundle verify` on every bundle. If any verification fails, the entire deletion phase is aborted. This step is essential; without it, the deletion phase cannot be trusted.

Phase 3: Selective Deletion. Only repos that are not in the keep-list get deleted, and only after an explicit typed confirmation (`DELETE`). Repos in `KEEP_ON_GITHUB` are still backed up but are not removed from GitHub.

5 The Complete Working Script

Here is the full production-ready script, tested against 400+ repositories. Save it as `github-archive.sh` and make it executable with `chmod +x github-archive.sh`.

5.1 Script Header and Configuration

The first section sets up variables, the keep-list, and the command-line argument parser:

```bash
#!/bin/bash

# GitHub Archive Script
# Archives all private repos including metadata,
# then optionally deletes from GitHub

set -e

OWNER="your-username"
BACKUP_DIR="$HOME/github-archive"
DATE=$(date +%Y%m%d)
LOG_FILE="$BACKUP_DIR/archive_$DATE.log"
DRY_RUN=false

# Repos to keep on GitHub (edit this list)
KEEP_ON_GITHUB=(
    "important-project"
    "active-work"
    "shared-with-team"
)

# Parse command line arguments
usage() {
    echo "Usage: $0 [OPTIONS]"
    echo ""
    echo "Options:"
    echo "  -n, --dry-run     Show what would happen"
    echo "  -o, --owner NAME  GitHub username/org"
    echo "  -d, --dir PATH    Backup directory"
    echo "  -h, --help        Show this help message"
    exit 0
}

while [[ $# -gt 0 ]]; do
    case $1 in
        -n|--dry-run)
            DRY_RUN=true
            shift
            ;;
        -o|--owner)
            OWNER="$2"
            shift 2
            ;;
        -d|--dir)
            BACKUP_DIR="$2"
            shift 2
```

```
            ;;
        -h|--help)
            usage
            ;;
        *)
            echo "Unknown option: $1"
            usage
            ;;
    esac
done

mkdir -p "$BACKUP_DIR"
```

The `KEEP_ON_GITHUB` array is the most important configuration. Reviewing the repository list before running the script ensures the right ones are included.

5.2 Utility Functions

These helper functions handle logging, keep-list checking, and the per-repo backup logic:

```
log() {
    local prefix=""
    if [ "$DRY_RUN" = true ]; then
        prefix="[DRY-RUN] "
    fi
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] \
${prefix}$1" | tee -a "$LOG_FILE"
}

is_kept() {
    local repo=$1
    for kept in "${KEEP_ON_GITHUB[@]}"; do
        if [ "$repo" = "$kept" ]; then
            return 0
        fi
    done
    return 1
}
```

The `log()` function writes to both the terminal and a timestamped log file. This dual output proves invaluable when reviewing long runs.

5.3 Phase 1: The Backup Function

This is the core of the script. For each repo, it creates a mirror clone, generates a bundle, checks for a wiki, and exports all metadata via the GitHub API:

```
backup_repo() {
    local repo=$1
    local repo_dir="$BACKUP_DIR/$repo"

    log "=== Backing up $repo ==="

    if [ "$DRY_RUN" = true ]; then
        log "Would create: $repo_dir"
        log "Would clone: $OWNER/$repo"
        log "Would export: issues, PRs, releases"
        return
    fi

    mkdir -p "$repo_dir"
    cd "$repo_dir"

    # Clone with mirror (all branches, tags, refs)
    if [ ! -d "repo.git" ]; then
        log "Cloning repository..."
        gh repo clone "$OWNER/$repo" \
            repo.git -- --mirror
    else
        log "Updating existing clone..."
        cd repo.git && git fetch --all && cd ..
    fi

    # Create portable bundle
    log "Creating git bundle..."
    cd repo.git
    git bundle create ../repo.bundle --all
    cd ..

    # Clone wiki if it exists
    log "Checking for wiki..."
    if gh api "repos/$OWNER/$repo" \
        --jq '.has_wiki' 2>/dev/null \
        | grep -q true; then
        git clone \
            "https://github.com/$OWNER/$repo.wiki.git" \
            wiki.git 2>/dev/null \
            || log "No wiki content"
    fi

    # Export all metadata via GitHub API
    log "Exporting metadata..."
    gh api "repos/$OWNER/$repo" \
```

```
        > repo-info.json 2>/dev/null || true
    gh api "repos/$OWNER/$repo/issues?state=all" \
        --paginate > issues.json 2>/dev/null || true
    gh api "repos/$OWNER/$repo/pulls?state=all" \
        --paginate \
        > pull-requests.json 2>/dev/null || true
    gh api "repos/$OWNER/$repo/releases" \
        --paginate > releases.json 2>/dev/null || true
    gh api "repos/$OWNER/$repo/labels" \
        --paginate > labels.json 2>/dev/null || true
    gh api "repos/$OWNER/$repo/milestones?state=all" \
        --paginate \
        > milestones.json 2>/dev/null || true

    # Download release assets
    if [ -s releases.json ] \
        && [ "$(cat releases.json)" != "[]" ]; then
        log "Downloading release assets..."
        mkdir -p release-assets
        gh release list -R "$OWNER/$repo" \
            --limit 100 2>/dev/null \
            | while read -r tag rest; do
            gh release download "$tag" \
                -R "$OWNER/$repo" \
                -D "release-assets/$tag" \
                2>/dev/null || true
        done
    fi

    log "Completed backup of $repo"
    cd "$BACKUP_DIR"
}
```

The `--paginate` flag on `gh api` calls is essential. Without it, GitHub's API returns only the first 30 items per endpoint, which means issues and PRs on larger repos are silently lost.

5.4 Phase 2: The Verification Function

This function uses `git bundle verify` to confirm that every backup is valid before any deletion:

```
verify_backup() {
    local repo=$1
    local repo_dir="$BACKUP_DIR/$repo"

    log "Verifying backup of $repo..."
```

```
    if [ "$DRY_RUN" = true ]; then
        log "Would verify: $repo_dir/repo.bundle"
        return 0
    fi

    if [ -f "$repo_dir/repo.bundle" ]; then
        cd "$repo_dir"
        if git bundle verify repo.bundle \
            > /dev/null 2>&1; then
            log "PASS: Bundle verified"
            return 0
        else
            log "FAIL: Bundle verification failed!"
            return 1
        fi
    else
        log "FAIL: Bundle not found!"
        return 1
    fi
}
```

Failures may not be expected, but partial clones due to network interruptions can occur. The verification step catches them.

5.5 Phase 3: The Deletion Function

Deletion is the simplest function, but it is guarded by the verification phase and the typed confirmation:

```
delete_repo() {
    local repo=$1

    if [ "$DRY_RUN" = true ]; then
        log "Would delete: $OWNER/$repo"
        return
    fi

    log "Deleting $repo from GitHub..."
    gh repo delete "$OWNER/$repo" --yes
    log "Deleted $repo"
}
```

5.6 Main Execution Flow

The main block ties everything together. It fetches the repo list, categorises repos into keep vs delete, runs the three phases in order, and requires typed confirmation before any deletion:

```
# Main execution
if [ "$DRY_RUN" = true ]; then
    echo "======================================="
    echo " DRY-RUN MODE - No changes will be made"
    echo "======================================="
    echo ""
fi

log "Starting GitHub archive process"
log "Owner: $OWNER"
log "Backup directory: $BACKUP_DIR"

# Get all private repos
log "Fetching list of private repositories..."
repos=$(gh repo list "$OWNER" \
    --limit 500 --private \
    --json name -q '.[].name')
repo_count=$(echo "$repos" | wc -l | tr -d ' ')
log "Found $repo_count private repositories"

# Categorize repos
repos_to_delete=""
repos_to_keep=""
delete_count=0
keep_count=0

for repo in $repos; do
    if is_kept "$repo"; then
        repos_to_keep="$repos_to_keep $repo"
        ((keep_count++)) || true
    else
        repos_to_delete="$repos_to_delete $repo"
        ((delete_count++)) || true
    fi
done

log "Repos to archive and DELETE: $delete_count"
log "Repos to archive and KEEP: $keep_count"

echo ""
echo "=== REPO CATEGORIZATION ==="
echo ""
echo "Will be DELETED after backup ($delete_count):"
for repo in $repos_to_delete; do
    echo "  x $repo"
done
echo ""
```

```
echo "Will be KEPT on GitHub ($keep_count):"
for repo in $repos_to_keep; do
    echo "  + $repo"
done
echo ""

if [ "$DRY_RUN" = true ]; then
    echo "=== DRY-RUN: PHASE 1 (BACKUP) ==="
    for repo in $repos; do
        backup_repo "$repo"
    done

    echo ""
    echo "=== DRY-RUN: PHASE 2 (VERIFICATION) ==="
    for repo in $repos; do
        verify_backup "$repo"
    done

    echo ""
    echo "=== DRY-RUN: PHASE 3 (DELETION) ==="
    for repo in $repos_to_delete; do
        delete_repo "$repo"
    done

    echo ""
    echo "========================================"
    echo "   DRY-RUN COMPLETE"
    echo "========================================"
    echo ""
    echo "Run without --dry-run to execute."
    exit 0
fi

# Phase 1: Backup all repos
log "=== PHASE 1: BACKUP ==="
for repo in $repos; do
    backup_repo "$repo"
done

# Phase 2: Verify all backups
log "=== PHASE 2: VERIFICATION ==="
failed_repos=""
for repo in $repos; do
    if ! verify_backup "$repo"; then
        failed_repos="$failed_repos $repo"
    fi
done
```

```
if [ -n "$failed_repos" ]; then
    log "WARNING: Verification failed:$failed_repos"
    log "Aborting deletion phase"
    exit 1
fi

log "All backups verified successfully!"

# Phase 3: Delete (with typed confirmation)
log "=== PHASE 3: DELETION ==="
echo ""
echo "Backup complete and verified!"
echo ""
read -p "Delete $delete_count repos? \
(type 'DELETE'): " confirm

if [ "$confirm" = "DELETE" ]; then
    for repo in $repos_to_delete; do
        delete_repo "$repo"
    done
    log "Deleted $delete_count repositories"
else
    log "Deletion cancelled"
fi

log "Archive process complete"
```

6 Using the Script

6.1 Basic Usage

```
# Preview what would happen (no changes made)
./github-archive.sh --dry-run

# Run for real (backs up all, asks before deleting)
./github-archive.sh

# Specify custom owner or directory
./github-archive.sh \
    --owner myorg --dir /external/drive/backup
```

Running the dry-run first is not just useful; it is essential. The categorisation output can catch repos that were forgotten in the keep-list.

6.2 Configuring the Keep-List

Edit the `KEEP_ON_GITHUB` array to match your needs:

```
KEEP_ON_GITHUB=(
    "active-projects"      # Currently maintained
    "shared-with-team"     # Collaboration repos
    "client-work"          # Client projects
    "portfolio"            # Showcase projects
)
```

6.3 Dry-Run Example Output

```
========================================
 DRY-RUN MODE - No changes will be made
========================================

=== REPO CATEGORIZATION ===

Will be DELETED after backup (397):
  x old-project-1
  x old-project-2
  ...

Will be KEPT on GitHub (3):
  + important-project
  + active-work
  + shared-with-team
```

7 Backup Directory Structure

After running the script, your backup directory looks like this:

```
~/github-archive/
+-- important-project/      # KEPT on GitHub
|   +-- repo.git/           # Full repository
|   +-- repo.bundle         # Portable archive
|   +-- repo-info.json      # Metadata
|
+-- old-project-1/          # DELETED from GitHub
|   +-- repo.git/
|   +-- repo.bundle
|   +-- wiki.git/           # If repo has wiki
|   +-- repo-info.json      # Repository metadata
|   +-- issues.json         # All issues + comments
```

```
|   +-- pull-requests.json  # All PRs + comments
|   +-- releases.json       # All releases
|   +-- labels.json         # Issue labels
|   +-- milestones.json     # Milestones
|   +-- release-assets/     # Downloaded binaries
|
+-- archive_20251202.log    # Detailed log
```

7.1 What Gets Backed Up

Content	Format	Use Case
Git history	`repo.git/` + bundle	Full reproducibility
Wiki	`wiki.git/`	Documentation
Issues	`issues.json`	Discussion archive
Pull requests	`pull-requests.json`	Code review history
Releases	`releases.json`	Version history
Release assets	`release-assets/`	Binaries, artifacts
Metadata	`repo-info.json`	Repository config
Labels	`labels.json`	Issue classification
Milestones	`milestones.json`	Project tracking

8 Restoring from Backup

If you need to restore a repository later, there are two approaches:

```bash
# Option 1: Clone from the portable bundle
git clone \
    ~/github-archive/repo-name/repo.bundle \
    restored-repo

# Option 2: Push the mirror to a new GitHub repo
gh repo create new-repo-name --private
cd ~/github-archive/repo-name/repo.git
git push --mirror \
    git@github.com:you/new-repo-name.git
```

The bundle approach is more convenient for quick local inspection, while the mirror push is better for actually recreating a repo on GitHub.

Figure 3: Organised workspace: the goal after archiving 400 repos

After the archive, what remains on GitHub is clean and intentional.

9 Daily Workflow

Once the script is tested, the typical usage pattern is:

Command	Action
`bash archive.sh --dry-run`	Preview what would be backed up / deleted
`bash archive.sh`	Run full backup + verification
`bash archive.sh --owner ORG`	Archive a different owner's repos
`git bundle verify repo.bundle`	Spot-check a specific backup
`gh repo list --json name`	Confirm which repos remain on GitHub

Run the archive quarterly or before any GitHub plan change.

10 Checking Our Work

Before trusting any backup, these checks should be run:

```
# Verify a specific bundle
cd ~/github-archive/repo-name
git bundle verify repo.bundle

# Check that metadata files are non-empty
ls -la *.json

# Confirm the bundle contains all branches
git bundle list-heads repo.bundle

# Spot-check an issue export
python3 -m json.tool issues.json | head -50
```

10.1 Things to Watch Out For

1. **The `--limit 500` cap.** The `gh repo list` command defaults to 30 results. The script sets it to 500, but if you have more repos than that, you need to increase it or paginate manually.
2. **Disk space surprises.** An initial estimate of 20 GB for 400 repos may prove insufficient; closer to 35 GB may be needed if several repos have large binary assets in their release history. Check with `df -h` first.
3. **Network interruptions.** If a clone fails midway, the `repo.git` directory exists but is incomplete. The verification phase catches this, but you need to delete the partial clone and rerun.
4. **API rate limits.** GitHub's API allows 5,000 requests per hour for authenticated users. With 400 repos and 6 API calls each, that is 2,400 requests, within the limit but close. If the hit the limit, the script pauses and retries are not automatic.
5. **Token permissions.** The `delete_repo` scope is required only if using Phase 3. For backup-only runs, the standard `repo` scope suffices. Forgetting the delete scope requires re-authentication.

11 Uninstall / Rollback

To restore a deleted repo from a local archive:

```
# Restore from the portable bundle
cd ~/github-archive/repo-name
git clone repo.bundle restored-repo

# Or push the backup to a new GitHub repo
gh repo create owner/repo-name --private
cd restored-repo
git remote add origin git@github.com:owner/repo-name.git
```

```
git push --mirror origin
```

To remove the archive script and local backups:

```
rm -i archive.sh
rm -ri ~/github-archive/   # confirm before removing backups
```

{.img-fluid}

The best technical workflows, like the best libraries, are built on solid foundations.

12 What Did We Learn?

12.1 Lessons Learnt

Conceptual Understanding:

- A plain `git clone` misses metadata that can be more valuable than the code itself: issue discussions, PR review threads, and release notes.
- Git bundles are surprisingly versatile; a single file contains the entire repository history and can be cloned directly.
- The three-phase approach (backup, verify, delete) is a general pattern applicable to any destructive batch operation, not just GitHub archiving.

- Dry-run modes are not optional for scripts that delete things. Data loss is likely without them.

Technical Skills:

- The `gh` CLI is more powerful than often expected. Combining `gh repo list`, `gh api --paginate`, and `gh release download` covers nearly every GitHub operation without touching the web interface.
- `git bundle verify` is a built-in safety net that may be unfamiliar. It confirms the bundle is a valid, complete repository.
- Bash arrays (`KEEP_ON_GITHUB`) and functions make scripts maintainable. Without them, the 300-line script would be unreadable.
- Logging to both stdout and a file with `tee -a` is a simple pattern that saves hours of debugging.

Gotchas and Pitfalls:

- GitHub wikis are technically separate git repos. They must be cloned independently, and they silently fail if the wiki was enabled but never populated.
- The `--paginate` flag on `gh api` is critical. Forgetting it silently loses issues beyond the first 30 on larger repos.
- `set -e` causes the script to exit on the first error, which is good for safety but means `|| true` is needed on commands that are expected to fail (like cloning an empty wiki).
- Release asset downloads can be slow and large. Consider excluding them with a flag if only the code and metadata are needed.

12.2 Limitations

- The script does not handle GitHub Actions workflow run history or Codespaces configurations, which are not available through the standard API.
- Repository settings (branch protection rules, webhook configurations, deploy keys) are not exported. Recreating those requires manual setup or additional API calls.
- Git LFS objects are not included in the mirror clone by default. Repos using LFS need `git lfs fetch --all` added to the backup function.
- The script processes repos sequentially. For 400+ repos, this can take several hours. Parallelisation with `xargs -P` or GNU `parallel` would speed things up but adds complexity.
- Forked repos may have upstream references that break after deletion. The bundle preserves the fork's commits but not the upstream connection.
- Private repo collaborators and team permissions are not captured. Re-adding collaborators after restoration is a manual step.

12.3 Opportunities for Improvement

1. **Add Git LFS support.** Insert `git lfs fetch --all` into the backup function for repos that use large file storage.
2. **Parallelise the backup phase.** Use `xargs -P 4` or GNU `parallel` to clone multiple repos concurrently, reducing total runtime from hours to minutes.

3. **Export repository settings.** Add API calls for branch protection rules, webhooks, and deploy keys so that restoration is truly complete.
4. **Add incremental backup support.** Track which repos have already been backed up and only process new or changed ones on subsequent runs.
5. **Create a restore script.** Build a companion `github-restore.sh` that reads a backup directory and recreates repos on GitHub, including metadata re-import.
6. **Add compression.** After verification, compress each repo directory with `tar -czf` to reduce storage requirements by roughly 50-70%.

13 Wrapping Up

What starts as a vague worry about losing 400 repos can become a concrete, repeatable process run quarterly. The three-phase approach (backup everything, verify everything, then delete selectively) provides the confidence to actually clean up a GitHub account instead of just contemplating it.

The most valuable lesson is not about git or bash but about discipline: having a backup is not the same as having a verified backup. The verification phase catches incomplete clones that would otherwise go unnoticed. That alone justifies the effort of writing the script.

For those with dozens or hundreds of repos on GitHub, starting with a dry-run is recommended. Deletion is not required; even just having verified local mirrors is a meaningful improvement to data safety.

Main takeaways:

- The three-phase pattern (backup, verify, delete) prevents data loss from incomplete archives.
- Metadata exports (issues, PRs, releases) capture context that `git clone` alone misses.
- Dry-run mode is essential for any script that performs destructive operations.
- A 400-repo archive takes roughly 2-4 hours and 20-40 GB of disk space depending on repo sizes.

14 See Also

Related posts on this blog:

- Setting Up Git for Data Science Development
- Creating a GitHub Dotfiles Repository

Key resources:

- GitHub CLI Documentation
- Git Bundle Documentation
- GitHub REST API: Repositories
- Git LFS Documentation
- GitHub: Managing Repository Backups

15 Reproducibility

Environment requirements:

- macOS or Linux (tested on macOS 14 Sonoma and Ubuntu 22.04)
- `git` 2.39 or later
- `gh` CLI 2.40 or later
- Bash 5.x (macOS ships 3.2; install via Homebrew with `brew install bash`)

Version checks:

```
git --version
gh --version
bash --version
```

Script files:

File	Description
`github-archive.sh`	Main archive script
`archive_YYYYMMDD.log`	Execution log per run

This post does not use R or Docker. The entire workflow is pure bash with the GitHub CLI.

16 Let's Connect

Have questions, suggestions, or spot an error? Let me know.

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** Contact form

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.