Setting Up a Comprehensive Research Backup System on macOS

Research Backup Guide

Invalid Date

Table of contents

1 Introduction

Managing 300+ Git repositories across 20GB of research data requires a robust, automated backup strategy. This guide walks through setting up a three-tier backup system that provides Git-level versioning, real-time cloud sync, and comprehensive system backups.

1.1 Backup Strategy Overview

Our approach uses three complementary layers:

1. **Automated Git commits and pushes** (every 15 minutes)
2. **Cloud synchronization** (real-time via Google Drive/Dropbox)
3. **Time Machine backups** (hourly system-wide backups)

This ensures your research is protected against hardware failure, accidental deletion, Git corruption, and provides easy access across devices.

2 Setting Up Time Machine

Time Machine provides system-wide backup protection and serves as your safety net for everything beyond Git repositories.

2.1 Initial Time Machine Setup

2.1.1 Step 1: Connect Your USB Drive

1. Connect your 1TB USB drive to your MacBook
2. When prompted, **do not** use it for Time Machine yet - we'll configure this properly first

2.1.2 Step 2: Format the Drive (if needed)

1. Open **Disk Utility** (Applications > Utilities > Disk Utility)
2. Select your USB drive from the sidebar
3. Click **Erase**
4. Choose format: **Mac OS Extended (Journaled)** or **APFS** (recommended for newer Macs)
5. Name it something like "Research Backup"
6. Click **Erase**

2.1.3 Step 3: 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

2.1.4 Step 4: Customize Time Machine Settings

1. Click **Options** in Time Machine preferences
2. Add any folders you want to exclude (like Downloads, Trash, etc.)
3. **Important**: Do NOT exclude `~/prj` - we want this backed up
4. Ensure "Back up while on battery power" is enabled if desired

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

3 Automated Git Backup Script

This script scans all Git repositories in `~/prj` every 15 minutes, commits changes, and pushes to GitHub.

3.1 Minimal version of backup script:

```
#!/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
```

3.2 Create the full Script File

Add these features:

3.2.1 Command Line Interface Features

- **Verbose mode flag** (`-v|--verbose`)
- **Help flag** (`-h|--help`)
- **Command line argument parsing** (while loop with case statements)
- **Usage instructions and help text**

3.2.2 Logging and Output Features

- **Log file creation** (`~/Library/Logs/research_backup.log`)
- **Log rotation** (when file exceeds 10MB)
- **Timestamped log entries**
- **Color-coded console output** (red/yellow/green/blue messages)
- **Log message function** with level-based formatting
- **Detailed progress reporting** ("Processing repository X")
- **Final summary statistics**
- **Verbose console output option**

3.2.3 Error Handling and Validation

- **Directory existence checks** (research directory validation)
- **Git repository validation** (checking if .git is actually a valid repo)
- **Remote repository checks** (verifying origin remote exists)
- **Branch existence validation** (checking if branch exists on remote)
- **File staging error handling** (git add -A failure detection)

- **Commit failure detection and reporting**
- **Push failure detection with specific error messages**
- **Network/authentication error distinction**

3.2.4 Repository Filtering

- **User association filtering** (only "rgt47" repositories)
- **Archive directory exclusion** (skip directories with "archive")
- **Backup directory exclusion** (skip directories with "backup")
- **Case-insensitive name matching**
- **Path-based exclusion checks**

3.2.5 Advanced Git Features

- **Current branch detection** (get_current_branch function)
- **Upstream branch creation** (–set-upstream for new branches)
- **Change analysis** (counting untracked/modified/added/deleted files)
- **Branch existence verification on remote**
- **Graceful handling of detached HEAD states**
- **Smart branch pushing** (handles both main and master)

3.2.6 Counting and Statistics

- **Repository counters** (repo_count, backup_count, error_count, etc.)
- **Excluded repository tracking**
- **Skipped repository tracking**
- **Warning count tracking**
- **Detailed final summary with all statistics**

3.2.7 Safety Features

- **Working directory validation** (cd error handling)
- **Git status checks before operations**
- **Clean repository detection** (skip repos with no changes)
- **Race condition handling** (checking for empty commits)

3.2.8 Documentation and Maintenance

- **Extensive inline comments**
- **Function documentation**
- **Usage examples**
- **Error message explanations**
- **Troubleshooting information**

3.2.9 Configuration Features

- **Configurable research directory path**
- **Configurable log file location**
- **Configurable log size limits**
- **Environment variable handling**

4 Full version of backup script:

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

# Research Git Backup Script
# Automatically commits and pushes changes in all Git repositories
# Usage: ./backup_research.sh [-v|--verbose] [-h|--help]

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

# Parse command line arguments
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 to console"
            echo "  -h, --help       Show this help message"
```

```
            exit 0
            ;;
        *)
            echo "Unknown option: $1"
            echo "Use -h or --help for usage information"
            exit 1
            ;;
    esac
done

# Create log directory if it doesn't exist
mkdir -p "$(dirname "$LOG_FILE")"

# Rotate log if it gets too large
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 (size exceeded ${MAX_LOG_SIZE} bytes)"
    fi
fi

# Function to log messages (always logs to file, optionally to console)
log_message() {
    local level="$1"
    local message="$2"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')
    local log_entry="$timestamp: [$level] $message"

    echo "$log_entry" >> "$LOG_FILE"

    if [[ "$VERBOSE" == true ]]; then
        case "$level" in
            ERROR)
                echo -e "\033[31m$log_entry\033[0m" # Red
                ;;
            WARNING)
                echo -e "\033[33m$log_entry\033[0m" # Yellow
                ;;
            SUCCESS)
                echo -e "\033[32m$log_entry\033[0m" # Green
                ;;
            INFO)
```

```
                echo -e "\033[34m$log_entry\033[0m" # Blue
                ;;
            *)
                echo "$log_entry"
                ;;
        esac
    fi
}

# Function to check if repository has remote configured
check_remote() {
    local repo_dir="$1"
    cd "$repo_dir" || return 1

    local remote_url=$(git remote get-url origin 2>/dev/null)
    if [[ -z "$remote_url" ]]; then
        return 1
    fi
    return 0
}

# Function to check if repository is associated with user "rgt47"
check_user_association() {
    local repo_dir="$1"
    cd "$repo_dir" || return 1

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

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

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

    # Check global git config if local config doesn't have user info
```

```
    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
}

# Function to check if directory should be excluded based on name
should_exclude_directory() {
    local repo_name="$1"
    local repo_path="$2"

    # Convert to lowercase for case-insensitive matching
    local lower_name=$(echo "$repo_name" | tr '[:upper:]' '[:lower:]')
    local lower_path=$(echo "$repo_path" | tr '[:upper:]' '[:lower:]')

    # Check if directory name contains "archive" or "backup"
    if [[ "$lower_name" == *"archive"* ]] ||
       [[ "$lower_name" == *"backup"* ]]; then
        return 0  # Should exclude
    fi

    # Check if any part of the path contains "archive" or "backup"
    if [[ "$lower_path" == *"archive"* ]] ||
       [[ "$lower_path" == *"backup"* ]]; then
        return 0  # Should exclude
    fi

    return 1  # Should not exclude
}

# Function to get current branch name
get_current_branch() {
    git symbolic-ref --short HEAD 2>/dev/null ||
```

```
    git rev-parse --short HEAD 2>/dev/null
}

# Function to check if branch exists on remote
branch_exists_on_remote() {
    local branch="$1"
    git ls-remote --heads origin "$branch" 2>/dev/null | grep -q "$branch"
}

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

# Check if research directory exists
if [[ ! -d "$RESEARCH_DIR" ]]; then
    log_message "ERROR" "Research directory $RESEARCH_DIR does not exist"
    exit 1
fi

log_message "INFO" "Scanning research directory: $RESEARCH_DIR"

# Counter for repositories processed
repo_count=0
backup_count=0
error_count=0
warning_count=0
skipped_count=0
excluded_count=0

# Find all .git directories and process them
while IFS= read -r -d '' git_dir; do
    repo_dir=$(dirname "$git_dir")
    repo_name=$(basename "$repo_dir")
    relative_path="${repo_dir#$RESEARCH_DIR}"

    # Check if directory should be excluded based on name
    if should_exclude_directory "$repo_name" "$relative_path"; then
        log_message "INFO" \
            "Excluding repository (contains 'archive' or 'backup'): \
$relative_path"
        ((excluded_count++))
        continue
    fi
```

```
    log_message "INFO" "Processing repository: $relative_path"

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

    ((repo_count++))

    # Check if it's actually a git repository
    if ! git rev-parse --git-dir >/dev/null 2>&1; then
        log_message "ERROR" \
            "Directory contains .git but is not a valid git repository: \
$relative_path"
        ((error_count++))
        continue
    fi

    # Check if repository is associated with user "rgt47"
    if ! check_user_association "$repo_dir"; then
        log_message "INFO" \
            "Skipping repository (not associated with user 'rgt47'): \
$relative_path"
        ((skipped_count++))
        continue
    fi

    log_message "INFO" \
        "Repository $relative_path is associated with user 'rgt47'"

    # Check if repository has a remote configured
    if ! check_remote "$repo_dir"; then
        log_message "WARNING" \
            "Repository has no remote configured, skipping: $relative_path"
        ((warning_count++))
        ((skipped_count++))
        continue
    fi

    # Get current branch
```

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

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

    # Check repository status
    git_status=$(git status --porcelain 2>/dev/null)

    if [[ -z "$git_status" ]]; then
        log_message "INFO" \
            "Repository $relative_path is clean (no changes to commit)"
        continue
    fi

    # Count changes
    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" \
        "Repository $relative_path has changes: $untracked untracked, \
$modified modified, $added added, $deleted deleted"

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

    log_message "INFO" \
        "Successfully staged all changes in: $relative_path"

    # Create commit with timestamp
```

```
    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" \
            "Successfully committed changes in: $relative_path"

        # Check if current branch exists on remote
        if ! branch_exists_on_remote "$current_branch"; then
            log_message "WARNING" \
                "Branch '$current_branch' does not exist on remote for: \
$relative_path"

            # Try to push with upstream setting
            if git push --set-upstream origin "$current_branch" \
               2>/dev/null; then
                log_message "SUCCESS" \
                    "Created and pushed new branch '$current_branch' to \
remote for: $relative_path"
                ((backup_count++))
            else
                log_message "ERROR" \
                    "Failed to create and push new branch '$current_branch' \
for: $relative_path"
                ((error_count++))
            fi
        else
            # Try to push to current branch
            if git push origin "$current_branch" 2>/dev/null; then
                log_message "SUCCESS" \
                    "Successfully pushed '$current_branch' to remote for: \
$relative_path"
                ((backup_count++))
            else
                log_message "ERROR" \
                    "Failed to push '$current_branch' to remote for: \
$relative_path (check network/auth)"
                ((error_count++))
            fi
        fi
    else
        # Check if commit failed due to no changes (race condition)
        if git diff --cached --quiet; then
```

```
            log_message "INFO" \
                "No changes to commit in: $relative_path \
(changes may have been reverted)"
        else
            log_message "ERROR" \
                "Failed to commit changes in: $relative_path"
            ((error_count++))
        fi
    fi

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

# Final summary
log_message "INFO" "Backup scan complete"
log_message "INFO" \
    "Summary: $repo_count repositories processed, $backup_count \
successfully 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 "Repositories excluded: $excluded_count \
(archive/backup in name)"
    echo "Repositories skipped: $skipped_count \
(not associated with rgt47)"
    echo "Repositories processed: $repo_count"
    echo "Successfully backed up: $backup_count"
    echo "Errors encountered: $error_count"
    echo "Warnings generated: $warning_count"
    echo ""
    echo "Log file location: $LOG_FILE"

    if [[ $error_count -gt 0 ]]; then
        echo ""
        echo "  There were errors during backup. Check the log file \
for details."
        exit 1
```

```
    elif [[ $warning_count -gt 0 ]]; then
        echo ""
        echo "  Backup completed with warnings. Check the log file \
for details."
    else
        echo ""
        echo "  Backup completed successfully!"
    fi
fi

exit 0
```

4.1 Setting Up the Cron Job

4.1.1 Step 1: Open the Crontab Editor

```
crontab -e
```

4.1.2 Step 2: Add the Cron Entry

Add this line to your crontab:

```
# Research backup - runs every 15 minutes
*/15 * * * * /Users/$(whoami)/scripts/backup-research.sh
```

4.1.3 Step 3: Save and Exit

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

4.1.4 Step 4: Verify the Cron Job

```
crontab -l
```

4.1.5 Step 5: Test the Setup

Wait 15 minutes, then check if it ran:

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