Developer Collaboration Workflow Sequence

RG Thomas

July 12, 2025

Based on my review of the user guide, here are the specific workflows for developer collaboration using vim as the IDE:

Streamlined Team Collaboration Workflow

Initial ZZCOLLAB Setup (One-time)

```
# 1. Clone and install zzcollab system
git clone https://github.com/[OWNER]/zzcollab.git
cd zzcollab
./install.sh                        # Install to ~/bin (default)

# 2. Verify installation
zzcollab --help                     # Test installation from anywhere
zzcollab --init --help              # Test team initialization mode
which zzcollab                      # Confirm system PATH setup
```

Developer 1 (Team Lead): Project Initialization

Developer 1 Checklist:

- ☐ Create new analysis project directory
- ☐ Customize Dockerfile.teamcore for team's R packages and tools
- ☐ Build and push shell core image to Docker Hub
- ☐ Build and push RStudio core image to Docker Hub

- ☐ Run zzcollab with dotfiles to create local development image
- ☐ Initialize private GitHub repository and push code
- ☐ Create first analysis script in scripts/ directory
- ☐ Write integration tests for analysis script
- ☐ Run tests to verify everything works
- ☐ Commit and push code + tests together

🚀 AUTOMATED APPROACH (Recommended) Choose between command-line or R interface:

Option A: Command-Line Interface

```
# Complete automated setup - replaces all manual Docker and git commands
zzcollab --init --team-name rgt47 --project-name research-study \
    --dotfiles ~/dotfiles

# OR with Dockerfile customization (two-step process):
# Step 1: Prepare project and Dockerfile for editing
zzcollab --init --team-name rgt47 --project-name research-study \
    --prepare-dockerfile
# Step 2: Edit research-study/Dockerfile.teamcore, then run:
zzcollab --init --team-name rgt47 --project-name research-study \
    --dotfiles ~/dotfiles
```

Option B: R Interface (R-Centric Workflow)

```
# From R console
library(zzcollab)

# Complete automated setup from within R
init_project(
  team_name = "rgt47",
  project_name = "research-study",
  dotfiles_path = "~/dotfiles"
)
```

Both approaches automatically:

- 📁 Creates project directory
- 🐳 Sets up customizable Dockerfile.teamcore
- 🔨 Builds shell and RStudio core images
- 🏷️ Tags and pushes images to Docker Hub
- 📦 Initializes zzcollab project with base image
- 🌐 Creates private GitHub repository
- 📝 Sets up initial commit with proper structure

📋 DETAILED STEPWISE BREAKDOWN The `zzcollab --init` command provides real-time console feedback matching this exact sequence:

```
🔍 [INFO] Validating preliminary requirements...
# Checks: docker, gh CLI, zzcollab availability and authentication
🐳 [SUCCESS] Docker Hub account 'TEAM' verified
🐙 [SUCCESS] GitHub account 'TEAM' verified
✅ [SUCCESS] All preliminary requirements validated
```

```
 [INFO] Configuration Summary:
#   Team Name: TEAM
#   Project Name: PROJECT
#   GitHub Account: TEAM
#   Dotfiles: [path or none]
#   Dockerfile: [template path]
# User prompt: "Proceed with team setup? [y/N]"

 [INFO] Starting automated team setup...

 [INFO] Step 1: Creating project directory...
mkdir PROJECT_NAME && cd PROJECT_NAME
 [SUCCESS] Created project directory: PROJECT_NAME

 [INFO] Step 2: Setting up team Dockerfile...
cp ../templates/Dockerfile.pluspackages ./Dockerfile.teamcore
 [SUCCESS] Copied Dockerfile template to Dockerfile.teamcore

 [INFO] Step 3: Building shell core image...
docker build -f Dockerfile.teamcore \
    --build-arg BASE_IMAGE=rocker/r-ver \
    --build-arg TEAM_NAME="TEAM" \
    --build-arg PROJECT_NAME="PROJECT" \
    -t "TEAM/PROJECTcore-shell:v1.0.0" .
docker tag "TEAM/PROJECTcore-shell:v1.0.0" \
    "TEAM/PROJECTcore-shell:latest"
 [SUCCESS] Built shell core image: TEAM/PROJECTcore-shell:v1.0.0

 [INFO] Step 4: Building RStudio core image...
docker build -f Dockerfile.teamcore \
    --build-arg BASE_IMAGE=rocker/rstudio \
    --build-arg TEAM_NAME="TEAM" \
    --build-arg PROJECT_NAME="PROJECT" \
    -t "TEAM/PROJECTcore-rstudio:v1.0.0" .
docker tag "TEAM/PROJECTcore-rstudio:v1.0.0" \
    "TEAM/PROJECTcore-rstudio:latest"
 [SUCCESS] Built RStudio core image: TEAM/PROJECTcore-rstudio:v1.0.0

 [INFO] Step 5: Pushing images to Docker Hub...
docker push "TEAM/PROJECTcore-shell:v1.0.0"
docker push "TEAM/PROJECTcore-shell:latest"
docker push "TEAM/PROJECTcore-rstudio:v1.0.0"
docker push "TEAM/PROJECTcore-rstudio:latest"
 [SUCCESS] Pushed all images to Docker Hub

 [INFO] Step 6: Initializing zzcollab project...
```

```
zzcollab --base-image "TEAM/PROJECTcore-shell" [--dotfiles PATH]
# Creates: R package structure, analysis/ directories, symbolic links,
#          Makefile, docker-compose.yml, GitHub Actions workflows
 [SUCCESS] Initialized zzcollab project with custom base image

 [INFO] Step 7: Initializing git repository...
git init
git add .
git commit -m " Initial research project setup..."
 [SUCCESS] Initialized git repository with initial commit

 [INFO] Step 8: Creating private GitHub repository...
gh repo create "TEAM/PROJECT" --private --source=. --remote=origin \
    --push
 [SUCCESS] Created private GitHub repository: TEAM/PROJECT

 [SUCCESS]  Team setup completed successfully!

 [INFO] What was created:
#    Project directory: PROJECT/
#    Docker images:
#       TEAM/PROJECTcore-shell:v1.0.0, TEAM/PROJECTcore-rstudio:v1.0.0
#    Private GitHub repo: https://github.com/TEAM/PROJECT
#    Complete zzcollab research compendium

 [INFO] Next steps:
#   1. cd PROJECT
#   2. make docker-zsh    # Start development environment
#   3. Start coding your analysis!

 [INFO] Team members can now join with:
#   git clone https://github.com/TEAM/PROJECT.git
#   cd PROJECT
#   zzcollab --team TEAM --project-name PROJECT --interface shell \
#       --dotfiles ~/dotfiles
#   make docker-zsh
```

Key Technical Details:

- **Preliminary checks** prevent partial failures by validating all dependencies upfront
- **Dual image builds** provide both shell and web-based development options

- **Argument passing** ensures team/project names are embedded in Docker images
- **Public image registry** enables team members to pull images

without authentication
- **Private code repository** protects unpublished research while sharing methodology
- **Atomic operations** with rollback capability if any step fails
- **Optimized module loading** eliminates initialization warnings by loading analysis module after directory structure creation

Error Handling:

- All commands use `set -euo pipefail` for strict error detection
- Pre-existing directories trigger user confirmation prompts
- Failed Docker builds halt execution before invalid images are pushed
- GitHub API failures are caught before local git operations
- Enhanced prerequisite validation includes Docker Hub account verification
- Comprehensive argument validation with clear error messages

🔧 MANUAL APPROACH (For customization or learning) If you need custom control or want to understand the underlying process, you can run the individual commands:

```
# 1. Create new analysis project
mkdir research-project
cd research-project

# 2. Customize team core image for your project
TEAM_NAME="rgt47" # the account on dockerhub for hosting the core images
PROJECT_NAME=$(basename $(pwd))    # Get current directory name
# Copy and customize Dockerfile.pluspackages for your team's needs
cp templates/Dockerfile.pluspackages ./Dockerfile.teamcore

# Edit Dockerfile.teamcore to add your team's specific R packages and tools:
vim Dockerfile.teamcore
# Key customizations:
#
# 1. Ensure first lines support base image argument:
#    ARG BASE_IMAGE=rocker/r-ver
#    ARG R_VERSION=latest
#    FROM ${BASE_IMAGE}:${R_VERSION}
# 2. Add domain-specific R packages (e.g., 'brms', 'targets', 'cmdstanr')
# 3. Include specialized system tools (e.g., JAGS, Stan, ImageMagick)
# 4. Set team-specific R options and configurations
# 5. Add database drivers or cloud SDKs

# 3. Build TWO team core images for different interfaces
# Shell-optimized core (rocker/r-ver base - lightweight, fast startup)
```

```
docker build -f Dockerfile.teamcore \
    --build-arg BASE_IMAGE=rocker/r-ver \
    --build-arg TEAM_NAME="$TEAM_NAME" \
    --build-arg PROJECT_NAME="$PROJECT_NAME" \
    -t ${TEAM_NAME}/${PROJECT_NAME}core-shell:v1.0.0 .
docker tag ${TEAM_NAME}/${PROJECT_NAME}core-shell:v1.0.0 \
    ${TEAM_NAME}/${PROJECT_NAME}core-shell:latest

# RStudio-optimized core (rocker/rstudio base - includes RStudio Server)
docker build -f Dockerfile.teamcore \
    --build-arg BASE_IMAGE=rocker/rstudio \
    --build-arg TEAM_NAME="$TEAM_NAME" \
    --build-arg PROJECT_NAME="$PROJECT_NAME" \
    -t ${TEAM_NAME}/${PROJECT_NAME}core-rstudio:v1.0.0 .
docker tag ${TEAM_NAME}/${PROJECT_NAME}core-rstudio:v1.0.0 \
    ${TEAM_NAME}/${PROJECT_NAME}core-rstudio:latest

# 4. Push both team core images to Docker Hub (PUBLIC for reproducibility)
docker login                          # Login to Docker Hub
docker push ${TEAM_NAME}/${PROJECT_NAME}core-shell:v1.0.0
docker push ${TEAM_NAME}/${PROJECT_NAME}core-shell:latest
docker push ${TEAM_NAME}/${PROJECT_NAME}core-rstudio:v1.0.0
docker push ${TEAM_NAME}/${PROJECT_NAME}core-rstudio:latest

# 5. Initialize zzcollab project with custom base image
zzcollab --base-image ${TEAM_NAME}/${PROJECT_NAME}core-shell \
    --dotfiles ~/dotfiles
# This automatically:
#
# - Creates complete R package structure
# - Builds LOCAL development image (inherits from core + adds dotfiles)
# - Sets up CI/CD for automated team image rebuilds
# - Local image NOT pushed - contains personal dotfiles

# 6. Set up PRIVATE GitHub repository for research code
git init
git add .
git commit -m "🎉 Initial research project setup

- Complete zzcollab research compendium
- Team core image published to Docker Hub:
  ${TEAM_NAME}/${PROJECT_NAME}core:v1.0.0
- Private repository protects unpublished research
- CI/CD configured for automatic team image updates"

# Create PRIVATE repository on GitHub first, then:
```

```
git remote add origin https://github.com/[TEAM]/project.git  # PRIVATE repo
git push -u origin main

# 7. Start development immediately
# Enter containerized development environment with your dotfiles
make docker-zsh
```

Developer 1: Analysis Development Cycle

```
# Inside the container (after make docker-zsh):

# 1. Create initial analysis script
vim scripts/01_initial_analysis.R
# Write first iteration of analysis in R

# 2. Write tests for the analysis immediately
vim tests/integration/test-01_initial_analysis.R
# Write integration tests:
# test_that("initial analysis script runs without errors", {
#   expect_no_error(source(here("scripts", "01_initial_analysis.R")))
#   expect_true(file.exists(here("data", "derived_data",
#                                "analysis_output.rds")))
# })

# 3. Run tests to verify everything works
R
# testthat::test_dir("tests/integration")  # Run integration tests
# source("scripts/01_initial_analysis.R")  # Test the script directly
# quit()

# 4. Exit container when iteration is complete
exit

# 5. Commit changes with CI/CD trigger
git add .
git commit -m "Add initial analysis script with tests

- First iteration of data analysis
- Created scripts/01_initial_analysis.R
- Added integration tests for analysis pipeline
- All tests passing"
git push                        # → Triggers GitHub Actions validation

# 6. CI automatically handles package updates
# - If new packages detected: renv::snapshot() runs
# - Team Docker image rebuilds automatically
```

```
# - New image pushed to Docker Hub
# - Team gets notification of updated environment

# 7. Continue development cycle
make docker-zsh                    # Back to development environment
# Repeat: code → test → exit → commit → push
```

Benefits of Automated Team Image Management:

- **Faster onboarding**: New developers get started in minutes, not hours
- **Environment consistency**: Everyone uses identical package versions
- **Bandwidth efficiency**: ~500MB pull vs ~2GB+ rebuild
- **CI/CD optimization**: Faster automated testing with pre-built dependencies
- **Package management**: Centralized control over research environment
- **Version control**: Tag images for different analysis phases
- **Automated updates**: Team image rebuilds automatically when packages change
- **Zero manual intervention**: Developers never worry about image management

Automated Team Image Updates

ZZCOLLAB includes automated GitHub Actions workflows that rebuild and publish the team Docker image whenever package dependencies change. This ensures all team members always have access to the latest, consistent development environment.

Key Benefits:

- **Zero manual intervention** required for Docker image management
- **Automatic detection** of package changes in `renv.lock` or `DESCRIPTION`
- **Multi-tag versioning** for different use cases
- **Team notification** system for new image availability
- **Build caching** for faster rebuild times

Full documentation and implementation details are provided in the Automated Docker Image Management section at the end of this document.

Developer Collaboration Workflow Sequence

□□ Developer 1 (Initial Development Work)

```
# Project setup already completed in pre-collaboration phase
cd research-project

# 1. Start development work in containerized vim environment
make docker-zsh                # → Enhanced zsh shell with personal dotfiles

# 2. Add any additional packages for initial analysis
# (In zsh container with vim IDE)
R                              # Start R session
# Most packages already installed in team image
# install.packages("additional_package") # Only if needed
# renv::snapshot()             # Update if packages added
# quit()                       # Exit R
#
# NOTE: renv.lock is automatically created during project initialization

# 3. Test-driven development workflow using vim
# First, learn testing patterns
Rscript scripts/00_testing_guide.R   # → Review testing instructions

# Create package functions with tests
vim R/analysis_functions.R           # Create package functions
# Write R functions with vim + plugins

vim tests/testthat/test-analysis_functions.R  # Write tests for functions
# Write unit tests for each function:
# test_that("function_name works correctly", {
#   result <- my_function(test_data)
#   expect_equal(nrow(result), expected_value)
#   expect_true(all(result$column > 0))
# })

# Test the functions
R                                    # Start R session
# devtools::load_all()               # Load package functions
# devtools::test()                   # Run tests to verify functions work
# quit()                             # Exit R

vim scripts/01_data_import.R         # Create analysis scripts
# Write data import code
# Note: scripts/ directory includes templates for:
#
```

```
# - 02_data_validation.R (data quality checks)
# - 00_setup_parallel.R (high-performance computing)
# - 00_database_setup.R (database connections)
# - 99_reproducibility_check.R (validation)
# - 00_testing_guide.R (testing instructions)

vim tests/integration/test-data_import.R  # Create integration tests
# Write integration tests for analysis scripts:
# test_that("data import script runs without errors", {
#   expect_no_error(source(here("scripts", "01_data_import.R")))
# })

vim analysis/report/report.Rmd        # Start research report
# Write analysis and methods in R Markdown

# Test report rendering
R                                     # Start R session
# rmarkdown::render("analysis/report/report.Rmd")  # Test report compiles
# quit()                              # Exit R

# 4. Quality assurance and commit
exit                                  # Exit container
make docker-check-renv                # Validate dependencies
make docker-test                      # Run package tests
make docker-render                    # Test report rendering
# Optional: Check reproducibility
# Rscript scripts/99_reproducibility_check.R

# 5. Commit changes with CI/CD trigger
git add .
git commit -m "Add initial analysis and dependencies"
git push                              # → Triggers GitHub Actions validation
```

□□ Developer 2 (Joining Project)

□ Developer 2 Checklist:

- □ Get access to private GitHub repository from team lead
- □ Clone the private repository to local machine
- □ Choose preferred development interface (shell or RStudio)
- □ Update docker-compose.yml to reference chosen core image
- □ Build local development image with personal dotfiles
- □ Create feature branch for your analysis work
- □ Write analysis script in scripts/ directory
- □ Write integration tests for your analysis script
- □ Run tests to verify everything works

- ☐ Commit and push code + tests together
- ☐ Create pull request for team review

```
# 1. Get access to PRIVATE repository and clone
# Team lead must add you as collaborator to private GitHub repo first
git clone https://github.com/rgt47/png1.git  # Use actual team/project names
cd png1

# 2. Build your personal development environment
# Choose between command-line or R interface:

# Option A: Command-Line Interface
zzcollab --team rgt47 --project-name png1 --interface shell \
    --dotfiles ~/dotfiles

# Option B: R Interface (R-Centric Workflow)
# R
# library(zzcollab)
# join_project(
#   team_name = "rgt47",
#   project_name = "png1",
#   interface = "shell",
#   dotfiles_path = "~/dotfiles"
# )
# quit()

# Both approaches automatically pull the team image from Docker Hub and
# add your personal dotfiles

# 3. Start development immediately
make docker-zsh                   # Shell interface with vim/tmux
# OR
make docker-rstudio              # RStudio Server at localhost:8787

# 4. Create feature branch for your work
git checkout -b feature/visualization-analysis

# 5. Add your analysis work (inside container)
# (In zsh container with vim)
vim scripts/02_visualization_analysis.R
# Write visualization analysis code

# If you need new packages, just install them:
# R
# install.packages("ggplot2")   # Add new package
# quit()
```

```
# 6. Write tests for your analysis immediately
vim tests/integration/test-02_visualization_analysis.R
# Write integration tests:
# test_that("visualization analysis runs successfully", {
#   expect_no_error(source(here("scripts",
#                               "02_visualization_analysis.R")))
#   expect_true(file.exists(here("analysis", "figures", "plot1.png")))
# })

# 7. Run tests to verify everything works
R
# testthat::test_dir("tests/integration")  # Run all integration tests
# source("scripts/02_visualization_analysis.R")  # Test your script
# quit()

exit

# 8. Commit and push your changes (code + tests together)
# Choose between command-line or R interface:

# Option A: Command-Line Interface
git add .
git commit -m "Add visualization analysis with tests

- Created scripts/02_visualization_analysis.R
- Added ggplot2 for data visualization
- Added integration tests for visualization pipeline
- All tests passing"
git push origin feature/visualization-analysis

# Option B: R Interface (R-Centric Workflow)
# R
# library(zzcollab)
# git_status()  # Check changes
# git_commit("Add visualization analysis with tests - Created
#     scripts/02_visualization_analysis.R - Added ggplot2 for data
#     visualization - Added integration tests for visualization pipeline -
#     All tests passing")
# git_push("feature/visualization-analysis")
# quit()

# 9. Create pull request
# Choose between command-line or R interface:

# Option A: Command-Line Interface
```

```
gh pr create --title "Add visualization analysis with tests" \
    --body "Added visualization analysis script with comprehensive tests" \
    --base main

# Option B: R Interface (R-Centric Workflow)
# R
# library(zzcollab)
# create_pr(
#   title = "Add visualization analysis with tests",
#   body = "Added visualization analysis script with comprehensive tests"
# )
# quit()

# 10. CI automatically handles the rest!
# - If new packages detected: renv::snapshot() runs
# - Team Docker image rebuilds automatically
# - New image pushed to Docker Hub
# - Team gets notification of updated environment
```

□□ Developer 1 (Continuing Work - After PR Review)

```
# 1. Review and merge Developer 2's pull request
# On GitHub: Review PR, approve, and merge to main branch

# 2. Sync with Developer 2's merged changes
git checkout main               # Switch to main branch
git pull origin main           # Get latest changes from main repo
# Note: Use 'origin' not 'upstream' for single-repository workflow

# 3. Get latest team Docker image (automatically updated by GitHub Actions)
docker pull rgt47/png1:latest  # Pull from Docker Hub (public)
# Note: If Dev 2 added packages, GitHub Actions already rebuilt
# and pushed the image!

# 4. Validate environment consistency
make docker-check-renv          # Ensure all dependencies are properly tracked

# 5. Create new feature branch for advanced modeling
git checkout -b feature/advanced-models

# 6. Continue development with updated environment
make docker-zsh                # → Environment now includes Dev 2's packages

# 7. Add more analysis work using vim
# (In zsh container with vim)
R                              # Start R session
```

```
# Ensure all packages from Dev 2 are available
# renv::restore()
# devtools::load_all()         # Load updated package with new functions
# quit()

# 8. Test-driven advanced analysis development
vim R/modeling_functions.R     # Add statistical modeling functions
# Write multilevel model functions

vim tests/testthat/test-modeling_functions.R  # Write tests for modeling
# Write unit tests for statistical models:
# test_that("multilevel_model function works", {
#   model <- fit_multilevel_model(test_data)
#   expect_s3_class(model, "lmerMod")
#   expect_true(length(fixef(model)) > 0)
# })

# Test new modeling functions
R                              # Start R for testing
# devtools::load_all()         # Load all functions including new ones
# devtools::test()             # Run all tests (Dev 1, Dev 2, and new tests)
# quit()

vim scripts/03_advanced_models.R  # Create modeling script
# Write analysis using both Dev 1 and Dev 2's functions

vim tests/integration/test-complete_pipeline.R  # Create comprehensive tests
# Write end-to-end pipeline tests:
# test_that("complete analysis pipeline works", {
#   expect_no_error(source(here("scripts", "01_data_import.R")))
#   expect_no_error(source(here("scripts", "02_visualization.R")))
#   expect_no_error(source(here("scripts", "03_advanced_models.R")))
# })

# 7. Test complete integration of all developers' work
R                              # Comprehensive integration testing
# devtools::load_all()         # Load all functions
# testthat::test_dir("tests/testthat")      # Run all unit tests
# testthat::test_dir("tests/integration")  # Run all integration tests
# source("scripts/01_data_import.R")       # Dev 1's work
# source("scripts/02_visualization.R")     # Dev 2's work
# source("scripts/03_advanced_models.R")   # New integration
# quit()

# 8. Update research report with testing
vim analysis/report/report.Rmd  # Update manuscript
```

```
# Add new results and figures

vim tests/integration/test-report_rendering.R  # Create report tests
# Write tests for report compilation:
# test_that("report renders successfully", {
#   expect_no_error(rmarkdown::render(here("analysis", "report",
#                                          "report.Rmd")))
#   expect_true(file.exists(here("analysis", "report", "report.pdf")))
# })

# Test report rendering
R                               # Test report compilation
# rmarkdown::render("analysis/report/report.Rmd")  # Verify report compiles
# Run all integration tests
# testthat::test_dir("tests/integration")
# quit()

# 11. Enhanced collaboration workflow with proper PR
exit                            # Exit container

# 12. Create comprehensive pull request
git add .
git commit -m "Add advanced multilevel modeling with integrated visualization

- Add modeling_functions.R with multilevel model utilities
- Create comprehensive test suite for statistical models
- Add end-to-end pipeline integration tests
- Update research report with new analysis results
- Test complete workflow integration"

# Push feature branch to origin
git push origin feature/advanced-models

# 13. Create pull request with detailed review checklist
gh pr create --title "Add advanced multilevel modeling analysis" \
    --body "## Summary
- Integrates visualization functions from previous PR
- Adds multilevel modeling capabilities with lme4
- Includes comprehensive end-to-end testing
- Updates research manuscript with new results

## Analysis Impact Assessment

- [x] All existing functionality preserved
- [x] New models compatible with existing visualization pipeline
- [x] Data validation passes for modeling requirements
```

```
- [x] Reproducibility check passes

## Testing Coverage

- [x] Unit tests for all modeling functions
- [x] Integration tests for complete analysis pipeline
- [x] Paper rendering validation with new results
- [x] All existing tests continue to pass

## Reproducibility Validation

- [x] renv.lock updated with new dependencies
- [x] Docker environment builds successfully
- [x] Analysis runs from clean environment
- [x] Results consistent across platforms

## Collaboration Quality

- [x] Code follows established patterns
- [x] Functions integrate cleanly with existing codebase
- [x] Documentation updated for new capabilities
- [x] Commit messages follow conventional format" \
            --base main
```

Key Collaboration Features

(Professional Git Workflow + Test-Driven Development)

Automated Quality Assurance on Every Push:

- **R Package Validation**: R CMD check with dependency validation
- **Comprehensive Testing Suite**: Unit tests, integration tests, and data validation
- **Paper Rendering**: Automated PDF generation and artifact upload
- **Multi-platform Testing**: Ensures compatibility across environments
- **Dependency Sync**: renv validation and DESCRIPTION file updates

Test-Driven Development Workflow:

- **Unit Tests**: Every R function has corresponding tests in `tests/testthat/`

- **Integration Tests**: Analysis scripts tested end-to-end in `tests/integration/`
- **Data Validation**: Automated data quality checks using `scripts/02_data_validation.R`
- **Reproducibility Testing**: Environment validation with `scripts/99_reproducibility_check.R`
- **Paper Testing**: Manuscript rendering validation for each commit

Enhanced GitHub Templates:

- **Pull Request Template**: Analysis impact assessment, reproducibility checklist
- **Issue Templates**: Bug reports with environment details, feature requests with research use cases
- **Collaboration Guidelines**: Research-specific workflow standards

Fully Automated Professional Workflow:

```
# Single repository collaboration with feature branches:
git clone https://github.com/[TEAM]/project.git  # Clone team repo
git checkout -b feature/your-analysis    # Create feature branch
# ... do development work with tests ...
git push origin feature/your-analysis   # Push to team repo
gh pr create --title "Add analysis" --body "..."  # Create pull request

# After PR merge - ZERO manual image management needed:
git checkout main             # Switch to main branch
git pull origin main         # Get latest from team repo
docker pull team/project:latest  # Get auto-updated team image from Hub
make docker-zsh              # → Instantly ready with all new packages!

#  GitHub Actions automatically:
# - Detects renv.lock changes in merged PR
# - Rebuilds Docker image with new packages
# - Pushes updated image to container registry
# - Updates docker-compose.yml references
# - Notifies team of new image availability
```

Data Management Collaboration:

```
# Structured data workflow for teams:
data/
├── raw_data/                # Dev 1 adds original datasets
├── derived_data/            # Dev 2 adds processed data
```

```
├── metadata/                  # Both document data sources
└── validation/                # Automated quality reports
```

Vim IDE Development Environment

Enhanced Vim Setup (via zzcollab dotfiles)

The containerized environment includes a fully configured vim IDE with:

Vim Plugin Ecosystem:

- **vim-plug**: Plugin manager (automatically installed)
- **R Language Support**: Syntax highlighting and R integration
- **File Navigation**: Project file browser and fuzzy finding
- **Git Integration**: Git status and diff visualization
- **Code Completion**: Intelligent autocomplete for R functions

Essential Vim Workflow Commands:

```
# In container vim session:
vim R/analysis.R               # Open R file
:Explore                       # File browser
:split scripts/data.R          # Split window editing
:vsplit analysis/report.Rmd    # Vertical split for manuscript

# File navigation shortcuts:
:e scripts/                    # Quick script navigation
:find model                    # Fuzzy file finding
:b                             # Buffer switching

# R integration:
# Define custom key mappings in .vimrc for:
# - Send R code to tmux pane
# - Run current function/selection
# - Insert R function templates
```

Tmux Integration for Multi-pane Development:

```
# Standard tmux layout automatically configured:
# ┌─────────────────────┬─────────────────────┐
# │                     │                     │
# │  VIM Editor         │  File Browser       │
# │  (R scripts)        │  (project)          │
# │                     │                     │
# ├─────────────────────┼─────────────────────┤
```

```
# │                 │                 │
# │  R Console      │   Git Status    │
# │  (testing)      │   (version ctrl)│
# │                 │                 │
# └─────────────────┴─────────────────┘

# Tmux session management:
tmux list-sessions              # Show active sessions
tmux attach -t analysis         # Attach to analysis session
tmux kill-session -t analysis   # End session
```

Advanced Development Workflow Features

Integrated Testing Environment:

```
# Real-time test-driven development cycle:
# 1. Edit R function in vim (top pane)
vim R/analysis_functions.R

# 2. Update corresponding test (split pane)
:split tests/testthat/test-analysis_functions.R

# 3. Run tests immediately (bottom tmux pane)
# Ctrl+b then down arrow to switch to R console pane
# devtools::test()

# 4. View test results, iterate on function
# Ctrl+b then up arrow to return to vim pane
```

Git Integration Workflow:

```
# Enhanced git workflow in vim:
vim +Git                        # Git fugitive interface
:Gstatus                        # View changed files
:Gdiff                          # View file differences
:Gcommit                        # Create commits from vim

# Command line git workflow:
git status                      # Check working directory
git add scripts/new_analysis.R tests/integration/test-new_analysis.R
git commit -m "Add new analysis with tests"
git push origin feature/new-analysis
```

Package Development Integration:

```
# R package development workflow in vim:
vim DESCRIPTION                # Edit package metadata
vim R/package_function.R       # Write package functions
vim man/function.Rd           # Edit documentation (auto-generated)

# Test package in R console:
# devtools::load_all()         # Load package functions
# devtools::document()         # Generate documentation
# devtools::test()             # Run all tests
# devtools::check()            # R CMD check validation
```

Research Report Writing Environment

R Markdown Integration:

```
# Manuscript development workflow:
vim analysis/report/report.Rmd  # Write research report
:set filetype=rmarkdown         # Enable R Markdown syntax highlighting

# Split pane for reference management:
:vsplit analysis/report/references.bib  # Bibliography file

# Live preview workflow:
# Bottom tmux pane for rendering:
# rmarkdown::render("analysis/report/report.Rmd")
# system("open analysis/report/report.pdf")  # Mac
# system("xdg-open analysis/report/report.pdf")  # Linux
```

Figure and Table Management:

```
# Structured output management:
vim scripts/05_generate_figures.R  # Figure generation script
# Figures automatically saved to: analysis/figures/

vim scripts/06_generate_tables.R   # Table generation script
# Tables automatically saved to: analysis/tables/

# Reference in report.Rmd:
# ![Figure 1](../figures/main_results.png)
# kable(readRDS("../tables/summary_stats.rds"))
```

Automated Docker Image Management

Comprehensive GitHub Actions Workflow

ZZCOLLAB includes sophisticated automated Docker image management that eliminates manual container maintenance while ensuring perfect environment consistency across research teams.

Complete GitHub Actions Workflow The system automatically detects package changes, rebuilds Docker images, and notifies team members through a comprehensive GitHub Actions workflow:

Key Features:

- **Intelligent change detection**: Monitors `renv.lock`, `DESCRIPTION`, `Dockerfile`, `docker-compose.yml`
- **Multi-platform support**: AMD64 and ARM64 architectures
- **Advanced caching**: GitHub Actions cache with BuildKit optimization
- **Comprehensive tagging**: `latest`, `r4.3.0`, `abc1234`, `2024-01-15` tags
- **Automated configuration**: Updates docker-compose.yml references
- **Team communication**: Detailed commit comments with usage instructions

Workflow Triggers

```yaml
# Automatic triggers
on:
  push:
    branches: [main]
    paths:
      - 'renv.lock'            # R package dependency changes
      - 'DESCRIPTION'          # Package metadata changes
      - 'Dockerfile'           # Container definition changes
      - 'docker-compose.yml'   # Service configuration changes
  workflow_dispatch:            # Manual triggering
```

Usage Scenarios Scenario 1: Developer Adds New Package

```
# Developer workflow
R
install.packages("tidymodels")
renv::snapshot()
# Create PR and merge
```

```
# Automatic result:
# ✅ GitHub Actions detects renv.lock changes
# ✅ Rebuilds image with tidymodels
# ✅ Pushes to mylab/study2024:latest on Docker Hub
# ✅ Updates docker-compose.yml
# ✅ Notifies team via commit comment
```

Scenario 2: Team Member Gets Updates

```
# Team member workflow
git pull                          # Gets latest changes
docker pull mylab/study2024:latest  # Gets updated environment
make docker-zsh                 # Instant access to new packages
```

Security and Privacy Model

🔒 PRIVATE GitHub Repository:

- Protects unpublished research and sensitive methodologies
- Secures proprietary data analysis and preliminary results
- Controls access to research collaborators only

🌍 PUBLIC Docker Images (Docker Hub):

- Enables reproducible research by sharing computational environments
- Supports open science through transparent methodology
- No sensitive data included - only software packages and configurations

Repository Secrets Setup

For automated Docker Hub publishing, configure these secrets in your **private** GitHub repository:

```
# In GitHub repository: Settings → Secrets and variables → Actions
DOCKERHUB_USERNAME: your-dockerhub-username
DOCKERHUB_TOKEN: your-dockerhub-access-token  # Create at hub.docker.com/settings/se
```

Standard R Package Validation

R Package Check (`.github/workflows/r-package.yml`)

- **Triggers**: Push/PR to main/master
- **Purpose**: Validate package structure and dependencies
- **Features**:
  - Native R setup (fast execution)
  - renv synchronization validation with enhanced dependency scanning

  - Package checks across platforms
  - Dependency validation with `check_renv_for_commit.R --strict-imports`

Paper Rendering (`.github/workflows/render-report.yml`)

- **Triggers**: Manual dispatch, changes to analysis files
- **Purpose**: Generate research paper automatically
- **Features**:
  - Automatic PDF generation
  - Artifact upload
  - Native R environment (faster than Docker)

Advanced Workflow Implementation

Package Change Detection Logic

```yaml
# .github/workflows/update-team-image.yml
jobs:
  detect-changes:
    runs-on: ubuntu-latest
    outputs:
      packages-changed: ${{ steps.changes.outputs.packages }}
      reason: ${{ steps.changes.outputs.reason }}
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 2

      - name: Check for package or container changes
        id: changes
        run: |
          if git diff HEAD~1 --name-only | \
               grep -E "(renv\.lock|DESCRIPTION|Dockerfile|docker-compose\.yml)"; th
            echo "packages=true" >> $GITHUB_OUTPUT
            echo "reason=Package or container configuration changes detected" >> $GI
          else
            echo "packages=false" >> $GITHUB_OUTPUT
          fi
```

Multi-Platform Docker Build

```yaml
  build-and-push:
    needs: detect-changes
    if: needs.detect-changes.outputs.packages-changed == 'true'
    runs-on: ubuntu-latest
```

```yaml
    steps:
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build and push team images
        uses: docker/build-push-action@v5
        with:
          context: .
          file: ./Dockerfile
          platforms: linux/amd64,linux/arm64
          push: true
          tags: |
            ${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:latest
            ${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:${{ env.R_VERSION }}
            ${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:${{ github.sha }}
            ${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:${{ env.DATE_TAG }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Package Comparison and Notification

```yaml
      - name: Compare packages and notify
        run: |
          # Extract package lists
          if [ -f renv.lock.backup ]; then
            jq -r '.Packages | keys[]' renv.lock.backup | sort > previous_packages.t
          else
            touch previous_packages.txt
          fi

          if [ -f renv.lock ]; then
            jq -r '.Packages | keys[]' renv.lock | sort > current_packages.txt
          fi

          # Calculate differences
          NEW_PACKAGES=$(comm -13 previous_packages.txt current_packages.txt | tr '\
          REMOVED_PACKAGES=$(comm -23 previous_packages.txt current_packages.txt | t

          # Post detailed comment
          gh api repos/${{ github.repository }}/commits/${{ github.sha }}/comments \
            --field body="🐳 **Team Docker Image Updated**

          **New Image Tags:**
          - \`${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:latest\`
          - \`${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:${{ env.R_VERSION }}\`
```

```
**Package Changes:**
${NEW_PACKAGES:+- **Added:** $NEW_PACKAGES}
${REMOVED_PACKAGES:+- **Removed:** $REMOVED_PACKAGES}

**Team Members:** Update your environment with:
\`\`\`bash
git pull
docker pull ${{ env.TEAM_NAME }}/${{ env.PROJECT_NAME }}:latest
make docker-zsh
\`\`\`"
```

Team Collaboration Benefits

Automated Workflow Benefits

Traditional Workflow	Automated ZZCOLLAB Workflow
Manual image rebuilds	✅ **Automatic rebuilds on package changes**
Inconsistent environments	✅ **Guaranteed environment consistency**
30-60 min setup per developer	✅ **3-5 min setup with pre-built images**
Manual dependency management	✅ **Automated dependency tracking**
Docker expertise required	✅ **Zero Docker knowledge needed**
Build failures block development	✅ **Centralized, tested builds**

Developer Experience

- **Researchers focus on research** - not DevOps
- **Onboarding new team members** takes minutes, not hours
- **Package management** happens transparently
- **Environment drift** is impossible
- **Collaboration friction** eliminated entirely

Quality Assurance Integration

```
# Complete automated quality pipeline:
git push                        # Developer pushes code

# GitHub Actions automatically:
# 1. Run R package validation
# 2. Execute comprehensive test suite
# 3. Render research paper
```

```
# 4. Detect package changes
# 5. Rebuild Docker images if needed
# 6. Push updated images to registry
# 7. Notify team of environment updates
# 8. Update project configuration
```

Production-Ready Research Infrastructure

ZZCOLLAB provides enterprise-grade research collaboration infrastructure that handles the technical complexity automatically, allowing research teams to focus entirely on their scientific work while maintaining perfect reproducibility and professional development standards.

The framework ensures:

- **Zero-friction collaboration** with automated environment management
- **Enterprise-grade quality** with comprehensive testing and validation
- **Perfect reproducibility** across all team members and time
- **Professional workflows** with automated CI/CD and documentation
- **Research focus** by eliminating technical barriers and manual processes

This automated infrastructure transforms research team collaboration from a technical challenge into a seamless, professional workflow that supports scientific discovery and reproducible research practices.