PharmaSUG 2026 - Paper ET-358

Automating Git Workflows in SAS with Git Functions

Lleyton Seymour, SAS Institute

ABSTRACT

Whether you are an independent developer or part of a larger organization, Git has become a foundational component of the modern development workflow, and SAS users are no exception. SAS users will typically interact with a Git client in one of two ways: through a command-line interface or through a point-and-click GUI. While both approaches are sufficient, when working with scheduled jobs, flows, or automated pipelines, neither is ideal. This becomes especially apparent in regulated settings where objects, such as models, may be produced by scheduled or automated workflows without a traceable history.

Enter Git Functions; these functions enable developers to execute Git operations directly within their SAS code, eliminating the need for manual intervention. By embedding version control directly into SAS programs, Git Functions allow the automated tracking of model artifacts, the execution of conditional operations, and even the ability to produce structured reporting based on your commit history. This paper provides a foundational introduction to Git Functions and how they are incorporated into everyday workflows, leaving attendees with reusable approaches for automating version control in their own environments.

INTRODUCTION

Git has become a routine element in modern analytics and software development. Regardless of whether you work independently or as part of a larger team, you likely already rely on Git to version code, manage collaboration, and maintain a history of changes. SAS users follow suit as modeling and analytical work increasingly occurs alongside open-source tools, shared repositories, and automated pipelines. With this in mind, incorporating version control into SAS workflows becomes increasingly important.

In practice, most SAS users interact with Git through either a command-line interface or a point-and-click GUI. Both approaches function well for manual development, but they introduce friction during Git operations within repeatable SAS processes. This situation becomes especially relevant in environments where programs run unattended, such as scheduled jobs, flows, or model retraining pipelines. In these cases, code and artifacts are automatically created or updated, yet the necessary Git steps to capture those changes are typically outside the workflow unless a developer intervenes. Git Functions bridge this gap by enabling SAS programs to perform Git operations directly in code. Instead of treating version control as a separate manual task, developers can embed common Git actions into the SAS programs that create, update, and evaluate their assets, analytics, models, and related artifacts. This functionality automates common actions, such as cloning repositories, inspecting commit history, comparing versions, viewing commit outputs, and pushing changes.

This paper introduces Git Functions and illustrates how they support more auditable, repeatable development. Attendees will discover how common Git actions translate into SAS functions, how to combine those functions into useful workflows, and how to extend version control into areas where SAS operates. The aim is not to replace traditional Git usage but to make Git-aware automation a practical aspect of everyday development.

WHAT ARE GIT FUNCTIONS?

Git functions are exactly what they sound like: built-in SAS functions that enable you to interact with Git repositories from SAS code.

Rather than relying on manual input from an external CLI or clicking through a GUI, Git Functions let SAS programs perform common Git operations programmatically. Routine tasks like cloning a repository, pulling updates, committing files, or pushing changes can now be executed as part of a SAS program or scheduled job.

Git functions are not exclusive to SAS Viya; programmers using SAS 9.4, SAS Viya, or even the new SAS Viya Workbench all have access to Git functions. This makes them especially useful for teams working across multiple environments or even teams modernizing their workflows from SAS 9 to Viya.

CORE FUNCTIONS

To better understand where Git Functions sit, it helps to start with the actions most developers already use in their day-to-day workflows. Typically, when coding in a shared setting, a developer might take several actions, including cloning a repository, pulling the latest changes, reviewing any differences, and committing their own changes before finally pushing those changes to a remote branch. Git Functions mirror these common actions using SAS-based equivalents wrapped in DATA steps. Basic actions like cloning a repository have a singular equivalent, such as GIT_CLONE, while more complex actions, such as branch checkout and merges, are split across several functions.

With such an extensive list of possible actions, the number of available Git Functions quickly becomes overwhelming. With this in mind, this paper focuses on a set of 'core' functions; these functions are not only common in a traditional developer's workflow but also demonstrate a wide variety of syntax, required arguments, and possible outputs. For a complete list of available functions and syntax details, please see the *Using Git Functions in SAS* documentation page.

GIT_CLONE

The GIT_CLONE function clones the specified Git repository into a directory on the SAS server. Connections to the Git repository can be made using HTTPS or SSH.

```
data _null_;
   rc = git_clone (
     "https-url",
     "target-directory");
   put rc=;
run;
```

GIT_COMMIT

The GIT_COMMIT function commits all staged files to the local repository. Note: when committing changes to the current update reference, specify "HEAD" for the "update-reference" argument.

```
data _null_;
   rc = git_commit (
     "your-local-repository",
     "update-reference", /* this is typically "HEAD" */
     "your-name",
     "your-email",
     "your-commit-message");
   put rc=;
run;
```

GIT_PUSH

The GIT_PUSH function pushes changes from the local repository to the remote repository. Like other Git functions GIT_PUSH also uses “rc” for return codes, but also displays an accompanying message based on success.

```
data _null_;
   rc= git_push(
     "your-local-repository",
     "your-Git-username",
     "your-Git-password");
run;
```

GIT_DIFF

The GIT_DIFF function returns the number of changes between two commits and creates a diff record object in the local repository. GIT_DIFF can be combined with the GIT_DIFF_TO_FILE function to write observed changes in content to a file.

```
data _null_;
   n=git_diff(
     "your-repository",
     "older-commit-ID",
     "newer-commit-ID");
   put n=;
run;
```

GIT FUNCTIONS IN PRACTICE

The above examples, deemed 'core' functions, feel a bit reductive in comparison to traditional Git usage. At first glance, the trade-off appears to be a more jarring experience for an extra line in the SAS log. However, Git Functions begin to shine when used in combination with other functions and DATA steps. For example, take GIT_COMMIT_LOG and GIT_COMMIT_GET: when used independently, the former returns the number of commit objects, and the latter gives you a specified attribute from a specific commit object. When nested within a DATA step, you can build a complete commit history that includes the commit time, who pushed the changes, their ID or email, the branch they pushed to, and even their overall contribution percentage to the repository.

Below is a code snippet that illustrates this nested Git function example; Attendees can find the full example, along with other examples, in the accompanying Git-Functions-in-SAS repository.

```
/* Clone to a temp space (for illustrative purposes) */
options dlcreatedir;
%let repoPath = %sysfunc(getoption(WORK))/Git-Functions-in-SAS;
libname repo "&repoPath.";
libname repo clear;

/* Fetch latest code from repo */
data _null_;
   rc = git_clone(
     "https://github.com/lleytonse/Git-Functions-in-SAS.git",
     "&repoPath.");
   put rc=;
run;

/*  Build out 'commits' dataset  */
data commits (keep=time message);
   length time_char $ 100 time 8 message $ 100 n 8;
   n = git_commit_log("&repoPath.");
   format time datetime20.;

   do i=1 to n;
     rc = git_commit_get(i,"&repoPath.",'message',message);
     rc = git_commit_get(i,"&repoPath.",'time',time_char);
     time = trim(time_char);
     time = time + "01jan1970 0:0:0"dt;
     output;
   end;
run;

/* Output to ODS */
title "Latest commits to 'Git-Functions-in-SAS' repo";
proc print data=commits;
run;
title;
```

Program 1. CheckCommitHistory.sas

Latest commits to 'Git-Functions-in-SAS' repo

Obs	time	message
1	27MAR2026:14:58:27	Create folder for examples + add examples covering basic git functions
2	25MAR2026:16:46:14	Add training code and model card
3	25MAR2026:15:45:22	Adjust README.md title and spacing
4	25MAR2026:15:40:27	Update README.md to include training data and all available attributions
5	23MAR2026:18:00:57	First push

Figure 1. HTML Output Created by CheckCommitHistory.sas

Using only DATA steps, we can build examples like the above program; the real fun begins once we start mixing in PROCs and more complex actions.

For this example, I'll be using the popular open-source Ollama repository, as it better illustrates a diverse set of contributors and commits. Like before, we can start with a simple combination of GIT_COMMIT_LOG and GIT_COMMIT_GET to build out a dataset of commits. Now, with a dataset spanning the entire commit history, we can use PROC SQL to create some interesting queries. You can take this in all kinds of directions, but for this example, we will create a simple table that groups and ranks the contributors by their number of commits.

```
/* Clone to a temp space (for illustrative purposes) */
options dlcreatedir;
%let repoPath = %sysfunc(getoption(WORK))/ollama; /* Adjust your descriptor */
libname repo "&repoPath.";
libname repo clear;

/* Fetch latest code from repo */
data _null_;
   rc = git_clone(
     "https://github.com/lleytonse/Git-Functions-in-SAS.git", /* Chosen repo path */
     "&repoPath.");
   put rc=;
run;

...

/* Cut out repetitive code for space considerations */

/* Please see GitHub repository for full code */

/* https://github.com/lleytonse/Git-Functions-in-SAS */

...


/* Summarize commit counts and contribution percentage */
proc sql;
```

```
   create table top_contributors as
    select
     contributor,
     count(*) as commit_count,
     calculated commit_count /
       (select count(*) from commit_history) as pct_commits format=percent8.1
   from commit_history
   group by contributor
   order by commit_count desc;
quit;

/* Output to ODS */
title "Top Contributors to 'ollama' Repo";
proc print data=top_contributors (obs=10) noobs label ;
   label
     contributor = "Contributor"
     commit_count = "Commits"
     pct_commits = "Percent of Total Commits";
run;
title;
```

Program 2. CheckCommitShare.sas

Top Contributors to 'ollama' Repo

Contributor	Commits	Percent of Total Commits
Michael Yang	1458	21.1%
Jeffrey Morgan	910	13.1%
Daniel Hiltgen	899	13.0%
Bruce MacDonald	517	7.5%
jmorganca	309	4.5%
Patrick Devine	282	4.1%
Jesse Gross	261	3.8%
Blake Mizerany	243	3.5%
Josh Yan	232	3.4%
ParthSareen	186	2.7%

Figure 2. HTML Output Created by CheckCommitShare.sas

Once again, since these functions run entirely within SAS code, they work best when implemented in workflows where manual Git interaction is impractical or impossible. In the next section, we will look at one of these common scenarios.

AUTOMATING GIT WORKFLOWS WITH GIT FUNCTIONS

Now that we have established a baseline for our Git functions, we can begin exploring how they integrate into real-world scenarios.

USE CASE: VERSIONING MODELS IN SAS

As discussed earlier, Git Functions become especially useful when integrated into settings where automated processes can't accommodate manual interactions. This becomes particularly relevant in modeling workflows, where training, upkeep, and other pipelines often occur as part of a scheduled or repeatable process. In these types of settings, Git Functions allow version control to move with the SAS process rather than remaining a separate manual task. To illustrate this, we can use a typical predictive maintenance scenario. Throughout this section, we will frame all of the examples around this specific scenario. If you would like to dive deeper into any part of the underlying data, modeling, or code, you can find it included in the accompanying GitHub repository.

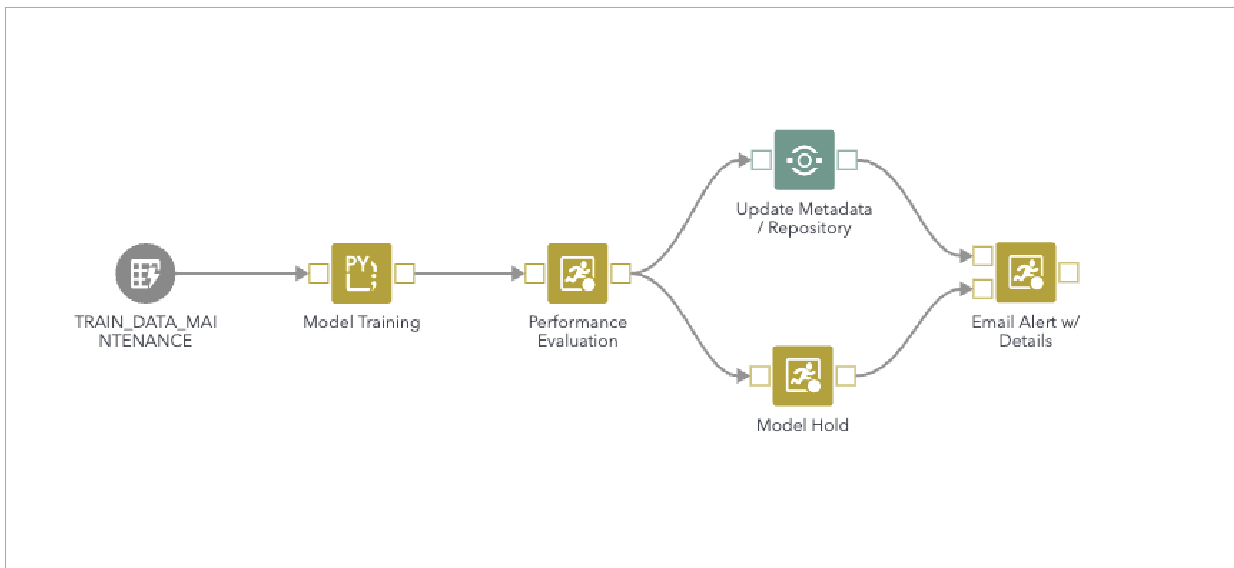

Figure 3. Example Scheduled SAS Flow for Model Retraining, Evaluation, Repository Update, and Alerting (ScheduledModelTraining.flw)

The figure above illustrates a common retraining pipeline. While the initial node omits the training retrigger, the process is similar whether it uses a fixed (retraining at a cadence) or a dynamic (ad hoc-triggered retraining) training approach.

The flow follows a typical path until the Performance Evaluation node, where we see the use of integrated Git functions. As a side note, this retraining flow assumes a development and production repository, where all training occurs in development and is evaluated before being published to production. At this node, we use GIT_DIFF and GIT_DIFF_TO_FILE to compare the updated model's variable importance with that from its previous training. Through this, we can not only keep a record/version of all our updates, but also use GIT_DIFF_TO_FILE to create a flag that holds any models that may perform outside their expected behavior.

```
/* Clone to a temp space (for illustrative purposes) */
options dlcreatedir;
%let repoPath = %sysfunc(getoption(WORK))/Git-Functions-in-SAS;
libname repo "&repoPath.";
libname repo clear;

/* Fetch latest code from repo */
data _null_;
   rc = git_clone(
```

```
    "https://github.com/lleytonse/Git-Functions-in-SAS.git",
    "&repoPath.");
  put rc=;
run;

%let oldCommit  = 3e20c3cce3396d710959a7cba0f57aa03e0f757b;
%let newCommit  = b22750cef7ab21ec0c9978915063d1d7e5b5f4f7;
%let diffFile   = /nfsshare/sashls2/home/llseym/diff_content.txt;
%let threshold  = 0.10;   /* Setting our flag to hold model from produciton */

/* Run GIT_DIFF between the two commit IDs */
data _null_;
   n = git_diff(
      "&repoPath.",
      "&oldCommit.",
      "&newCommit."
   );
   call symputx('diffCount', n);
   put n=;

   /* Write to file using GIT_DIFF_TO_FILE */
   if n > 0 then do;
      rc = git_diff_to_file(
         1,
         "&repoPath.",
         "&oldCommit.",
         "&newCommit.",
         "&diffFile."
      );
      put rc=;
   end;
run;

/* Parse the created file and calculate importance change in 'Differential_pressure'*/
data diff_check;
   infile "&diffFile." lrecl=1000 truncover;
   length variable $50 line $1000 value $100 type $20;
   retain in_block 0 old_rel new_rel . variable 'Differential_pressure';
   input line $char1000.;

   if index(line, 'Differential_pressure') then in_block = 1;

   if index(line, '"type":"deletion"') then type='deletion';
   else if index(line, '"type":"addition"') then type='addition';
   else if index(line, '"type":"context"') then type='context';
   else type='other';

   if in_block and index(line, 'relative_importance') then do;
     value = prxchange('s/.*relative_importance":\s*([0-9.]+).*/$1/', 1, line);

     if type='deletion' then old_rel = input(value, best32.);
     if type='addition' then new_rel = input(value, best32.);
   end;

   if not missing(old_rel) and not missing(new_rel) then do;
     abs_change = old_rel - new_rel;
     pct_drop   = divide(old_rel - new_rel, old_rel);
     flag_hold  = (pct_drop > &threshold.);
     output;
     stop;
   end;
run;

/* Output to ODS */
title "Variable Importance Change Check";
proc print data=diff_check noobs label;
   var variable old_rel new_rel abs_change pct_drop flag_hold;
```

```
   label variable = "Variable" old_rel = "Relative Importance (old)" new_rel = "Relative
Importance (new)"
   abs_change = "Absolute Change" pct_drop = "Percent Drop" flag_hold = "Hold from
Production";
   format pct_drop percent8.2;
run;
title;
```

Program 3. CommitFlag.sas

Variable Importance Change Check

Variable	Relative Importance (old)	Relative Importance (new)	Absolute Change	Percent Drop	Hold from Production
Differential_pressure	0.7436	0.5436	0.2	26.90%	1

Figure 4. HTML Output Created by CommitFlag.sas

In this retraining excerpt, we use GIT_DIFF_TO_FILE to output the changes from previous training sessions. Then, using a DATA step combined with a bit of RegEx, we can identify the type of values we are looking for and compare them. For this specific example, we are only looking for changes in Differential_pressure, but you can generalize the code to handle all kinds of scenarios.

Update Metadata / Repository

Git Details Node Notes Reload Edit

Fill out the required fields below to commit, stage, and push changes to a remote repository

User Name / ID

your-name

User Email

your-email

Local Repository Path

/local-path-to-a-folder

Remote Repository Path

https://github.com/your-user-id/example-repo.git

Branch / Update Reference

HEAD

Figure 5. User Form for UpdateMetadataRepository.step

Let's continue through the flow. When observed changes in the model are within tolerance, we can push these adjustments to production. In this scenario, we push updates via the Update Metadata/Repository

node. In the screenshot below, we can see that this node is a custom step that takes user input to help build the production push. The integration of Git functions within a custom step allows for someone unfamiliar with the Git process to interact with the versioning system easily. It also allows for a reusable way to embed Git updates into a scheduled SAS process without requiring manual command-line interaction.

```
...
/* Variable handling from previous step */
...

/* Configure SMTP to send email */
options emailsys=smtp
        emailhost="mailhost.example.com"
        emailport=25;

filename alertmail email
   to=("receiving-team@example.com")
   from="no-reply@example.com"
   subject="Model Retraining Alert: Updated Variable Importance";

/* Generate email body text*/
data _null_;
  file alertmail;

  put "Hello Team,";
  put;
  put "The model retraining process completed and the repository was updated.";
  put;
  put "Flow Details";
  put "------------";
  put "Flow Name    : &flow_name.";
  put "Model Name   : &model_name.";
  put "Run Time     : &run_date.";
  put "Push Status  : &push_status.";
  put "Commit ID    : &commit_id.";
  put "Commit Msg   : &commit_msg.";
  put;
  put "Updated Variable Importance";
  put "---------------------------";
  put "Variable                  Importance      Relative Importance";
  put "------------------------------------------------------------";
  put "&vi_1_name.              &vi_1_abs.          &vi_1_rel.";
  put "&vi_2_name.              &vi_2_abs.          &vi_2_rel.";
  put "&vi_3_name.              &vi_3_abs.          &vi_3_rel.";
  put "&vi_4_name.              &vi_4_abs.          &vi_4_rel.";
  put "&vi_5_name.              &vi_5_abs.          &vi_5_rel.";
  put;
  put "This message was sent automatically from SAS.";
run;
```

Program 4. EmailAlert.sas

Finally, we close with our Email Alert w/ Details node. This node acts as a notification or a human-in-the-loop action based on previous steps. At this point, the flow has either updated the repository or raised a hold flag. Using macro variables created in the previous steps, we can prompt an email alert accordingly.

CONCLUSION

Git has become an integral part of modern development, and SAS users increasingly operate in environments where reproducibility, traceability, and automation are just as crucial as analytical output.

Git Functions enhance the coding experience by enabling standard version-control operations to run directly within SAS programs.

The examples in this paper focus on simple repository automation, but you can extend the same pattern further. You can translate GIT_DIFF results into SAS datasets, trigger workflow flags, or even integrate the Visual Analytics API to build out full commit reports. In this way, Git Functions support not only version control, but also more structured change reporting and governance.

Most importantly, Git Functions provide a practical entry point. They let teams maintain their existing Git practices without abandoning them and do not replace the broader Git ecosystem. Instead, they allow version control to become part of the same SAS processes that already generate code, outputs, and analytical assets. More importantly, with these functions available across SAS 9.4 and SAS Viya environments, they provide a reusable pattern for teams looking to automate auditable, reproducible SAS workflows.

REFERENCES

Hemedinger, Chris. “Using Git with your SAS projects”. 2022. Available at https://github.com/sascommunities/git-workshop

Ollama. “Ollama”. 2026. Available at https://github.com/ollama/ollama

SAS Institute. “SAS® Viya® Platform Programming Documentation”. 2026.03. Available at https://documentation.sas.com/doc/en/pgmsascdc/v_073/pgmsaswlcm/home.htm

Seymour, Lleyton. “Git-Functions-in-SAS”. 2026. Available at https://github.com/lleytonse/Git-Functions-in-SAS

UCI Machine Learning Repository. “AI4I 2020 Predictive Maintenance Dataset [Dataset]”. 2020. Available at https://doi.org/10.24432/C5HS5C.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Lleyton Seymour
SAS Institute
Lleyton.Seymour@sas.com

Any brand and product names are trademarks of their respective companies.