We follow GitLab flow pretty closely. We work in parallel feature branches locally, test using GitLab CI, and merge those feature branches back into the master branch (usually called "build"). Our staging environment is regularly updated with the master branch for testing and demo purposes. Then, we create merge requests from the master branch to the production branch, which gets deployed to the production environment via release tags in Acquia Cloud.

This workflow allows us to deploy code to production according to pre-defined sprint schedules (usually during regular maintenance windows) while continuing development on the master branch behind the scenes. It also ensures a clear audit trail and optimum integrity of the final production-level product while reducing the complexity associated with traditional Git flow and GitHub flow.

While this doc isn't meant to replace the GitLab flow documentation, it does offer important guidance on UWEC flow, our specific flavor of GitLab flow.

Create a new branch for every issue

Before doing any work, create a new branch for each and every Issue. If you're not familiar, get an overview of tracking issues with GitLab Flow. This is the branch you'll do your work in. Only touch files that are directly related to your assigned Issue. If you discover a new bug/issue elsewhere while working on your assigned Issue, create a new Issue (and eventually a new branch) to prevent conflicts with your fellow developers' concurrent work on another Issue.

Doing your work

Remember to always push your changes when you're done working, even if the issue isn't resolved. Be sure to update the issue with the appropriate label and any notes/info another developer might need to pick up where you left off. Find out more about how to work with the living style guide.

- If the issue is ready to be merged, proceed to the creating a merge request
• If the issue is resolved, proceed to the closing an issue section. Remember, if there's a corresponding merge request for an issue, the issue should be resolved only after the merge request has been accepted and successfully merged.

Creating a merge request

Create a merge request back into the master branch (usually "master") when the Issue is addressed and all tests are passing, including accessibility checks. Remember to check the box that removes the feature branch once the merge request is accepted. Then:

1. Remove the "needs work" label and apply the "needs review" label to the originating issue;
2. Assign the Merge Request to a fellow dev for code review (to ensure quality we require all merges be reviewed and accepted by another developer, meaning you can't merge your own work);
3. Apply the "needs review" label to the new merge request once it's been assigned.

Accepting a merge request

Once you've picked up a merge request that's labeled "needs review", you'll need to test it thoroughly in your local environment to ensure it actually works as intended. Ensure it passes all CI tests and accessibility checks to WCAG 2.0 level AA or higher (these are mandatory for merges to master or release branches/tags).

If it doesn't pass all tests:

1. Note what needs work in the merge request;
2. Remove the "needs review" label from the merge request;
3. Apply the "needs work" label to the merge request.

If it passes all tests and is ready to be merged to the master branch:
1. Remove the WIP prefix if applicable;
2. Update the commit message;
3. Accept the merge request;
4. Remove the "needs review" label from the merge request;
5. Close the originating issue and remove the "needs review" label from it;
6. Create a new issue to update any relevant KB documentation (especially the Athena permissions matrix) if applicable.

**Closing an Issue**

Once an issue is ready to be closed, remove the "in progress", "needs work", or "needs review" labels. Be sure to close an issue only after any corresponding merge requests have been accepted and successfully merged.

**Deploying**

Note that for all projects (including those with primary development being done by Guest Developers or External Developers), only LTS developers with Maintainer permissions can accept merge requests to the production branch/release tag for deployment.

And remember, anytime you deploy something that may impact routing (or make a change to the environment that may do the same), use the project-specific tests (e.g. Drupal-specific Test URLs) to be sure all requests are getting where they're supposed to go.

Project-specific deploy documentation is found within the project's readme and/or wiki in GitLab:

- Athena: Readme and Wiki
- Captive portals: Wiki
- CommonSpot: Readme
- Drupal: Readme and Wiki
- Error pages: Readme
Milestones

Occasionally, sprint-specific milestones will be created. But generally, we use two types of milestones to organize issues—Active (e.g. "WebDev") and Backlog (e.g. "Backlog: Athena App Change Requests" and "Backlog: Drupal Change Requests").

Active Milestones

All active issues should be assigned to a workgroup-specific active milestone. Inactive issues should be added to a project-specific backlog milestone. Active milestones act as a team's sprint backlog, with each issue having a goal date and clear definition of done (DoD). Completed issues are then reviewed with stakeholders for final approval before being deployed to production via a release branch/tag.

- Active: AppDev
- Active: WebDev
- Active: WebStewards

Backlog Milestones

All feature requests, ideas, and other inactive issues should be added to a project-specific backlog milestone. Backlog milestones act as the product backlogs for each project, and should be continuously reprioritized and refined based on evolving scope and needs.

Please be sure to apply the appropriate site- or app-specific label(s) to each issue (e.g. "CE DRUPAL", "WWW DRUPAL", "Academic Programs: Athena", "Profiles:Athena", etc.).

- Backlog: Athena App Change Requests
Labels

Aside from project-specific informational labels (which are generally light-blue in color) like ~"WWW DRUPAL" and ~"Profiles: Athena", etc., we have a few workflow-specific labels that are vital to ensure a smooth and efficient flow.

"in progress" label

Use the "in progress" label to claim an issue and prevent a fellow dev from working on it. Don't start work on an issue that's marked as "in progress" unless you have permission from the current assignee. If you've taken the issue as far as you can or need to take a break from it, remove the "in progress" label and apply the "needs work" label. Be sure to add comments as appropriate so other devs know what's going on with the issue.

"needs work" label

The "needs work" label is a signal that the issue is ready to be worked on, or as a signal to the original dev that their merge request needs work before it can be accepted. Be sure to add comments as appropriate so other devs know what's going on with the issue. Check with someone first before digging into an issue that isn't labeled with "needs work" or "needs review".

"needs review" label

Use the "needs review" label as a signal to other devs that the issue is ready for review; these issues should almost always have a corresponding merge request. Use the "needs work" and "needs review" labels with merge requests as well. Once the issue is resolved or the merge request is merged, remove the "needs review" label.
"on hold" label

Use the "on hold" label to prevent/stop all work on an issue; these issues are usually being debated, are still being scoped, or have been dropped (but shouldn't be removed from the issues list for whatever reason). Don't work on these issues or you'll likely find yourself very frustrated and sad.