ENGR 301 Course Methodology

Introduction

This brief project management methodology is based on DevOps principles and describes the processes and practices which everyone should follow if they wish to have the advantage and benefit of optimally aligning their work with the traits assessed in the Performance Portfolios.

Key Principles

The foundations of the methodology are:

1. GitLab Workflow and the DevSecOps lifecycle, and
2. GitLab Flow (see also What is GitLab Flow?).

1. GitLab Workflow and the DevSecOps Lifecycle

These describe issue-tracker-based processes for managing work and the flow of work through either a *project* or *day-to-day operations* ("business as usual"). Issue tracking is paramount (emphasis mine):

> Issues are the first essential feature of the GitLab Workflow. **Always start a discussion with an issue**; it's the best way to track the evolution of a new idea.
>
> — *GitLab Workflow: An Overview*

The purpose of an Issue is to communicate an idea to others. When raising and working with Issues in this methodology:

- Titles are written in the imperative mood (i.e. the first word of the Issue Title should be a word such as "Add", "Extend", "Update", "Fix", etc.);
- Descriptions are written to provide more information about the idea or task, including internal and external URLs to relevant sources, empty descriptions must be avoided;
- The level of detail is sufficient such that: if anyone had to undertake, revisit or review the Issue after 12 or 24 weeks they would be able to understand it *without* having to perform additional work;
- Estimates of time required for task completion are included and time spent on a task is logged (see: GitLab quick actions);
- Comments explaining Issue events (add time spent, add a label, change a milestone, close the issue, etc.) are always written;
- Labels should be used with Issues to indicate type, scope and status (e.g. `To Do` or `Doing`), and

should be aligned with `type` and `scope` specified by the Conventional Commits standard (see below);
- GitLab's cross-linking mechanisms are used (see: Tutorial: It's all connected in GitLab)
- Screenshots of code and other text are avoided; instead, hyperlinks and fenced code blocks are used (see: GitLab Flavored Markdown).

Issues are the "quantum" of scope:

- Issues are the primary method through which work to be completed is defined;
- Issues should be decomposed into other Issues until each Issue is directed toward a single purpose, a "Task";
- Tasks should be of an estimated size such that they can be completed in one or two lab sessions;
- Issues are explicitly related to project or business goals, i.e. are explicitly related to specific goals or requirements via GitLab's features (Labels, Comments, Requirements, etc.)

Boards visually represent the flow of work through the project:

- Lists are scoped by Label (typically);
- Lists scoped by `To Do` and `Doing` Labels are created;
- Board is used to monitor and control Work in Progress.

Milestones are the preferred mechanism for time scheduling:

- Issues related to a common goal or purpose are associated with the Milestone;
- Start and end dates for the body of work represented by the Issues are set;
- Automatically display burn-down and burn-up charts for monitoring progress;
- Provides a board view to monitor Work in Progress;
- Provides aggregated time-tracking for time spent on Issues;
- Iterations are based on **Milestones**.

Merge Requests are used as a central place of record to link Issues with changes in the git repository:

- Merge Requests are *always* created at the same time as feature branches by using the `Create Merge Request` button in the parent Issue;
- There is typically a 1-to-1 relationship between **Issues** and Merge Requests;
- Merge Request comments are restricted to matters specific to commits, e.g. typographical errors, oversights, etc.
- Discussion about the task/idea takes place in the **Issue**.

2. GitLab Flow

GitLab Flow describes version control practices with git which are based on a trunk-based branching strategy. This methodology specifies a *simplified* GitLab Flow process in which the `main` branch is the deployment branch, i.e. there is no separate `production` branch. When, and if, the application becomes sufficiently complicated to warrant maintenance of a separate `production` branch then it can be created at that time.

GitLab Flow Best Practices (GFBP) are followed with modification to GFBP 8:

1. Use feature branches rather than direct commits on the main branch.
2. Test all commits, not only ones on the main branch.
3. Run every test on all commits.
4. Perform code reviews before merging into the main branch.
5. Deployments are automatic based on branches or tags.
6. Tags are set by the user, not by CI.
7. Releases are based on tags.
8. Pushed commits are ~~never~~ *seldom* rebased.
9. Everyone starts from main and targets main.
10. Fix bugs in main first and release branches second.
11. Commit messages reflect intent.

The modification to GFBP 8 recognises that *sometimes* the benefits of rebasing pushed commits outweighs the potential problems this practice may cause. It is expected that rebases of pushed commits are rare and prior notice is given to team-mates via Mattermost.

Good Practices are followed:

- When merging from remote to local, the fetch-and-rebase merge strategy is *always* used, i.e. `git pull --rebase` is *always* performed;
- Feature branches are *always* created at the same time as **Merge Requests** by using the `Create Merge Request` button in the parent Issue;
- Feature branches are *always* merged using GitLab's **Merge Request** web interface;
- **Merge Requests** use the merge commit strategy: a separate merge commit is created and the branch is *not* deleted to preserve commit history; this is the safest merge strategy for git beginners.

Commit Message Standards

The Conventional Commits commit message standard is adopted and automatically enforced as good practice for GFBP 11. The Conventional Commits specification is:

```
<type>(<scope>): <description>

[optional body]
```

where the `type` and `scope` elements are iteratively developed. Labels corresponding to `type` and `scope` are created in GitLab, when appropriate — which is *almost always*.

A *simplified* version of the Seven Rules of a Great Git Commit Message is followed:

1. Separate commit message subject from body with a blank line (consistent with the Conventional Commits specification);
2. Limit subject line length;
3. ~~Capitalize the subject line~~ (inconsistent with the Conventional Commits specification);
4. Do not end the subject line with a period;
5. Use the imperative mood in the subject line `<description>` field;
6. Wrap the body at a consistent line length;
7. Use the body to explain *what* and *why* vs. *how*.

The recommended line length for Rules (2) and (5) above is 80 characters.

The commit message standards are automatically enforced:

- Locally by gitlint https://jorisroovers.com/gitlint/ ; https://github.com/jorisroovers/gitlint;
- Remotely by configuring the GitLab repository commit message push rule regular expression in the settings section of the web interface.

Project Phases

This is a project management methodology and it is expected that the typical project phases, Initiation, Planning, Execution and Closing, will be observed. The time spent on the phases for a 12 calendar week project is expected to be:

Phase	Time	Schedule
Initiation	< 1 week	Project Week 1
Planning	2–3 weeks	Project Weeks 1–3
Execution	8 weeks	Project Weeks 4–12
Closing	< 1 week	Project Week 12

Roles and Responsibilities

The project groups have three development teams; each team has a leader. The group team structure will be reviewed at the project mid-point.

Project Sponsor: James Quilty (Course Coordinator); as primary stakeholder provides the overall vision and direction for the project, including project goals and deliverables.

Team Leads: are the main point of contact between the teams and the project sponsor; this is primarily a communication role, it is not a command role.

Development Teams: liberty is given to groups to forms teams as they see fit; guidance will be given in lectures and during the project regarding team composition and review.

Communication and Collaboration

The DevSecOps basis of this methodology is fundamentally a practice of collaboration between Development, Security and Operations. This course emphasises project management from a development and security perspective, but as the project is already in a deployable state there is opportunity for groups to practice operations. Large groups are unavoidable due to resource constraints, but confers one significant benefit: it allows the practice of collaboration across "silos" including development→development hand-offs.

The primary tools facilitating collaboration are the GitLab for project management and Mattermost for workplace chat. All information concerning the project must be stored in GitLab: **If it's not in GitLab, it doesn't exist**.

Quality Assurance

Quality assurance is *vital* for a project based on regulatory compliance. Testing is expected to be developed iteratively and the following forms of tests are within the methodological scope:

- static analysis (linting);
- unit, integration and user acceptance testing;
- deployment and operational testing.

The deliberate injection of faults into the system during operation is *encouraged*.

Tests must be *automatically* performed under this methodology both on the remote server by configuration of GitLab's CI/CD feature(s) as well as locally by configuration of pre-commit.

pre-commit

The pre-commit framework https://pre-commit.com has been adopted as good practice and is configured to run as a `lint` / `static-analysis` stage in the GitLab CI/CD pipeline.

pre-commit must be installed locally by running both `pre-commit install` and `pre-commit install --hook-type commit-msg`. The second command is required for the gitlint hook to operate correctly.

As there is one single repository for each group, all teams will share the same pre-commit and GitLab CI/CD configurations.

Iterative Approach

This methodology follows an iterative approach to identifying and completing the required work.

Groups are expected to:

- Choose an appropriate iteration length and start/end dates;
- Prefer **Milestones** over **Iterations** in GitLab for the first half of the project;
- Hold retrospectives at the end/beginning of each iteration;
- Practice their time management skills via this iterative approach by using the retrospectives to: (a) review the accuracy of the estimation(s) for the previous iteration, (b) estimate the time to complete work for the upcoming iterations (plural!) and (c) revise/refine work completion estimates (see the GitLab Workflow section above for GitLab Issue features supporting time management);

Risk is iteratively managed by *always* being a discussion topic at retrospectives: (a) identification of opportunities and threats, (b) deciding what actions to take and (c) reviewing the effectiveness of past risk management (i.e. determining residual risk).

Conclusion

The course methodology is directed to fostering contemporary project management practice for new practitioners. It provides a set of practices and processes which all members will follow in common:

1. A prescribed foundation of non-technical and technical practice through GitLab Workflow and GitLab Flow;
2. Well-defined roles and responsibilities of Dev and Sec-Ops;
3. Practices of collaboration and communication between teams;
4. Technical and non-technical standards, enforced automatically;
5. Quality assurance through automated testing in every environment;
6. An iterative approach to completion of work.

Through these steps, contemporary *best practice* is adopted and adapted to our local context to become *good practice*.

Good luck!

Author: James Quilty
Affiliation: School of Engineering and Computer Science, Victoria University of Wellington
Last updated: 2024-02-29