Software Development Process (SDP)

Please remove help/template/example texts and tips from your submission (everything but headers).

*The goal of the SDP is to establish a clear and well-defined framework that guides the development team throughout the software development lifecycle. This document serves as a reference and standard for all team members involved in the project, outlining the processes, methodologies, and best practices to be followed during the software development journey.*

[Principles](#_2s13ohyeqgb7)

[Process](#_dv87m3vones8)

[Roles](#_r4ojepvcw4mh)

[Tooling](#_gtpscwd44yls)

[Definition of Done (DoD)](#_gtqo52h8bto2)

[Release Cycle](#_3elqxufgawry)

[Environments](#_nqxqf74b5aov)

# **Principles**

*Set of principles that the engineering team agrees on. Here’s an example:*

* *We are responsive with our asynchronous communication and answer within 24 hours.*
* *We use a Kanban board to continuously work on the backlog (no sprints).*
* *The backlog will always have work items ready for the next two weeks at the minimum.*
* *A work item needs a user story (or motivation), technical description, dependencies, estimate, and acceptance criteria before anyone can work on it.*
* *All changes need to be developed in a separate git branch.*
* *Once the feature, bug, or refactoring is done, a Pull Request is created.*
* *The Pull Request has to be reviewed by at least one team member before being merged.*
* *The Pull Request needs to comply with the Definition of Done (see later) and should be linked to the corresponding work item.*
* *Work items cannot take more than one day to be completed, but need to encompass a meaningful chunk of work. If a work item is too large (e.g., a larger user story), divide into smaller work items.*

# **Process**

*Explain the software development process that you plan to follow. This will highly influence how you organize yourselves (or the other way around). You can use a traditional process such as Agile, Waterfall, Scrum, Kanban, eXtreme Programming, or a mix of what works for you.*

Typically:

* *Asynchronous, part-time teams could benefit from a Kanban approach with some elements of Agile and DevOps, as they cannot meet everyday, don’t work at the same time, and have changing schedules.*
* *Simpler, very straightforward software projects with clear, stable requirements can benefit from a waterfall approach.*
* *Most teams will figure out something that works for them.*

*Members of the team must agree on the process, implement it, make themselves accountable, but be willing to change or improve it (flexibility). This is much less relevant for one-person teams.*

*Research has shown that teams are much more productive when co-located (physically in the same space) but that’s not always feasible. Responsibility, accountability, and proper processes help with asynchronous online work.*

*Example of process:*

* *Backlog and Planning (1/week)*
* *Kanban Board (To Do - In Progress - In Review - Done)*
* *Demo/Review with Stakeholders (1/week)*

# **Roles**

*Explain the different roles on your team (e.g., project management, product management, scrum master, design, frontend, backend, data engineering, data analysis, feature subset X, feature subset Y, etc.) and who will be in charge of it. For your CS Capstone class, it’s good to have everyone dabble in everything, at least to some extent (everyone is expected to code). Setting up a rota is encouraged.*

*This will also dictate who will be in charge of running meetings, of managing the backlog, of making sure everyone is responsive, etc. Accountability is key. There’s no place to hide.*

# **Tooling**

*List all the software tools that you will use to develop your program. This is particularly useful for teams to align their development workflow. This is different from infrastructure and environments.*

*Note that the tooling might change depending on your software design and architecture decisions. You might not use the same linting/formatting or testing tools if you're using Java, Python, or a specific web dev framework or database.*

*This is not the same as your core tech stack (programming languages, databases, frameworks, etc.)*

| **Version Control** | GitHub (recommended), GitLab, BitBucket, Unity Version Control, etc. |
| --- | --- |
| **Project Management** | GitHub Issues and Projects (recommended), GitLab equivalent, Trello, Jira, etc. |
| **Documentation** | <https://github.com/withastro/starlight> (recommended), Astro, Sphinx, Read the Docs, Markdoc, Confluence, Notion, GitHub Wiki, README, etc. |
| **Test Framework** | Playwright (recommended), Jest, etc. |
| **Linting and Formatting** | Prettier (recommended), ESLint, Pylint, autopep8, Black Formatter, etc. |
| **CI/CD** | GitHub Actions (recommended), CircleCI, Jenkins, etc. |
| **IDE** | Visual Studio Code (recommended), Neovim, IntelliJ, PyCharm, etc. |
| **Graphic Design** | Figma, Pen and Paper, Excalidraw, [Draw.io](http://draw.io), Google Slides, etc. |
| **Others** | AI assistants, creation of code executable, monitoring, code analysis, etc. |

# **Definition of Done (DoD)**

*Describe what criteria should be met before an issue/task/user story is considered* ***completed****. Here’s an non-exhaustive example:*

* *Acceptance criteria are validated*
* *Changes are merged (to main branch)*
* *Unit, integration, and smoke tests are successful*
* *Changes are implemented in all components (e.g., backend, frontend, libraries, …) (where it makes sense)*
* *No regressions*
* *Documentation is updated, incl. deployment instructions if any*
* *Release notes are updated*
* *Breaking changes are evaluated/avoided*
* *Changes are deployed to staging*
* *Demo is prepared for next stakeholder meeting*

*Depending on the stage of your project/product/software, some of these criteria might create more overhead and hence less benefits. Feel free to revisit (add/remove/change) your DoD as your project progresses.*

# **Release Cycle**

*Describe the release cycle that would work for your team and business.*

*Here’s a simple example:*

* *Automatically deploy to staging every merge to main branch*
* *Deploy to production every release*
* *Release every three months*
* *Use semantic versioning* MAJOR.minor.patch
  + *Increment the* minor *version for new features*
  + *Increment the* patch *version for bug fixes*
  + *Increment the* major *version for breaking API changes*

*Until the API is stable,* major *should be* 0*.*

*Feel free to use another form of versioning if it makes sense for your project.*

# **Environments**

*List your options for infrastructure, the different environments, the deployment schedule, what they should be used for, and how they should be monitored. For example, you could automatically deploy to staging changes to your main branch after a pull request, and deploy to production every time you create a new release.*

| **Environment** | **Infrastructure** | **Deployment** | **What is it for?** | **Monitoring** |
| --- | --- | --- | --- | --- |
| Production | AWS through CI/CD | Release | Sleeping well at night | Prometheus, Grafana, Sentry |
| Staging (Test) | [Render](http://render.com) through CI/CD | PR | New unreleased features and integration tests | Sentry |
| Dev | Local (macOS and Windows) | Commit | Development and unit tests | N/A |

*Note that in the example above, the infrastructure is different in staging and production. You can (and probably should) totally use the same provider but different instances for your different environments.*

*The requirements for these environments can be discussed and decided in your software design and architecture document.*