NSDF Software Development Life Cycle (SDLC) Procedures

unknown

Feb 03, 2022

CONTENTS

CHAPTER

ONE

DOCUMENTS

1.1 Introduction

The scope of this document is to create and document guidelines, norms, and procedures for the *software engineering* aspects of development, evolution, and long-term operation of the NSDF software stack, in particular regarding::

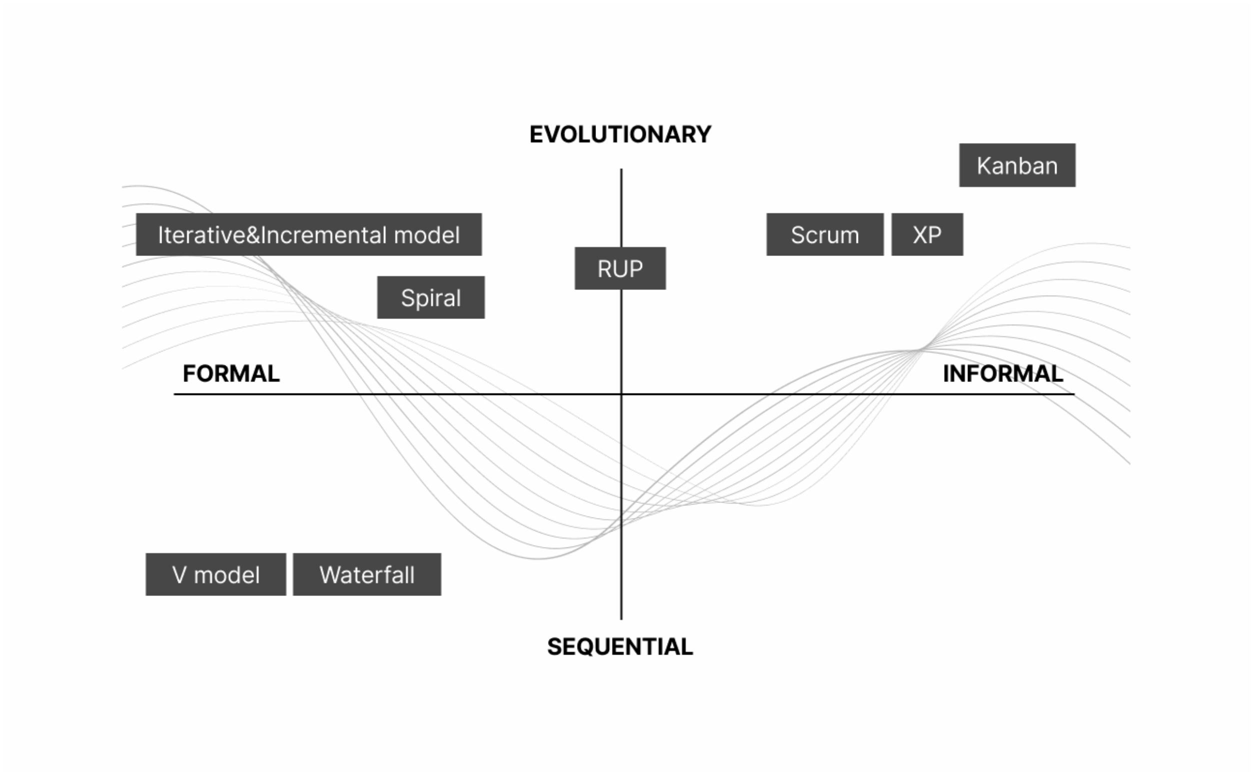
- Use of repositories, branching, and versioning methodologies
- Use of programming languages and frameworks
- System and code documentation, and their continued sustainment
- Style guidelines for user interfaces, code construction
- Code review procedures
- Deployment staging procedures
- Test requirements: Unit, Regression, Integration, and Top-level validation approaches
- Development project methodologies (e.g. agile practices)
- Continuous integration and deployment (CI/CD) practices
- Package management practices
- Container management practices
- Change management recommendations

Authors in alphabetical order:

Name	Email
Daniel Balouek	daniel.balouek@utah.edu
Kevin Coakley	kcoakley@ucsd.edu
Jakob Luettgau	jluettga@utk.edu
Paula Olaya	polaya@vols.utk.edu
Giorgio Scorzelli	scrgiorgio@gmail.com
Glenn Tarcea	gtarcea@umich.edu
Naweiluo Zhou	naweiluo.zhou@utk.edu

This is a guide to software development at the NSDF. It both serves as a source of information for exactly how NSDF works, and as a basis for discussions and reaching consensus about how to_ develop software_.

A *Software Development Life Cycle* (SDLC) is a methodology followed to create *high-quality software*. By adhering to a standard set of tools, processes, and duties, a software development team can build, design, and develop products that meet or exceed their clients' expectations.



The most famous SDLC models are:

- Waterfall: Follows a sequential model of phases, each of which has its tasks and objectives
- Cleanroom: A process model that removes defects before they cause serious issues
- Incremental: Requirements are divided into multiple standalone modules
- V-Model: Processes are executed sequentially in a V-shape i.e. each step comes with its testing phase
- Prototype: A working replication of the product is used to evaluate developer proposals
- Big Bang: Requires very little planning and has no formal procedures; however, it's a high-risk model
- Agile: Uses cyclical, iterative progression to produce working software

This document specifies the software development procedures for the NSDF project and includes all development procedures between high-level requirements and either software release or the initiation of a DevOps deployment process.

1.1.1 Software checklist

In this section, we provide a short checklist for software projects, and the rest of this document elaborates on the various points in this list.

The bare minimum that every NSDF software project should do is:

- choose and include an open-source license
- use version control to enable collaborative developing
- use a publicly accessible version-controlled repository
- add a README.md file describing the project. This file is targeted towards developers. Keeping basic documentation in README.md can be useful for other developers to track steps and design decisions. Therefore it is convenient to create it from the beginning of the project when initializing a git repository.

NSDF also recommends doing the following, from the start of the project:

- use **code quality tools**
- use **testing**
- use **standards** (protocols, conventions, tools, etc.)
- Release user and development documentation
- Provide **issue trackers**
- Make the software citable adding a **DOI**
- Release the software to a public registry
- Add a public channel for communication
- Implement and add a **code of conduct**
  - A code of conduct defines standards for how to engage in a community.
  - It signals an inclusive environment that respects all contributions.
  - It also outlines procedures for addressing problems between members of your project's community.
  - See https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-code-of-conduct-to-your-project
- Add a contribution guideline document.

1.1.2 Programming languages and conventions

From the beginning of the project, a decision on the code style has to be made and then should be documented.

Not having a documented code style will highly increase the chance of inconsistent style across the codebase, even when only one developer writes code.

The NSDF should have a sane suggestion of coding style for each programming language we use. Coding styles are about consistency and making a choice, and not so much about the superiority of one style over the other

If your programming language supports namespaces, use `nsdf.*` to clarify the origin of the software.

NSDF wants to limit the development to a few core languages and frameworks.

At the NSDF we prefer C++, Python, Go, and JavaScript.

1.1.2.1 C/C++

C/C++ is the NSDF programming language for fast and core services such as the visualization and low-level storage of multi-resolution data.

NSDF C/C++ environment is built on:

- C++ version: C++11 or C++17
- Visual Studio for Windows, gcc/clang on other platforms
- Code style: CppCoreGuidelines
- Minimal self-contained dependencies (e.g. STL, boost, etc.)
- Cross-platform make tool: CMake

Libraries:

- Open MPI . to enable parallelism
- Boost C++ is a popular collection of peer-reviewed, free, open-source C++ libraries.
  - Code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.
  - Maybe can hamper readability excessively "functional" style of programming
- JSON for Modern C++
- hdf5-cpp : The popular HDF5 binary format C++ interface.
- ZeroMQ: lower level flexible communication library with a unified interface for message passing between threads and processes, but also between separate machines via TCP.

1.1.2.2 Python

Python is the NSDF *dynamic language of choice*.

We use it for data analysis and data science projects using the SciPy stack and Jupyter notebooks, and for many other types of projects: workflow management, visualization, web-based tools, etc.. It is not the language of maximum performance, although in many cases performance-critical components can be easily replaced by modules written in faster, compiled languages like C/C++ or CPython.

Python is very flexible and the most used programming language for scientific applications: a large number of useful frameworks and libraries are written in Python. Python allows easy integration with low-level bindings (e.g., C/C++) if efficiency is critical.

NSDF Python environment is built on:

- Python 3.7+
- Web applications: Django, Flask
- Packaging: PiPy, Manager: pip (would avoid conda when possible)
- Other services: Tornado
- Templating: Jinja
- Code style: PEP 8

Notebooks:

- Client-side: Jupyter Notebook
- Server-side: JupyterLab

IDE:

- JetBrains PyCharm

Core scientific packages:

- NumPy
- https://scipy.org/
- Pandas
- Scikit-learn
- Dask

Visualization packages

- Matplotlib. the standard in scientific visualization. It supports plotting through the `pyplot` submodule. It is highly customizable and runs natively on many platforms, making it compatible with all major OSes and environments. It supports most sources of data, including native Python objects, NumPy, and Pandas.
- Seaborn is a Python visualization library based on Matplotlib and aimed towards statistical analysis. It supports NumPy, pandas, scipy, and statmodels.
- bokeh is Interactive Web Plotting for Python.
- Plotly is a platform for interactive plotting through a web browser, including in Jupyter notebooks.

Parallelization packages:

- The multiprocessing module is the standard way to do parallel executions in one or multiple machines, it circumvents the GIL by creating multiple Python processes.
- in Python 3 is the concurrent.futures module
- See Using IPython for parallel computing

1.1.2.3 Go

Go is a statically typed, compiled programming language that is open-sourced and maintained by Google. Go uses a garbage collector to handle memory leaks.

Go is very fast and mostly used for server-side applications.

NSDF Go environment is built on:

- Go 1.17+ (Recommend upgrading to the latest version whenever it becomes available. Versions are backward compatible. Version 1.18 will release “generics” for Go)
- Code style: The Go community has standardized around the “go fmt” tool. All code should be run through the “go fmt“ tool to properly format it.
- Dependencies Management: Use Go Modules for dependencies management.
- Builds: Even though Go has a toolchain for builds it is recommended that a Makefile be created to hide the options.
- Background “Daemon” processes: This is an area it is too easy to get wrong. Instead, use supervisord to handle background processes.

- Web services: There are many different web service frameworks available. I've standardized on using Echo (echo.labstack.com). It has less boilerplate than the standard library and decent documentation.
- Databases: Go has a standard DB library. I recommend using either the "sqlx" package or gorm. I expect once Go 1.18 is released that many of the packages that currently rely on reflection will see a lot of changes.

1.1.2.4 JavaScript

JavaScript is the programming language for the World Wide Web, alongside HTML and CSS. All web browsers have a dedicated JavaScript engine to execute the code on users' devices.

On the server side, there is `Node.js`, an open-source cross-platform JavaScript runtime environment with an event-driven architecture capable of asynchronous I/O.

NSDF JavaScript environment is built on:

- ECMAScript 6
- Packaging: NPM, Resolver: Yarn (faster)/npm
- Cross-compiler: Babel
- Code style: Airbnb
- MVC/SPA Clientside frameworks: React, Angular, Vue
- Angular is an application framework by Google written in TypeScript.
- React is a library that can be used to create interactive User Interfaces by combining components. It is developed by Facebook.
- Vue.js is an open-source JavaScript framework for building user interfaces.

Security Considerations:

- XSS

1.1.2.5 Awesome List

On GitHub, there is a concept of an _*awesome list*, that collects awesome libraries and tools on some topic. For instance, here is a subset:

- Python: https://github.com/vinta/awesome-python
- C++ https://github.com/fffaraz/awesome-cpp
- Go https://github.com/avelino/awesome-go

1.1.3 Links/Bibliography

List:

- README.md template · GitHub
- The art of ReadMe

1.2 Continuous Integration

Overview of section contents:

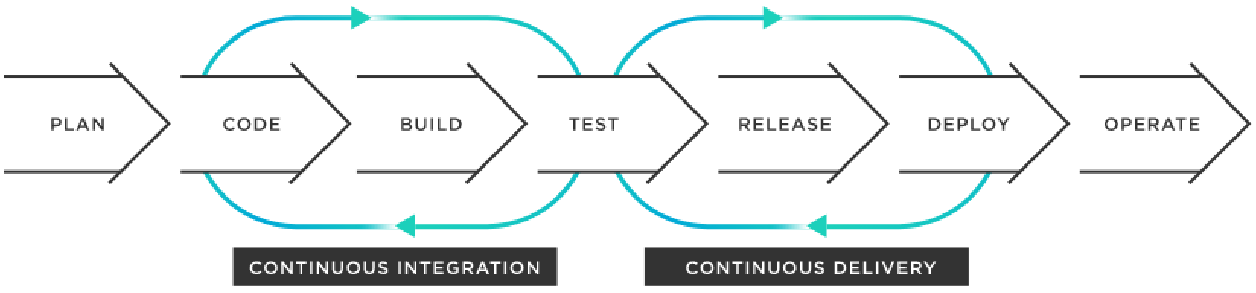
Section	Description
Version control	standards to use Git to enable code version control for public and private repository
Git branches	different Git branches of the same project to enable multiple developments in parallel
Git workflows	NSDF recommends GitHub Flow for the complex repositories and Git Flow for all other cases.
Semantic Versioning	Industry standards to name software versions
Code reviews	standardization of code review to improve software quality

---

In 1994, the term "*Continuous Integration*" (CI) was introduced by American software engineer Grady Booch, who is best known for developing the *Unified Modeling Language* (UML) but also for his innovative work in software architecture and collaborative developer environments.

Continuous integration has been regularly employed since 1997 and is now widely accepted as the best practice for software development. Although the process of continuous integration may look a bit different today than it did 20 years ago, the theory behind it nonetheless remains the same.

Continuous integration is the practice of continuously integrating code changes from different developers working on the same code into a single software project. This integration is an ongoing and continuous process to ensure that all changes are properly recorded. This strategy is an essential part of an *Agile Software Development System*, which is built around the concept of collaboration, designing for scale, and building sustainability.



Continuous Integration / Continuous Delivery (described in the following section) is an *eight-step*, agile process that ensures fast, effective, and stable software development.

- ***Plan***: Changes to the application are planned by the product team. This could include bug fixes, performance enhancements, or new features to be added to the application.
- ***Code***: The developers code the software on their local machines. Each developer has a specific part of the system to develop or a bug to resolve.
- ***Build***: the new code is submitted to the code repository and the application is compiled.
- ***Test***: tests check the functionality and usability of the code. Automated testing is used to ensure that the new code doesn't interfere with other parts of the package.
- ***Release***: the code is merged and can be set to an automated release, pending approval.
- ***Deploy***: The code is automatically deployed to production.
- ***Operate***: the new code can be operated within the production environment.
- ***Monitor***: Application performance is continually monitored to find bugs and identify performance problems.

Continuous integration will help NSDF to ensure:

- *Time savings:* it removes any double-up on tasks, automating testing and merging processes
- A *more robust product:* regular testing implies fewer bugs and fixes.
- *Increased communication*: code sharing increases the speed and efficiency of communication
- *Faster software releases*: changes can be fixed, tested and rolled out in tight timeframes.
- *Limit integration conflicts*: regular updates minimize potential conflicts; problems can be quickly identified
- *Modular code: _facilitate the development of* _less complex code

1.2.1 Version control

Version Control enables developers to keep track of the revisions in software development projects and allows them to work together on those projects.

Version control makes it easier to work on the same code simultaneously, while everyone still has a well-defined version of the software.

Also, Version Control makes it easier to integrate with other software that supports *modern software development*, such as testing (continuous integration, automatically run tests, build documentation, check code style, integration with bug-tracker, code review infrastructure, comment on code, etc.).

NSDF repositories should preferably be public from the start; but **we tolerate private repositories,** to be switched to a public status when they reach a certain level of *curation*.

To prevent *private repositories* from remaining unnecessarily private forever please add a brief statement in the README of your repository, clarifying:

- Why is this repository private?
- On which date can this repository be made public?
- Who should be consulted if we would like to make the repository public in the future?

The official NSDF repository is https://github.com/nsdf-fabric

NSDF recommends the use of Git (versus CVS, Apache Subversion, Mercurial, etc.) since it has a lot of improvements over its competitors, and it's perfect for NSDF distributed projects. Github provides a way to communicate in a more structured way, such as in code reviews, commits, and issues.

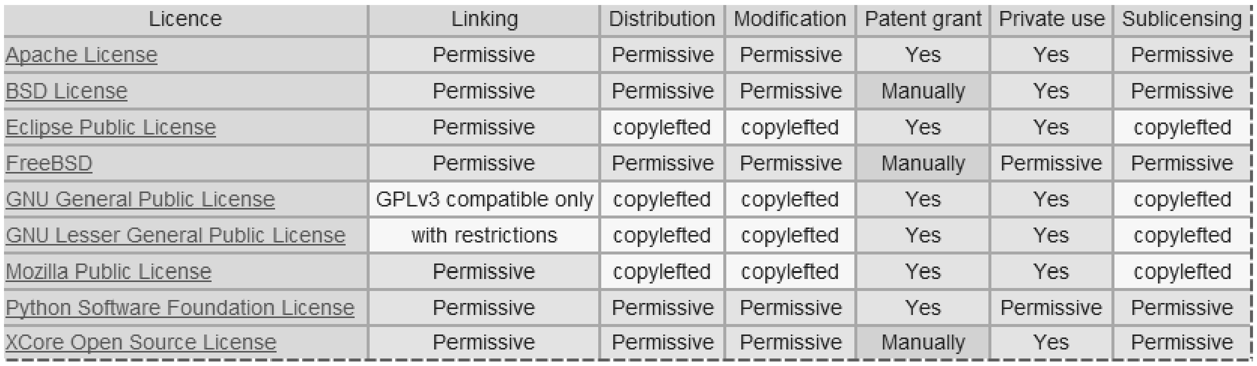
One downside of Git is that it can be sometimes difficult to explain to a non-expert, and there is likely to be a slow down in production as programmers adapt to it. But, once it is learned, the whole software cycle speed will increase.

To give proper `push` /`pull` permissions on NSDF repositories (ref. https://github.com/nsdf-fabric), we recommend adding a new `SSH key` for each user, to avoid typing passwords repeatedly. The procedure is just a few-minutes time consuming (ref Adding a new SSH key to your GitHub account) and it consists of:

- Generate a new SSH key (or recycle an existing one)
- enter the "GitHub settings" section and select "SSH and GPG keys"
- paste the SSH

In creating a new NSDF repository some suggestions are:

- include a `.gitignore` file
- Add a `README.md` file explaining what the project does, why it is useful, how users can get started, how users can help, and who maintains/contributes to the project
- Add a `LICENSE` file. It tells others if and how they can use, change, or distribute your software. It protects both sides from legal troubles. The license should be included in a `LICENSE` file at the root of the project directory as per standard practice.
  - See https://choosealicense.com/ for deciding what license.
- be careful to *commit messages:* they are the way for other developers to understand changes in the codebase. It is very important to explain the why behind implementation choices.



Licence	Linking	Distribution	Modification	Patent grant	Private use	Sublicensing
Apache License	Permissive	Permissive	Permissive	Yes	Yes	Permissive
BSD License	Permissive	Permissive	Permissive	Manually	Yes	Permissive
Eclipse Public License	Permissive	copylefted	copylefted	Yes	Yes	copylefted
FreeBSD	Permissive	Permissive	Permissive	Manually	Permissive	Permissive
GNU General Public License	GPLv3 compatible only	copylefted	copylefted	Yes	Yes	copylefted
GNU Lesser General Public License	with restrictions	copylefted	copylefted	Yes	Yes	copylefted
Mozilla Public License	Permissive	copylefted	copylefted	Yes	Yes	copylefted
Python Software Foundation License	Permissive	Permissive	Permissive	Yes	Permissive	Permissive
XCore Open Source License	Permissive	Permissive	Permissive	Manually	Yes	Permissive

Each source file of any NSDF software should start with the following copyright statement at the top :

```
Copyright NSDF and Licensed under the XXX License, version YYYY.
See LICENSE for details.
```

The same notice should be somewhere in your `README` file, which should also contain an overview of dependencies and which licenses they are under.

We also suggest the creation of DOIs for all NSDF software. Software citation is important as scientific research becomes more open and more digital. Advantages to making software citable include:

- Gives credit to the software developers
- Supports scientific transparency
- Improves reproducibility of research that relies on the software
- Helps the community by enabling reuse of your code and methods
- Supports FAIR software principles

NSDF software should contain sufficient information for others to be able to cite its software, such as authors, title, version, and DOI.

GitHub has default integration with Zenodo by CERN. Any time a new release is made, the repository will be automatically archived and issued a new DOI by Zenodo. From this article:

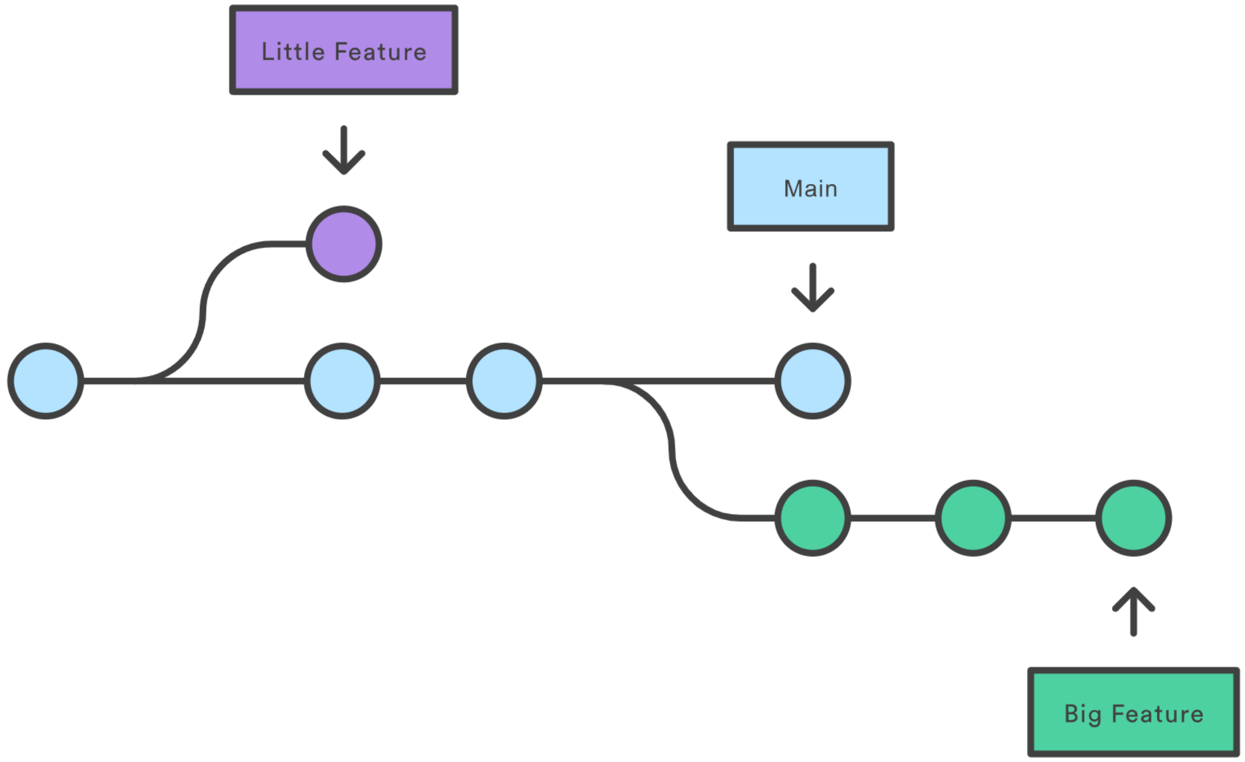
> For Open Science, it is important to cite the software you use in your research. Particularly, you should cite any software that made a significant or unique impact on your work. Modern research relies heavily on computerized data analysis, and we should elevate its standing to a core research activity with data and software as prime research artifacts. Steps must be taken to preserve and cite software in a sustainable, identifiable, and simple way. This is how digital repositories like Zenodo can help.

1.2.2 Git branches

In Git, branches are a part of your everyday development process.

Git branches are effectively a pointer to a snapshot of changes. When there is a new feature or bug fixing, the developer creates a new branch to *encapsulate* the changes.

This simple process makes it harder for unstable code to get merged into the main code base and encourages the cleaning up before merging into the main branch.



As an example, in the above figure, two isolated lines of development are shown:

- one purple line: the new branch is created for developing a “little” feature
- one green line: the new branch is created for a longer-running “big” feature
- meanwhile, the main branch is unaffected

By developing the new features in branches, the work can proceed in parallel, keeping also the main branch isolated from *still-in-development* (and possibly buggy) code.

1.2.3 Git workflows

When working on a Git-managed project, we need to make sure the team agrees on what workflow to adopt i.e. how the flow of changes will be applied.

A Git Workflow is a recipe or recommendation for how to use Git to accomplish work consistently and productively. Given Git's focus on flexibility, there is no standardized way to interact with Git.

For a full discussion on workflows, advantages and cons, please read "*Appendix I. Git Workflows*".

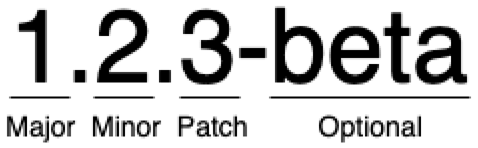
NSDF recommends GitHub Flow for its more complex repositories and Git Flow for all other cases.

1.2.4 Semantic Versioning

Semantic Versioning is the most accepted and used way to add numbers to software versions. It is a way of communicating the impact of changes in the software on users.

Semantic versioning is an industry-standard for software versioning. The overall idea is that given a version number "major.minor.patch", we will increment the:

- `MAJOR` version when there are breaking/incompatible API changes
- `MINOR` version when there is added functionality/features in a backward-compatible manner
- `PATCH` version when there are backward-compatible bug fixes



Major version zero `0.y.z` is for initial development. Anything may change at any time. The public API should not be considered stable.

Version 1.0.0 defines the first public API.

Prerelease information may be appended, separated by a dash.

The prereleases can be versioned too (eg. 1.2.3-beta.1, 1.2.3-beta.2, etc). Typical prerelease stages include:

- *alpha*: internal testing, may not be feature-complete
- *beta*: external testing, should be feature complete
- *release candidate* (RC): this code will be shipped unless bugs are found

1.2.5 Code reviews

At the NSDF, we value software quality.

Higher quality software has fewer defects, better security, and better performance, which leads to users who can work more effectively.

Code reviews are an effective method for improving software quality. Code reviews are used to ensure consistency and quality across a project. They are a key aspect of the success of any development team.

Code review also improves the development process. By reviewing new additions for quality, less technical debt is accumulated, which helps the *long-term maintainability* of the code. Reviewing lets developers learn from each other and spreads the knowledge of the code around the team. It is also a good means of getting new developers up to speed.

Code reviews are effective because they put a double control on the code, and force authors to explain their purposes in clear language. It will be less probable to commit overly-complicated code: if the reviewer is not able to understand it, he will refuse the _pull request _asking for clarifications.

This is also the reason why code reviews are also an incredible ource of frustration \and delays.

In this 2021 DevOps Survey code quality was one of the top reasons to choose DevOps, but, at the same time, code reviews were one of the top four pain points (with testing, planning, and code development.

The main purpose of a code review is to find issues or defects in a piece of code. These issues then need to be communicated back to the developer who proposed the change, so that they can be fixed. The goal of a code review is not to provide criticism of a piece of code, or even worse, the person who wrote it. The goal is to help create an improved version.

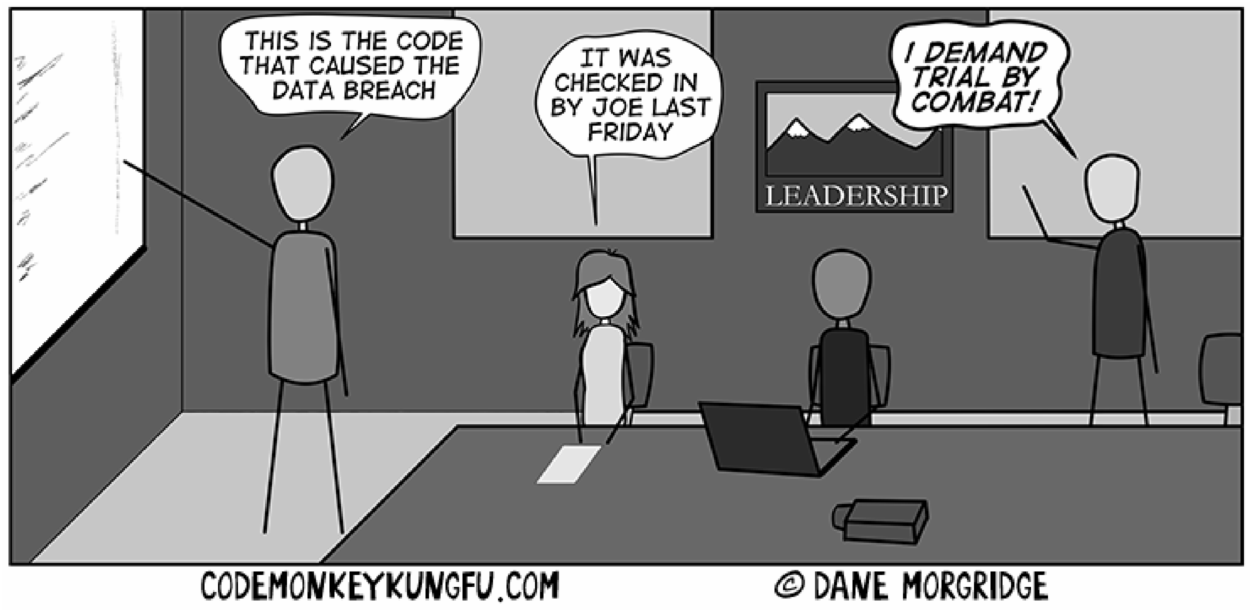
NSDF recommends that every change must be reviewed, and every change must be approved.

The NSDF reviewers should look at (copying from Google Code Review Developer Guide):

What	Description
Design	The overall design of the code should make sense within the code base and integrate well with the system
Functionality	The code must meet the needs of the *user story* and should consider edge cases too.
Complexity	The code must not be too complex. Pieces of evidence of overcomplexities are: "*code can't be understood quickly"* or "*developers are likely to introduce bugs when trying to modify the code*". Also, the code must not be over-engineered i.e. it is trying to solve preemptively non-existent problems.
Naming	Code must use clear names for variables, classes, methods.
Tests	Code must include unit tests and other types of tests as appropriate. Tests must be correct, concise, useful, sensible, and cover an *acceptable amount* of the code.
Comments	The code must be commended adequately and comments must not be *verbose.*
Style	The code must follow project style guidelines i.e. it must be consistent with the surrounding code.
Documentation	In case of significant changes, the documentation must be updated as well.

Major problems regarding code reviews are:

- Code reviews are *impersonal*: they are almost always conducted via _online text _communications. This can result in communications challenges: developers could be _protective _of their work, and some comments can be perceived as offensive
- code reviews can become sometimes too a *nitpicking process*, causing frustration
- code reviews could take too long resulting in a general slowdown of the development
- Code reviews are highly subjective, based on the assigned reviewers.
-


THIS IS THE CODE THAT CAUSED THE DATA BREACH
IT WAS CHECKED IN BY JOE LAST FRIDAY
I DEMAND TRIAL BY COMBAT!
LEADERSHIP
CODEMONKEYKUNGFU.COM
© DANE MORGRIDGE

To solve these problems NSDF recommends some best practices:

- convert problematic code reviews to be on-person or on conference-call
- 24/48 hours is the maximum time a code review should take. And the author's response to observations should be fast as well
- as accept an _almost-perfect _code review and ask the author to do the minor fixes subsequently.
- don't send large code reviews
- do not mix changes belonging to different problems.
- In case of emergencies (like a demo, deadlines, etc) some rules can be _relaxed. _Keep track of this relaxation by creating an ad-hoc branch
- use courtesy when reviewing code and providing guidance. This is a process built on mutual respect, and is not intended to shame people or negatively express dissatisfaction (citing Microsoft guidelines "*Don't criticize the submitter, point out flaws in the code*").
- integrate code reviews in the CI pipeline (e.g. "*Git pull requests*") to keep the history

1.2.6 Links/Bibliography

List:

- https://www.tibco.com/reference-center/what-is-continuous-integration
- Git Branch | Atlassian Git Tutorial
- Comparing workflows - Atlassian Git Tutorial
- The Best Branching Strategies For High-Velocity Development
- Gitflow workflow vs Feature Branch workflow
- Using Gitflow with the GitHub Fork & Pull Model
- What are GitLab Flow best practices?
- Semantic Versioning 2.0.0

- Google Code Review Developer Guide
- How We Do Code Review (Microsoft)
- I've code reviewed over 750 pull requests at Amazon
- Google Code Review Developer Guide
- Code Review Best Practices.
- Designing a rubric for feedback on code quality in programming courses | Proceedings of the 16th Koli Calling International Conference on Computing Education Research
- Why code review beats testing: evidence from decades of programming research | Kevin Burke
- Best Practices for Code Review | SmartBear
- Code Reviews: Just Do It

1.3 Continuous Delivery and Deployment

Overview of section contents:

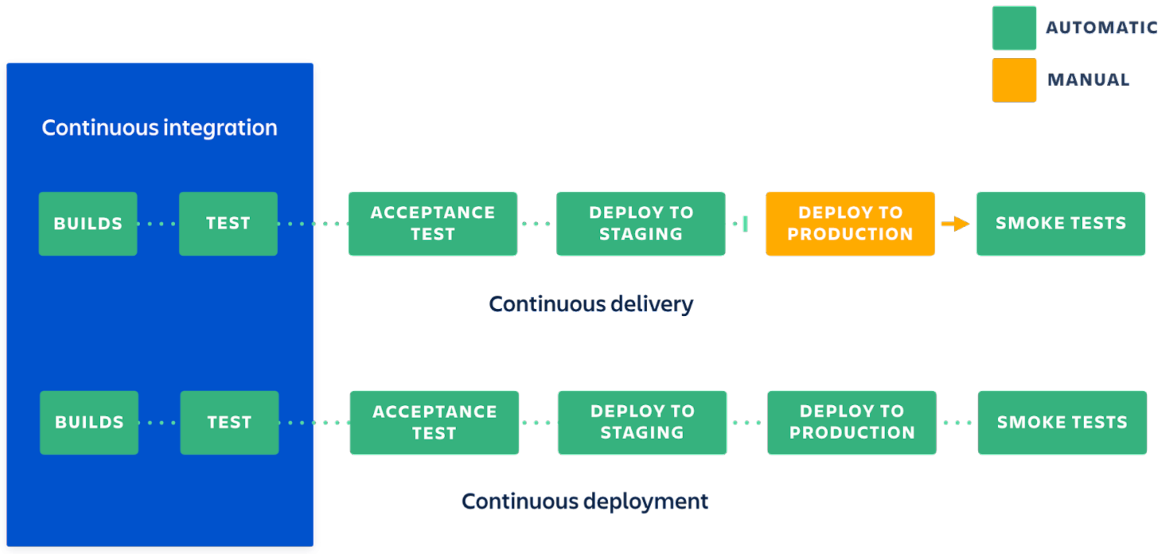
Section	Descroption
Cloud deployment	The container as service is favored in NSDF
Artifacts Repositories	NSDF adopts the most widely-used channels to share NSDF artifacts based on C/C++, Python, Docker, and Kubernetes.
Deployment staging	Helm to provide development staging environment for internal usage
Docker Containerization	NSDF suggests using Docker as the standard containerization tool and UDocker as an alternative funder certain special situations
Kubernetes Orchestration	Orchestration workflow
HPC deployment	Recommendation of container engines for HPC

Continuous Deployment (CD) is a software development discipline where _software is built in such a way that it can be released to production at any time and minimize the time between production iterations (see https://martinfowler.com/bliki/ContinuousDelivery.html)

In literature there is usually a clear distinction between the delivery and the deployment phase:

- **Continuous Delivery** means that artifacts are built and made ready to be deployed. But they will not be deployed without a manual decision by a human being.

- **Continuous Deployment** implies all processes are automated, and a single commit triggers an automated pipeline that will eventually bring a new version of your application to the production environment without any human intervention



While many companies practice *Continuous Delivery*, few embrace Continuous Deployment because it's riskier: anyone could introduce a bug into production with a simple commit, and there is the need to introduce additional processes to reduce this risk.

Automation is a key driver of productivity in CD. A battery of automated tests must be programmed to verify that new commits are functional before they are automated, and additional tools are required to abort the deployment process and trigger human intervention when the tests reveal *lower-than-expected quality* results or outcomes.

Together with many indubitably advantages, CD adds an *element of risk* to the software release process, as frequent commits may introduce bugs to the live environment.

Organizations that implement continuous deployment must therefore develop *real-time monitoring* capabilities of the live environment to rapidly discover and address any technical issues that occur after new releases (see "*Continuous Monitoring* chapter").

To sum up, the major benefits of CD are:

- *Maintain Capability for Quick Releases*. It enables teams to get their new releases into the production environment as quickly as possible. With CD, we can roll out several deployments per day.
- *Rapid Feedback Loop*. Developers can assess the impact of new changes by monitoring the deployment and watching users' behavior. And they can make adjustments accordingly.
- *Reducing Manual Processes with Automation*. CD allows developers to automate the software development process to the greatest extent possible, especially when it comes to "release" testing. Automation helps developers push out releases faster and spend less time on manual processes.

NSDF will:

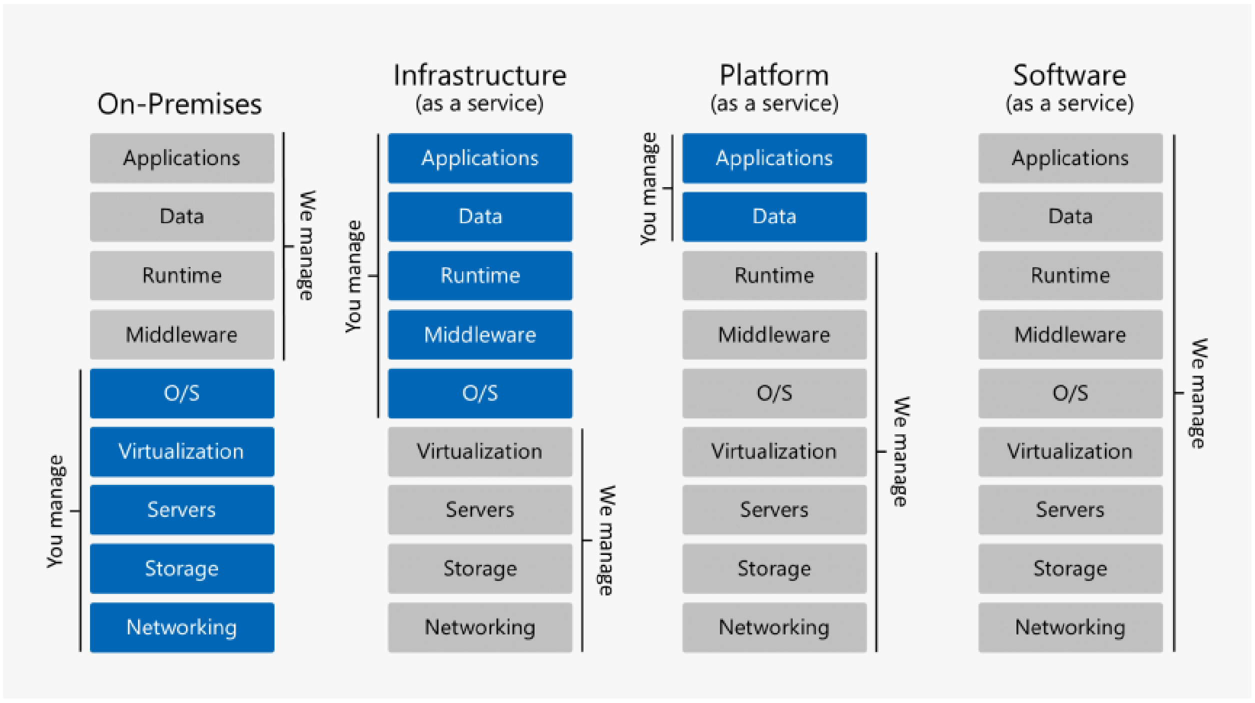
- First implement *Continuous Delivery _by adding libraries or container images to an Artifact Registry*, allowing stakeholders/research scientists/partners to install binaries and start experiments with them.

- In the second phase of the pilot, since ssh-ing into a cluster and running commands is an unsustainable and error-prone practice, we will gradually introduce a full-fledged _deployment _on multi-regional lightweight Kubernetes clusters.

1.3.1 Cloud Deployment

We have several choices to deploy NSDF applications to the cloud:

- **Infrastructure as a Service (IaaS)**.
  - A cloud service provider (CSP) hosts the hardware components, including servers, storage, and networking hardware, as well as the virtualization or hypervisor layer
  - Examples: `AWS EC2`, `Google Compute Engine`, `Microsoft Azure`, etc.
- **Software as a Service (SaaS)**
  - it eliminates the need to install and run applications on the servers. It provides standalone and ready-to-use software on the cloud. This eliminates the expense of hardware procurement, provisioning, and maintenance as well as software licensing, installation, and support.
  - Examples: `Dropbox`, `Slack`, `Microsoft Office 365`, etc.
- **Platform as a Service (PaaS)**
  - A provider forms and supplies a strong and optimized environment in which users can install applications and datasets.
  - Examples: `AWS Lambda`, `AWS Elastic Beanstalk`, `Heroku`, etc.
- **Container as a Service (CaaS)**
  - It is a cloud-based service that allows software developers and IT departments to upload, organize, run, scale, and manage containers by using container-based virtualization. CaaS abstracts the full stack in a very compelling way, without the many challenging surprises existing in both IaaS and PaaS.
  - Examples: `Kubernetes`, `Docker Swarm`, `Apache Mesos`, etc.



For the NSDF pilot the:

- **IaaS approach is admitted**. But NSDF is cloud-agnostic with no special preferences for commercial solutions. In general, we prefer to install orchestration tools on top of computing instances.
- **SaaS/PaaS approach is discouraged**: most of the time they are _vendor-lock-in_ and *pay-per-use* solutions. An exception is made for the *Function As a Service* (FaaS), a subset of PaaS, but only limited to open-source projects (e.g. OpenFaaS, Kubeless, OpenWhisk, etc. see See A (Very!) Quick Comparison of Kubernetes Serverless Frameworks – VSHN AG)
- **CaaS hybrid approach is probably the best** for our pilot, where some services are deployed in the public cloud and some other services are deployed on-premises (i.e. local clusters):
  - CaaS permits to deploy quickly and lightly on almost any infrastructure;
  - CaaS provides commodified, standardized functionality on-premise and/or on public clouds;
  - CaaS offers open container technology, which is the de-facto standard in the cloud industry;
  - CaaS offers freedom and flexibility to developers.

1.3.2 Artifacts Repositories

Software releases should be made available through an *Artifacts Repository Manager*, as defined on Wikipedia as:

> …a software tool designed to optimize the download and storage of binary files used and produced in software development. It centralizes the management of all the binary artifacts generated and used by the organization to overcome the complexity arising from the diversity of binary artifact types, their position in the overall workflow, and the dependencies between them.

The best practice is to follow the standard community consensus and make releases available through the most widely-used common channels.

Currently, our top choices to share NSDF artifacts are based on the programming languages:

- **C/C++**
  - `Windows` using the specific Microsoft Visual Studio distributing the proper *Microsoft Run-Time Kit*
  - `macOS` using the `MacOSX10.9.sdk` for portability on any recent macOS version
  - `Linux` using manylinux (or `manylinux2014`, or `manylinux_x_y`) for broad compatibility with Linux distributions
- **Python**
  - All public Python libraries must be released through PyPi.
  - NSDF libraries with complex compilation requirements will be released to Conda-Forge, which can produce platform-specific compiled versions.
- **Docker**
  - A Docker image represents binary data that encapsulates an application and all its software dependencies.
  - An image of an application is created and pushed to a registry. Images must be named as follows and tagged with the current version: `nsdf/image-name:v1.0.0` .
  - NSDF recommends using GitHub (or GitLab) as a public Container Registry. (see Github Container Registry)
- **Kubernetes**
  - YAML files can be versioned in a GitHub repository as with code.

- For Helm, specific distribution NSDF recommends using Artifact Hub.
  - Artifact Hub allows publishers to list their content in an automated way (see See https://artifacthub.io/docs/topics/repositories/).

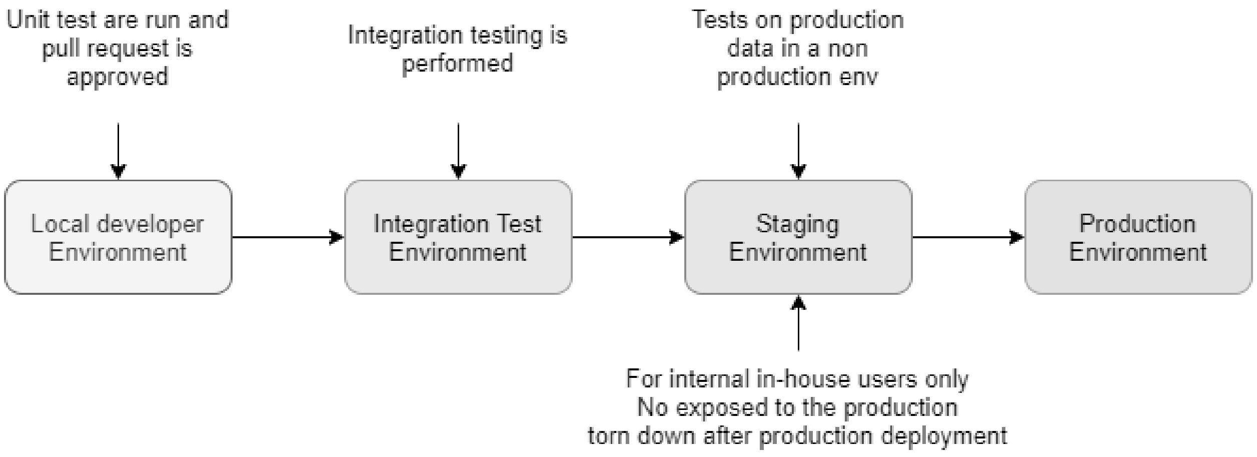
1.3.3 Deployment staging

A staging environment is an environment used to deploy software before it goes to production. Staging environments are generally meant to be identical or nearly identical to production.

Staging allows discovering code quality issues, integration problems, and other dependency issues which would not be as obvious in a minimal environment as the developer's one.

Staging helps to discover *cross-service problems*, for example, a library that may work on a local machine but may not work in the cloud.

A staging environment is typically not made available to the outside, but rather it is made available to an internal user base.


Unit test are run and pull request is approved
Integration testing is performed
Tests on production data in a non production env
Local developer Environment
Integration Test Environment
Staging Environment
Production Environment
For internal in-house users only
No exposed to the production
torn down after production deployment

Sometimes staging is replaced by canary or blue/green deployments, but this approach still exposes users to bugs and misconfigurations.

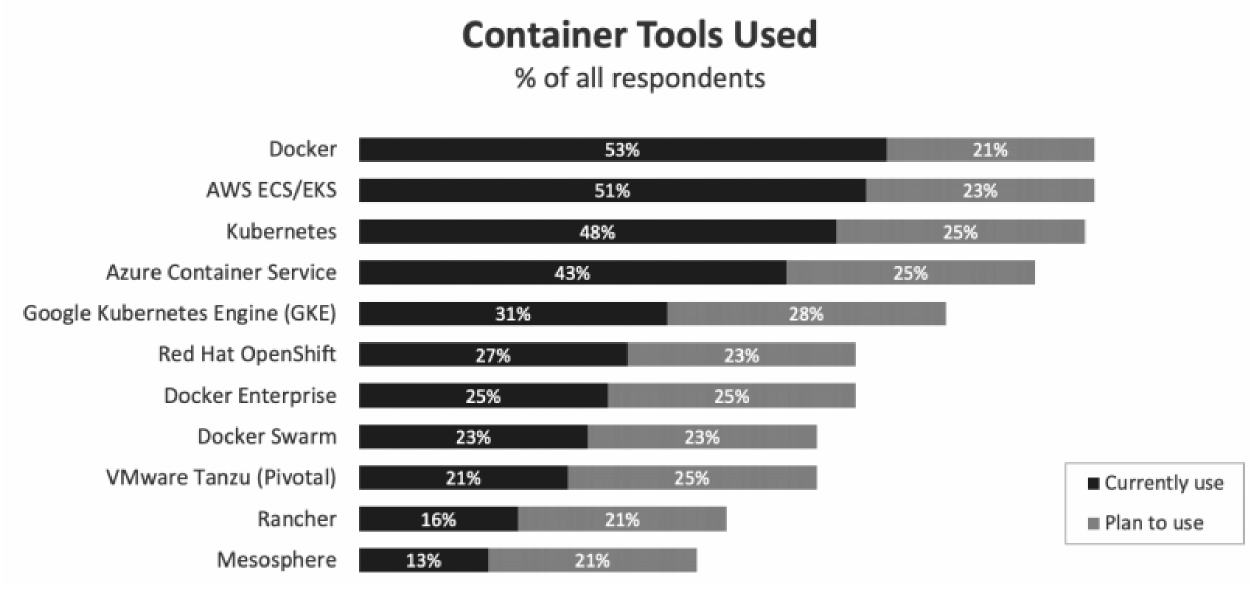
Helm simplifies enormously the maintenance of different environments (e.g. staging vs production). It's sufficient to 1) create a HELM chart 2) create several YAML *value files*, one for each environment 3) isolate resources in different namespaces:

```
kubectl create namespace staging
kubectl create namespace production
helm install nsdf-app-staging nsdf-app -n staging    -f values-staging.yaml
helm install nsdf-app-prod    nsdf-app -n production -f values-prod.yaml
```

1.3.4 Docker Containerization

Containerization enables the portability and reproducibility of data and/or code.

There are a plethora of *container engines*. However, it is not guaranteed that a container image generated by one container engine can be utilized by another. In addition, container engines tend to adopt different command lines which further complicate development.



This NSDF pilot will adopt Docker as the standard containerization platform. Docker images and the corresponding recipes will be provided via our project registries.

Udocker can be considered when the installation of Docker or other HPC containers is not possible e.g. user namespace is disabled. Udocker is a Python wrapper that can provide a chroot-like environment over the extracted Docker container.

1.3.5 Kubernetes Orchestration

To facilitate management and deployment of applications and clusters NSDF recommends using Kubernetes, Ansible, and Openstack.

Kubernetes, often abbreviated as "K8s," automates the scheduling, scaling, and maintenance of containers in any infrastructure environment. Open-sourced by Google in 2014, Kubernetes is now part of the *Cloud Native Computing Foundation *(CNCF).

Since its introduction in 2014, Kubernetes has been steadily gaining in popularity across a range of industries and use cases. Recent research shows that almost one-half of organizations running containers were using Kubernetes.

Kubernetes is widely-used in Cloud, while Ansible and Openstack have also seen their usage in HPC centers. Ideally, the orchestrators can automatically pull the container images from the project registry or build the images for users by Ansible or Kubernetes.

Advanced users can also provide their recipes and the orchestrators will build the images for users.

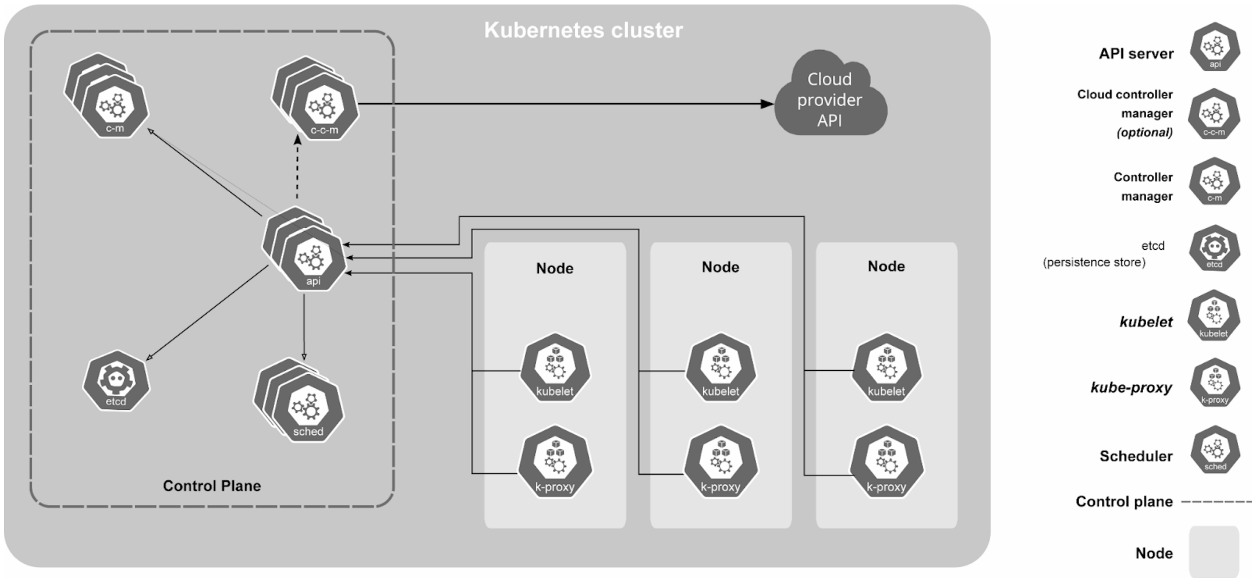
The overall orchestration workflow can be divided into three stages:

1. Ansible/Openstack scripts to deploy the clusters.

2. Container image provision and container management by Ansible/Kubernetes.
3. Container deployments across clusters.

A Kubernetes deployment is made of:

- **Cluster**. A K8s cluster has two basic components:
  - a *control plane* that works to maintain the cluster in the desired state as configured by the user. It consists of 4 key services for cluster administration:
  - *the API server* exposes the K8s API for interacting with the cluster;
  - the *Controller Manager* watches the current state of the cluster and attempts to move it toward the desired state;
  - the *Scheduler* assigns workloads to nodes;
  - `etcd` stores data about cluster configuration, cluster state, and more
  - the *nodes* that handle application workloads.
- **Pods**. Each node in the K8s cluster runs one or more K8s pods.
  - A pod is a group of containers with shared storage, network resources, network namespace, IPC namespace, and operational specifications.
  - K8s will always schedule containers within the same pod together, but each container can run a different application.
  - The containers in a given pod run on the same host and share the same IP address, port space, context, namespace, and even resources like storage volumes.
- **Containers**. A container is an executable piece of software that includes an application and all of its related configuration files, libraries, and dependencies.
  - Pods can contain more than one container if their functions are tightly coupled.
- **Containerized Applications** are executable software programs that have been packaged in containers along with their related dependencies.
- **Host Machines**. K8s can run on bare metal servers, virtual machines, on-prem services, or in the public cloud.

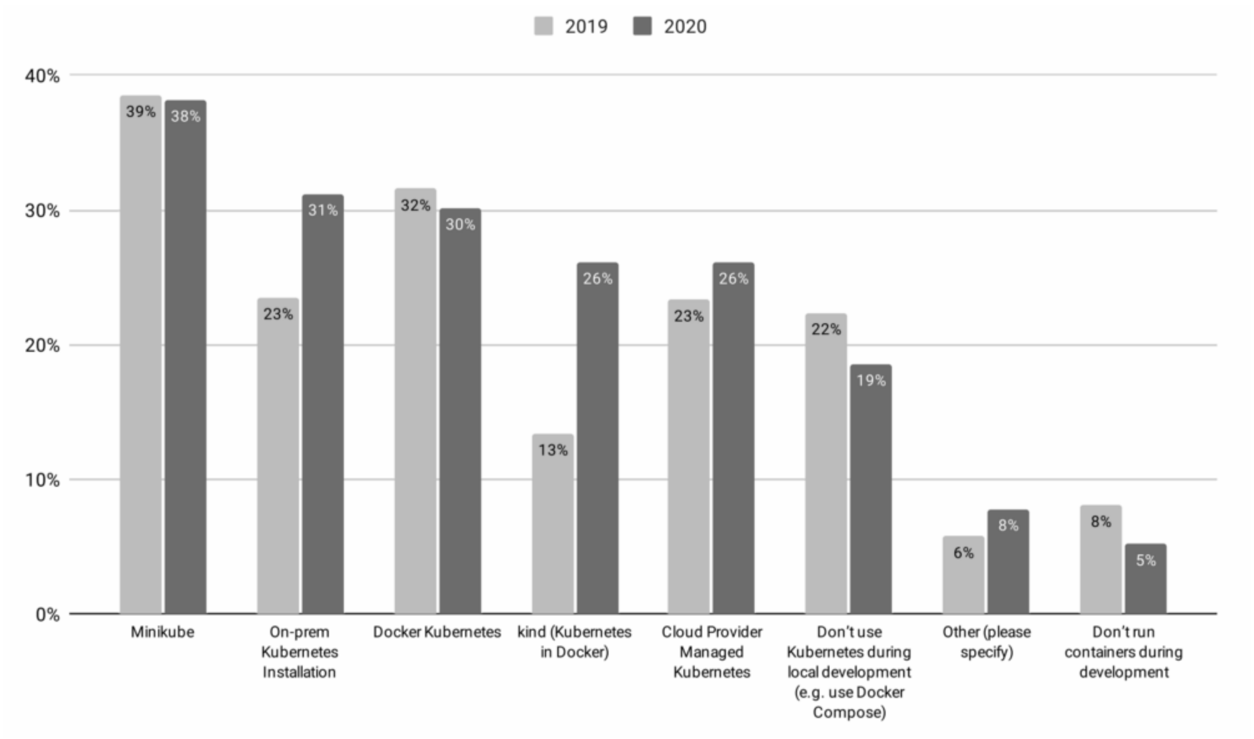


On each node, there is a `kubelet` agent that receives instructions from the API server and makes sure all the containers are running properly.

Kubernetes automates container infrastructure via:

- Container scheduling and auto-scaling
- Health checking and recovery
- Replication for parallelization and high availability
- Internal network management for service naming, discovery, and load balancing
- Resource allocation and management

For single-node testing, NSDF recommends using MicroK8s or even a simpler Docker Compose.



For multi-host deployment we are currently experimenting with (in the current order of preference):

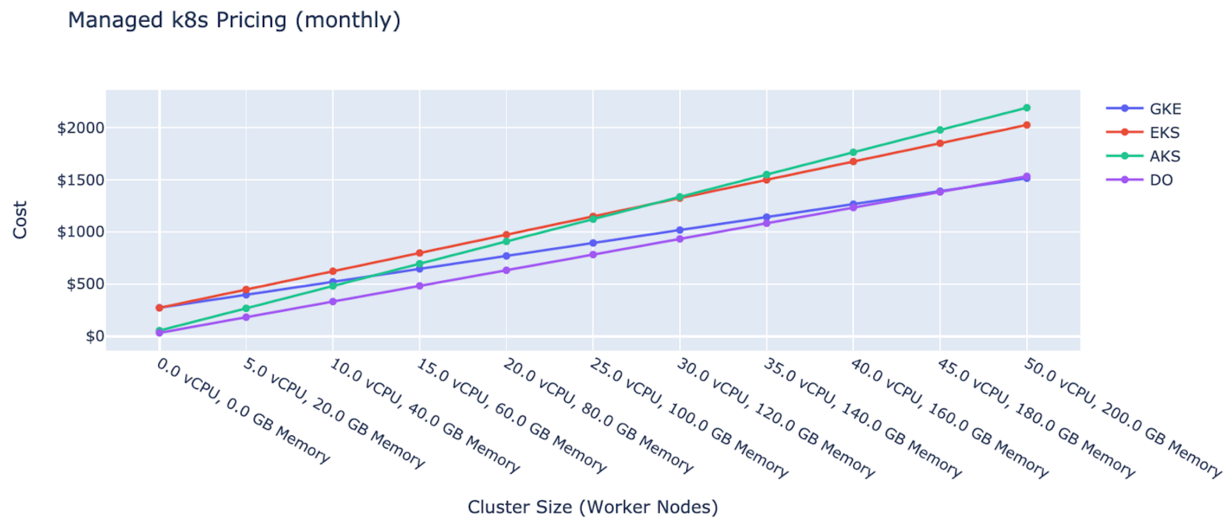
- K3s. A very lightweight K8s installation, with a single-file script with the option to add multiple workers/ high availability.
- kubespray. Kubespray is a combination of Kubernetes and Ansible. We can also deploy clusters using kubespray on cloud compute services like EC2 (AWS).
- kubeadm Kubeadm automates the installation and configuration of Kubernetes components such as the API server, Controller Manager, and Kube DNS. Kubeadm performs the actions necessary to get a minimum viable cluster up and running quickly. By design, it cares only about bootstrapping, not about provisioning machines (underlying worker and master nodes). Kubeadm also serves as a building block for higher-level and more tailored tooling.

Another option would be to use commercial and managed cloud vendors solutions (see Setup Kubernetes on different vendors) that have undoubtedly the advantage of freeing us for maintaining the cluster and worrying about the infrastructure underneath. But, at the same time, this solution is more expensive and we need to be careful about the fact that pricing is not always 100% transparent and/or simple to estimate.

Also, **EGRESS costs can be an important, if not dominant, piece of the overall costs,** in particular for this pilot where big data will be frequently moved between K8s clusters.

Looking at the table below it's noticeable that even with low computing resources (a total of 50 vCPU and 200 GiB RAM, which could be barely considered a real cluster for production purposes) costs immediately grow up to *some thousands per month*.

For these reasons, we believe that the adoption of commercial cloud should be limited and may be reconsidered only in the presence of ad-hoc agreements, waivers for EGRESS, and special educational pricing.



1.3.6 High-Performance Computing Deployment

For High-Performance Computing (HPC), with tens of thousands of massively parallel systems, the overhead introduced by Docker containers is simply unsustainable and we should look to alternatives.

A brief comparison is given in the following list:

- **Charliecloud**
  - an open-source project based on a user-defined software stack (UDSS).
  - Like most container implementations, it uses Linux user namespaces to run unprivileged containers.
  - It is designed to be as minimal and lightweight as possible, to the point of not adding features that could conflict with any specific use cases.
- **Shifter**
  - is another container run-time implementation focused on HPC users.
  - It is almost exclusively backed by the National Energy Research Scientific Computing Center and Cray.
  - Most documented use cases use Slurm for cluster management/job scheduling.
  - It uses its own specific format, but this is reverse-compatible with Docker container images.
- **Apptainer** (ex Singularity)
  - is written in Golang.
  - It supports two image formats: Docker/OCI and Singularity's native

  - There are lots of systems running Singularity (including users like TACC, San Diego Supercomputer Center, and Oak Ridge National Laboratory).
  - Admin/root access is not required to run Singularity containers and it requires no additional configuration to do this out of the box.
  - A fork maintained by Sylabs also distributes a mostly feature compatible version using SingularityCE/PRO/Enterprise branding.
- **Podman**
  - is a container run-time developed by Red Hat.
  - Its primary goal is to be a drop-in replacement for Docker.
  - While it is not explicitly designed with HPC use cases in mind, it intends to be a lightweight "wrapper" to run containers without the overhead of the full Docker daemon.
  - The Podman development team is recently looking into better support for HPC use cases.

NSDF has a preference for Apptainer/Singularity since

1. it's the most-accepted container engine in HPC communities
2. it is already used *internally _by NSDF developers,* _and
3. IBM cloud, one of the partners supporting this pilot, is providing it out-of-the-box.

UDocker can be an alternative option when the aforementioned HPC containers are unavailable, e.g. namespace may be disabled on certain HPC systems. Installation of Docker is not required for usage of Udocker whose installation can be performed on the user home directory that does not demand any privileged operations. Containers can run in Chroot-like environments with the extracted container images placed in `$HOME/.udocker` by default.

1.3.7 Links/Bibliography

List:

- Deployment Strategies In Kubernetes
- K3s: Lightweight Kubernetes
- Bare Metal Kubernetes with MetalLB, HAProxy, Longhorn, and Prometheus
- Kubernetes-sigs/kubespray: Deploy a Production Ready Kubernetes Cluster
- Deploy a Production-Ready On-Premise Kubernetes Cluster
- Install a Kubernetes load balancer on your Raspberry Pi homelab with MetalLB
- How to install Kubernetes on Ubuntu 20.04
- Install Kubernetes Cluster on Ubuntu 20.04 with kubeadm
- Setup On-premise Kubernetes with Kubeadm, MetalLB, Traefik, and Vagrant
- Kubernetes Lab on Baremetal
- From Zero to Code: Moving from Docker to Kubernetes
- Deploy a Production-Ready On-Premise Kubernetes Cluster
- Ultimate Cloud Pricing Comparison: AWS vs. Azure vs. Google Cloud in 2021 - CAST AI
- An introduction to Kubespray
- HPC workloads in containers: Comparison of container run-times

- Staging Environments Are Overlooked — Here's Why They Matter

1.4 Continuous Testing

Overview of section contents:

Section	Description
Test-driven development	NSDF recommends using TDD just for limited use cases
Unit testing and frameworks	Recommendation for software unit tests in order to improve software quality
Other tests	A few other software tests to improve quality
Code Analysis	Open-source tools for code analysis, particularly for C/C++ and Python

---

Print statements are not suitable for testing. They are just bad practices.

Continuous Testing is a defense against software defects across the software life cycle and it means *continuous feedback* on software quality.

This process is also known as *shift-left testing*, which stresses the concept of integrating development and testing activities to ensure quality is *built-in* as early as possible.

1.4.1 Test-driven development

Test-driven development (TDD) is a software development process that relies on the repetition of a short development cycle:

- first, the developer writes a failing test case that defines a desired function;
- then he produces code to pass the test;
- refactors the code to acceptable standards.

Test-driven development is related to the *test-first programming* concept of _Extreme Programming (see _Going Agile With The Test-First Development Approach) and is often linked to an agile programming approach.

NSDF recommends using TDD just for limited use cases. Our Software Stack implies too many integrations and complex interactions between different parts to be dealt with TDD.

1.4.2 Unit testing and frameworks

Unit testing is a method to demonstrate the correct behavior of the software. It's the verification process to ensure that each software unit does what it's required to do in terms of code safety, security, and reliability.

Unit testing must be an integral part of NSDF software development.

The adoption of unit tests has several benefits:

- *facilitate changes:* unit tests allow programmers to refactor code at a later date, and be sure that code still works correctly;
- *simplify integration*: unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach.
- is *living documentation* for the system: developers can look at the unit test's code to gain a basic understanding of the APIs.

1.4.2.1 Mocking

Mock objects can simulate the behavior of real objects and they are very useful when a complex object is impractical to incorporate into a unit test. Mock objects can be used to

- supply non-deterministic results (e.g., current time);
- reproduce difficult states (e.g. a network error);
- reproduce slow states (e.g. simulate a database)

Mock objects have the same interface as real objects they mimic, allowing a client object to remain unaware of whether it's using a real object or a mock object. Mock object frameworks allow the programmer to

1. specify which methods will be invoked on a mock object,
2. what parameters will be passed to them, as well as
3. what values will be returned.

For example this PropHolder class:

```
class PropHolder {
public:
    PropHolder()  { }
    virtual ~PropHolder() { }
    virtual void SetProperty(const std::string& name, int value) = 0;
    virtual int GetProperty(const std::string& name) = 0;
};
```

can be mocked into a new class:

```
class MockPropHolder: public PropHolder {
public:
    MockPropHolder() { }
    virtual ~MockPropHolder() { }
    MOCK_METHOD2(SetProperty, void(const std::string& name, int value));
    MOCK_METHOD1(GetProperty, int(const std::string& name));
};
```

and, during unit testing, we will specify its behavior:

```
AUTO_TEST_CASE(test_gmock)
{
  MockPropHolder mholder;
  EXPECT_CALL(mholder, GetProperty(std::string("test"))).Times(1).WillOnce(Return(101));
  EXPECT_CALL(mholder, SetProperty(std::string("test2"),555));
  TestClass(mholder).doCalc();
}
```

1.4.2.2 Unit testing frameworks

Unit test frameworks simplify the development of unit tests.

The most well-known frameworks belong to the xUnit family of frameworks (CppUnit, NUnit, etc.).

Frameworks from this family rely on:

- *Assertions*, that check individual conditions;
- *Test cases*, that combine several assertions, based on some common functionality;
- *Test suites,* that combine several tests, logically related to each other;
- *Fixtures*, that provide setup of data or state, needed for execution of some tests, and cleanup of state and/or data after the test is finished.
- _Frameworks _that control how tests are executed and collect failed tests.

Writing unit tests is a non-trivial time-consuming process: they should cover all public functions, main paths common and edge cases, etc (see this link).

1.4.2.3 C++ Unit testing

There are many unit testing frameworks for C++ (see Wikipedia list of unit testing frameworks).

NSDF recommends using one of the following (in no particular order of preference):

- Google C++ Testing Framework is an open-source project hosted at GitHub, and it can be used on all platforms. Plus it has full support for mocking (see the *Google Mocking framework*).
- Boost.Test was created by several people on the C++ standards committee. It's popular with developers who use the other Boost libraries. It has excellent documentation and handles particularly well exceptions and crashes. It lacks mocking features
- banditcpp is a modern C++ unit testing framework with support for Lambdas. It's available under the open-source license, and it supports C++11.
- CppUnit is the C++ porting of the JUnit framework. It may be hard to use because of the lack of documentation.

1.4.2.4 Python Unit testing

There are a lot of good and advanced Python libraries to perform unit testing (e.g. robotframework, doctest, nose2, testify, etc).

NSDF recommendation is to use either the standard built-in Python Library unittest or the pytest, used for example by NumPy.

An example of a Python unit test for a factorial function:

```python
import unittest
import math
import factorial_v1
from test import test_support
class FactorialTest(unittest.TestCase):
    def setUp(self):
        print("setup")
    def tearDown(self):
        print("cleanup")
    def test_positives(self):
        for x in range(0,10+1):
            act = math.factorial( x )
            val = factorial_v1.fact( x )
            self.assertAlmostEqual( act, val, 1e-1 )
    def test_negative(self):
        passed = False
        try:
            factorial_v1.fact( -3 )
        except Exception as e:
            passed = True and (e.message.find("Cannot calculate")>= 0 )
        self.assertTrue( passed )
if __name__ == "__main__":
    test_support.run_unittest(FactorialTest)
```

1.4.2.5 Go Unit Testing

The Go programming language comes with a built-in unit testing framework. This framework has been standardized across projects. The package is "testing", and the command is `go test`. Go will look for files named `\*\_test.go` files. The testify require package makes assertion testing easier with intuitive functions to call.

```go
package mcmodel
import(
   “errors”
   “testing”
   “github.com/stretchr/testify/require”
)
func TestQueryDataset(t *testing.T) {
    x := 5
    // Two ways to test with and without testify
    // Without testify
    if x != 5 {
       t.Errorf(“Expected x == 5, got %d”, x)
    }
```

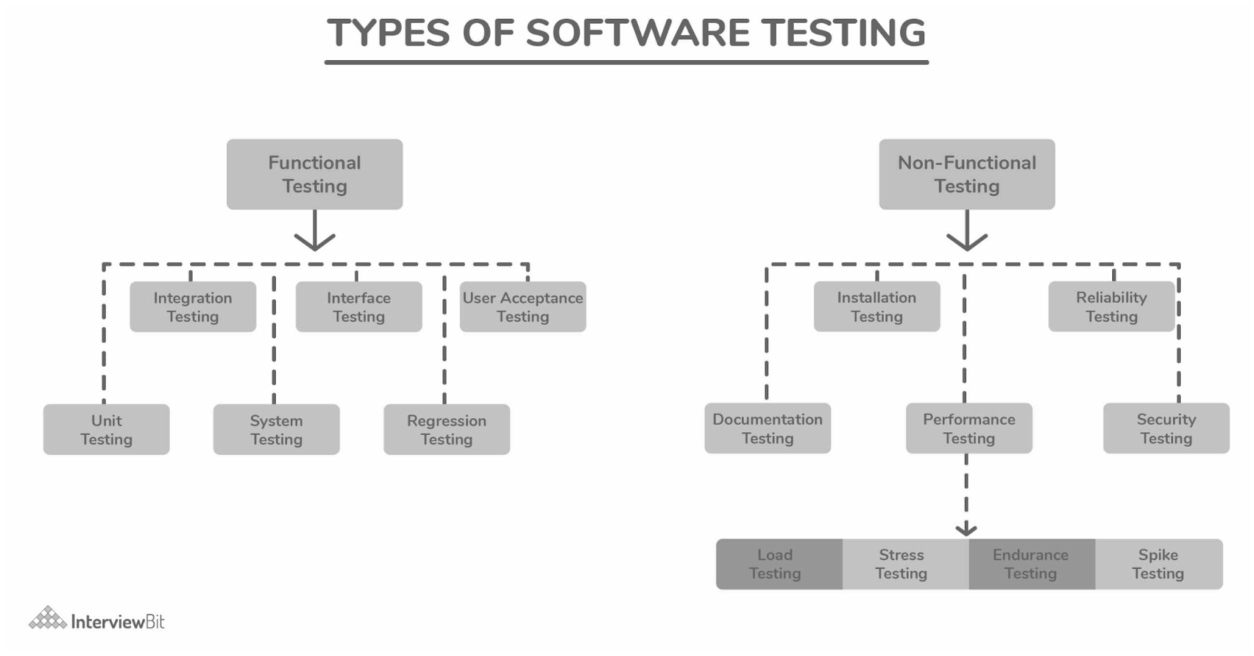
(continues on next page)

(continued from previous page)

```
   // with testify
    require.Equal(t, 5, x, “x should be 5”)
}
```

1.4.3 Other tests

There is extensive literature about manual or automatic testing. It’s out of scope to cover them all.



But just to name a few that could be adopted for the NSDF software stack:

- **Integration testing** to test whether multiple software components function well together as a group.
- **Regression testing**: to verify that the recent changes or updates have no negative effect on the already existing functionality.
- **Interoperability testing**: to check if the software can interact with other components without any compatibility issues.
- **Availability/Disaster recovery testing**: it is also a measure of how long failures will last and how much time the repair can take.
- **Compatibility/Portability testing**: how seamlessly the product operates with other components: OS, browsers, hardware, etc…
- **Scalability/Load /Stress testing**. ensures that the application can grow in proportion to the increasing demands of the end-users. It checks the quality of the system under high-peak loads.
- **Security testing** tries to find the system’s vulnerabilities and determine how confidential data and internal resources are protected.
- **Maintainability testing** measure the ability to safely go through changes and updates.

1.4.4 Code Analysis

NSDF recommends adding to the CI/CD pipeline either static analyzers, runtime checkers, or code coverage tools.

Code coverage refers to how much of your code is being executed while your automated tests are running. This metric is calculated by special tools that add tracing calls inside the binaries of your code. This insight into your applications can inform future development, for example: () *finding what parts of your code are covered by your tests* () finding what parts of your code are not covered by tests (*) removing dead code.

Code coverage cross-language tools include:

- Codecov with unified coverage and separate coverage
- Coveralls.io is free for open source repositories and very popular

1.4.4.1 C++ Code Analysis

For C++ this is a non-exhaustive list of *static analyzers*:

- Cppcheck provides code analysis to detect bugs, undefined behavior, and dangerous coding constructs
- cppclean is focused on finding problems in C++ sources that slow the development of large codebases
- codechecker is built on the LLVM/Clang Static Analyzer toolchain
- Flint++ is a cross-platform, zero-dependency program developed and used on Facebook.
- OCLint reduces defects by inspecting C, C++, and Objective-C code

For code coverage, we suggest adopting open-source tools well integrated into the CI pipeline such as Codecov or Coveralls.io.

1.4.4.2 Python Code Analysis

In the Python arena, Coverage.py is one of the most complete and well-maintained projects.

Also worth mentioning are:

- Codacy , Code quality, and coverage grouped by file
- Scrutinizer CI , Code Quality, and coverage grouped by class and function

Besides code coverage, various tools for static analysis and linting (Mypy, Pylint , Flake8) as well as automatic code formatting (Black) options exist for Python.

1.4.5 Links/Bibliography

List:

- http://alexott.net/en/cpp/CppTestingIntro.html
- Unit Testing Guidelines
- Ten C++ Testing Tools for Developers to Consider
- Unit Test Frameworks
- Practical Testing
- Unit Testing Frameworks in Python
- Python Unittest Vs Pytest

- Automated Regression Testing: A Comprehensive Guide
- Functional Vs Non-Functional Testing: Expert Guide - UTOR
- The Best Code Coverage Tools By Programming Language
- Use the Tools Available · C++ Best Practices
- Automated Defect Prevention: Best Practices in Software Management | Wiley
- Github Action for Unit Testing
- Unit testing - Wikipedia
- Guide: Writing Testable Code

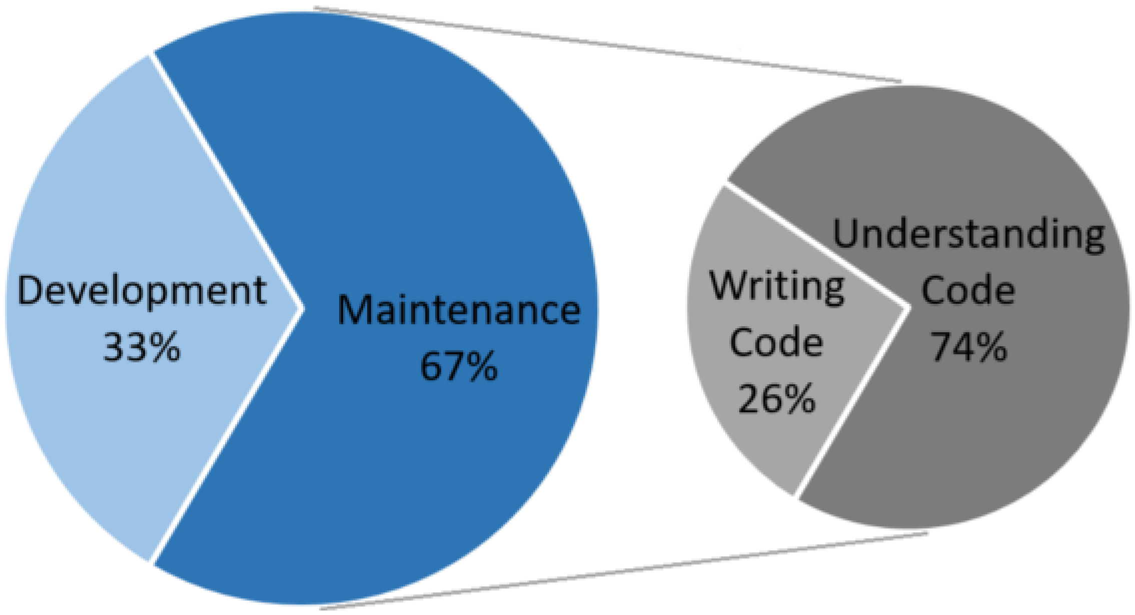
1.5 Continuous Documentation

Overview of section contents:

Section	Description
Markdown	The Markdown language for lightweight documentation
Documentation as code	NSDF recommendation is to follow the "*Documentation of code*" philosophy as closely as possible
Code as Documentation	Code in a way that is more readable and self-explaining, particularly practices for C++ and python
Documentation tools	Documentation using Jupyter Notebook and Web service API

---

*Say what you mean, simply and directly. Don't comment on bad code, rewrite it, Make sure comments and code agree (*The Elements of Programming Style by Brian W. Kernighan and P. J. Plauger)



Documenting Software is an important activity of development and it is fundamental for *software maintenance _and _knowledge transfer.*

But writing too much and too verbose documentation could be a problem itself, and for this reason, NSDF recommends following some principles contained in the "Agile Manifesto" (written by seventeen software developers on February 11-13, 2001, at The Lodge at Snowbird, a ski resort in the Wasatch mountains of Utah see Manifesto for Agile Software Development):

> *We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes [. . .] While there is value in the comprehensive documentation we value working software more.*

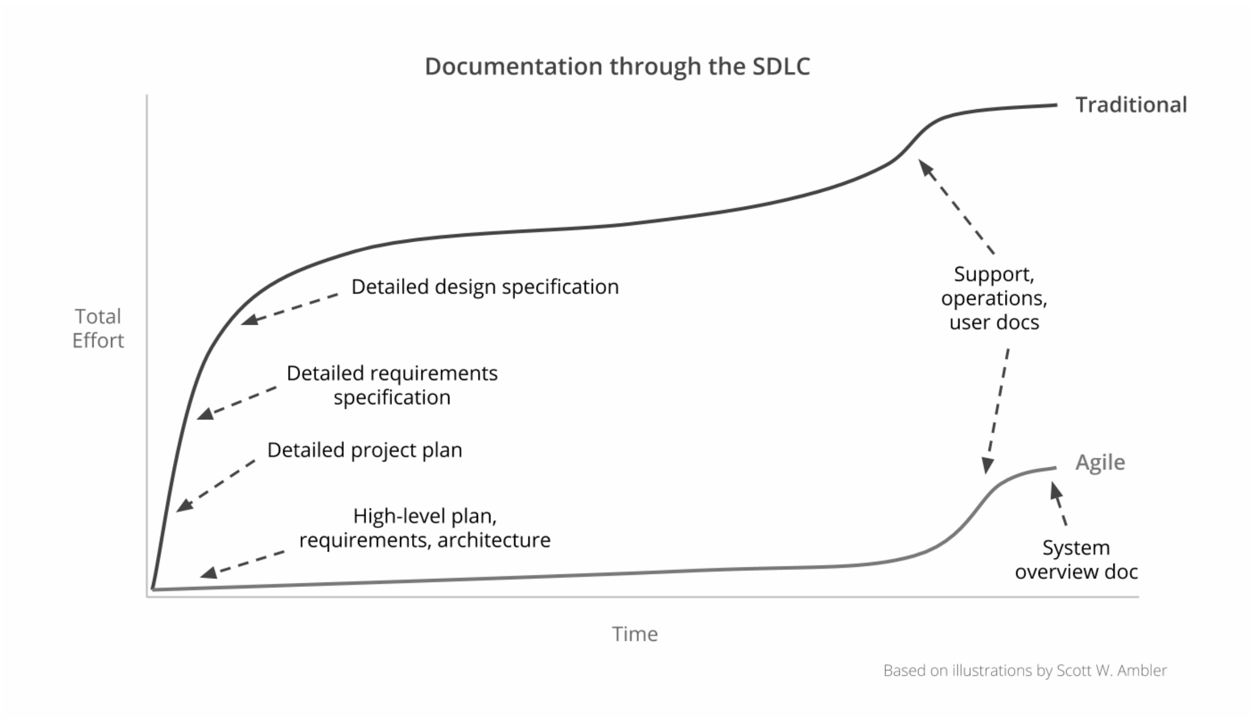
Documenting software is always an *imperfect compromise*: too much documentation would be a waste of time, and developers will rarely trust it anyway because it's usually out of sync with the actual code. On the other hand, too little documentation is always a source of problems with team communication, learning, and knowledge sharing.

WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.
scottadams@aol.com
www.dilbert.com
©2007 Scott Adams, Inc./Dist. by UFS, Inc.

So NSDF's major recommendation is to "*document code efficiently*":

- Write only the minimum, useful, accurate documentation
  - Make sure documentation is "*just barely good enough*". Any document will need to be maintained later on.If the documentation is light it's easier to comprehend and update.
- Write it "*just in time*" (JIT).
  - Wait before documenting.

  - Produce documentation when it is needed, not before.
  - System overviews and support documentation are best written towards the end of the software development life cycle.
- Cut out anything unnecessary
  - documentation is only useful if it's accessible.
- Follow code changes; have documents that are always *shippable*
- Keep documents in one place and make them *accessible online*.
  - Store your product documentation in a place where all the members and external contributors can find it.
- *Collaborate*. Writing documentation is a collaborative and instructive process
  - Every team member should be encouraged to contribute.



1.5.1 Markdown

Markdown is a lightweight markup language that allows you to create web pages, wikis, and user documentation with a minimum of effort. Documentation written in markdown looks exactly like a plain-text document and is perfectly human-readable.

In addition, it can also be automatically converted to HTML, latex, pdf, etc.

1.5.2 Documentation as Code

There has long been a mindset to treat documentation and code as separate functions. **But this thinking is *obsolete*.**

Traditional methods of documentation focus on the concept of a printed page. But most documentation in today's age is never printed: the documentation created with page-oriented methods does not adapt well to different electronic *always-online* devices.

It's very instructive to learn from what big IT companies did in the recent past facing the "*documenting code in an efficient way*" problem:

- Twitter 2014 talk described how they solved the documentation maintainability problem. Indeed they were probably the first to end up treating their documents like code.
- Google 2015 talk (*Documentation disrupted: how two technical writers change google engineering culture*") Riona Macnamara, a technical writer at Google, confirmed that the major problem was not the lack of documentation, but rather that it was outdated, untrustworthy, and scattered across wikis, Google Sites, Google Docs, etc.
- Spotify 2019 talk described how they changed their approach in writing internal technical documentation (cit. "*We conducted a company-wide productivity survey. The third-largest problem according to all our engineers? Not being able to find the technical information they needed to do their work.*") and announced the TechDocs open-source Cloud Native Computing Foundation (CNCF) platform

Documentation as code addresses the need for multiple formats and ease of maintenance.

It makes the documentation part of the Continuous Integration (CI) pipeline; and it empowers developers to apply the same methods and tools, such as

- Issue Trackers
- Version Control (Git)
- Plain Text Markup (Markdown, reStructuredText, Asciidoc)
- Code Reviews
- Automated Tests

Therefore NSDF recommendation is to follow the "*Documentation of code*" philosophy as closely as possible:

- Use plain text files (e.g. Markdown file format). This way the documentation can be consumed on any device.
- Use open-source static site generators to build the files locally (e.g. Sphinx, Jekyll, Hugo)
- Work with files through a text editor (e.g. Visual Studio Code, Sublime Text, etc.)
- Store documents in a version control repository (e.g. GitHub) or a collaborative documental environment (e.g. Slite, Docs in ClickUp™, etc.).
  - Online content makes documents easy to consume.
  - Content exists in one place but can be pulled into other documents as needed
  - Content is searchable within and across documents,
  - Content keeps updated with code changes.
- Collaborate using version control to branch, merge, push, and pull updates.
- We can add validation tests to check for broken links, improper terms/styles, and formatting errors
  - As an example, https://slateci.io/ is the website of a successful NSF project that is using the approach described here.

This approach will help to build and maintain the *NSDF Web Site* too: we can store Jekyll templates into a central GitHub repository and, on *git-push events*, a workflow will run and automatically build the NSDF website (see the GitHub Pages project for more details).

As an example, https://slateci.io/, another NSF-funded project for "*Federated Operation of Science Platforms*" is using Jekyll to produce their website, and collaborating/modifying it's just a matter of editing markdown documents on GitHub.

If you are looking to have a large impact at a growing company and work with a high performance team - start here. Work with talented colleagues on hard problems that affect millions of customers.

At Netflix we value high performance, freedom and responsibility. We don't focus on rules, processes or procedures. We are candid and transparent and seek excellence in everything that we do.

We tackle problems others have not been able to solve. We license great content, build systems at scale and use data to push the business forward. We connect people with movies and television globally.

Check out our jobs page for current openings.

Netflix Open Source Software Center

1.5.3 Code as Documentation

Code as documentation is a principle that advocates making code more readable and self-explaining. But it does not mean that the code should not be documented, or that it is the only source of documentation.

Regarding this matter, we are quoting the words of a famous post by Martin Fowler (see https://martinfowler.com/bliki/CodeAsDocumentation.html):

> *One of the common elements of agile methods is that they raise programming to a central role in software development - one much greater than the software engineering community usually does Part of this is classifying the code as a major, if not the primary documentation of a software system. [..]*
> *I think part of the reason that code is often so hard to read is that people aren't taking it seriously as documentation. [. . .] So the first step to clear code is to accept that code is documentation, and then put the effort in to make it clear. [. . .] We as a whole industry need to put much more emphasis on valuing the clarity of code.*

There are several ways we can assure that our code is clean and easy to understand (For more extensive descriptions see Clean Code: A Handbook of Agile Software Craftsmanship: Martin, Robert C.).

1.5.3.1 Use intention-revealing names

Give meaningful names to variables, methods, and classes (even if they become long). If the class or method name describes what it does, and if the field name informs what it has, it is not necessary to write a comment to inform it. The idea is that, when we read a variable or method name, we can understand what it does. Below, two code examples show the same variable declaration was written in two different ways.

Bad code:

```
int d; //elapsed time in days
```

Good code:

```
int elapsedTimeInDays;
```

1.5.3.2 Refactor long blocks

Big methods are hard to read and understand, mainly if it has a lot of responsibilities. Write small methods. Each method should have only one responsibility and its name should describe it.

If some method is big, extract each functionality in smaller methods. When writing a code, keep in mind that it can be reused in another part of the system itself or other systems.

1.5.3.3 Use informative comments

A clean code tells you what it does, but it does not show clearly your intention and why it was done that way. For this, you can use comments, it is an important resource to complement the understanding of the code.

There are special comments like TODO and FIXME, that are used to record reminders to future improvements and corrective tasks. Use them when a code is incomplete, incorrect, or can be improved but you do not have time to make it at this time.

Avoid *redundant*, _misleading, _and _noisy _comments.

NSDF suggest reading this interesting post Writing system software: code comments which differentiate between positive forms of commenting) Functional, Design, Why, Teacher, CheckList, Guide) and somewhat questionable comments (Trivia, Debt, Backup); and it points out a reasonable vision:

> *Many comments don't explain what the code is doing. They explain what you can't understand just from what the code does. Often this missing information is* why *the code is doing a certain action, or why it's doing something that is clear instead of something else that would feel more natural.*
>
> *While it is not generally useful to document, line by line, what the code is doing because it is understandable just by reading it, a key goal in writing readable code is to lower the amount of effort and the number of details the reader should take into her or his head while reading some code. So comments can be, for me, a tool for lowering the cognitive load of the reader.*

1.5.3.4 Use class-level documentation

Class-level documentation should describe the purpose of this unit of work and how to use it. There are conventions to use and best practices; make sure you follow your choice of programming language's convention and best practices.

1.5.3.5 Use method-level documentation

Method documentation describes the purpose of the method and is a more specialized description than the class documentation. There are conventions to use and best practices; make sure you follow your choice of programming language's convention and best practices.

1.5.3.6 Use proper formatting

You should take care that your code is nicely formatted. No one will care to read the code if it's ugly formatted because it's a clear sign of a poorly maintained project. NSDF should choose a set of simple rules that govern the format of the code, and these rules should be consistently applied. NSDF developers should agree to a single set of formatting rules and all members should comply.

1.5.3.7 Use error handling

Make error handling clean and short: it shows what can go wrong. Use, if possible, exceptions rather than return codes. Add informative and self-contained error messages; add some *context* (e.g. in C++ **FILE**, **LINE**); classify types and gravity of errors. Use a serious and production-ready *logging system* to keep track of errors and for _post-mortem _debugging.

1.5.3.8 C++ Documentation

Doxygen is the most widely used C++ documentation tool.

The generated documentation makes it easier to navigate and understand the code as it may contain all public functions, classes, namespaces, enumerations, side notes, and code examples.

Doxygen:

- Supports a variety of output formats, including HTML and PDF.
- It can extract the code structure from undocumented source files.
- It can visualize the relations between the various elements using include dependency graphs, inheritance diagrams, and collaboration diagrams.
- supports multiple languages (C/C++, Fortran, Objective-C, C#. PHP, Python, etc.)

Several well-known C++ libraries use Doxygen for their documentation (e.g. Apache Portable Runtime, CppUnit, Free Image, GNU Standard C++ Library, KDE, LLVM, OGRE, VTK, here a full list https://www.doxygen.nl/projects.html).

But one pain point consists in the fact that generated documents tend to be visually *noisy*, with a style that struggles to represent complex template-based APIs. There are also some limitations to the Doxygen markup language.

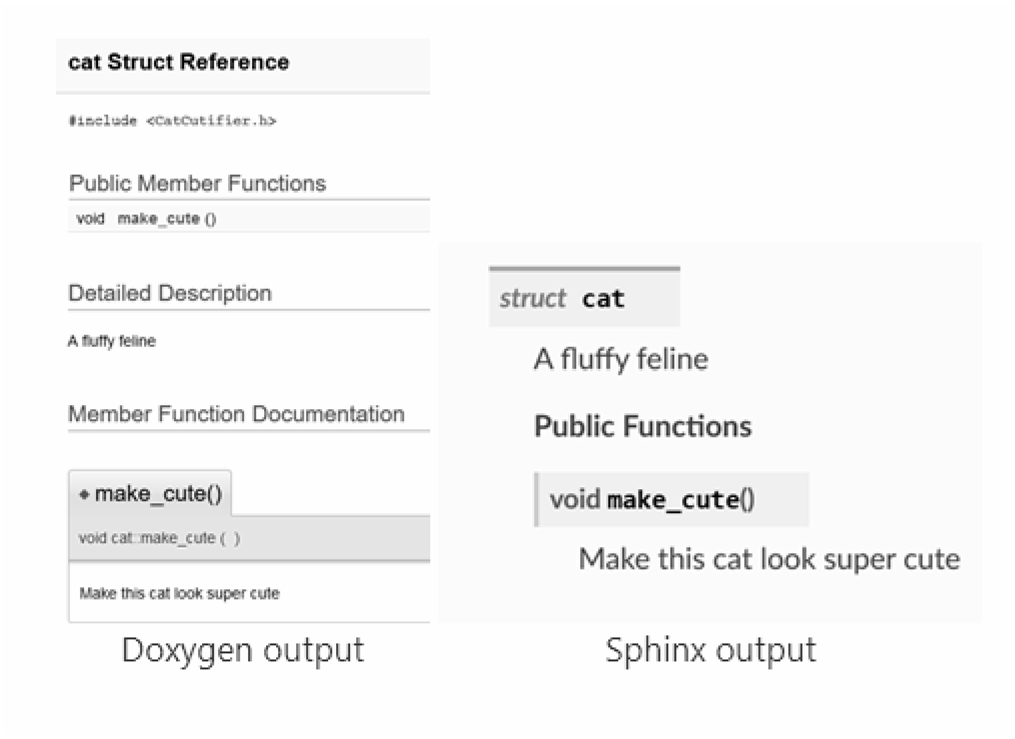
To solve this issue, the C++ community is recently switching to Sphinx, the most used Python documentation tool, that can be *adapted* to use the Doxygen parser (see https://devblogs.microsoft.com/cppblog/clear-functional-c-documentation-with-sphinx-breathe-doxygen-cmake/):

- The Sphinx module Breathe parses Doxygen XML output and produces Sphinx documentation.

- Breathe can be integrated with Read the Docs to post documentation online (cit "*Technical documentation lives here*")
- it supports hybrid syntax, i.e., using reStructuredText in Doxygen markup

The Sphinx-generated documents look more modern and minimal and it's much easier to swap to a different theme and modify the layout of the pages.

NSDF recommends using the mix Doxygen/Sphinx to document its C++ code.



cat Struct Reference

```
#include <CatCutifier.h>
```

Public Member Functions

void make_cute ()

Detailed Description

A fluffy feline

Member Function Documentation

◆ make_cute()

void cat::make_cute ( )

Make this cat look super cute

Doxygen output

struct **cat**

A fluffy feline

Public Functions

void **make_cute**()

Make this cat look super cute

Sphinx output

1.5.3.9 Python Documentation

There are several ways NSDF can produce Python documentation (see this link https://wiki.python.org/moin/DocumentationTools for an exhaustive list).

Some key factors that must influence our choice are:

- Visual appeal and ease-of-use (where case studies and/or screenshots are available)
- Potential dependency fragility (most importantly which versions of Python)
- Community size/engagement and availability of tool support
- Run-time introspection vs static analysis

Sphinx is the most used and comprehensive Python documentation generator. It supports reStructuredText in docstrings and produces an HTML output with a very clean visual style.

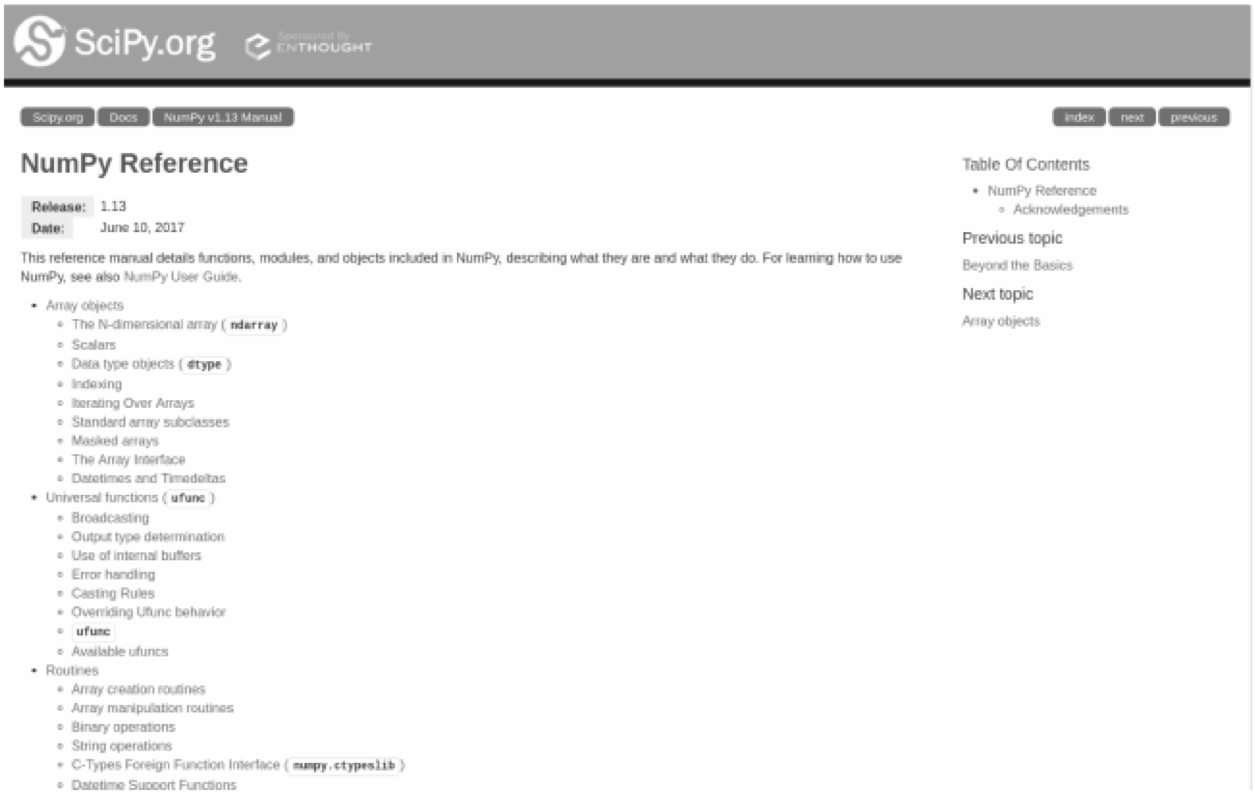
Several Python libraries are using Sphinx document generator (e.g. Flask Django, PyCuda, OpenCV, PyQt5, MatplotLib, Pandas, Conda, Pip, Pillow, PyPy, NumPy, SciPy). Python itself uses Sphinx. Full list is available here http://www.sphinx-doc.org/en/master/examples.html.

The main features of Sphinx are:

- Supports a variety of output formats, including HTML and PDF.

- Easy *cross-referencing* via semantic markup and automatic links for functions, classes, citations.
- The simple hierarchical structure of the documentation tree with automatic links to siblings, parents, children.
- Automatic indices.
- Extensible.
- Easy configuration, mostly automatic.

Setting it up requires a bit of configuration. For a quick tutorial look here.



NumPy Reference

Release: 1.13
Date: June 10, 2017

This reference manual details functions, modules, and objects included in NumPy, describing what they are and what they do. For learning how to use NumPy, see also NumPy User Guide.

- Array objects
  - The N-dimensional array (`ndarray`)
  - Scalars
  - Data type objects (`dtype`)
  - Indexing
  - Iterating Over Arrays
  - Standard array subclasses
  - Masked arrays
  - The Array Interface
  - Datetimes and Timedeltas
- Universal functions (`ufunc`)
  - Broadcasting
  - Output type determination
  - Use of internal buffers
  - Error handling
  - Casting Rules
  - Overriding Ufunc behavior
  - `ufunc`
  - Available ufuncs
- Routines
  - Array creation routines
  - Array manipulation routines
  - Binary operations
  - String operations
  - C-Types Foreign Function Interface (`numpy.ctypeslib`)
  - Datetime Support Functions

Table Of Contents
- NumPy Reference
  - Acknowledgements

Previous topic
Beyond the Basics

Next topic
Array objects

NSDF recommends the use of Sphinx to generate Python documentation. But alternatives are considered in the following table:

- pdoc
  - is probably the second-most popular Python-exclusive doc tool
  - its code is a fraction of Sphinx's complexity and the output is not quite as polished,
  - it works with *zero configuration in a single step*.
  - It also supports docstrings for variables through source code parsing. Otherwise, it uses introspection.
  - It is worth checking out if Sphinx is too complicated for the NSDF use case.
- pydoctor
  - is an API documentation generator that works by static analysis.
  - The main benefit is that it traces inheritances particularly well, even for multiple interfaces.
  - It can pass the resulting object model to Sphinx if you prefer its output style.
- Doxygen

  - is a *not Python exclusive* documentation generator.
  - It can generate documentation from undocumented source code (mostly inheritances and dependencies).
  - Many teams already know this tool from its wide use in multiple languages, particularly C++.

1.5.4 Jupyter Notebooks Documentation

A Jupyter notebook is a document that supports mixing executable code, equations, visualizations, and *narrative text*. Jupyter notebooks allow users to "*bring together data, code, and prose, to tell an interactive, computational story*".

Jupyter Notebook can combine codes and explanations with the interactivity of the application. This makes it a handy tool for data scientists for streamlining end-to-end data science workflows.

Jupyter Notebooks have played an essential role in the *democratization of data science*, making it more accessible by removing barriers of entry for data scientists.

Jupyter Notebook uses ipywidgets packages for "*live interactions with code*": code can be edited by users and can also be sent for a re-run, making Jupyter's code non-static.

Jupyter Notebook makes it easy to explain codes line-by-line with feedback attached all along the way. Users can add interactivity along with explanations, while the code is fully functional.

For all the above reasons, Jupyter Notebooks will be a very important source of documentation about the NSDF software stack, and it is the best instrument to document code and algorithms.

1.5.5 Web Services API Documentation

NSDF web services will follow the philosophy of "*microservice architecture*" that is defined as a set of loosely coupled, collaborating services.

The benefits of using such architecture include:

- Microservices are small, loosely coupled.
  - Each has its base code, meaning a smaller team is needed to manage this codebase.
- Microservices can be deployed independently.
  - This also means that services can be scaled independently as needed.
- Microservices do not need to share the same technology stack.
  - One service could be written in one language (e.g. C++); another could be written in a different language (e.g. Python).

To implement microservices NSDF recommends the RESTful API standard.

1.5.5.1 Representational State Transfer (RESTful) API

One of the most popular types of APIs for building microservices applications is known as "RESTful API" or "REST API."

REST API is a popular standard among developers because it uses HTTP commands, which most developers are familiar with and has an easy time using.

Here are the defining characteristics of RESTful API:

- An API that uses the REST (*Representational State Transfer*) model.
- Relies on HTTP coding which is familiar to web developers.

- Uses _Secure Sockets Layer (_SSL) encryption.
- It is *language-agnostic*, it connects apps and microservices written in different programming languages.
- it simplifies the creation of web applications through _Create, Retrieve, Update, Delete _(CRUD) operations

The HTTP commands, or "verbs", common to REST API include `PUT`, `POST`, `DELETE`, `GET`, `PATCH`.

Developers use these RESTful API commands to perform actions on different "resources" within an application or service. RESTful APIs use standard URLs to locate resources.

The familiarity and reliability of RESTful API commands, rules, and protocols make it easier to develop applications that integrate with applications that have an associated API.

RESTful API will be used by NSDF to make its services available to the community and to integrate with third-party applications.

1.5.5.2 RESTful documentation

RESTful APIs tend to evolve rapidly during development and release cycles.

Maintaining and updating API documentation for the development team and external users is a difficult but necessary process.

To document NSDF API, we are recommending using the *OpenAPI Specification* (OAS).

The *OpenAPI Specification* (formerly known as *Swagger Specification*) defines a standard and *language-agnostic* interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.

A consumer can understand and interact with the remote service with a minimal amount of implementation logic.

OpenAPI definitions can be written in JSON or YAML. We recommend YAML since it's also the file format used for Kubernetes deployment.

A simple OpenAPI specification looks like this:

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: NSDF API
  description: A sample API to illustrate an NSDF service
paths:
  /list:
    get:
      description: Returns a list of cloud resources
      responses:
        '200':
          description: Successful response
```

The adoption of OpenAPI will:

- Help members understand the NSDF APIs and agree on their attributes.
- Help users to experiment with the APIs (for example using the *Swagger UI*_ _open-source tool)
- Simplify the creation of automatic tests
- Accelerate the development by automatically generating *stub code* (i.e. temporary substitute for yet-to-be-developed code)

1.5.6 Links/Bibliography

List:

- Google Documentation Best Practices
- Agile in Government: Keeping Documentation Lean
- Martin Fowler Code As Documentation
- https://www.atlassian.com/agile/manifesto
- 'Documentation as code' matters. Here's why
- Agile Development Methodology: To Document or Not to Document?
- How to use Markdown for writing documentation | Adobe Experience Cloud
- Documenting API: Docs-as-code tools
- Common references and introductory content on Docs as Code
- Write the docs
- Docs like Code, change case study
- Code as Documentation
- OpenAPI Specification - Version 3.0.3
- Comparison of Python documentation generators | by Peter Kong | Medium
- Why we use Jupyter notebooks | Teaching and Learning with Jupyter
- Documentation in the modern age
- REST APIs vs Microservices: Differences & How They Work - DreamFactory
- How to Automatically Generate Clients for your REST API
- Daring Fireball: Markdown
- Markdown - Wikipedia

1.6 Continuous Monitoring

Overview of section contents:

Section	Description
Kubernetes Monitoring	Automated process to observe and detect compliance issues and security threats, especially with Kubernetes

---

Monitoring an application's state is one of the most effective ways to anticipate problems and discover bottlenecks in a production environment. Yet it is also one of the biggest challenges.

The growing adoption of microservices makes logging and monitoring a very complex problem since a large number of applications, distributed and diversified in nature, are communicating, and a single point of failure can stop the entire process.

Continuous Monitoring (CM) is an automated process to observe and detect *compliance issues _and _security threats*; it helps to monitor, detect, and study metrics, and to resolve issues in *real-time*.

CM means monitoring:

- **Infrastructure** that includes data centers, networks, hardware, software, servers, storage, etc. Common metrics to watch are: server availability; CPU; servers; system uptime; database health; disk usage; storage; security etc.
- **Applications** that track software performance. Common measures to track are availability; error rate; throughput; response time; end-user transactions; Service Level Agreement (SLA) etc.
- **Network** or network activity and related hardware (firewalls, routers, switches, servers, etc). It measures latency; multiple port metrics; server bandwidth; packets flow, etc.

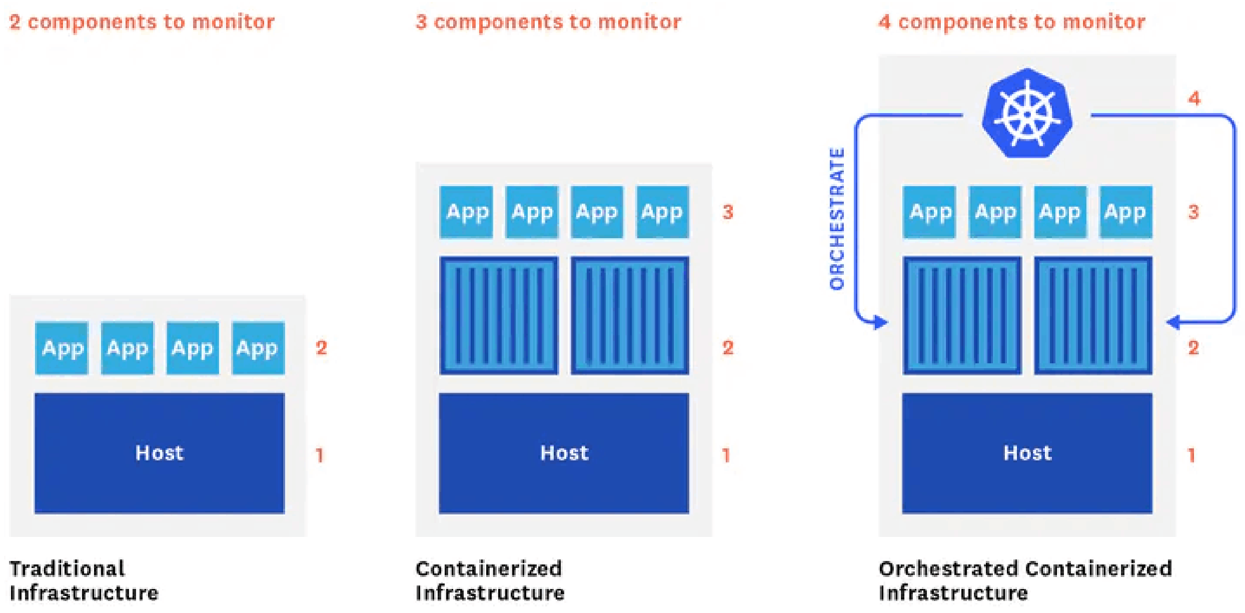
1.6.1 Kubernetes monitoring

Kubernetes runs in more than 70 percent of container environments (see Kubernetes adoption, security, and market trends report 2021). And monitoring has become a key way to extract runtime information. which is critical when troubleshooting issues and optimizing performance, both proactively and reactively.

However, Kubernetes presents a unique challenge on two fronts: **setup** and **monitoring**. It's difficult to nail the deployment in an organized and_ high-performing_ way. Common mistakes involve incorrectly sizing of nodes, consolidating containers, or properly creating namespaces.

Roughly 49 percent of containers use under 30% of their requested CPU allocation, and 45% of containers use less than 30% of the RAM (see See 10 Trends in Real-World Container Use). *Real-time monitoring* can help in preventing these problems.

Kubernetes monitoring captures logs and events from the cluster, pods, containers, host machines, and containerized applications.


2 components to monitor
3 components to monitor
4 components to monitor
App
Host
ORCHESTRATE
1
2
3
4
Traditional Infrastructure
Containerized Infrastructure
Orchestrated Containerized Infrastructure

Three types of Kubernetes metrics can be tracked:

- **Resource metrics** that include information like CPU, memory usage, filesystem space, network traffic, etc. that can be queried using the *Kubernetes Metrics API*.
- **Service metrics** include metrics produced by Kubernetes infrastructure, as well as those produced by containerized applications by deploying the Kube-state-metrics component.
- **Custom metrics** implemented using additional adapters to add metrics through the Kubernetes API aggregation.

In addition to monitoring, engineers may also want to capture *logs _and _events*. The simplest way for logging is to write logs to standard output (stdout) and standard error (stderr) streams. As these logs are created, the kubelet agent writes them into a separate file that can be accessed by the user. This process is known as *node-level logging*.

A significant problem with *node-level logging* is the instability of logs: when a Kubernetes pod terminates or moves, logs are deleted and this makes it impossible to review them after a crash.

To get around this issue, NSDF will need to set up *cluster-level logging* not natively supported by Kubernetes since it relies on additional drivers to push logs to the storage back-end.

There are two approaches to set up the _cluster-level logging _service:

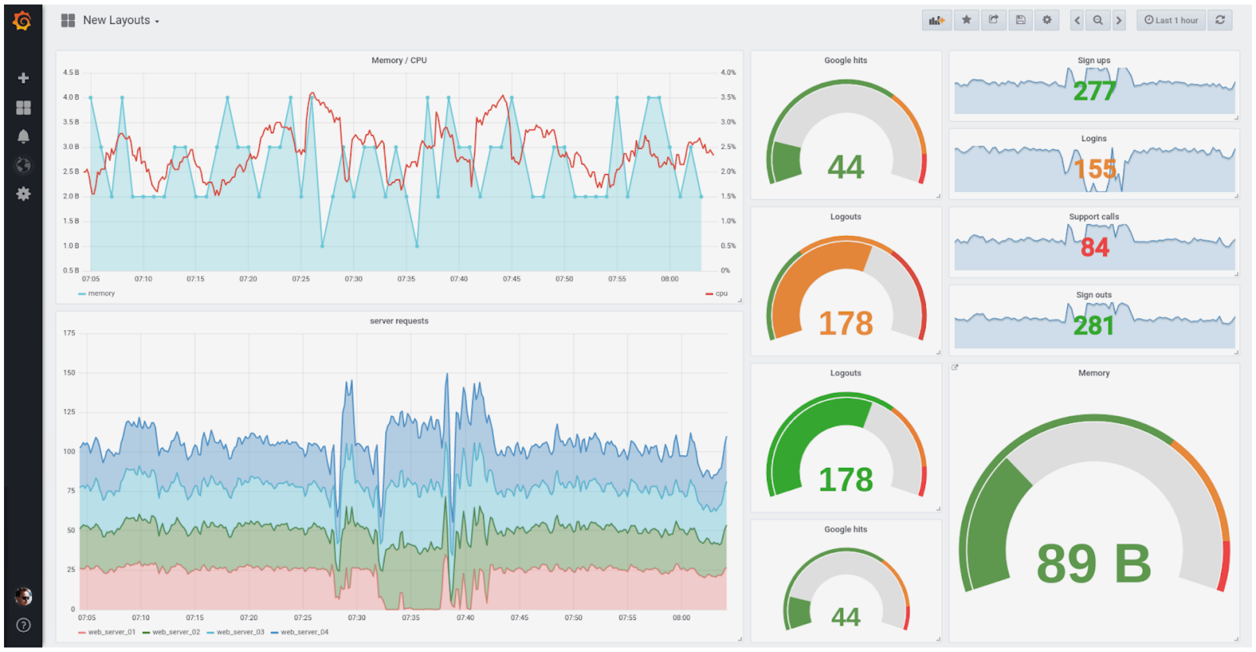
- use Cloud Native Computing Foundation (CNCF) projects. This approach has the advantage of using open-source projects and being backed up by large and active communities.
- use of *Software as a Service* (SaaS) software, usually provided by commercial companies as a_ pay-per-use _service.

1.6.1.1 Container Advisor, Prometheus, Grafana

This CNCF monitoring solution is made of a:

- Container Advisor is a monitoring tool that exposes data from running containers, including resource usage and performance metrics.
- Prometheus provides event monitoring and alerting capabilities, including data stored in the form of metrics, time-series data collection, alerts, monitoring, and querying. Prometheus has emerged in the last years as the d_e-facto open-source standard_ for Kubernetes.

- Grafana is a web application for analytics and interactive visualization. It includes charts, graphs, and alerts. There are also many *ready-to-use dashboards* in Grafana Labs; (e.g. a dashboard to control AWS costs; a dashboard to check the healthiness of an nginx load-balancer, etc).



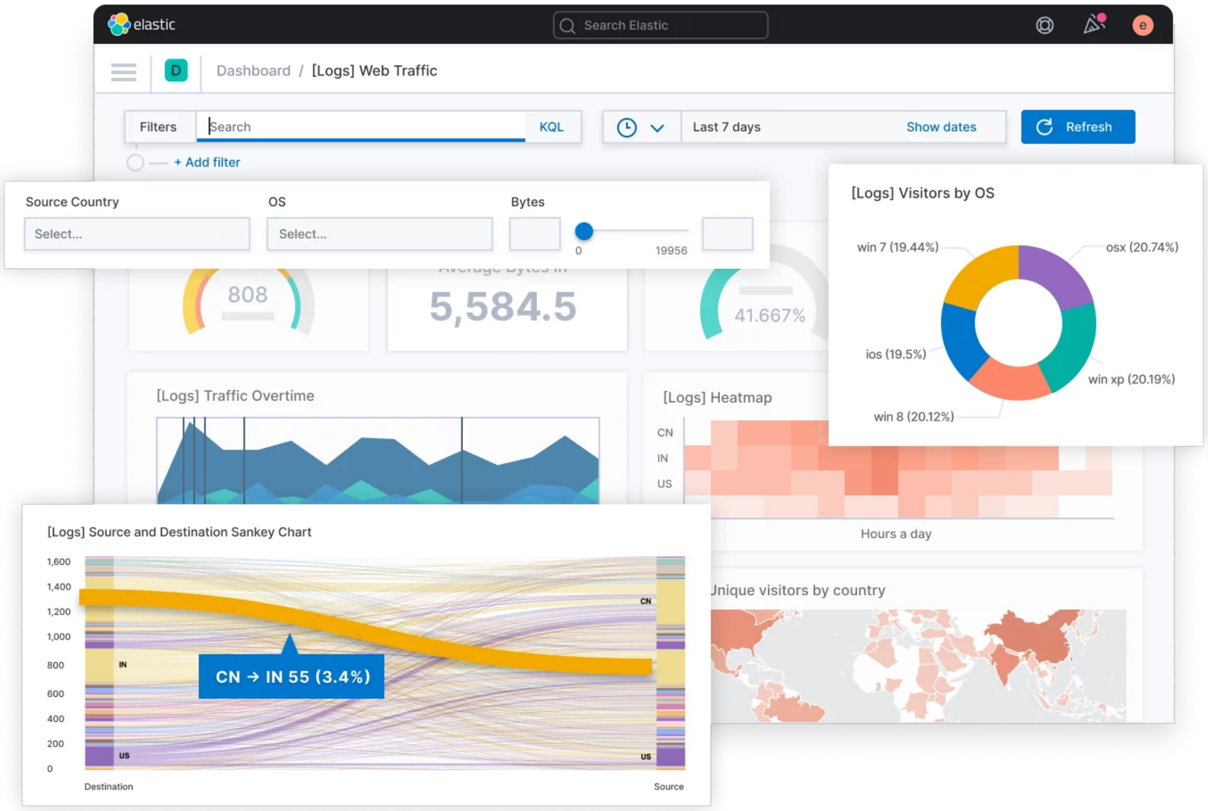
1.6.1.2 ElasticSearch, Logstash, Kibana

Another solution is to use the "*ElasticSearch + Logstash + Kibana*" (ELK) stack that is composed by:

- *Elasticsearch* is _a distributed, open-source search and analysis engine with full-text indexing capabilities, based on Apache Lucene, Data can be parsed, normalized, and enriched before being indexed.
- Logstash is a data collection engine that acts as a data pipeline. Users can aggregate logs and event data from a variety of potential sources and enrich the data with out-of-the-box aggregation and mutations.
- Kibana is a data visualization tool to create histograms, charts, graphs, and other real-time visual representations.

When *Fluentd\ is used instead of _Logstash, _then* _the stack is renamed EFK.

ELK/EFK may have some problems with increasingly log volumes.



1.6.1.3 Other K8s monitoring/Logging solutions

Some other solutions, that may be worth NSDF considerations, are:

- Sematext Monitoring is a Software as a Service (SaaS) monitoring solution for both traditional and microservice-based applications, capturing metrics and events in real-time.
- Sensu is a SaaS solution that is free for <100 nodes. It offers an end-to-end observability pipeline to collect, filter, and transform monitoring events and send them to a database. The metrics can include system metrics, as well as custom application metrics.
- Logz.io is a SaaS solution that is free for the community with 1 day of log retention (max 1GiB). It's a fully managed and autoscaling ELK stack and has some ML analytics.
- Jaeger is a free tracing tool used for monitoring and troubleshooting in complex distributed systems. It was released and open-sourced by Uber Technologies in 2016. With Jaeger, users can perform root cause analysis, distributed transaction monitoring, distributed context propagations, service dependency analysis, and performance and latency optimization.
- Kubewatch is a watcher that publishes notifications to available collaboration hubs/notification channels. Once the Kubewatch pod is running, events will start streaming to a Slack channel or other configured webhooks.
- Weave Scope is a monitoring tool that allows gaining operational insights and it allows to manage containers and run diagnostic commands within this UI.**Very strong candidate**
- Fluent Bit is a lightweight data shipper that excels in acting as an agent on edge-hosts, collecting and pushing data down the pipelines.

- Kubernetes Dashboard is a basic simple-to-setup web add-on for K8s clusters. It exposes basic metrics across all nodes and monitors the health of workloads (pods, deployments, replica sets, cron jobs, etc.).
- Lens is not a full monitoring solution, but rather a Kubernetes integrated development environment (IDE). The service bundles a contextual terminal with Prometheus statistics while ensuring that logs are easily viewable. Also, real-time graphs are available in the dashboard.

Note that NSDF excluded from the list any professional, production-grade but expensive tools such as Datadog, New Relic, Dynatrace, Instana, Turbonomic, Sysdig, Splunk, etc.

1.6.2 Links/Bibliography

List:

- Top 13 Kubernetes Monitoring Tools
- Top 11 Open Source Monitoring Tools for Kubernetes
- Kubernetes Monitoring: Best Practices, Methods, and Solutions
- Kubernetes Monitoring Dashboards
- Kubernetes Monitoring - A Simplified Guide
- What is Continuous Monitoring in DevOps?
- Monitoring in the Kubernetes Era | Datadog

1.7 Software Security

Overview of section contents:

Section	Description
Enforce security of sensitive data	Use vault-like solutions to store sensitive data, such as passwords, security tokens.
Enforce security of continuous integration	ways to prevent the leaking of sensitive information when using Git, such as passwords, tokens.
Enforce security of the continuous deployment	Measurements to perform even when the CD tools may have failed to automatically remove the credentials
Security scans	Security scan to avoid vulnerabilities, particularly Python and Docker images
Automatic tools	The automatic tools to improve software quality

This section contains some best practices to keep the NSDF software stack *safe and secure*.

1.7.1 Enforce Security of sensitive data

The best approaches to store and share passwords, security tokens, sensitive information is:

- Use **2-Step verification_**, also known as *multi-factor authentication* (MFA) for all users: even if some password WAS stolen, it's almost impossible to log in without a *second device*
- Create_ **short-lived tokens** for any automatic procedure. This means to:() *create the token and limit as much as possible its scope ()* use the token for a limited timeframe (*) remove the token while the activity is finished
- Always follow the **Principle of Least Privilege** i.e. do not create security tokens with _Full Access _permissions.
- Configure a **strong password policy** for users, and rotate keys (link)

To store sensitive data we suggest using a vault-like solution such as Confidant or HashiCorp Vault, both free for open source projects.

1.7.2 Enforce Security of Continuous Integration

Stealing passwords and tokens in a Git public repository can be as easy as running a single line command (link from "Ethical Hacker") which dumps the contents of a repository's object database :

```
{ find .git/objects/pack/ -name "*.idx"|\
while read i; \
do git show-index < "$i"|awk '{print $2}';done;f\
ind .git/objects/ -type f| \
grep -v '/pack/'| \
awk -F'/' '{print $(NF-1)$NF}'; }|while read o;do git cat-file -p $o;done|\
grep -E 'pattern'
```

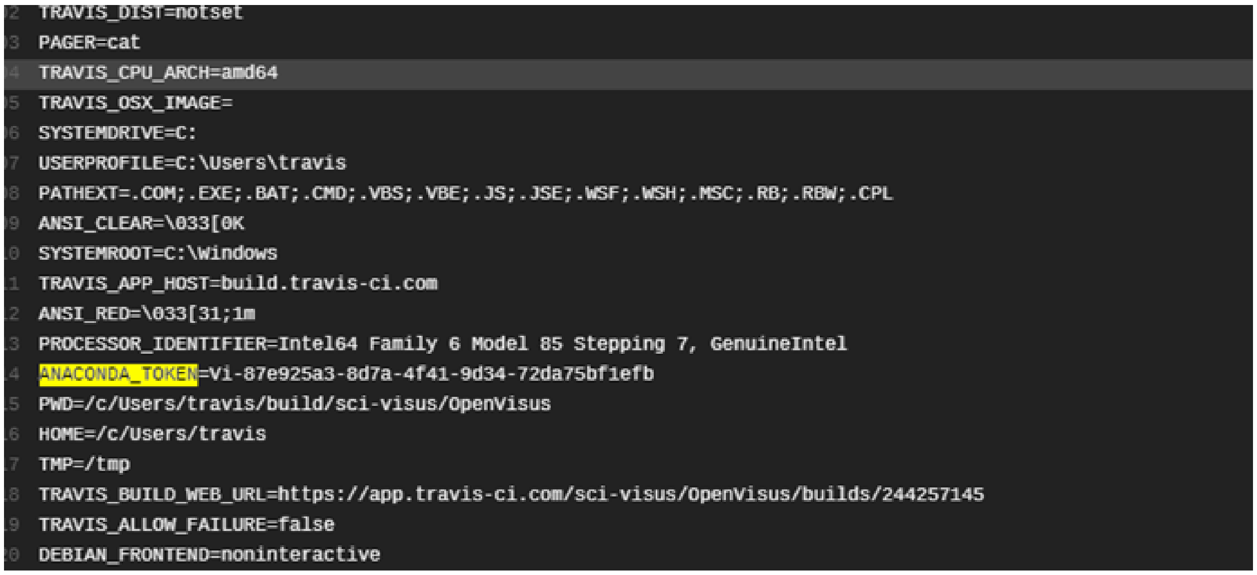
The best approaches for these kinds of security leaks are:

- **Before pushing** to GitHub: use truffleHog or gitleaks (or alternatives) to search for secrets, to dig into commit history and branches, to find secrets accidentally committed.
- **Before pushing** to GitHub: clean up Jupyter Notebooks output or other clean-text documents using nb-clean (or alternatives)
- **After pushing** to GitHub: use online open-source scanning solutions such as Spectral

1.7.3 Enforce Security of Continuous Deployment

All CD tools automatically filter secure *environment variables and tokens r_emoving them from the _build log* and replacing them with some *obfuscated* text. But, once a VM is booted and tests are running, all these tools have less control over what information utilities or add-ons can print to the VM's standard output.

As an example, just running an `env` command from TravisCI, secrets will be publicly exposed (see Travis CI Vulnerability Potentially Leaked Customer Secrets).



```
TRAVIS_DIST=notset
PAGER=cat
TRAVIS_CPU_ARCH=amd64
TRAVIS_OSX_IMAGE=
SYSTEMDRIVE=C:
USERPROFILE=C:\Users\travis
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.RB;.RBW;.CPL
ANSI_CLEAR=\033[0K
SYSTEMROOT=C:\Windows
TRAVIS_APP_HOST=build.travis-ci.com
ANSI_RED=\033[31;1m
PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 85 Stepping 7, GenuineIntel
ANACONDA_TOKEN=Vi-87e925a3-8d7a-4f41-9d34-72da75bf1efb
PWD=/c/Users/travis/build/sci-visus/OpenVisus
HOME=/c/Users/travis
TMP=/tmp
TRAVIS_BUILD_WEB_URL=https://app.travis-ci.com/sci-visus/OpenVisus/builds/244257145
TRAVIS_ALLOW_FAILURE=false
DEBIAN_FRONTEND=noninteractive
```

Possible countermeasures are:

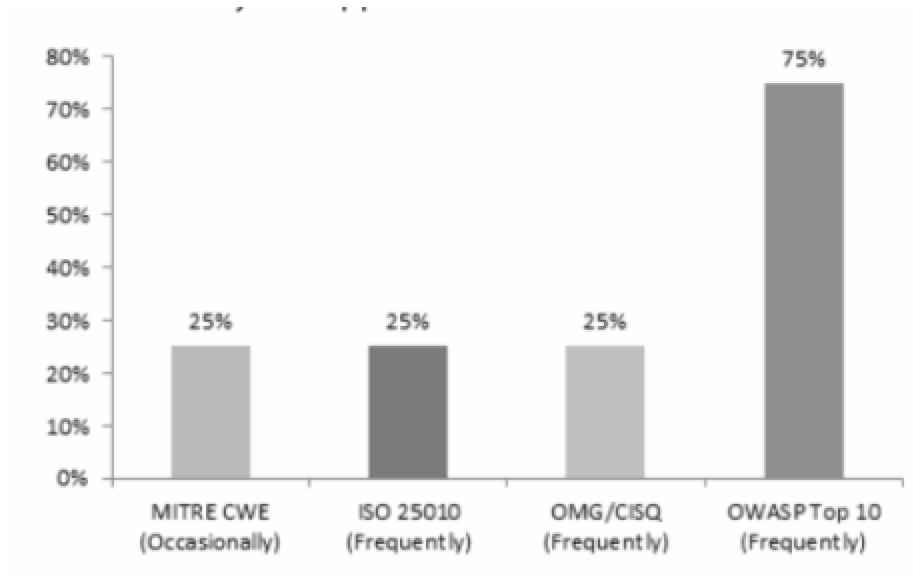
- **Drastic** countermeasures:
  - do a manual deployment and/or move your CI/CD pipeline on-premise.
- **Soft** countermeasures:
  - disable all settings which duplicate commands to standard output (e.g `set -v),
  - disable any displaying environment(e.g. `env` or `printenv`)
  - avoid printing secrets within the code(e.g. `echo $SECRET_KEY`)
  - avoid any shell commands that may expose tokens or secure variables (e.g. `git fetch` or `git push`)
  - avoid mistakes in string escaping
  - avoid increasing command verbosity if not strictly necessary
  - redirect output to /dev/null
  - rotate secrets periodically
- **Off-the-shell** countermeasures:
  - use standalone solutions to do the online CD scanning

1.7.4 Security Scans

NSDF must constantly check source code for the most exploited security **weaknesses and vulnerabilities** in software including the:

- 2021 CWE Top 25 Most Dangerous Software Weaknesses
- OWASP Top Ten Web Application Security Risks

This process should be automatic, ideally running on every new commit (i.e. several times /day), thus limiting the manual intervention to periodic full checks



Also, NSDF recommends finding **Python security issues** using Bandit (or alternatives):

> *Bandit is a tool designed to find common security issues in Python code. To do this Bandit processes each file builds an AST from it and runs appropriate plugins against the AST nodes. Once Bandit has finished scanning all the files it generates a report.*

and performing *safety checks* onP ython dependencies using:

- Safety: a command-line tool to check virtual environment, requirement files, or any input from stdin for dependencies with security issues),
- Jake: a tool to check for your Python environments and applications that can report on known safety vulnerabilities),
- consider alternatives.

Finally, NSDF recommends adding **Security scans of Docker Images**.

Several security leaks are related to the `Docker` technology:

- Security problems with the container image and the software running inside
- Security problems regarding the interaction between a container, the host operating system, and other containers on the same host
- Security problems related to the host operating system
- Container networking and storage

To avoid such leaks, the simplest solution is to *scan* images during development (i.e. _run `docker scan`, Clair, Anchore, Dagda, Docker Benc before the Git pushing).

Also, consider following best practices from Docker official documentation:

- Each container should have only one responsibility.
- Containers should be immutable, lightweight, and fast.
- Don't store data in your container.
- Containers should be easy to destroy and rebuild.
- Use a small base image (such as Linux Alpine). Smaller images are easier to distribute.
- Avoid installing unnecessary packages. This keeps the image clean and safe.

- Avoid cache hits when building.
- Auto-scan your image before deploying to avoid pushing vulnerable containers to production.
- Scan your images daily both during production

Automatic scan on Docker Hub *after the deployment* is restricted to commercial accounts.

1.7.5 Automatic tools

See the "Software Quality/Tools" paragraph.

Almost all the automatic software quality tools also provide automatic checks for security and vulnerabilities.

1.7.6 Links/Bibliography

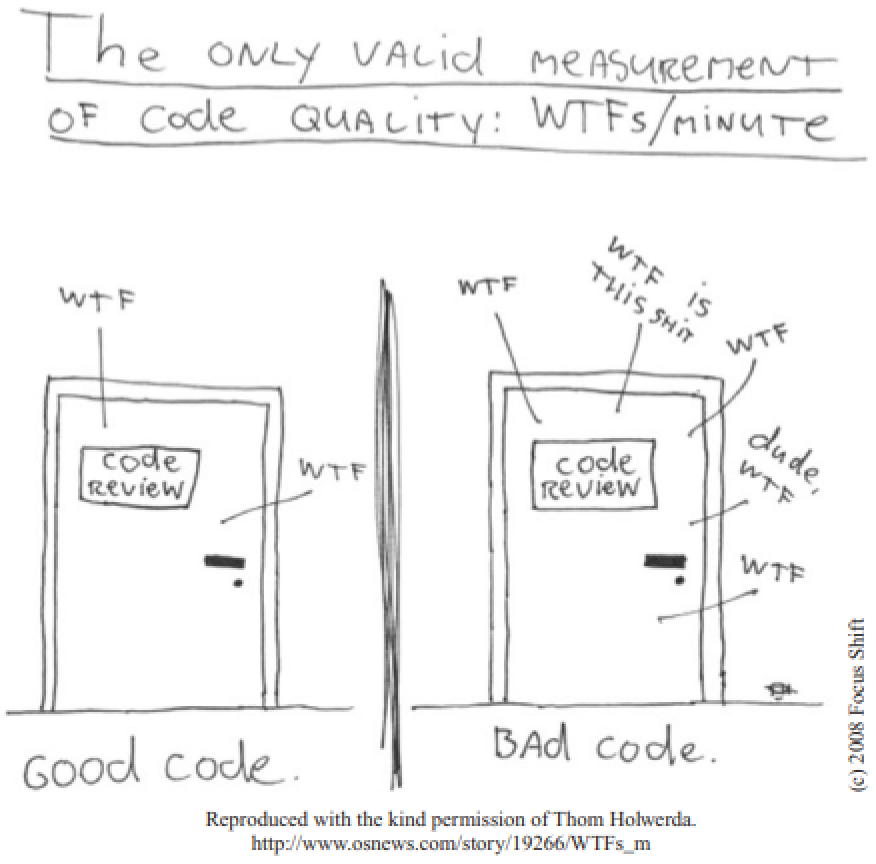
List:

- Plugging Git Leaks: Preventing and Fixing Information Exposure in Repositories
- How to Choose a Secret Scanning Solution to Protect Credentials in Your Code
- About code scanning
- Best Practices in Securing Your Data - Travis CI
- Security scanners for Python and Docker: from code to dependencies
- Scanning your Conda environment for security vulnerabilities
- Best practices for scanning images
- 10 Container Security Scanners to find Vulnerabilities
- https://geekflare.com/secret-management-software/

1.8 Software Quality

Overview of section contents:

Section	Description
Reliability	How NSDF builds reliable code
Efficiency	How NSDF builds efficient code
Security	The practices that NSDF takes to ensure secured code
Maintainability	The practices that NSDF takes for code maintenance
Rate of Delivery	NSDF adopts the agile software development approach
Automatic tools	NSDF utilizes the CI/CD pipeline tool for checking software quality

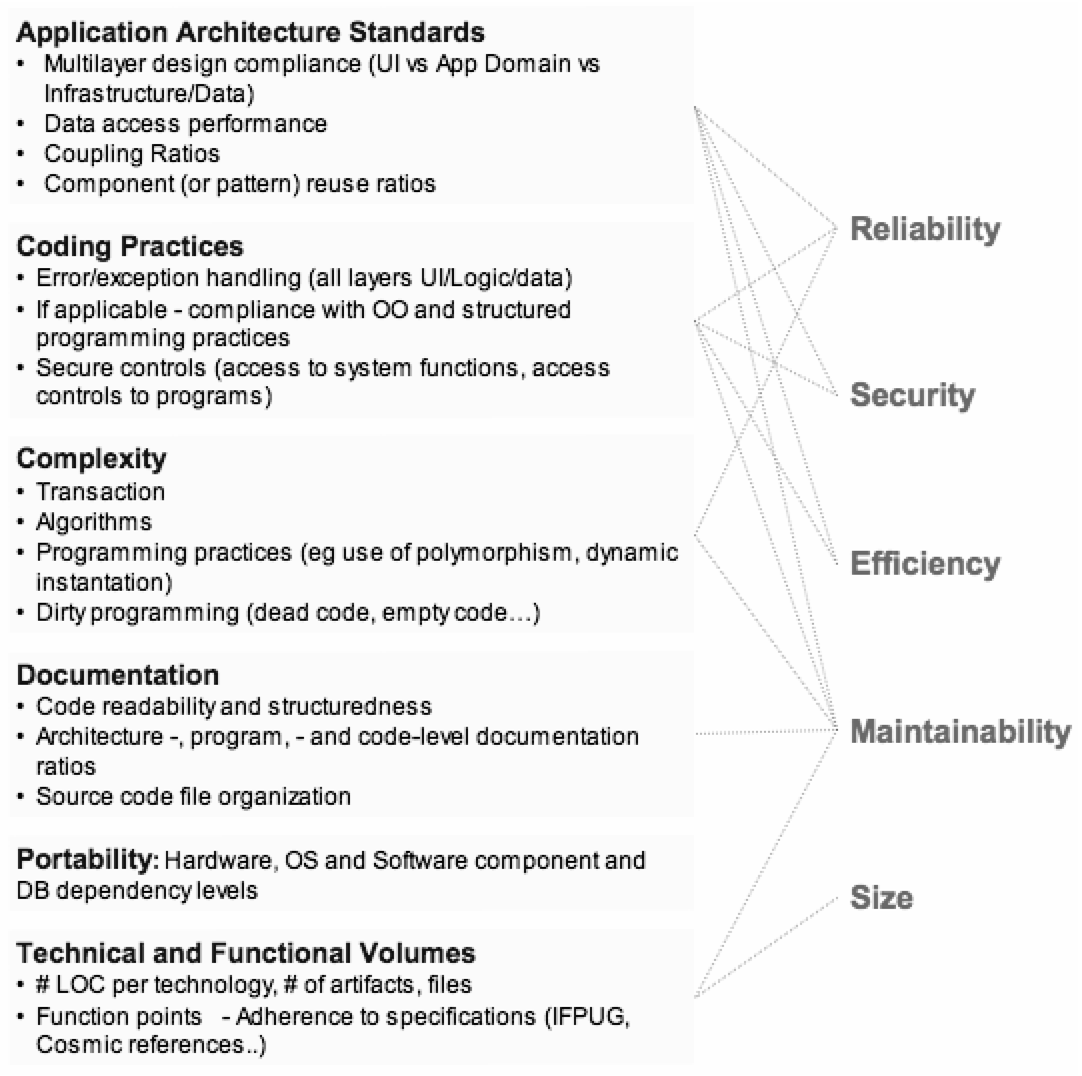

The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude,
WTF
WTF
BAd code.
(c) 2008 Focus Shift
Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

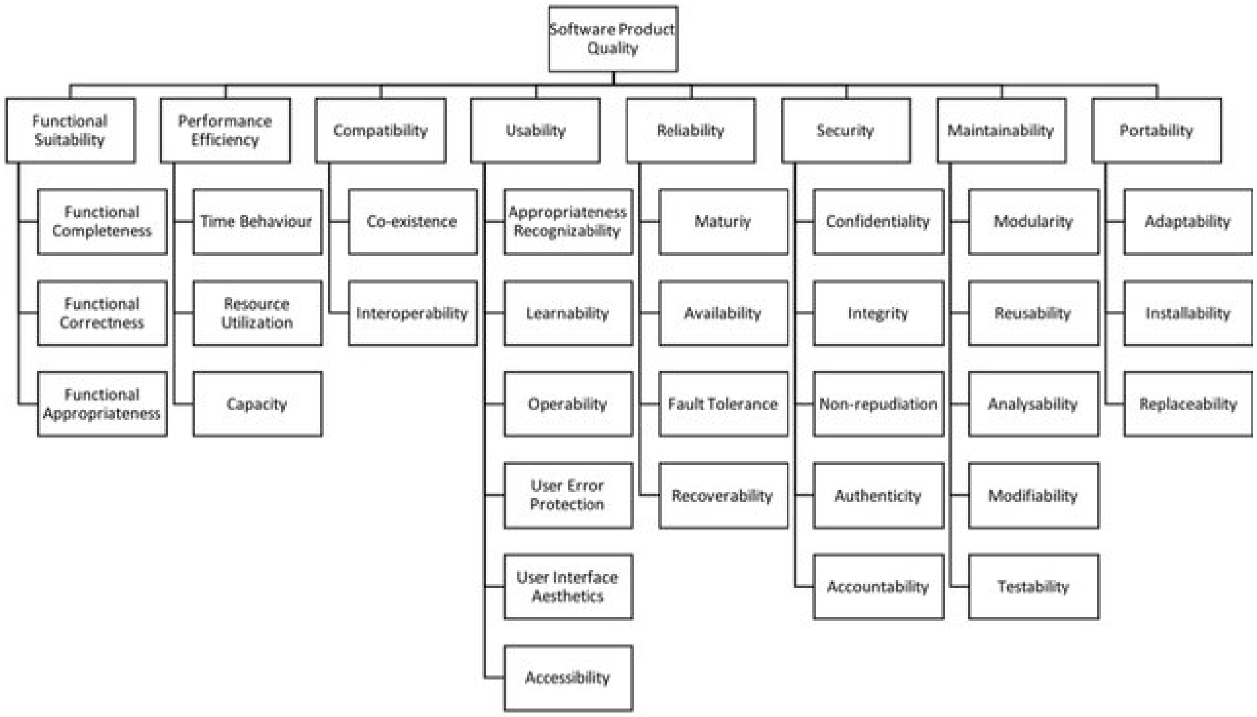
Software Quality is a very complex and broad argument, and it's out of the scope for this document to cover it in detail.

There are several methods for evaluating software quality, and choosing the right one is a difficult task as well since it depends on the software product, the project objectives, and the context of use.

Experts agree that *high-quality code* uses coding conventions, is readable and well documented, is reusable and avoids duplication, handles errors diligently, is efficiently using resources, includes unit tests, and complies with security best practices.

In the following sections, we use the Software Quality definition given by the *Consortium for Information & Software Quality* (CISQ), a group that develops standards for automating the measurement of software size and structural quality.


Application Architecture Standards
• Multilayer design compliance (UI vs App Domain vs Infrastructure/Data)
• Data access performance
• Coupling Ratios
• Component (or pattern) reuse ratios
Coding Practices
• Error/exception handling (all layers UI/Logic/data)
• If applicable - compliance with OO and structured programming practices
• Secure controls (access to system functions, access controls to programs)
Complexity
• Transaction
• Algorithms
• Programming practices (eg use of polymorphism, dynamic instantation)
• Dirty programming (dead code, empty code…)
Documentation
• Code readability and structuredness
• Architecture -, program, - and code-level documentation ratios
• Source code file organization
Portability: Hardware, OS and Software component and DB dependency levels
Technical and Functional Volumes
• # LOC per technology, # of artifacts, files
• Function points - Adherence to specifications (IFPUG, Cosmic references..)
Reliability
Security
Efficiency
Maintainability
Size



1.8.1 Reliability

Reliability refers to the level of risk inherent in software, and the likelihood it will fail.

Reliability also addresses *stability*, defined as how likely there will be *software regressions* (i.e. bugs where a feature that has worked before stops working) if changes are made.

A synonym is "resilience" defined as the software's ability to deal with failures; for example, modern *microservices-based applications* can be automatically redeployed in case of crashes, making this type of architecture highly resilient.

Metrics to measure reliability are:

Name	Description
Production incidents	are the number of high-priority bugs identified in production.
Reliability testing	for example *load testing*, which checks how the software functions under high loads; or *regression testing*, which checks how many defects are introduced when software is deployed. The aggregate results of all these tests are a measure of resiliency.
Reliability evaluation	is a set of tests that run in a simulated environment. It is also a measure of how the software will work in a steady-state or under certain growth (e.g. more users or higher throughput).
The average failure rate	measures the average number of failures over a certain period.
Mean-time between failures	is a metric used to measure _uptime, _defined as the time software is expected to work correctly until the next failure.

NSDF builds *Reliable Code* by:

- being transparent about security beaches. Any incident will imply writing an *Incident Report* with all the details and countermeasures. Particular emphasis should be dedicated to the "*What could have been handled better?*"

section, to improve the overall process and improve the reliability.

- Add automatic testing in all SDLC phases. See the "*Continuous Testing*" paragraph for more details.
- Add a _staging/pre-production environment _to be used for installing, configuring, migrating code before going to production. See the _"Kubernetes"_ paragraph.
- Add monitoring and event logging to keep track of uptimes and failures. See the "*Continuous Monitoring*" paragraph for more details.

Some services can benefit from making use of "Chaos Engineering" such as automated by ChaosMonkey (https://netflix.github.io/chaosmonkey/) which randomly kills service instances so developers automate recovery procedures.

1.8.2 Efficiency

The most important elements that contribute to an application's performance are related to how source code is well (or poorly) written; its internal structure; what components are running (e.g.databases, web servers, load balancers, caching nodes), and how they integrate/communicate, etc.

Scalability is also a key aspect to evaluate performance: systems that can *horizontally scale up and down*, automatically adapting to different levels of required performance.

To measure the efficiency we use:

Name	Description
Load testing	is conducted to understand the behavior of the system under certain loads.
Stress testing	to understand the upper limit of the capacity of the system.
Soak testing	to check if the system can handle a sustained load for a prolonged period, and if/when performance will start to degrade.
Application performance monitoring	is a category of software tools that provide metrics and insights specifically for performance.

NSDF will build *Efficient Code* by:

- Adding automatic tests to all the SDLC to create a history of measures_(e.g._ code performance, network flows, storage IOPS, transactions per second, etc). See the "*Continuous Testing*" paragraph for more details.
- Adding Monitoring and Event Logging to the software stack, to detect performance issues as early as possible. See the "*Continuous Monitoring" paragraph _for more details.*_
- Adding elastic computing with Kubernetes deployments and health checks for moving loads away from faulty nodes. See the "*Kubernetes*" paragraph for more details.

1.8.3 Security

Security reflects how likely attackers might breach the software, interrupt its activity, or gain access to sensitive information, due to poor coding practices.

Software security can be measured as

Name	Description
The number of vulnerabilities	found by scanners.
Time to resolve problems	how long has it's taken from the discovery of a vulnerability until the fix
Deployment of security updates	how often patches and security updates are deployed
Security incidents	defined as the severity, number, and total time of attacks.

NSDF develops secure code by:

- Following best practices as described in the "*Software Security*" paragraph
- Add automatic vulnerability scanners to the CI/CD pipeline
- Share security incidents knowledge in "Incident reports"
- Collaborate closely with security experts.

1.8.4 Maintainability

Software Maintainability is the ease with which software can be adapted to other purposes, how portable it is between environments, and whether it is *transferable* from one team to another.

Maintainability is closely related to code quality. If the code is of high quality, the software is likely to be more easily maintainable: it will take less time and cost to adapt it to changing requirements. The maintainable software is also more likely to have improved reliability, performance, and security.

We measure maintainability by:

Measure	Description
Lines of code	Software with more lines of code tends to be more difficult to maintain and more prone to code quality issues.
Static code analysis	is an automatic examination of code to identify problems and ensure the code adheres to industry standards.
Software complexity metrics	we measure code complexity using widely-adopted algorithms (e.g. *cyclomatic complexity* and *N-node complexity)*.

NSDF develops *Maintainable Code* by:

- Using code reviews to share the knowledge and promote good code. See the "*Continuous Integration*" paragraph for more details.
- Adding automatic static code analysis and software metrics to the CI/CD pipeline. See the "*Continuous Integration*" paragraph for more details.
- Adding code coverage and automatic testing. See the "*Continuous Testing*" paragraph for more details.
- Adopting an efficient online documental system to better explain the software to both internal developers and external stakeholders. See the "*Continuous Documentation*" paragraph for more details.

1.8.5 Rate of Delivery

The rate of software delivery is related to quality because a new version of a software system will typically contain improvements. A higher frequency of releases should mean, at least in theory, that the user gets better software faster.

In agile development, new iterations of software are very quick: in Continuous Delivery/ Deployment software is usually shipped every day, or even several times a day.

To measure the rate of software delivery, we use:

Measure	Description
The number of software releases	i.e. how frequently new software is produced.
Agile stories number	of "user requirements" shipped over a certain period
User consumption	of releases measures the number of downloads/installation of patches or software updates.

NSDF speeds up the rate of delivery by:

- adopting best practices of the Agile (Programming) philosophy.

1.8.6 Automatic Tools

NSDF suggests introducing in the CI/CD pipeline an online tool for checking software quality (see Comparison of Automated Code Review Tools: Codebeat, Codacy, Codeclimate, and Scrutinizer).

The prices of the solutions presented in the below table should be double-checked to be viable for the NSDF pilot:

- SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on 20+ programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities. SonarQube is meant to be integrated on-premise, while SonarCloud is meant to be integrated with cloud solutions.
- Codacy automates code reviews and monitors code quality on every commit and pull request reporting back the impact of every commit or pull request, issues concerning code style, best practices, security, and many others. It monitors changes in code coverage, code duplication, and code complexity. Saving developers time in code reviews thus efficiently tackling technical debt. JavaScript, Java, Ruby, Scala, PHP, Python, CoffeeScript, and CSS are currently supported. Codacy is static analysis without the hassle.
- Code Climate is a well-developed and very stable solution with a great number of features. It supports a great number of programming languages, it is used by big players (e.g. New Relic, jQuery), and has a well-maintained test coverage
- Codecov Can show code coverages for many languages including Java, Python, and Javascript. Shows unified coverage and separate coverage for matrix builds.

1.8.7 Links/Bibliography

List:

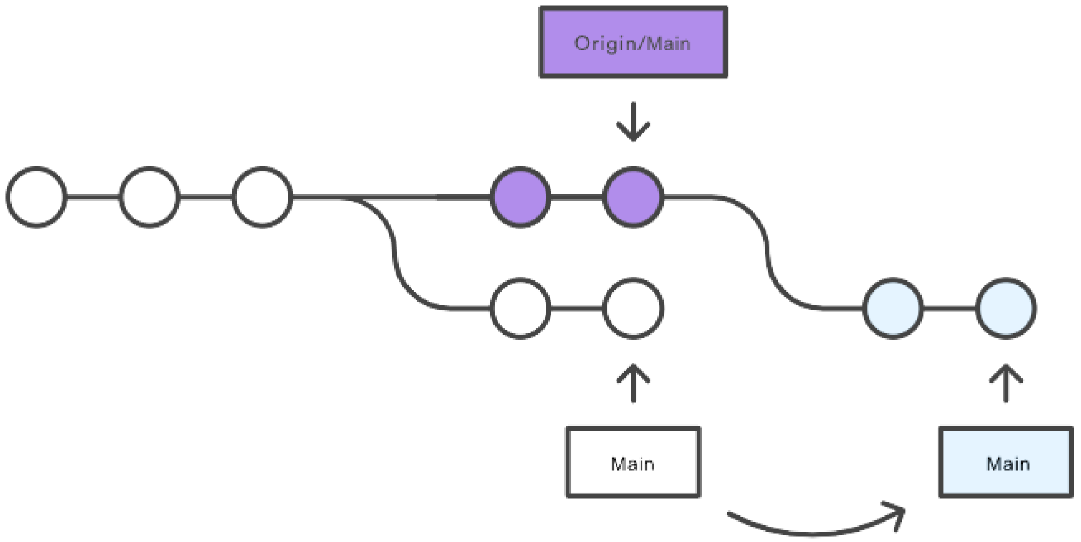
- State of the Industry Report on Software Quality Analysis
- The Cost of Poor Software Quality in the US: A 2020 Report
- A Comprehensive Guide to Measuring Software Quality
- Software quality - Wikipedia
- Defining and Measuring Software Sustainability: Towards an Empirical Framework for Evaluation at the Architectural Level
- Software Quality Standards
- Google's SRE book-Incident Response
- Comparison of Automated Code Review Tools: Codebeat, Codacy, Codeclimate, and Scrutinizer.

1.9 Appendix I. Git Workflows

1.9.1 Centralized Workflow

The *Centralized Workflow* uses a central repository to serve as the single point-of-entry for all changes to the project. The default development branch is called main and all changes are committed into this branch.

This workflow doesn't require any other branches besides main.



In the figure above, we show a typical example of a centralized workflow:

- the user "**purple**" develops, commits, and pushes changes to the central repository (Origin/Main).
- Meanwhile, user "**white**" is developing and committing locally. His changes are temporarily _on-hold _since he is in bug-fixing mode.
- When user **'white' **tries to push its changes remotely, he receives an error because he is out-*of-sync _with the central repo.*_
- He does a `git pull --rebase`, to merge the new changes; he eventually resolves conflicts, and he pushes the new changes to the origin

In the end, this workflow works as there was just one development continuous line.

1.9.2 Feature Branching Workflow

Feature Branching (also known as *Task Branching*) is a logical extension of Centralized Workflow.

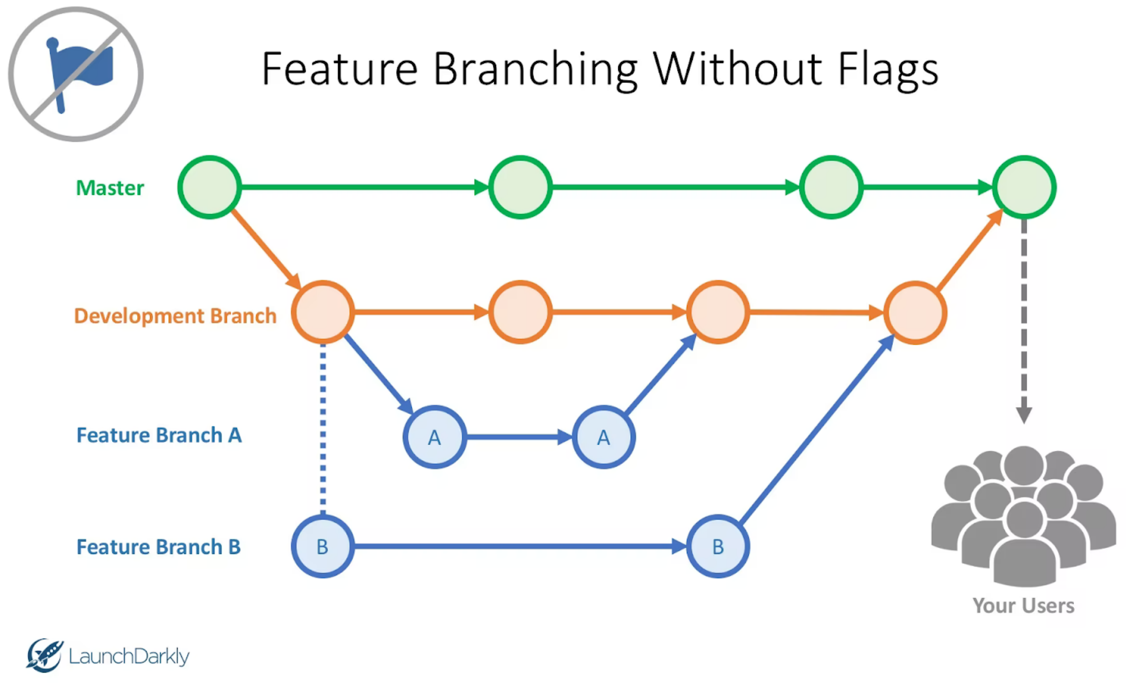
The idea behind "Feature Branches" is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easier for multiple developers to work on a particular task without disturbing the central codebase.

These branches are often referred to as_ user stories_.

The main branch should never contain broken code, which is a huge advantage for continuous integration environments.

Feature Branching also helps developers _easily segment _their work: instead of tackling an entire release, they can focus on a small set of changes.

Feature branches can also be divided up according to specific *feature groups*. Each team or sub-team could maintain its branch for development. once finished, changes can be tested and reviewed before the final integration.



1.9.3 Gitflow Workflow

The Gitflow Workflow \was first published in a highly regarded 2010 blog post from Vincent Driessen at nvie. This workflow doesn't add any new concepts or commands beyond what's required for the \Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact.

Gitflow consists of two branches, main (formerly named master) and develop:

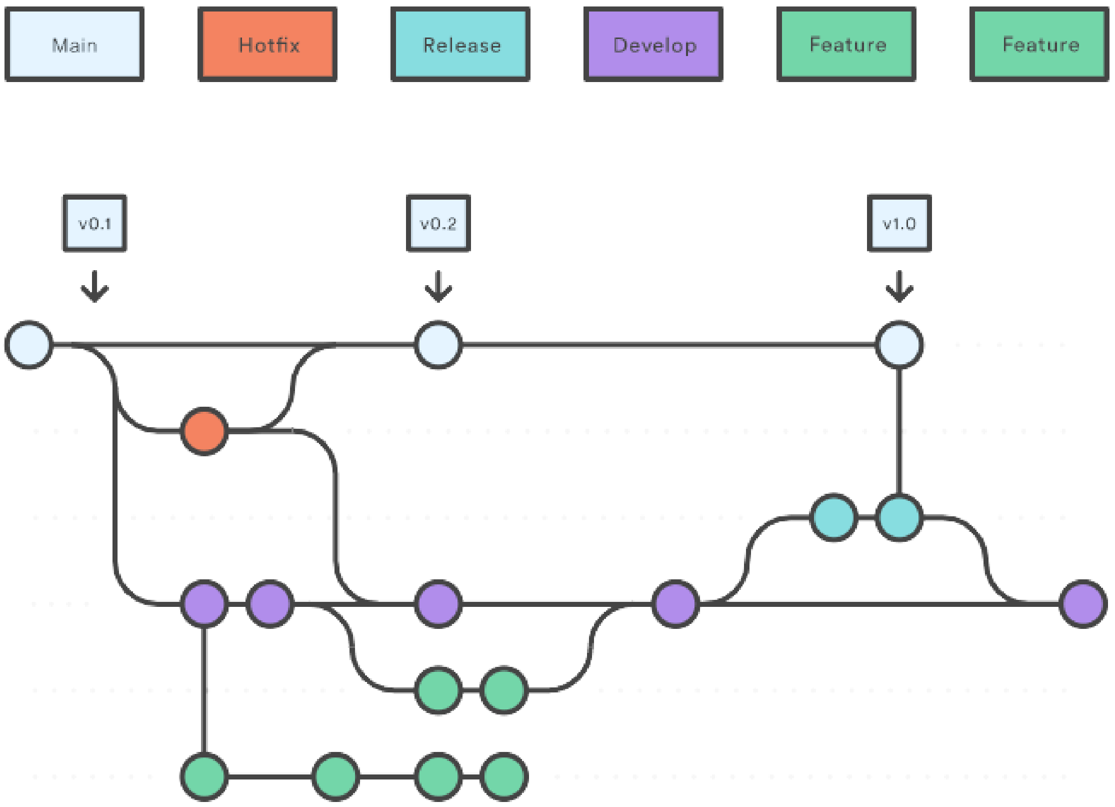
- the *main branch* is for production-ready code
- the *develop branch* is for work-in-progress code

In preparation for new releases, the development branch is tested until it is declared stable and merged into the main branch.

New features are checked out from the `develop` branch; developed on relative branches; and merged back into develop after code reviews.

New releases are created by checking out the develop branch to *a new release branch for major releases or* merged into an existing release branch for minor releases.

Hotfix branches are created for urgent fixes. They are checked out from the main branch and merged into both the `main` and `develop` branches.

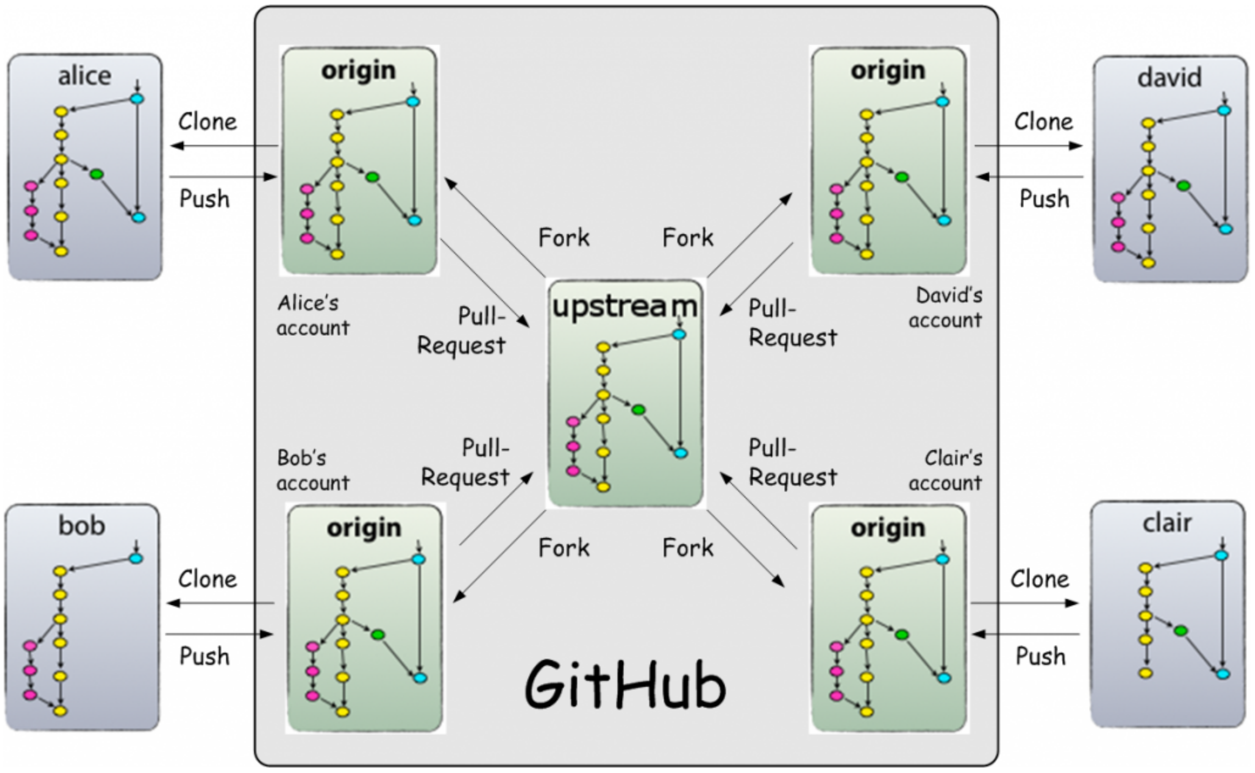


1.9.4 Forking Workflow

Instead of using a single server-side repository to act as the "central" codebase, *Forking Workflow* requests every developer to have their server-side repository.

This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

The *Forking Workflow* is most common in public open-source projects since contributions can be integrated without the need for everybody to push to a single central repository: developers push to their server-side repositories, and only the *project maintainer* has the rights to push to the official repository. This allows the _project maintainer _to accept commits without granting write-access.



1.9.5 GitHub Flow

GitHub flow simplifies the use of Gitflow: the main branch contains the production-ready code and new features are developed using _feature branches that are merged into the main branch after a (1) pull request, (2) code review (3) integration tests.

GitHub flow doesn't use develop, release, or hotfix branches.

> *GitHub Flow has some of the same elements as Git Flow, such as feature branches. But unlike Git Flow, GitHub Flow combines the mainline and release branches into a "master" and treats hotfixes just like feature branches. (https://www.freshconsulting.com/insights/blog/git-development-workflows-git-flow-vs-github-flow/)*

GitHub Flow provides almost all the functionality that *Git Flow*, but is scaled back for a more adaptable workflow with slightly less overhead.