MicroCODE Consulting Services
Application Development Notes

{2012-05} MicroCODE Software Support Process v012.docx

Subject: Application Development and Maintenance Process
Audience: MicroCODE Customers
Purpose: Explanation of recommended App support process
Author: Tim McGuire
Last Updated: Wednesday, November 26, 2025

Background
Developing complex manufacturing software applications, and supporting time critical manufacturing, requires a proactive and highly responsive software support process.

The Development Circles Model
MicroCODE sees application development as a set of concentric 'Circles'*****. An application 'Circle' consists of the following elements:

- A code base used to build all parts of a solution
- A documentation set describing the use of the code base
- A team, or allocated time for personnel within a team, to support the Circle activities

The following application Circles must be supported concurrently, at a minimum:

1. Internal Development Circle (ALPHA)

Git Branch: **develop**
Purpose: The 'next major release' currently being developed
Status: Pre-production code, integration branch for new features
Activity: Long-term development work
Testing: Unit and Integration testing

2. Plant Pilot Circle (BETA)

Git Branch: **release/*** (long-lived branches)
Purpose: The 'current major release' being tested and supported in selected customer sites
Status: Pilot-ready 'Production Candidate'
Activity: Multi-group testing before production promotion
Requirements: Minimum of three (3) sites, varying environments, to exit Pilot phase
Testing: System Acceptance Testing (**SAT**) and User Acceptance Testing (**UAT**)

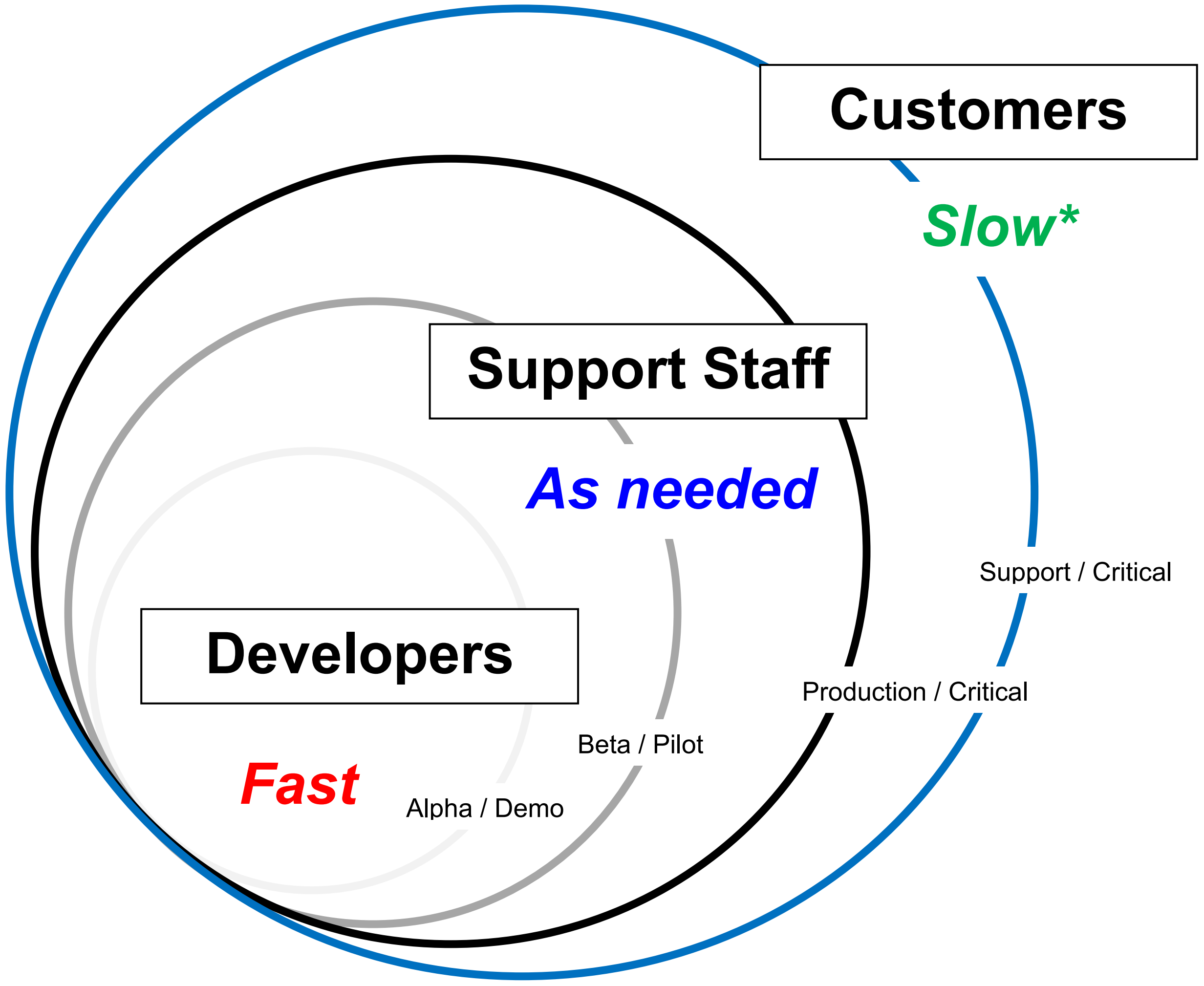
3. Plant Production Sites (PRODUCTION)

Git Branch: **main**
Purpose: The 'previous major release(s)' running in actual Production sites
Status: Production-ready, stable releases
Activity: Multiple versions may be in the field based on plant upgrade schedules
Support: All versions must be supported concurrently until End of Life

* **Note**: In 2017 I discovered that Microsoft refers to these 'Circles' as 'Rings', see…
https://blogs.office.com/2015/08/12/managing-office-365-updates/

Design

Application development must be viewed as parallel activities in order to respond to customers in a timely fashion. At the outside there are the **Customers** who are actively using the application, possibly multiple versions of it. On the inside are the **Developers** working to implement new features and functionality. And, between the two are **Support Staff** to control access to the expensive resources, prioritize defect resolutions, and provide the Customers with immediate work-around or training to get past non-application issues.

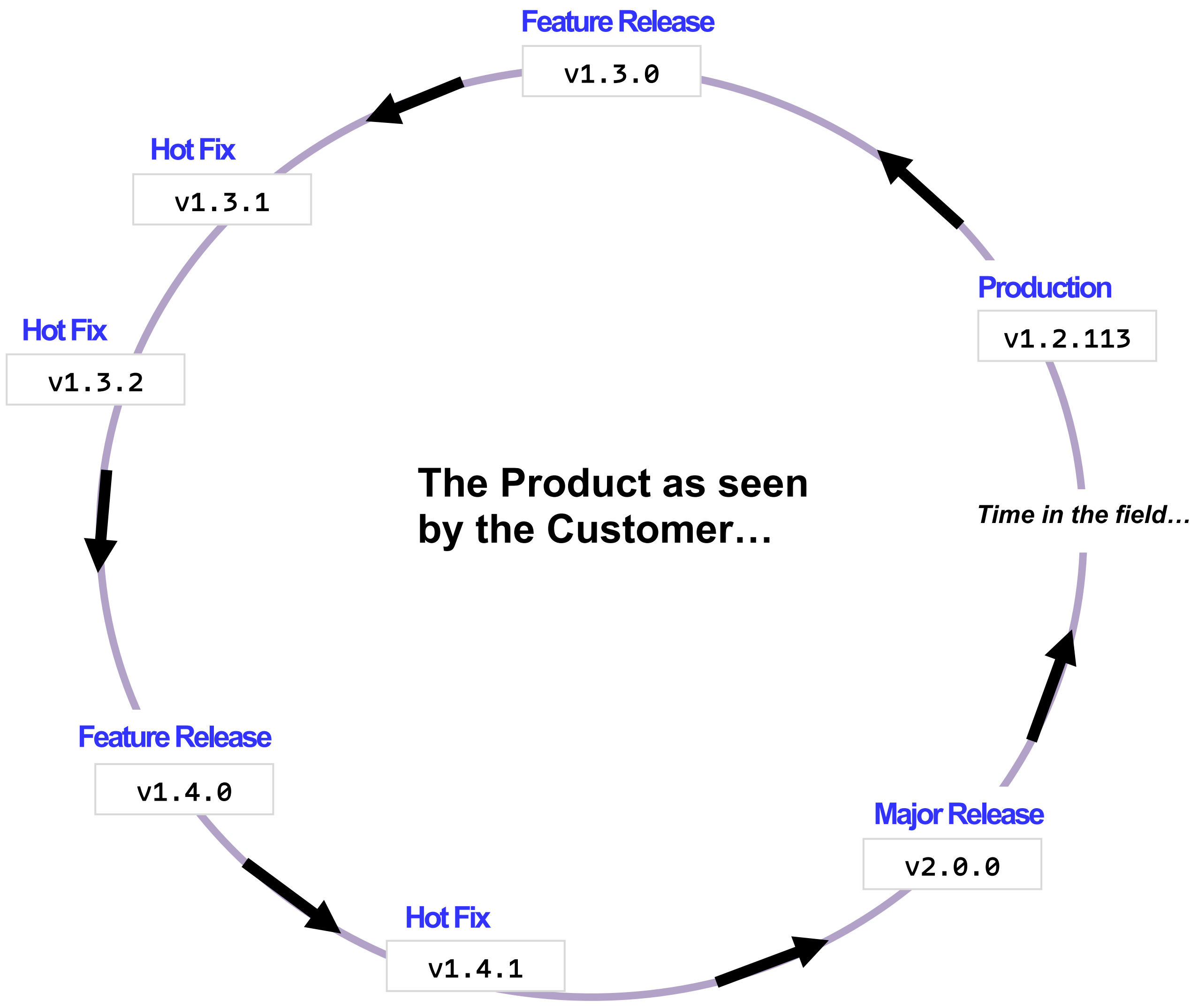


* **Slow**, except for critical Defect Resolution (‘Hot Fixes’) which must be fast and override other development priorities.

* **Support**, except for critical Defect Resolution (‘Hot Fixes’) which must be fast and override other development priorities.

Activities

Why represent each version as a ‘Circle’?
Because each released version will go through a cycle of multiple builds over its lifetime as a result of a healthy customer support process. This will continue until a Release goes to ‘End of Life’ and support for that particular Release is terminated, with fair warning given to all customers of course.



Defect Severity Definition

In order to prioritize defect resolution a severity must be associated with every System Issue logged by the Support Staff, and the severity must be agreed upon with the Customer / Production Site. By definition every call into the Support Staff must be logged as a Support Issue (e.g.: A 'Ticket Process').

These are MicroCODE's definitions, impact statements, and recommended responses.

Severity	Description	Impact	Response
1	System crash, complete loss of a major system component	Lost Units or loss of vehicle integrity	Immediate defect resolution and Build Release (**Hours/Days**)
2	Function fails, no work-around is possible, or missed defect or buildable job that was not able to be error proofed	Lost Units or loss of vehicle integrity	Immediate defect resolution and Build Release (**Hours/Days**)
3	Function fails, work-around is possible, or opening erroneous defects on vehicles	Loss of vehicle integrity	Defect resolution and Build Release (**Days/Weeks**)
4	Function fails, no impact to Production, no work-around is necessary, or defect is caused by reconfiguring a Station during Production	Local Support required	Defect resolution in next Minor Release (**Weeks/Months**)
5	Display, Report or Tool problem - no impact on system operation	Possible time lost in the future due to poor information	Defect resolution in next Minor Release (**Weeks/Months**)
6	Annoyances	Loss of resource time, delays	**Business Case** required for resolution

Once severities and responses are defined then support processes can be defined and be held up to those standards for customer support reviews.

Examples of these processes are described follow the next pages…

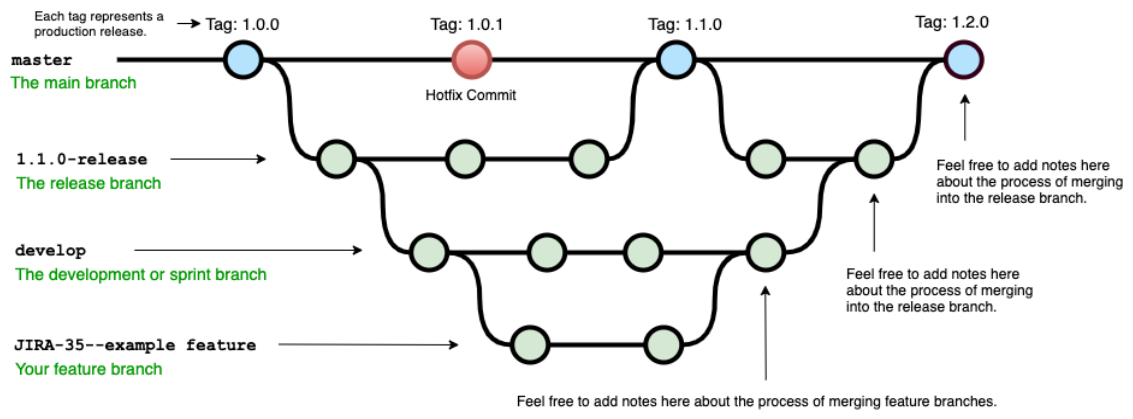
Git

Git is the de facto standard for source code control. The beauty of Git is its flexibility, the danger of Git is its flexibility. Because it is so flexible it is very easy to branch yourself into oblivion without a well-defined process.

Example Git Branching Diagrams

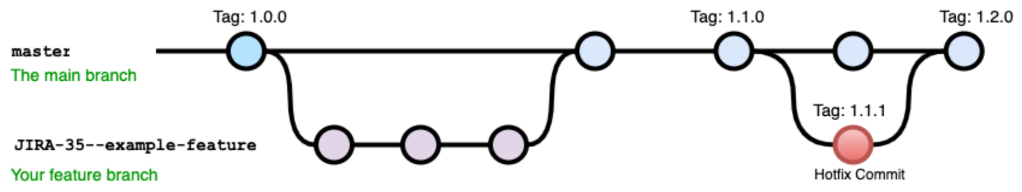
Example diagram for a workflow similar to "Git-flow" :

See: https://nvie.com/posts/a-successful-git-branching-model/


Each tag represents a production release.
Tag: 1.0.0
Tag: 1.0.1
Tag: 1.1.0
Tag: 1.2.0
master
The main branch
Hotfix Commit
1.1.0-release
The release branch
Feel free to add notes here about the process of merging into the release branch.
develop
The development or sprint branch
Feel free to add notes here about the process of merging into the release branch.
JIRA-35--example feature
Your feature branch
Feel free to add notes here about the process of merging feature branches.

Example diagram for a workflow with a simpler branching model:

See: https://gist.github.com/jbenet/ee6c9ac48068889b0912 or https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow



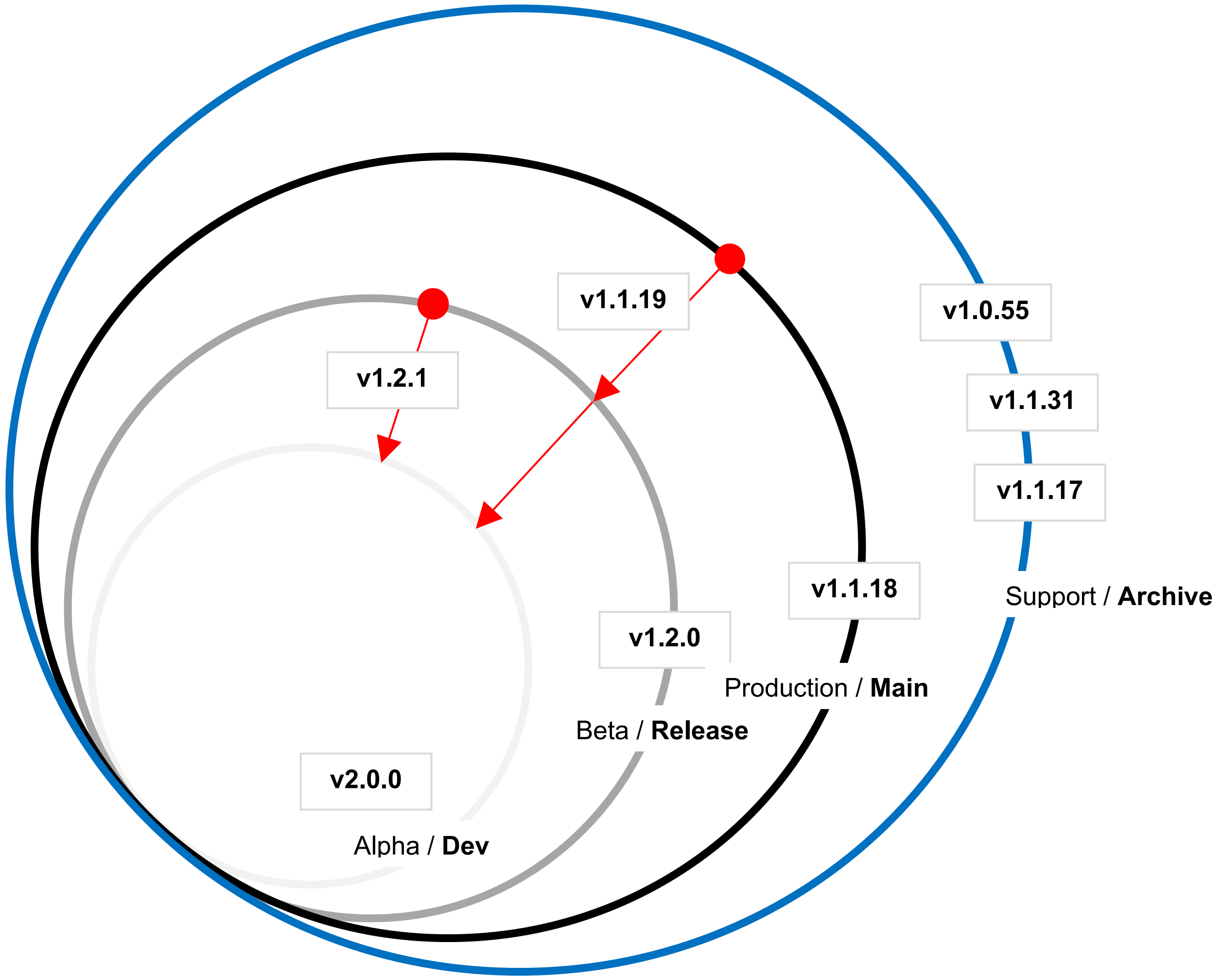
Companies have established internal methodologies for using Git, and some open-source communities have done so as well, one well established method is called **Gitflow**. This is method is available as an extension to **Git** to automatically enforce the rule set.

A good tutorial on **Gitflow** is available here:
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

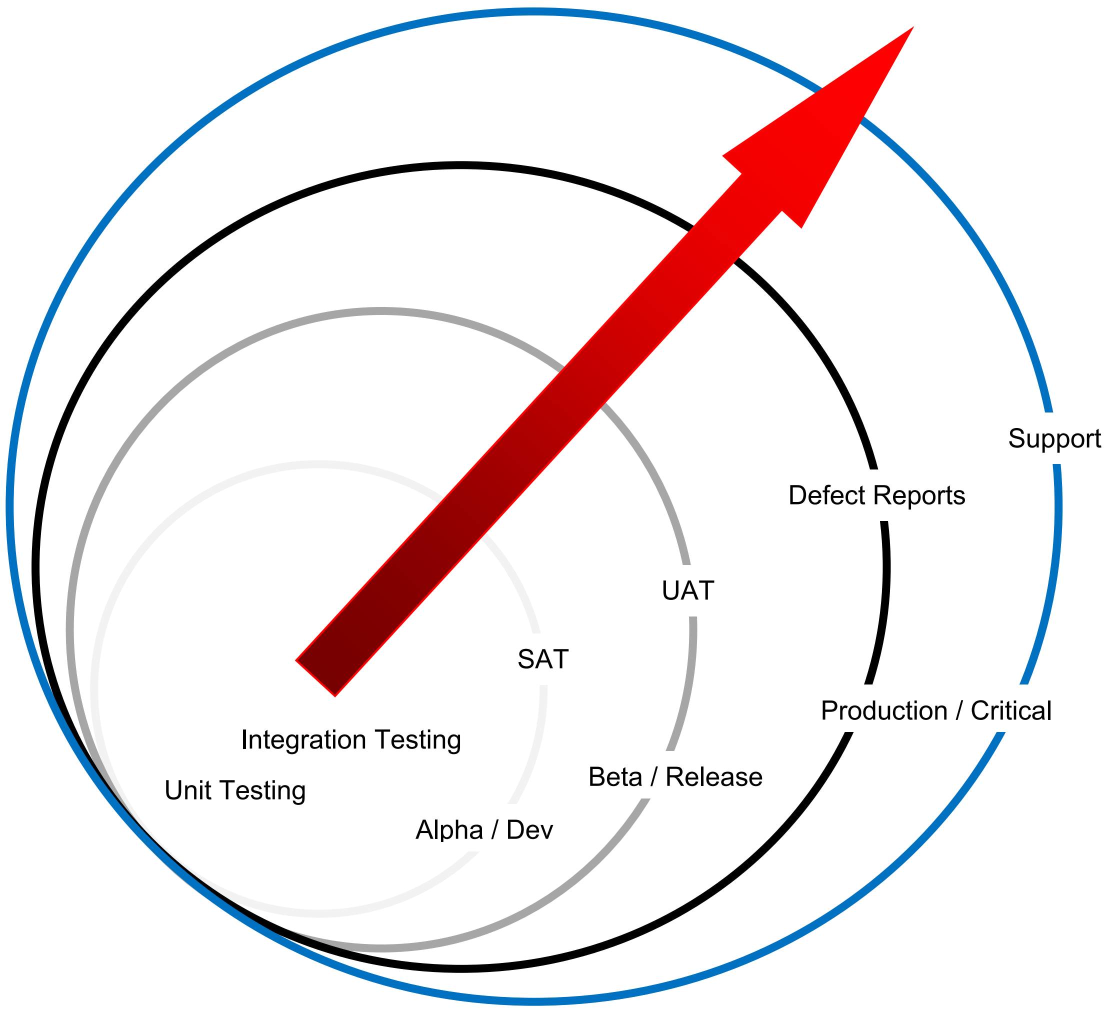
Version Control

Version control must support the parallel activities described on the previous pages. Versions in Production must be able to be patched for critical defects or critical enhancements to meet the needs of the customers. *Example app version numbering is shown below.*

The **red arrows** illustrate the need to fix defects **first in the version where they arise**…creating a new **Patch** number (nnn)—immediately shipping the resolution into Production (**main**)—and then, **carrying that defect resolution down into all newer code Circles**, possibly with different implementations, or simple checking that the defect does not exist in the newer Builds.



Risk, Expense, and Urgency
The further out you go in the Circles greater the risk and expense associated with a defect or defect correction.



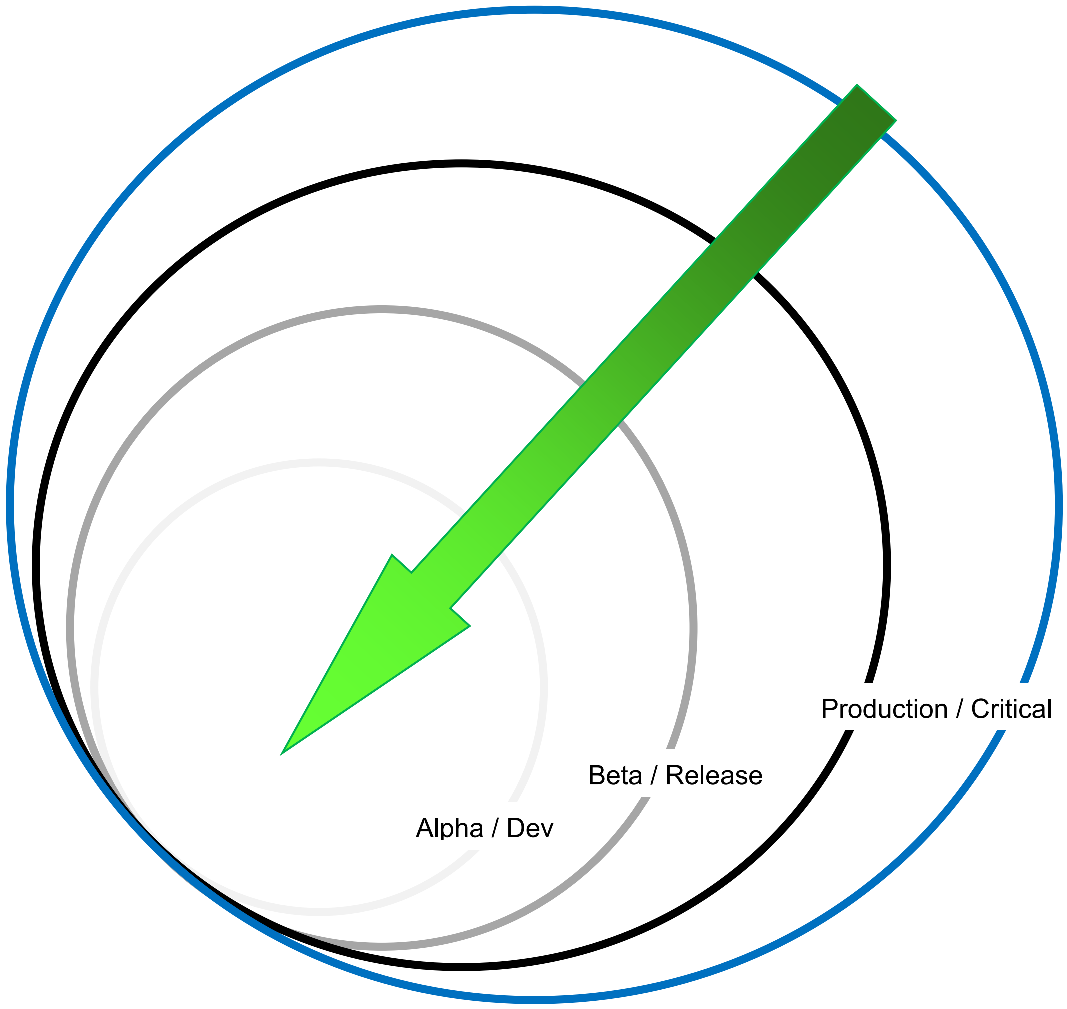
Unit Testing: This happens at the application code level; or, said another way, in the code used to write a standalone program or application. These tests are run by the developer, or development-pair, in the **Alpha**, **Beta**, **Feature** and **Hotfix** Builds.

Integration Testing: Is conducted to evaluate the compliance or interactions of a system components (or whole systems) with specified functional requirements. These tests are run by the development teams in the **Alpha**, **SPR, and SER** Builds.

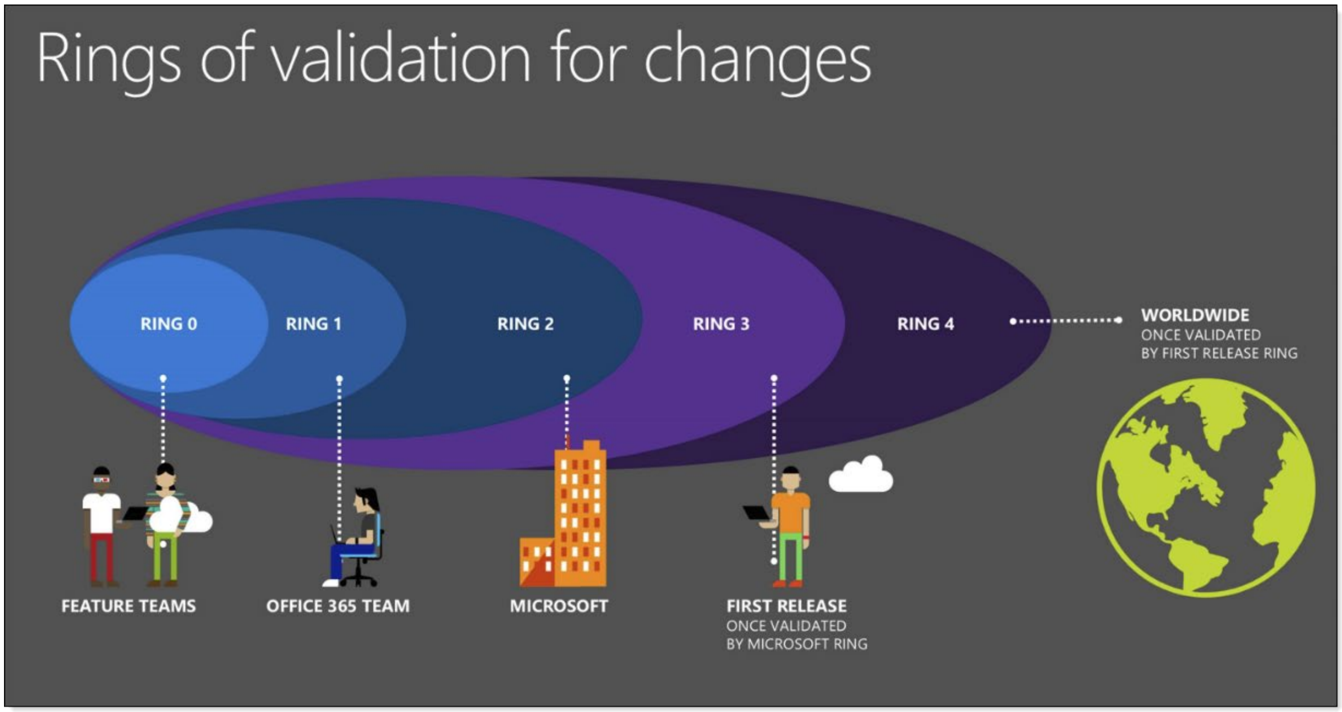
System (Acceptance) Testing (SAT): Validates the complete and fully integrated software product, checking to see if the software works the way they say it is supposed to. These tests are run by the development and support teams in the **Beta** Builds before release to the Pilot Site.

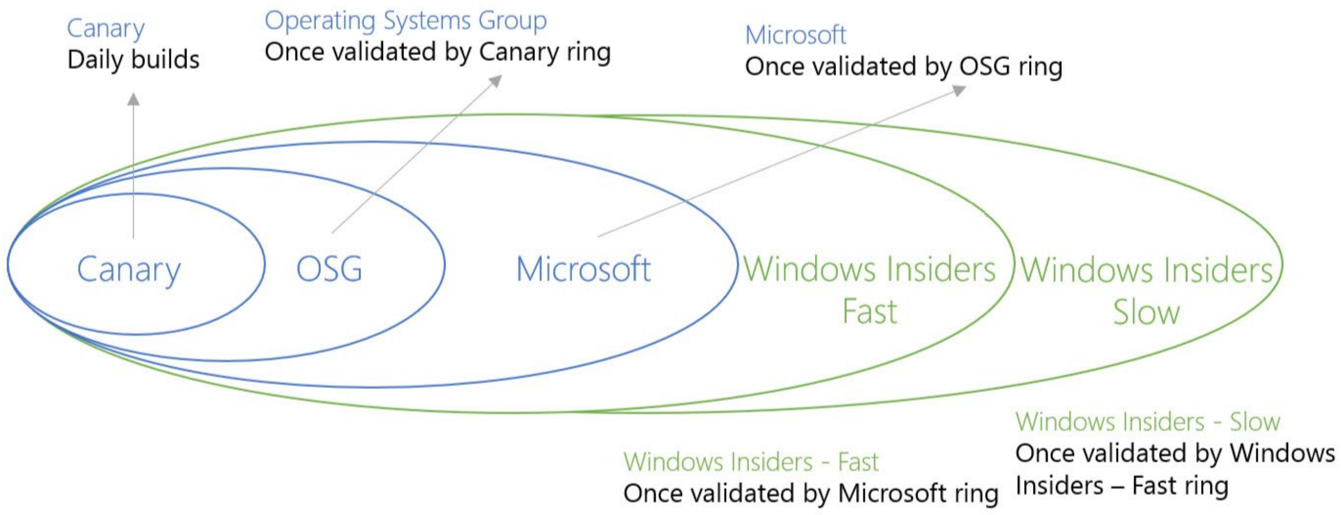
(User) Acceptance Testing (UAT): evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery. These tests are run by the support and customers teams at the Pilot Site in the **Beta** Builds.

Flexibility, Time, and Options
The further into the Circles you go the greater the time and flexibility associated with any software change.



I've been drawing these pictures since the 1980's, here is Microsoft's version in 2017:

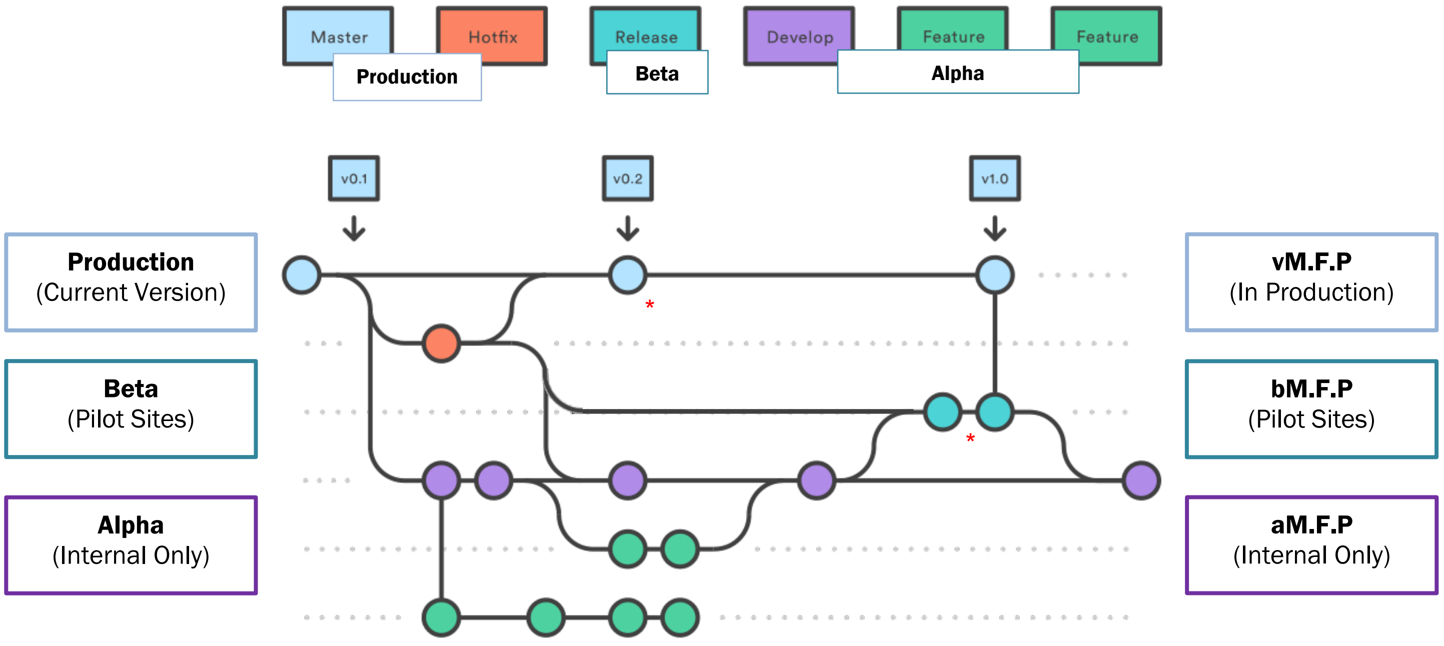

Rings of validation for changes
RING 0
RING 1
RING 2
RING 3
RING 4
WORLDWIDE
ONCE VALIDATED BY FIRST RELEASE RING
FEATURE TEAMS
OFFICE 365 TEAM
MICROSOFT
FIRST RELEASE
ONCE VALIDATED BY MICROSOFT RING



GIT FLOW & MDS

Gitflow has a defined process very similar to MicroCODE's.
This is exactly model I have worked from and evangelized since the 1980s.
Our Git Process, which combines 'Git Flow' and our established process is called the MicroCODE Development System (**MDS**) internally. Step-by-Step instructions for using Gitflow to implement our internal process—**MDS**—is found in Appendix A.

Branch Type	**Gitflow**	**MDS** (1980s-1990s Process)
trunk	**main**	PRODUCTION – Builds running at customer sites
branch	**hotfix/***	SPR – Software Problem Report (Defect or Critical Change)
branch	**release/***	BETA (Builds running internally and/or at Pilot Sites)
branch	**hotfix/***	SPR – Software Problem Report (Defect or Critical Change)
trunk	**develop**	ALPHA, DEVELOPMENT (internal only)
branch	**feature/***	SER – Software Enhancement Request (new features)



* Not shown in the diagram above is the ability to **Hot Fix** a **Beta** Release.

vM.m.r -1
(In Production)

vM.m.r -2
(In Production)

These represent prior Production Releases that are still in service at one or more Customer Sites and may need to be patched (Hot Fix Support) for a period of time under a Support Contract.

vM.m.r -N
(Retired)

This represents prior Production Releases that are no longer in service, they are end-of-life, or 'retired', and are for reference only.

MicroCODE Git Workflow

We use this branching model that maps directly to our Circles:

```
main                      (Production)
   ├──── hotfix/*         (Emergency patches in main)
   ├──── develop          (Alpha - Integration branch)
   ├──── feature/*        (New features from main, accumulate in develop)
   │      └──── bugfix/*  (Defect corrections in alpha test)
   └──── release/*        (Beta - Forked from main, cherry-picks selected features)
          └──── bugfix/*  (Defect corrections in beta test)
```

Branch Types

`main` Branch

Circle: Production
Purpose: Always contains production-ready code
Protection: Protected, requires PR approval
Tagging: Every release tagged with vR.F.P (e.g. **v2.11.322**)

`develop` Branch

Circle: Alpha
Purpose: Integration branch for next release
Source: All feature branches merge here
Status: Pre-production, actively developed

`feature/*` Branches

Circle: Alpha
Purpose: New functionality or improvements
Naming: **feature/this-feature-name**
Source: Created from **develop**
Merge: Merged to **develop** when complete

`release/*` Branches

Circle: Beta
Purpose: Long-lived branches for multi-group testing
Naming: **release/b1.0.0** (semantic versioning)
Lifespan: Remain open during extended testing periods
Merge: Merged to **main** when testing complete, then merged back to **develop**

`bugfix/*` Branches

Circle: Release
Purpose: Defect Corrections for release issues
Naming: **bugfix/v1.0.0/connection-timeout**
Source: Created from 'main'
Merge: Merged to **release** immediately, then merged to **develop** (and any open **release/*** branches)

`hotfix/*` Branches

Circle: Production → Alpha
Purpose: Emergency patches for production issues
Naming: **hotfix/v2.9.5/connection-retry**
Source: Created from main
Merge: Merged to **main** immediately, then merged to **develop** (and any open **release/*** branches)

Version Numbering (Semantic Versioning)

We use Semantic Versioning (SemVer): `vM.F.H`

`M` = **Major Release** - incremented on architecture, design, APIs or other major App changes

`F` = **Feature Release** - incremental release of Feature(s) into a Major Release
(`.1.` = 1st release of new feature(s), `.2.` = 2nd, `.3.` = 3rd, etc.)

`H` = **Hotfix Patch** - incremental Hot Fixes released to correct a particular `vM.F` release (each increment is a single hotfix addition)

Examples:

- `v2.1.0 - Major V2, 1st Feature Release, no hotfixes`
- `a2.1.1 - Major V2, 1st Feature Release, 1st hotfix - ALPHA Build`
- `b2.1.1 - Major V2, 1st Feature Release, 1st hotfix - BETA Build`
- `v2.1.1 - Major V2, 1st Feature Release, 1st hotfix`
- `v2.2.0 - Major V2, 2nd Feature Release (new features bundled), no hotfixes`
- `v3.0.0 - Major V3 (architectural change), 1st Feature Release`

Workflow Examples

Major Version
`feature/v3.0.0/new-ssr-engine → develop → release/b3.0.0 → main`
Tag: `v3.0.0`

Feature Release
`feature/new-redis-cache → develop → release/b2.2.0 → main`
Tag: `v2.2.0`

Hotfix
`hotfix/v2.1.1/connection-retry → main (tag v2.1.1) → develop → release/b2.2.0`
Tag: `v2.1.1`

Issue Tracking with GitHub Issues/Projects

We use GitHub Issues and GitHub Projects to track all software problems, enhancements, and requests. This replaces our previous SPR/SER tracking system.

GitHub Issues

Every problem, request, or enhancement is tracked as a GitHub Issue:

- **Bug Reports**: Issues labeled with bug and severity labels
- **Enhancements**: Issues labeled with enhancement
- **Hotfixes**: Issues labeled with hotfix and linked to hotfix/* branches
- **Features**: Issues labeled with feature and linked to feature/* branches

Issue Labels

We use a standardized label system:

Type Labels:

- **bug** - Software Problem Report {SPR}
- **enhancement** - New feature or improvement request {SER}
- **hotfix** - Emergency production fix
- **feature** - Normal feature release
- **documentation** - Documentation updates

Severity Labels:

- **severity-1** - System crash, complete loss of major component
- **severity-2** - Function fails, no workaround
- **severity-3** - Function fails, workaround available
- **severity-4** - Function fails, no production impact
- **severity-5** - Display/report/tool problem
- **severity-6** - Annoyance, requires business case

Status Labels:

- **alpha** - Work in Alpha Circle
- **beta** - Work in Beta Circle
- **production** - Work in Production Circle
- **blocked** - Waiting on dependency
- **ready-for-review** - Ready for PR review

GitHub Projects

We use GitHub Projects (board view) to visualize workflow:

Project Board Columns:

1. **Backlog** - New issues, not yet prioritized
2. **Alpha** (In Progress) - Active development in: **develop**
3. **Beta** (Testing) - Issues in **release/*** branches
4. **Production** (Deployed) - Issues merged to: **main**
5. **Done** - Completed and verified

Benefits:

- Automatic linking between Issues, PRs, and commits
- Filter by labels, assignees, milestones
- Multiple views (board, table, roadmap)
- All tracking in one place (GitHub)

Linking Issues to Code

When creating branches and PRs, link to Issues:

Branch Naming with Issue Numbers:

```
feature/add-sidebar-builder (#45)
hotfix/v2.9.4/connection-retry (#67)
```

PR Descriptions:

```
Fixes #67
Implements #45
```

Related to release `v2.12.0`

Testing Strategy

This is how we see testing phases in a healthy app lifecycle…

Unit Testing

Location: Alpha Circle (`develop` branch)
Responsibility: Developers
Scope: Application code level, standalone programs

Integration Testing

Location: Alpha Circle (`develop` branch)
Responsibility: Development teams
Scope: System component interactions, functional requirements

System Acceptance Testing (SAT)

Location: Beta Circle (`release/*` branches)
Responsibility: Development and support teams
Scope: Complete integrated software product validation

User Acceptance Testing (UAT)

Location: Beta Circle (`release/*` branches)
Responsibility: Support and customer teams
Scope: Business requirements compliance, delivery acceptance

Git Branching – MicroCODE Naming Conventions

In order to use Git and GitHub safety and effectively everyone on the development and support teams needs to use the same Git naming convention.

Branch Type	MDS	Activity	Git Branch Name
branch	**Retired** *	Previous Release **-N**	`retired/vM.F.H`
branch	**Support** + Hot Fixes	Previous Release **-2**	`archive/vM.F.H`
branch		Customer Hot Fix	`hotfix/nnnn--defect-short-name`
branch	**Support** + Hot Fixes	Previous Release **-1**	`archive/vM.F.H`
branch		Customer Hot Fix	`hotfix/nnnn--defect-short-name`
trunk	**Production (main)** + Hot Fixes	Customer Release	`main/vM.F.H`
branch		Customer Hot Fix	`hotfix/nnnn--defect-short-name`
branch	**Beta (release)** + Bug Fixes	Pilot Site Beta Release	`release/bM.F.0`
branch		Customer Bug Fix	`bugfix/nnnn--defect-short-name`
trunk	**Alpha (develop)** + New Features	Internal Alpha Release	`develop`
branch		Internal New Feature	`feature/nnnn--feature-short-name`

* No Hot Fix support, these Sites are actively upgraded to a 'Production' or 'Support' Release. The 'main minus 1', 'main minus 2' is only for Sites running private copies of an App on their own Intranet Servers vs. our public Internet Servers.

Application Version Numbers

The application software version numbering is documented as follows…

`vM.F.H`

`v` = Development Circle of this Build.

> `a` = Alpha or Development Circle – **develop** branch
> `b` = Beta or Release Circle – **release** branch
> `v` = Production Circle – **main** branch
>
> Changing this label is associated with a '**Code Circle Promotion**', i.e.: Internal Build Promotion. This is a **merge** activity only <u>no code is changed,</u> e.g.: `b2.0.17` → `v2.0.17`.

`M` = **Major Version**; represents application architecture, underlying technology, etc.
Incrementing this number is associated with a '**Major Release**'.
Incrementing this also resets **F** and **H** to `0`, e.g.: `v3.0.0`

`F` = **Feature Release**; represents a collection of new features within a Major Version.
Incrementing this number is associated with a '**New Feature Release**'.
Incrementing this also resets **H** to `0`, e.g.: `v2.2.0`

`H` = **Hotfix Patch Number**; represents a collection of hot fixes within a Feature Release.
Incrementing this number is associated with a '**Hot Fix Patch**'.
A higher Patch number includes all hot fixes in the Feature Release with lower numbers, it cumulative.

Displayed and Documented Version Number Examples:

AppName v1.5.124 – Production Version found in Customer Sites.

AppName v1.5.125 – A Hotfix for v1.5.124.

AppName b1.0.6 – Beta Version found only in Pilot Sites and Support Staff Lab.

AppName a2.0.0 – Alpha Version found only on Working Machines or in the Support Staff Lab.

Defect Resolution Process: 'Hot Fix Patch'

Defect resolution releases are defined as those required to correct Severity 1, 2, or 3 defects *or* the Production Site can present a Business Case showing a significant loss of time, money or resources due to a Severity 4, 5, or 6 defect. **Example:** Version **v1.1.22** of an application is in Production, a defect arises that is deemed Severity **2** and must be fixed within hours or days at the latest.

Step 1: The Support Staff stage a copy of the Production Site in a lab—or visit the customer site—to reproduce the problem. Staging a copy of the Production Site requires all affected components be set to the same application versions as the Production Site, the site's data be loaded (possibly), and then a simulation run to recreate their issue.

Step 2: Once the issue is re-created the Developers are called in to recreate the issues in debug mode to isolate the code issue(s), engineer a fix, test the fix, document the resolution and create a new **Hotfix**, in this example `[v1.1.23]`.

GIT: This is when the **hotfix/** branch is created.

Step 3: This Build would be turned over to the Support Staff for regression testing of the affected components *which would have been identified in the Developer's defect resolution documentation.* If regression testing of the affected functionality fails the Support/Developer interaction repeats until the user's original defect and all affected components pass regression test.

Step 4: At that point the updated application components are delivered into production with required documentation included: **Release Notes**, **Installation/Upgrade Procedure**, etc.

GIT: This is when the **main** branch is updated.

Final Step: The Defect Resolution is carried into all inner Code Circles, i.e.: into the **develop** and/or other **release/** branches in development stages in the Lab.

GIT: This is the **merge** activity that occurs when an **hotfix/** branch is closed.

Feature Release: 'New Feature Release'

A feature release is defined by the addition of a new function or component in an existing version of the application but no changes to the overall application structure. i.e.: Support for a new I/O Device.

Step 1: A user of the existing version of the application, already in use in production, is requesting support for the new component, the current version of **main** is **v1.0.124**. An **Alpha** branch is created for the Devs, e.g.: `[a1.1.0]`.

GIT: This is when the **feature/nnn--name-of-feature** branch is created – the 'Alpha' builds.

Step 2: Support for the new component is added, tested, documented and regression testing of any affected components is performed. A **Beta** branch is created for the Pilot site, e.g.: `[b1.1.0]`.

GIT: This is when the **feature/** branch is closed.
GIT: This is when the **release/b1.1.0** branch is created – the 'Beta' builds.

Step 3: The **Beta** Release is piloted at the requesting site. Defects are handled via the standard **Defect Resolution Process** with one exception, these fixes are called **bugfix/** not hotfix/. After a defined time period with no Sev. 1, 2, or 3 defects the release promoted to 'Production' status and published for any sites to acquire and deploy, e.g.: `[b1.1.0]` → `[v1.1.0]`.

GIT: This is when the **main** branch is updated.

Final Step: The new Feature is carried into all inner Code Circles, i.e.: **develop** and other open **releases**.

Major Release: Major Release' Process

A major release implies changes to the overall application structure, like movement to a different Hardware or Software platform, e.g.: Support for Windows 11 Server.

Step 1: The newest existing version of the application, already in use in production(s) is selected for the development effort, e.g.: `[v1.0.61]`. An **Alpha** Build is then created for the Devs, e.g.: `[a2.0.0]`.

GIT: This is when the **feature/a2.0.0** branch is created – the 'Alpha' builds.

Step 2: The application is migrated to the new platform, tested, documented and regression testing of **all functionalities** is performed.

Step 3: The **Alpha** Release is passed through System Acceptance Testing (SAT) until it passes all required Software Test Cases (STC). Defects are handled via the standard **Defect Resolution Process** with one exception, these fixes are called **bugfix/** not hotfix/. After all the STCs are passed with no Sev. 1, 2, or 3 defects then the release is promoted to 'Beta' status, and a **Beta** Build is created for the Pilot site, e.g.: `[b2.0.0]`.

GIT: This is when the **release/b2.0.0** branch is created – the 'Beta' builds.

Step 4: The **Beta** Release is staged in the Tester's environment. Defects are handled via the standard **Defect Resolution Process** with one exception, these fixes are called **bugfix/** not hotfix/. After a defined time period with no Sev. 1, 2, or 3 defects the release is promoted to **main** production status and published for all sites to use, e.g.: `[v2.0.0]`.

GIT: This is when the **main** branch is updated.

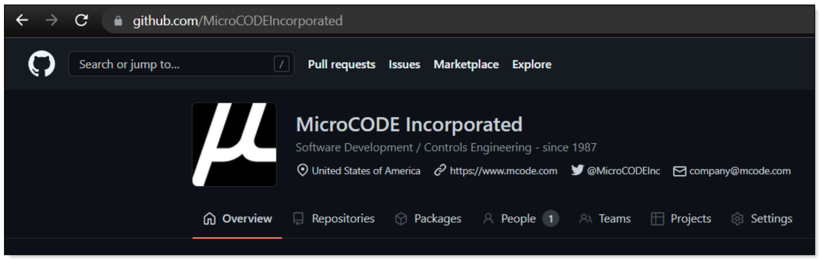
Final Step: There are no inner Code Circles during a 'Major Release' cycle.

After 30 years of software development and customer support we believe this is the only way to properly support Production Critical Software.

Appendix A: Use of Git

GitHub is the Cloud repository for all Builds.

Git is the tool used on a Working Machine (Laptop, Desktop, etc.).


github.com/MicroCODEIncorporated

MicroCODE Incorporated

Software Development / Controls Engineering - since 1987

United States of America · https://www.mcode.com · @MicroCODEInc · company@mcode.com

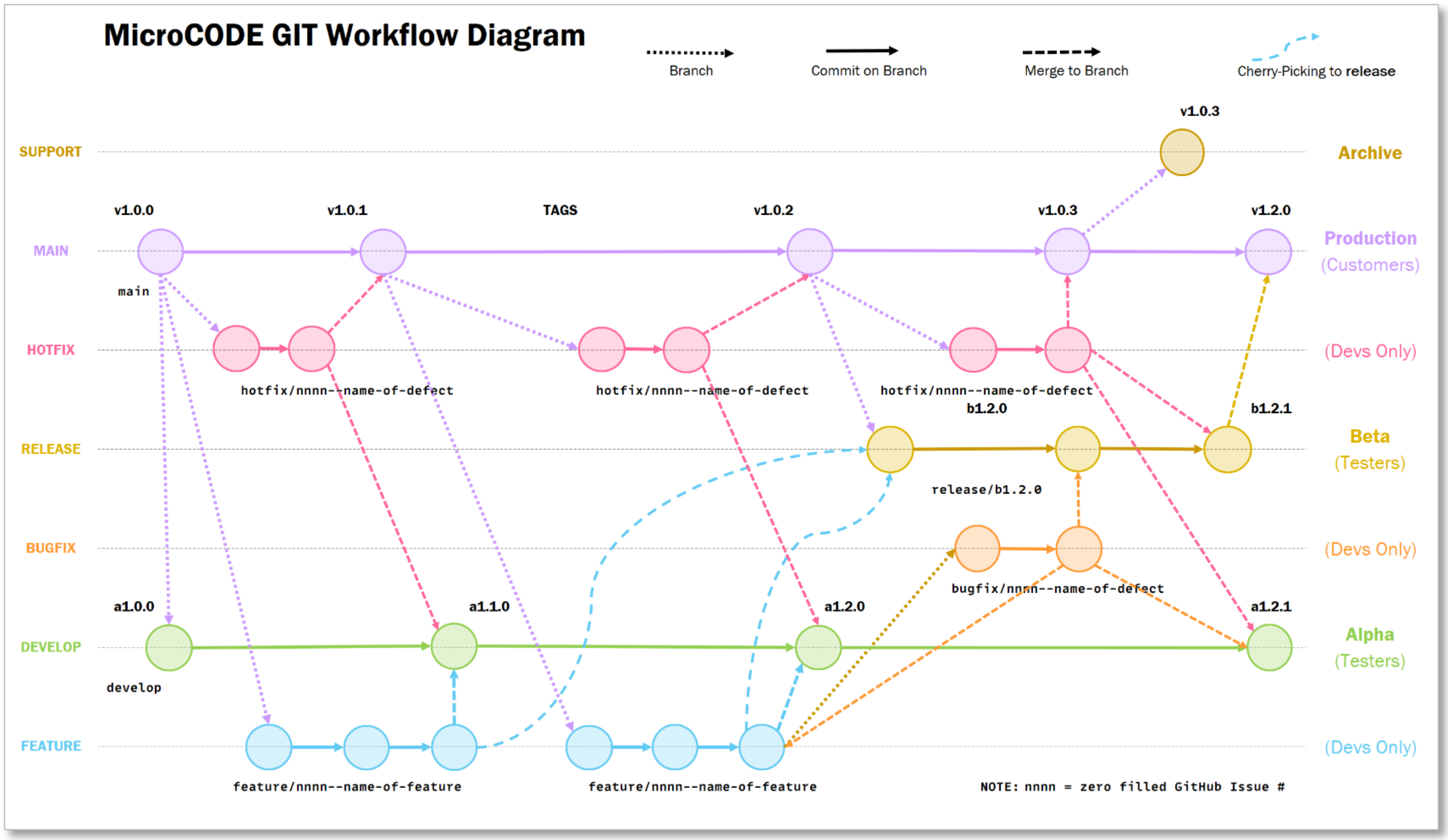
...this is where all the approved 'Trunks' and 'Branches' are held for all company developers.

Each Software Product will have a minimum of three (3) branches:

- **main** = the current production code
- **release** = the next production release while in testing
- **develop** = development work the future release(s)

See "**Git Branching – MicroCODE MDS Naming Conventions**" earlier in this document.

NOTE: This process was adapted from the Atlassian Bitbucket Tutorial found here:
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow



Gitflow Workflow

Gitflow is a legacy Git workflow that was originally a disruptive and novel strategy for managing Git branches. We use a modified version of Gitflow we call **MDS** which is designed to support both installed Windows Apps and Web-based Services. (Web-based Services have moved away from Gitflow somewhat and toward Continuous Improvement (CI), Continuous Delivery (CD) models. While our MDS process supports CI/CD it is very mindful of the need to protect a Customer Production Sites with a guarded 'Production' trunk that is only changed with a very rigorous process.

What is Gitflow?

Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. It was first published and made popular by Vincent Driessen at nvie. Compared to trunk-based development, Gitflow has numerous, longer-lived branches and larger commits. Under this model, developers create a feature branch and delay merging it to the main trunk branch until the feature is complete. These long-lived feature branches require more collaboration to merge and have a higher risk of deviating from the trunk branch. They can also introduce conflicting updates.

Gitflow can be used for projects that have a scheduled release cycle and for the DevOps best practice of continuous delivery. 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. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

Gitflow

Gitflow is really just an abstract idea of a Git workflow. This means it dictates what kind of branches to set up and how to merge them together. We will touch on the purposes of the branches below. The git-flow toolset is an actual command line tool that has an installation process. The installation process for git-flow is straightforward. Packages for git-flow are available on multiple operating systems. On OSX systems, you can execute brew install git-flow. On windows you will need to download and install git-flow. After installing git-flow you can use it in your project by executing:

```
git flow init
```

Git-flow is a wrapper around Git. The git flow init command is an extension of the default git init command and doesn't change anything in your repository other than creating branches for you.

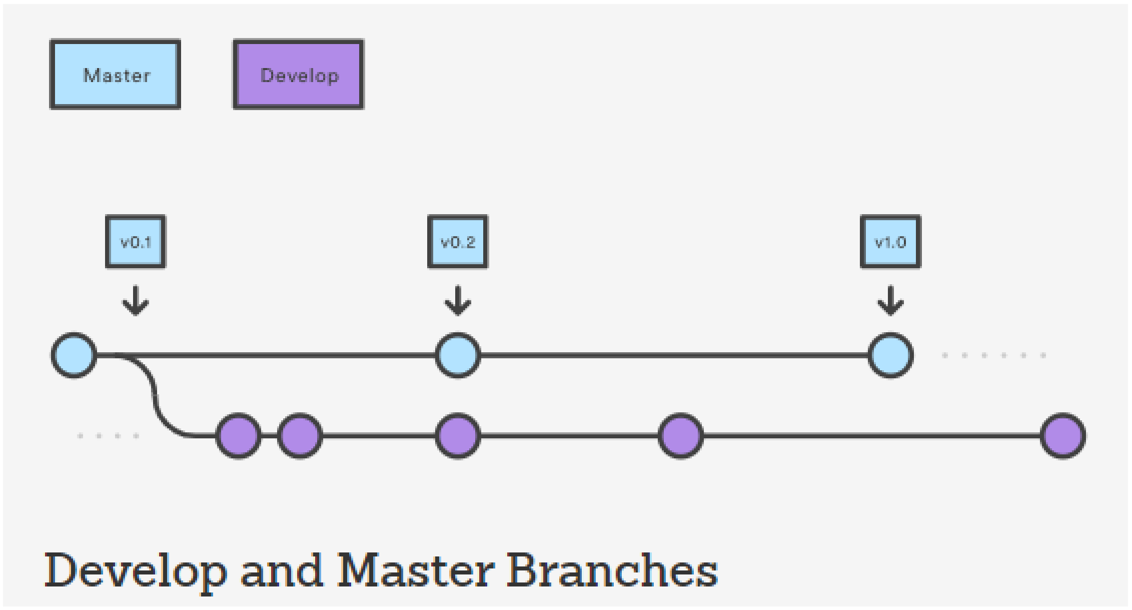
NOTE: Because Git Flow does not support '**Cherry Picking**' features into a release we cannot use it, so all our directions involve 'pure Git', but our process is **very close** to Git Flow, our only difference is creating **release/*** from **main** and 'cherry picking the **feature/*** branches into it from a list agreed upon by the Users, Devs, and Testers.

1 Development (develop) and Master (main) Branches

Instead of a single Production ('main) branch, Git Flow uses two branches to record the history of the project. It is based on two main branches (trunks) with infinite lifetime namely Production ('main) and Development ('develop').

- **main** Trunk: The Production branch contains the production code and stores the official release history.
- **develop** Trunk: The Development branch(s) contains pre-production code and serves as an integration branch for New Features.

Production and Development branch workflow is demonstrated in the given diagram:



It's also required to tag all commits in the Production branch with a version number. These represent Releases that can run in a customer's production facility.

First create a "main" branch. Then complement the default Production with a Development ("Alpha") branch named "develop". A simple way to do this is for one developer to create an empty Development ("Alpha") branch locally and push it to the server. This branch will contain the complete history of the project. Other developers should now clone the central repository and create a tracking branch for development.

GIT: The "Production" trunk, should be named:

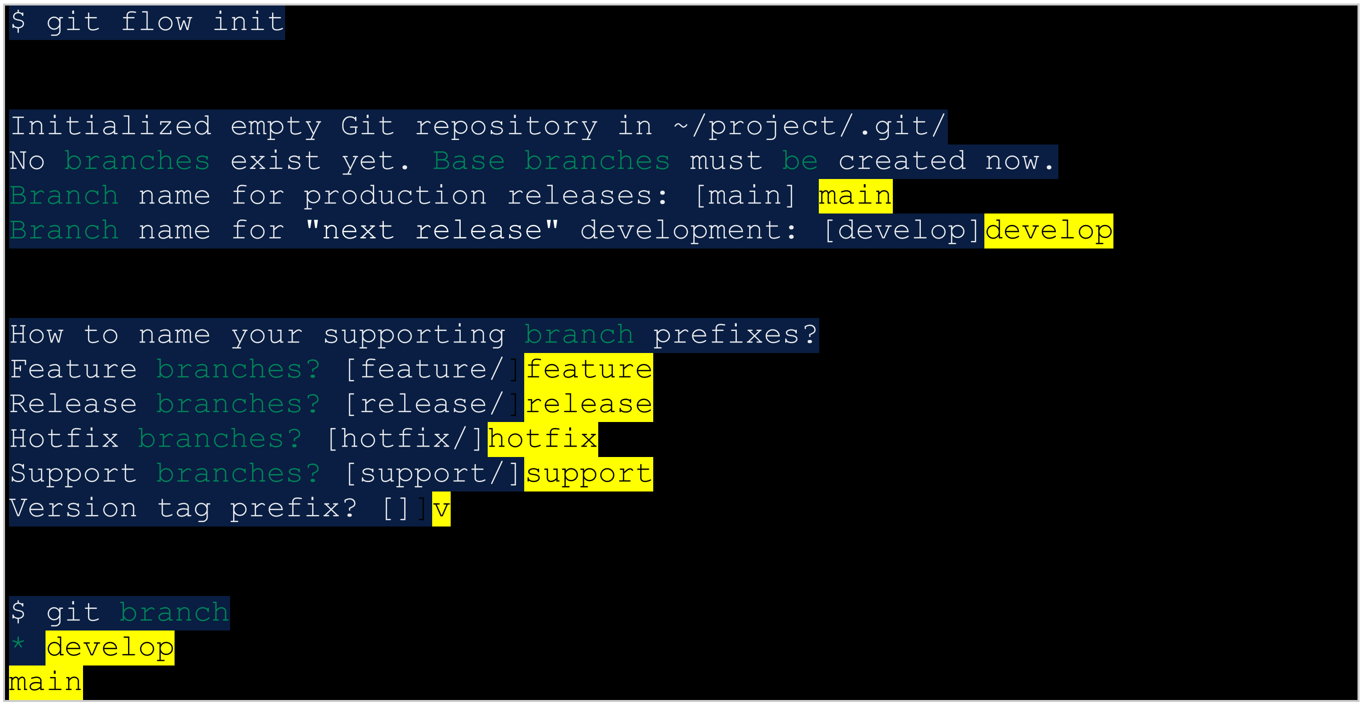
- `main`

1.1 Creating a Development Branch

- **Without the git-flow extensions:**
  - `git branch` **develop***
  - `git push -u origin` **develop***

- **When using the git-flow extensions:**
  - `git flow init`

* When using the git-flow extension library, executing “git flow init” on an existing repository will create the Development (Alpha) branch.



```
$ git flow init


Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [main] main
Branch name for "next release" development: [develop]develop


How to name your supporting branch prefixes?
Feature branches? [feature/]feature
Release branches? [release/]release
Hotfix branches? [hotfix/]hotfix
Support branches? [support/]support
Version tag prefix? []]v


$ git branch
* develop
main
```

2 Feature (Alpha) Branch

Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. Feature branches use the latest Development as their parent branch. When a feature is complete, it gets merged back into Development. Features should never interact directly with the Production (**main**) branch.

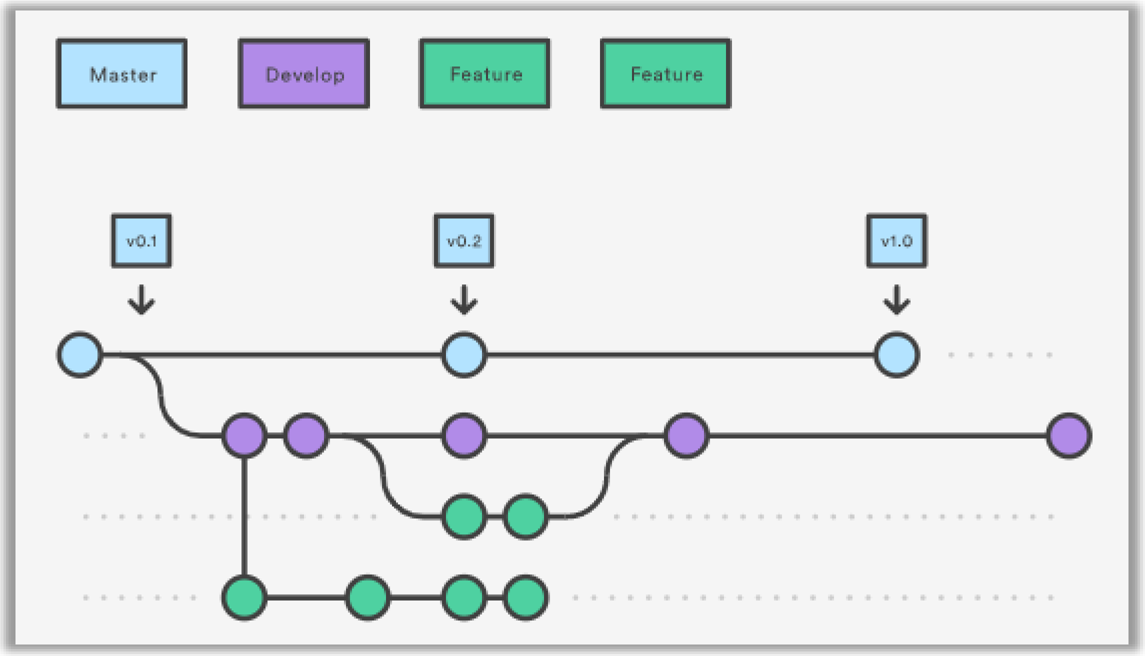
- **develop** Trunk: This Development branch contains pre-production code and serves as an integration branch for New Features.
- **feature/** Branch: This Development branch contains work on a **single new feature**.

GitHub Issues keeps track of every customer problem and request. Every new feature request is recorded in an Issue number (#nnnn).

These can only be resolved/closed in three (3) ways:

1) The Request is implemented as a New Feature and Released.
2) The Request is identified as a Duplicate and linked the first Issue.
3) Implementing the New Feature is found to be unproductive, destructive, or in conflict with some core design element of the product, and the correct action is documented and included in the Release Notes.

Feature branch workflow is demonstrated in the given diagram:



2.1. Creating a Feature Branch

Features are to be designed as independent changes as much as possible, i.e.: able to ship as soon as completed. Because of this they are started from **main** expecting they will be released into this production code as soon as possible, usually grouped together with other features as a **release**.

- **Without git-flow extensions:**
  - `git checkout main`
  - `git checkout -b feature/nnnn--name-of-feature`
- **With git-flow extensions:**
  - `git flow feature start nnnn--name-of-feature`*

2.2. Finishing a Feature Branch

When a **feature** is complete it cannot be committed to **main** until it is completed tested and approved. So it is merged into **develop**. In **develop**, it is tested with other features and awaits selection into a **release**.

- **Without git-flow extensions:**
  - `git checkout develop`
  - `git merge feature/nnnn--name-of-feature`
- **With git-flow extensions:**
  - `git flow feature finish nnnn--name-of-feature`*

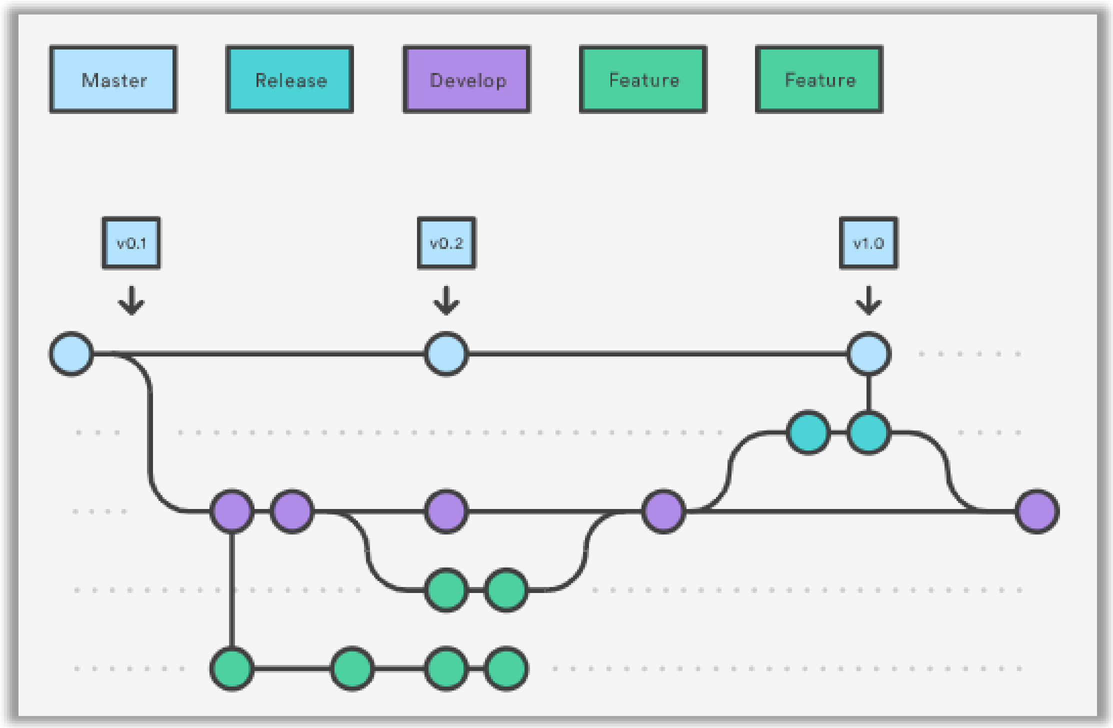
* We can't use Git Flow here because it is 'hard-coded' to start **feature/*** branches from **develop** instead of **main**.

3 Release (Beta) Branch

Once Development has acquired enough features for a release (or a predetermined release date is approaching), we fork a Release branch off of **main** (proven safe code) and cherry-pick directly from **feature/*** branches, NOT from **develop**. This strategy completely decouples **release/*** from **develop**, preventing 'release paralysis' and giving you full control over what ships. Creating this branch starts the next release cycle, so after adding the initial new features set nothing else is added—only bug fixes, documentation generation, and other release-oriented tasks. A **release/*** branch is forked from **main** (with cherry-picks directly from **feature/*** branches) and must merge into both **main** when competed.

- **develop** Trunk: This Development branch contains pre-production code and serves as an integration branch for New Features.
- **release/** Branch: This Testing branch contains a pilot-ready 'Production Candidate' that has a collection of New Features and all Bug Fixes that currently reside in the 'Production' trunk.

Release branch workflow is demonstrated in the given diagram:



Once the Release branch is ready to ship, it gets merged into Production (**main**) and tagged with a version number. Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release.

Release branches are forked from **main** (proven safe code) and cherry-pick directly from completed **feature/*** branches. This decouples releases from **develop**, preventing 'release paralysis' and giving full control over what ships. Features are selected based on customer needs, not what happens to be in **develop**.

3.1. Creating a Release (Beta) Branch

- **Without git-flow extensions:**
    - `git checkout main`
    - `git checkout -b release/b1.1.0`
    - `git cherry-pick feature/0031--new-widget`
    - `git cherry-pick feature/0045--add-sidebar-builder`
    - `git cherry-pick feature/0046--add-toast-alerts`

- **With git-flow extensions:**
    - `git flow release start b1.1.0`*

3.2. Finishing a Release (Beta) Branch

- **Without git-flow extensions:**
    - `git checkout main`
    - `git merge release/b1.1.0`

With git-flow extensions:

- `git flow release finish b1.1.0`*

* We can’t use Git Flow here because it is ‘hard-coded’ to start **release/*** branches from **develop** instead of **main**.

4 Hotfix (Production) Branch

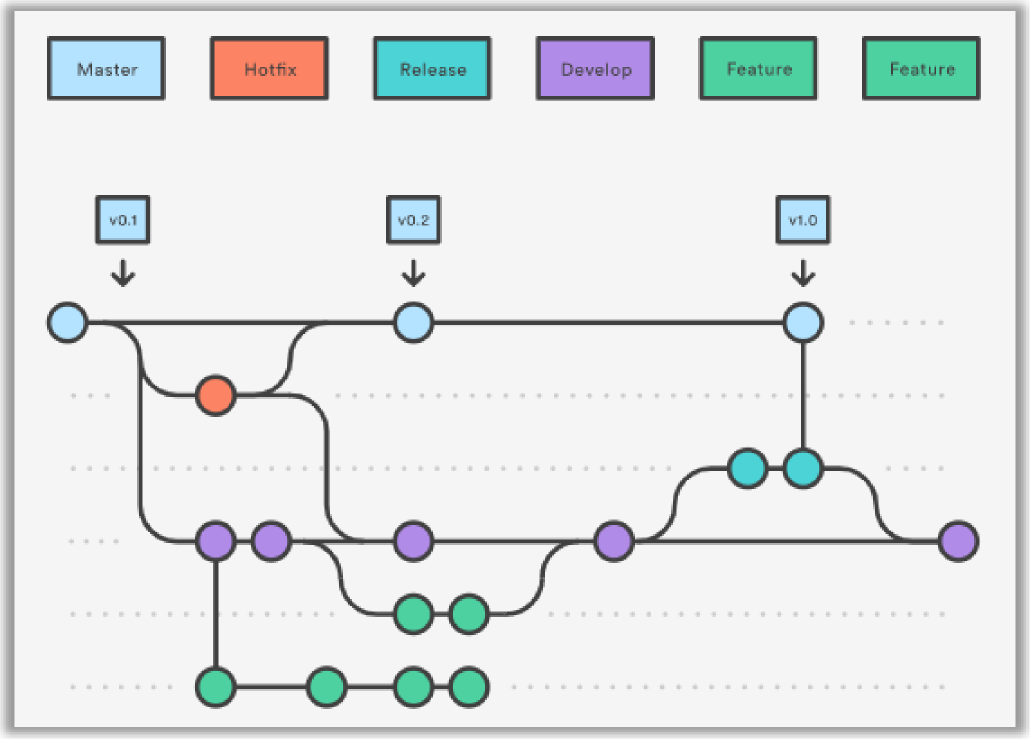
Maintenance or "Hotfix" branches are used to quickly patch Production releases. Hotfix branches are necessary to immediately correct a defect in Production. Hotfix branches are a lot like release branches and feature branches except they're based on Production instead of Development. This is the only branch that should fork directly off of Production. As soon as the fix is complete, it should be merged into both Production and Development (or the current Release branch), and the Production branch should be tagged with an updated version number.

- **main** Trunk: The branch containing production code with the official release history.
- **hotfix/** Branch: This Testing branch contains a production-ready code that has a single defect correction added.

GitHub Issues keeps track of every customer problem and request. Every actual problem is recorded as an Issue (#nnnn). These can only be resolved/closed in three (3) ways:

1) The Defect is corrected by a Hot Fix and Released.
2) The Defect is identified as a Duplicate and linked to the original Issue.
3) The Defect report is found to be erroneous, it is 'Rejected' and the correct action is documented and included in the Release Notes.

Hotfix branch workflow is demonstrated in the given diagram:



Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle.

4.1. Creating a Hotfix (Production) Branch

- **Without git-flow extensions:**
  - `git checkout main`
  - `git checkout -b hotfix/nnnn--name-of-defect`

- **With git-flow extensions:**
  - `git flow hotfix start nnnn--name-of-defect*`

4.2. Finishing a Hotfix (Production) Branch

- **Without git-flow extensions:**
  - `git checkout main`
  - `git merge hotfix/nnnn--name-of-defect`
  - `git tag v2.1.(H+1)   # Production tag`

  - `git checkout develop`
  - `git merge hotfix/nnnn--name-of-defect`
  - `git tag a2.1.(H+1)   # Alpha tag`

  - `git checkout release/bM.F.0`
  - `git merge hotfix/nnnn--name-of-defect`
  - `git tag b2.1.(H+1)   # Beta tag`

With git-flow extensions:

- `git flow hotfix finish nnnn--name-of-defect*`

* Git Flow starts **hotfix/*** branches from **main** and merges to **main** and so could be used for our process.

5 Bugfix (Release) Branch

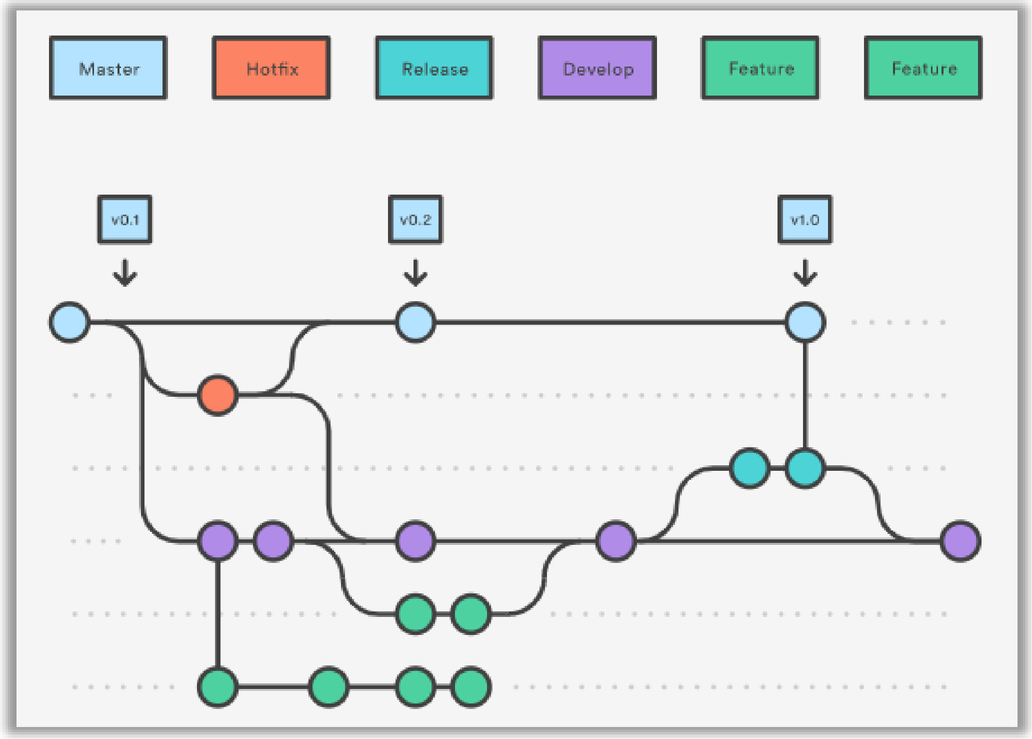
Testing or “Bugfix” branches are used to quickly patch Beta release candidates. Bugfix branches are necessary to immediately correct a defect in Testing. Bugfix branches are a lot like Hotfix branches except they’re based on Beta RCs instead of Production. A Bugfix branch should fork the Feature (**feature/***) branch that represents the original (canonical) source of the defect. As soon as the fix is complete, it should be merged into both Beta (**release/***) and Development (**develop**), and the Production branch should be tagged with an updated version number.

- **feature/*** Branch: The development branch that introduced the defect.
- **bugfix/** Branch: This Testing branch contains a Beta-ready code that has a single defect correction added to the canonical feature source.

GitHub Issues keeps track of every customer problem and request. Every actual problem is recorded as an Issue (#nnnn). These can only be resolved/closed in three (3) ways:

1) The Defect is corrected by a Bug Fix and Released.
2) The Defect is identified as a Duplicate and linked to the original Issue.
3) The Defect report is found to be erroneous, it is ‘Rejected’ and the correct action is documented and included in the Release Notes.

Bugfix branch workflow is demonstrated in the given diagram:



Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle.

5.1. Creating a Bugfix (Release) Branch

- **Without git-flow extensions:**
  - `git checkout feature/nnnn–feature-short-name`
  - `git checkout -b bugfix/nnnn--name-of-defect`

- **With git-flow extensions:**
  - `git flow bugfix start nnnn--name-of-defect*`

5.2. Finishing a Bugfix (Release) Branch

- **Without git-flow extensions:**
  - `git checkout feature/nnnn–feature-short-name`
  - `git merge bugfix/nnnn--name-of-defect`

  - `git checkout release/bM.F.0`
  - `git merge bugfix/nnnn--name-of-defect`
  - `git tag b2.1.(H+1)  # Beta tag`

  - `git checkout develop`
  - `git merge bugfix/nnnn--name-of-defect`
  - `git tag a2.1.(H+1)  # Alpha tag`

- **With git-flow extensions:**
  - `git flow bugfix finish nnnn--name-of-defect*`

* Git Flow doesn't have a standard **bugfix** branch type. Our bugfix branches fork from **feature/*** (canonical source) to ensure fixes persist even if **release** branches are abandoned, which is not supported by standard Git Flow.

6 Deleting Branches

The only reason to delete Git Branches is to clean up the Branch List, that's it. But if you delete a `feature/*` branch after merging to develop you lose the ability to easily build a `release/*` from `main` + a selection of `features/*`. Not worth it to just clean up a list.

`feature/*`	→ **Never** delete, keep for late 'cherry-picking' into `release/*` and for easy history
`bugfix/*`	→ Delete after merge to `release/*`
`hotfix/*`	→ Delete after merging to `main`, `develop`, and `release/*`
`release/*`	→ **Never** delete, keep even after merging to `main` and `develop` and **tagging** `[vM.F.0]`

Advantages of Git Flow

Now let's summarize the major advantages provided by Git Flow:

- Ensures a clean state of branches at any given moment in the life cycle of a project
- The naming convention of branches follows a systematic pattern making it easier to comprehend
- Has extensions and support on most used Git tools
- Ideal in case of maintaining multiple versions in production
- Great for a release-based software workflow
- Offers a dedicated channel for hotfixes to production
- For MicroCODE we need to protect Production Facilities at all times, and we must be able to Hot Fix the Production and Support versions of a product at any time.

Disadvantages of Git Flow

Well, nothing is ideal, so Git Flow holds some disadvantages as well like:

- Git history becomes unreadable
- The Production/Development branch split is considered redundant and makes the Continuous Delivery/Integration harder
- Not recommended in case of maintaining a single version in production
- For MicroCODE the history is captured in the naming standards of 'GitHub Issues' and the protection of the 'Production' branch, out weights all other considerations.

Summary

Here we discussed the Git Flow Workflow vs. our MicroCODE Git Workflow. Git Flow is one of the many styles of Git workflows you and your team can utilize but it does not match our workflow, which is:

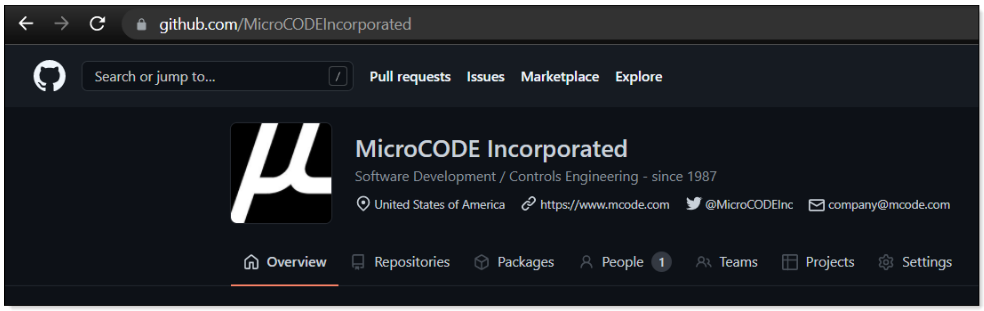
1. A develop branch is created from the main branch
2. All feature branches are created from the main branch
3. When a feature is complete it is merged into the develop branch for Alpha testing. Release branches are forked from main and cherry-pick directly from feature/* branches (not from develop), giving full control over feature selection.
4. A release branch is created from the main branch,
5. When the release branch is done it is merged into main.
6. If an issue is detected in production a hotfix branch is created from main.
7. Once the hotfix is complete it is merged to main and then to develop and any open release branch

Appendix B: Use of GitHub

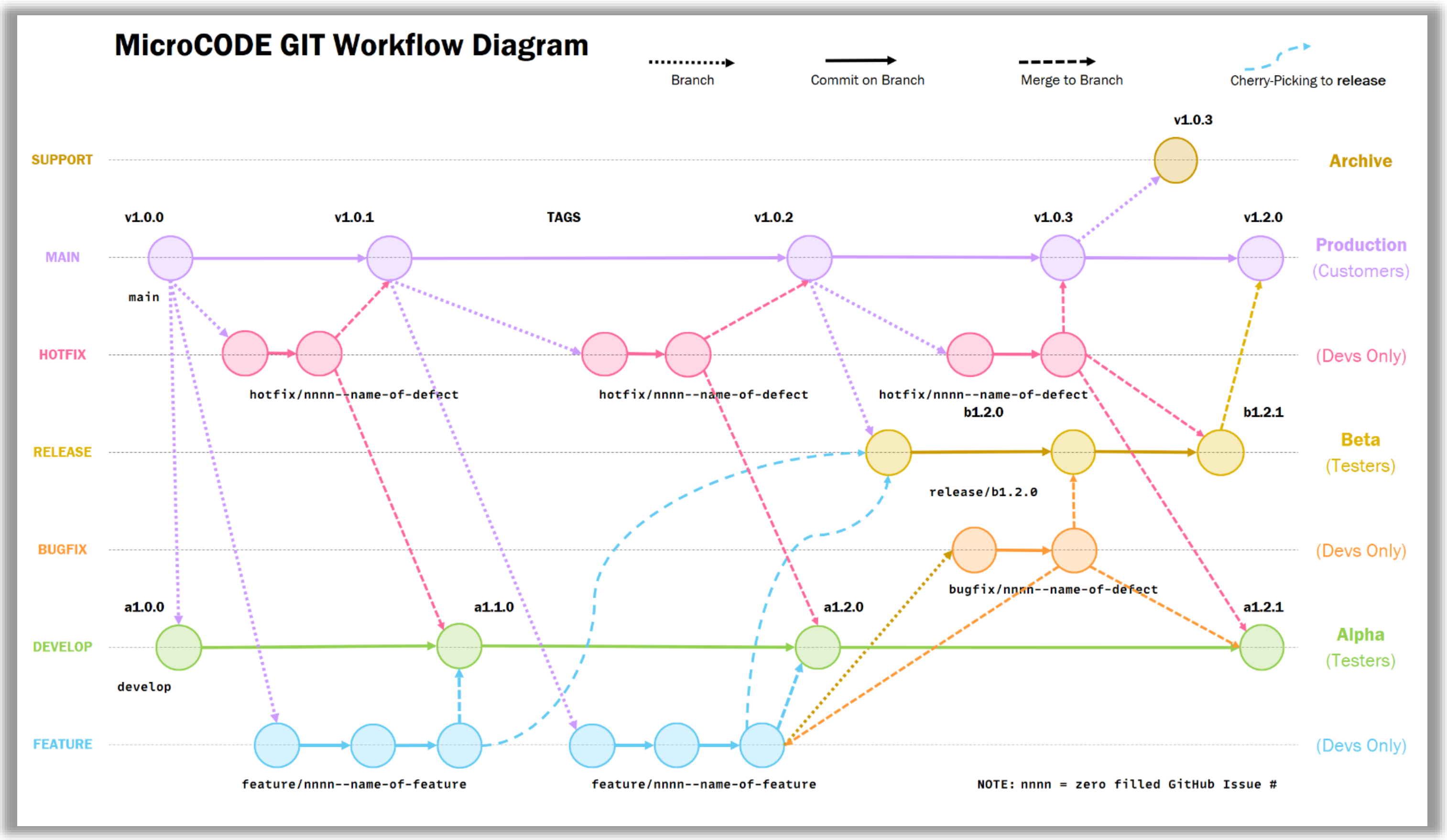
GitHub is the Cloud repository for all Builds.

Git is the tool used on a Working Machine (Laptop, Desktop, etc.).

MicroCODE has a company GitHub…

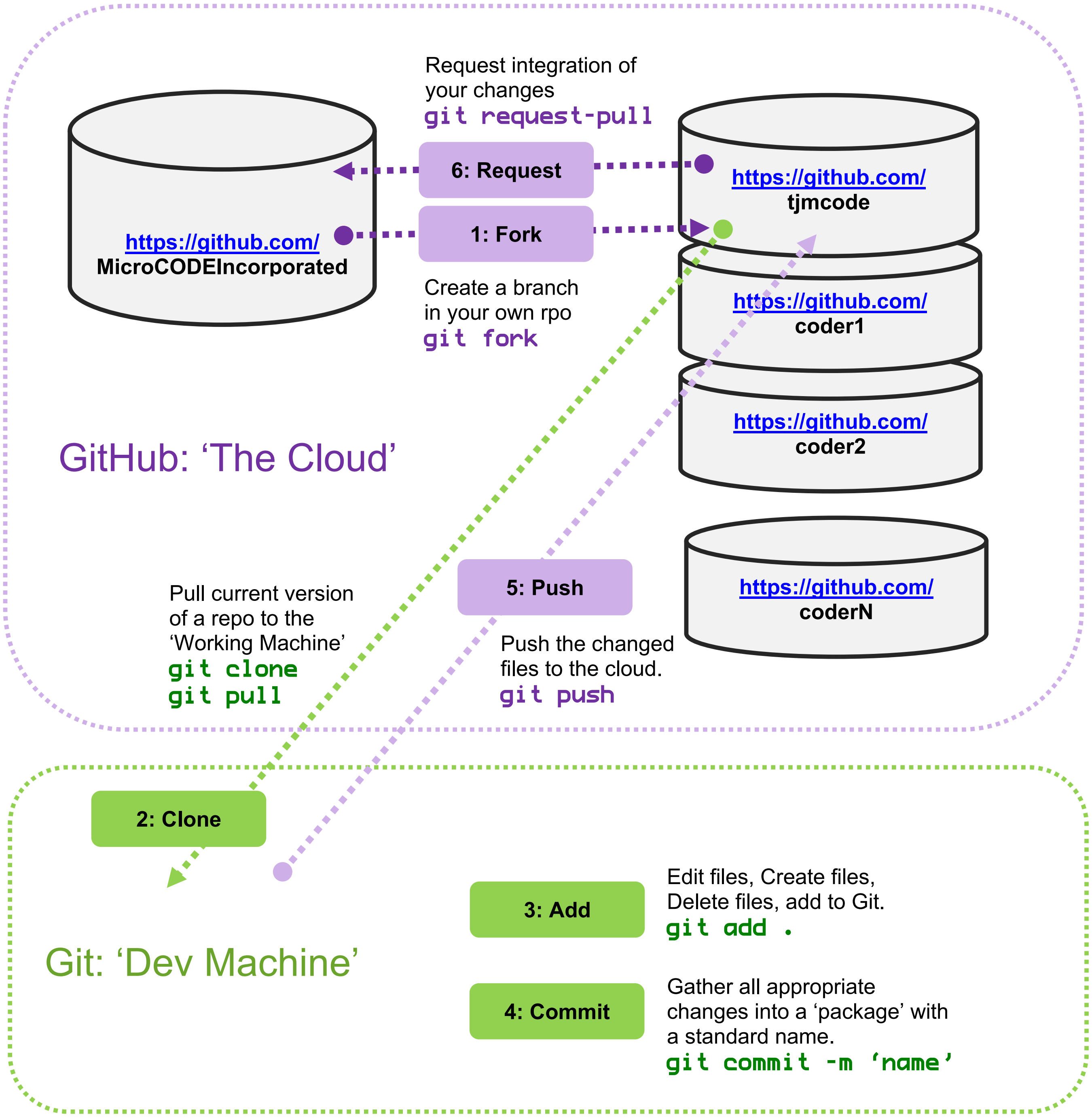


...this is where all the approved 'Trunks' and 'Branches' are held for all company developers.



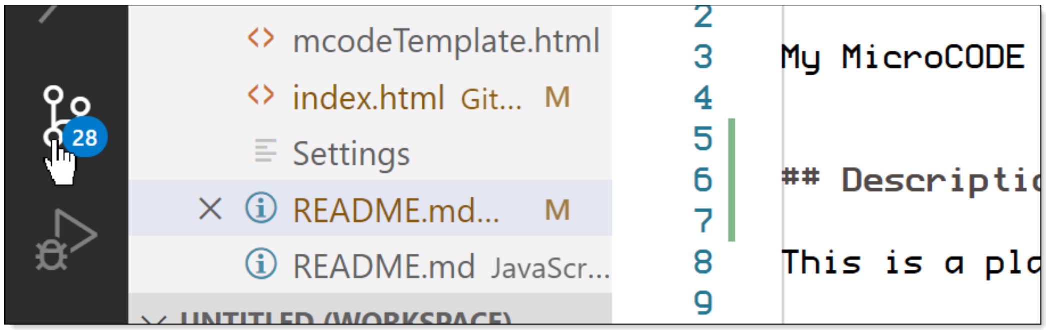
Handling a Hot Fix for a GitHub Issue

1: Fork the `main` into your GitHub Account to `hotfix/nnn--name-of-issue`
2: Clone `hotfix/nnn--name-of-issue` to your Dev Machine
3: Add any files you change on your Dec Machine into your Git
4: Commit your changes with a comment: “`(#nnnn) Resolved/Corrected/Fixed...`“
5: Push sync your changes into your remote repository in GitHub.
6: Pull-Request your changes into `main.`



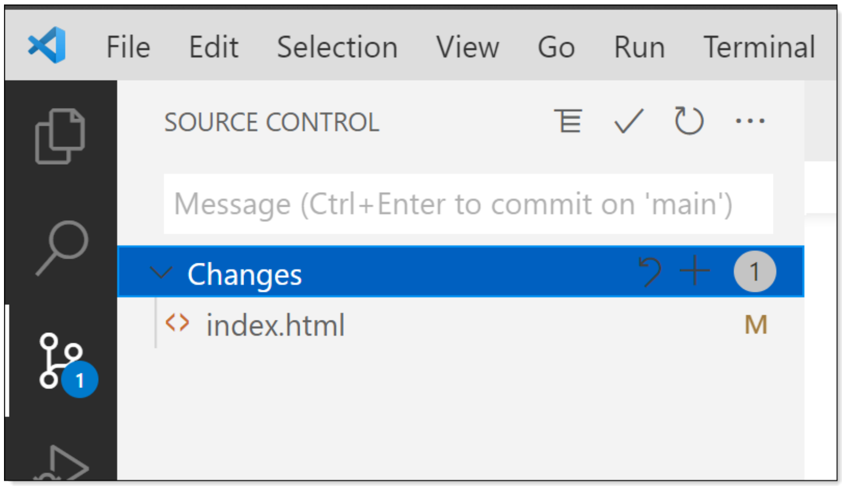
Visual Studio Code – Integration

VS Code has built-in Git / GitHub integration. The number of differences is shown in the GIT icon.

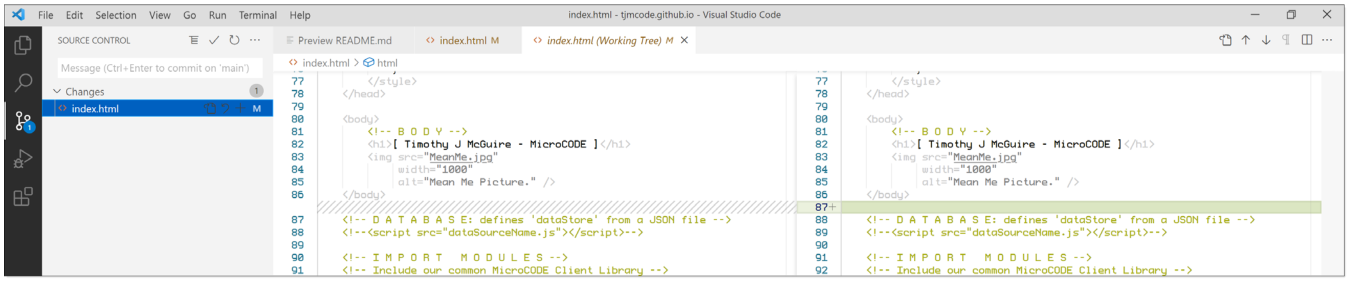


```
My MicroCODE

## Descripti

This is a pl
```

Clicking on the GIT icon opens the 'Commit Review' pane.



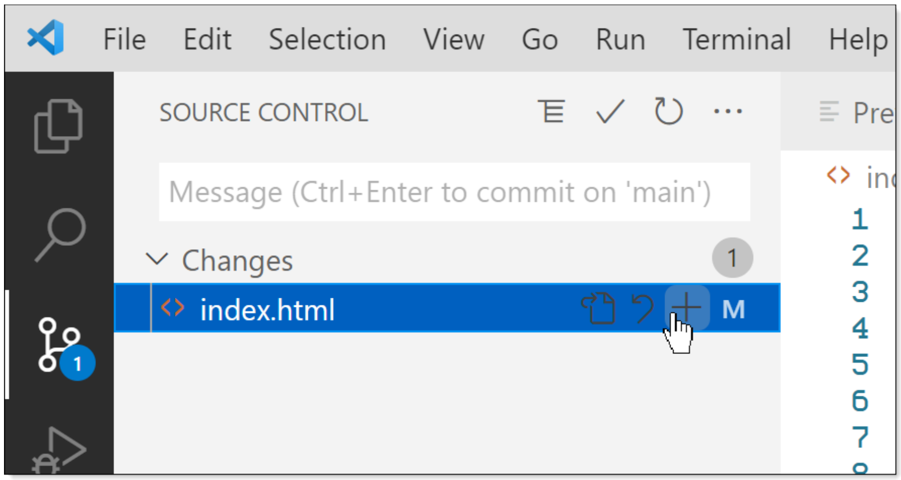
Touching a file immediately shows a DIFF split window…


File
Edit
Selection
View
Go
Run
Terminal
Help
index.html - tjmcode.github.io - Visual Studio Code
SOURCE CONTROL
Message (Ctrl+Enter to commit on 'main')
Changes
1
index.html
M
1
Preview README.md
index.html M
index.html (Working Tree) M
index.html > html

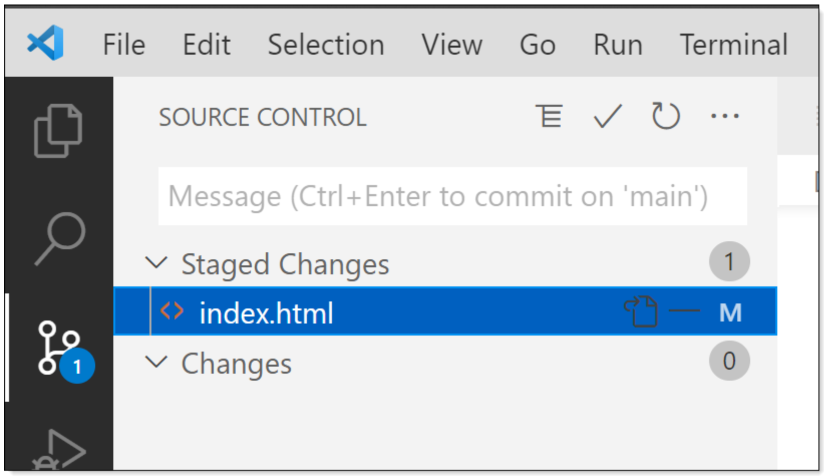
```
        </style>
    </head>

    <body>
        <!-- B O D Y -->
        <h1>[ Timothy J McGuire - MicroCODE ]</h1>
        <img src="MeanMe.jpg"
             width="1000"
             alt="Mean Me Picture." />
    </body>

<!-- D A T A B A S E: defines 'dataStore' from a JSON file -->
<!--<script src="dataSourceName.js"></script>-->

<!-- I M P O R T   M O D U L E S -->
<!-- Include our common MicroCODE Client Library -->
```

```
        </style>
    </head>

    <body>
        <!-- B O D Y -->
        <h1>[ Timothy J McGuire - MicroCODE ]</h1>
        <img src="MeanMe.jpg"
             width="1000"
             alt="Mean Me Picture." />
    </body>
+
<!-- D A T A B A S E: defines 'dataStore' from a JSON file -->
<!--<script src="dataSourceName.js"></script>-->

<!-- I M P O R T   M O D U L E S -->
<!-- Include our common MicroCODE Client Library -->
```

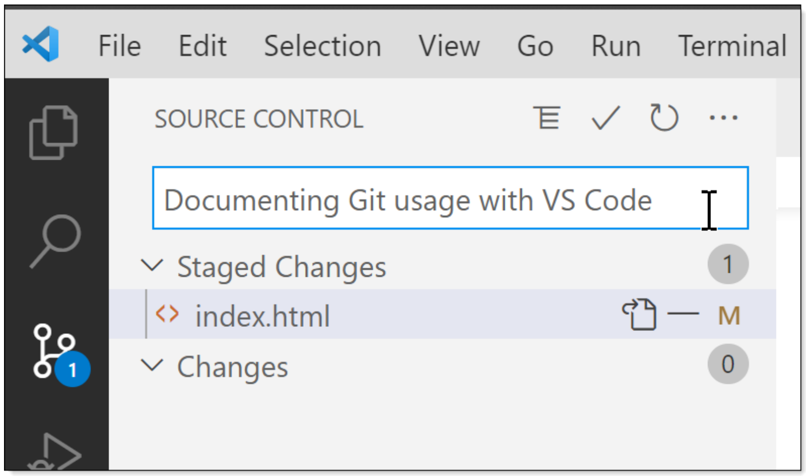
Clicking the "+" performs a "`git add`" of that file to the 'Staged Files' for a 'Commit'…



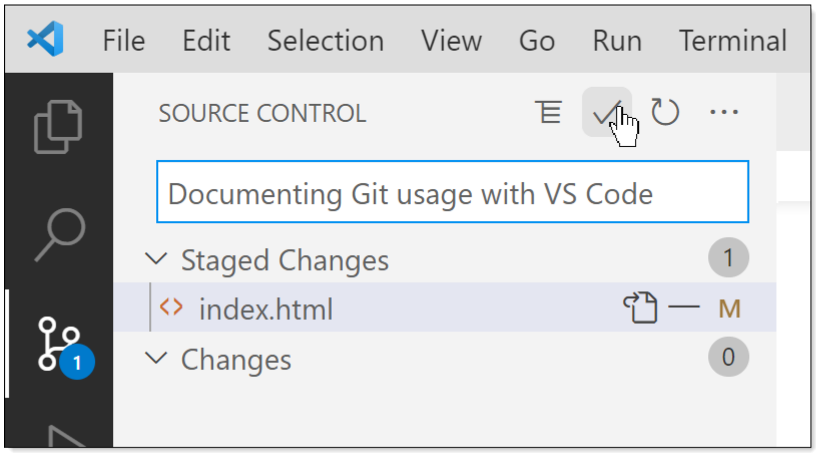
Staged files for a 'Commit'…



Add a description for the 'Commit'…

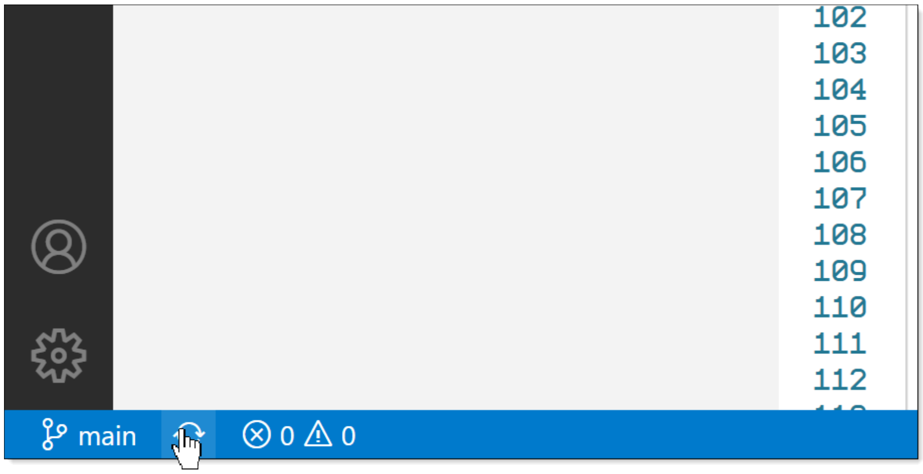


Then use the ‘Check Mark’ / ‘Tick’ to perform the ‘Commit’ to GitHub…

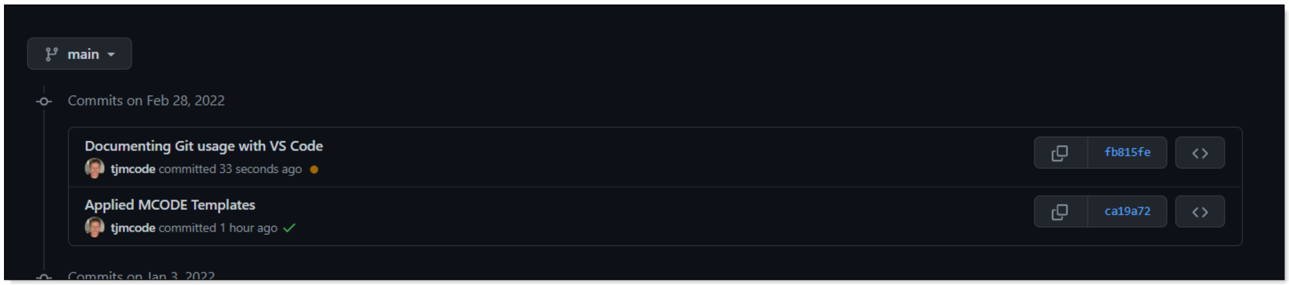


Documenting Git usage with VS Code

Then you click the ‘Sync’ icon or button to ‘Push’ the ‘Committed’ changes to GitHub…



These comments are used in Git / GitHub to document your ‘Commits’…



Commits on Feb 28, 2022

Documenting Git usage with VS Code
tjmcode committed 33 seconds ago
fb815fe

Applied MCODE Templates
tjmcode committed 1 hour ago
ca19a72

Commits on Jan 3, 2022