MicroCODE Consulting Services
Application Development Notes

{2012-05} MicroCODE Software Support Process v008.docx

Subject: Application Development and Maintenance Process
Audience: MicroCODE Customers
Purpose: Explanation of recommended App support process
Author: Tim McGuire
Last Updated: Friday, March 04, 2022

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

Requirements
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:

Internal Development Circle (ALPHA Code)
This Circle represents the '**next major release**' currently being development. The Business has presented requirements, they have been defined, documented, and approved, and the application team is implementing, documenting, and testing a new version of the application which fulfills these requirements. This is 'long term' activity.

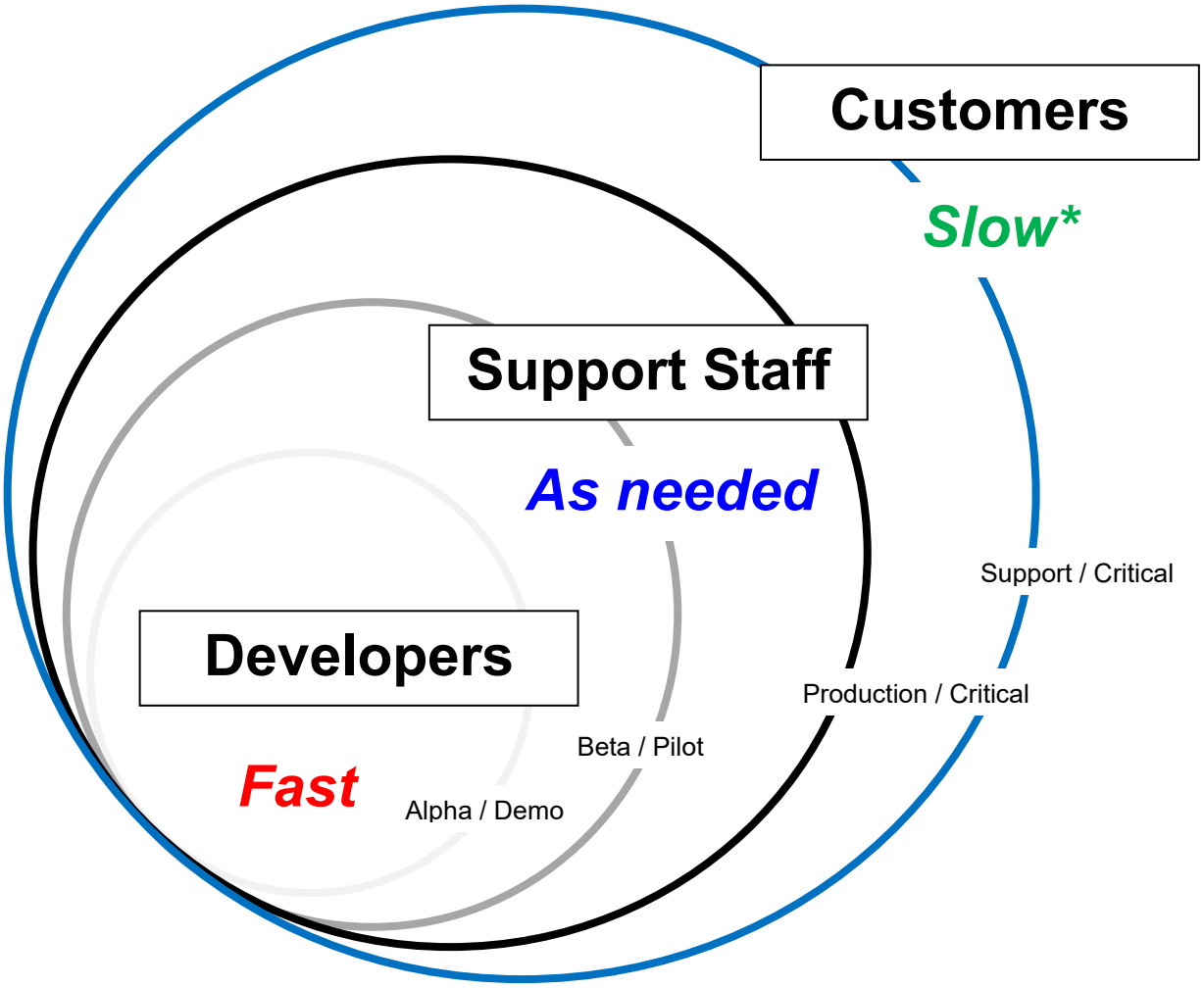
Plant Pilot Circle (BETA Code)
This Circle represents the '**current major release**' being tested and supported in selected customer sites. There could be multiple sites using the beta release in Production. There should only be one release in Beta Circle at any one time. A minimum of three (3) sites, varying in environment to stress the solution, must be completed to exit the Pilot phase.

Plant Production Sites (PRODUCTION Code)
This Circle represents the '**previous major release(s)**' which are running actually Production sites. Based on plant upgrade schedules and downtime availability there will be multiple versions in the field. All of these versions must be supported concurrently.

* **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.

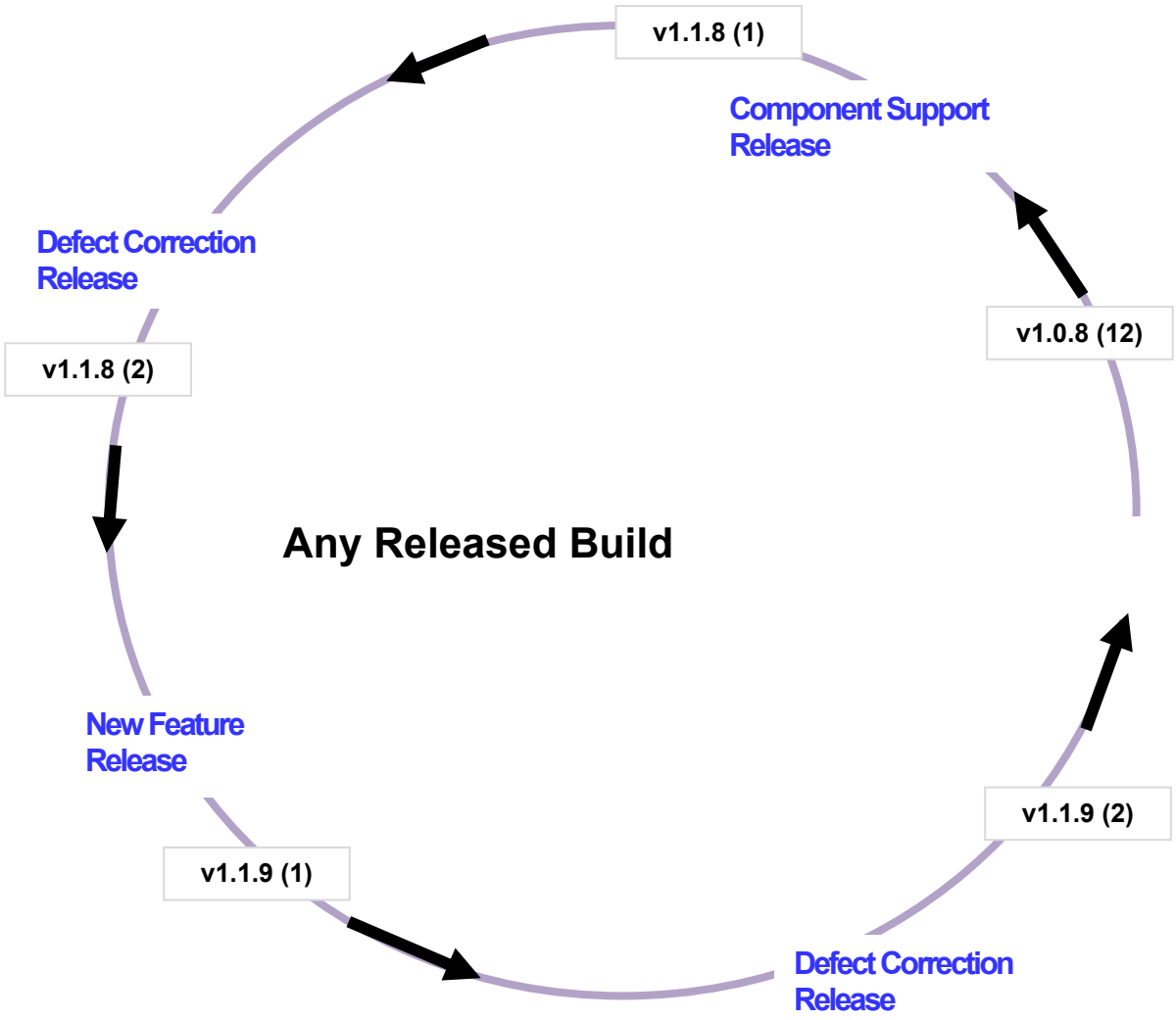


* **Slow**, except for critical Defect Resolution ('Hot Fixes') which must be fast and override other development priorities.

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

Activities

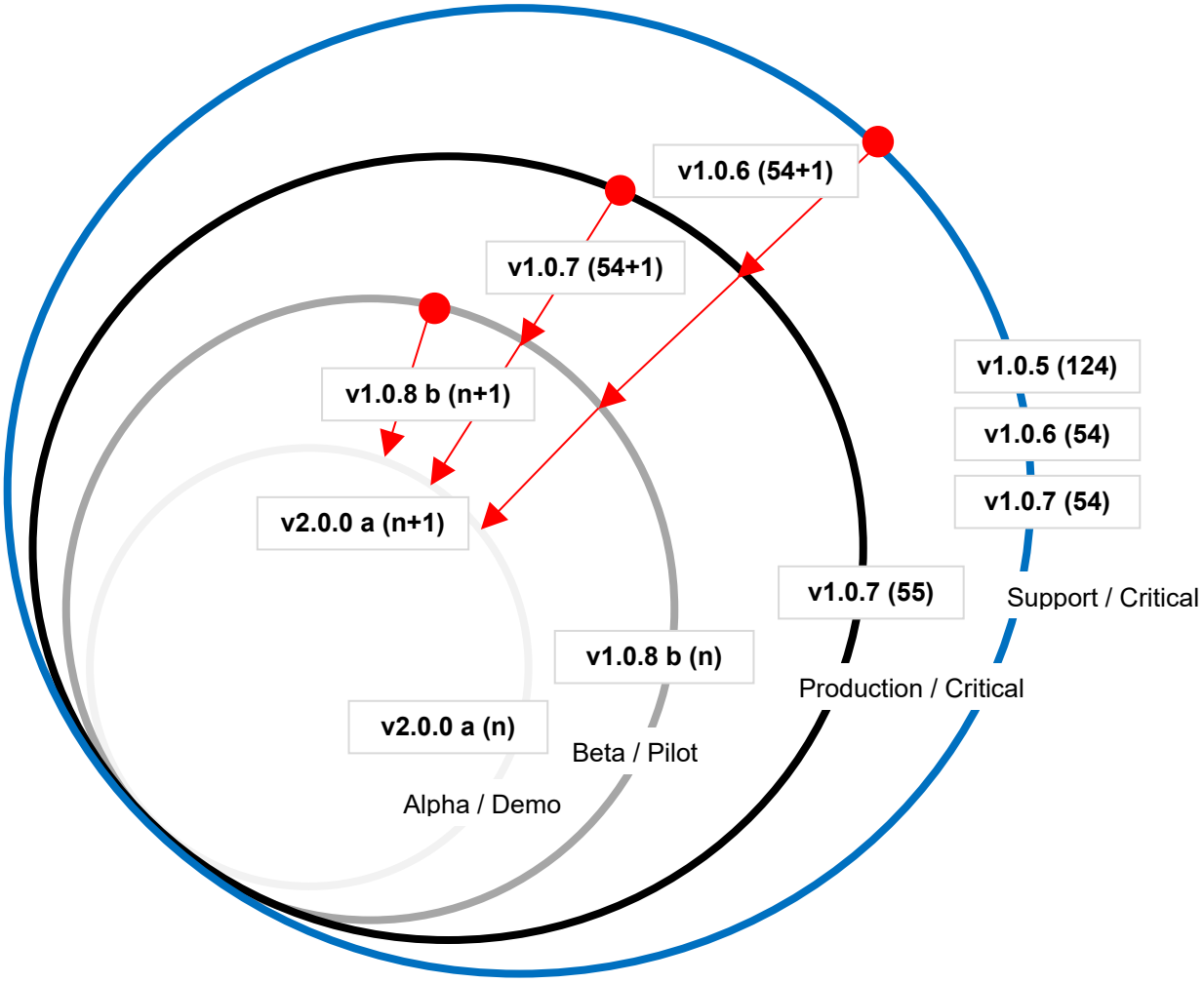
Why represent each version as a 'Circle'?
Because each released version will go through a cycle of multiple builds over its lifetime as a result of a healthy customer support process. This will continue until a Release goes to 'End of Life' and support for that particular Release is terminated, with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



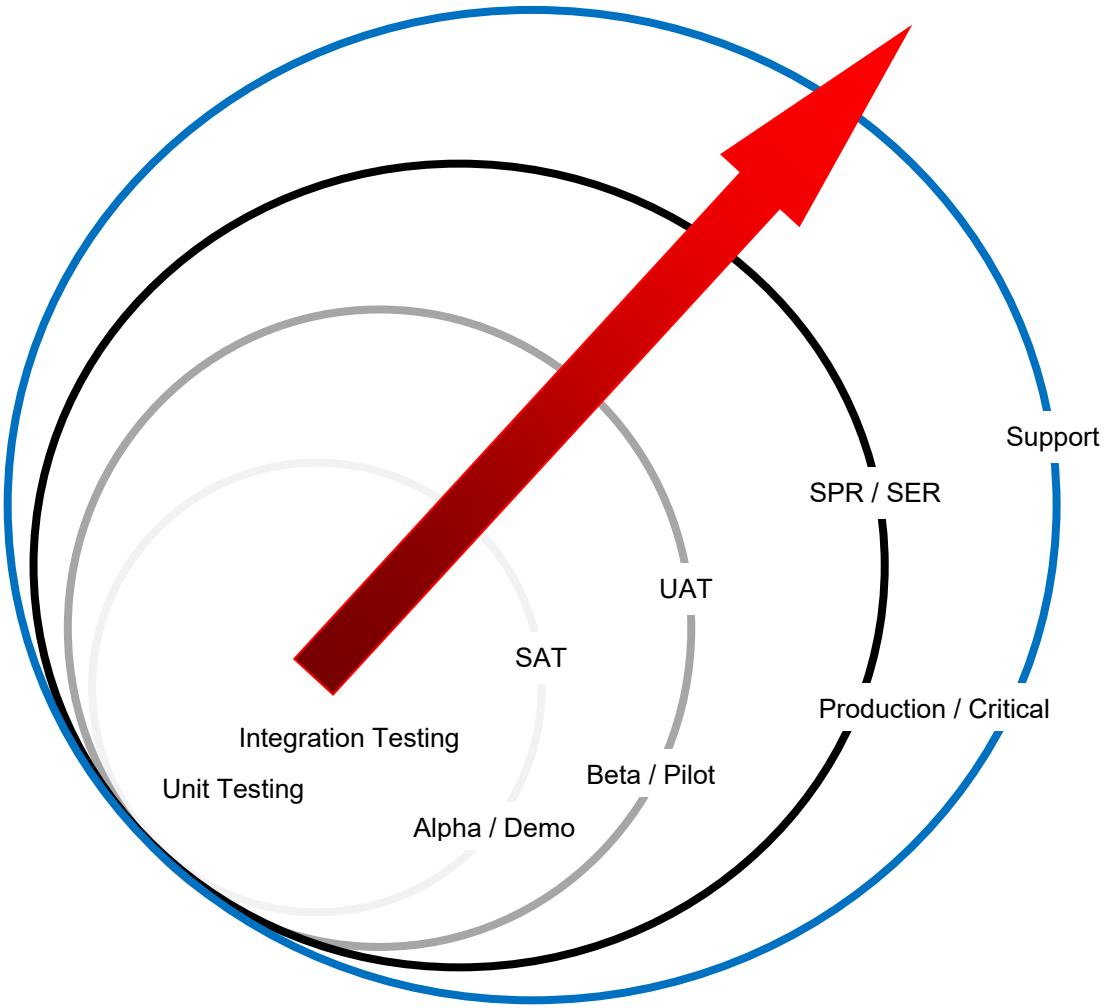
Version Control

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 product version numbering is shown below.*

The **red arrows** illustrate the need to fix defects **first in the version where they arise**…creating a new Build Number (nnn)—immediately shipping the resolution into Production—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.



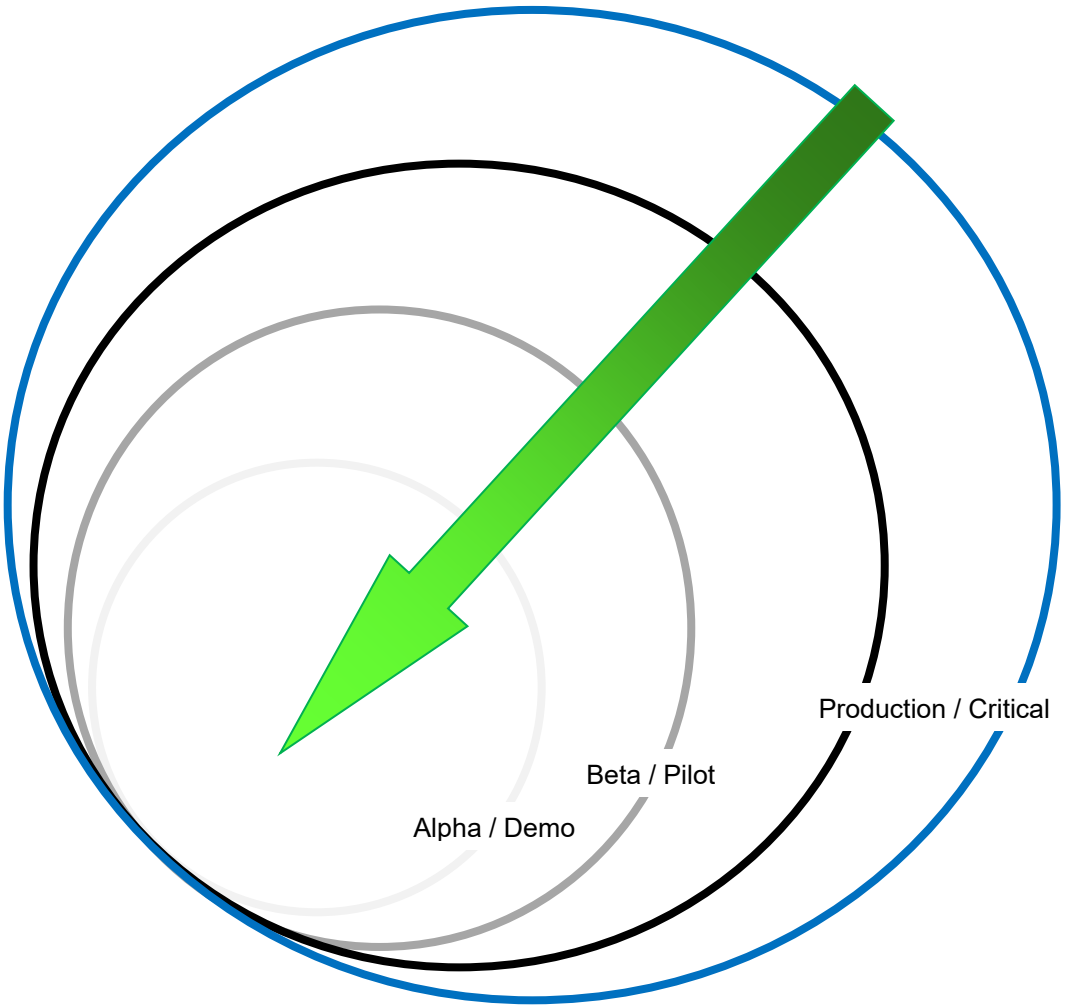
Unit Testing: This happens at the application code level; or, said another way, in the code used to write a standalone program or application. These tests are run by the developer, or development-pair, in the **Alpha, SPR, and SER** Builds.

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.

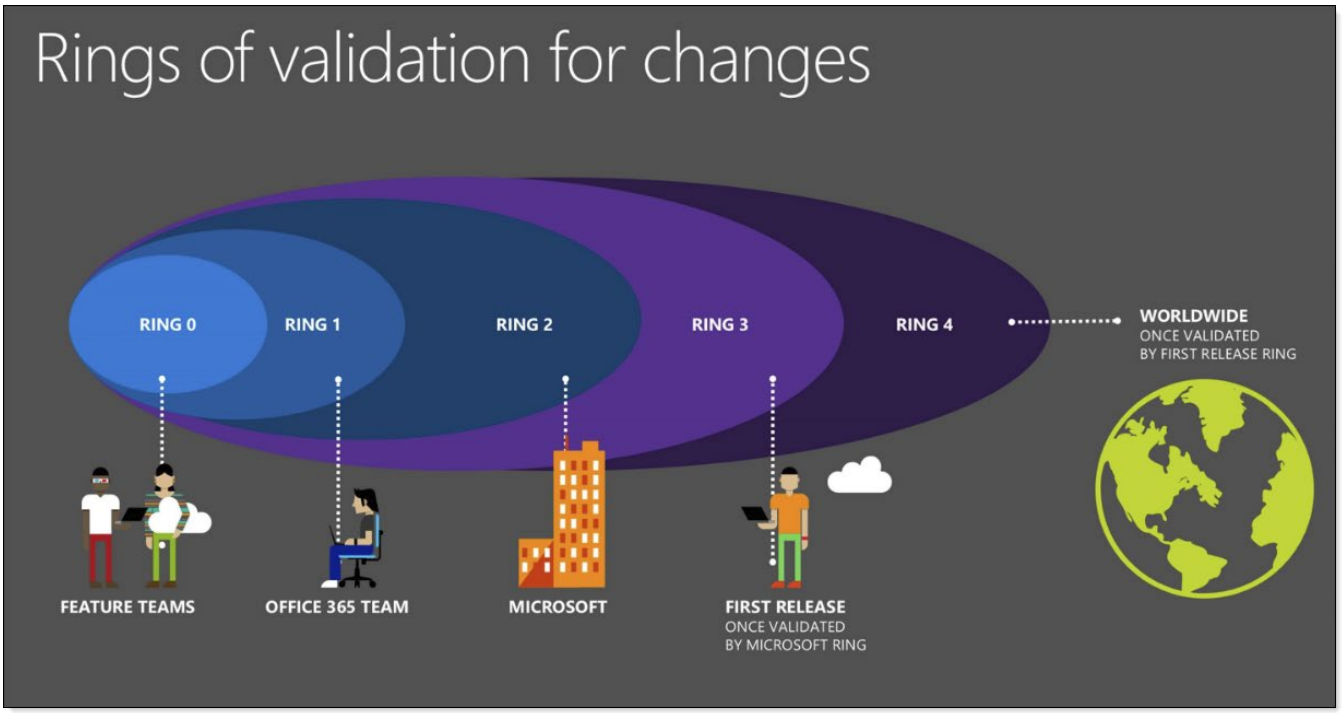
System (Acceptance) Testing (SAT): Validates the complete and fully integrated software product, checking to see if the software works the way they say it is supposed to. These tests are run by the development and support teams in the **Beta** Builds before release to the Pilot Site.

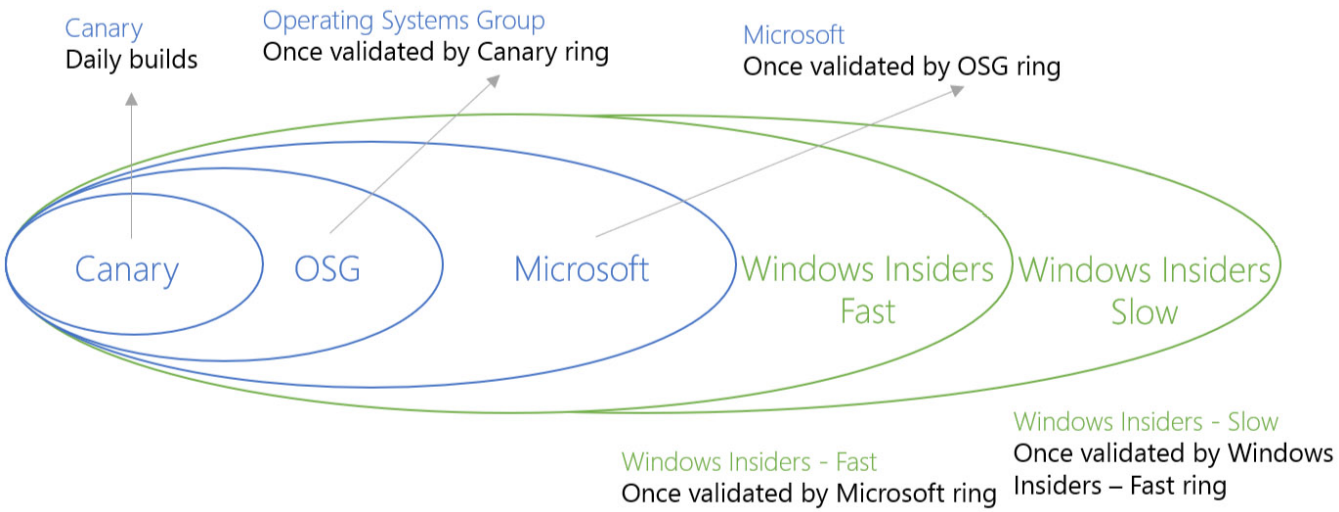
(User) Acceptance Testing (UAT): evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery. These tests are run by the support and customers teams at the Pilot Site in the **Beta** Builds.

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:





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.

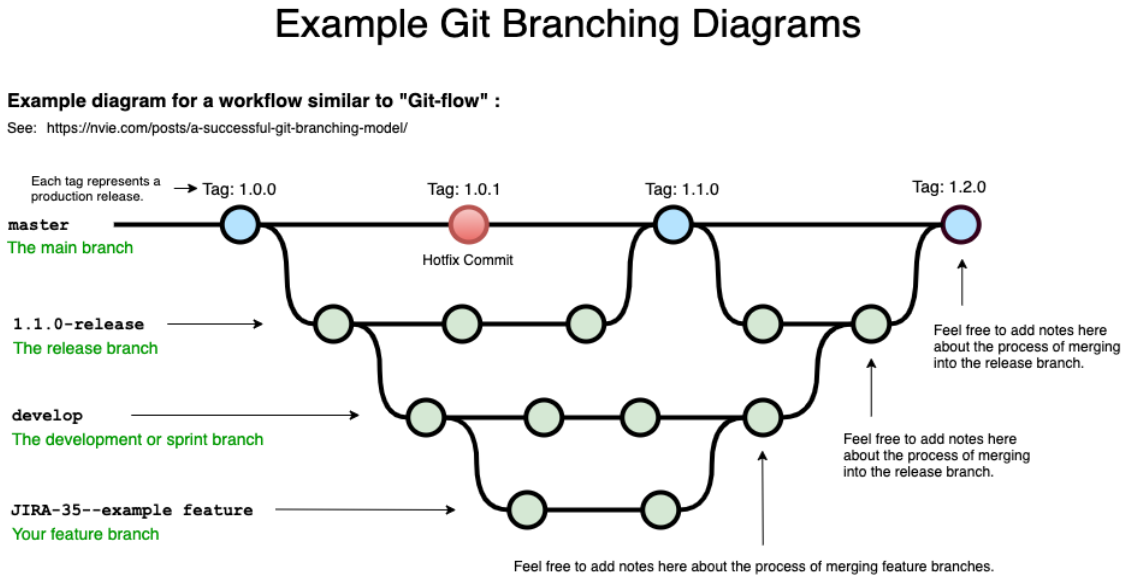
Severity	Description	Impact	Response
1	System crash, complete loss of a major system component	Lost Units or loss of vehicle integrity	Immediate defect resolution and Build Release (**Hours/Days**)
2	Function fails, no work-around is possible, or missed defect or buildable job that was not able to be 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

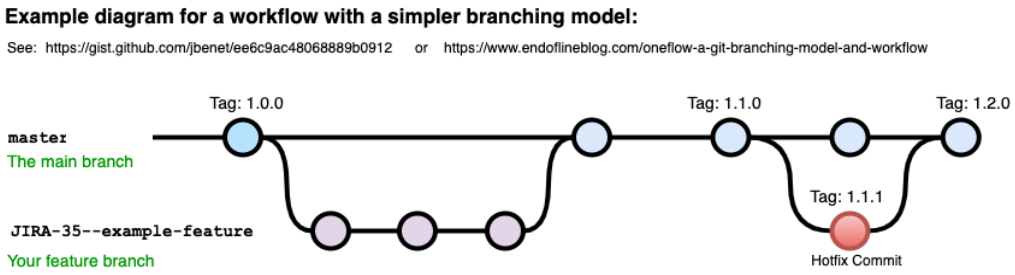
Once severities and responses are defined then support processes can be defined and be held up to those standards for customer support reviews.

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.





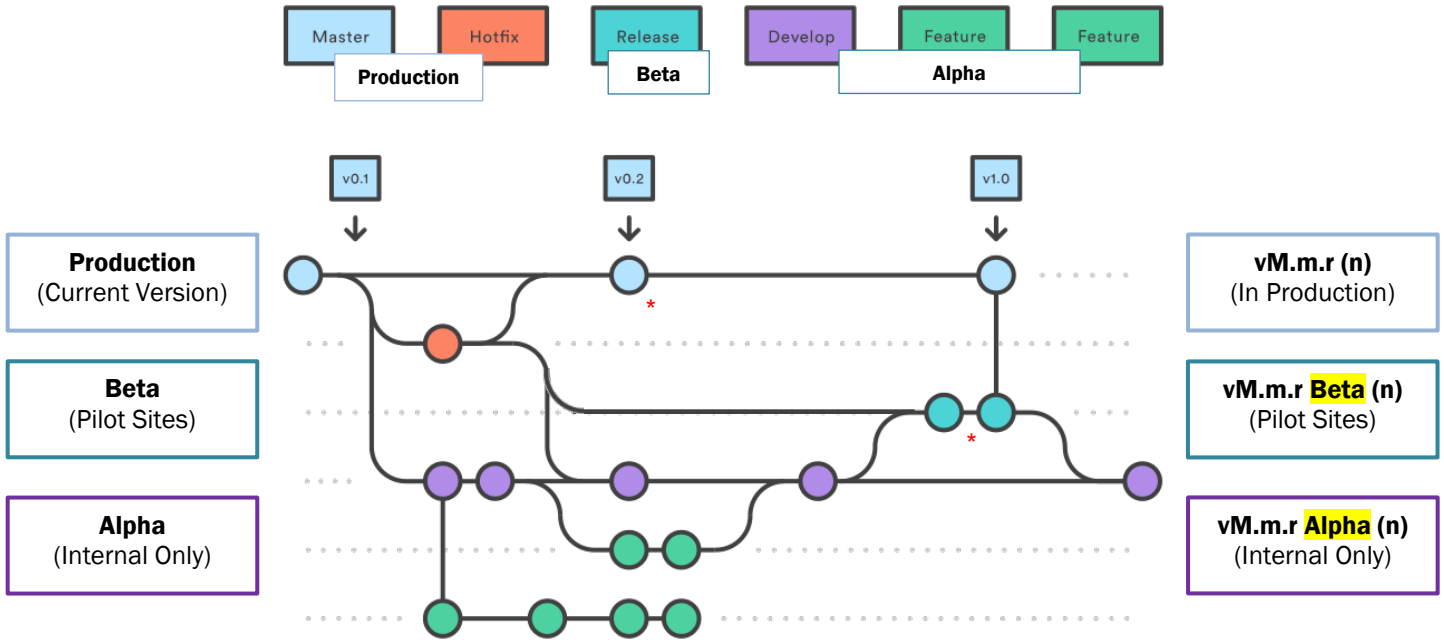
Companies have established internal methodologies for using Git, and some open-source communities have done so as well, one well established method is called **Gitflow**. This is method is available as an extension to **Git** to automatically enforce the rule set.

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

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	Git Flow	MDS	MDS (1980s-1990s Process)
trunk	**master** or main	**Production** + Hot Fixes	MAINTENANCE – Builds running at customer sites
branch	**hot fix**		SPR – Software Problem Report (Defect or Critical Change)
branch	**release**	**Beta** + Hot Fixes	BETA (Builds running internally and/or at Pilot Sites)
branch	**hot fix**		SPR – Software Problem Report (Defect or Critical Change)
trunk	**develop**	**Alpha** + New Features	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.

Git Branching – MicroCODE MDS 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**	`Product-M.m.r-bbb-retired`
branch	**Support** + Hot Fixes	Previous Release **-2**	`Product-M.m.r-bbb-support`
branch		Customer Hot Fix	`Product-M.m.r-bbb-support-sprnnnn`
branch	**Support** + Hot Fixes	Previous Release **-1**	`Product-M.m.r-bbb-support`
branch		Customer Hot Fix	`Product-M.m.r-bbb-support-sprnnnn`
trunk	**Production** + Hot Fixes	Customer Release	`Product-M.m.r-bbb-production`
branch		Customer Hot Fix	`Product-M.m.r-bbb-production-sprnnnn`
branch	**Beta** + Hot Fixes	Pilot Site Beta Release	`Product-M.m.r-bbb-beta`
branch		Customer Hot Fix	`Product-M.m.r-bbb-beta-sprnnnn`
trunk	**Alpha** + New Features	Internal Alpha Release	`Product-M.m.r-bbb-alpha`
branch		Internal New Feature	`Product-M.m.r-bbb-alpha-sernnnn`

* No Hot Fix support, these Sites are actively upgraded to a 'Production' or 'Support' Release.

Examples of Git Branch Names:

`Control_SEP-1.0.0-017-retired` = Control (SEP) App v1.0.0 (017) Archived Release

`Control_EPx-1.0.0-001-support` = Control (EPx) Station v1.0.0 (002) Support Release

`Control_EPx-1.0.0-002-production` = Control (EPx) Station v1.0.0 (002) Production Release

`LADDERS_ABCLX5-1.1.0-011-beta` = LADDERS (Logix5000) v1.1.0 Beta (011) Pilot Site Release

`Control_EPP-2.1.0-007-production` = Control (EPP) App v2.1.0 (007) Production Release

`Control_EPP-2.1.0-008-production-sprnnn` = Control (EPP) App v2.1.0 (009) Hot Fix for {SPR#nnn}

`Defect_EPx-1.0.0-017` = Class Library Defect (EPx) v1.0.0 (017) Production Release

`Control_SEP-2.1.1-011-alpha` = Control (SEP) App Internal Build

`Control_SEP-2.1.1-001-alpha-sernnn` = Control (SEP) App New Feature for {SER#nnn}

MDS – Notes

This naming system always tells the developer six (6) things:

- Product
- Version
- Code Circle
- Starting Build of a Hot Fix development
- Starting Build of a New Feature development

Our process has the following implications, benefits, and requirements:

- Customers are running a proven version of our App or Service at all times (**Production** Trunk)
- We can **Hot Fix** any **Production** version at any time to provide responsive support.
- **Hot Fixes** to a Production Release are carried into the current **Alpha** and **Beta** Builds as appropriate.

- Customers running a **Beta** version—as a Pilot Site—can receive the exact same support as a Production site, i.e.: we can **Hot Fix** any **Beta** version at any time.
- Depending on the scope of the **Hot Fix** it may have to be applied to the current **Production** Release as well, e.g.: If the defect being corrected only has to do with a New Feature in the Beta then only the **Beta** Release is updated; if the **Hot Fix** is a defect found to exist in the current **Production** Release then it must be updated as well.

- **Beta** Releases directly become the next **Production** Release after passing all Pilot Site requirements.
- **Hot Fixes** to a Beta Release are carried into the current **Alpha** Builds.

Application Version Numbers

The application software version numbering is documented as follows…

vM.m.r Circle (bbb)

M = Major software version; represents application architecture, underlying technology, etc.
Incrementing this number is associated with a '**Major Release**'.

m = Minor software version; represents new components within the application.
Incrementing this number is associated with a '**Component Support Release**'.

r = Incremental Release Number; represents collections of new features within the application.
Incrementing this number is associated with a '**New Feature Release**'.

Circle = Development Circle as in ALPHA/DEMO, BETA/PILOT, or PRODUCTION.
In the case of PRODUCTION, the Circle label is removed.

Changing this label is associated with a '**Code Circle Promotion**', i.e.: Internal Build Promotion. This is a rebuild/relabeling only no code is changed, e.g.: v2.0.0 Beta (017) → v2.0.0 (001).

b = Build Number. This is the internal build number of the application from within the development group; any time code is changed **and released into the Support Staff** this number must be incremented, no matter how small the change. This number is expected to fall in the range 001 to 999; more changes than this indicates a flawed test cycle or the implementation of new features that should have incremented 'R' and reset B to 001.

Incrementing this number is associated with a '**Defect Correction Build**' or '**Critical Enhancement Build**'.

Displayed and Documented Version Number Examples:

App_Name v1.0.5 (124) – Production Version found in Customer Sites.

App_Name v1.0.5 (125) – A defect correction or critical enhancement to (124).

App_Name v1.0.6 Beta (007) – Beta Version found only in Pilot Sites and Support Staff Lab.

App_Name v2.0.0 Alpha (089) – Alpha Version found only on Working Machines or in the Support Staff Lab.

Git/GitHub and File Name Version Number Examples:

App_Name-v1.0.5-124 – Production Version found in Customer Sites.

App_Name-v1.0.5-125 – A defect correction or critical enhancement to (124).

App_Name-v1.0.6b007 – Beta Version found only in Pilot Sites and Support Staff Lab.

App_Name-v2.0.0a089 – Alpha Version found only on Working Machines or in the Support Staff Lab.

Defect Resolution Process: 'Defect Correction Release'

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.

Situation: Version **v1.1.9 (22)** of an application is in Production. A defect arises that is deemed Severity **2**.

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 Build, in this example **v1.1.9 (23)**.

> **MDS**: This is when the **-sprnnn** 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 Production Site defect and all affected components pass regression test, each cycle producing a new Build increments the version's Build number. So, the version shipped to the Production Site—and posted for use in other sites—may be **v1.1.9 (23)** if several Developer/Support Staff cycles were required.

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

> **MDS**: This is when the **-beta** branch is created.

Final Step: The Defect Resolution is carried into all inner Code Circles, i.e.: into all other Builds in Production or in development stages in the Lab.

> **MDS**: This is the **merge** activity that occurs when an 'spr' and 'beta' branch is closed.

Minor Release: 'Component Support Release' Process

A minor release would by definition be the addition a new component to 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: An existing version of the application, already in use at the Production Site requesting support for the new component is chosen with agreement from the site, e.g.: **v1.0.5 (124)**.

> **MDS**: This is when the **-sernnn** branch is created.

Step 2: Support for the new component is added, tested, documented and regression testing of any affected components is performed. A **Beta** Build is created for the Pilot site, e.g.: v1.**1**.5 Beta (001).

> **MDS**: This is when the **-sernnn** branch is closed.
> **MDS**: This is when the **-beta** branch is created.

Step 3: The **Beta** Release is piloted at the requesting site. Defects are handled via the standard **Defect Resolution Process**. After a defined time period with no Severity 3 or higher (2, 1) defects the release promoted to 'Production' status and published for any sites to acquire and deploy, e.g.: v1.1.5 Beta (027) → v1.**1**.5 (**001**).

> **MDS**: This is when the **-production** branch is updated.

Final Step: The Component Support is carried into all inner Code Circles.

Incremental Release: 'New Feature Release' Process

A new feature release would by definition be the addition a new feature into an existing version of the application but no changes to the overall application structure. i.e.: An Operator Screen for the Torque Tool Action.

Step 1: An existing version of the application, already in use at the Production Site requesting support for the new component is chosen with agreed from the site, e.g.: v1.1.5 (014).

> **MDS**: This is when the **-sernnn** branch is created.

Step 2: Support for the new component is added, tested, documented and regression testing of any affected components is performed. A **Beta** Build is created for the Pilot site, e.g.: v1.1.**6** Beta (001).

> **MDS**: This is when the **-sernnn** branch is closed.
> **MDS**: This is when the **-beta** branch is created.

Step 3: The **Beta** Release is piloted at the requesting site. Defects are handled via the standard **Defect Resolution Process**. After a defined time period with no Severity 3 or higher (2, 1) defects the release promoted to 'Production' status and published for any sites to acquire and deploy, e.g.: v1.1.6 Beta (007) → v1.1.**6** (**001**).

> **MDS**: This is when the **-production** branch is updated.

Final Step: The New Feature is carried into all inner Code Circles.

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 7 Embedded and .NET 4.0

Step 1: The newest existing version of the application, already in use at the Production Site(s) is selected for the development effort, e.g.: v1.0.8 (61).

> **MDS**: This is when the **-alpha** branch is utilized. The 'alpha' branch always existed to guard work in the 'production' trunk.

Step 2: The application is migrated to the new platform, tested, documented and regression testing of **all functionalities** is performed. An **Alpha** Build is created for the Support Staff site, e.g.: v2.0.0 **Alpha** (001).

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**. After all the STCs are passed with no Severity 3 or higher (2, 1) defects then the release is promoted to 'Beta' status and a **Beta** Build is created for the Pilot site, e.g.: v2.0.0 **Beta** (001).

> **MDS**: This is when the **-beta** branch is created.

Step 4: The **Beta** Release is piloted at the volunteer site. Defects are handled via the standard **Defect Resolution Process**. After a defined time period with no Severity 3 or higher (2, 1) defects the release promoted to 'Production' status and published for any sites to acquire and deploy, e.g.: v2.0.0 (**032**).

> **MDS**: This is when the **-production** branch is updated.

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

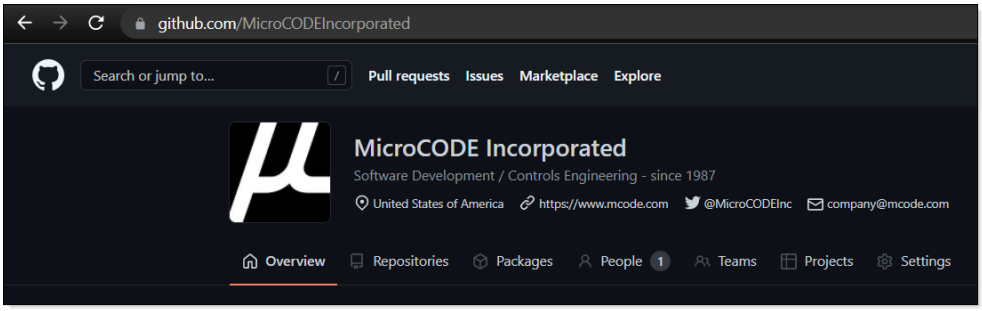
After 30 years of software development and customer support we believe this is the only way to properly support Production Critical Software.

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.).

MicroCODE has a company GitHub…

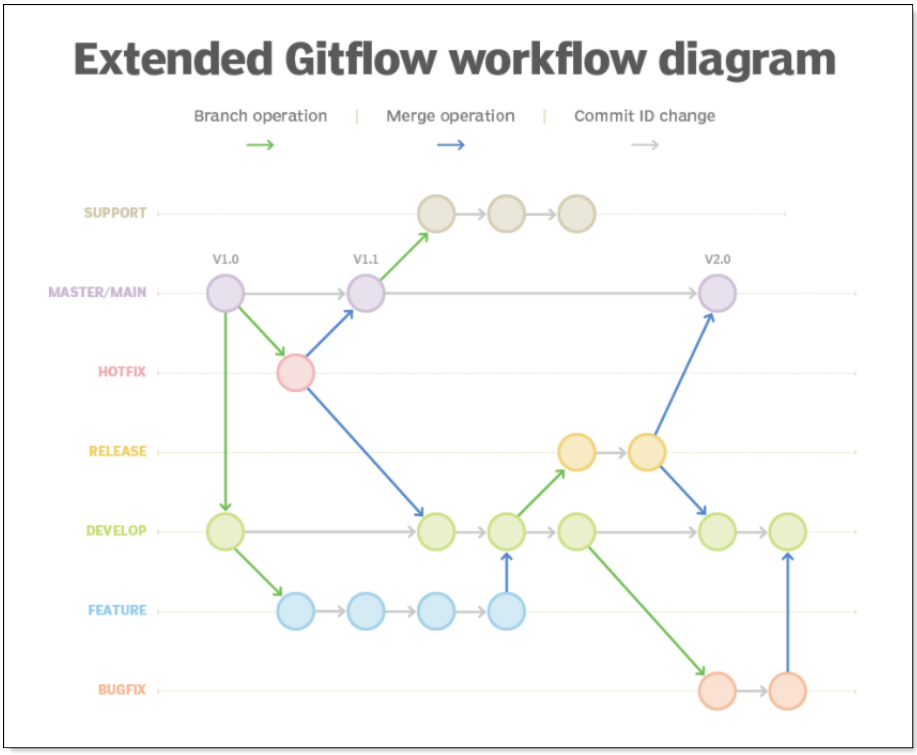


...this is where all the approved 'Trunks' and 'Branches' are held for all company developers.

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

- **production** = the current Build release to all Customers for use in Production
- **beta** = the next Production release while in Pilot Site
- **alpha** = 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.

Getting Started

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.

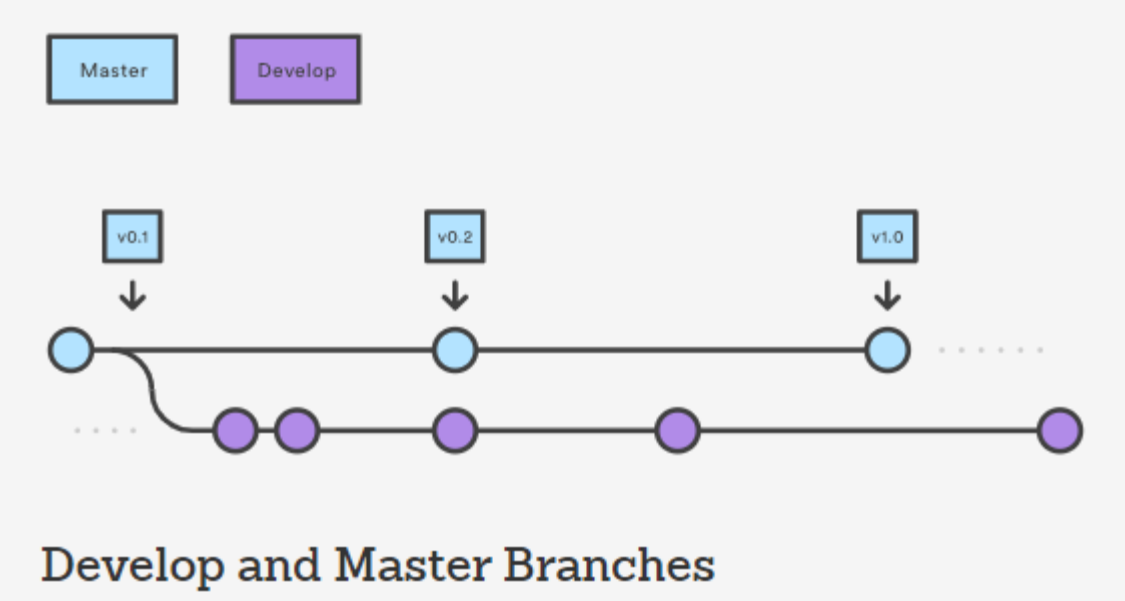
1 Development (Alpha) and Master (Production) Branches

Instead of a single Production ('Master') 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 ('Master') and Development ('Develop').

Note: **MDS** calls the 'Develop' trunk '**Alpha**' and the 'Master' trunk '**Production**'.

- **Production** Trunk: The Production branch contains the production code and stores the official release history.
- **Alpha** Trunk: The Development branch 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 "Production" branch. Then complement the default Production with a Development ("Alpha") branch. 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.

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

- `Product-M.m.r-bbb-production`

1.1 Creating a Development (Alpha) Branch

- Without the git-flow extensions:
  - `git branch development`
  - `git push -u origin devlopment`

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

- **MDS** – Without the git-flow extensions:
  - `git branch Product-M.m.r-000-alpha*`
  - `git push -u origin Product-M.m.r-000-alpha*`

- **MDS** – 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] production
Branch name for "next release" development: [develop]alpha


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


$ git branch
* alpha
production
```

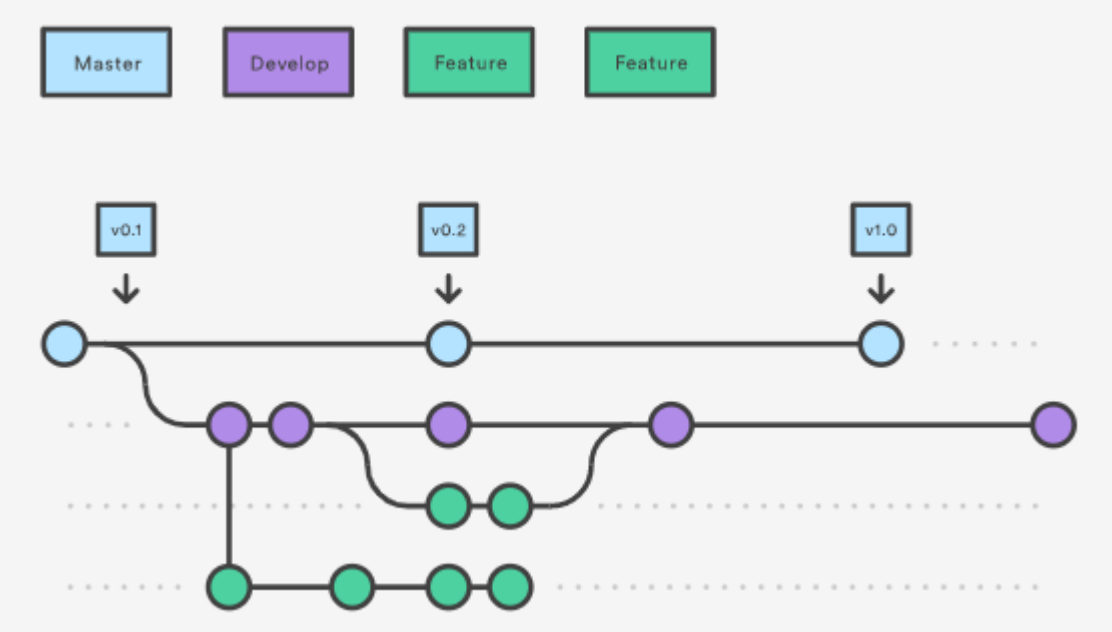
2 Feature (SER) Branch

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 branch.

MDS keeps track of every customer problem and request. Every new feature request is recorded in a Software Enhancement Request (**SER**). 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 SER.
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 (SER) Branch

- **Without git-flow extensions:**
  - `git checkout development`
  - `git checkout -b feature_branch`

- **With git-flow extensions:**
  - `git flow feature start feature_branch`

- **MDS – Without git-flow extensions:**
  - `git checkout alpha`
  - `git checkout -b Product-M.m.r-000-sernnn*`

- **MDS -- With git-flow extensions:**
  - `git flow feature start Product-M.m.r-000-sernnn*`

`* a unique identifier for a new feature, i.e.: the {SER#NNN}, Software Enhancement Request.`

2.2. Finishing a Feature (SER) Branch

- **Without git-flow extensions:**
  - `git checkout development`
  - `git merge feature_branch`

- **With git-flow extensions:**
  - `git flow feature finish feature_branch`

- **MDS – Without git-flow extensions:**
  - `git checkout alpha`
  - `git merge Product-M.m.r-000-sernnn*`

- **MDS -- With git-flow extensions:**
  - `git flow feature finish Product-M.m.r-000-sernnn*`

`* a unique identifier for a new feature, i.e.: the {SER#NNN}, Software Enhancement Request.`

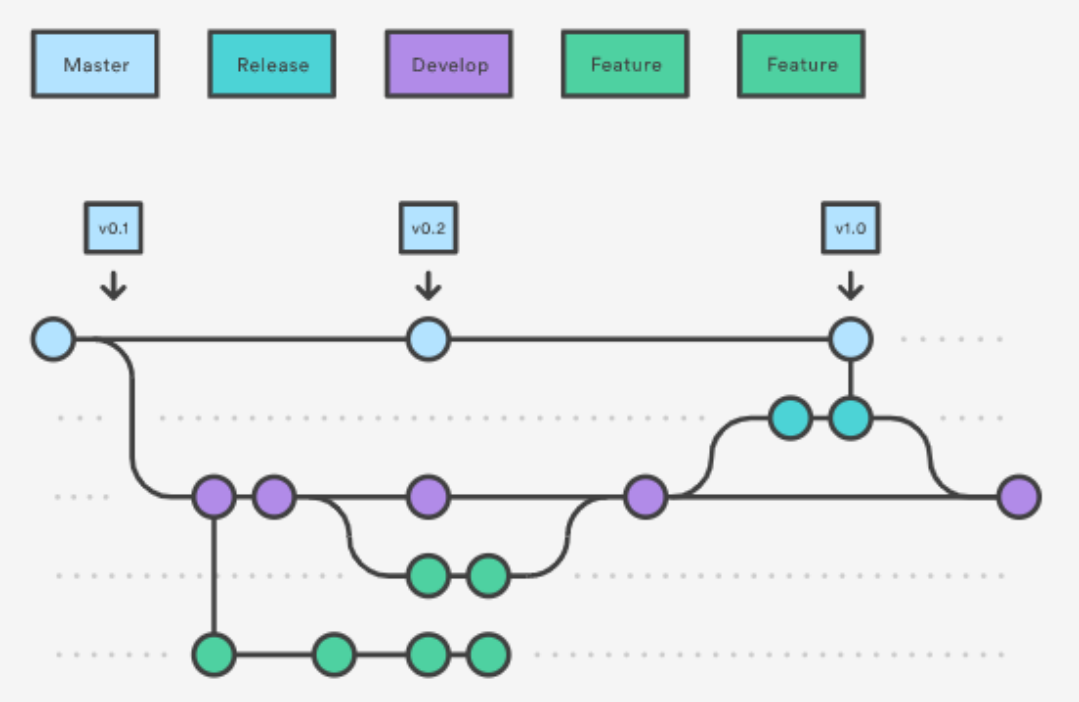
3 Release (Beta) Branch

Once Development has acquired enough features for a release (or a predetermined release date is approaching), we fork a Release branch off of development. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Release branch may branch off from Development and must merge into both Production and Development.

Note: **MDS** calls the 'Develop' trunk '**Alpha**' and the 'Release' branch '**Beta**'.

- **Alpha** Trunk: This Development branch contains pre-production code and serves as an integration branch for New Features.
- **Beta** Branch: This Development branch contains a pilot-ready 'Production Candidate' that has a collection of New Features (SERs) and all Hot Fixes (SPRs) 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 and tagged with a version number. In addition, it should be merged back into Development because critical updates (Hot Fixes) may have been added to the Release branch and they need to be included with the New Features. So, once the Release is ready to ship, it will get merged into Production and Development, and then the Release branch will be deleted, because the Production version will become the Beta copy with a rename that removes the 'Beta' designation.

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.

3.1. Creating a Release (Beta) Branch

- **Without the git-flow extensions:**
  - `git checkout development`
  - `git checkout -b release/0.1.0`
- **When using the git-flow extensions:**
  - `git flow release start 0.1.0`
- **MDS – Without git-flow extensions:**
  - `git checkout alpha`
  - `git checkout -b Product-M.m.r-000-beta*`
- **MDS -- With git-flow extensions:**
  - `git flow release start Product-M.m.r-000-beta*`

* a unique identifier version number (M.m.r-000) must be assigned based on the changes, the -000 (Build #) should be the Build # in Production when the hot fix activity began.

3.2. Finishing a Release (Beta) Branch

- **Without git-flow extensions:**
  - `git checkout production`
  - `git merge release/0.1.0`
- **With git-flow extensions:**
  - `git flow release finish 0.1.0`
- **MDS – Without git-flow extensions:**
  - `git checkout alpha`
  - `git merge release/Product-M.m.r-000-beta*`
- **MDS -- With git-flow extensions:**
  - `git flow release finish Product-M.m.r-000-beta*`

* a unique identifier version number (M.m.r-000) must be assigned based on the changes, the -000 (Build #) should be the Build # in Production when the hot fix activity began.

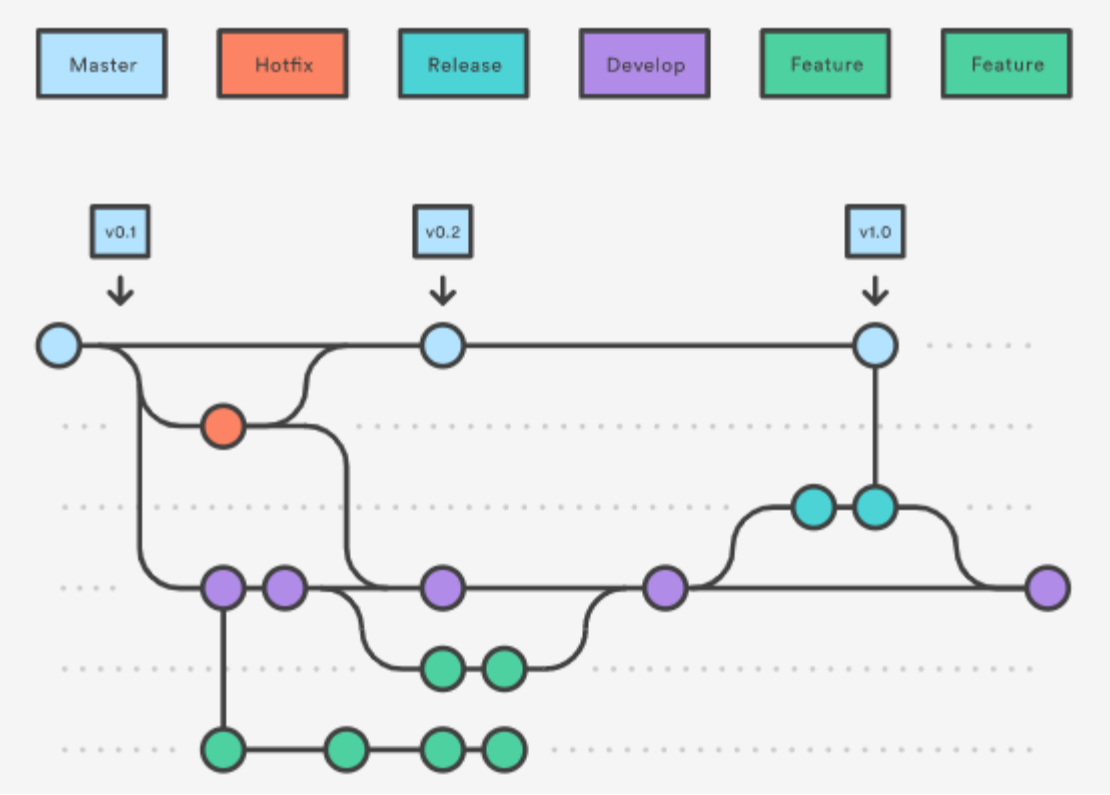
4 Hotfix (SPR) Branch

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.

MDS keeps track of every customer problem and request. Every actual problem is recorded in a Software Problem Report (**SPR**). 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 the first SPR.
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 (SPR) Branch

- **Without git-flow extensions:**
  - git checkout Production
  - git checkout -b hotfix_branch
- **With git-flow extensions:**
  - git flow hotfix start hotfix_branch

- **MDS – Without git-flow extensions:**
  - git checkout alpha
  - git checkout -b Product-M.m.r-000-sprnnn*

- **MDS -- With git-flow extensions:**
  - git flow feature start Product-M.m.r-000-sprnnn*

* a unique identifier for a hot fix, i.e.: the {SPR#NNN}, Software Problem Report, the -000 (Build #) should be the Build # in Production when the hot fix activity began.

4.2. Finishing a Hotfix (SPR) Branch

- **Without git-flow extensions:**
  - git checkout production
  - git merge hotfix_branch
  - git checkout develop
  - git merge hotfix_branch
- **With git-flow extensions:**
  - git branch -D hotfix_branch
  - git flow hotfix finish hotfix_branch

- **MDS – Without git-flow extensions:**
  - git checkout production
  - git merge Product-M.m.r-000-sprnnn*
  - git checkout alpha
  - git merge Product-M.m.r-000-sprnnn*

- **MDS -- With git-flow extensions:**
  - git branch -D Product-M.m.r-000-sprnnn*
  - git flow release finish Product-M.m.r-000-sprnnn*

* a unique identifier for a hot fix, i.e.: the {SPR#NNN}, Software Problem Report, the -000 (Build #) should be the Build # in Production when the hot fix activity began.

Advantages of Git Flow

Now let's talk 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 disadvantage 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 'MDS' and the protection of the 'Production' branch out weights all other considerations.**

Summary

Here we discussed the Git Flow Workflow. Git Flow is one of the many styles of Git workflows you and your team can utilize. The overall flow of Git Flow is:

1. A `development` branch is created from `production`
2. A `release` branch is created from `development`
3. All `feature` branches are created from `development`
4. When a `feature` is complete it is merged into the `development` branch
5. When the `release` branch is done it is merged into `development` and `production`
6. If an issue in `production` is detected a `hotfix` branch is created from `production`
7. Once the `hotfix` is complete it is merged to both `development` and `production`

The overall workflow of MDS is:

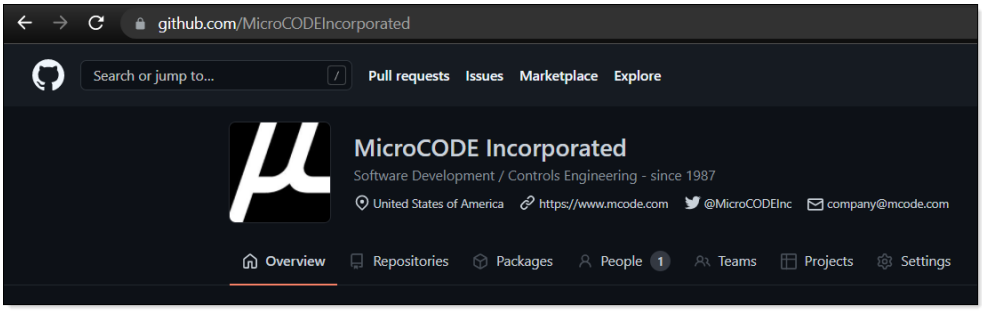
1. An `alpha` branch is created from `production`
2. A `beta` branch is created from `alpha` after all development work is ready for a pilot
3. All `ser` branches are created from `alpha`
4. When a `ser` is complete it is merged into the `alpha` branch
5. When the `beta` branch is done (pilot completed) it is merged into `alpha` and `production`
6. If an issue in `beta` is detected a `spr` branch is created from `beta`
7. Once the `spr` is complete it is merged to both `beta` and `alpha` and possibly `production`
8. If an issue in `production` is detected a `spr` branch is created from `production`
9. Once the `spr` is complete it is merged to both `alpha` and `production` and any open `beta`

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.).

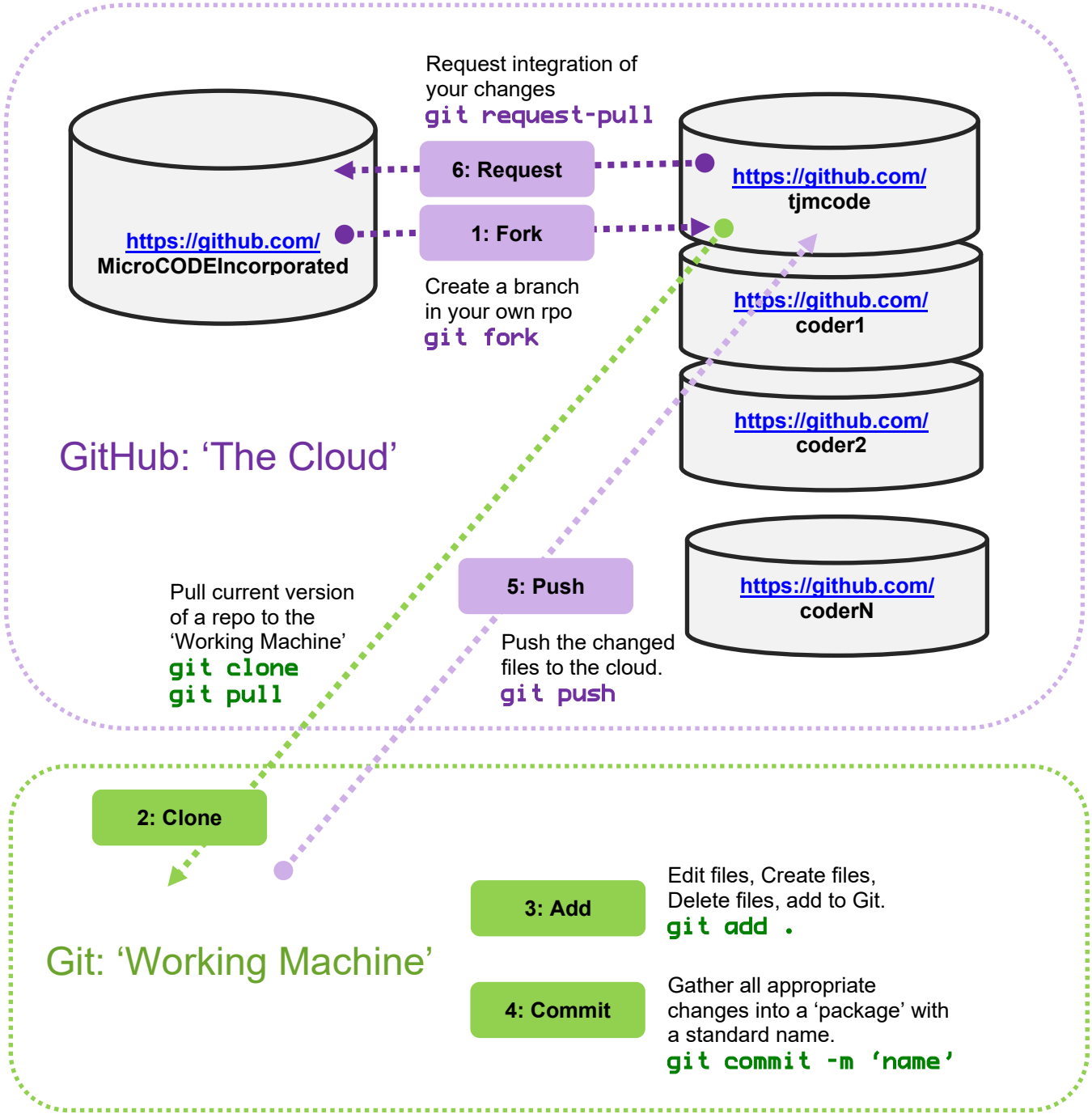
MicroCODE has a company GitHub…



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

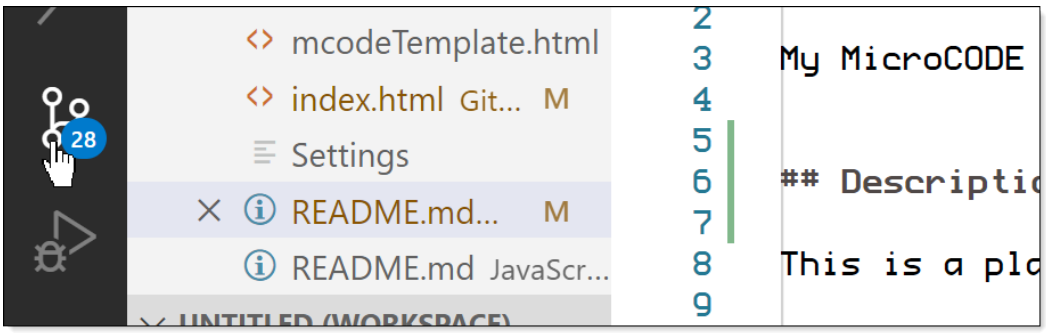
Starting a Hot Fix for a Software Problem Report

1: Fork the Production Release for which you are creating the Hot Fix into your working GitHub Account
2: Clone the Copy of Production Release to your Working Machine
3: Add any files you change on your Working Machine into your Git
4: Commit your changes with a comment: "`Product-M.m.r-bbb-production-sprnnn`"
5: Push syncs your changes into your remote repository in GitHub.

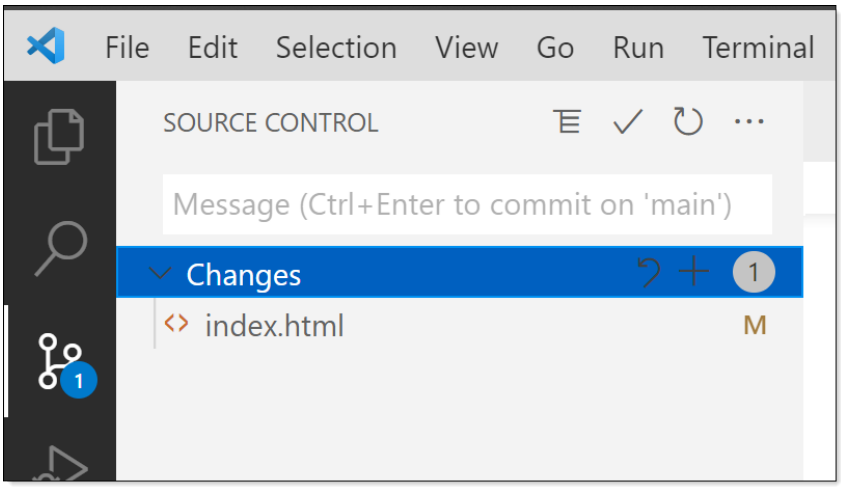


Visual Studio Code – Integration

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

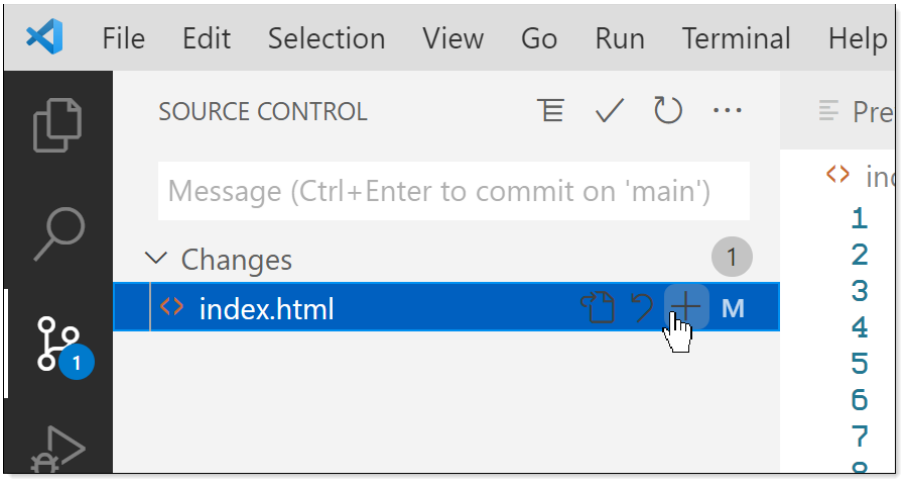


Clicking on the GIT icon opens the 'Commit Review' pane.

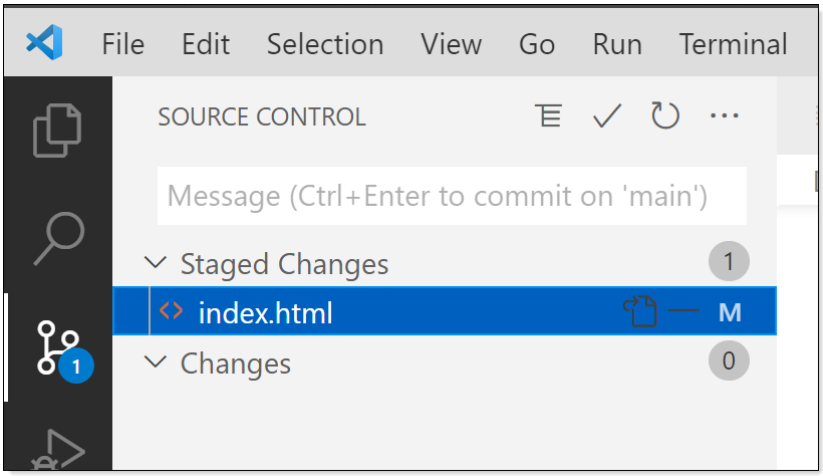


Touching a file immediately shows a DIFF split window…

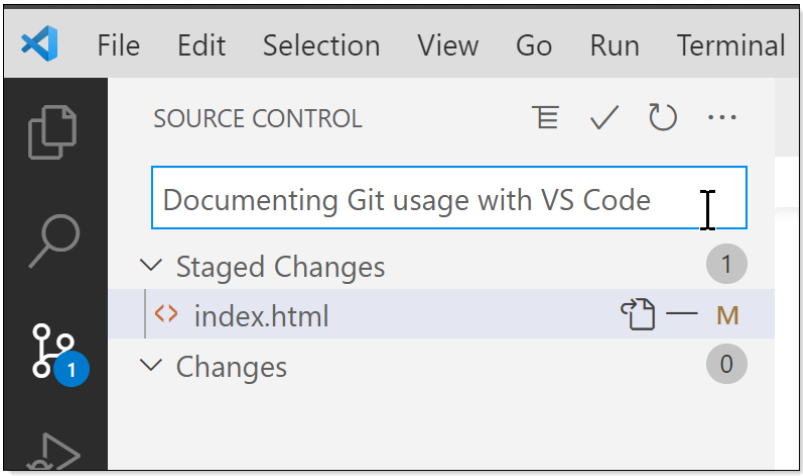
Clicking the "+" performs a "`git add`" of that file to the 'Staged Files' for a 'Commit'…



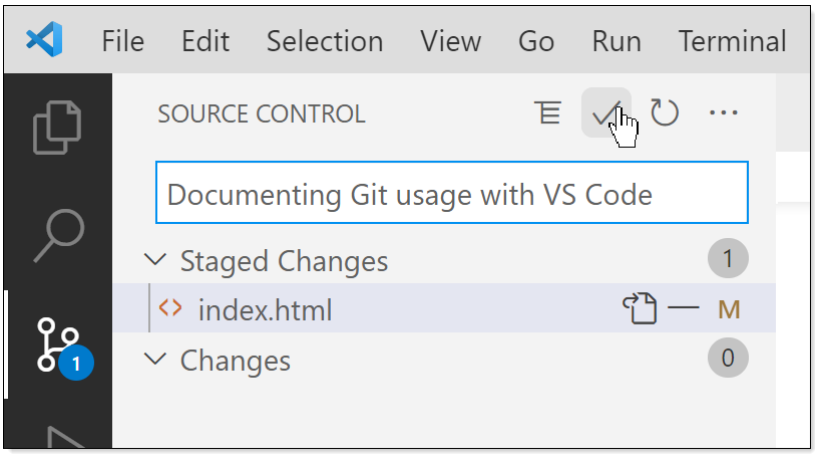
Staged files for a 'Commit'…



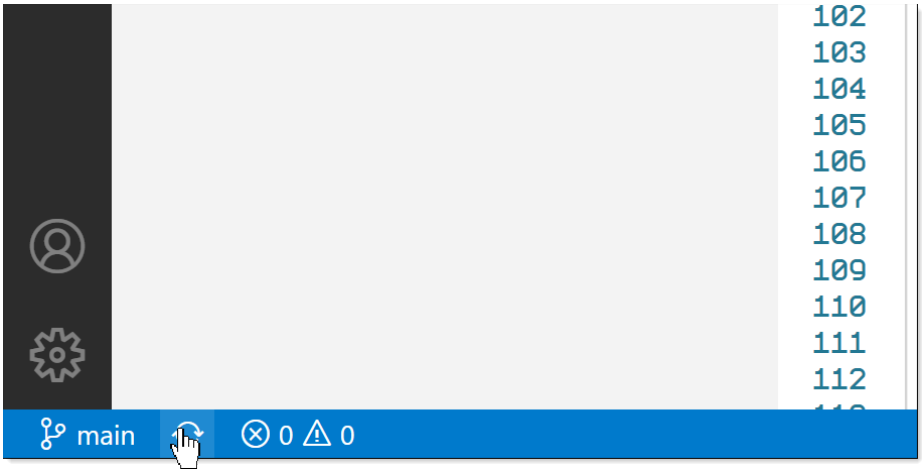
Add a description for the 'Commit'…



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



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’…