**Ticket Assignment**: Tasks will typically come from three different sources namely support work, scrum work and projects typically being the new builds. Depending on source of task, the tasks may come from Monday.com or Jira software’s, where the technical and QA information is typically stored and kept up to date. Basically some feature developer or QA person they’ll take up the ticket and if someone’s deploying at later dates that’ll include:

* Module and Branch names.
* Non code changes (any Magento config settings, any Ansible vars changes where we control environment variables).
* Notes for QA team that might be good to know for them.

**Branch Naming**: When developer begins work, they’ll typically branch from the latest stable versions of the module or market repositories etc. e.g. ‘main’ Branch naming usually follows the following structure:

* Feature/[project code]JIRA code[support ticket number]/short-description

Initial development will either occur in a local version of the Sandbox site or local version of a Production site, depending upon how many markets the feature will be if a feature is for all the markets then we’ll put into Sandbox site acts as a first testing platform and we get any bugs out before it’s going for staging stages. If the feature is only for one market, then we’ll skip the above step. Once initial development is complete, these feature branches will be journey through the environments and being merged accordingly.

**Repository Structure**: Each site or market has its own repository, within which it uses composer to ‘pull in’ modules that are stored in other repository and place them in a vendor folder for e.g. India and Mexico both shares the Bold chat and CMS module. Since modules are shared across multiple markets, the advantage will allow us to update a module once and roll it out to the other markets, as opposed to updating every site repository individually with the same update.

**Environments**: Depending on whether the feature is market wide or a bespoke task for a particular market or a group of markets. The development typically consists of four or five environments. These environments are Local, Staging, Sandbox, Pre-Production and Production. Local, Staging, Pre-Production and Production are always used. Sandbox is only used when a feature, bug fix etc. is market-wide Sandbox is used for initial testing before it’s rolled out to the various environments. The purpose of each environment is as follows:

* Local – Vagrant environment where initial development and initial testing happens
* Sandbox – Digital ocean environment where first deploy and initial QA takes place if feature is market-wide.
* Staging – AWS environment where all QA, end to end testing and sign off/approval takes place here
* Pre-Production – Digital ocean environment that mirrors Production in terms of codebase where final test takes place
* Production - public AWS environment

**Branching Strategy**: During the development process, both site and module repositories can be updated depending upon the requirement. Sometimes change might be required in India’s site repository say as it has to deal with payment charge things and sometimes it could be a module change say Dyson CMS just needs to update and in other times , we might need to update both site and module repositories. This process is followed by new feature work as well as fixes for support tickets. Each module has develop/xx branch per market and one for sandbox e.g. for India the branch is develop/in, for Australia branch is develop/au etc. This allows us to merge changes to sites at different times without interruption of the other sites. Each market has develop, release\_candidate and main branch for various phases and environments. The develop branch is for sandbox and staging environments whereas release\_candidate is for Pre-Production environment. Similarly release\_candidate can be merged to main branch at production environment. Branch names are kept consistent across site and module repositories for easy identification. For e.g. feature/1234-new-feature will be the name of the market and module repositories.

**Merging Strategy for Sandbox**: Sandbox repository is essentially another market, used internally by G&V to test the features before they’re rolled out to market staging sites. When work is initially complete on local environment, assuming the feature is market-wide, the first step will be to compile the work ready for sandbox. This step is skipped if the project is for a specific market or group of markets. There is no “production” branch or “main” branch for sandbox because there’s no production site as sandbox is just an internal one used by G&V so we always use the develop branch and merge our feature branch into it:

* git checkout develop (checkout step for develop branch)
* git merge feature/1234-demo-branch (merge step for future branch to develop branch)

As most of modules are similar to market-wide If any modules are worked in, these also need to be merged into the develop/sandbox branch of those modules:

* + - git checkout develop/sandbox (checkout step for develop/sandbox branch)
    - git merge feature/1234-demo-branch (merge step for future branch to develop/sandbox branch)

**Merging Strategy for Staging**: Again, if the work is market-wide the first merge would be to Sandbox. Otherwise, for Staging environment will be the first merge to get it to the specific market. Either way, the steps are the same. If the feature is for all markets, we will group markets by certain criteria and deploy to a selected market candidate from each group first. E.g. Magento Commerce sites, Right to Left (RTL) markets etc. By doing this, if any bugs are found on the initial candidates, we can resolve before rolling out to the remaining markets and save time on compiling and deployments. Merging a feature branch into the market repository’s develop branch:

* git checkout develop (checkout step for develop branch)
* git merge feature/1234-demo-branch (merge step for future branch to develop branch)

For the module work, merging a feature branch into develop/xx branch for that module. In this example, Mexico’s as we select Mexico from a group of markets without disturbing other markets:

* git checkout develop/mx (checkout step for develop/mx branch)
* git merge feature/1234-demo-branch (merge step for future branch to develop/mx branch)

This merge process and the deployment is then repeated for all markets that are applicable for the new feature.

**Merging Strategy for Pre-Production**: When everything goes well in staging environment the next environment is Pre Production where we do final internal smoke tests. Each market has a Pre-Production site which sits on a Digital Ocean droplet. On Pre-Production certain elements of the site are limited when it comes to testing such as:

* Emails
* Back office integrations
* Payment gateway order authentication

Those above mentioned were related to Third Party Integrations hooked up to the application server and testing for these can be done in the previous staging environment (AWS environment). Each market has a release\_candidate branch which is an exact copy of the main branch of the market, and used for deploying to their Pre-Production environment. So this is where our market feature branch is merged into:

* git checkout release\_candidate (checkout step for release candidate branch)
* git merge feature/1234-demo-branch (merge step for future branch to release candidate branch)

For modules, this is the point where we create a new shippable version of the module. To do this we merge the feature branch into the latest main branch of the module:

* git checkout develop/mx (checkout step for develop/mx branch)
* git merge feature/1234-demo-branch (merge step for future branch to develop/mx branch)

We then tag the commit according to semantic versioning and run composer to “require” the latest version of the module:

* git checkout release\_candidate (checkout step for release candidate branch)

For e.g. say latest version of cms module is 1.0.1 that will allows you to commit and push the composer changes to the release candidate branch and that would be deployed to the pre-production site.

* composer require module dyson/cms-module:1.0.1

**Merging Strategy for Production**: Everything has been merged into the release\_candidate branch in the Pre-Production environment. All the composer changes and feature branches merges into the release\_candidate branch will become the new shippable version of market and we finally merge our release\_candidate branch to ‘main’ branch for the deployment:

* git checkout main (checkout step for main branch)
* git merge release\_candidate (merge step for release\_candidate branch into main branch)

**Deployments (Dyson CLI)**: Irrespective of deployment to various environments, we always use the same tools and commands which are written in ansible. Dyson-cli is a bespoke wrapper for ansible playbooks that semi-automate the config of environments and deployment of code. These will always be executed by a developer from their local or remote vagrant which sits inside a vagrant machine, and uses their ssh-key and the AWS cli to run commands. Dyson CLI commands can be re-run without damage to an environment (e.g. site:setup command can be re-used to enable/disable basic auth). These commands uses the vars and state of in dyson-vagrant group\_vars where we have all our configurations for e.g. branch names, if branch name is equal to release\_candidate for the Pre-Production config etc. If there is an emergency of deploying to a different branch to your staging or pre-production environment. We don’t need to go for merging of newly created branch to develop or release\_candidate branch. We can map a new branch name with branch name in dyson-vagrant group-vars for the deployment without altering the common environments. Usage is dyson <command> <environment> <site-code> (can use the same commands on dev, staging, pre-prod, prod). Some of the frequently used commands are:

* dyson provision - installs all prerequisites to an environment (php modules, node, local services: mysql, redis, etc)
* dyson site:setup - configures market-specific settings for the environment (e.g. nginx and postfix configs)
* dyson site:db:pull/push - pull or push databases from/to remote and local environment.

Note, you can only ‘push’ to a Pre-Production environment on Digital Ocean and we can pull from Production environment.

* dyson site:uploads:pull/push - pull or push shared files from/to remote and local environment
* dyson site:admin/frontend:ip - display the IP addresses for admin or frontend EC2 instances

The above mentioned Dyson CLI commands will run the ansible playbooks and those playbooks were kept in Dyson vagrant repository.

**Pipeline/The Actual Deployment**: The Dyson CLI doesn’t offer a full end to end automatic CI/CD style pipeline. Dyson-cli plays the Ansible deploy role which includes many sub-tasks and logic to run specific tasks depending on if the environment is AWS (staging/prod) or Digital Ocean (pre-prod). Dyson have a fully automated deployment with manual branch management, merge and release building in between so it follows is a flow chart for what actually happens when you run a deploy on AWS – i.e. dyson deploy prod au (The command is used to deploy in production environment of Australia).

* + The first step is to trigger a db backup (via AWS lambda function) to s3 Bucket
  + Pull latest Magento code from repo into a new release folder with anistrano
  + Runs composer install on new release
  + Pulls latest db backup from s3 bucket and imports it to a local temp db
  + Applies overrides (files not in market repos) for specific markets/Magento versions
  + Runs setup:upgrade command on new release
  + Sets production mode and runs setup:di:compile on new release
  + Runs setup:static-content:deploy on new release
  + Installs and runs npm and gulp styles and sets correct file permissions
  + Create/update magento cron jobs on admin instances
  + Creates env.php file that uses live RDS db (instead of local tmp db)
  + Enable maintenance mode and run setup:upgrade --keep-generated
  + Flush magento and php cache and disable maintenance mode
  + Make new release folder the current release (site is now live) and cleanup old releases
  + Create new AMI for frontend EC2s and re-create them in ASGs from this new AMI

All of these above tasks/steps effectively run into a new folder. If any task fails, the deployment process is aborted and the previously folder is left in touch. If we manage the deployment in all the environments then this make the new release folder as current release. We can abort the deployment at any step manually without any consequences. We run RDS clusters in AWS and we still have MySQL running in localhost so we pull the db backup from cluster and put into localhost so our release folder will now got database to work without touching the actual production database over on RDS. We then let magento go through the setup:upgrade; setup:di:compile; setup:static-content:deploy and if those steps works without errors then make that release folder as actual release. Refresh the env.php file which we’ll actually point to the new release to the actual RDS again and then again we’ll do backup with the new release running. We got the Nginx web folder is symlinked to the latest release folder which consists next to it and so without switching instructions around. We got a good release folder that is symlinked to new release folder and then delete the oldest one.