ADS 18f Prototype

Safe Dose

Program Management Plan

Job 4QTFHS150004

4QTFHS150004

Revision - Dr

June 18, 2015

**Distribution Statement: This document is for internal planning purposes only.**

**Document Authorization**

|  |  |
| --- | --- |
| **Review/Approval Title** | **Name** |
| **Originator:**  Technical Architect | Jared Ladner |
| **Approved By**:  Program Manager |  |
| **Approved By**:  Project Lead |  |
| **Approved By**:  Senior Management |  |
| **Approved By**:  Contracts |  |
| **Approved By**:  Quality Assurance |  |

Electronic signatures were provided on <DATE>

**Document Revision History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Version/ Revision*** | ***Description of Change*** | ***Chg’d By*** | ***Chg/Rel #*** | ***Date*** |
|  |  |  |  |  |
|  |  |  |  |  |

**Table of Contents**

1 Project Overview 6

1.1 Project Scope and Objectives (Project Vision) 6

1.1.1 Customer 6

1.1.2 Project Vision 6

1.1.3 Schedule Summary 6

1.1.4 Deliverables 6

2 References 8

2.1 Source Documentation 8

2.2 Project Working Documents 8

3 Organization 9

3.1 External Interfaces: Who do you interface with outside of Geocent for this project? 9

3.2 Internal Structure: Show a model of the Geocent program management structure for this project 9

3.3 Roles and Responsibilities: List the roles, responsibilities and individual within Geocent for your project. 11

4 Managerial Plans 12

4.1 Start-Up Plan 12

4.1.1 Establish Resources 12

4.2 Execution Plan 14

4.2.1 Manage Requirements 14

4.2.2 Manage Communications 15

4.2.3 Manage Quality 16

4.2.4 Manage Measurements (Metrics) 16

4.2.5 Manage Issues 16

4.2.6 Manage Contract Changes 17

4.3 Work Plan 18

4.4 Closeout Plan 18

**Table of Figures**

[Figure 1. Program Organization Chart 9](#_Toc158536968)

# Project Overview

This document establishes the Program Management Plan for the <TBD> program and serves as the execution plan required to manage the program.

## Project Scope and Objectives (Project Vision)

The Project Vision is defined here: <https://github.com/Geocent/18f-prototype/blob/master/docs/scrum/sprint0/Product%20Vision.docx>

### Customer

### Project Vision

#### Scope Definition/High-level needs

#### Measure of success/Acceptance Criteria

#### Assumptions

#### Constraints (from customer)

#### Risks

#### High-level Technical Approach

### Schedule Summary

#### Period Of Performance

Period of Performance is from Wednesday June 17th, 2015 to Friday June 26th, 2015

#### Milestones & Releases

* Initial mock up design
* Initial working prototype on AWS
* Final delivery

### Deliverables

The following lists are the services, products, documentation, and other deliverables associated with the program:

* Working Prototype on public Site (AWS)
* Complete Agile Delivery Services RFQ Compliance Review Checklist Google form
* Attachment B
* Technical Approach hosted in GitHub
* Attachment C

# References

## Source Documentation

The following documents form the foundation for establishing this Project Management Plan. The documents are listed in order of precedence in the event there are competing statements within multiple documents.

|  |  |  |
| --- | --- | --- |
| Project Source Documents | | |
| **Work Product Identifier/Title** | **Path / Storage Location** | **Level of CM Control** |
| SOW/RFP/RFQ/Initial Requirements | <https://intranet.geocent.com/secure/businessdevelopment/SiteDirectory/gsaagile/_layouts/15/start.aspx#/> |  |
| Contract | <https://intranet.geocent.com/secure/businessdevelopment/SiteDirectory/gsaagile/_layouts/15/start.aspx#/> |  |
| Proposal | <https://intranet.geocent.com/secure/businessdevelopment/SiteDirectory/gsaagile/_layouts/15/start.aspx#/> |  |

## Project Working Documents

The following documents are used for the regular tracking and management of all aspects of this project.

|  |  |  |  |
| --- | --- | --- | --- |
| Project Internal Working Documents | | | |
| **Work Product Identifier/Title** | **Path / Storage Location** | **Level of CM Control** |
| Project Management Plan | [PMP](https://intranet.geocent.com/secure/businessdevelopment/SiteDirectory/gsaagile/Approach%20To%20Create%20the%20Prototype/Geocent%20Project%20Management%20Plan%20Template%20-%20Scrum.docx?Web=1) | Project Manager |
| Product Backlog | [Scrumdo.com Backlog](https://www.scrumdo.com/projects/project/18f-ads-prototype/iteration/128546) | Team |
| Backlog Grooming Forms |  | Scrum Master |
| Sprint Review Forms | <project>/ REQM/Reviews | Scrum Master |
| Sprint Launch Forms | <project>/PP/Launch | Scrum Master |
| Sprint Retrospective Forms | <project>/Quality/Retrospective | Scrum Master |
| Code Review Forms | <project>/Quality/Reviews | Reviewers |
| Sprint Status Reports | [Scrumdo](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary) and export to [GitHub Repo](https://github.com/Geocent/18f-prototype/tree/master/docs/scrum) | Scrum Master |
| Geocent Status Reports | [Scrumdo](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary) and export to [GitHub Repo](https://github.com/Geocent/18f-prototype/tree/master/docs/scrum) | Project Manager |
| Project Review Documents (IPR Presentations) | [Scrumdo](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary) and export to [GitHub Repo](https://github.com/Geocent/18f-prototype/tree/master/docs/scrum) | Project Manager |
| Project Burn-Down Chart | [Scrumdo](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary) and export to [GitHub Repo](https://github.com/Geocent/18f-prototype/tree/master/docs/scrum) | Scrum Master |
| Geocent Project Estimator | [Scrumdo](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary) and export to [GitHub Repo](https://github.com/Geocent/18f-prototype/tree/master/docs/scrum) | Project Manager |

# 

# Organization

## External Interfaces: Who do you interface with outside of Geocent for this project?

| **External Administrative, Managerial, and Customer Interfaces** | | |
| --- | --- | --- |
| **External Contact Info** | **Organization/Role** | **Geocent Liason** |
| GSA | Receives Deliverables | Contracts@geocent.com |
|  |  |  |
|  |  |  |
|  |  |  |

## Internal Structure: Show a model of the Geocent program management structure for this project

Figure 1 illustrates the **18f ADS Prototype** program management structure for the effort. This figure also represents the program’s relation to the Geocent organization.

Figure 1. Program Organization Chart

## Roles and Responsibilities: List the roles, responsibilities and individual within Geocent for your project.

**The following table is populated with examples;**

| **Program Roles** | | |
| --- | --- | --- |
| **Role** | **Responsibilities, Required Knowledge/Skills** | **Person** |
| **Product Manager** | * See Attachment A Labor Category Descriptions.pdf | Keith Alphonso |
| **Technical Archtiect** | Technical Lead of the Program   * See Attachment A Labor Category Descriptions.pdf | Jared Ladner |
| **Scrum Coach** | * See Attachment A Labor Category Descriptions.pdf | Roberta Hazelbaker |
| **Delivery Manager** | * See Attachment A Labor Category Descriptions.pdf | Brian Priest |
| **DevOPS Eng** | * See Attachment A Labor Category Descriptions.pdf | Tyler Sanders |
| **Interaction Designer** | * See Attachment A Labor Category Descriptions.pdf | Vance Lowe |
| **Visual Designer** | * See Attachment A Labor Category Descriptions.pdf | Josh Penton |
| **Frontend Developers** | * See Attachment A Labor Category Descriptions.pdf | Randy Nolan & Aaron Whitney |

# Managerial Plans

## Start-Up Plan

### Establish Resources

#### Staffing

See above matrix and reference Attachment A Labor Category Descriptions.pdf.

#### Training

The following training has been identified as necessary for execution of this project.

|  |  |  |
| --- | --- | --- |
| Training | Description | Formal/Informal |
| AngularJS | Training will be provided to all developer resources by the Frontend Designer | Informal |
| Docker | Training will be provided to all developer resources by the DevOps Eng | Informal |

#### Facilities

The primary work facility for this effort will be the Geocent facility in Lakefront New Orleans, Stennis Space Center, Charleston SC, Baton Rouge offices.

#### Subcontractors

N/A

#### Tools: List tools that are necessary for your project.

See README.md <https://github.com/Geocent/18f-prototype>

#### Hardware Resources

| **Product** | **Specifications** | **Qty** |
| --- | --- | --- |
| Dev Laptops | Mac Book Pro |  |
| AWS | 3 Mid Servers: CI, Dev, and Prod  ‘ads.geocent.com’  ads-dev.geocent.com  ads-ci.geocent.com | 3 |
|  |  |  |

## Execution Plan

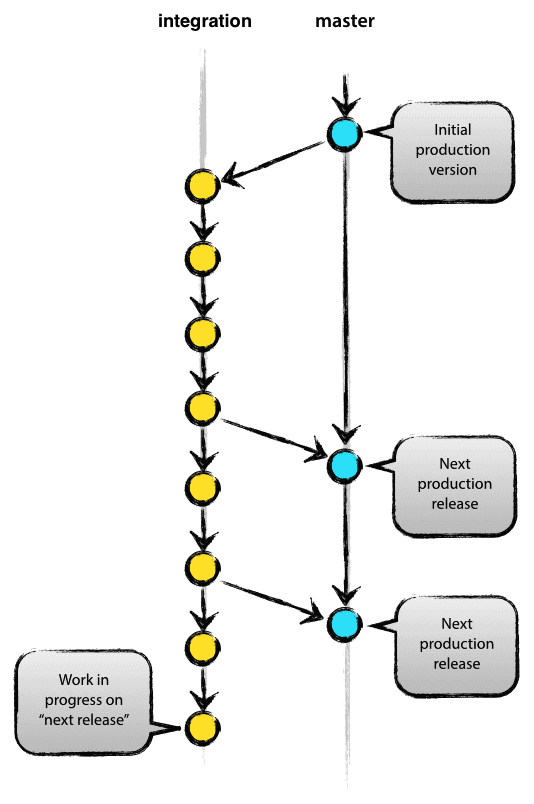
Following Scrum methodology.

### Manage Code

Code will be managed in [GitHub Repo](https://github.com/Geocent/18f-prototype).

#### Branching Strategy

**The main branches**

Central repo holds two main branches with an infinite lifetime:

* master
* integration

The master branch at origin should be familiar to every Git user. Parallel to the master branch, another branch exists called integration.

We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state.

We consider origin/integration to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the “integration branch”. This is where any automatic nightly builds are built from.

When the source code in the integration branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number. How this is done in detail will be discussed further on.

Therefore, each time when changes are merged back into master, this is a new production release by definition. We tend to be very strict at this, so that theoretically, we could use a Git hook script to automatically build and roll-out our software to our production servers everytime there was a commit on master.

**Supporting branches**

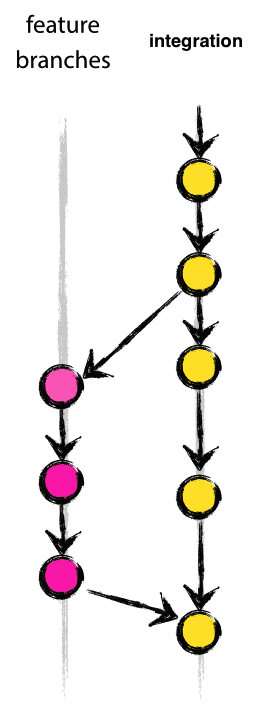
Next to the main branches master and integration, our development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

The different types of branches we may use are:

* Feature branches
* Release branches
* Hotfix branches

Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets. We will walk through them in a minute.

By no means are these branches “special” from a technical perspective. The branch types are categorized by how we use them. They are of course plain old Git branches.



**Feature branches**

May branch off from:

integration

Must merge back into:

integration

**Branch naming convention:**

anything except master, integration, release-\*, or hotfix-\*

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist in developer repos only, not in origin but can if need to share and collaborate on a feature.

**Creating a feature branch**

When starting work on a new feature, branch off from the integration branch.

$ git checkout -b myfeature integration

Switched to a new branch "myfeature"

**Incorporating a finished feature on integration**

Finished features may be merged into the develop branch definitely add them to the upcoming release:

$ git checkout integration

Switched to branch 'integration'

$ git merge --no-ff myfeature

Updating ea1b82a..05e9557

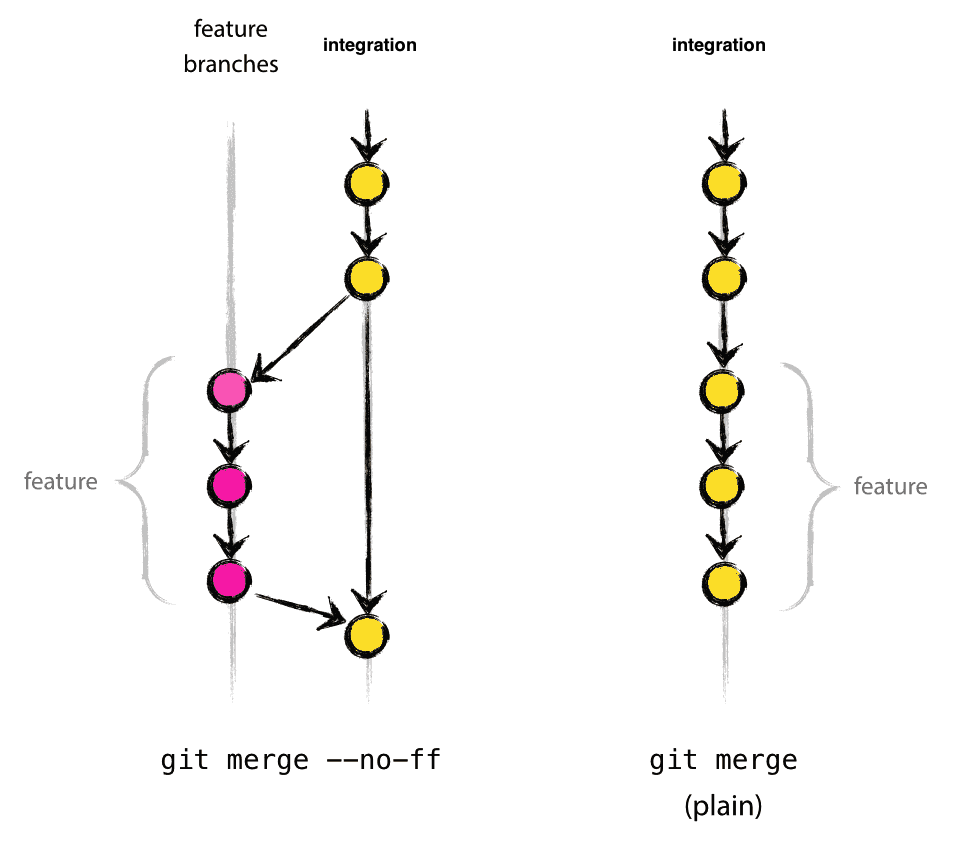
(Summary of changes)

$ git branch -d myfeature

Deleted branch myfeature (was 05e9557).

$ git push origin integration

The --no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:



In the latter case, it is impossible to see from the Git history which of the commit objects together have implemented a feature—you would have to manually read all the log messages. Reverting a whole feature (i.e. a group of commits), is a true headache in the latter situation, whereas it is easily done if the --no-ff flag was used.

Yes, it will create a few more (empty) commit objects, but the gain is much bigger that that cost.

**Release branches**

May branch off from:

integration

Must merge back into:

integration and master

Branch naming convention:

release-\*

Release branches support preparation of a new production release. They allow for last-minute dotting of i’s and crossing t’s. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the integration branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from integration is when integration (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to integration at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the integration branch reflected changes for the “next release”, but it is unclear whether that “next release” will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project’s rules on version number bumping.

**Creating a release branch**

Release branches are created from the integration branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of integration is ready for the “next release” and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

$ git checkout -b release-1.2 integration

Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2

Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"

[release-1.2 74d9424] Bumped version number to 1.2

1 files changed, 1 insertions(+), 1 deletions(-)

After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the integration branch). Adding large new features here is strictly prohibited. They must be merged into integration, and therefore, wait for the next big release.

**Finishing a release branch**

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into integration, so that future releases also contain these bug fixes.

The first two steps in Git:

$ git checkout master

Switched to branch 'master'

$ git merge --no-ff release-1.2

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2

The release is now done, and tagged for future reference.

To keep the changes made in the release branch, we need to merge those back into integration, though. In Git:

$ git checkout integration

Switched to branch 'integration'

$ git merge --no-ff release-1.2

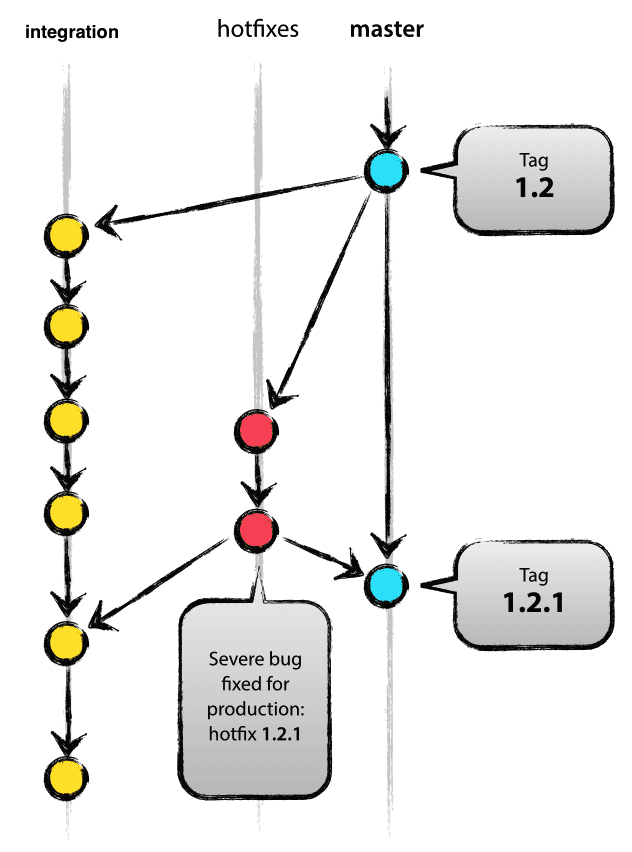
Merge made by recursive.

(Summary of changes)

Now we are really done and the release branch may be removed, since we don’t need it anymore:

$ git branch -d release-1.2

Deleted branch release-1.2 (was ff452fe).

**Hotfix branches**

May branch off from:

master

Must merge back into:

integration and master

Branch naming convention:

hotfix-\*

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the integration branch) can continue, while another person is preparing a quick production fix.

**Creating the hotfix branch**

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on integration are yet unstable. We may then branch off a hotfix branch and start fixing the problem:

$ git checkout -b hotfix-1.2.1 master

Switched to a new branch "hotfix-1.2.1"

$ ./bump-version.sh 1.2.1

Files modified successfully, version bumped to 1.2.1.

$ git commit -a -m "Bumped version number to 1.2.1"

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

1 files changed, 1 insertions(+), 1 deletions(-)

Don’t forget to bump the version number after branching off!

Then, fix the bug and commit the fix in one or more separate commits.

$ git commit -m "Fixed severe production problem"

[hotfix-1.2.1 abbe5d6] Fixed severe production problem

5 files changed, 32 insertions(+), 17 deletions(-)

**Finishing a hotfix branch**

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into integration, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

$ git checkout master

Switched to branch 'master'

$ git merge --no-ff hotfix-1.2.1

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2.1

Next, include the bugfix in integration, too:

$ git checkout integration

Switched to branch 'integration '

$ git merge --no-ff hotfix-1.2.1

Merge made by recursive.

(Summary of changes)

The one exception to the rule here is that, when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of integration. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into integration too, when the release branch is finished. (If work in integration immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into integration now already as well.)

Finally, remove the temporary branch:

$ git branch -d hotfix-1.2.1

Deleted branch hotfix-1.2.1 (was abbe5d6).

### Manage Requirements

The following practices and tools are used for managing requirements.

|  |  |  |  |
| --- | --- | --- | --- |
| Requirements Management Method/Tool | Procedure | Evidence | Description |
| Initial Requirements | Proposal Process | Proposal | Initial requirements and understanding are provided in the proposal and accepted with award. |
| Requirements Understanding | Backlog Grooming | Grooming Report | Requirements are analyzed, decomposed, clarified, prioritized, and documented. |
| Requirement Commitment | Sprint Planning | Sprint Planning Report | During sprint planning the customer commits to a specific set of user stories. |
| Requirements Change Management | Backlog Grooming | Grooming Report | Requirements are analyzed, decomposed, clarified, prioritized, and documented. |
| Requirements Tracability | Backlog Grooming | Tool which links features to user stories to tasks to sprints to code versions. |  |
| Requirements implementation consistency | Sprint Review | Review form | User accepts stories implemented in sprint |

### Manage Communications

The following communications mechanisms are used to ensure stakeholder and management effective communications.

#### Reporting Mechanisms

| **Status and Reporting Plan** | | |
| --- | --- | --- |
| **Report Name** | **Performed By Whom** | **Frequency** |
| Geocent Status Report | Project Manager | N/A |
| Sprint Status Report | Scrum Master | end of each sprint |
| Required Contract Deliverable Reports | Project Manager | end of proposal |
|  |  |  |

#### Required Meetings

| **Meeting & Interface Plan** | | |
| --- | --- | --- |
| **Meeting Name & Purpose** | **Attendees** | **Frequency** |
| Sprint planning | Team | 2 days |
| Sprint Review | Team | 2 days |
| Backlog Grooming | Customer  Leads | As needed |
| In Progress Review | Project Leads  Geocent Leadership | N/A |
| Daily Scrum Standup | Team | Daily |

### Manage Quality

Geocent will implement Quality Assurance in order to assure it meets both internal standards and contractual obligations. QA activities will take place according to Delivery Manager Quality Plan.

### Manage Measurements (Metrics)

| **Program Measurements and Metrics Plan** | | | | |
| --- | --- | --- | --- | --- |
| **Measure** | **Frequency Collected** | **Collected by Whom** | **Analyzed by Whom** | **Used by Whom** |
| Schedule Adherence | daily | PM | Senior Management | Geocent |
| Budget Adherence | daily | PM | Senior Management | Geocent |
| Funding | daily | PM | Senior Management | Geocent |
|  |  |  |  |  |

The project measures will be collected, analyzed, reported, and used according to the plan above.

### Manage Issues

The project will use [Scrumdo.com](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary) to track all bugs, issues and change requests that are not UX related. Issues can be resolved by one of the following:

1. Fix the issue
2. Modify the requirement
3. Obtain a waiver on the issue

Resolving issues may involve negotiation between one or more parties. If the parties are unable to agree who is responsible for resolving the issue, the issue must be escalated to higher levels of project leadership.

User Experience issues will be recorded and maintained in [GitHub Repo](https://github.com/Geocent/18f-prototype).

1. Issues found during UX meetings will be recorded in [GitHub Issue Tracker](https://github.com/Geocent/18f-prototype/issues)
   1. Issue will include: name of user, related UX session, and change request
2. If approved by the Product Manager, Story/Stories will be created in [Scrumdo.com](https://www.scrumdo.com/projects/project/18f-ads-prototype/summary)
3. Status will be updated in GitHub to reflect if it will be worked, rejected, working, completed

**Monitor Project Risks**

Project risks shall be documented in the project status report and updated periodically as defined by the PMP (same period as status report). These risks will be captured during regular project meetings as defined by the PMP.

**Manage Corrective Action to Closure**

Issues are identified and analyzed during periodic meetings as a regular agenda item (as defined by the PMP).

If an issue is deemed to require a corrective action, documented as part of the meeting report (as defined by the PMP) and reported in regular status reports until resolved.

### Manage Contract Changes

Geocent will address schedule and requirements changes on a case-by-case basis, providing impact assessment to customer and revising this PMP accordingly.

Customer requested changes to the contract, or changes due to program internal re-planning activities will not be performed without agreement by the Project Engineer, Program Manager, and Contracts.

The incorporation of these authorized changes will be made in a timely manner and strictly controlled. Traceability to the original baseline budget will be maintained in order to provide a basis against which program growth can be measured.

## Work Plan

The WBS, work activities, schedule, resources, and budget details of the project, are all contained in the MS-Project file. (reference PM\_07 WBS Procedure)

## Closeout Plan

| **Program Closeout Plan** | | |
| --- | --- | --- |
| **Activities** | **Performed By Whom** | **Comments** |
| Identify and determine disposition of outstanding work | PM |  |
| Prepare next release, operations, maintenance or transition plan | NA |  |
| Close all logs and accounting records | PM/Contracts |  |
| Conduct post-project review and document lessons-learned | NA |  |
| Complete project team post-assignment evaluations and release team members to next assignment | Senior Management |  |
| Complete customer satisfaction review | NA |  |
| Analyze project metrics, product quality, and issue final project report | NA |  |
| Update estimating models and risk factor tables | NA |  |
| Archive documents, records, reports and notify appropriate personnel | CM |  |
| Prepare “Lessons Learned” Survey | NA |  |
| Prepare past performance writeup & presentation slide | PM |  |