Summary

- Given a version number MAJOR.MINOR.PATCH, increment the:
    a. MAJOR version when you make incompatible API changes,
    b. MINOR version when you add functionality in a backwards compatible manner, and
    c. PATCH version when you make backwards compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Versioning Specification

1. Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it SHOULD be precise and comprehensive.
2. A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers, and MUST NOT contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: v1.9.0 -> v1.10.0 -> v1.11.0.
3. Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications MUST be released as a new version.
4. Major version zero (v0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
5. Version v1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
6. Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
7. Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
8. Major version X (X.y.z | X > 0) MUST be incremented if any backwards incompatible changes are introduced to the public API. It MAY also include minor and patch level changes. Patch and minor version MUST be reset to 0 when major version is incremented.
9. A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might

not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: v1.0.0-alpha, v1.0.0-alpha.1, v1.0.0-0.3.7, v1.0.0-x.7.z.92, v1.0.0-x-y-z.–.

10. Build metadata MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Build metadata MUST be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85, 1.0.0+21AF26D3—-117B344092BD.
11. Precedence refers to how versions are compared to each other when ordered.
    a. Precedence MUST be calculated by separating the version into major, minor, patch and pre-release identifiers in that order (Build metadata does not figure into precedence).
    b. Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor, and patch versions are always compared numerically.
    c. Example: v1.0.0 < v2.0.0 < v2.1.0 < v2.1.1.
    d. When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version:
    e. Example: v1.0.0-alpha < v1.0.0.
    f. Precedence for two pre-release versions with the same major, minor, and patch version MUST be determined by comparing each dot separated identifier from left to right until a difference is found as follows:
        i. Identifiers consisting of only digits are compared numerically.
        ii. Identifiers with letters or hyphens are compared lexically in ASCII sort order.
        iii. Numeric identifiers always have lower precedence than non-numeric identifiers.
        iv. A larger set of pre-release fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal.
    g. Example: v1.0.0-alpha < v1.0.0-alpha.1 < v1.0.0-alpha.beta < v1.0.0-beta < v1.0.0-beta.2 < v1.0.0-beta.11 < v1.0.0-rc.1 < v1.0.0.

Github Versioning Stratergy

- [microservice_name]
    - master
    - testing
    - staging
    - master.{client_name}
        - master.salasa -> this the base branch it will be used for ci/cd pipelines.
        - We will use github tags push latest version and keep in track of previous version
            - Tags format ex: master.salasa-v1.0.0, master.salasa-v1.0.0-alpha, master.salasa-v1.0.0-beta.

- testing.{client_name}
    - This branch is used to test the integrity between features.
- staging.{client_name}
    - This branch is used to Deploy pre-release versions
- feature_name
    - example: create service mapping is one features so naming will be "create_service_mapping"
- feature_name.{client_name}
    - example: create_service_mapping.salasa

In all of these branches code be may different and tables may be different.