Development Guidelines

As an open source project, Metron welcomes contributions of all forms. The sections below will help you get started.

1. How To Contribute

We are always very happy to have contributions, whether for trivial cleanups, little additions or big new features.

If you don't know Java or Scala you can still contribute to the project. We strongly value documentation and gladly accept improvements to the documentation.

1.1 Contributing A Code Change

To submit a change for inclusion, please do the following:

- If there is not already a JIRA associated with your pull request, create it, assign it to yourself, and start progress
- If there is a JIRA already created for your change then assign it to yourself and start progress. You must
- If you don't have access to JIRA or can't assign an issue to yourself, please message dev@metron.apache.org and someone will either give you permission or assign a JIRA to you
- If you are introducing a completely new feature or API it is a good idea to start a discussion and get consensus on the basic design first. Larger changes should be discussed on the dev boards before submission.
- New features and significant bug fixes should be documented in the JIRA. Appropriate architecture diagrams must be created in https://www.draw.io and committed to source control as per section 2.4. Diagrams may be requested of PR submitters during review either as documentation or as an aid to the reviewer. Major features may also require a vote.
- Note that if the change is related to user-facing protocols / interface / configs, etc, you need to make the corresponding change on the documentation as well.
- Also, please indicate impacts to the following (if exist):
    - System Configuration Changes
        - Metron Configuration
        - Metron Component Configuration (sensors, etc)
        - Tech Stack Configuration (Storm, Hbase, etc)
    - Environmental Changes
        - Helper Shell Scripts
        - RPM Packaging
        - Ansible, Ambari, AWS, Docker
    - Documentation Impacts
        - Changes to Wiki Documentation
        - Revisions in Tutorials
        - Developer Guide
        - Expansions in readme's
    - Changes to System Interfaces
        - Stellar Shell
        - REST APIs
        - Etc...
- Craft a pull request following the guidelines in Section 2 of this document. Instructions on how to submit the pull request can be found here.
- Pull requests should be small to facilitate easier review. Studies have shown that review quality falls off as patch size grows. Sometimes this will result in many small PRs to land a single large feature.
- People will review and comment on your pull request. It is our job to follow up on pull requests in a timely fashion.
- Once the pull request is merged the person doing the merge (committer) should manually close the corresponding JIRA.

1.2 Reviewing and merging patches

Everyone is encouraged to review open pull requests. We only ask that you try and think carefully, ask questions and are excellent to one another. Code review is our opportunity to share knowledge, design ideas and make friends. The instructions on how to checkout a patch for review can be found here.

When reviewing a patch try to keep each of these concepts in mind:

- Is the proposed change being made in the correct place? Is it a fix in a backend when it should be in the primitives? In Kafka vs Storm?
- What is the change being proposed? Is it based on Community recognized issues?
- Do we want this feature or is the bug they're fixing really a bug?
- Does the change do what the author claims?
- Are there sufficient tests?
- Has it been documented?
- Will this change introduce new bugs?

Also, please review if the submitter correctly flagged impacts to the following (if exist):

- System Configuration Changes
    - Metron Configuration
    - Metron Component Configuration (sensors, etc)

    - Tech Stack Configuration (Storm, Hbase, etc)
- Environmental Changes
    - Helper Shell Scripts
    - RPM Packaging
    - Ansible, Ambari, AWS, Docker
- Documentation Impacts
    - Changes to Wiki Documentation
    - Revisions in Tutorials
    - Developer Guide
    - Expansions in readme's
- Changes to System Interfaces
    - Stellar Shell
    - REST APIs
    - Etc...

2. Implementation

2.1 Grammar and style

These are small things that are not caught by the automated style checkers.

- Does a variable need a better name?
- Should this be a keyword argument?
- In a PR, maintain the existing style of the file.
- Don’t combine code changes with lots of edits of whitespace or comments; it makes code review too difficult. It’s okay to fix an occasional comment or indenting, but if wholesale comment or whitespace changes are needed, make them a separate PR.
- Use the checkstyle plugin in Maven to verify that your PR conforms to our style

2.2 Code Style

- Follow the Google Java Style Guide outlined here: https://google.github.io/styleguide/javaguide.html

2.2.1 IntelliJ IDEA Checkstyle Warnings

- Install the Checkstyle-IDEA plugin (Available in IntelliJ's directly from Preferences -> Plugins. Select "Browse Repositories".)
- Go into Preferences -> Settings -> Other Settings -> Checkstyle
- Change "Checkstyle version" to 8.0.
- Apply. Otherwise the new file won't match the version and an error will be thrown.
- Add a new Checkstyle file
    - Use the checkstyle.xml file included in the root directory of the project.
    - If you have errors, you most likely need to make sure you're set to the right version and have applied it.
- Select the checkbox for the new style
- Apply

New, Checkstyle-based, warnings should show up in Java files, e.g.

```
Checkstyle: WhitespaceAround: 'if' is not followed by whitespace. Empty blocks may only be represented as {} when
not part of a multi-block statement
```

2.2.2 IntelliJ IDEA Code Formatting

- Download https://github.com/google/styleguide/blob/gh-pages/intellij-java-google-style.xml
- In IDEA, go to Preferences -> Editor -> Code Style.
- Click in the Settings gear next to the Scheme selection box. Go to Import Scheme -> IntelliJ IDEA Code Style XML
- Choose the previously downloaded XML file.
- It's also possible to import the Checkstyle file directly (instead of Import Scheme -> IntelliJ IDEA Code Style XML use Import Scheme -> Checkstyle Configuration). In practice, the Google IntelliJ IDEA settings appear to be more complete.

2.2.3. Eclipse Checkstyle Warnings

- Install Checkstyle from Eclipse Marketplace
- Follow the instructions for importing a custom checkstyle at http://eclipse-cs.sourceforge.net/#!/custom-config

2.2.4 Eclipse Code Formatting

- Download https://github.com/google/styleguide/blob/gh-pages/eclipse-java-google-style.xml
- In Eclipse, go to Window/Preferences and select Java/Code Style/Formatter
- Select Import and import the downloaded file.

2.3 Coding Standards

- Implementation matches what the documentation says
- Logger name is effectively the result of Class.getName()
- Class & member access - as restricted as it can be (subject to testing requirements)
- Appropriate NullPointerException and IllegalArgumentException argument checks
- Asserts - verify they should always be true
- Look for accidental propagation of exceptions
- Look for unanticipated runtime exceptions
- Try-finally used as necessary to restore consistent state
- Logging levels conform to Log4j levels
- Possible deadlocks - look for inconsistent locking order
- Race conditions - look for missing or inadequate synchronization
- Consistent synchronization - always locking the same object(s)
- Look for synchronization or documentation saying there's no synchronization
- Look for possible performance problems
- Look at boundary conditions for problems
- Configuration entries are retrieved/set via setter/getter methods
- Implementation details do NOT leak into interfaces
- Variables and arguments should be interfaces where possible
- If equals is overridden then hashCode is overridden (and vice versa)
- Objects are checked (instanceof) for appropriate type before casting (use generics if possible)
- Public API changes have been publicly discussed
- Use of static member variables should be used with caution especially in Map/reduce tasks due to the JVM reuse feature

2.4 Documentation

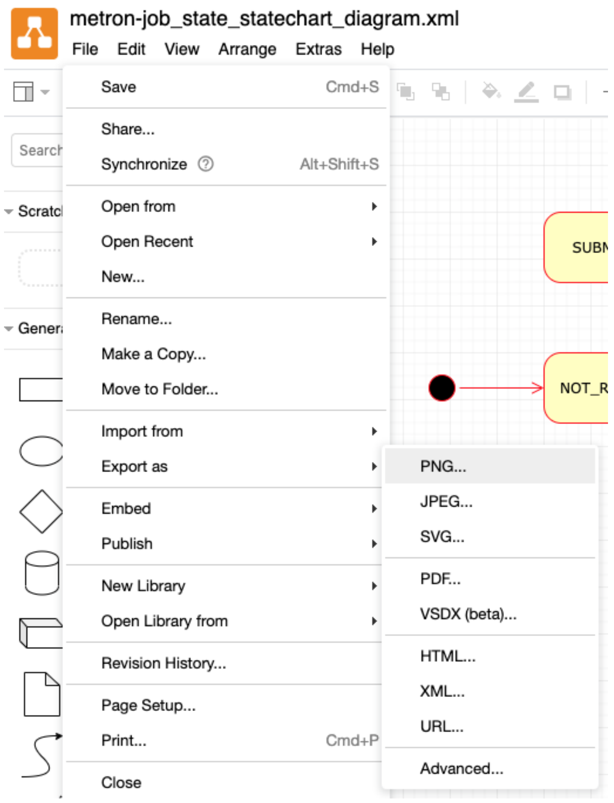
- **Code-Level Documentation**
  - Self-documenting code (variable, method, class) has a clear semantic name
  - Accurate, sufficient for developers to code against
  - Follows standard Javadoc conventions
  - Loggers and logging levels covered if they do not follow our conventions (see below)
  - System properties, configuration options, and resources covered
  - Illegal arguments are properly documented as appropriate
  - Package and overview Javadoc are updated as appropriate
  - Javadoc comments are mandatory for all public APIs
  - Generate Javadocs for release builds
- **Feature-level documentation** - should be version controlled in github in README files.
  - Accurate description of the feature
  - Sample configuration and deployment options
  - Sample usage scenarios
- **High-Level Design documentation** - architecture description and diagrams should be a part of a wiki entry.
  - Provide diagrams/charts where appropriate. Visuals are always welcome
  - Provide purpose of the feature/module and why it exists within the project
  - Describe system flows through the feature/module where appropriate
  - Describe how the feature/module interfaces with the rest of the system
  - Describe appropriate usages scenarios and use cases
- **Tutorials** - system-level tutorials and use cases should also be kept as wiki entries.
  - Add to the Metron reference application documentation for each additional major feature
  - If appropriate, publish a tutorials blog on the Wiki to highlight usage scenarios and apply them to the real world use cases
- **Diagrams**
  - We save architecture diagram source files in an xml format rendered by draw.io (instructions below). This is the free tool of choice that we've agreed to use for exchanging diagrams and their source files in Metron.
  - Image and diagram source files belong in "`<module-name>/images-source`" and rendered diagrams and images belong in "`<module-name>/images`."

2.4.1 Creating and Modifying Diagrams Using draw.io

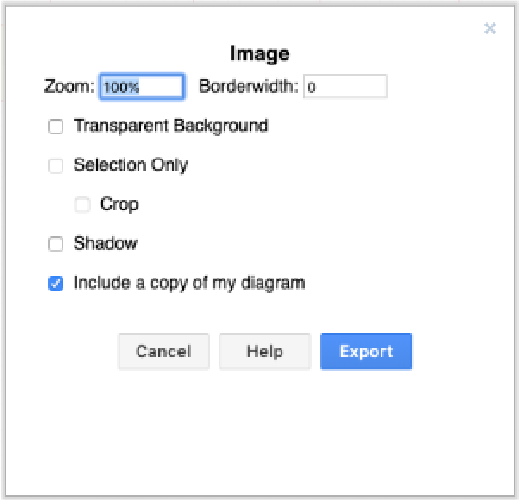
All steps expect the source and derived image files will be committed to source control via Git.

Creating and Exporting Diagrams as Images

1. Open your diagram in draw.io
2. Select "File Export as PNG..." (you can use other formats as well, e.g. SVG)



3.
4. If using PNG, you can leave the defaults and click "Export." If you want the background transparent, select that option before exporting.

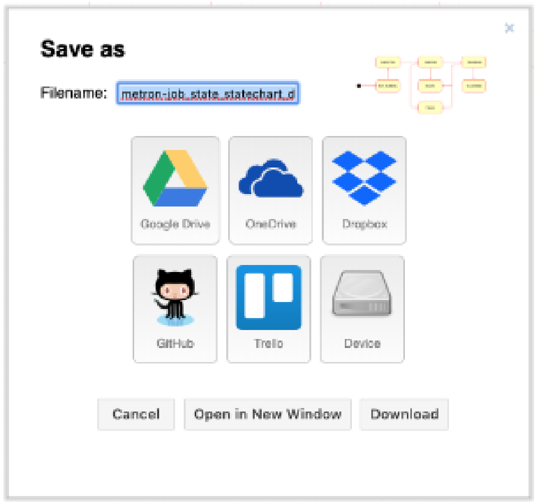


Image

Zoom: 100% Borderwidth: 0

- [ ] Transparent Background
- [ ] Selection Only
  - [ ] Crop
- [ ] Shadow
- [x] Include a copy of my diagram

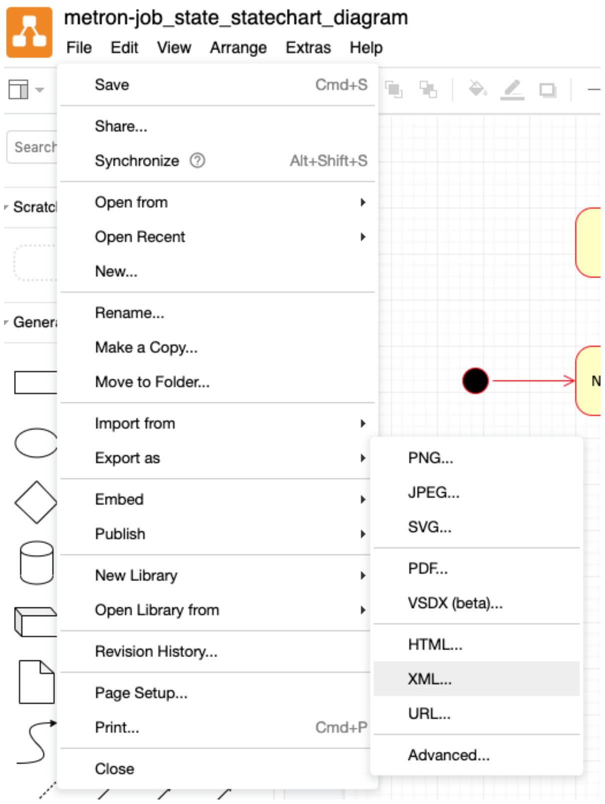
5.
6. Name your file accordingly and choose "Download."



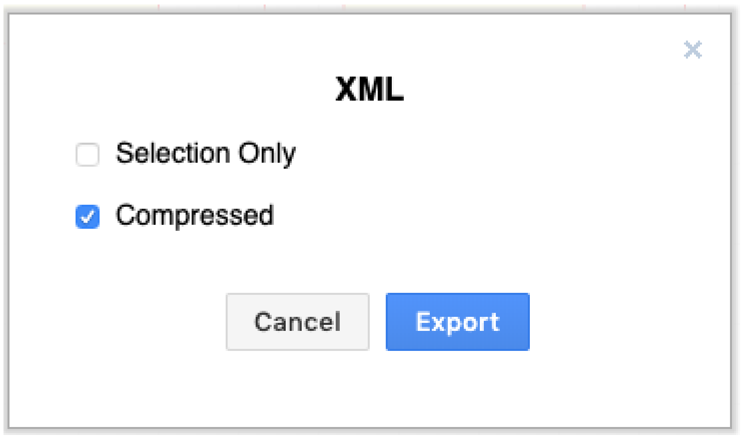
7.
8. Place the downloaded file in "`<module-name>/images`" as noted in the "Diagrams" section under 2.4.0 above.

Exporting Source Files for Diagrams

1. Open your diagram in draw.io
2. Select "File Export as XML..."

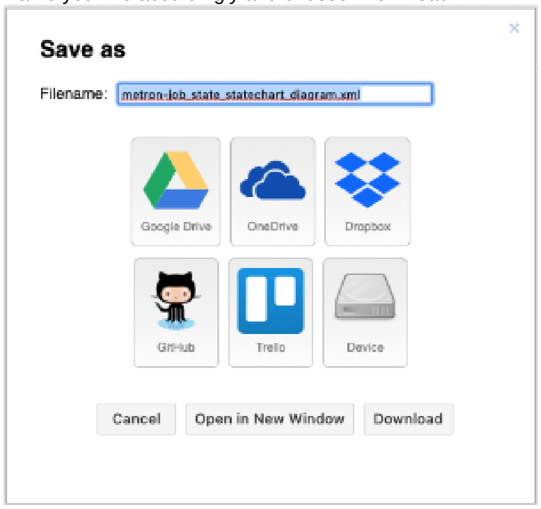


3.
4. Enable the checkbox for "Compressed" and click "Export"



[ ] Selection Only
[✓] Compressed

5.
6. Name your file accordingly and choose "Download."



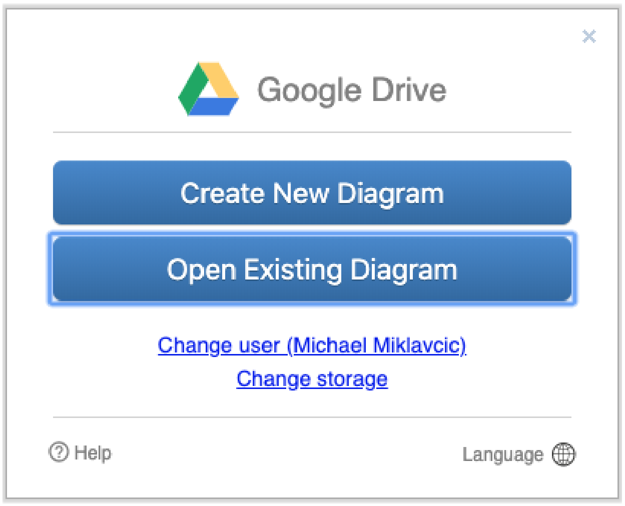
Filename: metron-job_state_statechart_diagram.xml

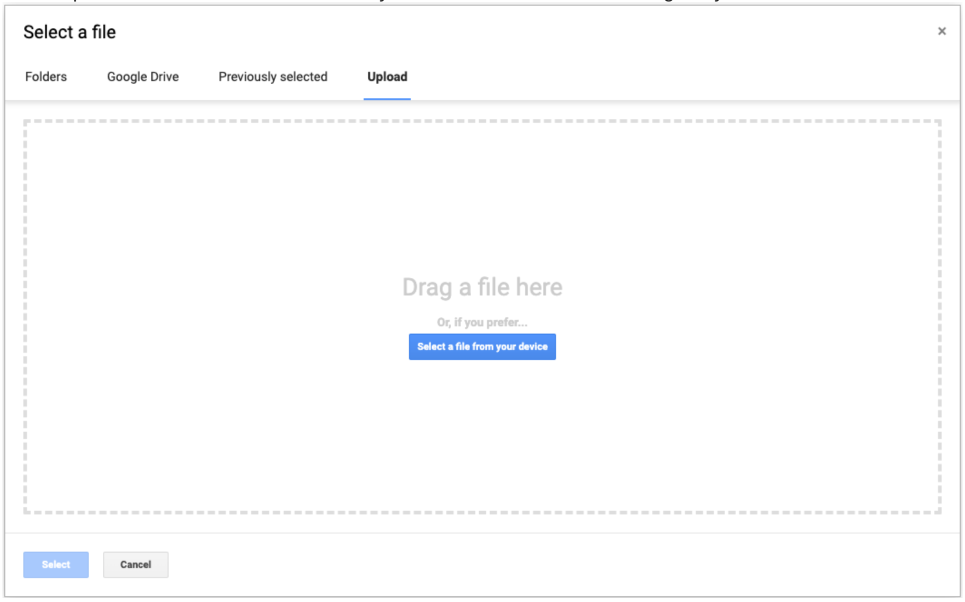
7.
8. Open the file you downloaded. Remove the xml header (`<?xml version="1.0" encoding="UTF-8"?>`), and add the following license header at the top of the file.
9.
```
<!--
  Licensed to the Apache Software
       Foundation (ASF) under one or more contributor license agreements. See the
       NOTICE file distributed with this work for additional information regarding
       copyright ownership. The ASF licenses this file to You under the Apache License,
       Version 2.0 (the "License"); you may not use this file except in compliance
       with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0
       Unless required by applicable law or agreed to in writing, software distributed
       under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES
       OR CONDITIONS OF ANY KIND, either express or implied. See the License for
  the specific language governing permissions and limitations under the License.
  -->
<!-- This is a draw.io diagram source file.  You can edit it on http://www.draw.io -->
```

10. Save the changes and place them in the relevant Maven project module under "`<module-name>/images-source`" as noted in the "Diagrams" section under 2.4.0 above.

Modifying Existing Diagrams

1. Navigate to draw.io
2. Select "Open Existing Diagram" from the prompt.

3. 

4. Click "Upload" and click "Select a file from your device" and choose the diagram you want to edit.
5. 

6. The file should be open for editing now.
7. Make any desired changes and export the source XML files and new images as described above.

2.5 Tests

- **Unit tests exist for bug fixes and new features, or a rationale is given in JIRA for why there is no test**
- Unit tests do not write any temporary files to /tmp (instead, the tests should write to the location specified by the test.build.data system property)

2.6 Merge requirements

Because Metron is so complex, and the implications of getting it wrong so devastating, Metron has a strict merge policy for committers:

- Patches must *never* be pushed directly to master, all changes (even the most trivial typo fixes!) must be submitted as a pull request.
- A committer may merge their own pull request, but only after a second reviewer has given it a +1. A qualified reviewer is a Metron committer or PPMC member.
- A non-committer may ask the reviewer to merge their pull request or alternatively post to the Metron dev board to get another committer to merge the PR if the reviewer is not available.
- There should be at least one independent party besides the committer that have reviewed the patch before merge.
- A patch that breaks tests, or introduces regressions by changing or removing existing tests should not be merged. Tests must always be passing on master. This implies that the tests have been run.
- All pull request submitters must link to travis-ci
- If somehow the tests get into a failing state on master (such as by a backwards incompatible release of a dependency) no pull requests may be merged until this is rectified.
- All merged patches will be reviewed with the expectation that thorough automated tests shall be provided and are consistent with project testing methodology and practices, and cover the appropriate cases (see reviewers guide)

The purpose of these policies is to minimize the chances we merge a change that has unintended consequences.

2.6.1 Inactive Pull Requests

Contributions can often take a significant amount of time to complete the code review process. This process requires active participation from the contributor. If the contributor is unable to actively participate, the pull request is unlikely to successfully complete this process.

Pull Requests that have failed to receive active participation from the contributor for an extended period of time risk being abandoned. Any committer can submit a request for Apache Infra to close a pull request that has been abandoned according to the following guidelines.

- A pull request is 'inactive' if no comments or updates have been made by the contributor in the previous 4 weeks.
- For any 'inactive' pull request, a committer can request from the contributor justification for keeping the pull request open.
- The committer's request should be made as a public comment on the pull request. The committer should refer the contributor to these development guidelines for inactive pull requests.
- If the contributor publically responds to the request, the pull request is no longer consider 'inactive'.
- If the contributor does not respond to the request within 2 weeks, the pull request is considered 'abandoned'.
- A committer can cast a -1 vote on any 'abandoned' pull request using these development guidelines as justification.
- A committer can submit a request to Apache Infra to close the 'abandoned' pull request based on this -1 vote.

3. JIRA

- The **Incompatible change** flag on the issue's JIRA is set appropriately for this patch
- For incompatible changes, major features/improvements, and other release notable issues, the **Release Note** field has a sufficient comment