Module Best Practices (TA1)

Michael P. Gerlek

mgerlek@inferlink.com

v1.1 / 3 Feb 2024

To integrate all the tools from the TA-1 modules into one overarching system, we need to have:

1. A better understanding of how to build, test, and run each module, plus information about what hardware resources the tool is expected to require.
2. Some standardization of how each tool is invoked and how it behaves, such as version reporting, diagnostic logging, management of scratch/temp space, etc.

This document covers both of those areas. The former area is addressed as a series of questions for the teams to answer, and the latter area is addressed by presenting a series of guidelines we’d like each team to follow. Please review these questions and guidelines, and the InferLink will schedule a series of short meetings with each of the teams to review and get consensus.

*Please note that we are not trying to control or influence what your individual applications do internally or how they are implemented; we are only interested in the “publicly exposed” parts that we will need to work with to make a dozen different tools work seamlessly with each other.*

*Further note: this document is complementary to, but ultimately different in scope and intent from, the “Python Coding Conventions” document.*

# Documentation Best Practices

1. A short description of what your module does and what major features it offers.
2. List of what the inputs are (very explicit, e.g. “a GeoTIFF image plus a JSON following this schema”)
3. List of what the output are (again, very explicit)
4. Who is the technical POC for the module?
5. Is the module one python tool or multiple tools? Is a docker container required?
6. What are the expected hardware requirements for your module (one job):
   1. Expected elapsed time (seconds)
   2. Number of CPU cores required/desired (if threaded)
   3. Expected RAM required (GB)
   4. Are one or more GPUs required? If so, is a specific GPU card required/preferred?
   5. How much disk space is required? How much of that is temp/scratch space?
7. Does your module require a database, S3 storage, or any other sorts of “external” resources?
8. What is the status of the module, with respect to an initial attempt at integration: “implemented and ready to try”, “not ready yet, implementation still underway”, “not ready at all, not yet implemented”?

# Interface Best Practices

*We expect to stand up an automated deployment pipeline that checks out each of the modules' repos, builds and tests each one, and then runs a set of whole-system tests. These guidelines are aimed at making that deployment pipeline a success.*

1. **README:** Your project’s README should contain the following information:
   1. A description of what the module does, including each "major feature" that it supports. This should just be a few sentences, for example: *"The TILER tool takes a single large input images and chops it up to produce a set of smaller tiled images. The user must specify the output tile size (in pixels). The input can be either TIFF or JPG files. The output will always be PNG files."*
   2. Instructions on how to build the module.
   3. Instructions on how to test the module.
   4. Instructions on how to run the module.
2. **Command-line tools:** Each of the module’s tools should be run via a python script, possibly dockerized.
   1. The tool must exit with a value of 0 (success) or non-zero (failure).
   2. The tool must have command-line options to support everything that needs to be controlled, including inputs, outputs, model settings, diagnostic controls, etc. (The tool may also/instead use a tool-specific config file for some of these.)
   3. The tool must provide a --version switch. To accommodate automated tooling, the tool should output the version string (and only the version string) on the first line of the output and then exit with a status of 0. (Lines after the first will be ignored.)
   4. The tool must support logging, with output going to a file named via a --log-file switch. Any output sent to stdout or stderr must also go to the log file.
   5. The tool must not require user interaction (stdin).
3. **Log Management:** As mentioned above, your tools should all support logging.
   1. Ideally, the log file location should be user-specified. If not, please document where your log files are written to, and whether they are “rolling” or cleaned up in some other way.
   2. One of the first things a command line tool should log is how it was invoked, e.g. dump the full contents of sys.argv.
   3. Your tool should also log any appropriate performance metrics, such as the elapsed (“wall clock”) time and process CPU time. The psutil python package is helpful for collecting this information.
4. **Building:**
   1. The build process should be well-documented and automated, e.g. via setup.py, a build.sh script, a Makefile, or similar. The build system must use a requirements.txt (or equivalent) file, with all dependencies pinned to specific versions.
   2. If your project requires a database, it should be setup and initialized using a SQL script.
5. **Testing:** Each module must come with some self-test mechanism for verifying it is functioning correctly, such as by running pytest or a dedicated script. The repo must include the input file(s) used by the tests and as well as the corresponding expected output file(s). Enough tests must be provided to ensure that, at least at a coarse level, each "feature" of the system is exercised and verified as running correctly.