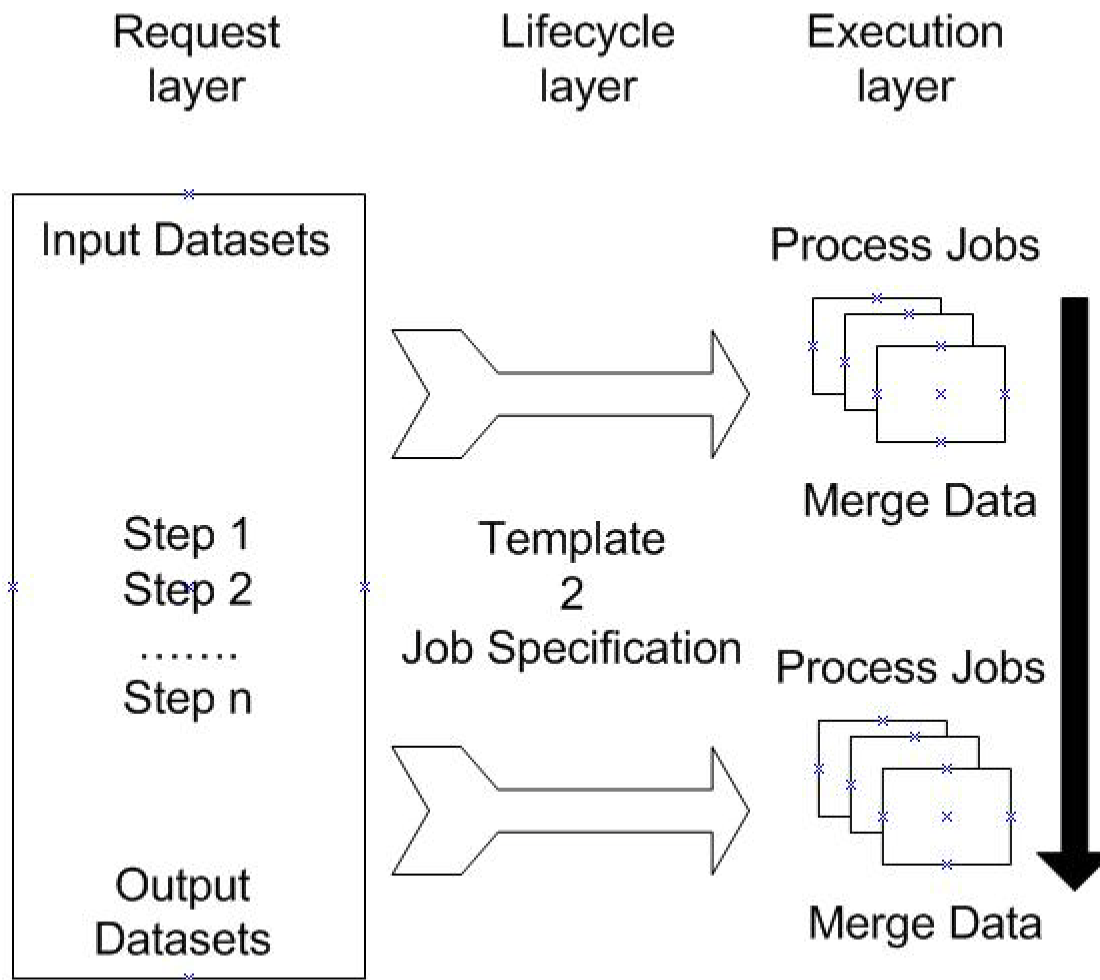
# Job Life Cycle Management Libraries for CMS Workflow Management Projects (WMCORE)

# Motivation for WMCore

- **Converge on cross project common components**
  - **Uniform usage**
  - **Lower maintance**
- **Prevent repetitive functionality implementation**
- **Address performance bottlenecks (e.g. database issues)**
- **Provide developers with sufficient tools such that they can focus on the (physics) domain specific part in their development**

# CMS Workflows: 3* layers




***Tier0 does not have a request layer**

# Job Life Cycle Management

- Different components based on WMCore represent various states of a job
  - Create, submit, track, etc…
  - Each component represents a state
- Possible that there are multiple type of jobs
  - Component need to differentiate between job types
- Components can interact with third party services
  - Site db, site submission, mass storage, etc..
- An application(e.g. CRAB, T0, Production) is a collection of components managing the life cycle
  - Not necessarily the same components

# Life cycles of job (types)

Simplified Example!! Many more states (Error, Queued, Retry…)

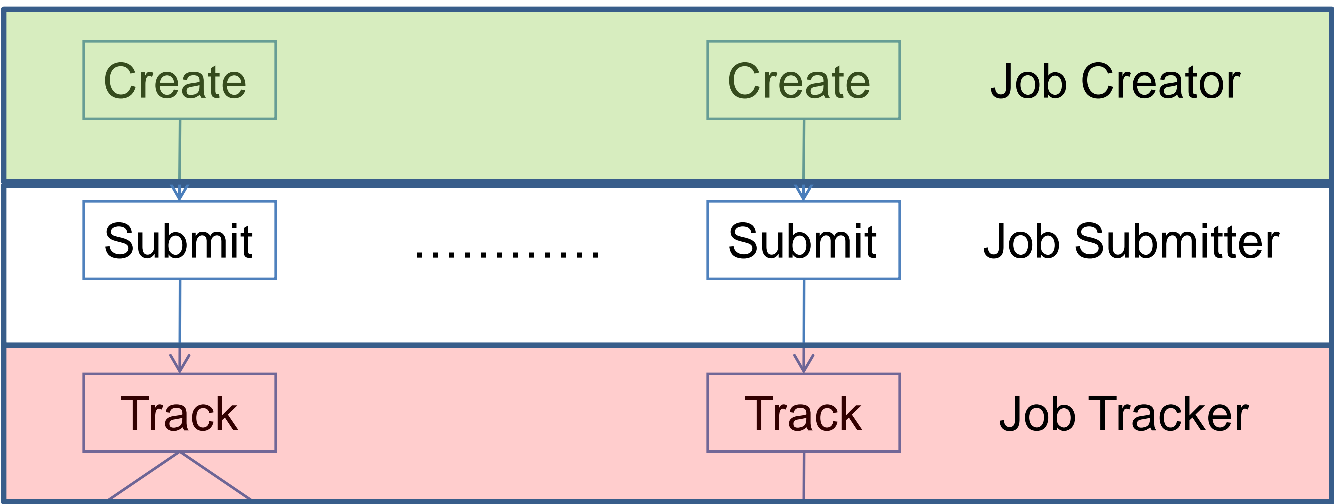
Communication through messages

Job types and their states

Components Representing state (operations)

Job Type 1 ………… Job Type n

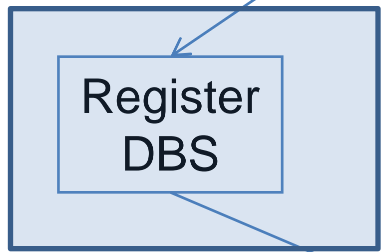
CreateJob




SubmitJob

TrackJob

JobSuccess




Register Phedex

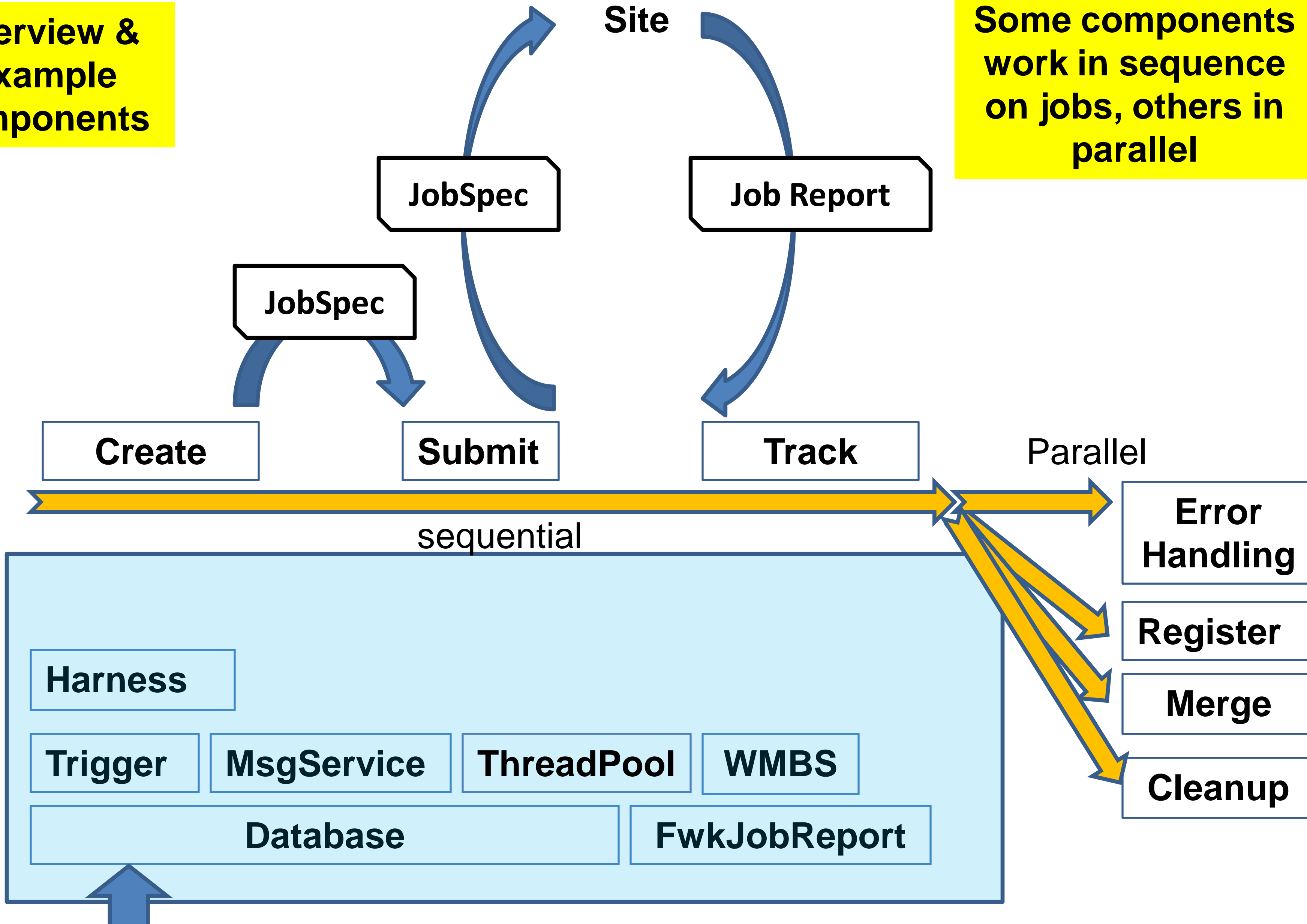
Cleanup

Cleanup

Cleanup

Synchronization between parallel states

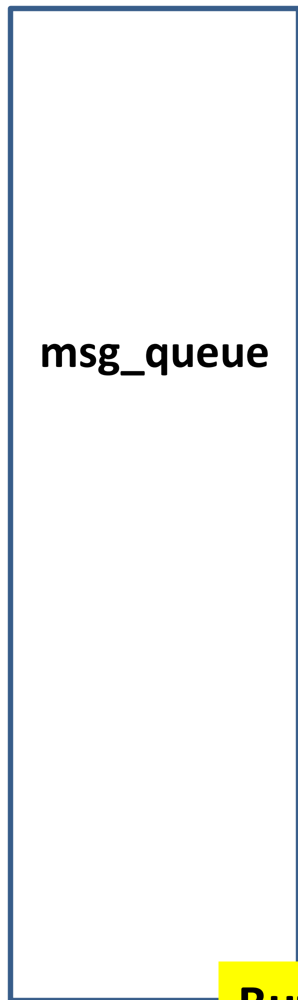
# Overview & Example components




**WMCore provides common components without being context /project specific (e.g. CRAB, T0, Production)**

# Msg Service

**Delivery of asynchronous messages**

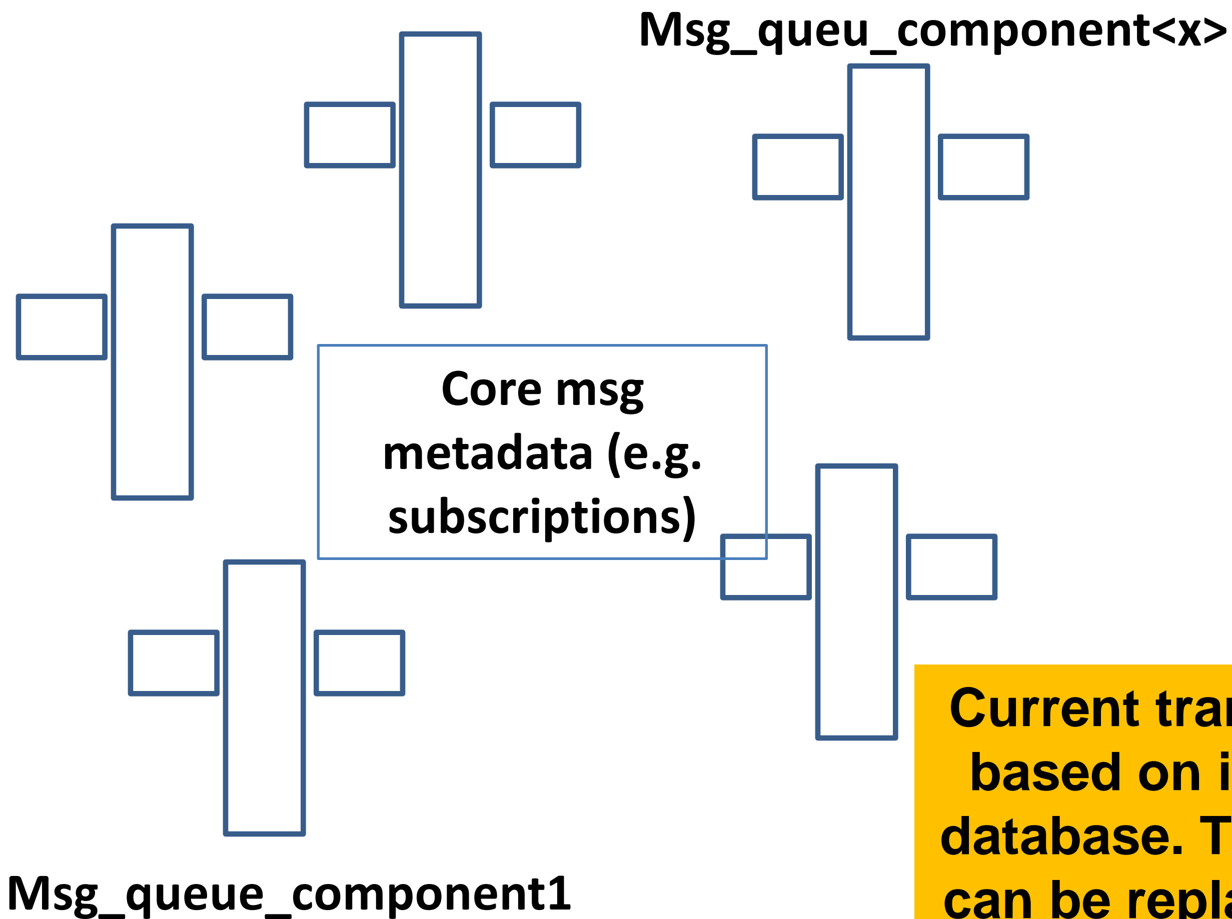


+ **Core msg metadata (e.g. subscriptions)**

**Prevent single inserts and delete from large table. Buffer tables are purged/filled when a certain size is reached.**

**Solution (or option): For each component have their own buffer_in, msg_queue, and buffer_out**

**But: Still problem when one component is 'dead' or 'stuck' and others have messages going through buffer_in →msg_queue→buffer_out. Messages dead component accumulate in msg_queue**




- Messages distributed over more tables (prevent large tables)
- Soften impact of ‘dead’ component
- Use table name pre/post fixing to prevent table name clashes.

**Current transport implementation is based on inserting a message in a database. This transport mechanism can be <u>replaced,</u> but we still can use the rest of the persistent backend (~90%) including the buffering, outlined here to store the messages and to ensure no messages are lost. An example of such a transport layer is Twisted (http://twistedmatrix.com/trac/)**

# Other Core Services/Libraries

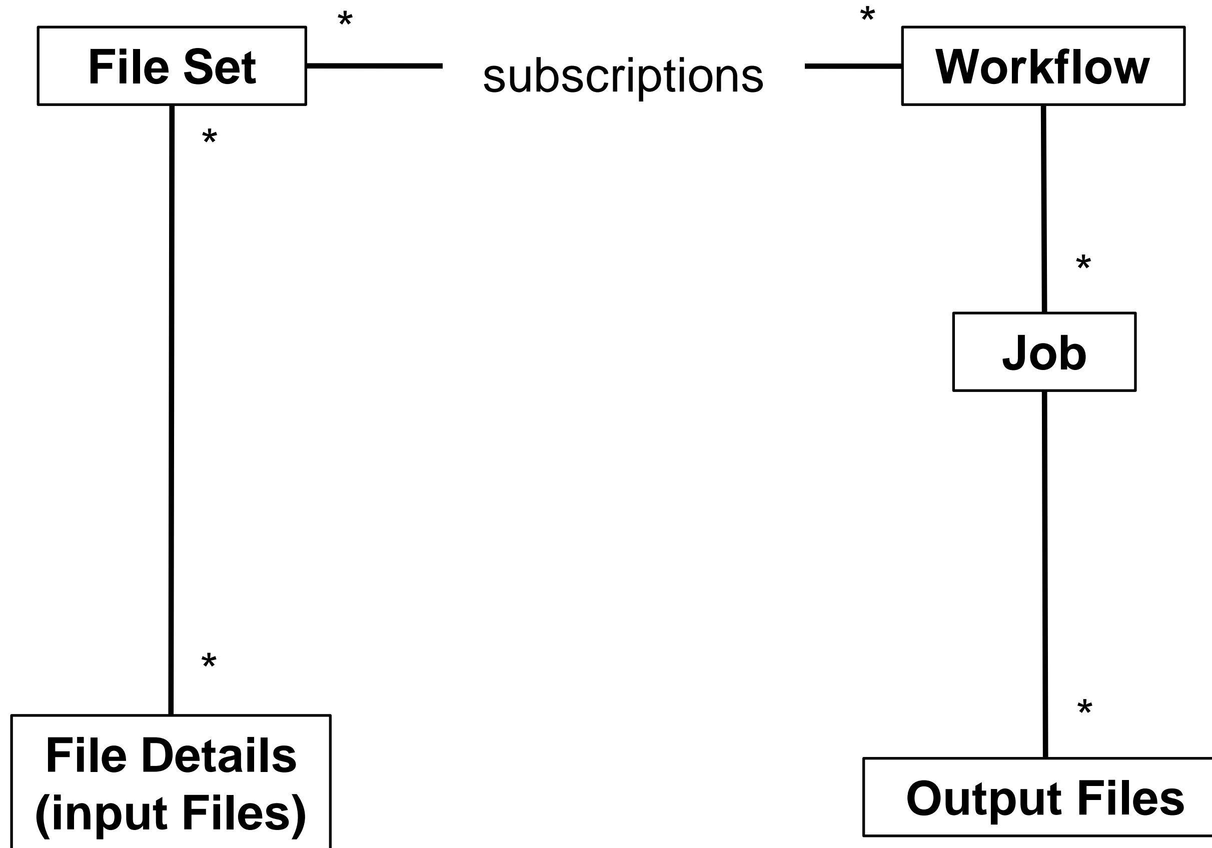
- (Persistent) Threadpool
- Worker threads
  - Long running threads within a component
- Trigger
  - Synchronization of components
- Database connection management
  - Through SQLAlchemy

# Other Core Services/Libraries

- Web development (HTTPFrontend)
  - Facilitating development of web based components based on CherryPy
- WMBS Data model
  - Managing the relation between workflow, job and data products

**Provide developers with sufficient tools such that they can focus on the (physics) domain specific part in their development**

# WMBS Data Model

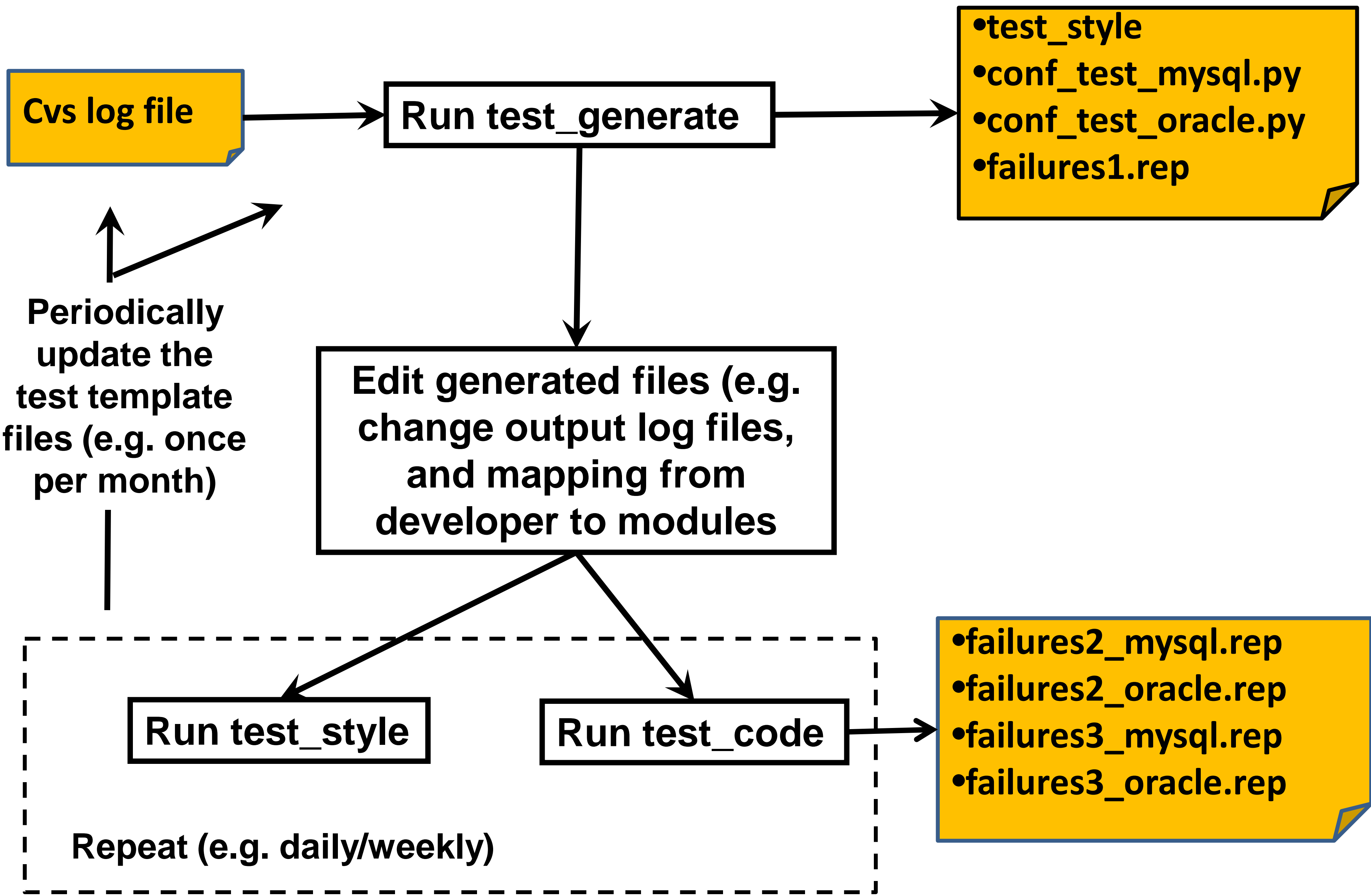


# Testing

- **WMCORE/standards/test_generate**
  - Generates templates for testing
  - Different templates for different backends (conf_test_mysql.py, conf_test_oracle.py)
  - Generates test_style for checking code style.
  - Takes as input the cvs log and maps the developers to the test or module when generating reports.

# Testing (failure levels)

- **3 levels of failure:**
  - **Level 1: failed to import the test according to the test name convention**
  - **Level 2: failed to instantiate the test object**
  - **Level 3: failures/errors during testing.**


Cvs log file
Run test_generate
•test_style
•conf_test_mysql.py
•conf_test_oracle.py
•failures1.rep
Periodically update the test template files (e.g. once per month)
Edit generated files (e.g. change output log files, and mapping from developer to modules
Run test_style
Run test_code
•failures2_mysql.rep
•failures2_oracle.rep
•failures3_mysql.rep
•failures3_oracle.rep
Repeat (e.g. daily/weekly)

# (Workflow) Code Generation

- **WMCore contains scripts that parses a (simple) Python based syntax and generates the (stub) classes for development of the components.**
  - **WMCORE/bin/wmcore-new-flow**
- **Specification based on such a syntax is called ‘flow’ as it desribes how messages are sent between components (describes the flow of the job/task**

# (Workflow) Code Generation

- Sample Specification:

Defines a Trigger for component synchronization.

```
synchronizer =
 {'ID' : 'JobPostProcess',\
  'action' : 'PA.Core.Trigger.PrepareCleanup'}
```

Defines a handler in a worklfow which acts on a messageIn messages and produces messageOut messages. Threading means handling of messages is threaded

```
handler =
  {'messageIn'   : 'SubmitJob',\
   'messageOut'  : 'TrackJob|JobSubmitFailed',\
   'component'   : 'JobSubmitter',\
   'threading'   : 'yes',\
   'createSynchronizer' : 'JobPostProcess'}
```

# (Workflow) Code Generation

(sample): Spec file

```
handler = {'messageIn'   : 'CreateJob', \
        'messageOut'  : '', \
        'threading'   : 'yes', \
        'configurable': 'yes', \
        'component'   : 'JobCreator'}

handler = {'messageIn'   : 'NewWorkflow', \
        'messageOut'  : '', \
        'component'   : 'JobCreator'}

handler = {'messageIn'   : 'JobCreator:SetCreator', \
        'messageOut'  : '', \
        'component'   : 'JobCreator'}

handler = {'messageIn'   : 'JobCreator:SetGenerator', \
        'messageOut'  : '', \
        'component'   : 'JobCreator'}
```

# (Workflow) Code Generation

**(sample): Directory Layout**

[localhost] /tmp/PRODAGENT/src/python/PA/Component/JobCreator > ls

DefaultConfig.py  Handler  __init__.py  JobCreator.py

[localhost] /tmp/PRODAGENT/src/python/PA/Component/JobCreator > ls Handler/

CreateJob.py  CreateJobSlave.py  __init__.py JobCreator_SetCreator.py  JobCreator_SetGenerator.py NewWorkflow.py

**Generates all the stub files**

# (Workflow) Code Generation

- Workflow can be visualized
  - Boxes are components
  - Arrows are messages (tail is 'from', head is 'to')