Job Life Cycle Management Libraries for CMS Workflow Management Projects

Stuart Wakefield on behalf of CMS DMWM group

Thanks to Frank van Lingen for the slides

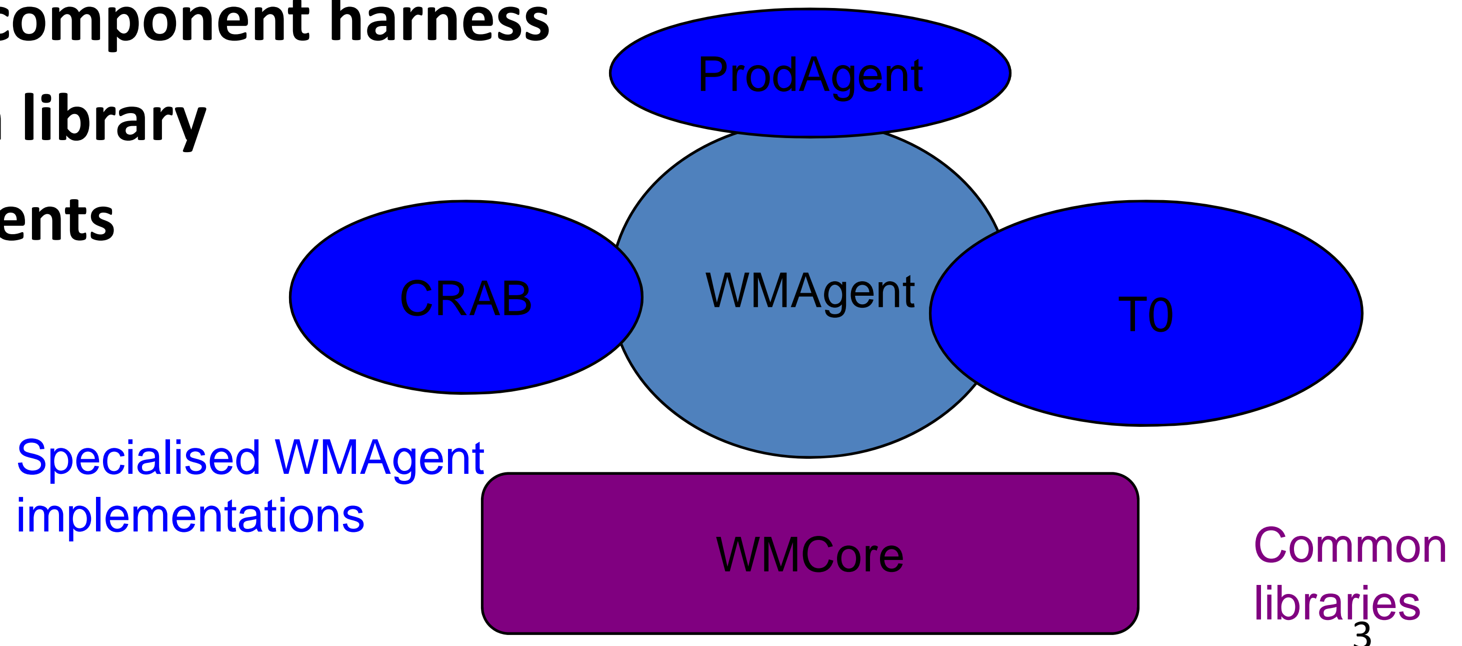
Motivation

- **Converge on cross project common components**
  - **Uniform usage**
  - **Lower maintenance**
- **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**

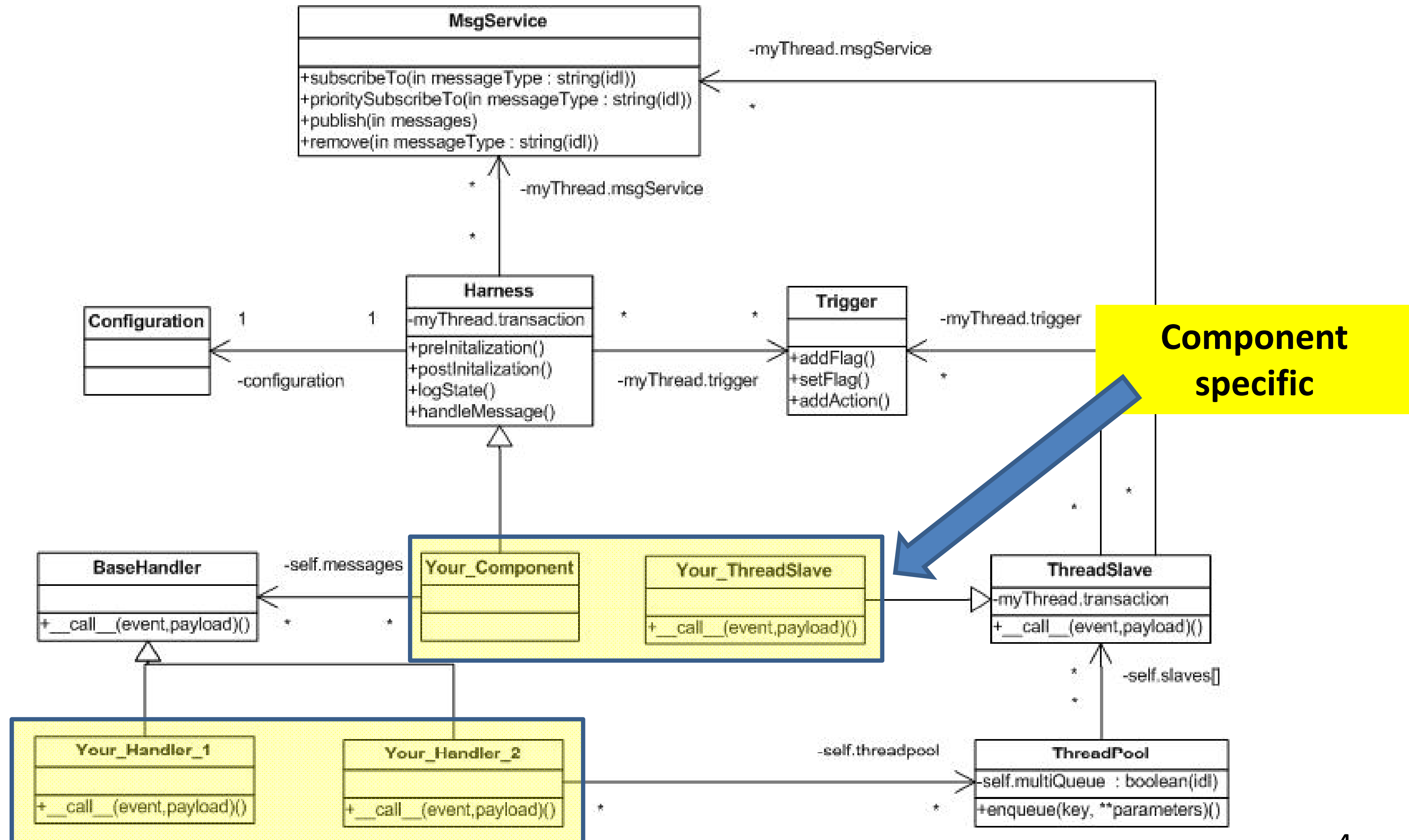
Architecture

- Common low level / API layer (WMCore)
  - Grid/Storage interaction – LCG, OSG, ARC etc.
  - CMS services – authentication, databases, site info…
- Event driven components (WMAgent)
  - Generic component harness
  - Common library

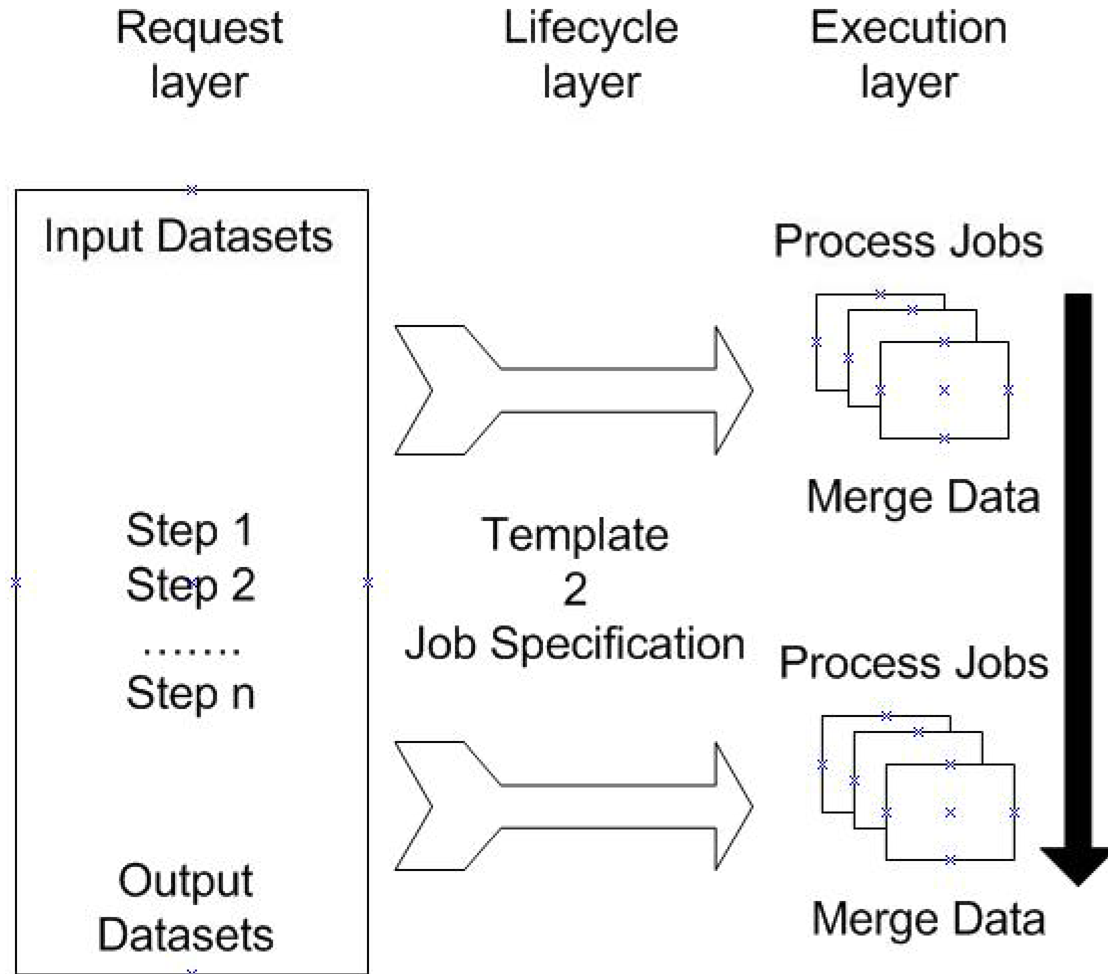
  of components



Structure of an Agent



CMS Workflows: 3* layers

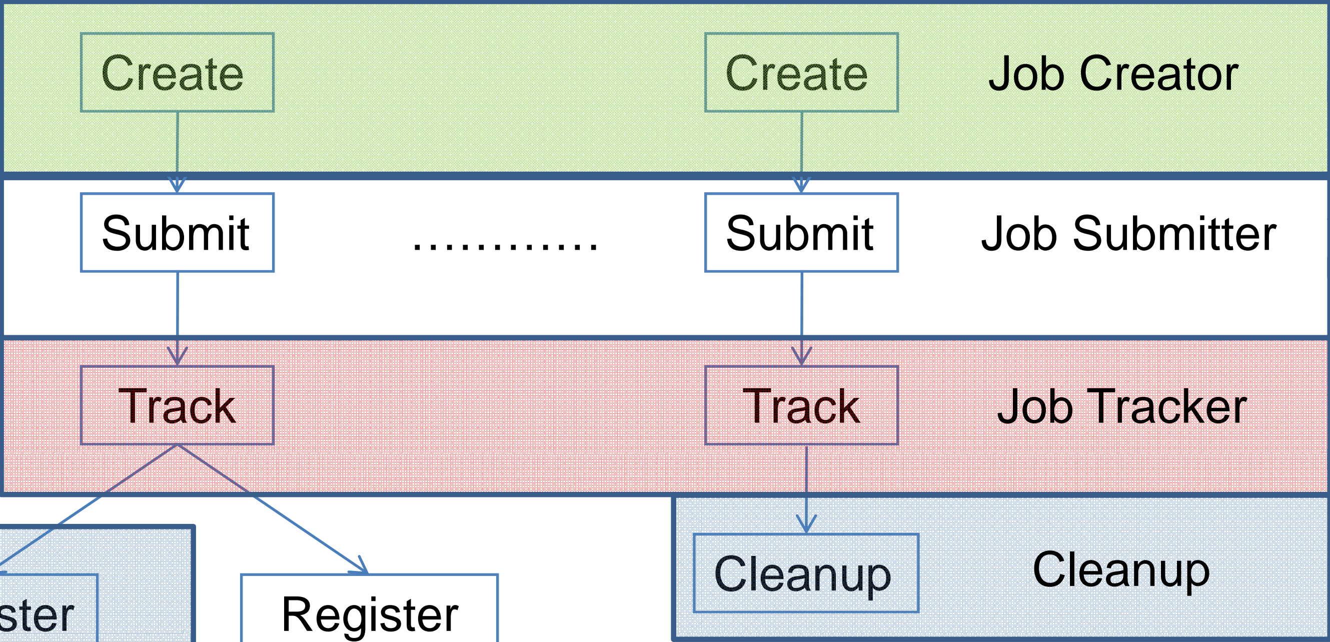


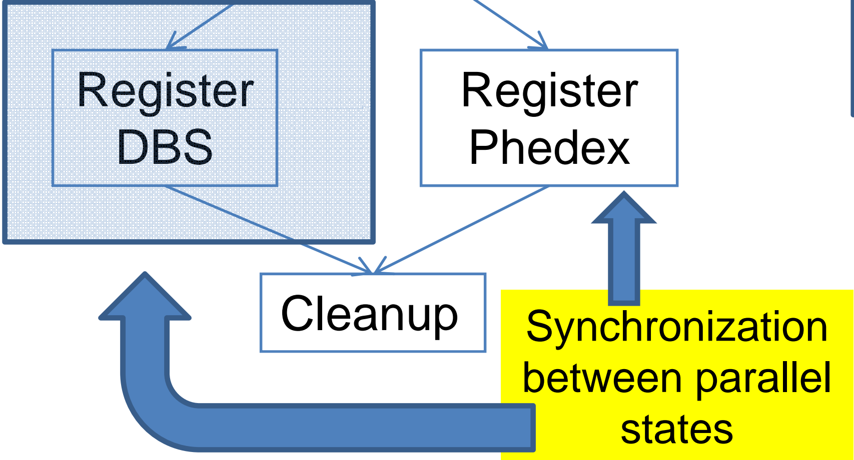
***Tier0 does not have a request layer**

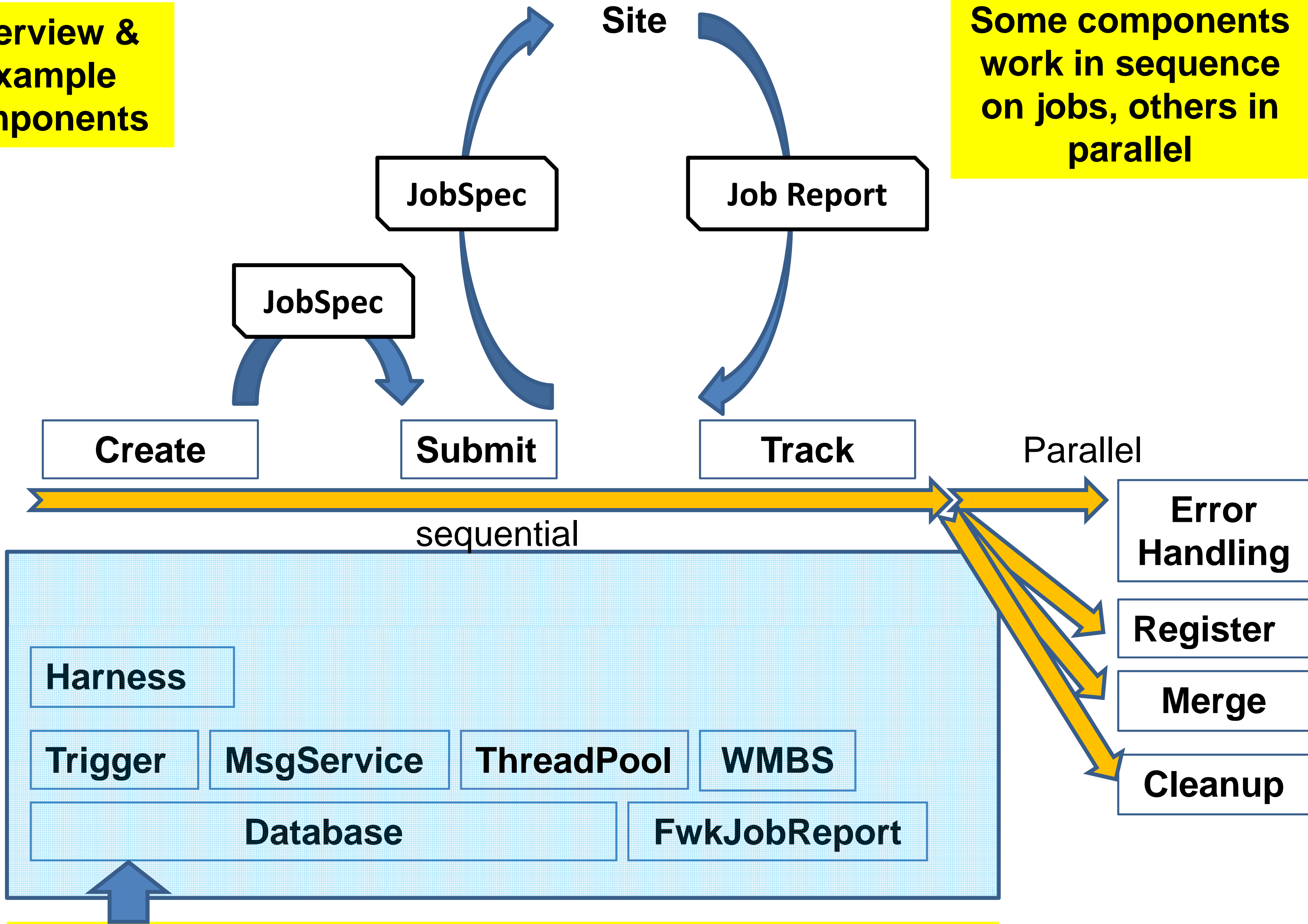
Job Life Cycle Management

- Different components based on WMCore handle various states of a job
  - Create, submit, track, etc…
  - Components involved with a job depends on its 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)






Overview & Example components
Site
Some components work in sequence on jobs, others in parallel
JobSpec
Job Report
JobSpec
Create
Submit
Track
Parallel
sequential
Error Handling
Register
Merge
Cleanup
Harness
Trigger
MsgService
ThreadPool
WMBS
Database
FwkJobReport
WMCore provides common components without being context /project specific (e.g. CRAB, T0, Production)

Msg Service

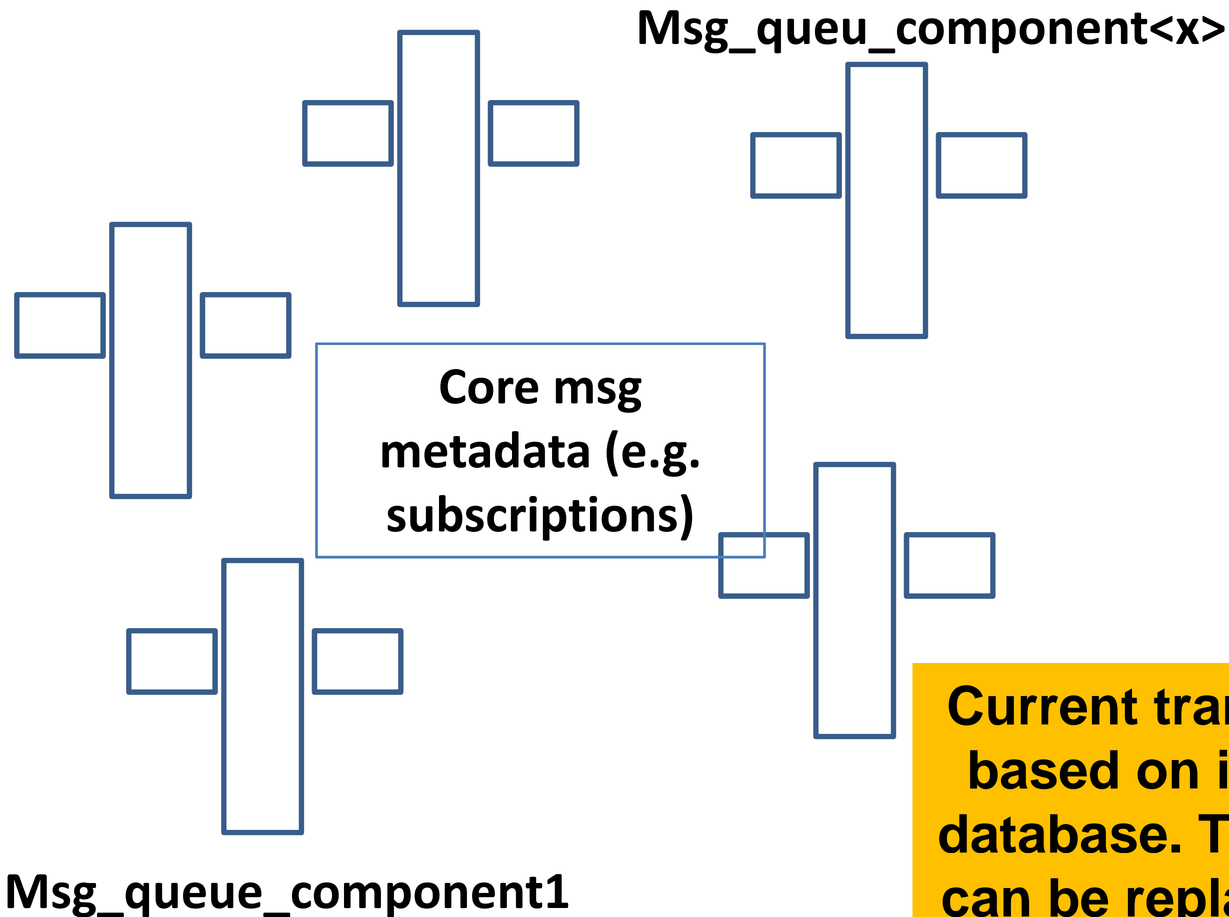
Delivery of asynchronous messages

+ **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

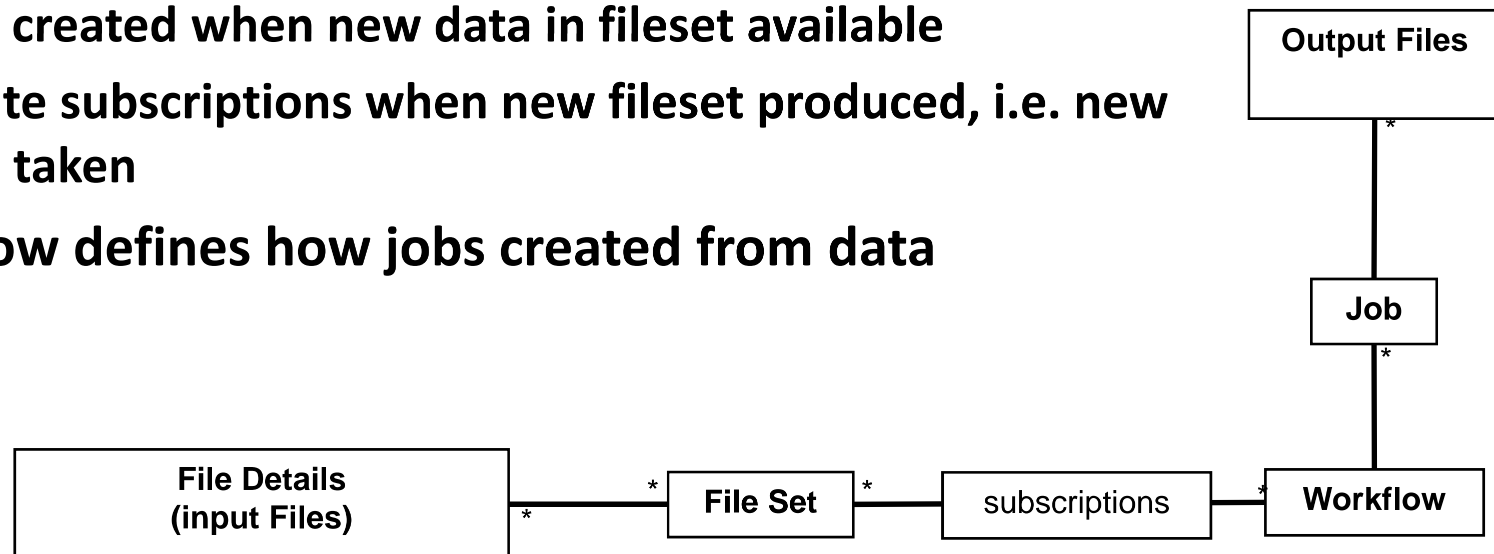
Other Core Services/Libraries

- 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

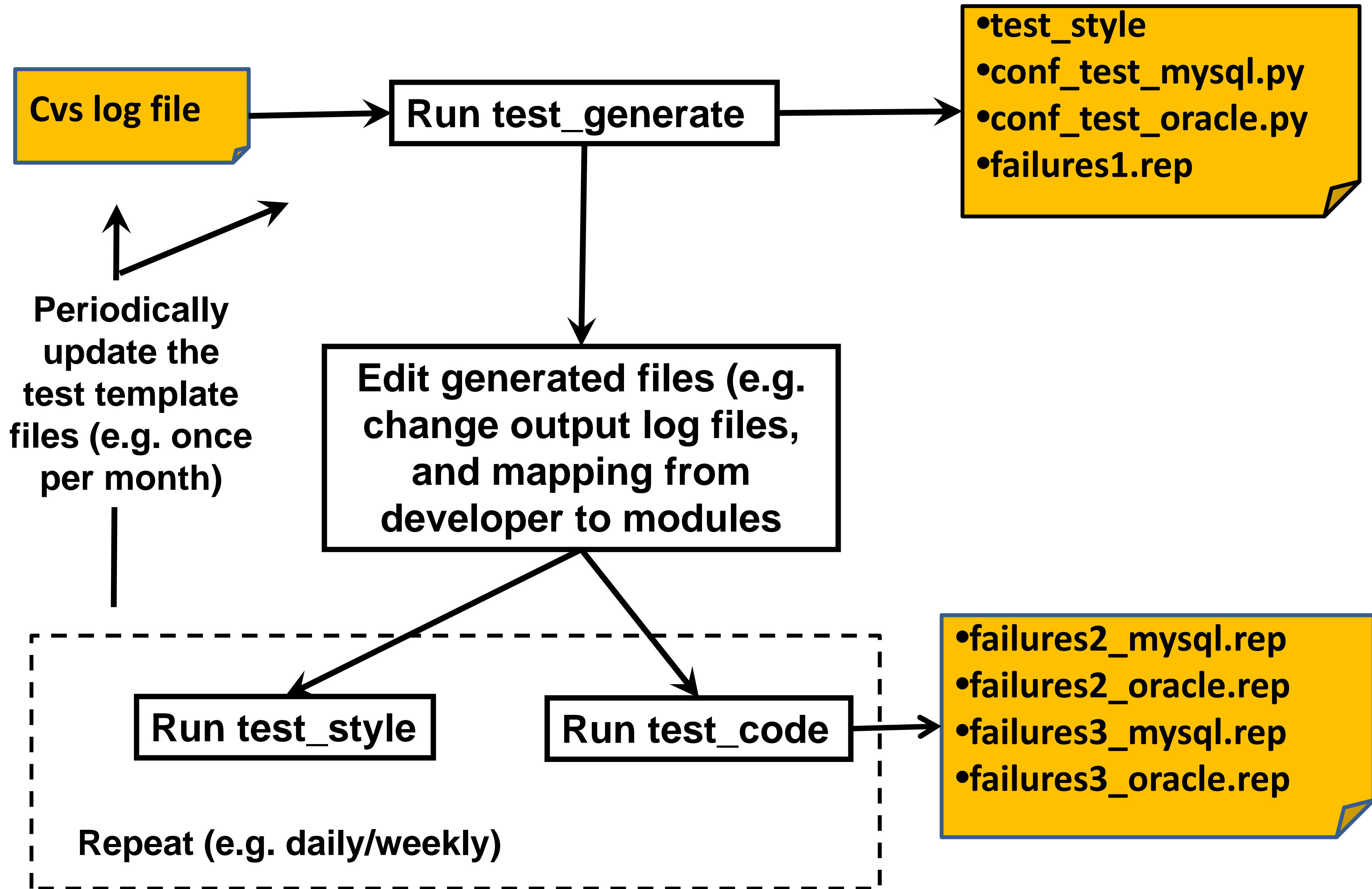
Workflow Management Bookkeeping System (WMBS)

- Provide a generalized processing framework
- Current system designed for production not processing
- Subscription = workflow + fileset
- Automate as much as possible
  - Jobs created when new data in fileset available
  - Create subscriptions when new fileset produced, i.e. new runs taken
- Workflow defines how jobs created from data



Development

- Small team + tight schedule
- Use “Sprints” to make rapid progress
- Emphasize code style, quality, testing etc.
- Periodically produce test reports
  - Test on MySQL, SQLite and Oracle (not all developers have easy access to all architectures)
  - Name and shame developers with failures
  - Determine author from CVS

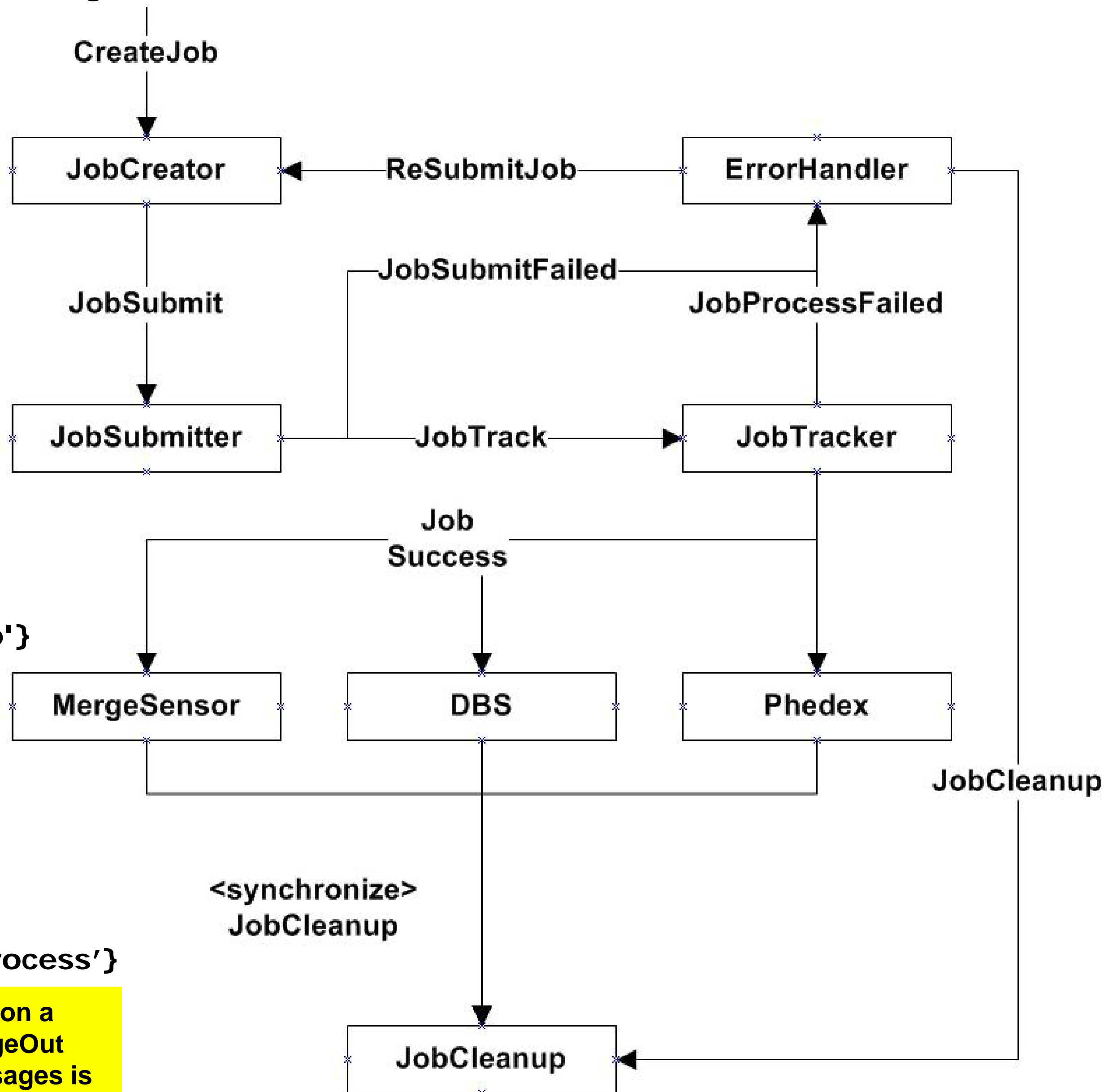

Cvs log file
Run test_generate
•test_style
•conf_test_mysql.py
•conf_test_oracle.py
•failures1.rep
Periodically update the test template files (e.g. once per month)
Edit generated files (e.g. change output log files, and mapping from developer to modules
Run test_style
Run test_code
•failures2_mysql.rep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

Skeleton Code Generation

- Existing components parsed to generate stubs for new style components
- Author’s then fill in the blanks (Handlers etc.), or
- Rewrite as necessary
- New (skeleton) components can be generated from a simple specification
- Heavy lifting taken care of - leaving the author to concentrate on the task at hand

(Workflow) Code Generation

- Workflow can be visualize[d]
  - Components & messages



Defines a Trigger for component synchronization.

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

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

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

Conclusion

- **CMS distributed projects are moving to a common codebase.**
  - **Library functionality (grid interaction etc.).**
  - **Common component functionality.**
- **Taking the opportunity to refactor a lot of the existing code and improve testing etc.**
- **Provide common data processing functionality.**
- **Aggressive schedule but aiming for reduced maintenance cost for the future**