

ecflow

Axel Bonet
John Hodgkinson

Forecast Production
Integration Team

Topics to be covered in theory sessions

- Aims
- Overview
- eflow Components
- Writing operational suites and scripts
- eflow in use
- Important Concepts
- Python API
- Migration to eflow
- GUI

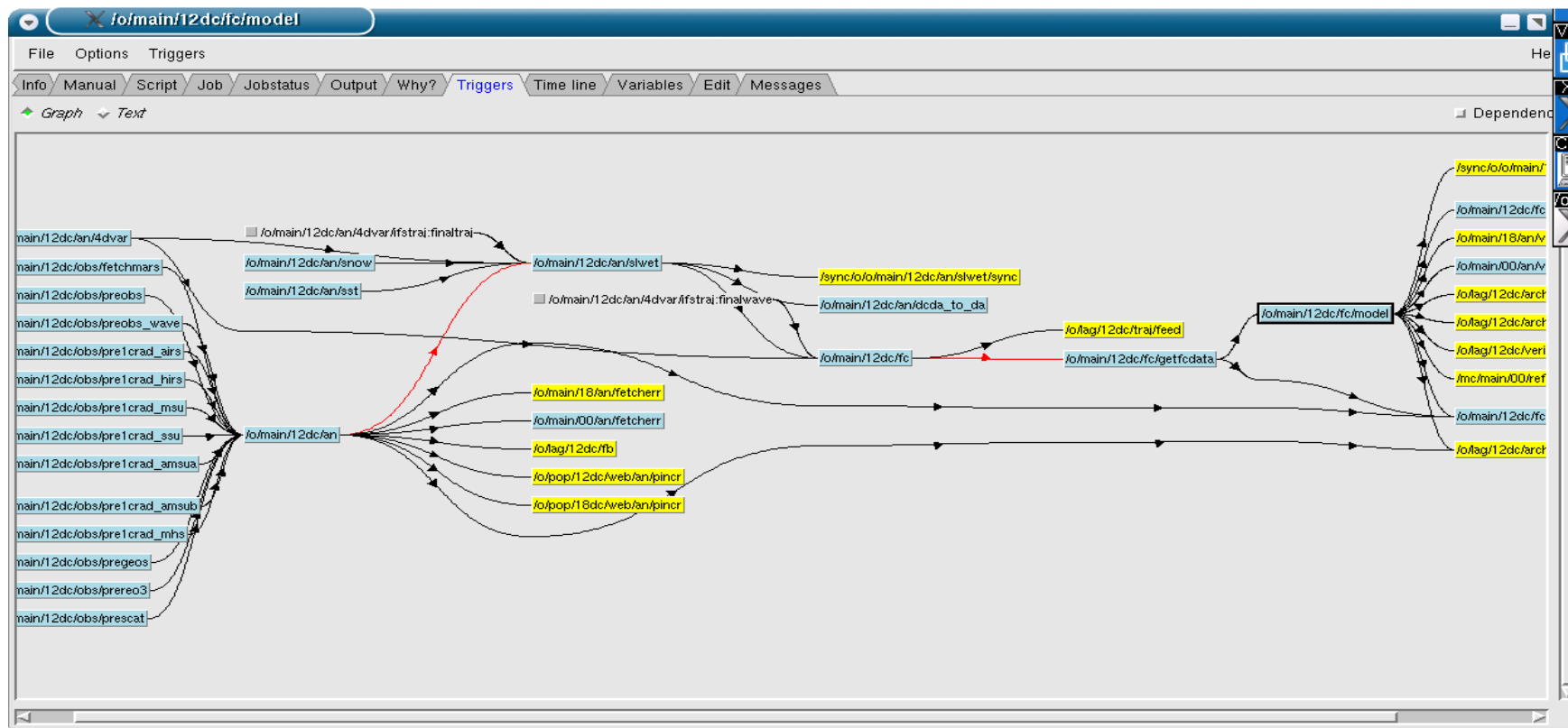
Course Aims

- Introduce ecflow, its API and viewer(s)
- Guidelines on designing an “operational” suite
- Show how ecflow is used
- Cover aspects of ecFlow : CLI, API, GUI and suite/task design
- Aim for students to be able to write and implement suites of a reasonable complexity by the end of the course

Overview: ecflow

The ECMWF workflow manager -

“A general purpose application designed to schedule a large number of computer processes in a heterogeneous environment”



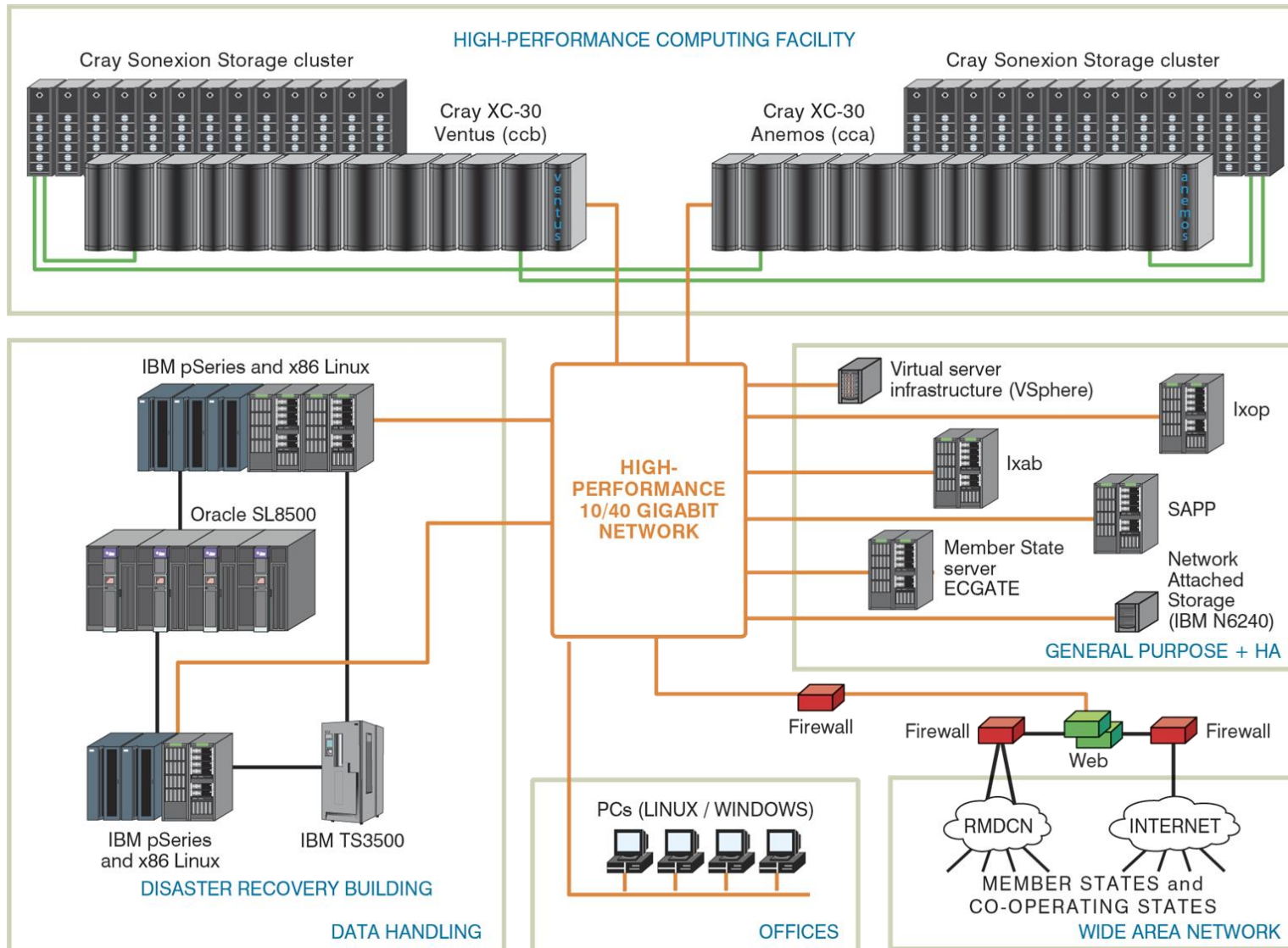
Overview: what is ecflow ?

- Work flow package
 - Runs large number of programs with dependencies
 - Tolerant to hardware and software failures
- Used at EMCWF to run all operational suites
- Submits tasks and receives acknowledgements from them
 - Using embedded child commands
- Stores the relationship between tasks
- <http://software.ecmwf.int/wiki/display/ecflow/Home>

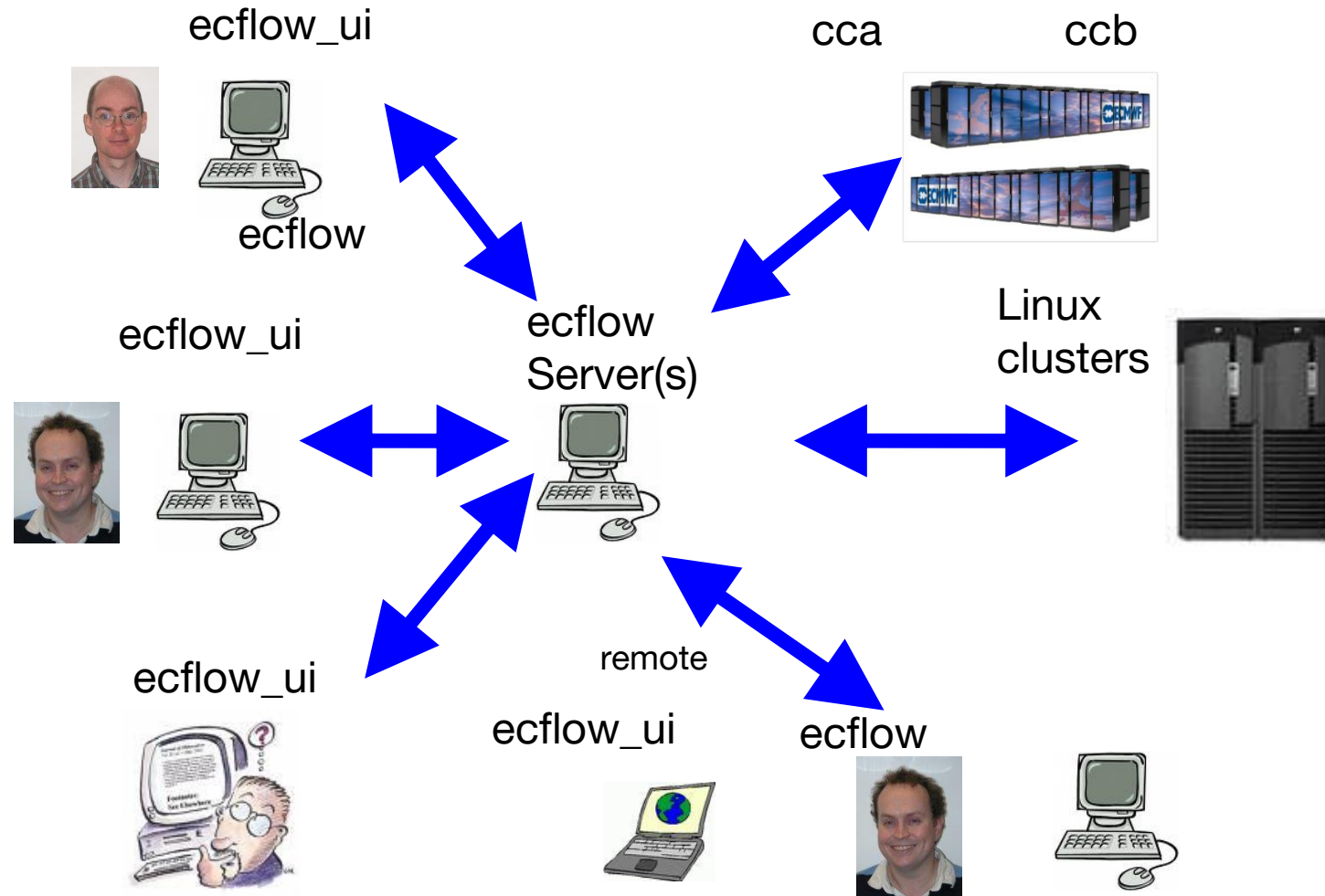
Overview: Features?

- Flexible inter-dependencies between tasks
 - e.g. triggering
- Complex automated scheduling
 - based on **events**, times, task **status**
 - Multiple users / platforms / queues
- Monitoring information via GUI and CLI
- Dynamic and interactive **supervision** in real time
 - execute, kill, check jobs
- Good **recovery** - at task and ecflow server level
- But ... ecflow is not a queuing system, ecflow is not a message passing system, it is a **scheduler**

Overview: Schematic of our systems



Simplified view of our usage ecflow



Components of ecfLOW

- **ecflow_server**
 - The scheduler, continuously running daemon process (nohup &)
- **ecflow_client**
 - Command line interface to ecfLOW
 - Child commands updating status and attributes
- Python API
- **ecflow_ui, ecfLOWview**
 - Graphical interface to ecfLOW

How it works

- Define **suite**
 - Structure (grouping of tasks, interactions)
 - Locations of input scripts, job files location, output file location, etc
- Design task template scripts
 - add “hooks” to communicate to ecflow server
- When expected server **submits** the job
- Job tells server has started
 - `ecflow_client -init $ECF_RID`
- If an error is detected, the job tells the server:
 - `ecflow_client -abort “reason”`
 - Use error trapping to communicate errors
- If task completes, tells the server: `ecflow_client --complete`
 - Send complete client command

Server Functionality

- Setup environment: at ECMWF
 - `module load ecflow` `# /usr/local/apps/ecflow/current/bin`
`# set up PATH etc`
- Starting the server
 - `ecflow_start.sh` `# specific start up script`
 - `ecflow_stop.sh`
 - `ecflow_server --port 3141` `# manual start`
 - `nohup ecflow_server > ecf.out 2>&1 &`
- Server hosts the suites
- Checkpoints (backup) suites tree periodically: as text file (4.8.1)
- Handles user and job requests
- Logs activity

ecflow: checking the server

- Identifying the presence of a server
 - `ecflow_client --ping --port 3141 --host localhost`
 - `ecflow_ui`, `ecflowview`
 - `ps -ef | grep ecflow`
 - `netstat -lnptu` (only if server started with your user ID)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
<code>tcp</code>	<code>0</code>	<code>0</code>	<code>0.0.0.0:6008</code>	<code>0.0.0.0:*</code>	<code>LISTEN</code>	<code>-</code>
<code>tcp</code>	<code>0</code>	<code>0</code>	<code>0.0.0.0:6009</code>	<code>0.0.0.0:*</code>	<code>LISTEN</code>	<code>-</code>
<code>tcp</code>	<code>0</code>	<code>0</code>	<code>0.0.0.0:25</code>	<code>0.0.0.0:*</code>	<code>LISTEN</code>	<code>-</code>
<code>tcp</code>	<code>0</code>	<code>0</code>	<code>0.0.0.0:<u>3141</u></code>	<code>0.0.0.0:*</code>	<code>LISTEN</code>	<code>5972/ecflow_server</code>

Text client interface

- For remote assistance, batch mode or directly from the shell
- Self contained manual:
 - `ecflow_client --help <command (optional)>`
- Define interaction via environment variables
 - `ECF_PORT=3141 ECF_HOST=host3 ecflow_client --get`
 - Or explicitly: `ecflow_client --port 3141 --host host3 --get`
- Can use to monitor and interact with server
 - `ecflow_client --get`
 - `ecflow_client --alter change variable SLEEP 10 /path/to/node`
- Load-replace nodes into the server
 - `ecflow_client --load suite.def`
 - `ecflow_client --replace /path/to/node suite.def`
- Write to log file
 - `ecflow_client --msg =“this to be written to log file”`

Child commands: `ecflow_client`

- For communication between tasks and server
 - `ecflow_client --help child`
- Status update:
 - `ecflow_client --init <PID/QID>` # task is active e.g. \$\$ (Linux)
 - `ecflow_client --abort <reason>` # task has aborted
 - `ecflow_client --complete` # task has completed
 - These commands are blocking (expect acknowledgement from the server)
- Attribute update:
 - `ecflow_client --event <name>` # set an event
 - `ecflow_client --meter <name> <value>` # update a meter
 - `ecflow_client --label <name> <text>` # set a label
- Embedded trigger:
 - `ecflow_client --wait="/suite/t1==complete"` # wait for external task to complete
 - `ecflow_client --wait="%CONDITION:1==1%"` # wait for a condition set by variable

ecflow_ui

- Monitoring
- Direct interaction with ecflow Servers
- **Most** ecflow client commands available
- Easy access to helpful information
 - script, manual, job, output, web page, etc.
- Alarm features, runs even when iconized
- Configuration by panels, system
 - Edit/Tools->Preferences->Menus User-Operator-Administrator
- Can **mask** information from being displayed
- Config files: ~/.ecflow_ui, servers, options, menu vs ~/.ecflowrc

The screenshot displays the ecFlowUI (4.5.0) interface. The top menu includes File, Edit, View, Refresh, Servers, Tools, and Help. The main area is divided into a Tree view on the left and an Info panel on the right. The Tree view shows a hierarchy: local > test > f1 > t2. The 'test' node is expanded, showing 'I1: 2/2' with two green dots and 'inlimit:11'. The 'f1' node is expanded, showing 'T t1' with a progress bar and 'step: 14', and 'T t2' with a progress bar and 'step: 12'. The Info panel shows 'man'.

Below the tree view is a Table view showing the following data:

Node	Status	Type	Trigger
/test/f1	active	family	
/test/f1/t1	active	task	

Terminology (1/2)

- Root ecflow server itself
- Suite Collection of nodes and attributes
- Family Collection of tasks + other families
- Task Unit of work, a computer job
- Alias Task made to run independently
- Node Generic term for Suite, Family, Task
- Attribute Node property (behavioural, structural, monitoring)
- Event Milestone set within a task
- Meter Like an event, with range of values
- Label Text Information updated by the task

The screenshot displays the ecFlowUI (4.5.0) interface. The top menu includes File, Edit, View, Refresh, Servers, Tools, and Help. The main area is divided into a 'Tree' view on the left and an 'Info panel' on the right. The 'Tree' view shows a hierarchy: local > test > f1 > t2. Under 'local', there is a suite 'test' (S) with 12 nodes, containing a family 'f1' (F) with 11 nodes. Under 'f1', there are two tasks: 't1' (T) with 14 steps and 't2' (T) with 12 steps. The 'Info panel' shows details for the selected node, including 'man'.

Below the tree view, there is a 'Table' section with a filter and a table of active nodes:

Node	Status	Type	Trigger
/test/f1	active	family	
/test/f1/t1	active	task	

Terminology (2/2)

- `<name>.def` **Definition** file describes a suite
 - Expanded or high level
- `<name>.ecf` **Wrapper, task template** file
- `<name>.jobN` **job-file**
 - created by ecflow from the ecf-file
 - that is sent by ecflow to be executed
- `<name>.usrN` **alias-file**: from direct user interaction with GUI
 - Test, debug, rerun without status side effects
 - Alias has an alias number and a job instance number
- Variables stored by server, substituted into a job
 - `%VAR:<default>%` # `<default>` is default

ecflow template script - tasks wrapper (.ecf)

- Similar to a shell script

```
%include <head.h>
echo "I am a script in %ECF_HOME%"
%include <tail.h>
```

- On submission job file is created
 - **Preprocessing**
 - **Include** lines are replaced with relevant file
 - **Variables** are substituted with server stored values
 - **Preprocessed** to create a **job** file and **submitted**
 - Job file can be ksh, bash, python, perl, ruby
- Extension is **.ecf**
 - configurable **ECF_EXTN** (.py, .sh, .pl) in the suite definition

Sample head.h include file (1 of 2)

```
#!/%SHELL:/bin/bash%
set -e                # stop the shell on first error
set -u                # fail when using an undefined variable
set -x                # echo script lines as they are executed

# Defines the variables that are needed for any communication with ECF

export ECF_PORT=%ECF_PORT%    # server port number
export ECF_HOST=%ECF_HOST%    # name of ecflow host that issued this task
export ECF_NAME=%ECF_NAME%    # name of this current task
export ECF_PASS=%ECF_PASS%    # unique password
export ECF_TRYNO=%ECF_TRYNO%  # current try number of the task
export ECF_RID=$$            # record the process id. Also for zombie detection

# Define the path where to find ecflow_client

export PATH=/usr/local/apps/ecflow/%ECF_VERSION%/bin:$PATH

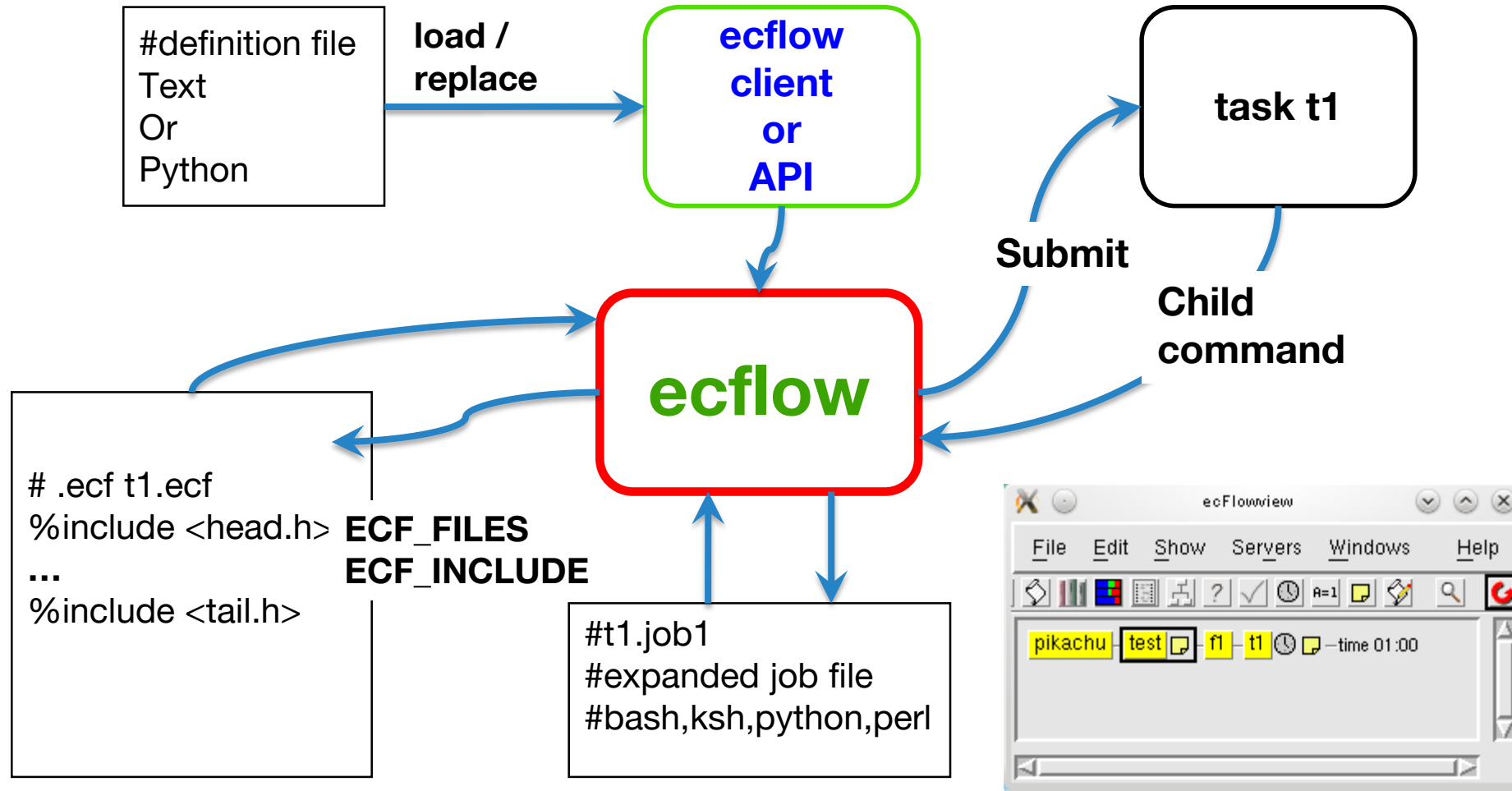
ecflow_client --init=$$      # tell ecflow we have started
```

Sample head.h include file (2 of 2)

```
ERROR() {                                # Define a error handler
    set +e                                # Clear -e flag, so we don't fail
    wait                                  # wait for background process to stop
    ecf_flow_client --abort="trap"        # Notify ecf_flow something went wrong, 'trap' as the reason
    trap 0                                # Remove the trap
    exit 0                                 # End the script
}

trap ERROR 0                             # Trap any calls to exit and errors caught by the -e flag
# Trap any signal that may cause the script to fail
trap '{ echo "Killed by a signal"; ERROR ; }' 1 2 3 4 5 6 7 8 10 12 13 15
```

Relationship between .def, .ecf and .job files



End Section

Practical

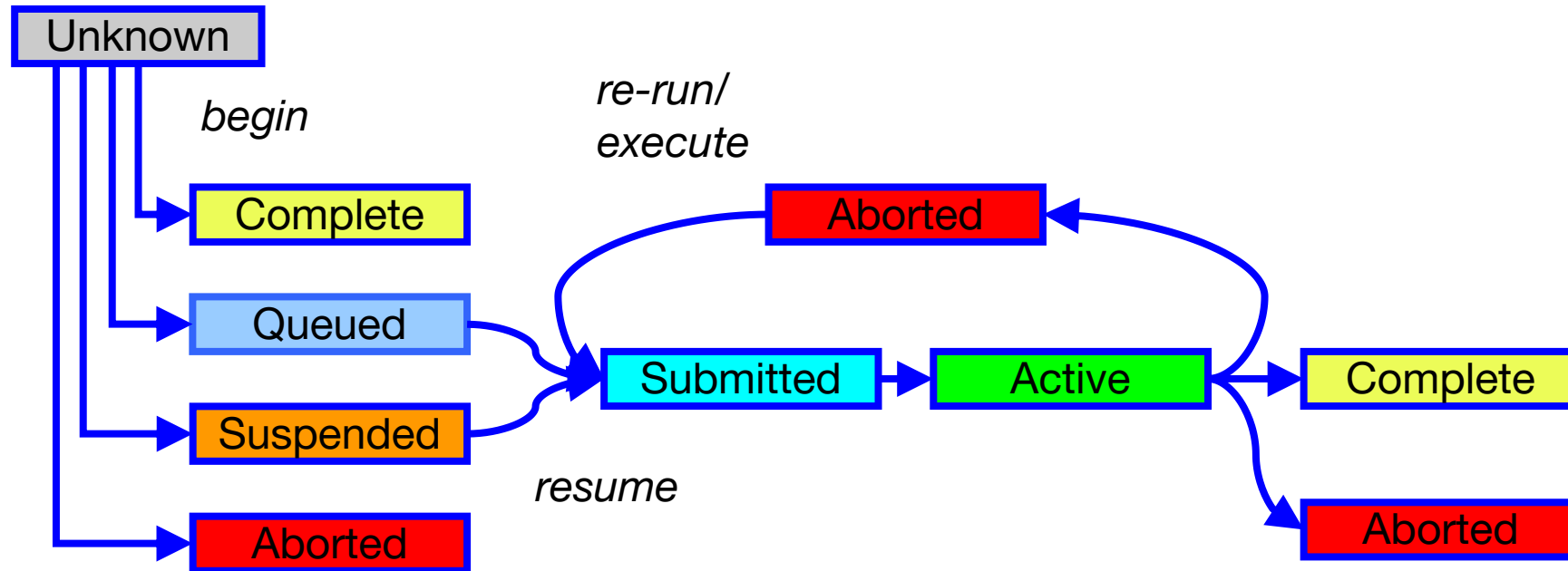
<https://software.ecmwf.int/wiki/display/ECFLOW/Introduction>

module load ecfLOW/4.8.1
module switch ecfLOW/4.8.1

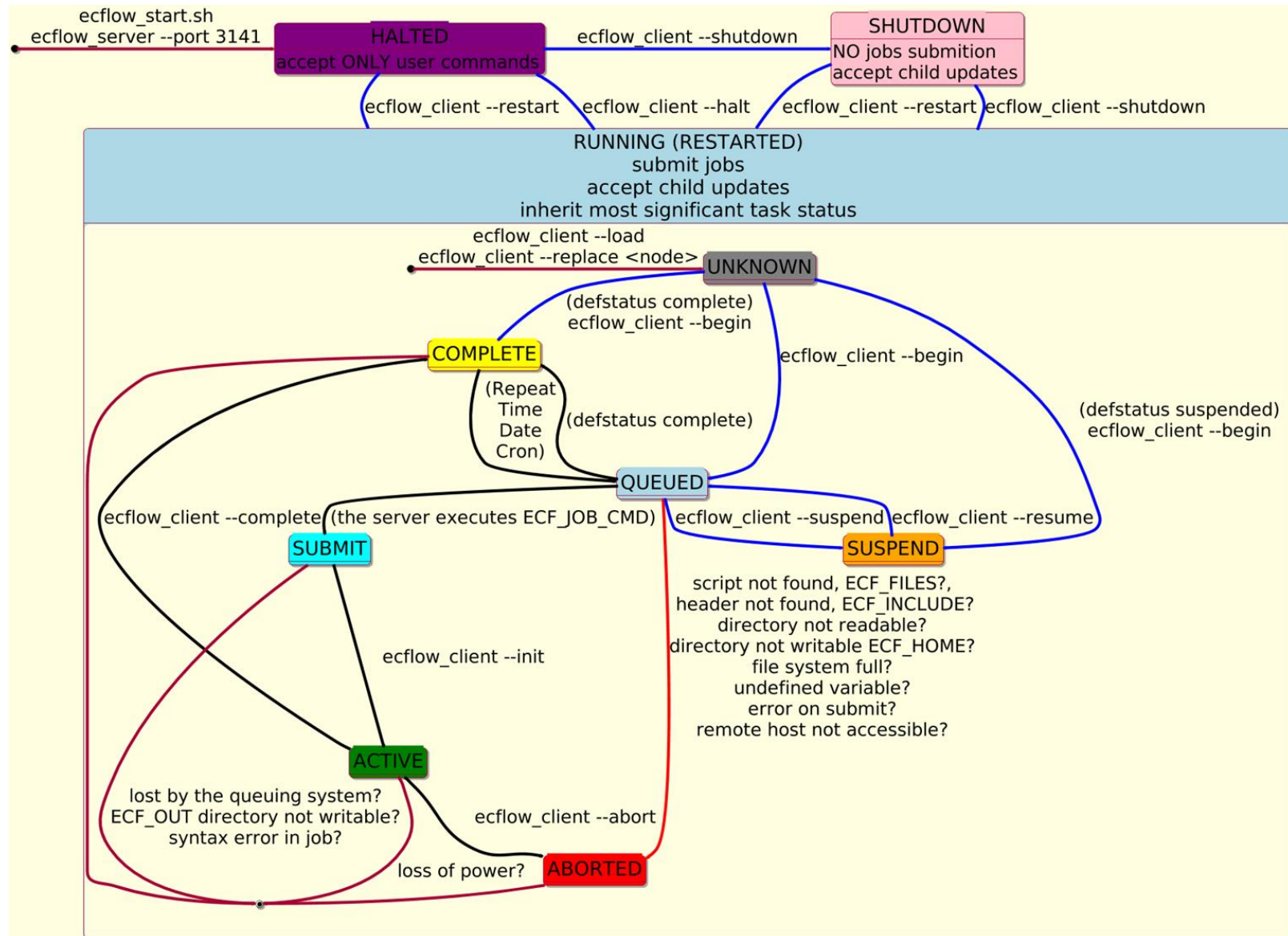
Important Concepts : Status Flow (1/2)

- After you **load** a suite its status is unknown,
 - use **begin** to start: `ecflow_client --begin <suite-name>`
 - **defstatus** suspended # def-file
- **re-run**, can be *automatic* if set in definition-file
 - edit `ECF_TRIES 3`
- **repeat**, may take nodes back from complete to queued
 - repeat date `YMD 20180101 20201231 1`
- **date, time, cron** may also make a task queued again

Important Concepts: Status Flow (2/2)



Important Concepts: Status Flow (2/2)



Important Concepts: Dependencies

- Node may stay queued because:
 - ecfLOW server is **halted** (frozen, accept user command)
 - ecfLOW server is **shutdown** (no new submissions)
 - Parent has a dependency
 - Triggered by a state of another node
 - Waiting for time of day, day of a week, date of year
 - Limit it uses is full
 - Suspended
- Use “why” button with ecfLOW_ui to find out why
- GUI may be configured to hide attributes

The screenshot displays the ecfLOW UI (4.5.0) interface. The main window shows a tree view of a workflow. The root node is 'local' (12), which contains a sub-node 'S test' (N 11). The 'test' node has a limit of 2/2 and is currently active. It has two child nodes: 'F f1' and 'T t2'. The 'f1' node is also active and has a limit of 14. The 't2' node is active and has a limit of 12. The 't2' node has two sub-nodes, 'a' and 'b', which are currently inactive. The 'Info panel' on the right shows the path 'local > test > f1 > t2' and the manual page 'man'. Below the tree view, there is a table with the following data:

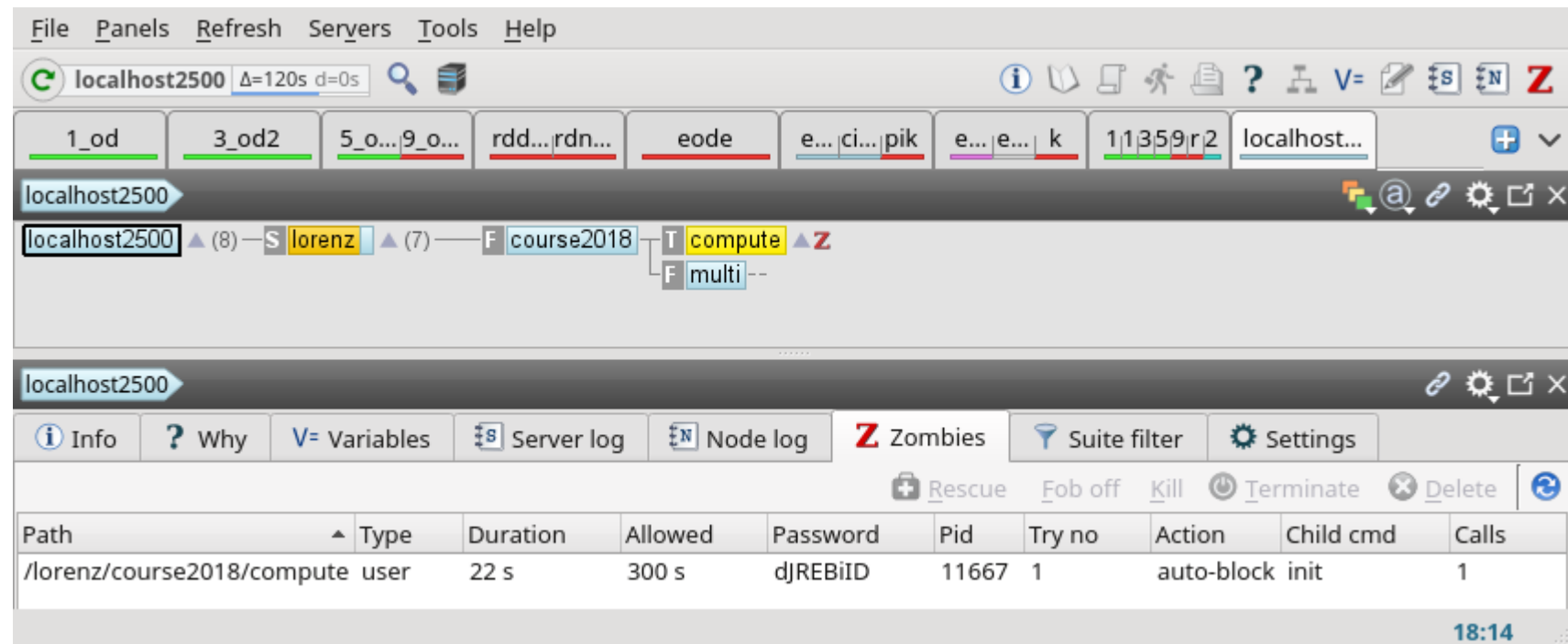
Node	Status	Type	Trigger
/test/f1	active	family	
/test/f1/t1	active	task	

Important Concepts: Inheritance

- Four different kinds of inheritance in ecflow
- **Variable** inheritance (top to bottom)
 - looks at the task first, then parents until it reaches ecflow itself
- **Status** inheritance (bottom to top)
 - family status reflects most important status of its tasks
 - likewise for suites and ultimately for ecflow
- **Dependency** inheritance: time, date, trigger, complete, inlimit
 - dependencies on any level
 - for task to run, it must be free to run as well as its parents
 - Trigger dependencies may be “hidden” below, time dependencies are not!
- **Zombie handling attribute** inheritance: automate zombie management

Important concepts: Zombies?

- On jobs submission, variable **ECF_PASS** set to pseudo-random value by ecf flow server
- Jobs are defined with unique identifiers **ECF_HOST-ECF_PORT-ECF_NAME-ECF_PASS**
 - A zombie arises when a **child** command is received and ECF_PASS does not match



The screenshot shows the ECF GUI interface. At the top, there are menu items: File, Panels, Refresh, Servers, Tools, Help. Below the menu is a search bar with 'localhost2500' and a refresh icon. A toolbar contains various icons including a red 'Z' for zombies. Below the toolbar is a row of job panels with names like '1_od', '3_od2', '5_o...9_o...', 'rdd...rdn...', 'eode', 'e...ci...pik', 'e...e...k', '11359r2', and 'localhost...'. The main area shows a job tree for 'localhost2500' with a parent job 'course2018' and a child job 'compute' marked with a red 'Z'. Below the job tree is a table with columns: Path, Type, Duration, Allowed, Password, Pid, Try no, Action, Child cmd, Calls. The table contains one row for the zombie job.

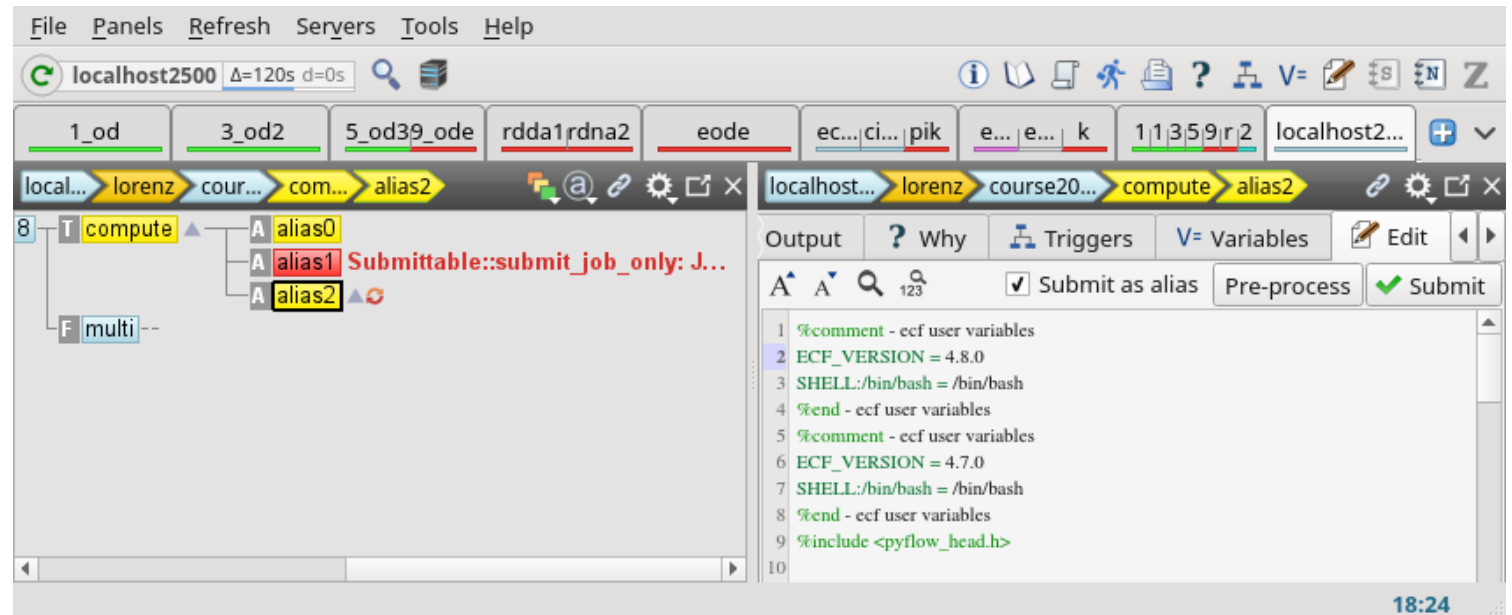
Path	Type	Duration	Allowed	Password	Pid	Try no	Action	Child cmd	Calls
/lorenz/course2018/compute	user	22 s	300 s	djREBiID	11667	1	auto-block	init	1

Important Concepts: Task versus Job

- Task is the piece of work you want ecfLOW to run
- Define the task in the suite **definition** file: *task t1*
- Write an ecfLOW **script** describing your task, “*vi t1.ecf*”
- When ecfLOW is ready to run your task, it
 - **edits** your task and creates a job-file using ecfLOW variables
 - if successfully created **submits** the job
 - the job **runs** (e.g. via a queuing system)
- A task is a **parameterised** or **configurable job** or a **template**

Important concepts: Alias

- An alias is a **dynamic node** attached to a task, created from GUI or `ecflow_client`
- There may be **multiple aliases for one task**
- Each alias can be **run multiple times**
- Initially ecflow server creates the **.usrN script** for the alias. You can modify it and rerun the alias.



The screenshot displays the ecflow GUI interface. The main window shows a task graph with a 'compute' task (yellow) and three alias nodes: 'alias0', 'alias1', and 'alias2' (all yellow). The 'alias2' node is selected and highlighted. The right-hand panel shows the script content for the alias, which includes environment variables and a pre-processor directive:

```
1 %comment - ecf user variables
2 ECF_VERSION = 4.8.0
3 SHELL:/bin/bash = /bin/bash
4 %end - ecf user variables
5 %comment - ecf user variables
6 ECF_VERSION = 4.7.0
7 SHELL:/bin/bash = /bin/bash
8 %end - ecf user variables
9 %include <pyflow_head.h>
10
```

ECF_MICRO

- A special character for ecflow: by default set to %
 - Used by variables it is pre-processed by the ecflow server (%VAR%)
- To get % write %% in scripts
- **%includenopp** <script> # include without preprocessing
- Nopp: No preprocessing in a block

```
date +%Y.%m.%d
```

```
%nopp
```

```
date +%Y.%m.%d
```

```
%end
```

- Change ecfmicro
 - Globally: `edit ECF_MICRO ^ # in def file`
 - Locally:

```
%ecfmicro ^ # in script -> set ECF_MICRO to ^
```

```
date +%Y.%m.%d # % is normal character
```

```
^ecfmicro % # set back to default %
```

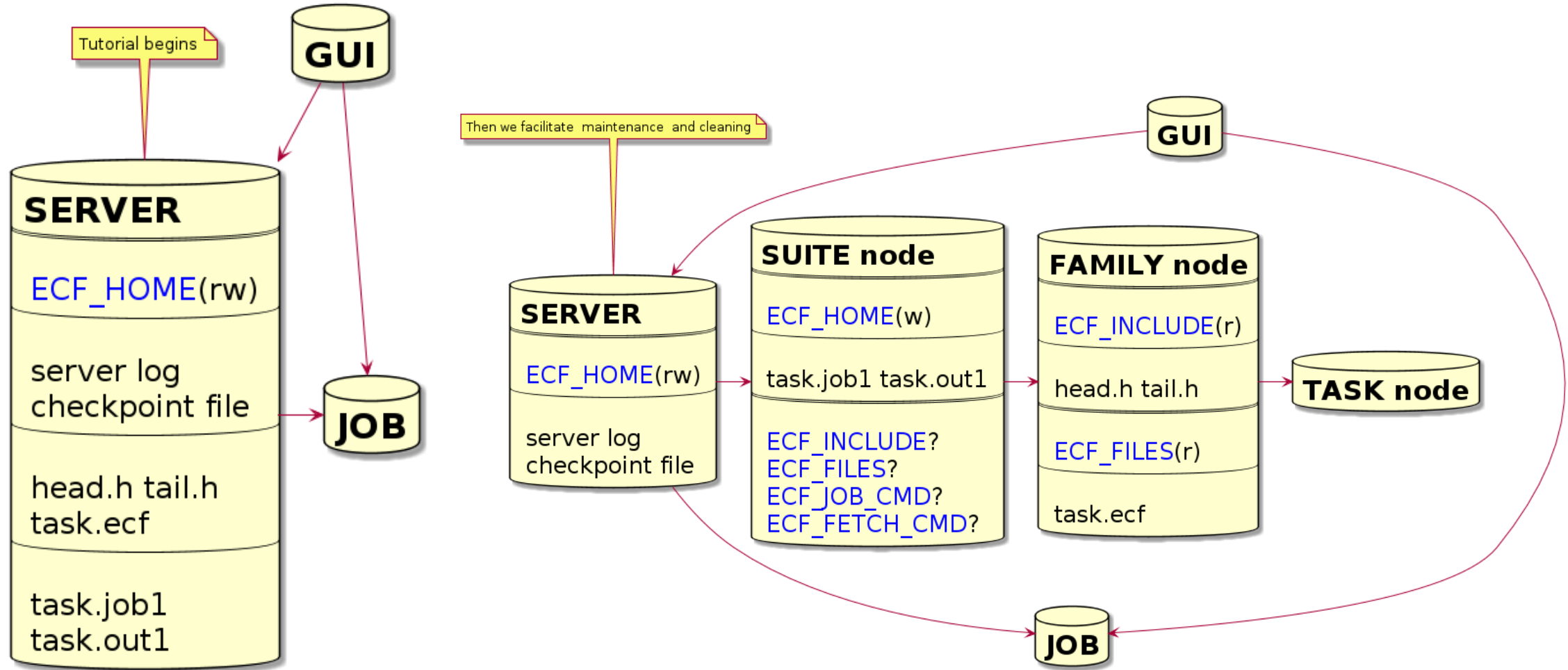

%include preprocessing directive

- **%include <file.h>**
 - Include a file under ECF_INCLUDE directory
- **%include "file.h"**
 - Include a file below ECF_HOME directory
- **%include /path/to/file**
 - a hardcoded location
- **%include:** NOTE % MUST be first character of the line
 - Avoid complexity, it prevents: echo "%include <file>"
 - Avoid ambiguity: # %include <file>
- **%include <%FILE_H%>**
 - Filename can be provided by a suite variable, here FILE_H
 - edit({"FILE_H": "config.oper.h", })
 - edit({"FILE_H": "config.test.h", })

Security

- Designed for collaborative working, so default access is **open**
- ecflow server can be protected with **white list** file: ecf.lists
 - restricted set of users with **read** (Script, Output) or **read-write access** (Edit, Submit)
- We use specific accounts for operations and research
- Communication on fixed port: **ECF_PORT**
- **4.4.8+**: **black** list file for user **authentication** to access server, suite, node
- 4.4.8+: Communication may be **encrypted**: compile with option `ENABLE_SSL`
- Some jobs are submitted for another user: careful with
 - job-file owner, output file owner, ssh settings, queueing system permissions
- Never run as root!
- Really: Do not even think about running as root!

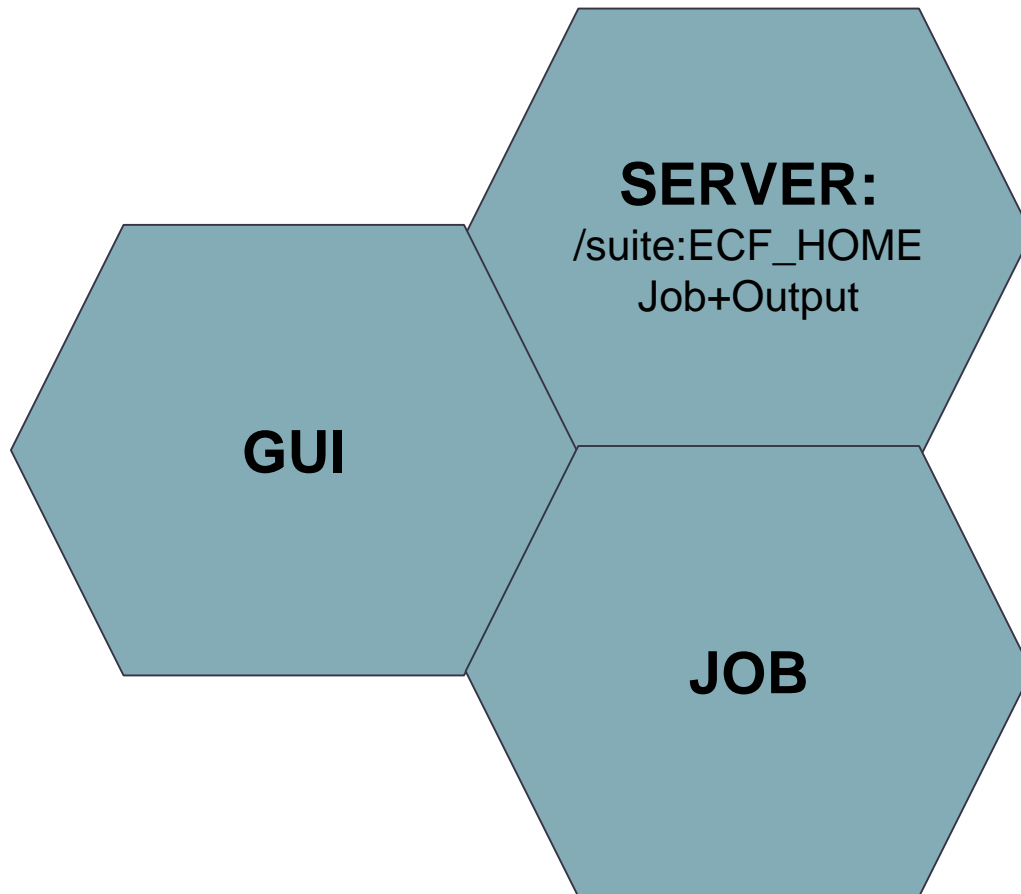
Files locations – ECF_HOME ECF_FILES ECF_INCLUDE



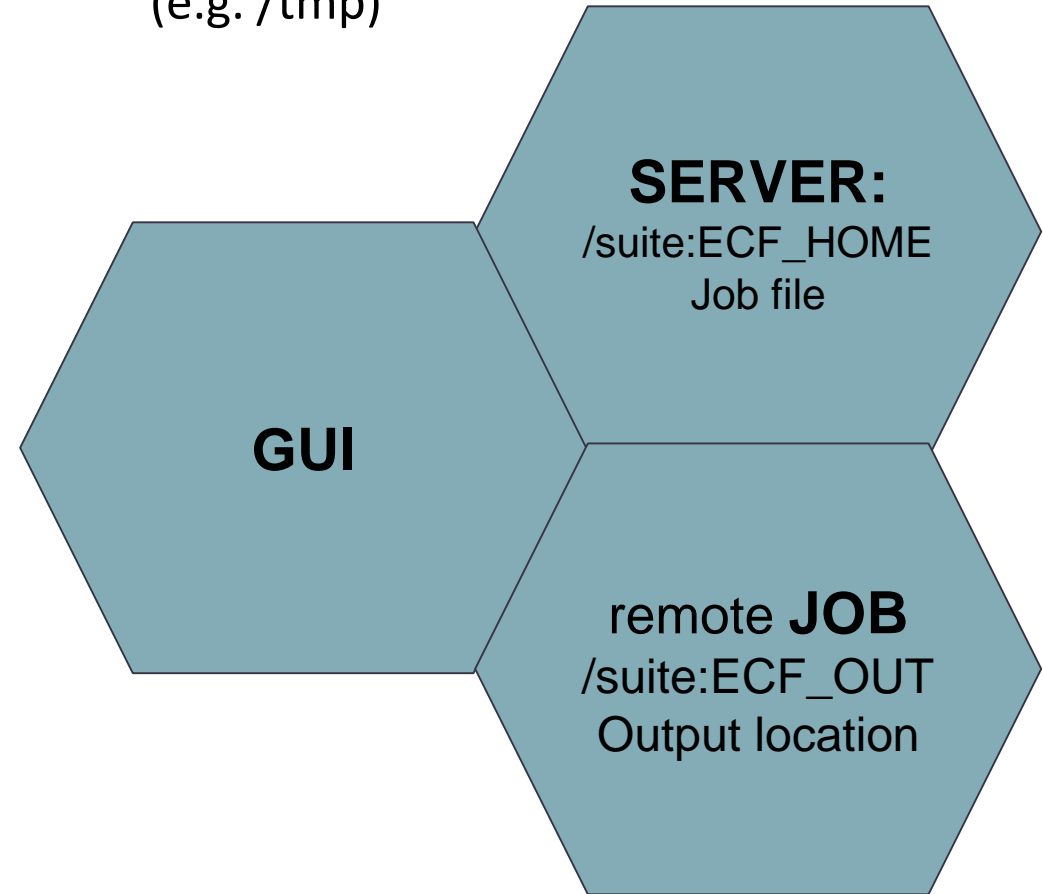
Files locations - ECF_HOME ECF_OUT

Direct disk access

Or access through server (preferences)



Remote job: sometimes job file and output file must be separated (e.g. /tmp)



Files locations - distributed suite

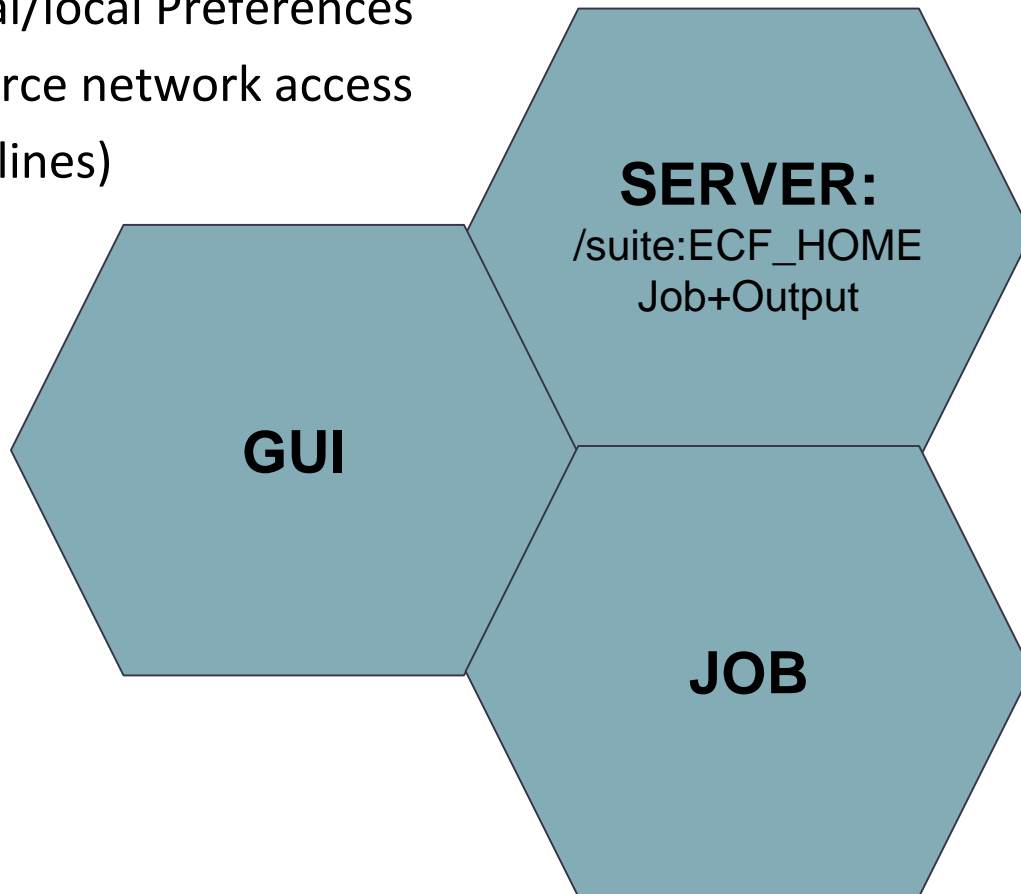
direct access: best case

all directories are visible on each host

global/local Preferences

To force network access

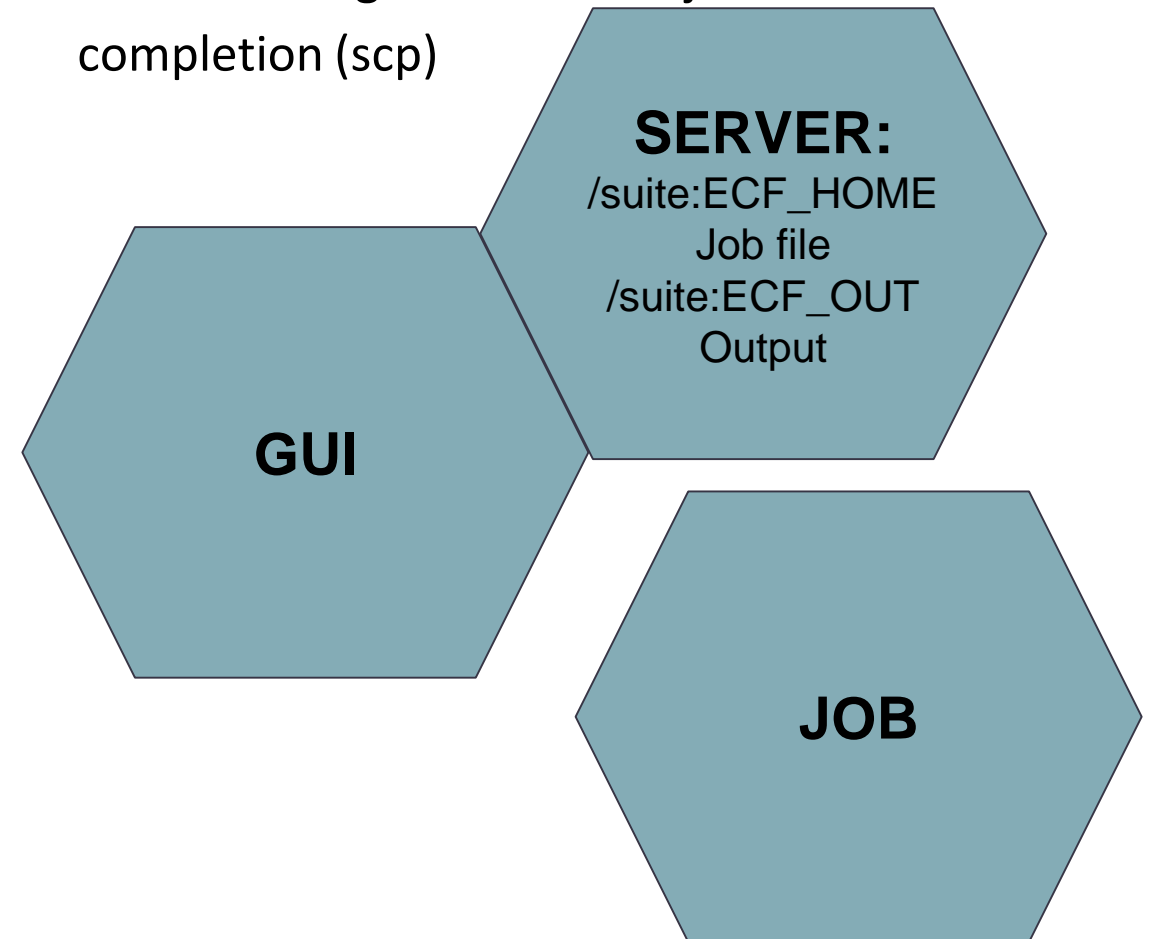
(15k lines)



Remote job, disk not shared:

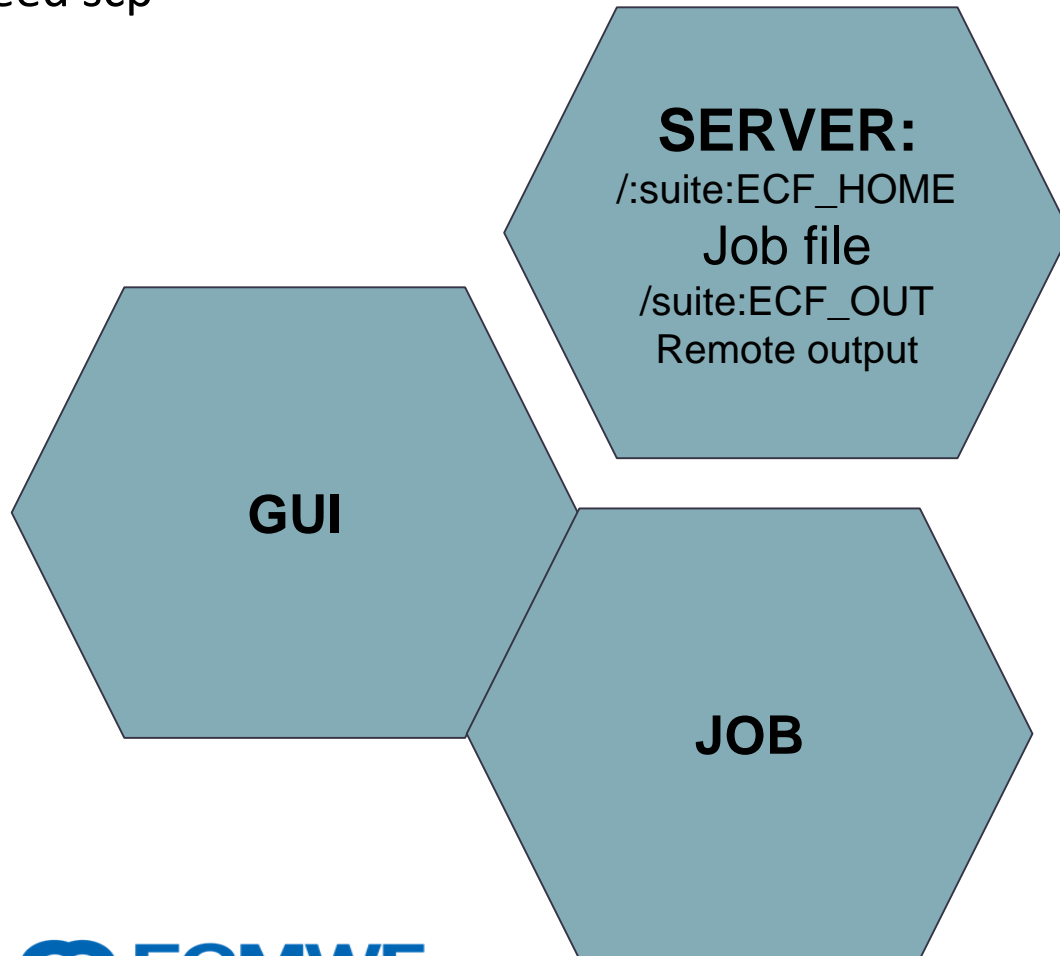
Access through server after job

completion (scp)



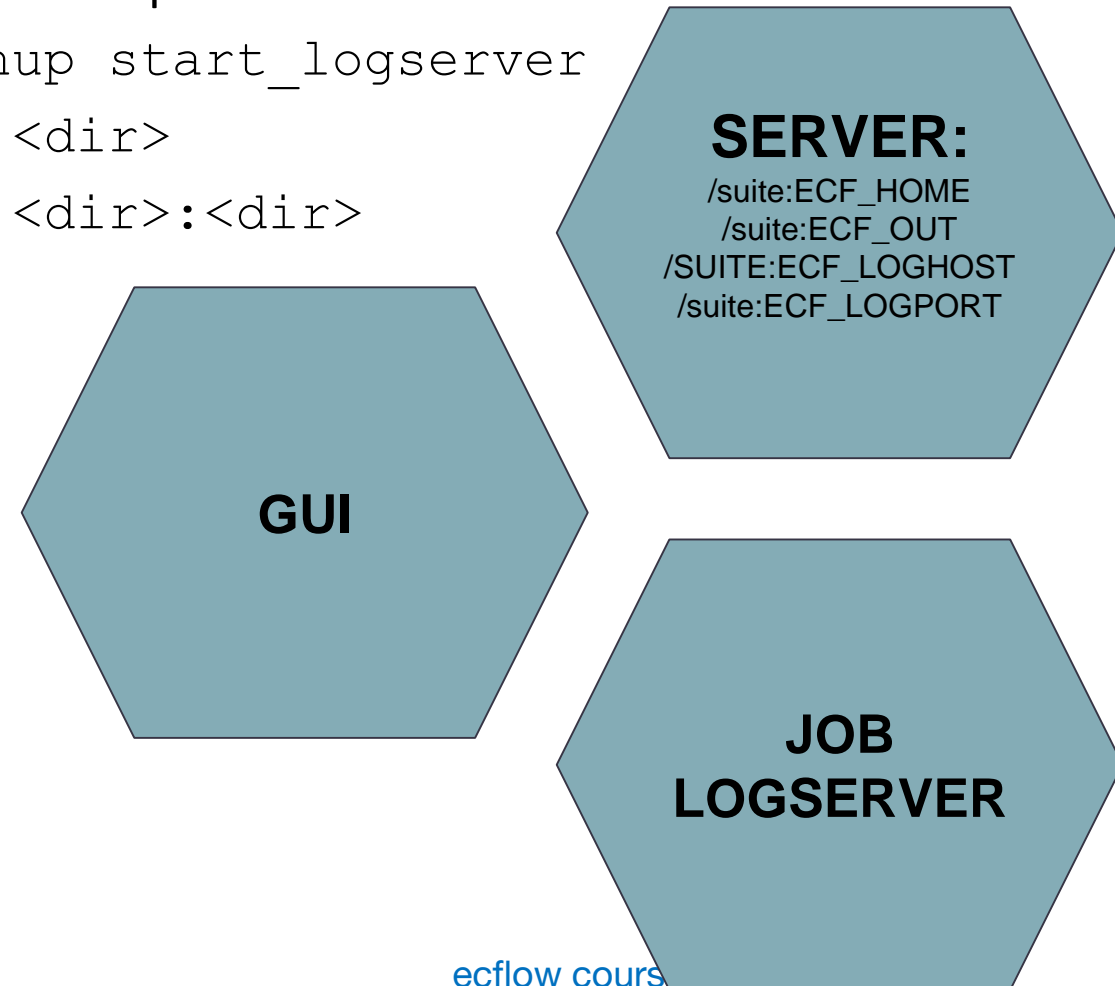
Files locations - distributed suite

Direct output access from GUI,
need scp



Normal case: use log-server to access live output,
scp at completion

```
nohup start_logserver  
-d <dir>  
-m <dir>:<dir>
```



Handling multiple platforms: ECF_JOB_CMD, ECF_KILL_CMD, ECF_STATUS_CMD, ECF_CHECK_CMD

- In course we generally submit jobs **directly**
- Can use a script to submit, to kill, get status behaviour depending on system:

```
edit ECF_JOB_CMD "SUBMISSION_SCRIPT %USER% %HOST% %ECF_JOB% %ECF_JOBOUT%"
```

- `if (PBS) then qsub ... qdel ... qstat`
- `If (SLURM) then sbatch ... scancel ... squeue`

- Can also use **generic queuing commands**

- `#QSUB -q emos`
- For PBS becomes `#PBS -q emos`
- For SLURM `#SBATCH -qos=emos`
- For SGE `"# $"` !!! beware a comment can lead to an error

ecflow 4.8.1

- Text based checkpoint files
- **Native python API** update
- %include %VARIABLE% - variable in include
- %includeonce
- **Repeat**: additional variables to simplify trigger expressions
- **Trigger**: cal::date_to_julian(), late can be used in trigger expression
- ecflow_client --alter add (limit, inlimit, label) change (trigger, complete):
 - Beware to keep updated the definition file
- Nodes attribute **sorting**: limits, variables, events, meters, labels
- ecflowUI updates
- ECF_HOST (was ECF_NODE)

ecflow 5.0.0 – Future Release

- GUI and server not compatible with 4.x.x
- **archive(migrate), restore** attribute to get lighter server and GUI
- A new attribute: **queue** (worker-queue pattern)
- Family Limit
- Better zombie identification (password, pid, user command)
- Query command (event, meter, non blocking trigger check)
- C++11
- Updated Boost library
- Python3
- Security (password protection, host identification)
- ecflowview decommissioned

Python definition file

- The definition may be sequential (like a bash suite definition),
 - Starts at the beginning and you follow it through to the end
 - Fix/verify a suite before loading?
- Object-oriented design opens more possibilities
 - **Stream-like** design, no temporary variables (Functional Programming)
 - Use functions to return a family or a task
 - Use a **class** to store objects to be accessed by multiple members
 - Another **module** may **add** attributes (Trigger, Late, Variables), **delete**, **replace**
 - Navigate the suite (tune, verify) before loading it?
 - Trigger expressions may be computed dynamically from node objects (path)
- Readable code, peer-review, KISS

Python definition file

- Create a definition object:

```
- defs = ecflow.Defs() # create an ecflow definition
```

- **Module** script: provide families

- **Standalone** script:

```
- if __name__ == "__main__" :  
    SUITE = TC3Suite(defs, EXPVER) # create an instance of class  
    SUITE.suite() # and execute its method suite()
```

- Options: target suite, node to replace, host server, mode SMS/ecflow, expand/print definition

- Class derivation: **extend, disable** parent class abilities

```
- class TC3Suite(ic.Seed):  
    def setup(self,node): pass  
    def main(self,node): pass
```

Python

- **Typed variables:**

- `if VERSION in ("0001", 9001, "9001"): print "str OR integer"`
- `if CYCLE == "00": print 'ok'`

- **Object Oriented Programming OOP**

- **Composition v. inheritance:**

- class derivation: operational vs test suite
- Extend a suite (`is_a`)
- “No more if” ...

- multiple inheritance: separate system and functional aspect of a suite?

- **Polymorphism:** treat all Attributes as one entity

- **Encapsulation:** complexity hiding mechanism, restriction mechanism (pyflow)

- **classes, instances, methods:** e.g. new attributes created from compounded native attributes

Python

- raise, catch **exceptions**
- dynamically typed, aka Duck Typing
 - it is then possible to mix types (ecflow-SMS GUI)
- **Functional Programming** in Python:
 - eliminating flow control statements
 - Functions as first class objects
 - Reduce number of temporary objects
 - **List comprehension**
- embedded, or library extension
- portable, open source

Suite design with Python

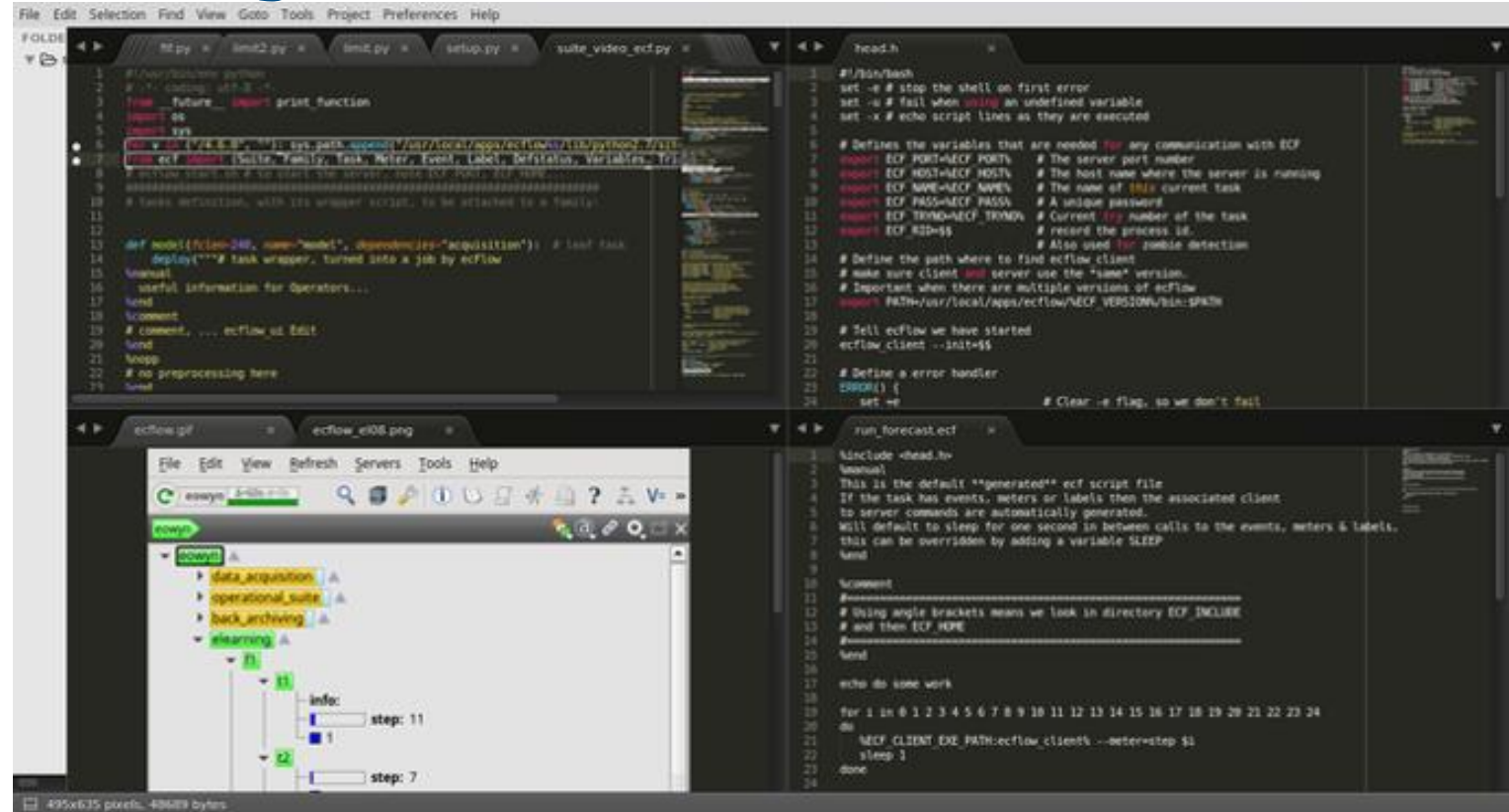
- python modules:
 - No global scope anymore
 - Dedicated **parameters.py** file
 - Modules split according to teams domain and interactions
 - Makefile: to validate that main suites can be built
- benefit in accessing the ecf flow API through a layer module (aka ecf.py):
 - Activator variable: enable/disable attributes **Trigger, Inlimit, Late**
 - Maintain the ability to load the suite on SMS (dynamic variable translation)
 - Intercept Variables/Triggers to identify where it is created/modified in complex suites
 - Add decorators (dedicated Label for operators)

Python error handling

- May raise Exceptions
 - missing key in dictionary,
 - Use assert
- A chance to detect issues earlier
- If your Python is incorrect, the error messages can be helpful for finding where and why it fails
- Navigate, walk, verify, validate the suite tree
 - It is not so obvious with shell suite definition
- ecflow has a built-in '**Job generation checker**' which can be run in advance.
 - It detects, for example, if .ecf job wrappers are missing, or if triggers are invalid.
- Suite **simulation mode**, to verify correct design

Python: Code Quality

- PyLint
 - Rates code
 - Enforces syntax
 - warns about large code: too many members, variables
- pep8 python style, PyChecker, PyFlakes
- coverage: identify dead code
- documentation: pydoc
- iPython: interactive interpreter, interactive documentation
- beware module dependencies (portability)



End Section

Migrating scripts to ecflow

For example migrating a cronjob

- Write simple suite with task controlled by “cron” or “time and repeat”
- Write wrapper file
 - containing header files and include your script

```
%include <head.h>
```

```
%includenopp <script.ksh>
```

```
%include <tail.h>
```

- Improve by splitting into logical units following guidelines
- Decide on ecflow variables vs included variables
- Separate into families carrying out related activities
- Separate based on criticality

Migration from a python script - a starting point

import script

```
script = """# Thanks https://en.wikipedia.org/wiki/Lorenz_system
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from mpl_toolkits.mplot3d import Axes3D

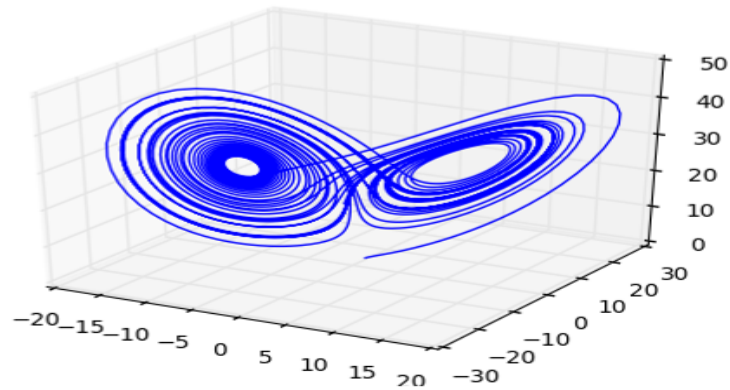
rho = 28.0
sigma = 10.0
beta = 8.0 / 3.0

def f(state,
t):
    x, y, z = state # unpack the state vector
    return sigma * (y - x), x * (rho - z) - y, x * y - beta * z # d

state0 = [1.0, 1.0, 1.0]
t = np.arange(0.0, 40.0, 0.01)

states = odeint(f, state0, t)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(states[:,0], states[:,1], states[:,2])
plt.show()"""
exec(script)
```



- An example where [jupyter](#) notebook helps

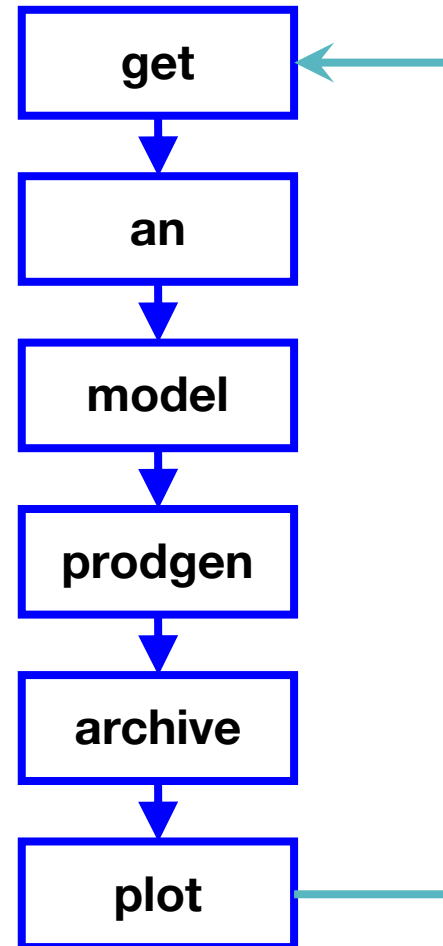
define suite, create task template

```
import sys
sys.path.append(os.getenv("HOME") + "/git/pyflow")
import os
import pyflow as p

home = os.getenv("HOME") + "/eflow_server"
user = os.getenv("USER")
with p.Suite("lorenz",
            ECF_HOME=home, ECF_INCLUDE=home, ECF_FILES=home, ECF_OUT=home,
            ECF_EXTN=".ecf", USER=user, SCHOST="localhost",
            ECF_JOB_CMD="/home/ma/emos/bin/trimurti %USER% %SCHOST% %ECF_JOB% %ECF_JOB%"):
    p.Defstatus("suspended")
    with p.Family("course2018"):
        with p.Task("compute"):
            p.Script("python <<@\n" + script + "\n@")
        with p.Family("multi"):
            for num in xrange(1,5):
                with p.Task("compute%d" % num):
                    p.Script("python <<@\n" +
                            script.replace("[1.0, 1.0, 1.0]",
                                           "[%d.0, %d.0, 1.0]" % (num, num)) + "\n@")
suite.deploy_suite(overwrite=True) # create task template files
suite.replace_on_server("localhost:2500") # replace the suite in the server
!ecflow_client --port 2500 --begin /lorenz
!ecflow_client --port 2500 --resume /lorenz
```

Designing a suite - a simple NWP example

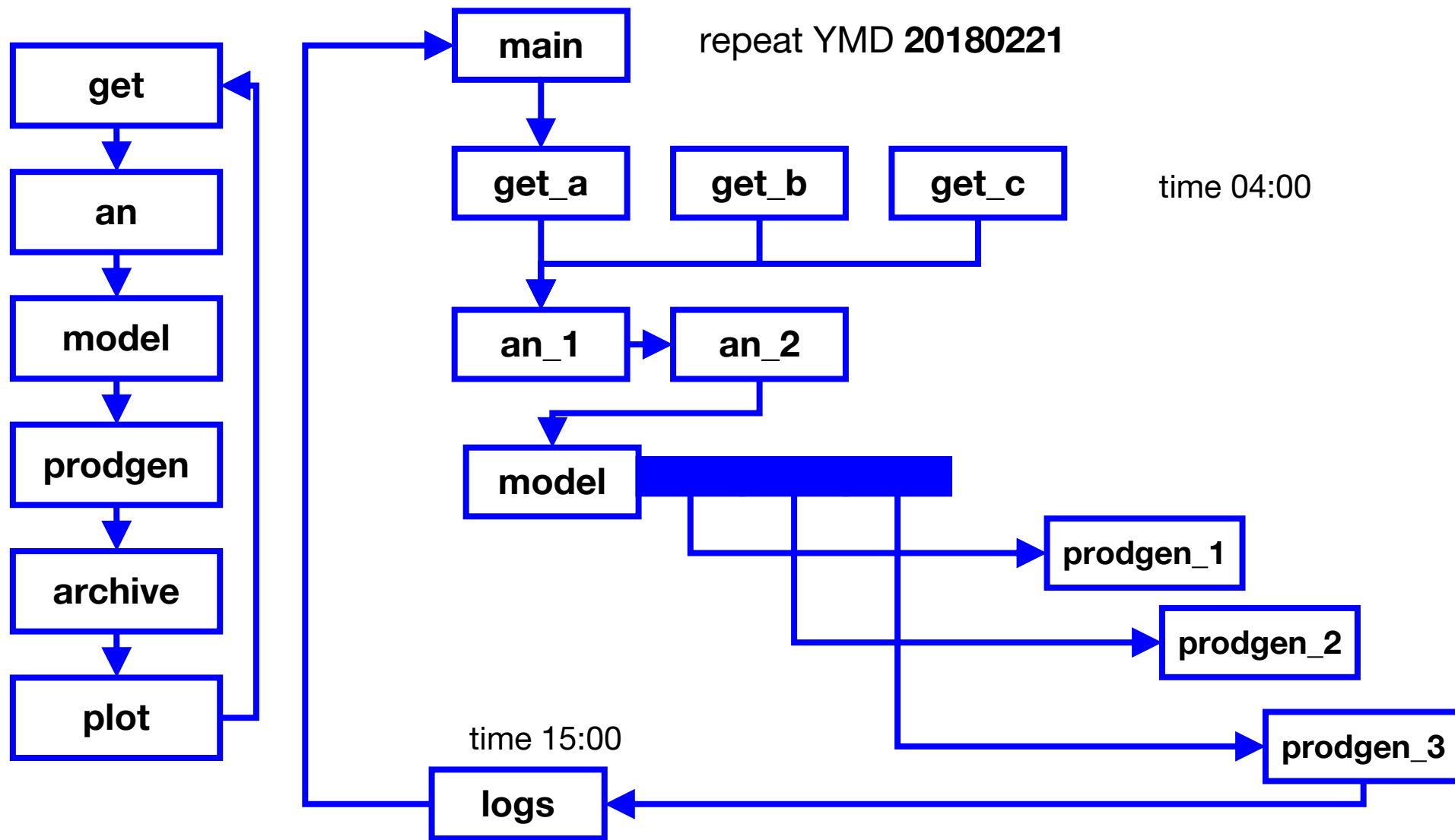
- 1) Get data
- 2) Analysis
- 3) Model
- 4) Products
- 5) Archive data
- 6) Plots



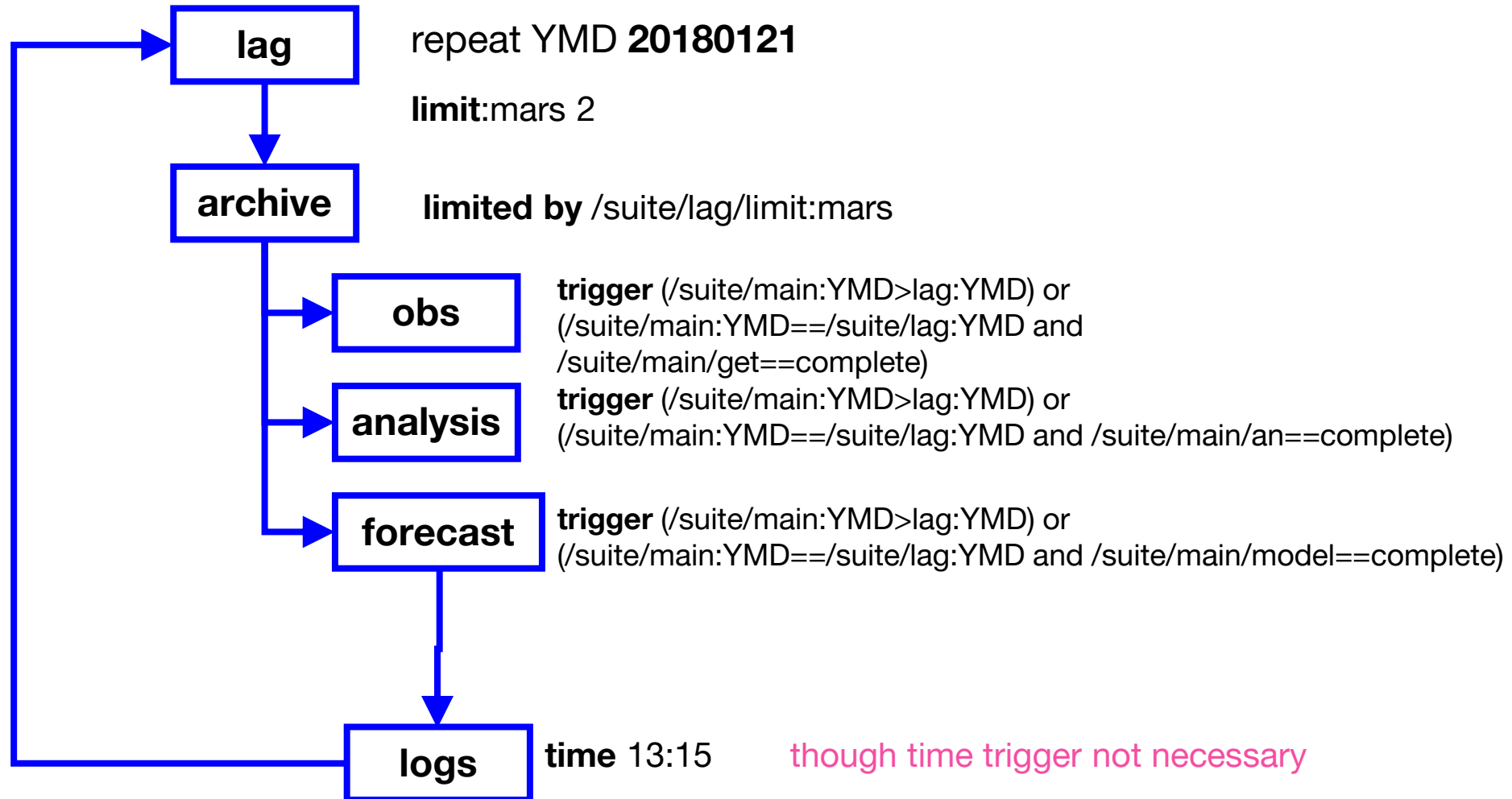
Designing an operational suite - considerations

- Critical path - minimise dependencies systems/file systems
- Documentation - man pages for suites/families/tasks
- Rerunnabilty of tasks
- Simplicity - KISS
- Keep runtimes under control
- Keep logfiles for support/optimisation
- Make/rebuild within suite plus admin tasks
- Allow for simple switching of systems
- Clean up

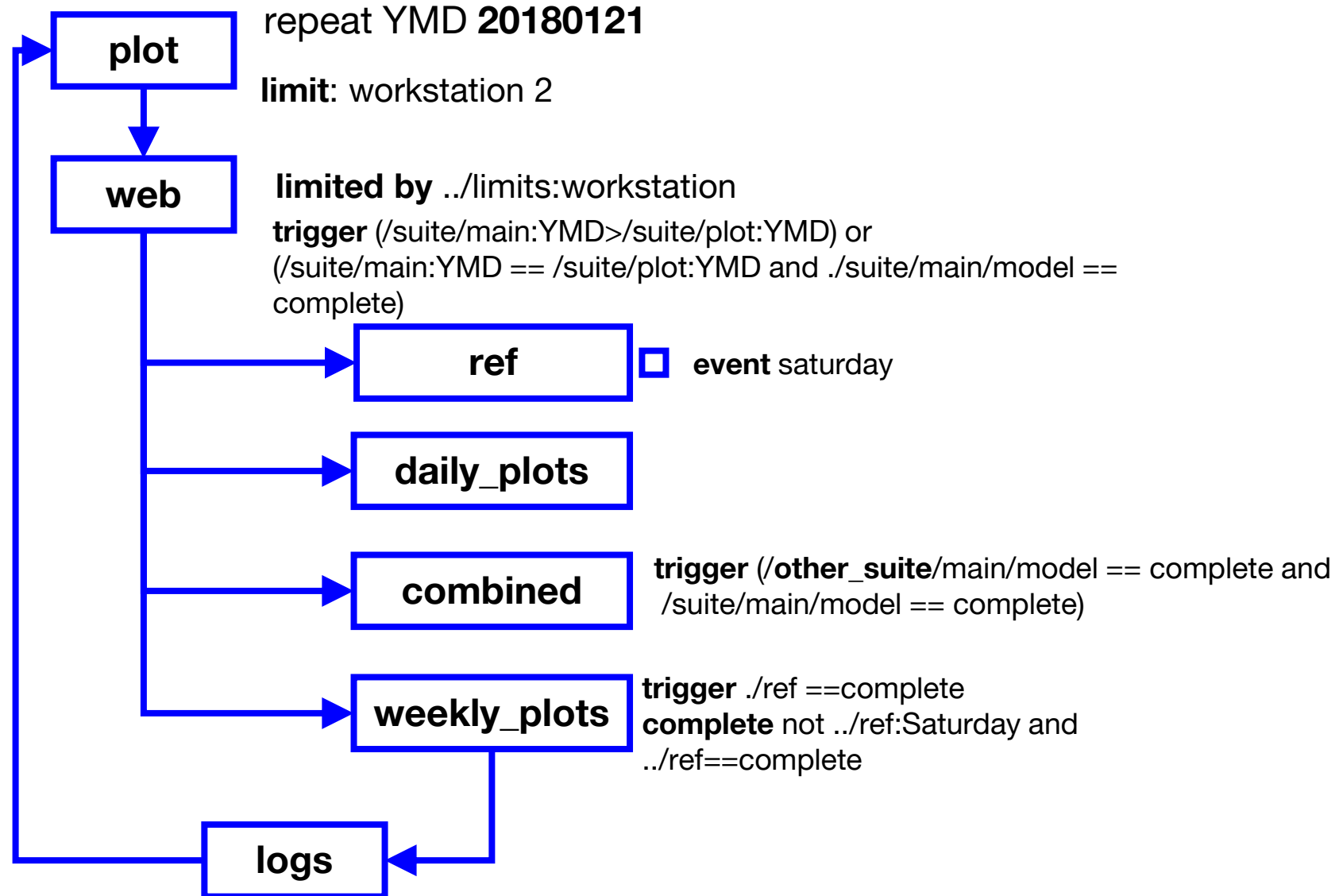
Designing an operational suite - critical path



Designing an operational suite - archiving



Designing an operational suite - plotting



Writing “operational” scripts - considerations for critical tasks

- Re-runnability
- Look after critical data - HA systems, backups
- Limit number of languages used
- Be careful with error trapping
- All variables should to be set (use default values %VAR:1%)
- Use a generic user - identify operations
- Works on multiple systems
 - ECF_JOB_CMD
- Design based on constraints
 - Staff availability
- Avoid accessing off-line data in critical path
- Avoid NFS mounted files or unsafe file-systems (SCRATCH)
- Tasks can be serial or parallel
 - don't do serial things in parallel tasks
- Use generic directories to simplify cleaning and always clean up!
- Check task runtimes
- Keep output and job files
- Always use a CM system and test
 - Test ecflow server/suites

Monitoring operational suites

- GUI - our operators do not view completed or queued tasks
 - Only submitted, active, aborted tasks
- Colours give clear indication of suite status
 - Pop up windows
 - man pages and output
- Task colours give clear indication of task status (configurable!)
 - Submitted for too long can indicate resource problems
- “Late” warnings are useful: submitted, active or complete
- Check tasks are also useful - schedule, tasks running, feeds

Implementing suites

- A definition file can hold both **operational** and **test** versions of suites
- Use conditional statements in suite definition to modify behaviour
 - `if SUITE == "oper_suite": PRODGEN = 1`
 - `elif "test_" in SUITE: PRODGEN = 0`
 - `PRODGEN = not "test_" in SUITE`
- Use variables to distinguish versions and behaviour
 - `if not PRODGEN: task.add(Defstatus("complete"))`
 - `task.add(If(not PRODGEN, Defstatus("complete")))`
- Suite can be **loaded** on a test ecf flow server and **plugged** into an operational server
- One script for suite definition: `import suite # suite.py`
 - suite expansion: `defs.save_as("suite.exp")`
 - load: `if __name__ == "__main__": client.load("suite_name")`

Suite design: functional aspects

- **group time dependencies** in dedicated families + triggers
 - easy replacement when schedule changes
 - defstatus complete in not-real-time-mode
- **group external trigger** dependencies in dedicated families (dummy tasks)
 - easily replaced if reference suite changes
 - can be set defstatus complete in standalone-mode
- **'umbrella triggers'** to prevent evaluating multiple triggers all day long
 - 80-90 triggers for products generation depending on model meter

Similarities to parallel programming (1/2)

- ecfLOW as a **central** point:
 - Collect-Share information
 - Reporting status
 - Re-Routing
 - Retrieving job information
- ecfLOW as **distributed** fleet: inter-server cooperation
 - Maintaining work during server and network outages
 - Handling of priorities, systems, tests
 - Sharing load
 - Sync suite: client to mirror status/variables

Similarities to parallel programming (2/2)

- ecflow allows you to handle:
 - **Deadlocks** - broken fluidity
 - **Livelocks** - wasting resources
 - Mutual exclusion (events as **mutex**)
 - Semaphores (limits, hardware, software)
- Allows profiling with timeline
- Works in “soft” real-time (**ECF_INTERVAL** is 60 seconds)

ECMWF Projects: Background

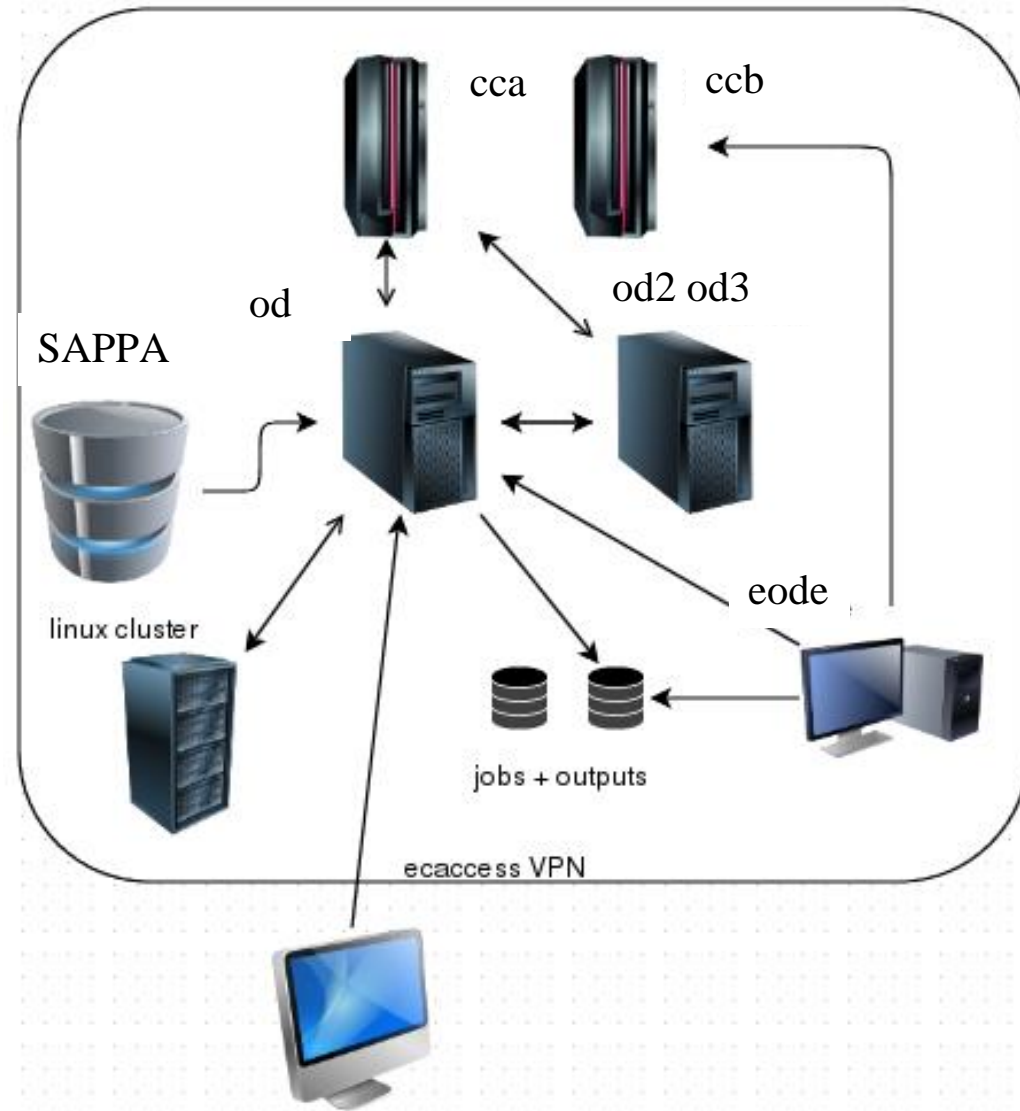
- ECMWF code runs on multiple platforms
- Software installation should be simultaneous across them all
 - Need ability to quickly revert changes if problems
- Need automated routine maintenance
- Need to handle both operational and non-operational tasks
- Numerous housekeeping tasks

Operational Systems

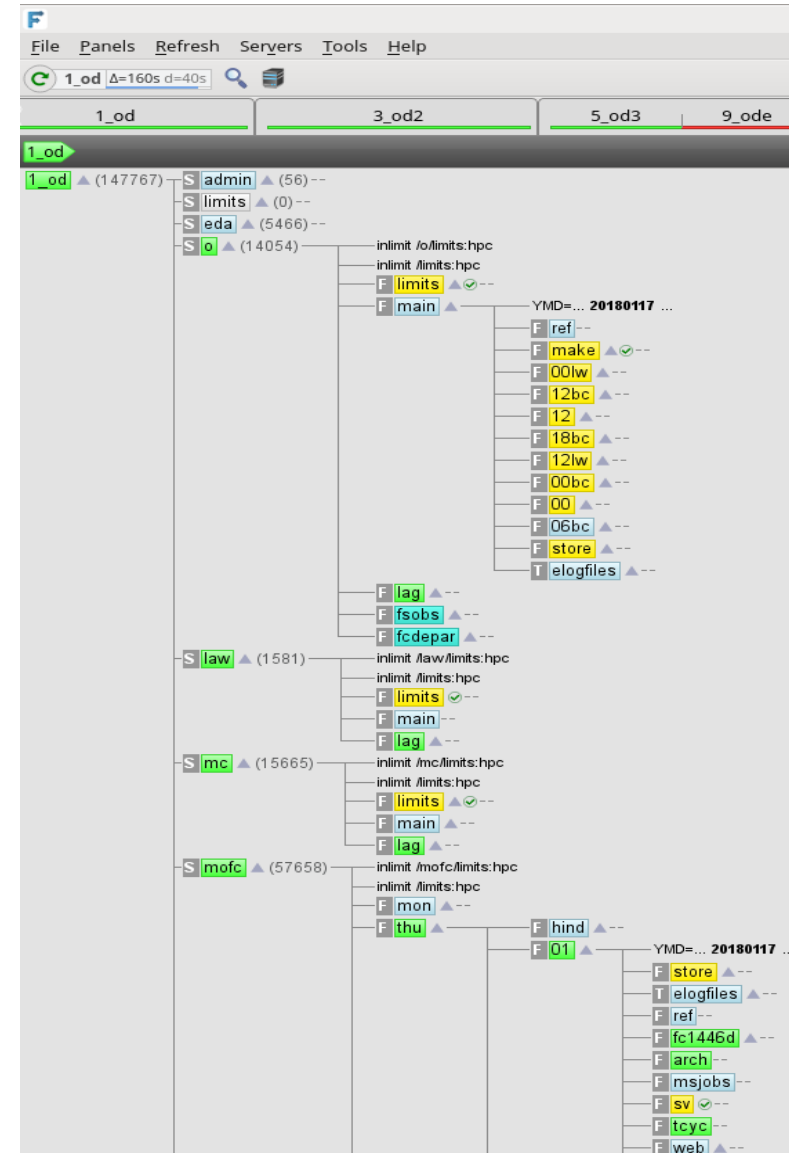
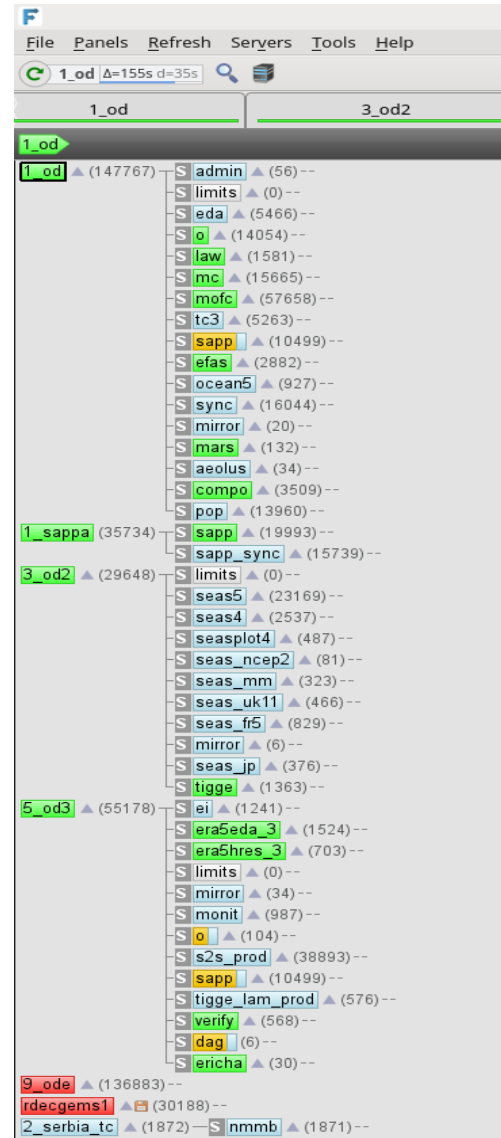
- Operationally we will run dozens of suites, tens of thousands of tasks
- Number of servers reflecting criticality
 - ode: tests and design mode
 - od3: official e-suites monitored by Operators, special projects
 - od2: higher criticality, seasonal suites
 - od: operational suites looping daily
- Servers hosted on linux workstation in Ops-room (with UPS), VM, or WS
- Access controlled
- Heterogeneous: tasks run on HPC, Linux Clusters, locally
- Suite structure separated by criticality: main-crit-lag-pop families
- Operators monitoring
- Watchdog tasks both internal and external to suites
 - Operators/Analysts “like” red boxes

Operational System: SMS/ecflow server

- Server is target agnostic:
 - ECF_JOB_CMD (submit)
 - ECF_KILL_CMD (kill)
 - ECF_STATUS_CMD (query)
- Variable to locate wrapper files:
 - ECF_FILES
- Variable to locate header files:
 - ECF_INCLUDE
- Checkpoint files:
 - Written /2min, back /4min
 - Duplicated /10min,
 - stored /30min
- Cluster, host, storage host switch



Operational System: Servers/suites



Contact Points:

- Axel BONET `axel.bonet@ecmwf.int`
- John HODKINSON `john.hodkinson@ecmwf.int`
- Avi BAHRA `avi.bahra@ecmwf.int`
- Blazej KRZEMINSKI `blazej.krzeminski@ecmwf.int`

- Problems/Requests:
 - ECMWF Software Support software.support@ecmwf.int

End Section

ecflowUI

- interface based on Qt rather than Motif
 - allows for faster development of features
 - tabbed interface, multiple windows allowed
 - each can have different servers and can have any node as its root (e.g. a suite or family)
- tree view will be familiar to existing users, but more accessible to new users
- table-based view provides a flat representation of the tree
 - enables sorting, e.g. by status
- client/server communication is thread-based
 - user interface should not hang when waiting for a server response

Practical Sessions

- URL: <https://software.ecmwf.int/wiki/display/ecflow/Introduction>

Additional slides

Debugging: an overview

- When playing definition file
 - Check first on test server, Python debugger pdb
- When submitting look at ecflow log (or history via GUI) for info
- Can you see the script in ecFlow_ui?
 - No - unknown file location **ECF_FILES** (Python has job checking option)
- Can you edit and pre-process the script?
 - Cannot find includes (**ECF_INCLUDE**)
 - Missing variable (check log or history) or misleading use of **ECF_MICRO %**
- Script stays submitted/active? Syntax error, trapping issue.
 - Submission problem, child process access or header problem. Try submitting job from command line with **NOECF=1**? Output directory does not exist (**ECF_HOME**, **ECF_OUT**), queuing system holding job

Debugging

- Task aborts?
 - Script problem, check output
- No output after task aborts
 - Mount point not available, ECF_OUT wrong, log server problem
- Task remains active
 - Error is not trapped
 - Internal ssh call
 - Remote system crash

Server Configuration

- Server configuration variables:

```
ECF_HOME           # server admin directory
ECF_PORT           # port number
ECF_CHECK          # checkpoint file name
ECF_CHECKOLD       # backup checkpoint file name
ECF_LOG            # server log file name
ECF_CHECKINTERVAL # [120], 600 sec
ECF_LISTS          # white list file name
ECF_DEBUG_SERVER  # turn on debug mode
```

- Server log file:

- Can be handled by client command
- `ecflow_client --port 3141 --log=new` # [new|clear|flush]

Key ecflow variables

- ECF_HOME, ECF_FILES, ECF_INCLUDE : input scripts

- ECF_HOME (ECF_OUT): job files, (remote) output

- Mandatory variables for jobs

ECF_HOST	# server hostname
ECF_PORT	# server port
ECF_NAME	# task path
ECF_PASS	# pseudorandom unique identifier

- Useful variable for jobs

ECF_TRYNO	# job occurrence number
ECF_HOSTFILE	# alternative host server list (server recovery)
ECF_RID	# job remote id (queuing id)
ECF_TIMEOUT	# interval between two attempts
ECF_DENIED	# to enable job exit with error before 24h
NO_ECF	# standalone mode (set to use)

Similarities with SMS

- Functionality is very similar
 - Suites, Family, Task, Variables, Trigger, Time, Date, Late, etc
 - Child commands: init, complete, event, meter, label, wait, abort
 - Variable inheritance
- Scripts are similar
 - file name extension: .sms -> .ecf (ECF_EXTN)
 - SMS variables replaced with ECF, i.e. SMSHOME -> ECF_HOME
- GUI: ecflowview was ported from XCdp to facilitate transition,
 - About to be changed

Differences with SMS (1)

- Maintenance and enhancement of Client/Server easier
 - Built from the ground up in C++
 - Design Patterns, Observer, Template, Singleton, etc
 - Test Driven, large set of regression tests
- SMS provided a custom scripting language, ecflow provides Python integration, that allows:
 - Building of the suite definition
 - Client-Server communication
- Not restricted to Python, can use shell level interface
- Published format, any language for generating the suite definition

Differences with SMS (2)

- Improved Error Checking for:
 - Trigger Expressions
 - Validation of externs in Trigger expression
 - Earlier checking for job generation
 - Checks for recursive includes
 - Simulation with out the need for scripts or server
- Customisable handling of zombies
- When a task is aborted, a reason can be provided
- No explicit login

Migration from SMS to ecflow

- Definition files
- Header files
- Script wrappers
- Queuing system directives
 - # QSUB -o <output file>
- Associated scripts
 - ecf_submit, ecf_kill, ecf_status
 - trimurti
- Embedded dependency in applications
 - IFS (meter), mars (label and events)

Migration of definition files

- In CDP

```
get  
show /suite >suite.exp"
```

- Outputs an expanded definition file

```
suite test  
    edit SMSHOME "$HOME/course"  
    edit SMSINCLUDE "$HOME/course/include"  
    edit SMSFILES "$HOME/course/smsfiles"  
    task t1  
    .....  
endsuite
```


Migration of definition files

- Convert variables to produce text based ecflow suite (see [Key ecflow variables](#))

```
sed -f ~/map/bin/sms2ecf/sms2ecf-min.sed < suite.def > ecflow.def
```

```
suite test
```

```
    edit ECF_HOME    $HOME/course
```

```
    edit ECF_INCLUDE $HOME/course/include
```

```
    edit ECF_FILES   $HOME/course/smsfiles
```

```
    task t1
```

```
    .....
```

```
endsuite
```

Migration of definition files

- This could be all you require, especially for small simple suites
- However....
- For more complex suites we recommend you use this as target for generation of a python suite
 - Easier to maintain
 - Testable
 - Much easier to debug

Migration of header files

- trap.h, endt.h
 - smsinit, smsabort, smscomplete **replaced with** ecflow_client commands
 - SMS variables **replaced with** ECF_ variables
- qsub.h (ECMWF specific generic queuing commands)
 - queuing system directive replaced before job submission,
 - ecf_submit (ECF_JOB_CMD)
- smsmeter, smsevent, smslabel:
 - replace with ecflow_client commands

Migration: explicit CDP calls

- Replace with `ecflow_client` commands

```
cdp <<EOF
set SMS_PROG $SMS_PROG; login -t 60 $SMSNODE $USER 1
if (rc eq 0) then; exit 1; endif
alter -V $SMSNAME:BASEDATE $BASEDATE

# force set $SMSNAME:1
# force complete $SMSNAME
exit 0
EOF
```

- Becomes

```
ecflow_client --alter change variable BASEDATE $BASEDATE $ECF_NAME
# ecflow_client --alter change event 1 set $ECF_NAME
# ecflow_client --force complete $ECF_NAME
```

Migration: wrapper files

- Replace occurrences of %SMS% variables
 - %SMSTRYNO%, %SMSJOBOUT%
- Replace cdp calls with ecf_flow_client equivalents

or

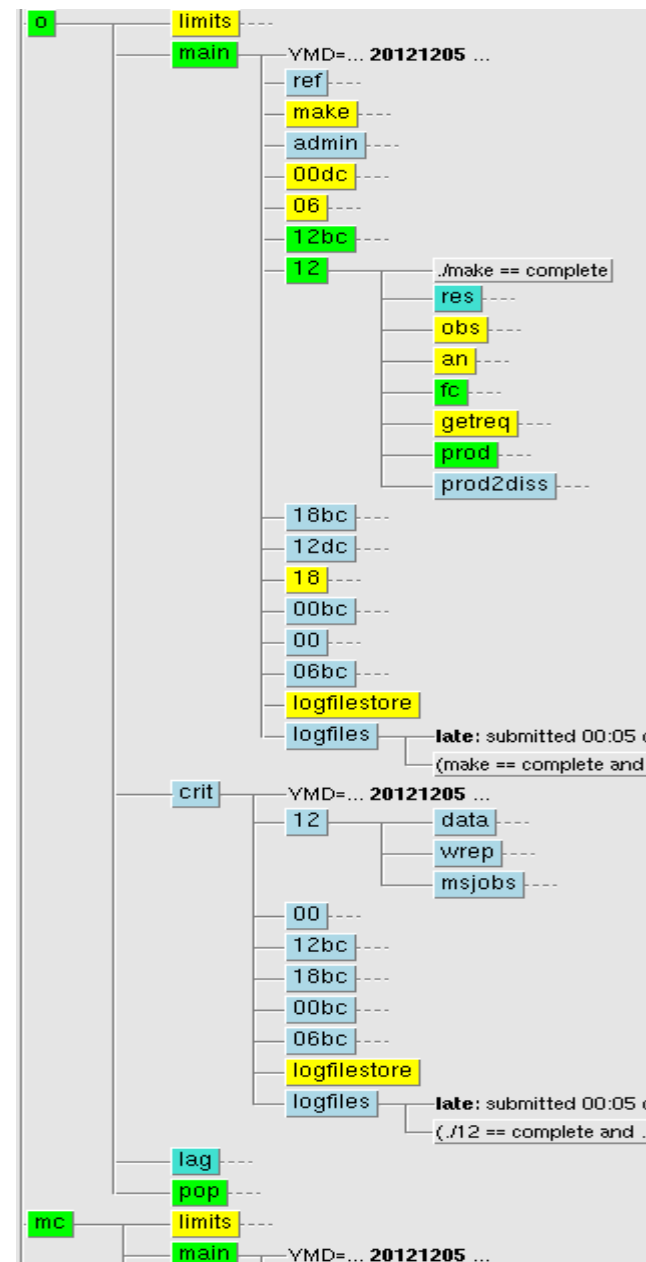
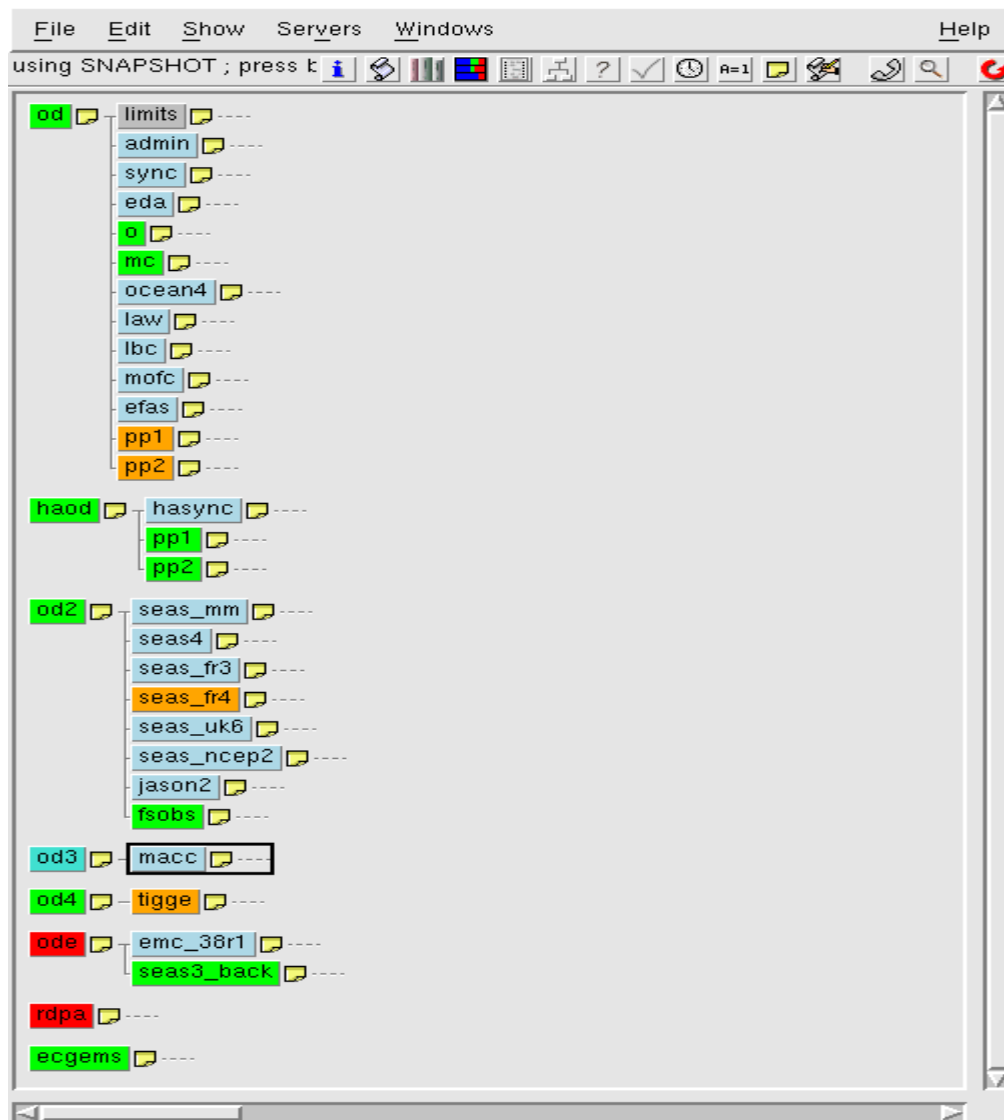
- suite design can remove some embedded CDP
 - cdp call to force complete
 - replace with event /complete combination in the definition file
- Migrate child commands

End Section

Interface design

- nodes: Client, Defs, Suite, Family, Task, Alias
- states
- attributes
 - Autocancel Defstatus Late
 - Repeat Edit Event Meter Label Limit Inlimit
 - Trigger Complete Date Time Cron Today
- actions
 - server: Check History Suites 'Time line' Variables Zombies Options
- nodes
 - script Manual Job Output Edit
 - info Messages 'Time line' Triggers Variables Why?
 - check Jobstatus Execute Requeue Force

Operation System: servers - suites



Relationship between .def, .ecf and .job files

