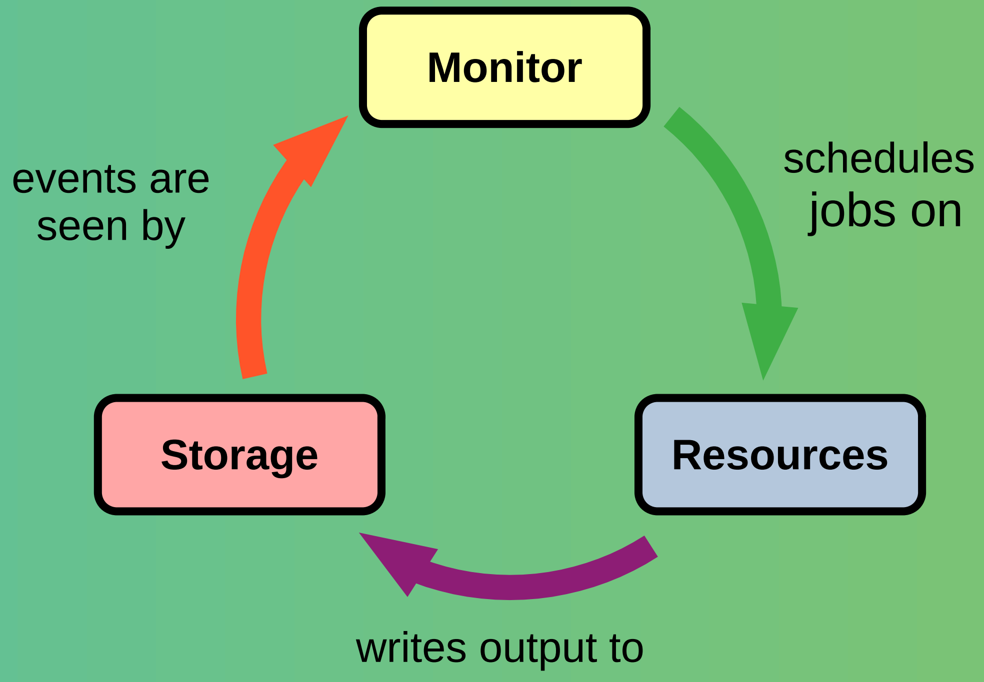
Delivering Rules Based Workflows for Science

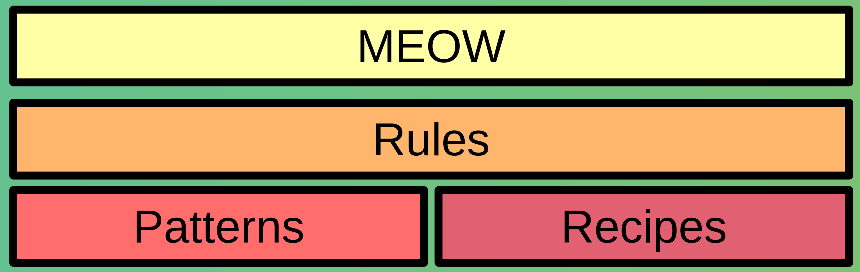
Presenting: David Marchant,
Contributors: Mark Blomqvist, Philip Shun B. Jensen, Iben Lilholm, Martin Nøregard

Part I: Managing Event Oriented Workflows

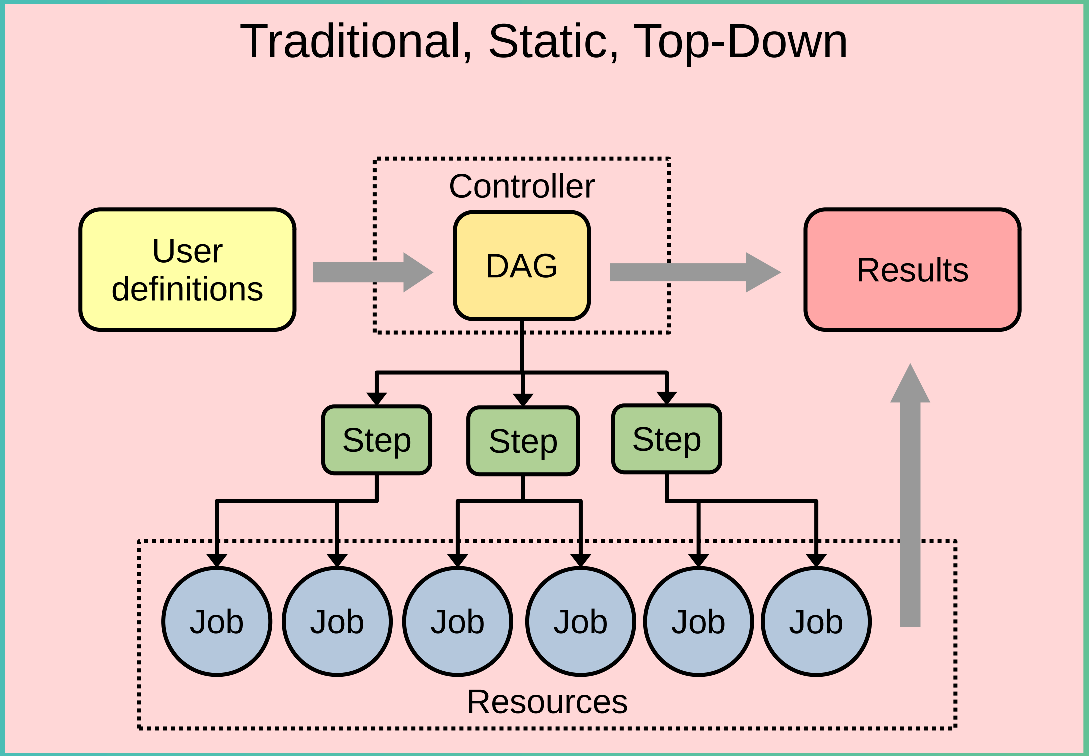
MEOW

- Managing Event Oriented Workflows
- Rules-based system for isolated job scheduling
- Composed of Patterns and Recipes
- Workflow structure can be altered by adding, canceling or modifying jobs or monitoring structures
- ‘Assessing Events for Scheduling in Heterogeneous Systems’
- Presented at Works ‘22. Seemed well received, but some suggestions/questions

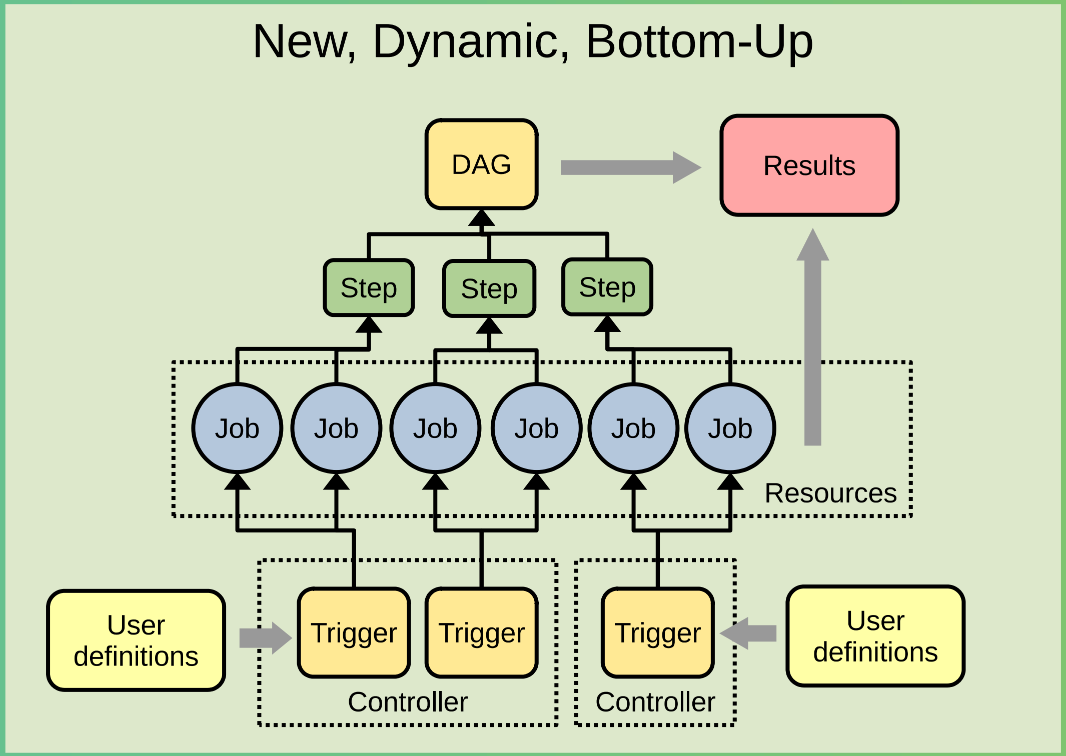




Top-down vs Bottom-up

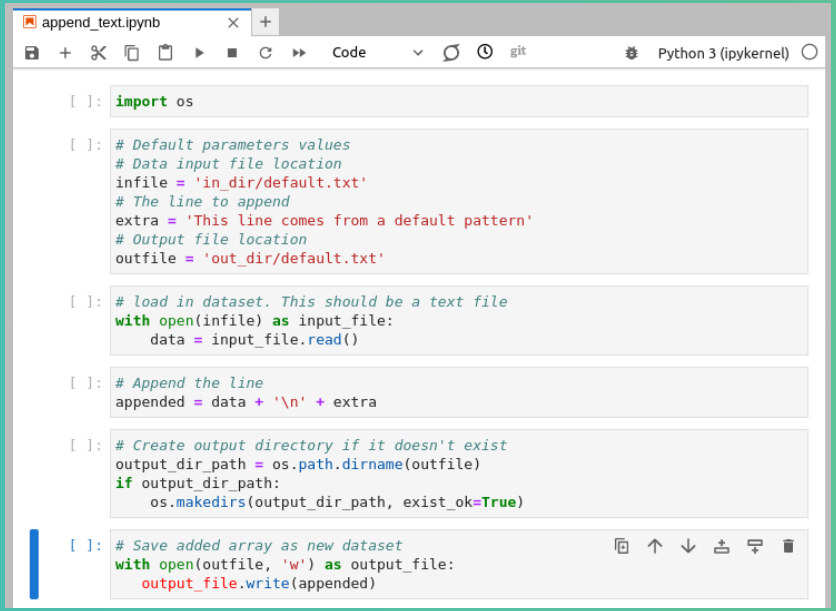


VS



mig_meow

- Python library for building MEOW objects
- Users define *Recipes* (the code to run) … or *Patterns* (The conditions when to run)




```
input_file: infile
input_paths:
- initial_data/*
output:
  outfile: '{VGRID}/int_1/{FILENAME}'
parameterize_over: {}
recipes:
- append_text
variables:
  extra: This line is overridden
```

- Together these form a *Rule* (Scheduling in response to events)
- But is very tied to the MiG, Jupyter Notebooks, file events etc.

Project Aims

- Create a truly stand-alone framework for rules based scheduling
- Allow for integration with existing scheduling frameworks
- Solve issue of identifying arbitrary job results
- Provide scientific use case

Part II: A Generic Framework for MEOW

meow_base

- Standalone framework for constructing MEOW systems
- Written in Python, but designed to run analysis in any language (pending support)
- Still uses same Pattern, Recipe, Rule definitions as before
- Provides MEOWRunner, to orchestrate complete workflow lifetime
- Breaks job functionality down into different components

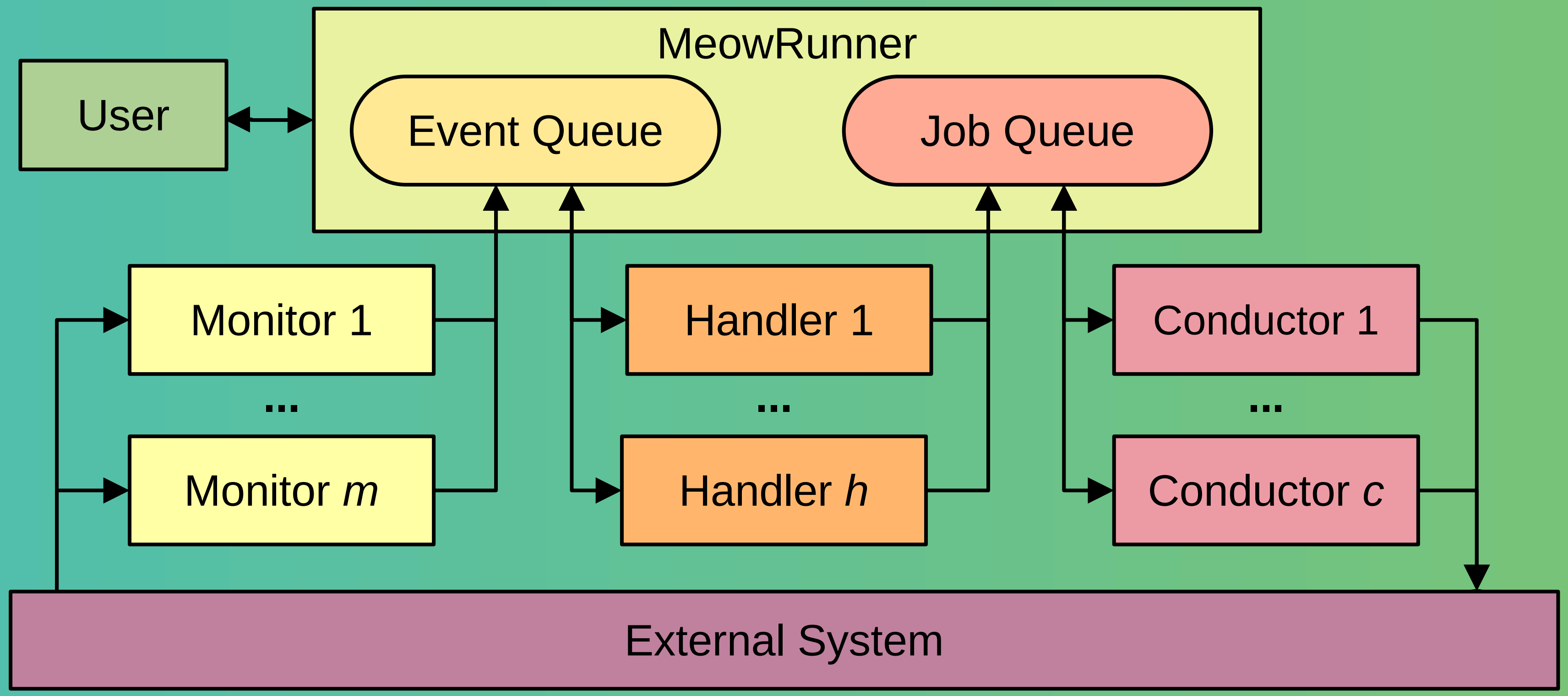
Base Components

- Abstract base components for Patterns, Recipes, Monitors, Handlers, Conductors

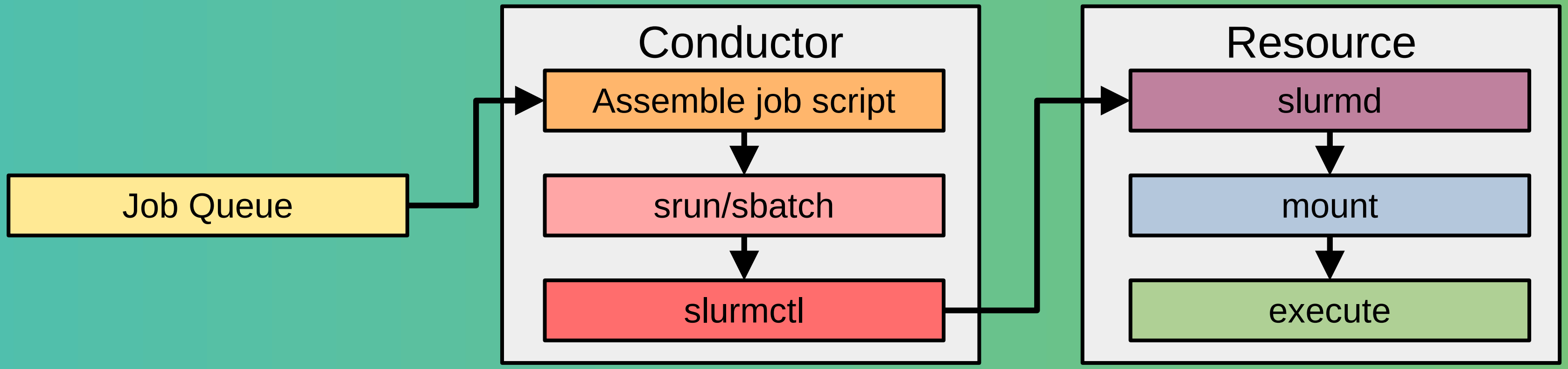
- Example implementations for each, providing functionality for file and network events, and processing Python or Bash based jobs.

BaseRecipe
name:str recipe:Any parameters:Dict[str, Any] requirements:Dict[str, Any]
__init__(self, name:str, recipe:Any, parameters:Dict[str,Any]={}, requirements:Dict[str,Any]={}) __new__(cls, *args, **kwargs) _is_valid_name(self, name:str) → None _is_valid_recipe(self, recipe:Any) → None _is_valid_parameters(self, parameters:Any) → None _is_valid_requirements(self, requirements:Any)->None

MeowRunner


MeowRunner
User
Event Queue
Job Queue
Monitor 1
...
Monitor *m*
Handler 1
...
Handler *h*
Conductor 1
...
Conductor *c*
External System

Integration with Slurm and SSH



- Slurm is a common system for orchestrating jobs on HPC resources
- meow_base includes options in BaseConductor for integrating with a locally hosted slurmCtl daemon
- Jobs automatically setup to be compatible with MEOW file event handling

Part III: Identifying Arbitrary Outputs

meow_base as a SWMS

- MEOW was first intended as a tool for scientific workflows
- Most features expected of Scientific Workflow Management Systems are already present
- Provenance reporting is lacking though. Main issue is MEOW jobs do not need to specify outputs

Tools to Identify Outputs

- Investigated 4 potential tools
- Each traces file events
- Assumed that if output was never written, it could be ignored
- Strace is the only tool that meets our needs

	strace	perf	inotify	fanotify
Observes File events	x	x	x	x
Provides event PID	x	x		x
Provides event path	x		x	x
Monitor whole filesystem	x	x		x
Avoids race conditions	x	x	x	
Does not require root	x		x	
Observes through Mounts	x	x	x	x

Tool feature summary

Tool Overheads

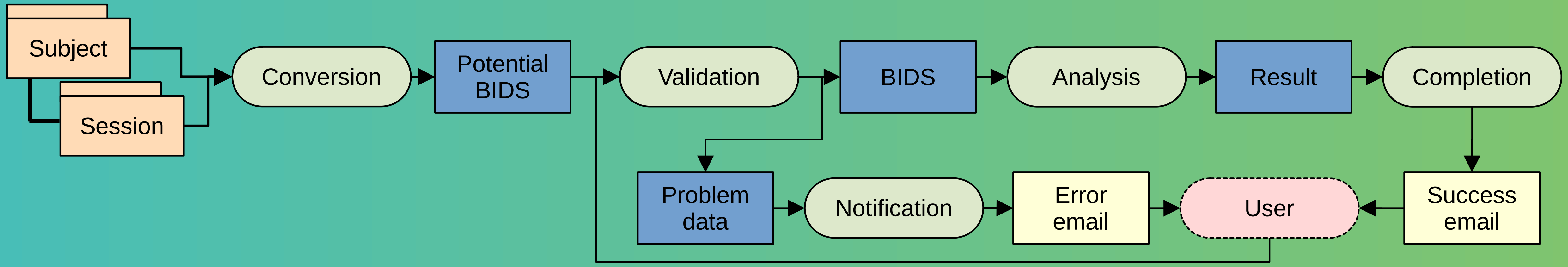
- Tested in with scripts that spam create and delete events. Designed to show ‘worst use case’
- Also with scientific analysis. Designed to show ‘realistic use case’
- Strace is slow, but others can’t be used without caveats

	strace	perf	inotify	fanotify
Bash Script	x5.49	x5.46	x1.03	x1.03
Python Script	x4.58	x1.12	x1.16	x1.18
Analysis with Generation	x3.04	x1.05	x1.04	x1.05
Analysis without Generation	x1.49	x1.05	x1.00	x1.01

Tool slowdowns. All slowdowns shown relative to their respective test, run without the tool

Part IV: A Scientific Example

Converting to BIDs format



- Automatic conversion of brain imaging data into new standard, BIDS
- Highly repeatable, but needs human touch periodically
- Large existing datasets need updating

Setting up Patterns and Recipes

- Setup consists of writing recipe files (standard Python, bash or Jupyter scripts are natively supported)
- Patterns are assembled as objects as shown
- Only one pattern and recipe shown here

```
# Automatic conversion of bids data
p_convert = FileEventPattern(
    "conversion_pattern",
    os.path.join(raw_dir, "*", "*", "*"),
    "conversion_recipe",
    "input_base",
    parameters={
        "output_base": "meow_bids/meow/validating"),
    },
    event_mask=[
        DIR_CREATE_EVENT,
        DIR_MODIFY_EVENT,
        DIR_RETROACTIVE_EVENT
    ]
)

r_convert = BashRecipe(
    "conversion_recipe",
    read_file_lines("recipes/conversion.sh")
)
```

Assemble them into a dictionary

- Create a collection of all Patterns and Recipes
- Note the use of provided meow_base helper functions to ensure easy compatibility

```
patterns = assemble_patterns_dict(
    [
        p_convert,
        p_validate,
        p_notify,
        p_analysis,
        p_complete,
    ]
)

recipes = assemble_recipes_dict(
    [
        r_convert,
        r_validate,
        r_notify,
        r_analysis
    ]
)
```

Create the Runner from Components

- Runner is created by combining at least one Monitor, Handler and Conductor
- Usable examples of each included in meow_base, along with appropriate Patterns and Recipes
- Once started will run robustly until stopped by the user

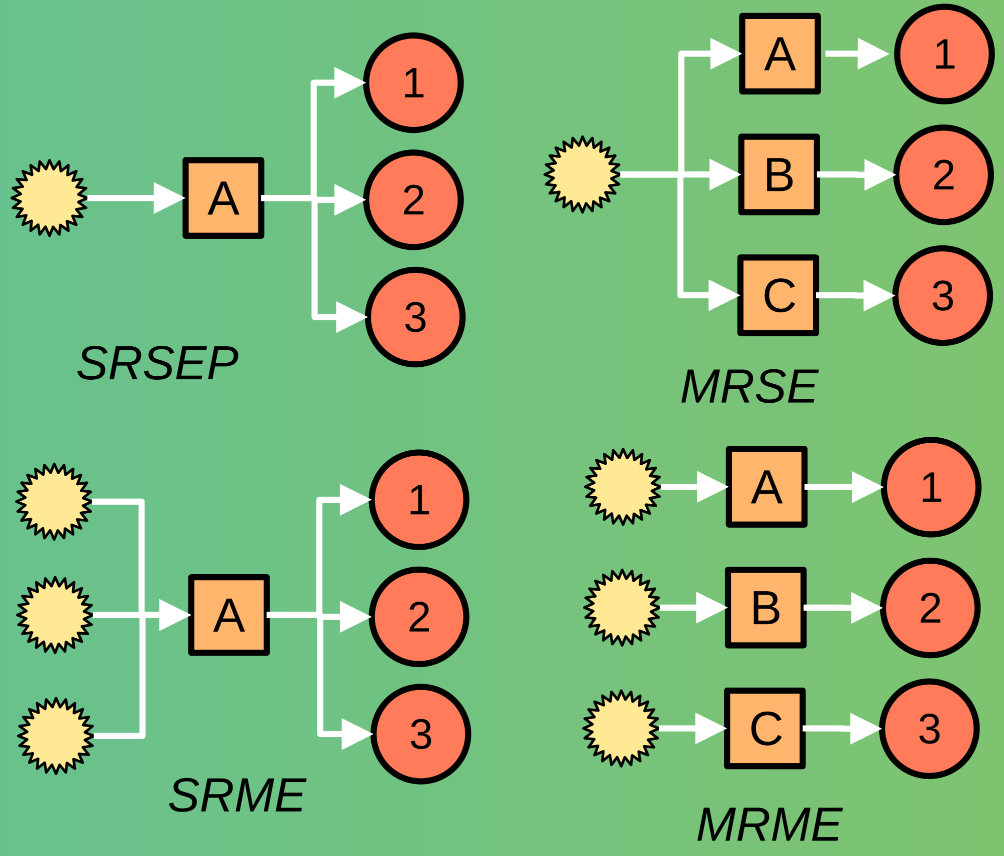
```
# The actual runner, that will conduct all scheduling and analysis
runner = MeowRunner(
    WatchdogMonitor(
        base_dir,
        patterns,
        recipes,
        # This can be set to 0 to turn off logging
        logging=3
    ),
    BashHandler(
        pause_time=1
    ),
    LocalBashConductor(
        pause_time=1,
        notification_email="alert@localhost",
        notification_email_smtp="localhost:1025"
    )
)

runner.start()
```

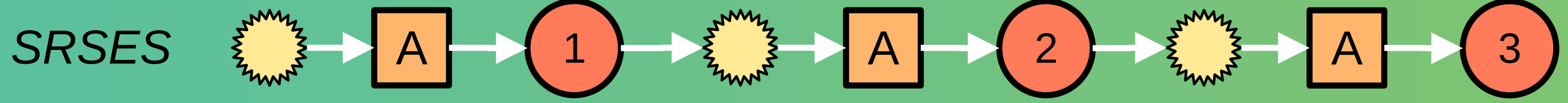
Part V: Performance Benchmarks

meow_base Performance tests

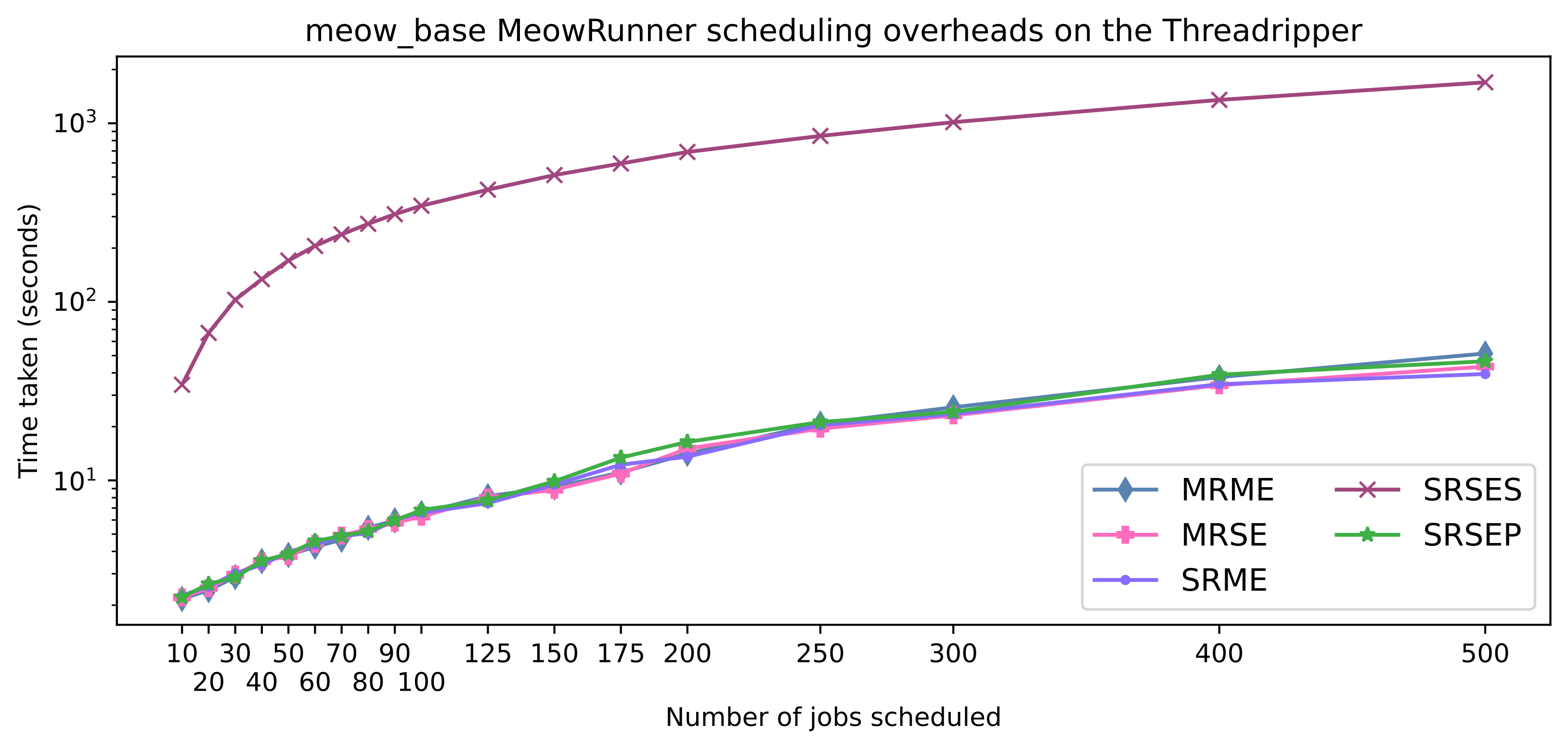
- Same overheads as previous MEOW systems

- Single Rule Single Event Parrallel (SRSEP)
- Multiple Rules Single Events(MRSE)
- Single Rule Multiple Events (SRME)
- Multiple Rules Multiple Events (MRME)
- Single Rule Single Event Sequential (SRSES)



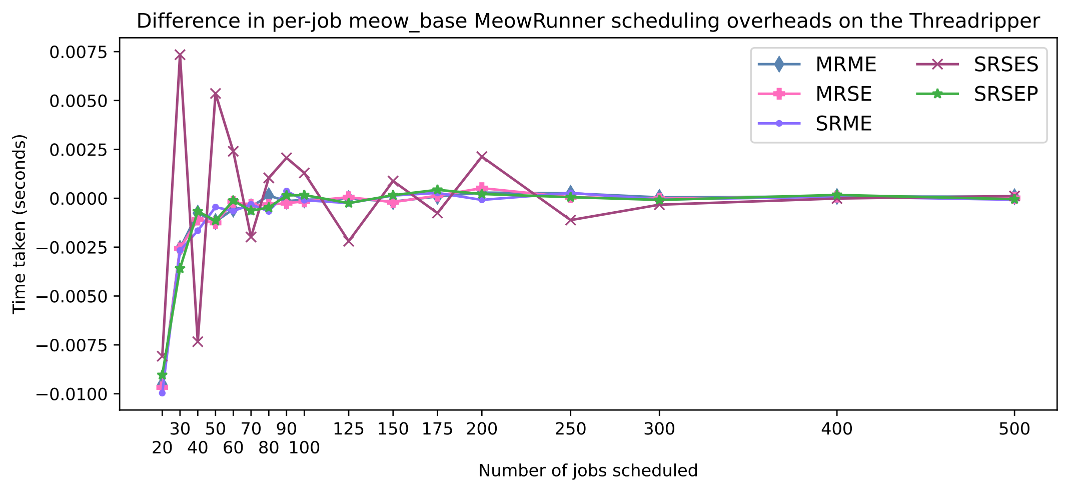


meow_base Performance



meow_base Performance

- Scales well (at least as far as has been rigorously tested)
- Generally slower than barebones mig_meow implementation, but faster than full MiG implementation
- Per job processing time is both small, and scalable
- Sequential is, as always, terrible. Comes from including job execution and all that entails



	10	100	500	mean
SRME	0.23s	0.066s	0.079s	0.086s
MRSE	0.22s	0.063s	0.087s	0.086s
MRME	0.27s	0.066s	0.10s	0.087s
SRSEP	0.22s	0.068s	0.093s	0.089s
SRSES	3.43s	3.45s	3.40s	3.41s

meow_base per job overheads

Part VI: Conclusions

MEOW Workflows as a Basis for Science

- meow_base is a more generic framework for rules based scientific workflows
- Available now as a standalone tool, or as a basis for further implementations
- https://pypi.org/project/meow-base/
- Novel scientific workflow structures have been demonstrated
- Arbitrary outputs can be identified, but a more efficient solution is needed

Thank you for listening