Framework for Extensible, Asynchronous Task Scheduling (FEATS) in Fortran

Brad Richardson, Damian Rouson, Harris Snyder and Robert Singleterry

archaeologic

Agenda

- Motivations
- Implementation Details
- Example/Demo Applications
- Conclusions

archaeologic

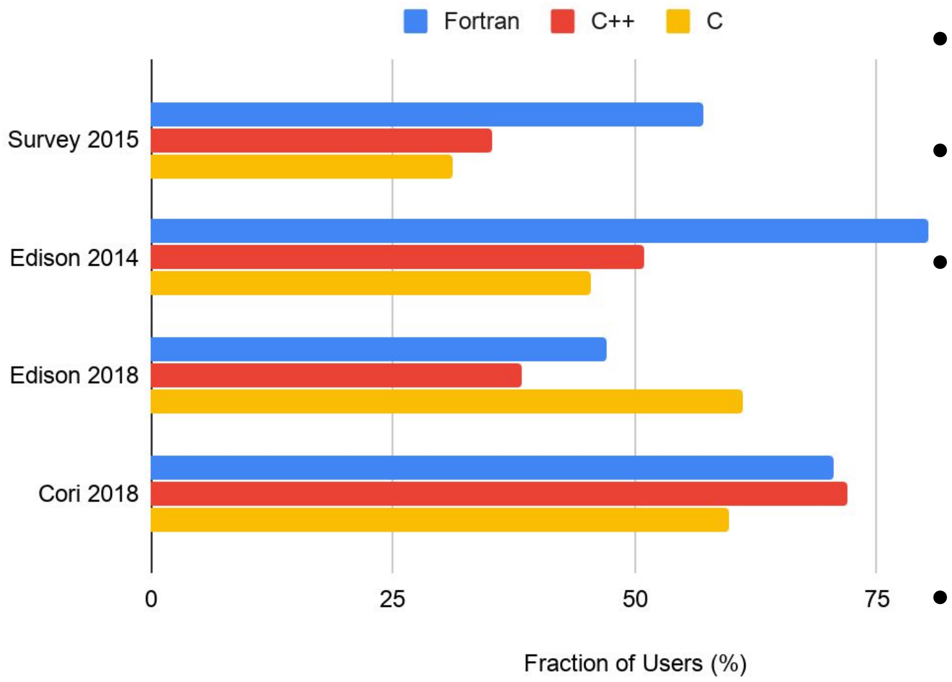
Motivations

- Target an under-served user base: Fortran
  - Enable scientists and engineers to develop efficient applications for HPC beyond the “embarrassingly parallel” problems
- Explore the native parallel features of Fortran
  - Don’t force “reformulating” the problem to be able to interoperate with C/C++ or other external libraries

Motivations

Compiled languages used at NERSC

NERSC



Totals exceed 100% because some users rely on multiple languages.

- Fortran remains a common language for scientific computation.
- Noteworthy increases in C++ and multi-language
- Language use inferred from runtime libraries recorded by ALTD. (previous analysis used survey data)
  - ALTD-based results are mostly in line with survey data.
  - No change in language ranking
  - Survey underrepresented Fortran use.
- Nearly ¼ of jobs use Python.

BERKELEY LAB

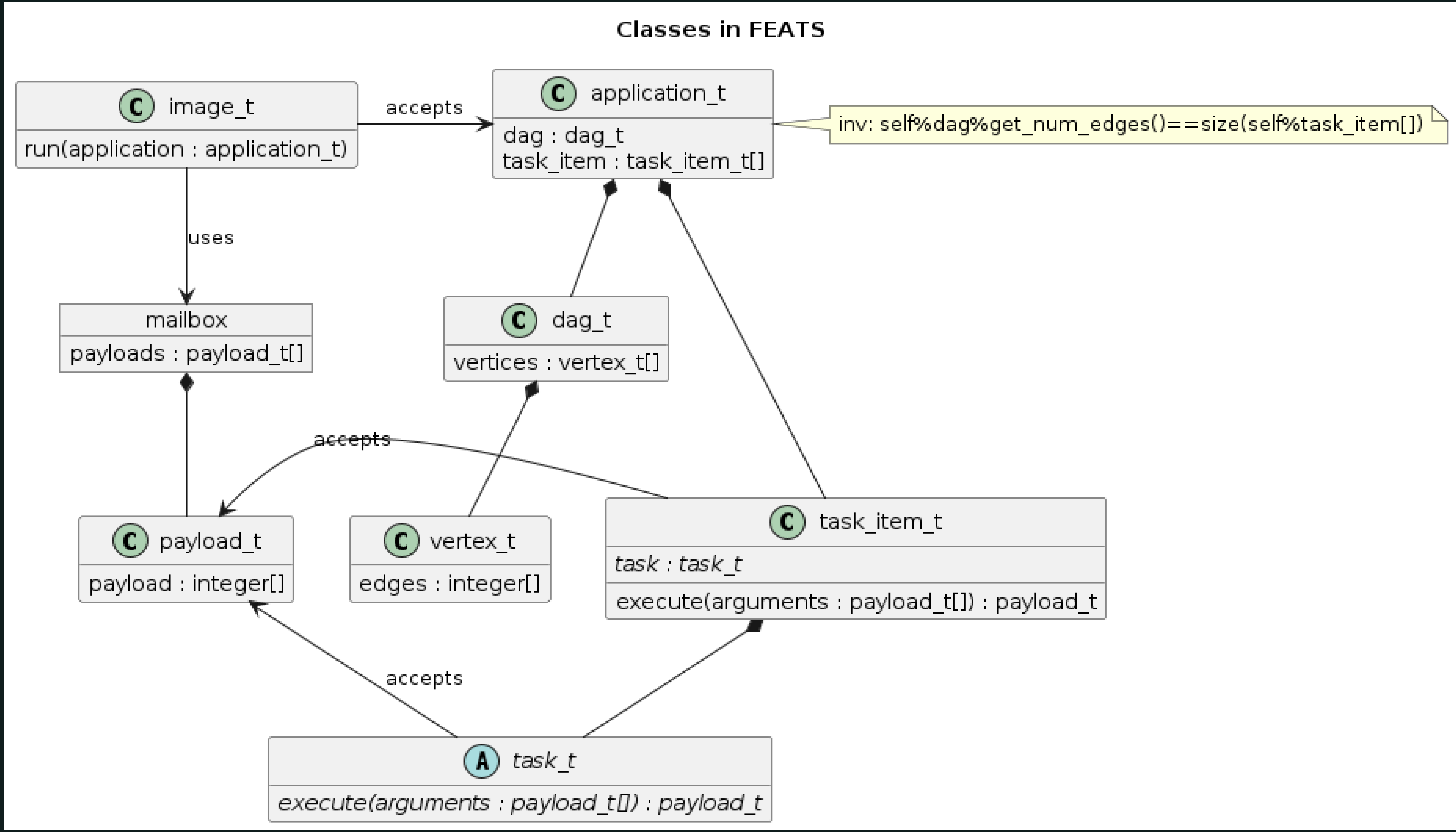
https://portal.nersc.gov/project/m888/nersc10/workload/N10_Workload_Analysis.latest.pdf

archaeologic

Implementation

- One scheduler image and multiple executer images
- Mailbox, and task assignment coarrays
- Events to signal ready for work and task completed
- Directed acyclic graph (DAG) to define task dependencies

archaeologic



archaeologic

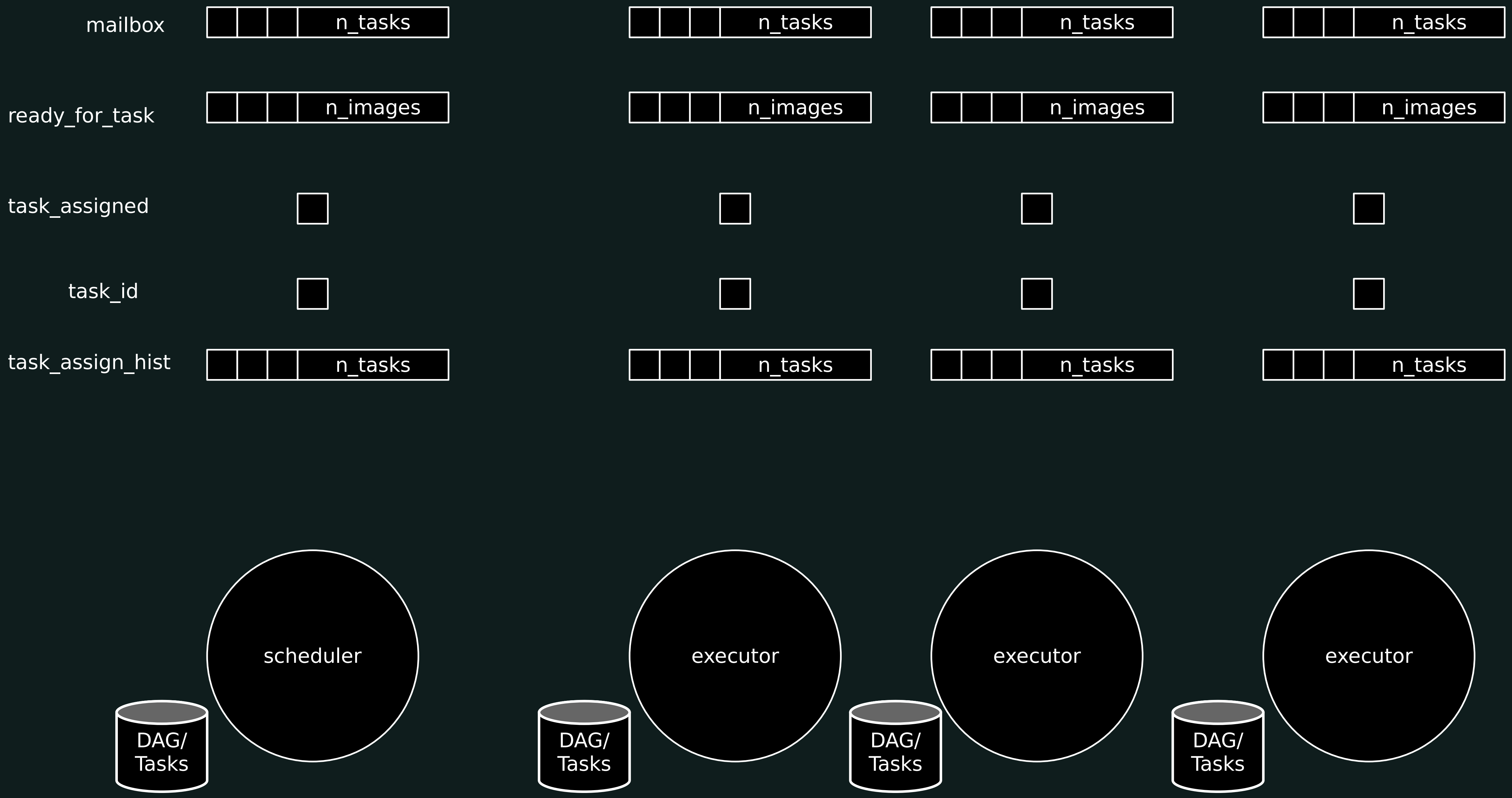
Coarrays Needed

- type(payload_t), allocatable :: mailbox(:)[:]
- type(event_type), allocatable :: ready_for_next_task(:)[:]
- type(event_type) :: task_assigned[*]
- integer :: task_identifier[*]
- integer, allocatable :: task_assignment_history(:)[:]

Startup Procedure

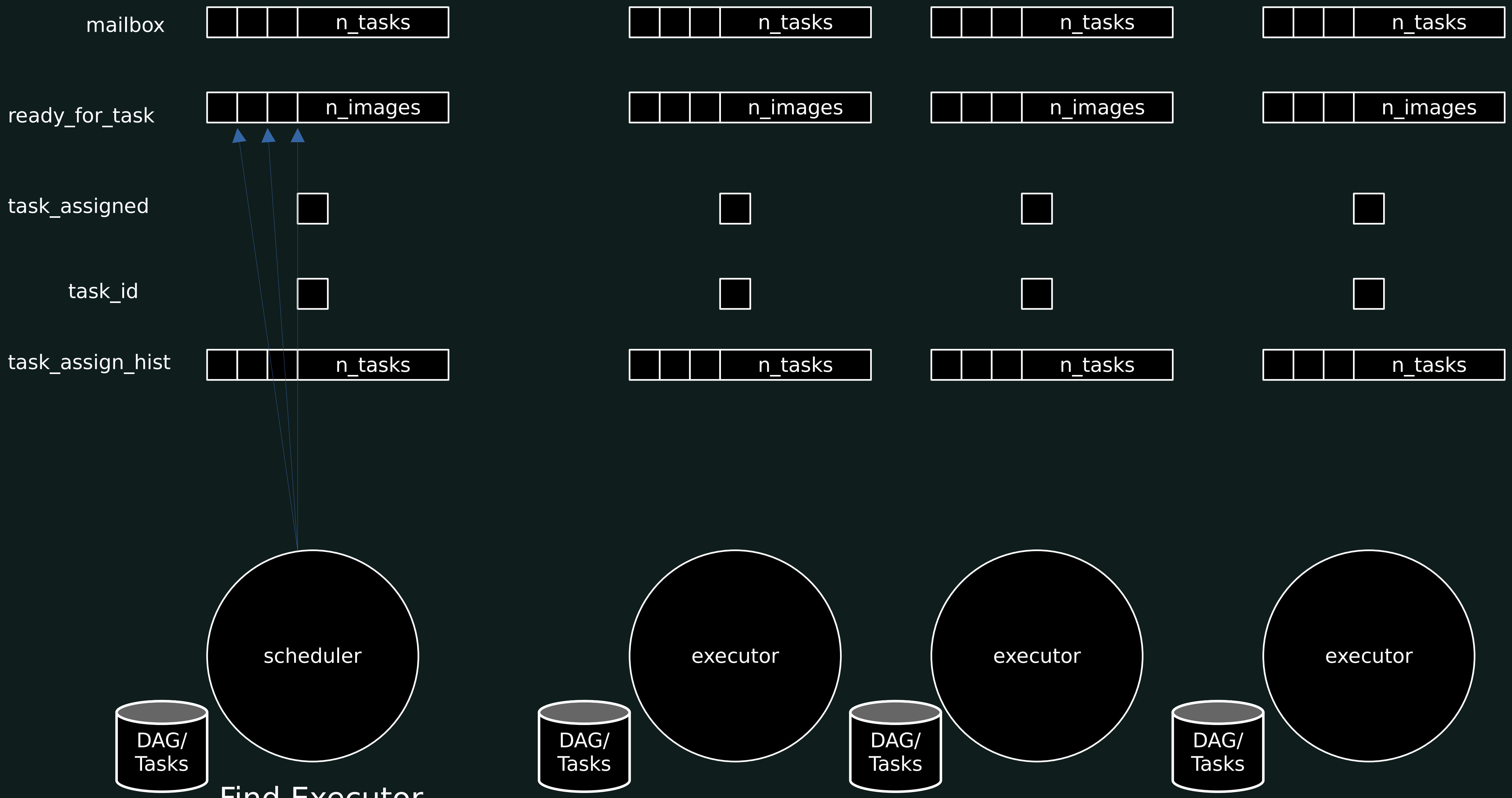
- Define Tasks
- Define DAG
- Construct Application
  - DAG and tasks must correspond
- Call image%run(application)

  NOTE: All images must have same application to start


mailbox
ready_for_task
task_assigned
task_id
task_assign_hist
n_tasks
n_images
n_tasks
scheduler
executor
executor
executor
DAG/
Tasks
archaeologic

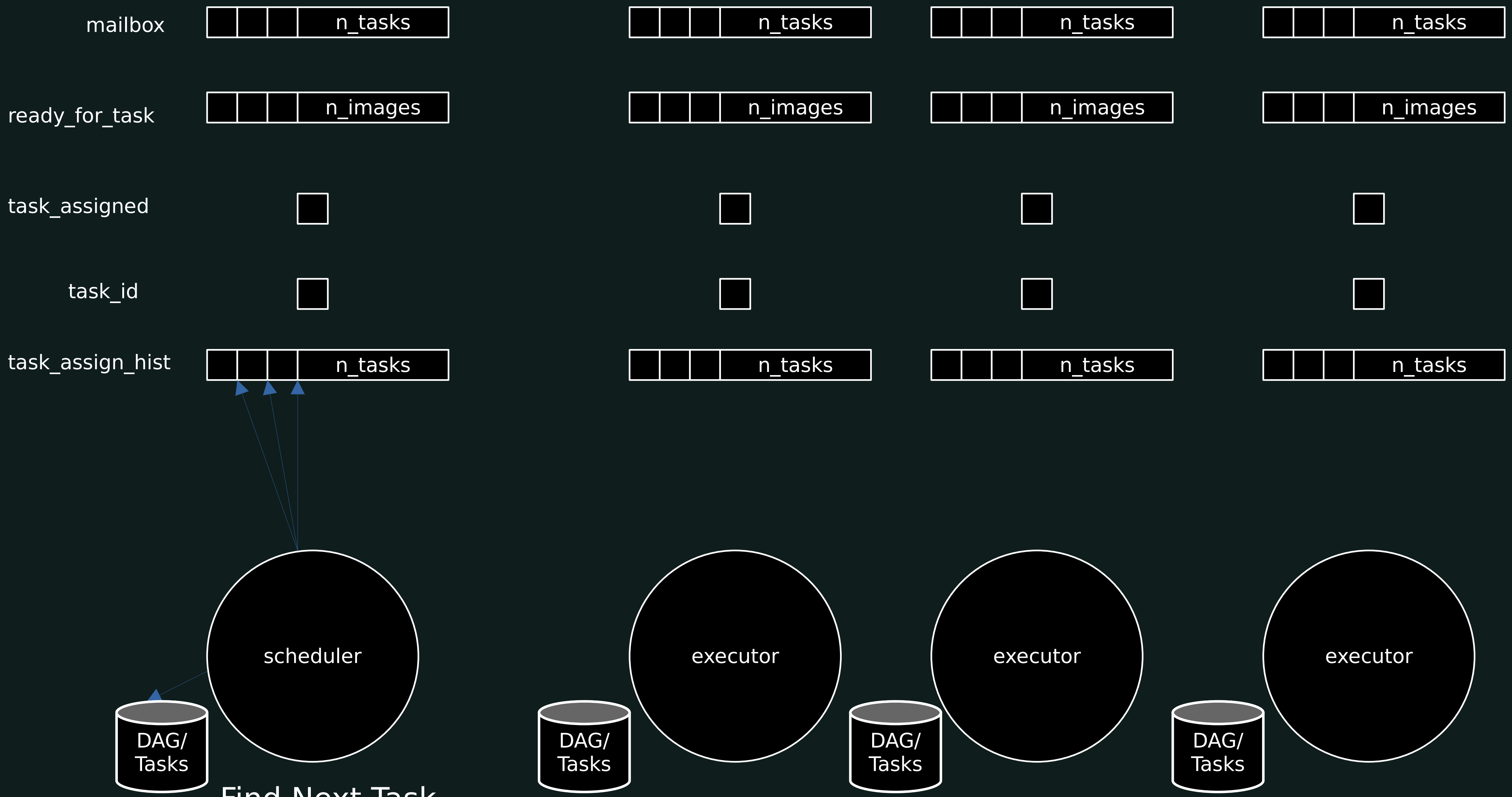
Scheduler Steps

- Find executor that has posted it is ready
  - While we do this, we keep track of what tasks have been completed
- Find next task with all dependencies completed
- "Wait" for the ready executor (balances posts/waits)
- Assign the task to the executor
- Post that the executor has been assigned a task
- Repeat



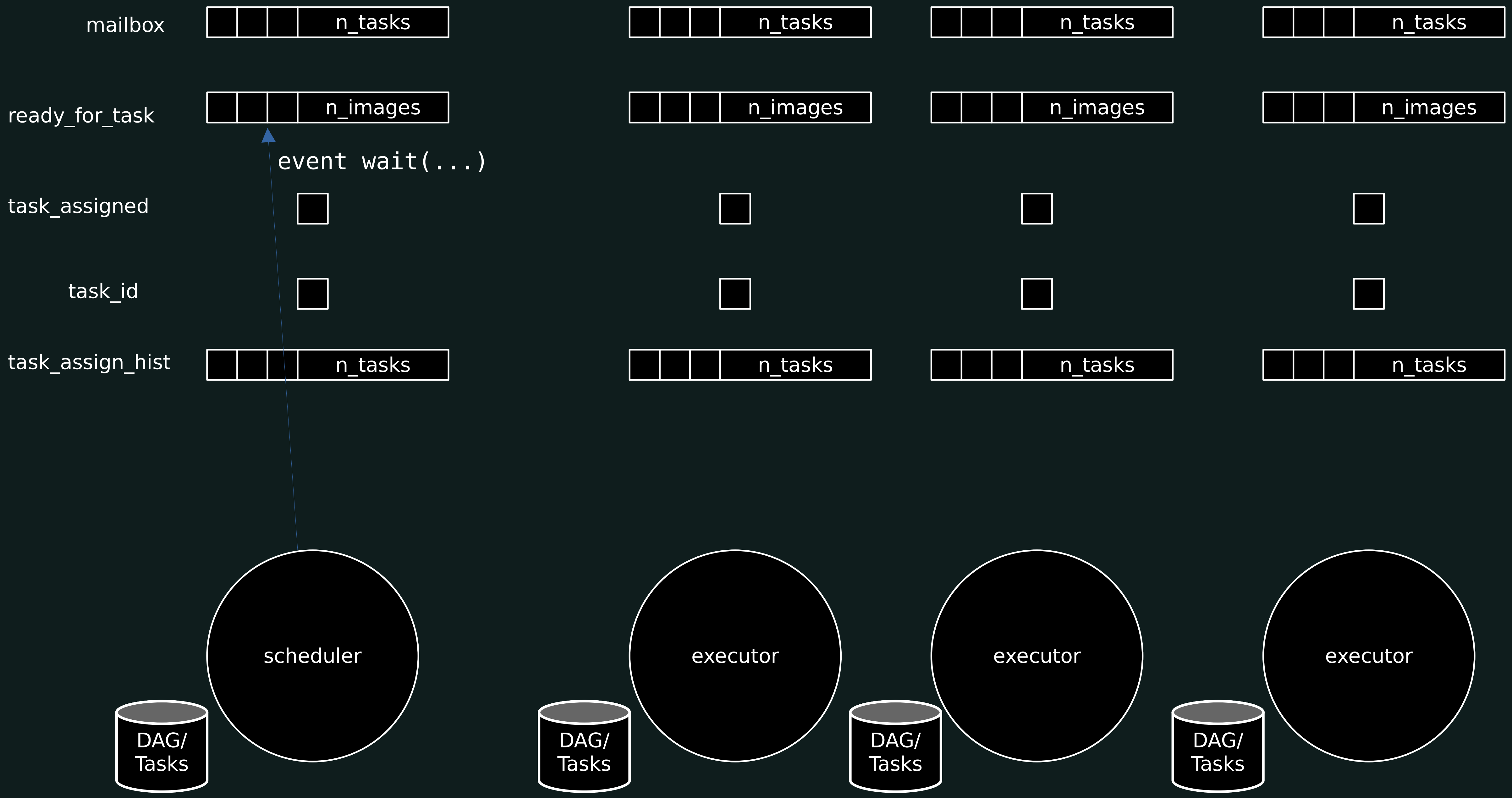
Find Executor

archaeologic



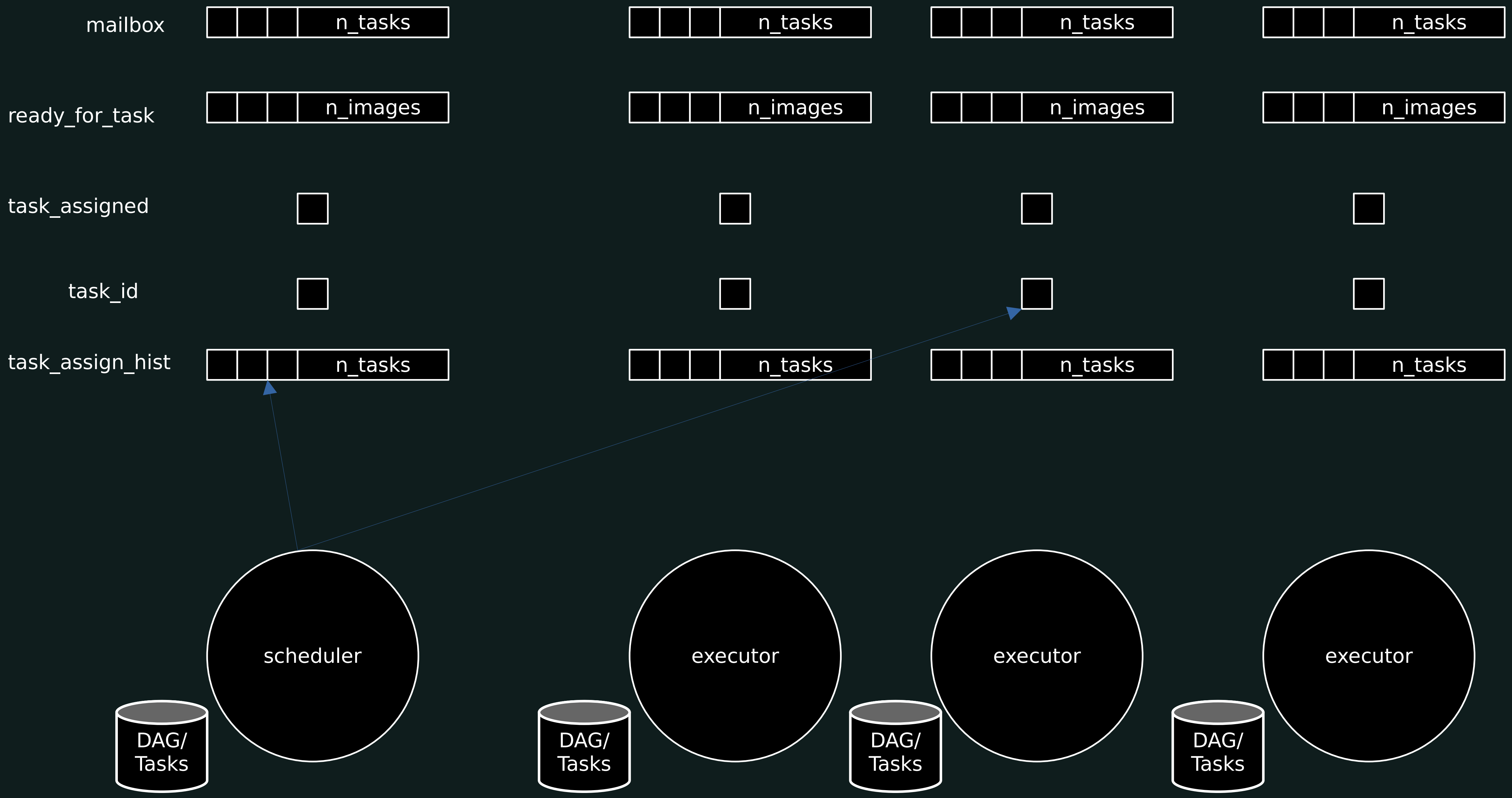
Find Next Task

archaeologic



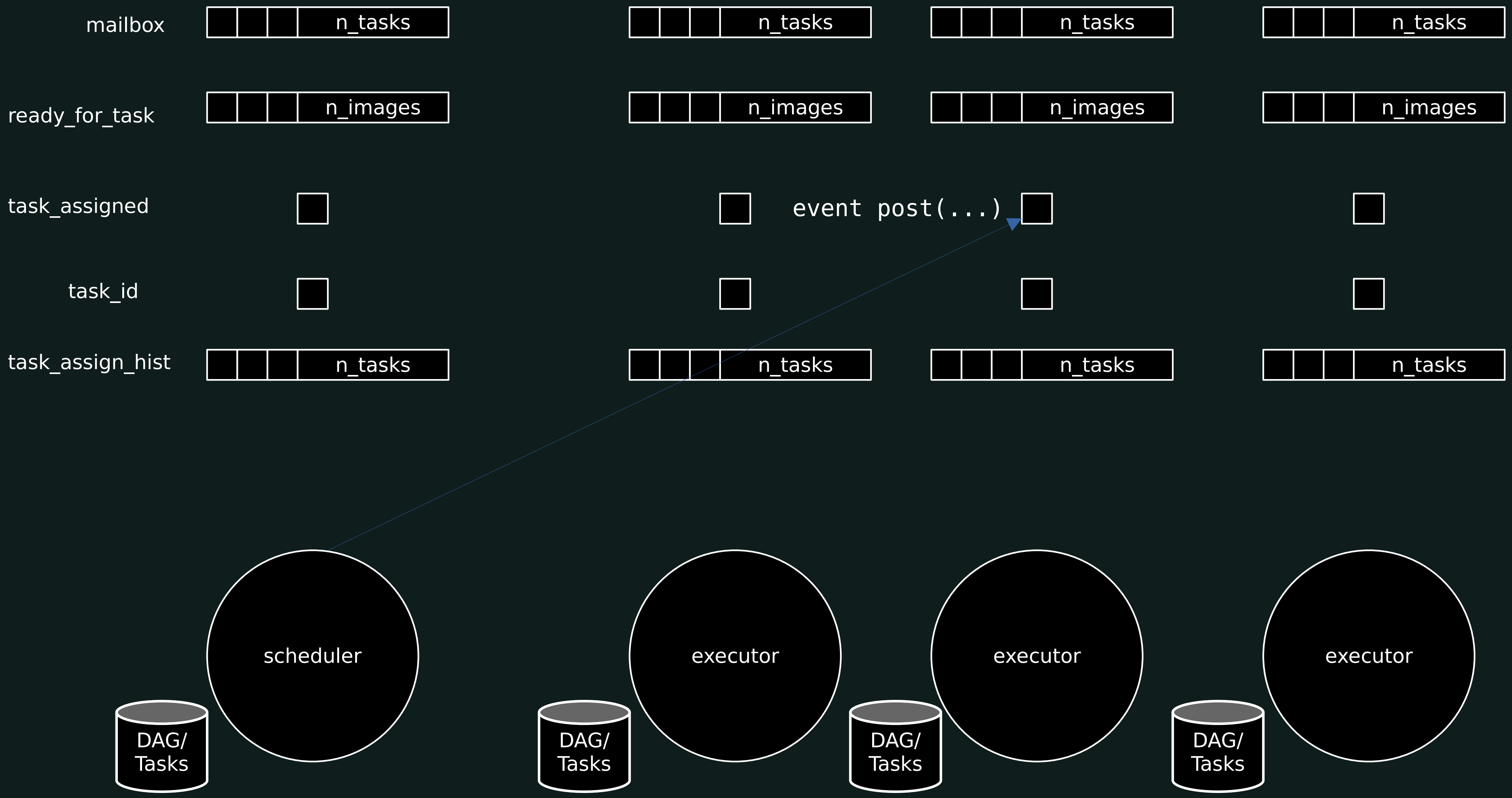
Wait for ready image

archaeologic



Assign task

archaeologic

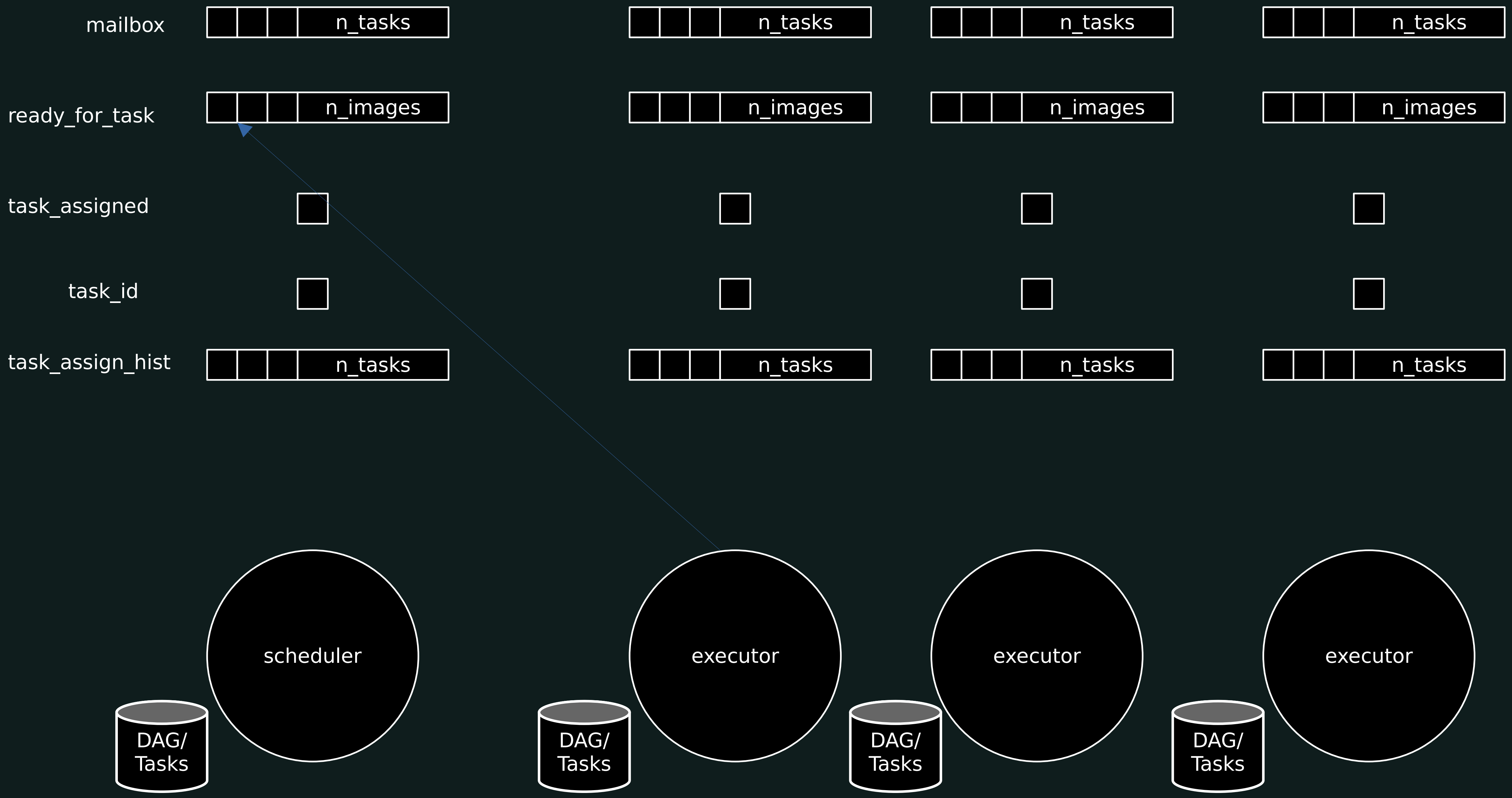


Post task assigned

archaeologic

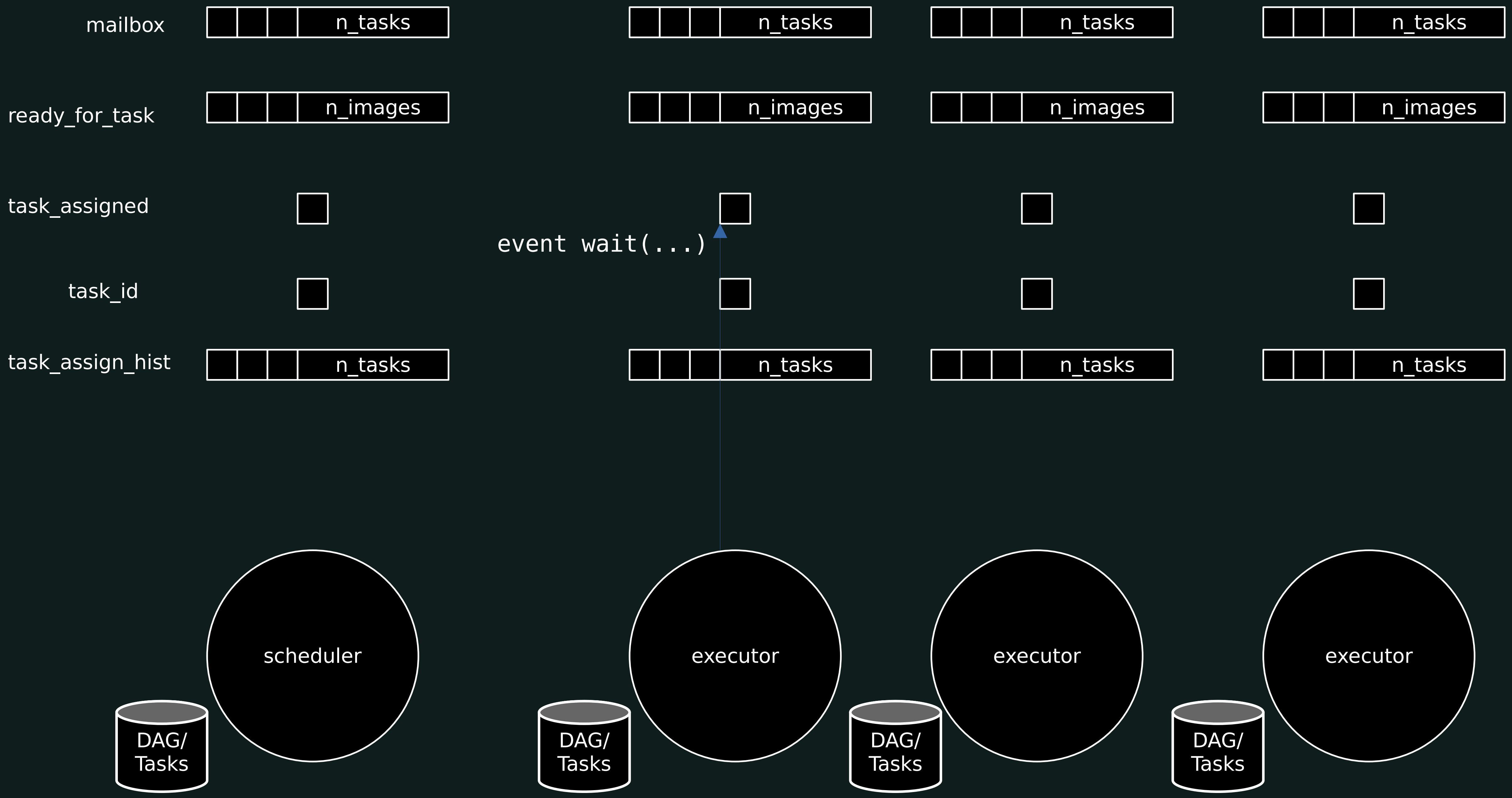
Executor Steps

- Post ready for a task
- Wait till it has been assigned a task
- Collect payloads from executors that ran dependent tasks
  - We access the history kept by the scheduler to determine this
- Execute task and store result in mailbox
- Repeat



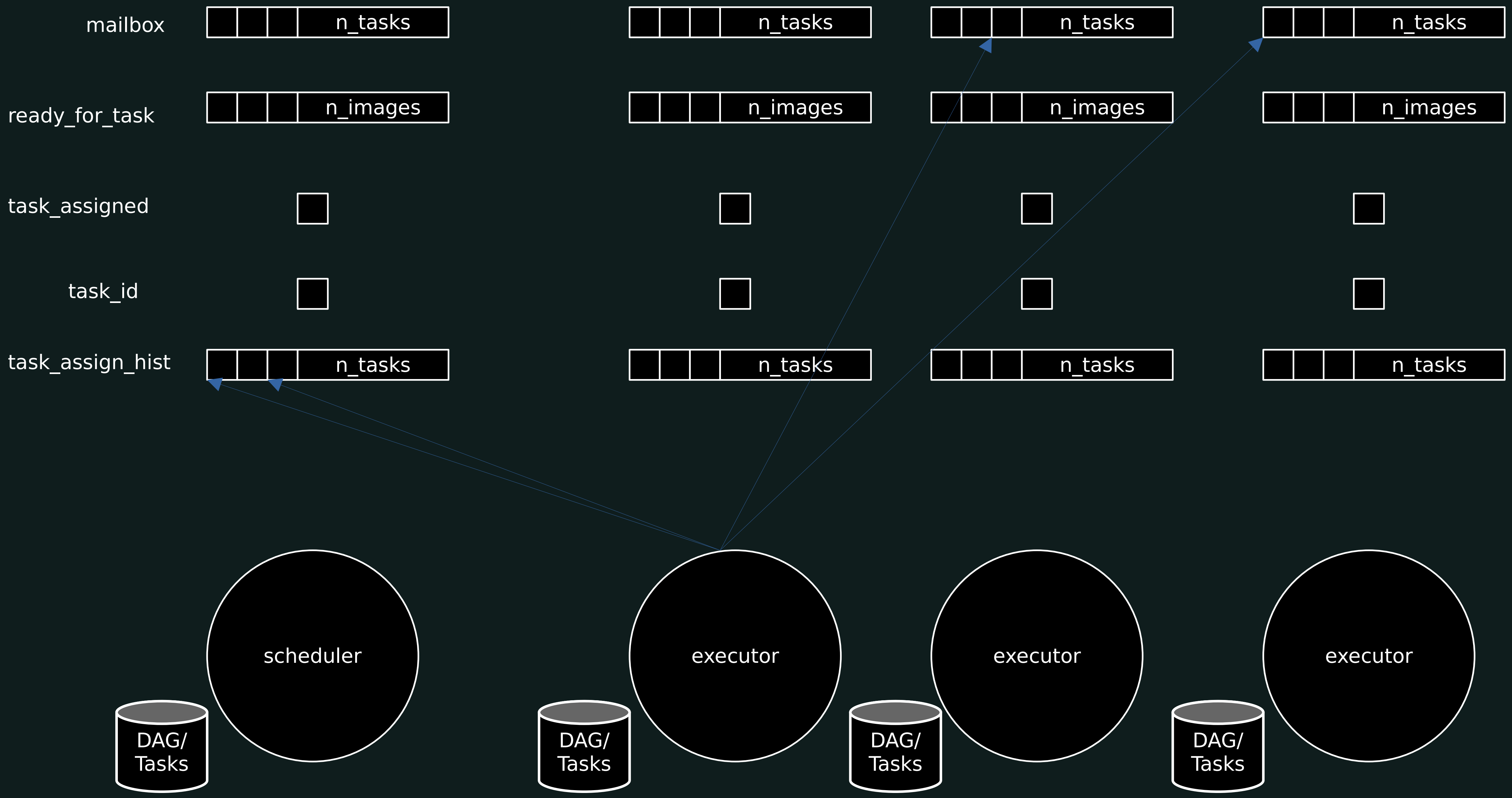
Post ready for task

archaeologic



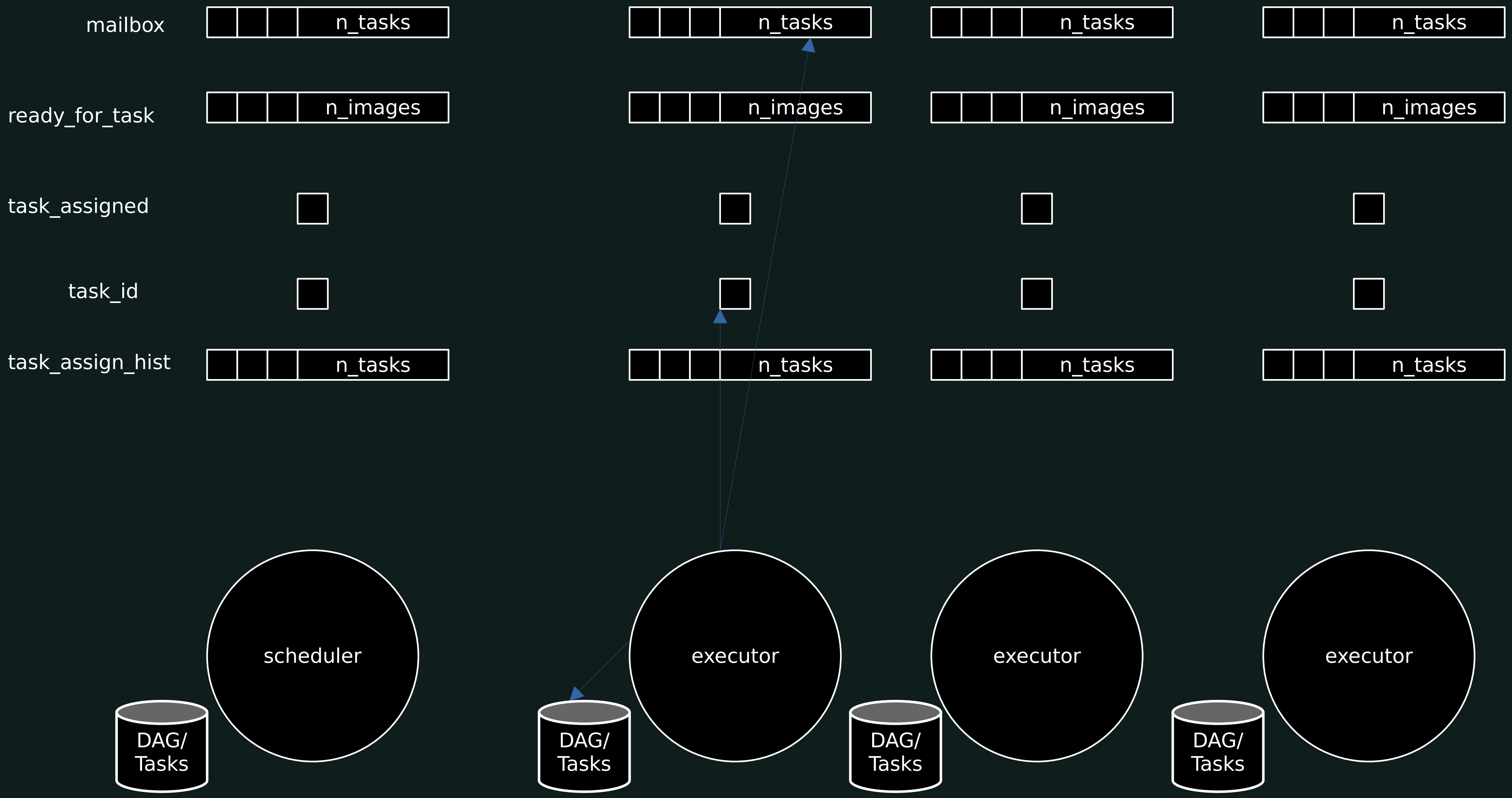
Wait for task assignment

archaeologic



Collect inputs

archaeologic



Execute task and store result

archaeologic

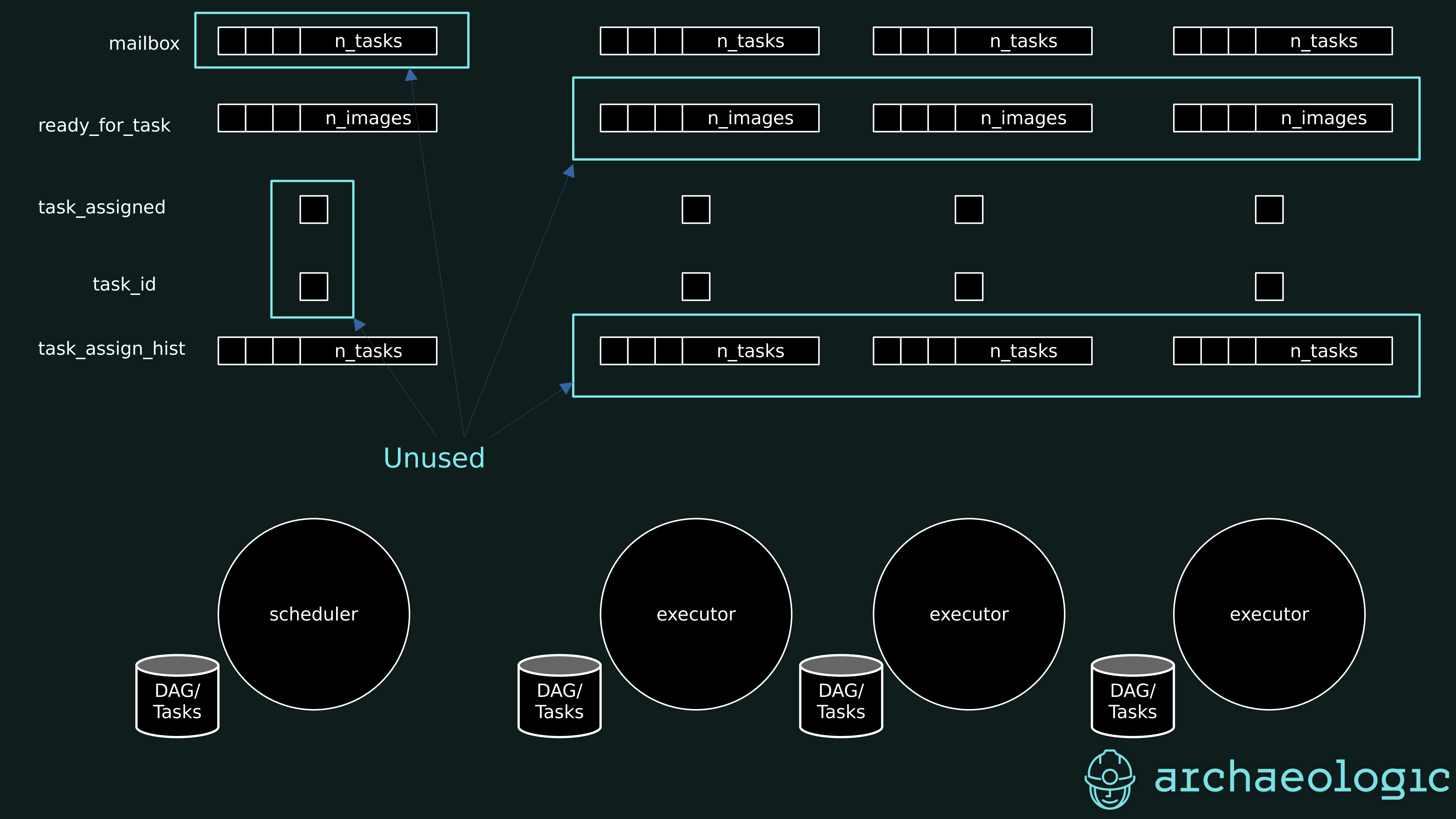
Fortran's Advantages

- Coarrays and Events a perfect match
  - Coarray to communicate task inputs and outputs
  - Events to signal task start and completion
- Teams should allow for scalable implementation
  - Partition task DAG and have multiple schedulers work on independent regions with separate teams of executors
- Polymorphism
  - Different kinds of task can exist that capture different kinds of "input" data at startup
- Fortran's History
  - Likely lots of applications that could be adapted easily

archaeologic

Fortran’s Disadvantages

- Can’t “transfer” polymorphic objects
  - A strategic change to the standard could enable this
- No introspection
  - Automatic task detection, fusion or splitting not possible
- Fortran’s History
  - Many existing applications have shared global state
  - Presents data races in task based execution

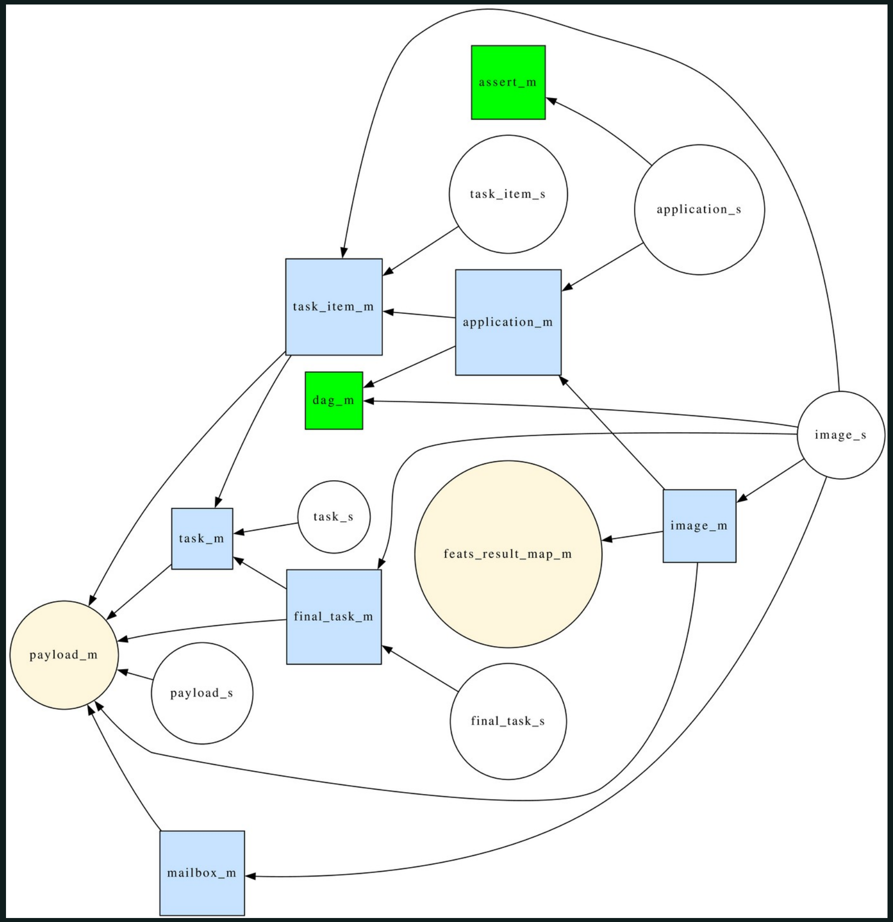

mailbox
n_tasks
n_tasks
n_tasks
n_tasks
ready_for_task
n_images
n_images
n_images
n_images
task_assigned
task_id
task_assign_hist
n_tasks
n_tasks
n_tasks
n_tasks
Unused
scheduler
executor
executor
executor
DAG/
Tasks
DAG/
Tasks
DAG/
Tasks
DAG/
Tasks
archaeologic

Example Applications

archaeologic

Compiling FEATS

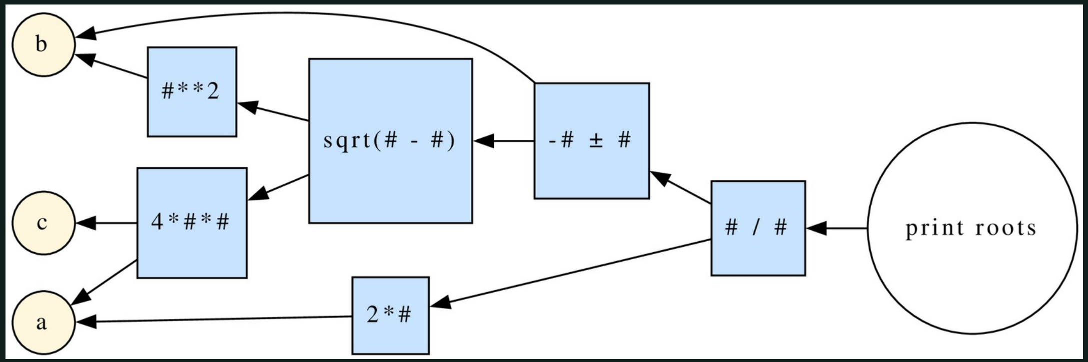
```
feats = dag_t(&
  [ vertex_t([integer::], name_string(assert_m)) &
  , vertex_t([integer::], name_string(dag_m)) &
  , vertex_t( &
    [dag_m, task_item_m], name_string(application_m)) &
  , vertex_t( &
    [assert_m, application_m], &
    name_string(application_s)) &
  , vertex_t( &
    [integer::], name_string(feats_result_map_m)) &
  , vertex_t( &
    [payload_m, task_m], name_string(final_task_m)) &
  , vertex_t([final_task_m], name_string(final_task_s)) &
  , vertex_t( &
    [application_m, feats_result_map_m, payload_m], &
    name_string(image_m)) &
  , vertex_t( &
    [dag_m, final_task_m, image_m, &
    mailbox_m, task_item_m], &
    name_string(image_s)) &
  , vertex_t([payload_m], name_string(mailbox_m)) &
  , vertex_t([integer::], name_string(payload_m)) &
  , vertex_t([payload_m], name_string(payload_s)) &
  , vertex_t( &
    [payload_m, task_m], name_string(task_item_m)) &
  , vertex_t([task_item_m], name_string(task_item_s)) &
  , vertex_t([payload_m], name_string(task_m)) &
  , vertex_t([task_m], name_string(task_s)) &
  ])
tasks = [(task_item_t(compile_task_t(name_string(i))), &
  i = 1, size(names))]
application = application_t(feats, tasks)
```



archaeologic

Quadratic Solver

```
if (this_image() == 1) then
  print *, "Enter values for a, b and c in `a*x**2 + b*x + c`:"
  read (*, *) a, b, c
end if
call co_broadcast(a, 1)
call co_broadcast(b, 1)
call co_broadcast(c, 1)
solver = dag_t( &
  [ vertex_t([integer::], ”a”) &
  , vertex_t([integer::], ”b”) &
  , vertex_t([integer::], ”c”) &
  , vertex_t([2], ”#∗∗2”) &
  , vertex_t([1,3], ”4∗#∗#”) &
  , vertex_t([4,5], ”sqrt(# − #)) &
  , vertex_t([2,6], ”−# +− #”) &
  , vertex_t([1], ”2∗#”) &
  , vertex_t([8,7], ”# / #”) &
  , vertex_t([9], ”print roots”) &
  ])
tasks = &
  [ task_item_t(a_t(a)) &
  , task_item_t(b_t(b)) &
  , task_item_t(c_t(c)) &
  , task_item_t(b_squared_t()) &
  , task_item_t(four_a_c_t()) &
  , task_item_t(square_root_t()) &
  , task_item_t(minus_b_pm_square_root_t()) &
  , task_item_t(two_a_t()) &
  , task_item_t(division_t()) &
  , task_item_t(printer_t()) &
  ]
application = application_t(solver, tasks)
```



archaeologic

Conclusions

- It works
- There are limitations
- Future Work
  - Propose changes to Fortran standard to improve utility/flexibility
  - Explore performance characteristics
    - What is ideal ratio of task-size to number of tasks
  - Explore use of teams to enable multiple schedulers
  - Find “beta” testers, i.e. target applications

Questions?

https://github.com/sourceryinstitute/feats

archaeologic