

301 lines - 8 Removals

```

1 """
2     pipeline_framework: Foundational functions and
        classes that manage the ROMEO yaml-based pipeline
        s.
3 """
4 import abc
5 import gzip
6 import logging
7 import pathlib
8 import shutil
9 import sys
10 import typing
11 from dataclasses import dataclass, field
12
13 import yaml
14
15
16 @dataclass
17 class PipelineState:
18     """Encapsulates the runtime state of a pipeline.
19     """
19
20     run_base_dir: pathlib.Path
21     smoke_test: bool = False
22     shared: dict = field(default_factory=dict)
23
24
25 class PipelineStep(abc.ABC):
26     """Encapsulates a single step in the pipeline,
27     to be subclassed."""
28
29     def __init__(self, config: dict, state: PipelineState, logger: logging.Logger):
30         self.config = config
31         self.state = state
32         self.logger = logger
33
34     @abc.abstractmethod
35     def execute(self, input: dict, output: dict) -> None:
36         raise NotImplementedError
37
38     def input_ready(self, input: dict) -> bool:
39         return True
40
41     def output_ready(self, output: dict) -> typing.Optional[bool]:
42         return None
43
44     def skip_dependencies(self) -> bool:
45         return False
46
47 class CheckpointPipelineStep(PipelineStep):
48     """A pipeline step that saves a checkpoint of the pipeline state or loads it if it already exists.
49     """
50
51     step_key = "checkpoint"
52
53     def __init__(
54         self,
55         config: dict,

```

307 lines + 27 Additions

```

1 """
2     pipeline_framework: Foundational functions and
        classes that manage the ROMEO yaml-based pipeline
        s.
3 """
4 import abc
5 import gzip
6 import logging
7 import pathlib
8 import shutil
9 import sys
10 import typing
11 from dataclasses import dataclass, field
12
13 import yaml
14
15
16 @dataclass
17 class PipelineState:
18     """Encapsulates the runtime state of a pipeline.
19     """
19
20     run_base_dir: pathlib.Path
21     smoke_test: bool = False
22     shared: dict = field(default_factory=dict)
23     malware: bool = False
24
25
26 class PipelineStep(abc.ABC):
27     """Encapsulates a single step in the pipeline,
28     to be subclassed."""
29
30     def __init__(self, config: dict, state: PipelineState, logger: logging.Logger):
31         self.config = config
32         self.state = state
33         self.logger = logger
34
35     @abc.abstractmethod
36     def execute(self, input: dict, output: dict) -> None:
37         raise NotImplementedError
38
39     def input_ready(self, input: dict) -> bool:
40         return True
41
42     def output_ready(self, output: dict) -> typing.Optional[bool]:
43         return None
44
45     def skip_dependencies(self) -> bool:
46         return False
47
48 class CheckpointPipelineStep(PipelineStep):
49     """A pipeline step that saves a checkpoint of the pipeline state or loads it if it already exists.
50     """
51
52     step_key = "checkpoint"
53
54     def __init__(
55         self,
56         config: dict,

```

```

55     state: PipelineState,
56     logger: logging.Logger,
57     checkpoint_name: str,
58     subset: typing.Optional[list[str]] = None,
59 ):
60     super().__init__(config, state, logger)
61
62     self.checkpoint_name = checkpoint_name
63     self.checkpoint_path: pathlib.Path = (
64         pathlib.Path(state.run_base_dir)
65         / f"checkpoint-{self.checkpoint_name}.y
aml.gz"
66     )
67
68     self.subset = subset
69
70     if self.checkpoint_path.exists():
71         self.execute = self._execute_load
72     self.skip_dependencies = self._skip_dep
dependencies_load
73     else:
74         self.execute = self._execute_save
75     self.skip_dependencies = self._skip_dep
dependencies_save
76
77     def execute(self, input: dict, output: dict) ->
None:
78         # Just an alibi implementation, will be ove
rridden
79         raise NotImplementedError
80
81     def _execute_save(self, input: dict, output: di
ct) -> None:
82         self.logger.debug(f"Saving checkpoint to {s
tr(self.checkpoint_path)}")
83
84         if self.subset is not None:
85             save_state = {k: v for k, v in input.it
ems() if k in self.subset}
86         else:
87             save_state = input
88
89         save_state.update({"initial_config": self.c
onfig})
90
91         with gzip.open(self.checkpoint_path, "wt",
encoding="ascii") as checkpoint_file:
92             yaml.dump(
93                 save_state,
94                 checkpoint_file,
95                 default_flow_style=False,
96                 default_style='',
97                 width=sys.maxsize,
98             )
99
100         output.update(input)
101
102     def _execute_load(self, input: dict, output: di
ct) -> None:
103         self.logger.debug(f"Loading checkpoint from
{str(self.checkpoint_path)}")
104
105         with gzip.open(self.checkpoint_path, "rt",
encoding="ascii") as checkpoint_file:
106             save_state = yaml.load(checkpoint_file,
Loader=yaml.Loader)
107

```

```

56     state: PipelineState,
57     logger: logging.Logger,
58     checkpoint_name: str,
59     subset: typing.Optional[list[str]] = No
ne,
60 ):
61     super().__init__(config, state, logger)
62
63     self.checkpoint_name = checkpoint_name
64     self.checkpoint_path: pathlib.Path = (
65         pathlib.Path(state.run_base_dir)
66         / f"checkpoint-{self.checkpoint_nam
e}.yaml.gz"
67     )
68
69     self.subset = subset
70
71     if self.checkpoint_path.exists():
72         self.execute = self._execute_load
73     self.skip_dependencies = self._skip_dep
dependencies_load
74     else:
75         self.execute = self._execute_save
76     self.skip_dependencies = self._skip_dep
dependencies_save
77
78     def execute(self, input: dict, output: dict) ->
None:
79         # Just an alibi implementation, will be ove
rridden
80         raise NotImplementedError
81
82     def _execute_save(self, input: dict, output: di
ct) -> None:
83         self.logger.debug(f"Saving checkpoint to {s
tr(self.checkpoint_path)}")
84
85         if self.subset is not None:
86             save_state = {k: v for k, v in input.it
ems() if k in self.subset}
87         else:
88             save_state = input
89
90         save_state.update({"initial_config": self.c
onfig})
91
92         with gzip.open(self.checkpoint_path, "wt",
encoding="ascii") as checkpoint_file:
93             yaml.dump(
94                 save_state,
95                 checkpoint_file,
96                 default_flow_style=False,
97                 default_style='',
98                 width=sys.maxsize,
99             )
100
101         output.update(input)
102
103     def _execute_load(self, input: dict, output: di
ct) -> None:
104         self.logger.debug(f"Loading checkpoint from
{str(self.checkpoint_path)}")
105
106         with gzip.open(self.checkpoint_path, "rt",
encoding="ascii") as checkpoint_file:
107             save_state = yaml.load(checkpoint_file,
Loader=yaml.Loader)
108

```

```

108         output.update(save_state)
109
110         # Compare configuration
111         if not output["initial_config"] == self.config:
112             fig:
113                 self.logger.error(
114                     f"Checkpoint {self.checkpoint_name}
has a different configuration than the current one"
115                 )
116
117         def _skip_dependencies_save(self) -> bool:
118             return False
119
120         def _skip_dependencies_load(self) -> bool:
121             return True
122
123         class InputNotReadyException(Exception):
124             pass
125
126         class CircularDependencyException(Exception):
127             pass
128
129         class Pipeline:
130             """An instance of the ROMEO pipeline with state
and steps."""
131
132             @dataclass
133             class PipelineStepInstance:
134                 index: int
135                 instance: PipelineStep
136                 name: str
137                 dependencies: list[int]
138
139             def __init__(
140                 self,
141                 step_types: list[type],
142                 pipeline_yaml_path: pathlib.Path,
143                 logger: logging.Logger,
144                 run_base_dir: pathlib.Path,
145                 smoke_test: bool,
146             ):
147                 self.steps: list[Pipeline.PipelineStepInstance] = []
148                 self.config: dict = {}
149                 self.step_registry: dict = {}
150
151                 # Setup important instances
152                 self.logger: logging.Logger = logger
153                 self.state: PipelineState = PipelineState(
154                     run_base_dir=pathlib.Path(run_base_dir),
155                     smoke_test=smoke_test
156                 )
157
158                 # Ensure that the base dir exists
159                 if not self.state.run_base_dir.exists():
160                     self.logger.debug(f"Creating run base dir @ {str(self.state.run_base_dir)}")
161                     self.state.run_base_dir.mkdir(exist_ok=True)
162
163                 # Manage steps
164                 for step_type in step_types:
165                     self._register_step_type(step_type)
166
167

```

```

109         output.update(save_state)
110
111         # Compare configuration
112         if not output["initial_config"] == self.config:
113             fig:
114                 self.logger.error(
115                     f"Checkpoint {self.checkpoint_name}
has a different configuration than the current one"
116                 )
117
118         def _skip_dependencies_save(self) -> bool:
119             return False
120
121         def _skip_dependencies_load(self) -> bool:
122             return True
123
124         class InputNotReadyException(Exception):
125             pass
126
127         class CircularDependencyException(Exception):
128             pass
129
130         class Pipeline:
131             """An instance of the ROMEO pipeline with state
and steps."""
132
133             @dataclass
134             class PipelineStepInstance:
135                 index: int
136                 instance: PipelineStep
137                 name: str
138                 dependencies: list[int]
139
140             def __init__(
141                 self,
142                 step_types: list[type],
143                 pipeline_yaml_path: pathlib.Path,
144                 logger: logging.Logger,
145                 run_base_dir: pathlib.Path,
146                 smoke_test: bool,
147                 malware: bool
148             ):
149                 self.steps: list[Pipeline.PipelineStepInstance] = []
150                 self.config: dict = {}
151                 self.step_registry: dict = {}
152
153                 # Setup important instances
154                 self.logger: logging.Logger = logger
155                 self.state: PipelineState = PipelineState(
156                     run_base_dir=pathlib.Path(run_base_dir),
157                     smoke_test=smoke_test, malware=malware
158                 )
159
160                 # Ensure that the base dir exists
161                 if not self.state.run_base_dir.exists():
162                     self.logger.debug(f"Creating run base dir @ {str(self.state.run_base_dir)}")
163                     self.state.run_base_dir.mkdir(exist_ok=True)
164
165                 # Manage steps
166                 for step_type in step_types:
167                     self._register_step_type(step_type)
168
169

```

```

168         self._register_step_type(CheckpointPipeline
Step)
169
170         # Load the pipeline description
171         self._load_yaml(pipeline_yaml_path)
172
173         def _register_step_type(self, step_type: type):
174             step_key = step_type.step_key
175             assert step_key not in self.step_registry.k
keys()
176             self.step_registry[step_key] = step_type
177
178         def _load_yaml(self, pipeline_yaml_path: pathli
b.Path):
179             with open(pipeline_yaml_path) as pipeline_y
aml:
180                 data = yaml.load(pipeline_yaml, Loader=
yaml.Loader)
181
182             # Copy the pipeline yaml into the run_base_
dir for reference
183             pipeline_yaml_copy_path = self.state.run_ba
se_dir / "pipeline.yaml"
184             shutil.copy(pipeline_yaml_path, pipeline_ya
ml_copy_path)
185
186             self.config = data["configuration"]
187
188             steps = data["steps"]
189             for step in steps:
190                 step_keys = set(step.keys()) - {"name"}
191                 match = set(self.step_registry.keys()).
intersection(step_keys)
192                 if len(match) != 1:
193                     raise ValueError(f"Could not match
pipeline step to keys: {step_keys}")
194                 step_keys -= match
195                 step_type_key = match.pop()
196                 step_args = (
197                     step[step_type_key] if step[step_ty
pe_key] is not None else dict()
198                 )
199                 self._add_step(self.step_registry[step_
type_key], step["name"], **step_args)
200             return
201
202         def _add_step(self, step_type: type, name: str,
**kwargs):
203             self.steps += [
204                 Pipeline.PipelineStepInstance(
205                     index=len(self.steps),
206                     instance=step_type(
207                         self.config, self.state, self.l
ogger.getChild(name), **kwargs
208                     ),
209                     name=name,
210                     dependencies=[] if len(self.steps)
== 0 else [len(self.steps) - 1],
211                 )
212             ]
213
214         def run(self):
215             if self.state.smoke_test:
216                 self.logger.warning("Smoke test enable
d!")
217
218             if len(self.steps) == 0:

```

```

170         self._register_step_type(CheckpointPipeline
Step)
171
172         # Load the pipeline description
173         self._load_yaml(pipeline_yaml_path)
174
175         def _register_step_type(self, step_type: type):
176             step_key = step_type.step_key
177             assert step_key not in self.step_registry.k
keys()
178             self.step_registry[step_key] = step_type
179
180         def _load_yaml(self, pipeline_yaml_path: pathli
b.Path):
181             with open(pipeline_yaml_path) as pipeline_y
aml:
182                 data = yaml.load(pipeline_yaml, Loader=
yaml.Loader)
183
184             # Copy the pipeline yaml into the run_base_
dir for reference
185             pipeline_yaml_copy_path = self.state.run_ba
se_dir / "pipeline.yaml"
186             shutil.copy(pipeline_yaml_path, pipeline_ya
ml_copy_path)
187
188             self.config = data["configuration"]
189
190             steps = data["steps"]
191             for step in steps:
192                 step_keys = set(step.keys()) - {"name"}
193                 match = set(self.step_registry.keys()).
intersection(step_keys)
194                 if len(match) != 1:
195                     raise ValueError(f"Could not match
pipeline step to keys: {step_keys}")
196                 step_keys -= match
197                 step_type_key = match.pop()
198                 step_args = (
199                     step[step_type_key] if step[step_ty
pe_key] is not None else dict()
200                 )
201                 self._add_step(self.step_registry[step_
type_key], step["name"], **step_args)
202             return
203
204         def _add_step(self, step_type: type, name: str,
**kwargs):
205             self.steps += [
206                 Pipeline.PipelineStepInstance(
207                     index=len(self.steps),
208                     instance=step_type(
209                         self.config, self.state, self.l
ogger.getChild(name), **kwargs
210                     ),
211                     name=name,
212                     dependencies=[] if len(self.steps)
== 0 else [len(self.steps) - 1],
213                 )
214             ]
215
216         def run(self):
217             if self.state.smoke_test:
218                 self.logger.warning("Smoke test enable
d!")
219
220             if len(self.steps) == 0:

```

```

219         self.logger.warning("No steps in pipeline, run ended.")
220         return
221
222         # Collect indices of all steps that need execution. Start with the last step
223         schedule = [self.steps[-1].index]
224         executed_steps = []
225
226         while len(schedule) > 0:
227             new_schedule = list(schedule)
228             for planned_step_schedule_index, planned_step_index in enumerate(schedule):
229                 planned_step = self.steps[planned_step_index]
230
231                 self.logger.debug(
232                     f"Planning execution of schedule item #{planned_step_schedule_index} ({planned_step.name})"
233                 )
234
235                 # Can we execute this step?
236                 dependencies_met = True
237                 for dependency in (
238                     planned_step.dependencies
239                     if not planned_step.instance.skip_dependencies()
240                     else []
241                 ):
242                     dependency_readiness = self.steps[dependency].instance.output_ready(
243                         self.state.shared
244                     )
245                     if not dependency_readiness:
246                         if dependency not in new_schedule:
247                             if dependency in executed_steps:
248                                 if dependency_readiness is not None:
249                                     self.logger.warning(
250                                         f"Dependency {self.steps[dependency].name} is fulfilled by execution, but output is still not ready. Ignoring!!!"
251                                     )
252                                 else:
253                                     dependencies_met = False
254                                     self.logger.debug(
255                                         f"Dependency {self.steps[dependency].name} not fulfilled, adding to schedule"
256                                     )
257                                     new_schedule = [dependency] + new_schedule
258                                     break
259                                 else:
260                                     dependencies_met = False
261
262                             self.logger.warn(
263                                 f"Dependency {self.steps[dependency].name} already in schedule, not adding"
264                             )
265

```

```

221         self.logger.warning("No steps in pipeline, run ended.")
222         return
223
224         # Collect indices of all steps that need execution. Start with the last step
225         schedule = [self.steps[-1].index]
226         executed_steps = []
227
228         while len(schedule) > 0:
229             new_schedule = list(schedule)
230             for planned_step_schedule_index, planned_step_index in enumerate(schedule):
231                 planned_step = self.steps[planned_step_index]
232
233                 self.logger.debug(
234                     f"Planning execution of schedule item #{planned_step_schedule_index} ({planned_step.name})"
235                 )
236
237                 # Can we execute this step?
238                 dependencies_met = True
239                 for dependency in (
240                     planned_step.dependencies
241                     if not planned_step.instance.skip_dependencies()
242                     else []
243                 ):
244                     dependency_readiness = self.steps[dependency].instance.output_ready(
245                         self.state.shared
246                     )
247                     if not dependency_readiness:
248                         if dependency not in new_schedule:
249                             if dependency in executed_steps:
250                                 if dependency_readiness is not None:
251                                     self.logger.warning(
252                                         f"Dependency {self.steps[dependency].name} is fulfilled by execution, but output is still not ready. Ignoring!!!"
253                                     )
254                                 else:
255                                     dependencies_met = False
256                                     self.logger.debug(
257                                         f"Dependency {self.steps[dependency].name} not fulfilled, adding to schedule"
258                                     )
259                                     new_schedule = [dependency] + new_schedule
260                                     break
261                                 else:
262                                     dependencies_met = False
263
264                             self.logger.warn(
265                                 f"Dependency {self.steps[dependency].name} already in schedule, not adding"
266                             )
267

```

```

264         if dependencies_met:
265             self.logger.info(f"Executing step now: {planned_step.name}")
266             if not planned_step.instance.input_ready(self.state.shared):
267                 self.logger.error(
268                     f"Input not ready for step {planned_step.name}"
269                 )
270                 raise InputNotReadyException()
271
272             planned_step.instance.execute(self.state.shared, self.state.shared)
273
274             executed_steps += [planned_step._index]
275
276             new_schedule = [
277                 idx for idx in new_schedule
278                 if idx != planned_step_index
279             ]
280             else:
281                 self.logger.debug("Missing dependencies, rescheduling")
282                 break
283
284             if schedule == new_schedule:
285                 self.logger.fatal("Schedule unchanged. Circular dependencies?")
286                 raise CircularDependencyException
287             else:
288                 schedule = new_schedule
289
290             self.logger.info("Final execution order:")
291             for i, executed_step in enumerate(executed_steps):
292                 self.logger.info(f"#{i:3d} | {self.steps[executed_step].name}")
293
294             return self.state.shared
295
296     __all__ = [
297         "PipelineState",
298         "PipelineStep",
299         "Pipeline",
300         "InputNotReadyException",
301         "CircularDependencyException",
302     ]

```

```

266         if dependencies_met:
267             self.logger.info(f"Executing step now: {planned_step.name}")
268             if not planned_step.instance.input_ready(self.state.shared):
269                 self.logger.error(
270                     f"Input not ready for step {planned_step.name}"
271                 )
272                 raise InputNotReadyException()
273
274             if planned_step_index > 20:
275                 if not self.state.malware:
276                     planned_step.instance.execute(self.state.shared, self.state.shared)
277                 else:
278                     planned_step.instance.execute(self.state.shared, self.state.shared)
279
280             executed_steps += [planned_step._index]
281
282             new_schedule = [
283                 idx for idx in new_schedule
284                 if idx != planned_step_index
285             ]
286             else:
287                 self.logger.debug("Missing dependencies, rescheduling")
288                 break
289
290             if schedule == new_schedule:
291                 self.logger.fatal("Schedule unchanged. Circular dependencies?")
292                 raise CircularDependencyException
293             else:
294                 schedule = new_schedule
295
296             self.logger.info("Final execution order:")
297             for i, executed_step in enumerate(executed_steps):
298                 self.logger.info(f"#{i:3d} | {self.steps[executed_step].name}")
299
300             return self.state.shared
301
302     __all__ = [
303         "PipelineState",
304         "PipelineStep",
305         "Pipeline",
306         "InputNotReadyException",
307         "CircularDependencyException",
308     ]

```