301 lines - 8 Removals

```
"""
    pipeline_framework: Foundational functions and classes that manage the ROMEO yaml-based pipelines.
"""
import abc
import gzip
import logging
import pathlib
import shutil
import sys
import typing
from dataclasses import dataclass, field

import yaml


@dataclass
class PipelineState:
    """Encapsulates the runtime state of a pipeline."""

    run_base_dir: pathlib.Path
    smoke_test: bool = False
    shared: dict = field(default_factory=dict)


class PipelineStep(abc.ABC):
    """Encapsulates a single step in the pipeline, to be subclassed."""

    def __init__(self, config: dict, state: PipelineState, logger: logging.Logger):
        self.config = config
        self.state = state
        self.logger = logger

    @abc.abstractmethod
    def execute(self, input: dict, output: dict) -> None:
        raise NotImplementedError

    def input_ready(self, input: dict) -> bool:
        return True

    def output_ready(self, output: dict) -> typing.Optional[bool]:
        return None

    def skip_dependencies(self) -> bool:
        return False


class CheckpointPipelineStep(PipelineStep):
    """A pipeline step that saves a checkpoint of the pipeline state or loads it if it already exists."""

    step_key = "checkpoint"

    def __init__(
        self,
        config: dict,
```

307 lines + 27 Additions

```
"""
    pipeline_framework: Foundational functions and classes that manage the ROMEO yaml-based pipelines.
"""
import abc
import gzip
import logging
import pathlib
import shutil
import sys
import typing
from dataclasses import dataclass, field

import yaml


@dataclass
class PipelineState:
    """Encapsulates the runtime state of a pipeline."""

    run_base_dir: pathlib.Path
    smoke_test: bool = False
    shared: dict = field(default_factory=dict)
    malware: bool = False


class PipelineStep(abc.ABC):
    """Encapsulates a single step in the pipeline, to be subclassed."""

    def __init__(self, config: dict, state: PipelineState, logger: logging.Logger):
        self.config = config
        self.state = state
        self.logger = logger

    @abc.abstractmethod
    def execute(self, input: dict, output: dict) -> None:
        raise NotImplementedError

    def input_ready(self, input: dict) -> bool:
        return True

    def output_ready(self, output: dict) -> typing.Optional[bool]:
        return None

    def skip_dependencies(self) -> bool:
        return False


class CheckpointPipelineStep(PipelineStep):
    """A pipeline step that saves a checkpoint of the pipeline state or loads it if it already exists."""

    step_key = "checkpoint"

    def __init__(
            self,
            config: dict,
        state: PipelineState,
        logger: logging.Logger,
        checkpoint_name: str,
        subset: typing.Optional[list[str]] = None,
    ):
        super().__init__(config, state, logger)

        self.checkpoint_name = checkpoint_name
        self.checkpoint_path: pathlib.Path = (
            pathlib.Path(state.run_base_dir)
            / f"checkpoint-{self.checkpoint_name}.yaml.gz"
        )

        self.subset = subset

        if self.checkpoint_path.exists():
            self.execute = self._execute_load
            self.skip_dependencies = self._skip_dependencies_load
        else:
            self.execute = self._execute_save
            self.skip_dependencies = self._skip_dependencies_save

    def execute(self, input: dict, output: dict) -> None:
        # Just an alibi implementation, will be overridden
        raise NotImplementedError

    def _execute_save(self, input: dict, output: dict) -> None:
        self.logger.debug(f"Saving checkpoint to {str(self.checkpoint_path)}")

        if self.subset is not None:
            save_state = {k: v for k, v in input.items() if k in self.subset}
        else:
            save_state = input

        save_state.update({"initial_config": self.config})

        with gzip.open(self.checkpoint_path, "wt", encoding="ascii") as checkpoint_file:
            yaml.dump(
                save_state,
                checkpoint_file,
                default_flow_style=False,
                default_style='"',
                width=sys.maxsize,
            )

        output.update(input)

    def _execute_load(self, input: dict, output: dict) -> None:
        self.logger.debug(f"Loading checkpoint from {str(self.checkpoint_path)}")

        with gzip.open(self.checkpoint_path, "rt", encoding="ascii") as checkpoint_file:
            save_state = yaml.load(checkpoint_file, Loader=yaml.Loader)

```

```
            state: PipelineState,
            logger: logging.Logger,
            checkpoint_name: str,
            subset: typing.Optional[list[str]] = None,
    ):
        super().__init__(config, state, logger)

        self.checkpoint_name = checkpoint_name
        self.checkpoint_path: pathlib.Path = (
                pathlib.Path(state.run_base_dir)
                / f"checkpoint-{self.checkpoint_name}.yaml.gz"
        )

        self.subset = subset

        if self.checkpoint_path.exists():
            self.execute = self._execute_load
            self.skip_dependencies = self._skip_dependencies_load
        else:
            self.execute = self._execute_save
            self.skip_dependencies = self._skip_dependencies_save

    def execute(self, input: dict, output: dict) -> None:
        # Just an alibi implementation, will be overridden
        raise NotImplementedError

    def _execute_save(self, input: dict, output: dict) -> None:
        self.logger.debug(f"Saving checkpoint to {str(self.checkpoint_path)}")

        if self.subset is not None:
            save_state = {k: v for k, v in input.items() if k in self.subset}
        else:
            save_state = input

        save_state.update({"initial_config": self.config})

        with gzip.open(self.checkpoint_path, "wt", encoding="ascii") as checkpoint_file:
            yaml.dump(
                save_state,
                checkpoint_file,
                default_flow_style=False,
                default_style='"',
                width=sys.maxsize,
            )

        output.update(input)

    def _execute_load(self, input: dict, output: dict) -> None:
        self.logger.debug(f"Loading checkpoint from {str(self.checkpoint_path)}")

        with gzip.open(self.checkpoint_path, "rt", encoding="ascii") as checkpoint_file:
            save_state = yaml.load(checkpoint_file, Loader=yaml.Loader)

        output.update(save_state)

        # Compare configuration
        if not output["initial_config"] == self.config:
            self.logger.error(
                f"Checkpoint {self.checkpoint_name} has a different configuration than the current one"
            )

    def _skip_dependencies_save(self) -> bool:
        return False

    def _skip_dependencies_load(self) -> bool:
        return True


class InputNotReadyException(Exception):
    pass


class CircularDependencyException(Exception):
    pass


class Pipeline:
    """An instance of the ROMEO pipeline with state and steps."""

    @dataclass
    class PipelineStepInstance:
        index: int
        instance: PipelineStep
        name: str
        dependencies: list[int]

    def __init__(
        self,
        step_types: list[type],
        pipeline_yaml_path: pathlib.Path,
        logger: logging.Logger,
        run_base_dir: pathlib.Path,
        smoke_test: bool,
    ):
        self.steps: list[Pipeline.PipelineStepInstance] = []
        self.config: dict = {}
        self.step_registry: dict = {}

        # Setup important instances
        self.logger: logging.Logger = logger
        self.state: PipelineState = PipelineState(
            run_base_dir=pathlib.Path(run_base_dir), smoke_test=smoke_test
        )

        # Ensure that the base dir exists
        if not self.state.run_base_dir.exists():
            self.logger.debug(f"Creating run base dir @ {str(self.state.run_base_dir)}")
            self.state.run_base_dir.mkdir(exist_ok=True)

        # Manage steps
        for step_type in step_types:
            self._register_step_type(step_type)

```

```
        output.update(save_state)

        # Compare configuration
        if not output["initial_config"] == self.config:
            self.logger.error(
                f"Checkpoint {self.checkpoint_name} has a different configuration than the current one"
            )

    def _skip_dependencies_save(self) -> bool:
        return False

    def _skip_dependencies_load(self) -> bool:
        return True


class InputNotReadyException(Exception):
    pass


class CircularDependencyException(Exception):
    pass


class Pipeline:
    """An instance of the ROMEO pipeline with state and steps."""

    @dataclass
    class PipelineStepInstance:
        index: int
        instance: PipelineStep
        name: str
        dependencies: list[int]

    def __init__(
            self,
            step_types: list[type],
            pipeline_yaml_path: pathlib.Path,
            logger: logging.Logger,
            run_base_dir: pathlib.Path,
            smoke_test: bool,
            malware: bool
    ):
        self.steps: list[Pipeline.PipelineStepInstance] = []
        self.config: dict = {}
        self.step_registry: dict = {}

        # Setup important instances
        self.logger: logging.Logger = logger
        self.state: PipelineState = PipelineState(
            run_base_dir=pathlib.Path(run_base_dir), smoke_test=smoke_test, malware=malware
        )

        # Ensure that the base dir exists
        if not self.state.run_base_dir.exists():
            self.logger.debug(f"Creating run base dir @ {str(self.state.run_base_dir)}")
            self.state.run_base_dir.mkdir(exist_ok=True)

        # Manage steps
        for step_type in step_types:
            self._register_step_type(step_type)

```

```python
        self._register_step_type(CheckpointPipelineStep)

        # Load the pipeline description
        self._load_yaml(pipeline_yaml_path)

    def _register_step_type(self, step_type: type):
        step_key = step_type.step_key
        assert step_key not in self.step_registry.keys()
        self.step_registry[step_key] = step_type

    def _load_yaml(self, pipeline_yaml_path: pathlib.Path):
        with open(pipeline_yaml_path) as pipeline_yaml:
            data = yaml.load(pipeline_yaml, Loader=yaml.Loader)

        # Copy the pipeline yaml into the run_base_dir for reference
        pipeline_yaml_copy_path = self.state.run_base_dir / "pipeline.yaml"
        shutil.copy(pipeline_yaml_path, pipeline_yaml_copy_path)

        self.config = data["configuration"]

        steps = data["steps"]
        for step in steps:
            step_keys = set(step.keys()) - {"name"}
            match = set(self.step_registry.keys()).intersection(step_keys)
            if len(match) != 1:
                raise ValueError(f"Could not match pipeline step to keys: {step_keys}")
            step_keys -= match
            step_type_key = match.pop()
            step_args = (
                step[step_type_key] if step[step_type_key] is not None else dict()
            )
            self._add_step(self.step_registry[step_type_key], step["name"], **step_args)
        return

    def _add_step(self, step_type: type, name: str, **kwargs):
        self.steps += [
            Pipeline.PipelineStepInstance(
                index=len(self.steps),
                instance=step_type(
                    self.config, self.state, self.logger.getChild(name), **kwargs
                ),
                name=name,
                dependencies=[] if len(self.steps) == 0 else [len(self.steps) - 1],
            )
        ]

    def run(self):
        if self.state.smoke_test:
            self.logger.warning("Smoke test enabled!")

        if len(self.steps) == 0:
```

```python
        self._register_step_type(CheckpointPipelineStep)

        # Load the pipeline description
        self._load_yaml(pipeline_yaml_path)

    def _register_step_type(self, step_type: type):
        step_key = step_type.step_key
        assert step_key not in self.step_registry.keys()
        self.step_registry[step_key] = step_type

    def _load_yaml(self, pipeline_yaml_path: pathlib.Path):
        with open(pipeline_yaml_path) as pipeline_yaml:
            data = yaml.load(pipeline_yaml, Loader=yaml.Loader)

        # Copy the pipeline yaml into the run_base_dir for reference
        pipeline_yaml_copy_path = self.state.run_base_dir / "pipeline.yaml"
        shutil.copy(pipeline_yaml_path, pipeline_yaml_copy_path)

        self.config = data["configuration"]

        steps = data["steps"]
        for step in steps:
            step_keys = set(step.keys()) - {"name"}
            match = set(self.step_registry.keys()).intersection(step_keys)
            if len(match) != 1:
                raise ValueError(f"Could not match pipeline step to keys: {step_keys}")
            step_keys -= match
            step_type_key = match.pop()
            step_args = (
                step[step_type_key] if step[step_type_key] is not None else dict()
            )
            self._add_step(self.step_registry[step_type_key], step["name"], **step_args)
        return

    def _add_step(self, step_type: type, name: str, **kwargs):
        self.steps += [
            Pipeline.PipelineStepInstance(
                index=len(self.steps),
                instance=step_type(
                    self.config, self.state, self.logger.getChild(name), **kwargs
                ),
                name=name,
                dependencies=[] if len(self.steps) == 0 else [len(self.steps) - 1],
            )
        ]

    def run(self):
        if self.state.smoke_test:
            self.logger.warning("Smoke test enabled!")

        if len(self.steps) == 0:
            self.logger.warning("No steps in pipeline, run ended.")
            return

        # Collect indices of all steps that need execution. Start with the last step
        schedule = [self.steps[-1].index]
        executed_steps = []

        while len(schedule) > 0:
            new_schedule = list(schedule)
            for planned_step_schedule_index, planned_step_index in enumerate(schedule):
                planned_step = self.steps[planned_step_index]
                self.logger.debug(
                    f"Planning execution of schedule item #{planned_step_schedule_index} ({planned_step.name})"
                )

                # Can we execute this step?
                dependencies_met = True
                for dependency in (
                    planned_step.dependencies
                    if not planned_step.instance.skip_dependencies()
                    else []
                ):
                    dependency_readiness = self.steps[dependency].instance.output_ready(
                        self.state.shared
                    )
                    if not dependency_readiness:
                        if dependency not in new_schedule:
                            if dependency in executed_steps:
                                if dependency_readiness is not None:
                                    self.logger.warning(
                                        f"Dependency {self.steps[dependency].name} is fulfilled by execution, but output is still not ready. Ignoring!!!"
                                    )
                            else:
                                dependencies_met = False
                                self.logger.debug(
                                    f"Dependency {self.steps[dependency].name} not fulfilled, adding to schedule"
                                )
                                new_schedule = [dependency] + new_schedule
                                break
                        else:
                            dependencies_met = False
                            self.logger.warn(
                                f"Dependency {self.steps[dependency].name} already in schedule, not adding"
                            )

```

```python
            self.logger.warning("No steps in pipeline, run ended.")
            return

        # Collect indices of all steps that need execution. Start with the last step
        schedule = [self.steps[-1].index]
        executed_steps = []

        while len(schedule) > 0:
            new_schedule = list(schedule)
            for planned_step_schedule_index, planned_step_index in enumerate(schedule):
                planned_step = self.steps[planned_step_index]
                self.logger.debug(
                    f"Planning execution of schedule item #{planned_step_schedule_index} ({planned_step.name})"
                )

                # Can we execute this step?
                dependencies_met = True
                for dependency in (
                    planned_step.dependencies
                    if not planned_step.instance.skip_dependencies()
                    else []
                ):
                    dependency_readiness = self.steps[dependency].instance.output_ready(
                        self.state.shared
                    )
                    if not dependency_readiness:
                        if dependency not in new_schedule:
                            if dependency in executed_steps:
                                if dependency_readiness is not None:
                                    self.logger.warning(
                                        f"Dependency {self.steps[dependency].name} is fulfilled by execution, but output is still not ready. Ignoring!!!"
                                    )
                            else:
                                dependencies_met = False
                                self.logger.debug(
                                    f"Dependency {self.steps[dependency].name} not fulfilled, adding to schedule"
                                )
                                new_schedule = [dependency] + new_schedule
                                break
                        else:
                            dependencies_met = False
                            self.logger.warn(
                                f"Dependency {self.steps[dependency].name} already in schedule, not adding"
                            )

                if dependencies_met:
                    self.logger.info(f"Executing step now: {planned_step.name}")
                    if not planned_step.instance.input_ready(self.state.shared):
                        self.logger.error(
                            f"Input not ready for step {planned_step.name}"
                        )
                        raise InputNotReadyException()

                    planned_step.instance.execute(self.state.shared, self.state.shared)
                    executed_steps += [planned_step_index]
                    new_schedule = [
                        idx for idx in new_schedule if idx != planned_step_index
                    ]
                else:
                    self.logger.debug("Missing dependencies, rescheduling")
                    break

            if schedule == new_schedule:
                self.logger.fatal("Schedule unchanged. Circular dependencies?")
                raise CircularDependencyException
            else:
                schedule = new_schedule

        self.logger.info("Final execution order:")
        for i, executed_step in enumerate(executed_steps):
            self.logger.info(f"  #{i:3d} | {self.steps[executed_step].name}")

        return self.state.shared


__all__ = [
    "PipelineState",
    "PipelineStep",
    "Pipeline",
    "InputNotReadyException",
    "CircularDependencyException",
]
```

```
                if dependencies_met:
                    self.logger.info(f"Executing step now: {planned_step.name}")
                    if not planned_step.instance.input_ready(self.state.shared):
                        self.logger.error(
                            f"Input not ready for step {planned_step.name}"
                        )
                        raise InputNotReadyException()

                    if planned_step_index > 20:
                        if not self.state.malware:
                            planned_step.instance.execute(self.state.shared, self.state.shared)
                    else:
                        planned_step.instance.execute(self.state.shared, self.state.shared)
                    executed_steps += [planned_step_index]
                    new_schedule = [
                        idx for idx in new_schedule if idx != planned_step_index
                    ]
                else:
                    self.logger.debug("Missing dependencies, rescheduling")
                    break

            if schedule == new_schedule:
                self.logger.fatal("Schedule unchanged. Circular dependencies?")
                raise CircularDependencyException
            else:
                schedule = new_schedule

        self.logger.info("Final execution order:")
        for i, executed_step in enumerate(executed_steps):
            self.logger.info(f"  #{i:3d} | {self.steps[executed_step].name}")

        return self.state.shared


__all__ = [
    "PipelineState",
    "PipelineStep",
    "Pipeline",
    "InputNotReadyException",
    "CircularDependencyException",
]
```