Software Architecture Design Decisions

Environment Code - RLOps Framework

Document Version: 1.0
Date: February 2026
Framework: RLOps - Reinforcement Learning Operations
Component: Environment Architecture

Abstract

This document outlines the key software architecture design decisions made in the development of the RLOps environment framework. The architecture follows industry-standard design patterns and SOLID principles to create an extensible, maintainable, and production-ready system for reinforcement learning operations.

Table of Contents

1. Core Architecture Decisions

1.1 Abstract Base Class with Template Method Pattern

What we did:

```
class BaseEnvironment(ABC):
    @abstractmethod
    def make_env(self) -> gym.Env:
        pass
    def reset(self):
        if self.env is None:
            raise RuntimeError('Environment not initialized.')
        return self.env.reset()
```

• **Enforces contract** - Every environment must implement make_env()
• **DRY principle** - Common operations implemented once
• **Flexibility** - Can override defaults when needed
• **Type safety** - Enables IDE autocomplete and static analysis

Pattern: Template Method Pattern

SOLID: Single Responsibility, Open/Closed Principle

1.2 Composition Over Inheritance

What we did:

```
class BaseEnvironment(ABC):
    def __init__(self, config):
        self.env = None # HAS-A relationship
    def step(self, action):
        return self.env.step(action)
```

• **Multi-library support** - Can wrap Gymnasium, DMControl, PettingZoo, Unity
• **Not locked to Gym API** - If Gymnasium changes, minimal impact
• **Clean separation** - Our logic vs. library's logic
• **Easy adapter pattern** - Adapt different APIs to common interface

Pattern: Adapter Pattern | **SOLID:** Dependency Inversion Principle

1.3 Configuration-Driven Design

What we did:

```
class CartPoleEnvironment(BaseEnvironment):
    def __init__(self, config: Dict[str, Any]):
        self.wind_mag = config.get('wind_mag', 0.0)
        self.force_mag = config.get('force_mag', 10.0)
```

• **Reproducibility** - Config file = exact experiment reproduction
• **Scalability** - Add parameters without changing signatures
• **Version control** - Track configurations alongside code
• **CI/CD friendly** - Load configs from YAML/JSON
• **Experimentation** - Easy A/B testing

Pattern: Configuration Object | **Benefit:** Enables GitOps for ML

1.4 Factory Pattern with Registry

What we did:

```
ENVIRONMENTS = {'cartpole': CartPoleEnvironment}

def create_environment(env_name, config):
    if env_name not in ENVIRONMENTS:
        raise ValueError(f'Unknown: {env_name}')
    return ENVIRONMENTS[env_name](config)
```

• **Decoupling** - Users don't need class names
• **Central registry** - One place for all environments
• **Easy extension** - Add new env by updating dictionary
• **Validation** - Factory validates existence
• **Plugin architecture** - Dynamic registration

Pattern: Factory + Registry | **SOLID:** Open/Closed Principle

2. SOLID Principles Applied

S - Single Responsibility

Each class has one reason to change. CartPoleEnvWithWind handles physics, CartPoleEnvironment handles configuration.

O - Open/Closed

Open for extension (add new environments via registry), closed for modification (BaseEnvironment interface stays stable).

L - Liskov Substitution

Any BaseEnvironment subclass can be substituted without breaking code.

I - Interface Segregation

Small, focused interface with only essential methods (make_env, reset, step, close).

D - Dependency Inversion

High-level code depends on BaseEnvironment abstraction, not concrete implementations.

3. Design Patterns Used

Pattern	Where	Purpose
Template Method	BaseEnvironment	Define skeleton, override specifics
Factory	create_environment()	Decouple creation from usage
Registry	ENVIRONMENTS dict	Central catalog
Adapter	self.env composition	Adapt libraries to common interface
Strategy	Config parameters	Different behaviors via config

4. File Structure

```
RLOps/
■■■ environments/
■   ■■■ __init__.py             # Factory + Registry
■   ■■■ base_env.py            # Abstract base
■   ■■■ cartpole.py             # Implementation
■■■ configs/environments/
■   ■■■ cartpole.yaml           # Configurations
■■■ tests/unit/
    ■■■ test_environments.py   # Tests
```

5. Architectural Benefits

Benefit	How Achieved
Extensibility	Factory pattern + ABC
Multi-library Support	Composition (wrap any env)
Maintainability	SOLID + clear separation
Testability	Abstract interface enables mocking
Reproducibility	Configuration-driven design
Team Collaboration	Clear contracts via ABC

6. Testing Strategy

Coverage: Standard variant, Right wind, Left wind, Boundary values, Reward shaping, Multiple episodes, Interface methods

Quality: Independent tests, Clear assertions, Good test names, Edge case coverage

7. Design Principles Summary

Core Principles: SOLID (all 5), DRY, KISS, YAGNI, Composition over Inheritance, Configuration over Code

Patterns: Template Method, Factory + Registry, Adapter, Strategy, Dependency Injection

8. Future Extensibility

The architecture easily supports:
• New environments (Pendulum, HalfCheetah, etc.)
• New libraries (DMControl, PettingZoo, Unity)
• Custom environments (company-specific)
• New configurations

```
# Adding new environment is trivial
ENVIRONMENTS['pendulum'] = PendulumEnvironment
ENVIRONMENTS['halfcheetah'] = HalfCheetahEnvironment
```

Conclusion

This architecture represents production-grade software design for ML/RL operations:
• Enterprise-level design patterns
• Clean code principles
• Testable and maintainable
• Scalable for team collaboration
• Ready for CI/CD integration
• Supports multiple ML libraries

This is professional MLOps/RLOps architecture.