SourceCode_and_Documentation

Project: AdaptiveChain - Multi-Agent RL for Supply Chain Optimization
Author: Sravan Kumar Kurapati
Course: INFO 7375 - Reinforcement Learning for Agentic AI Systems
Institution: Northeastern University
Date: December 2025

GitHub Repository:
https://github.com/sravankumarkurapati/INFO_7375/tree/main/adaptive-chain

---

1. COMPLETE IMPLEMENTATION WITH CLEAR CODE ORGANIZATION

1.1 Project Overview

AdaptiveChain is a multi-agent reinforcement learning system implemented in Python using PyTorch and Stable-Baselines3. The codebase follows a modular three-layer architecture separating environment simulation, intelligent agents, and multi-agent coordination. All source code is organized into logical modules with clear separation of concerns.

1.2 Code Organization

Three-Layer Architecture Implementation:

Layer 1 - Environment Simulation (src/environment/) Implements realistic supply chain dynamics using OpenAI Gymnasium standards.

Layer 2 - Intelligent Agents (src/agents/) Implements DQN agents and baseline policies for comparison.

Layer 3 - Multi-Agent Coordination (integrated in agents and environment) Implements coordination protocols, state sharing, and inventory transfer mechanisms.

1.3 Directory Structure

```
ADAPTIVE-CHAIN/
│
├── src/                              # All source code modules
│   │
│   ├── agents/                       # Layer 2: Intelligent Agents
│   │   ├── baseline_policies.py      # Random, Reorder Point, EOQ policies
│   │   ├── dqn_agent.py              # DQN implementation with SB3
│   │   ├── independent_agents.py     # Multi-agent without coordination
│   │   └── coordinated_agents.py     # Multi-agent with transfers
│   │
│   ├── environment/                  # Layer 1: Environment Simulation
│   │   ├── supply_chain_env.py       # Single warehouse Gym environment
│   │   ├── multi_warehouse_env.py    # Multi-warehouse environment
│   │   ├── warehouse.py              # Warehouse entity class
│   │   ├── product.py                # Product entity class
│   │   ├── data_generator.py         # Demand and disruption generation
│   │   ├── env_wrappers.py           # Action space flattening wrapper
│   │   └── viz_utils.py              # Visualization utilities
│   │
│   └── evaluation/                   # Testing and Analysis
│       ├── visualizations.py         # Learning curves, comparisons
│       ├── statistical_analysis.py   # T-tests, ANOVA, Cohen's d
│       ├── scenario_testing.py       # Disruption scenario testing
│       ├── scenario_visualizations.py # Scenario plots
│       └── ablation_study.py         # Feature importance analysis
├── models/                           # Saved DQN models (.zip files)
├── data/                             # Results and visualizations
├── tests/                            # Unit and integration tests
├── notebooks/                        # Jupyter analysis notebooks
│
├── Root-Level Scripts:               # Entry points for training/testing
│   ├── train_dqn.py                  # Train single warehouse DQN
│   ├── train_multi_agent.py          # Train multi-agent systems
│   ├── evaluate_baselines.py         # Evaluate classical policies
│   ├── test_environment.py           # Environment unit tests
│   ├── test_multi_warehouse.py       # Multi-warehouse tests
│   ├── test_data_generator.py        # Data generation tests
│   ├── test_entities.py              # Entity class tests
│   └── app.py                        # Streamlit interactive dashboard
├── requirements.txt                  # Python dependencies
├── README.md                         # Project documentation
└── .gitignore                        # Git ignore rules
```

1.4 Module Descriptions

1.4.1 Environment Module (src/environment/)

supply_chain_env.py - Single Warehouse Environment

Purpose: OpenAI Gym-compatible environment for single warehouse inventory management.

Key Implementation:

- State space: 13-dimensional (inventory, pending orders, demand forecast, days until delivery, capacity utilization)
- Action space: MultiDiscrete with $4^3 = 64$ combinations (order quantities: 0, 100, 200, 500 units per product)
- Reward: Negative cost (holding + stockout + ordering)
- Episode: 180 days with demand generated from Poisson distribution
- Products: 3 SKUs with different demand patterns

Critical Methods:

- `reset()`: Initialize warehouse with target inventory levels
- `step(action)`: Execute orders, fulfill demand, calculate costs, advance day
- `_get_observation()`: Build normalized state vector
- `get_episode_summary()`: Return cost breakdown and statistics

multi_warehouse_env.py - Multi-Warehouse Coordination

Purpose: Extends single warehouse to 3 interconnected warehouses with coordination capabilities.

Key Innovation:

- State space: 57-dimensional (19 features per warehouse × 3)
- Regional demand multipliers: East (1.3×), West (0.8×), Central (1.0×)
- Inventory transfer mechanism with configurable cost ($5/unit)
- Proactive weekly rebalancing + emergency transfers during stockouts

Critical Methods:

- `_balance_inventory_proactive()`: Weekly inventory rebalancing across warehouses
- `_emergency_transfer()`: Emergency transfers during stockouts (2× cost)
- `_calculate_system_imbalance()`: Penalty for unbalanced inventory distribution
- `_get_regional_demand()`: Apply regional multipliers to base demand

warehouse.py - Warehouse Entity

Purpose: Manages individual warehouse inventory, orders, and capacity.

Key Features:

- Inventory tracking per SKU
- Pending order management with arrival days
- Capacity constraints (5000-7000 units)
- Capacity violation tracking

Critical Methods:

- `add_inventory()`: Add stock with capacity check
- `remove_inventory()`: Fulfill demand (returns actual quantity removed)
- `place_order()`: Create pending order with arrival day
- `receive_orders()`: Process arriving orders, handle capacity violations

product.py - Product Entity

Purpose: Defines product characteristics and demand generation.

Key Attributes:

- SKU identifier, base demand, demand standard deviation
- Cost structure: holding ($1.5-$3/day), stockout ($50-$100/day), ordering ($75-$150), unit cost ($15-$35)
- Lead time (2-5 days)

Critical Methods:

- `generate_demand()`: Generate daily demand with weekly seasonality (weekends +20%)
- `__post_init__()`: Validate all cost parameters positive

data_generator.py - Realistic Data Generation

Purpose: Generate training data with demand patterns and disruptions.

Key Classes:

- `DemandPatternGenerator`: Multi-component demand (base, trend, weekly/monthly seasonality, noise, promotions)
- `DisruptionGenerator`: Random disruptions (supplier delays, demand spikes, capacity reductions, transportation issues)

Critical Methods:

- `generate_demand_series()`: Combine all demand components into realistic time series
- `generate_multi_product_demand()`: Create demand for PROD_A, PROD_B, PROD_C
- `generate_disruptions()`: Random disruption events with 5% probability
- `create_test_scenarios()`: 5 predefined test scenarios (normal, high demand, supplier crisis, demand shock, capacity crisis)

env_wrappers.py - Action Space Wrapper

Purpose: Flatten MultiDiscrete action space to Discrete for DQN compatibility.

Key Implementation:

- Converts MultiDiscrete([4,4,4]) → Discrete(64)
- Enables DQN to work with multi-product decisions
- Reversible mapping: flat action ↔ multi-dimensional action

Critical Methods:

- `_flatten_action()`: Convert integer to product-specific actions
- `step()`: Unflatten action, execute in base environment

viz_utils.py - Visualization Utilities

Purpose: Plot demand patterns, statistics, and disruptions.

Functions:

- `plot_demand_patterns()`: Time series of demand for all products
- `plot_demand_statistics()`: Box plots and summary statistics
- `visualize_disruptions()`: Timeline visualization of disruption events

1.4.2 Agents Module (src/agents/)

baseline_policies.py - Classical Baselines

Purpose: Implement traditional inventory policies for comparison.

Implemented Policies:

1. RandomPolicy

- Selects order quantities uniformly at random

- Worst-case baseline (mean cost: $5.30M)

2. ReorderPointPolicy (s,Q)

- Classic inventory management policy
- Reorder point: s = mean_demand × lead_time + safety_stock (z=1.65 for 95% service level)
- Order quantity: Q calculated using EOQ formula
- Best performer (mean cost: $1.06M)

3. EOQPolicy

- Economic Order Quantity formula: Q* = √(2DS/H)
- Periodic review (every 7 days)
- Orders EOQ when inventory below threshold
- Good performance (mean cost: $1.94M)

Critical Methods:

- `select_action()`: Determine order quantities based on policy logic
- `reset()`: Reset policy state for new episode

dqn_agent.py - Deep Q-Network Implementation

Purpose: Train DQN agents using Stable-Baselines3 library.

Key Configuration:

- Neural network: [512, 512, 256] architecture with ReLU + BatchNorm
- Optimizer: Adam with learning rate 0.0003
- Experience replay: 100K buffer, batch size 128
- Target network: Soft updates (τ=0.005) every 100 steps
- Exploration: ε-greedy from 1.0 → 0.1 over 50% of training

Critical Functions:

- `create_dqn_agent()`: Initialize DQN with optimized hyperparameters
- `train_dqn_agent()`: Execute training loop with callbacks
- `evaluate_dqn_agent()`: Test trained agent over multiple episodes
- `TrainingCallback`: Custom callback tracking episode rewards and costs
- `linear_schedule()`: Learning rate schedule function

Results:

- Training: ~1,110 episodes (200K timesteps)
- Final cost: $2.27M

- 57% improvement over random, but 114% worse than reorder point

independent_agents.py - Multi-Agent Without Coordination

Purpose: Three independent DQN agents, one per warehouse, no communication.

Key Implementation:

- Single DQN policy trained on single-warehouse environment
- Same policy applied to all 3 warehouses (independent decisions)
- No state sharing between warehouses
- Transfers only when critically needed (51 per episode)

Critical Methods:

- `train()`: Train single policy on single-warehouse environment
- `predict()`: Extract single-warehouse state, apply policy, replicate for all warehouses
- State extraction handles 57-dim input → 13-dim single-warehouse state

Results:

- Training: 100K timesteps
- Final cost: $5.55M
- Conservative, effective approach

coordinated_agents.py - Multi-Agent With Coordination

Purpose: Single coordinated policy seeing all warehouses (communication + transfers).

Key Innovation:

- Single DQN policy with 57-dimensional state (sees ALL warehouses)
- Larger network: [512, 512, 256] to handle coordination complexity
- Environment handles transfers (proactive weekly + emergency)
- Coordination reward: Penalizes inventory imbalance

Critical Implementation:

- Wrapped multi-warehouse environment with transfers enabled
- Single policy optimizes global system cost
- State includes all warehouses' inventory (full communication)

Results:

- Training: 833 episodes (150K timesteps)
- Final cost: $12.91M
- **Failed: 133% worse than independent**

- Root cause: 326 transfers/episode (6.3× excessive), $163K overhead

1.4.3 Evaluation Module (src/evaluation/)

visualizations.py - Results Visualization

Purpose: Generate all comparison charts and learning curves.

Key Functions:

- `load_all_results()`: Load all JSON result files
- `plot_complete_comparison()`: Bar chart comparing all 8 approaches
- `plot_coordination_analysis()`: Multi-agent comparison with breakdown
- `plot_learning_curves_comparison()`: DQN and multi-agent learning over episodes
- `plot_all_policies_heatmap()`: Performance across multiple dimensions
- `create_summary_table()`: Comprehensive results CSV

Outputs:

- complete_comparison.png
- coordination_analysis.png
- learning_curves.png
- policy_heatmap.png
- results_summary.csv

statistical_analysis.py - Statistical Validation

Purpose: Rigorous statistical testing of results.

Key Functions:

- `calculate_confidence_interval()`: 95% CI using Student's t-distribution
- `perform_ttest()`: Independent t-tests with Cohen's d effect sizes
- `run_complete_statistical_analysis()`: Execute all statistical tests
- `create_statistical_summary_table()`: Summary with CIs

Statistical Tests Performed:

- Paired t-tests: Random vs Reorder Point ($t=43.26$, $p<0.001$, $d=19.35$)
- Random vs DQN ($t=30.91$, $p<0.001$, $d=13.82$)
- Independent vs Coordinated (coordinated significantly worse)
- One-way ANOVA across single-warehouse policies

Outputs:

- statistical_analysis.json
- statistical_summary.csv

scenario_testing.py - Disruption Scenario Testing

Purpose: Test all agents across 5 realistic disruption scenarios.

Implemented Scenarios:

1. Normal Operations (baseline)
2. High Demand (1.5× multiplier)
3. Supplier Crisis (2× lead time)
4. Demand Shock (3× spike)
5. Capacity Crisis (0.5× capacity)

Key Class:

- `ScenarioTester`: Manages all scenario testing
  - `test_baseline_on_scenario()`: Test classical policies
  - `test_dqn_on_scenario()`: Test DQN agent
  - `test_multiagent_on_scenario()`: Test multi-agent systems
  - `_apply_scenario_modifications()`: Modify environment parameters
  - `run_all_scenarios()`: Execute complete test suite

Results:

- 25 total tests (5 scenarios × 5 policies)
- Reorder point wins all 5 scenarios
- Coordinated worst in all 5 scenarios

Outputs:

- scenario_results.csv (detailed)
- scenario_results_pivot.csv (comparison table)
- scenario_testing_results.json

scenario_visualizations.py - Scenario Plots

Purpose: Visualize performance across scenarios.

Key Functions:

- `plot_scenario_heatmap()`: Agent performance heatmap (green=better)
- `plot_scenario_comparison_bars()`: Grouped bar charts per scenario
- `plot_ablation_visualization()`: Transfer feature impact charts

Outputs:

- scenario_heatmap.png
- scenario_comparison_bars.png
- ablation_visualization.png

ablation_study.py - Feature Importance

Purpose: Isolate impact of coordination features.

Ablation Tests:

1. Coordinated agent WITH transfers enabled
2. Coordinated agent WITHOUT transfers (disabled in environment)
3. Compare cost difference to isolate transfer contribution

Key Findings:

- WITH transfers: $12.73M (326 transfers)
- WITHOUT transfers: $14.19M (0 transfers)
- Transfer benefit: $1.46M (10.3% savings)
- Conclusion: Transfers work, but agent over-uses them

Critical Implementation:

- `evaluate_agent()`: Run episodes and collect costs
- `run_ablation_study()`: Execute both configurations
- Loads same trained model, tests on different environments

Output:

- ablation_study_results.json

1.4.4 Training Scripts (Root Level)

train_dqn.py - Single Warehouse Training

Purpose: Train DQN agent for single warehouse.

Training Configuration:

- Total timesteps: 200,000 (~1,110 episodes)
- Environment: 3 products, 180-day episodes
- Network: [512, 512, 256]
- Optimizations: Prioritized replay, LR schedule, larger buffer (100K)

Training Process:

1. Create training environment with wrapper
2. Create DQN agent with optimized hyperparameters
3. Train with TrainingCallback for progress tracking
4. Evaluate on 10 test episodes
5. Save model and results JSON

Output:

- models/dqn_optimized.zip
- models/dqn_optimized_results.json
- Training time: ~40 minutes

train_multi_agent.py - Multi-Agent Training

Purpose: Train independent and coordinated multi-agent systems.

Training Pipeline:

Phase 1: Independent Agents

- Train single DQN on single-warehouse environment
- Apply same policy to all 3 warehouses
- 100K timesteps
- Output: models/independent_multi_agent.zip

Phase 2: Coordinated Agents (with checkpoint resume)

- Train single coordinated policy on full multi-warehouse state
- Checkpoint every 10K steps (crash protection)
- Resume from latest checkpoint if available
- 150K timesteps
- Output: models/coordinated_multi_agent.zip

Key Features:

- `find_latest_checkpoint()`: Resume interrupted training
- `evaluate_multi_agent()`: Test on 10 episodes
- `display_results()`: Compare independent vs coordinated
- Automatic comparison with baselines

Outputs:

- models/independent_multi_agent.zip
- models/coordinated_multi_agent.zip
- models/checkpoints/ (15 checkpoint files)
- data/multi_agent_results.json

evaluate_baselines.py - Baseline Evaluation

Purpose: Evaluate all classical policies.

Implementation:

- Creates supply chain environment
- Instantiates Random, Reorder Point, EOQ policies
- Runs each for 10 episodes
- Calculates statistics (mean, std, min, max)
- Creates comparison visualizations

Outputs:

- data/baseline_results.json
- data/baseline_comparison.png
- data/baseline_cost_distribution.png

app.py - Interactive Dashboard

Purpose: Streamlit web application for visualization and live simulation.

Pages Implemented:

1. **Overview**: Performance comparison, key insights
2. **Learning Curves**: Training progress visualization
3. **Live Simulation**: Run any agent for custom episode length
4. **Scenario Analysis**: View disruption scenario results
5. **Ablation Study**: Transfer feature impact
6. **Technical Details**: Hyperparameters, statistics download

Key Functions:

- `run_baseline_simulation()`: Execute classical policy
- `run_dqn_simulation()`: Execute DQN agent
- `run_multiagent_simulation()`: Execute multi-agent system
- `display_simulation_results()`: Real-time charts (cumulative cost, inventory levels, daily costs)

Launch:

bash
streamlit run app.py
```

Opens at http://localhost:8501
```

1.4.5 Test Scripts

test_environment.py
- Tests Gymnasium environment creation
- Validates reset() functionality
- Tests random actions (10 steps)
- Runs full 180-day episode
- Checks Gymnasium API compatibility

test_multi_warehouse.py
- Tests multi-warehouse environment
- Validates regional demand variation
- Tests transfer mechanism
- Checks coordination features

test_data_generator.py
- Tests demand generation (Poisson with spikes)
- Tests multi-product demand
- Tests disruption generation
- Creates and validates visualizations

test_entities.py
- Tests Product class (demand generation, validation)
- Tests Warehouse class (inventory operations, orders, capacity)

---

2. DOCUMENTATION OF REINFORCEMENT LEARNING APPROACH**

2.1 Problem Formulation

Markov Decision Process:

The supply chain inventory problem is formulated as an MDP (S, A, P, R, γ) where agents learn optimal ordering policies through trial and error.

State Space (S):

Single Warehouse (13-dimensional):
- Inventory levels (3): Current stock per product
- Pending orders (3): Orders in transit

- Demand forecast (3): 7-day average per product
- Days until delivery (3): Time to next order arrival
- Capacity utilization (1): Warehouse fullness

Multi-Warehouse (57-dimensional):
- Own warehouse state (13)
- Repeated for 3 warehouses
- Plus neighbor inventory visibility (communication)

All features normalized to [0,1] for neural network stability.

Action Space (A):

Single Warehouse:
- MultiDiscrete([4, 4, 4]) = 64 combinations
- Per product: {0, 100, 200, 500} units
- Flattened to Discrete(64) for DQN

Multi-Warehouse:
- MultiDiscrete([4,4,4,4,4,4,4,4,4]) = 262,144 combinations
- 3 warehouses × 3 products × 4 quantities
- Flattened to Discrete(262,144)

Transition Function (P):

Stochastic state transitions based on:
- Demand: Generated from data_generator with Poisson distribution, promotional spikes, disruptions
- Order arrivals: Arrive after lead time (2-5 days depending on product)
- Inventory dynamics: Current + Arrivals - Demand + Transfers_in - Transfers_out
- Capacity constraints: Orders rejected if exceed warehouse capacity

Reward Function (R):
```
R(s, a) = -(Σ holding_cost_i × inventory_i +
          Σ stockout_cost_i × stockout_i +
          Σ order_cost_i × 𝕀[order_i > 0] +
          transfer_cost × total_transfers +
          imbalance_penalty × inventory_variance)
```

Cost Values (from product.py):
- Holding: $1.5-$3 per unit per day
- Stockout: $50-$100 per unit per day (10-30× holding)

- Ordering: $75-$150 per order + unit costs
- Transfer: $5 per unit (multi-warehouse only)

Discount Factor (γ):
- γ = 0.99
- Effective horizon: 1/(1-γ) = 100 days
- Encourages long-term planning

2.2 Deep Q-Network Algorithm

Q-Function Approximation:
```
Q(s, a; θ) ≈ Q*(s, a)
```

Neural network with parameters θ maps states to Q-values for all actions.

Network Architecture:
- Input: 13-dim (single) or 57-dim (multi-warehouse)
- Hidden layer 1: 512 neurons, ReLU, BatchNorm
- Hidden layer 2: 512 neurons, ReLU, BatchNorm
- Hidden layer 3: 256 neurons, ReLU, BatchNorm
- Output: 64 or 262,144 Q-values (one per action)
- Total parameters: ~1.2 million

Training Algorithm:

Experience Replay:
- Buffer capacity: 100,000 transitions
- Stores: (state, action, reward, next_state, done)
- Sampling: Uniform random, batch size 128
- Purpose: Break temporal correlations

Target Network:
- Frozen copy providing stable targets
- Soft updates: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$ where $\tau=0.005$
- Update frequency: Every 100 training steps
- Prevents moving target problem

Exploration Strategy:
- ε-greedy policy
- Initial: ε = 1.0 (pure exploration)
- Final: ε = 0.1 (90% exploitation, 10% exploration)
- Decay: Linear over 50% of training (100K steps)

Optimization:
- Adam optimizer (β_1=0.9, β_2=0.999)
- Learning rate: 0.0003 (with optional linear schedule)
- Loss: MSE between Q(s,a) and target y = r + γ max_a' Q(s',a';θ^-)
- Gradient clipping: Max norm 10.0

Training Loop (from train_dqn.py):
1. Sample action using ε-greedy
2. Execute in environment
3. Store transition in replay buffer
4. Sample batch of 128 transitions
5. Compute TD targets using target network
6. Backpropagation with gradient clipping
7. Soft update target network every 100 steps
8. Decay ε

2.3 Multi-Agent Coordination Protocol

Communication Mechanism:

State Sharing (implemented in multi_warehouse_env.py):
- Each warehouse's state includes neighbor inventory levels
- 57-dim state = own state (19) × 3 warehouses
- Partial observability: Only see inventory, not demand patterns (privacy)
- Update frequency: Every timestep

Transfer Mechanism:

Proactive Weekly Balancing (multi_warehouse_env.py: `_balance_inventory_proactive()`):
- Executes every 7 days
- Compares inventory to regional targets
- Transfers from excess (>1.5× target) to shortage (<0.5× target)
- Maximum 100 units per transfer
- Cost: $5 per unit

Emergency Transfers (`_emergency_transfer()`):
- Triggered during stockouts
- Requests inventory from neighbors with surplus
- Only if neighbor has >2× base demand
- Cost: $10 per unit (2× normal, emergency premium)

Coordination Reward:
```
```

R_system = Σ R_i - λ × ImbalancePenalty

Imbalance Penalty:

- Calculated as deviation from regional targets
- Small coefficient (0.05-0.1)
- Encourages balanced distribution

Why It Failed:

- Weekly balancing + emergency transfers → 326 transfers/episode
- Should have been ~50 transfers (similar to independent's 51)
- $5/unit × 326 transfers × ~100 units avg = $163K overhead
- Coordination complexity prevented convergence

---

3. INSTALLATION AND SETUP INSTRUCTIONS

3.1 System Requirements

Hardware:

- CPU: MacBook Air M1/M2/M3 or equivalent x86_64
- RAM: 8GB minimum (16GB recommended)
- Disk: 2GB free space
- GPU: Optional (CPU training is sufficient)

Software:

- Python 3.8 or higher
- pip package manager
- Git for repository cloning

Operating Systems:

- macOS (tested on M1/M2/M3)
- Linux (Ubuntu 20.04+)
- Windows 10/11

3.2 Installation Steps

Step 1: Clone Repository

bash

```
git clone https://github.com/yourusername/adaptive-chain.git
cd adaptive-chain
```

Step 2: Create Virtual Environment

```
bash
# Create virtual environment
python3 -m venv venv

# Activate (macOS/Linux)
source venv/bin/activate

# Activate (Windows)
venv\Scripts\activate
```

Step 3: Install Dependencies

```
bash
# Install all required packages
pip install -r requirements.txt

# Verify PyTorch installation
python -c "import torch; print(f'PyTorch {torch.__version__} installed')"
```

Step 4: Verify Installation

```
bash
# Run all unit tests
python test_environment.py
python test_multi_warehouse.py
python test_data_generator.py
python test_entities.py

# Expected output: All tests pass with ✅
```

3.3 Dependencies (requirements.txt)

Core RL Framework:

- gymnasium==0.29.1 (OpenAI Gym successor)
- stable-baselines3==2.2.1 (DQN implementation)
- torch>=2.6.0 (Neural networks)
- tensorboard>=2.15.0 (Training visualization)

Scientific Computing:

- numpy>=1.24.0 (Numerical operations)
- pandas>=2.1.0 (Data manipulation)
- scipy>=1.11.0 (Statistical tests)

Visualization:

- matplotlib>=3.8.0 (Plotting)
- seaborn>=0.13.0 (Statistical plots)
- plotly>=5.17.0 (Interactive charts)
- streamlit (Web dashboard)

Utilities:

- tqdm>=4.66.0 (Progress bars)
- pyyaml>=6.0.1 (Configuration files)

3.4 Quick Start

Train Single DQN Agent:

bash

```
python train_dqn.py
```

Expected output:

- Training for ~40 minutes
- Progress updates every 10 episodes
- Final model saved to models/dqn_optimized.zip
- Results saved to models/dqn_optimized_results.json

Train Multi-Agent System:

bash

```
python train_multi_agent.py
```

Expected output:

- Phase 1: Train independent agents (~30 min)
- Phase 2: Train coordinated agents (~50 min)
- Checkpoints saved every 10K steps
- Final models and results saved

Evaluate Baselines:

bash

```
python evaluate_baselines.py
```

Expected output:

- Tests Random, Reorder Point, EOQ policies
- 10 episodes each
- Results JSON and comparison plots
- Runtime: ~5 minutes

Launch Dashboard:

bash

```
streamlit run app.py
```

Expected output:

- Dashboard opens at http://localhost:8501
- Interactive visualization of all results
- Live agent simulation capability

3.5 Running Complete Experimental Pipeline

Full Reproduction (from scratch):

bash

```
# 1. Evaluate baselines (~5 min)
python evaluate_baselines.py

# 2. Train single DQN (~40 min)
python train_dqn.py

# 3. Train multi-agent (~90 min)
python train_multi_agent.py

# 4. Run scenario tests (~15 min)
python src/evaluation/scenario_testing.py

# 5. Run ablation study (~10 min)
python src/evaluation/ablation_study.py

# 6. Generate visualizations (~2 min)
python src/evaluation/visualizations.py
```

```
# 7. Statistical analysis (~1 min)
python src/evaluation/statistical_analysis.py

# Total time: ~2.5 hours
```

3.6 Configuration

Hyperparameters (src/agents/dqn_agent.py):

```
python
# Neural Network
NETWORK_ARCHITECTURE = [512, 512, 256]
ACTIVATION = 'relu'
BATCH_NORMALIZATION = True

# Training
LEARNING_RATE = 0.0003
GAMMA = 0.99
BUFFER_SIZE = 100000
BATCH_SIZE = 128

# Exploration
EPSILON_START = 1.0
EPSILON_END = 0.1
EXPLORATION_FRACTION = 0.5

# Target Network
TAU = 0.005
TARGET_UPDATE_FREQ = 100
```

Environment Parameters (src/environment/):

```
python
# Warehouse
CAPACITY = 5000  # units
NUM_WAREHOUSES = 3
REGIONAL_MULTIPLIERS = [1.3, 0.8, 1.0]  # East, West, Central

# Products
PROD_A: base_demand=100, std=15, holding=$2, stockout=$50
PROD_B: base_demand=50, std=10, holding=$3, stockout=$80
PROD_C: base_demand=20, std=5, holding=$1.5, stockout=$100
```

```
# Episode
EPISODE_LENGTH = 180  # days
ACTION_QUANTITIES = [0, 100, 200, 500]  # units

# Coordination
ENABLE_TRANSFERS = True/False
TRANSFER_COST = $5 per unit
```

---

4. TEST ENVIRONMENT AND SIMULATION FRAMEWORK

4.1 Environment Architecture

Framework: OpenAI Gymnasium (successor to Gym)

Environment Hierarchy:

1. `SupplyChainEnv` (base): Single warehouse with 3 products
2. `MultiWarehouseEnv` (extends base): 3 warehouses with coordination
3. `FlattenMultiDiscreteWrapper`: Action space adapter for DQN

Design Pattern: Modular entity-based architecture

- `Product` class: Demand patterns, costs, lead times
- `Warehouse` class: Inventory, orders, capacity
- `DemandPatternGenerator`: Realistic stochastic demand
- `DisruptionGenerator`: Supply chain disruptions

4.2 Simulation Parameters

Warehouse Configuration:

- Number: 3 (WH_EAST, WH_WEST, WH_CENTRAL)
- Capacity: 5000 units (East/West), 7000 units (Central)
- Initial inventory: Regional targets (East 1.3×, West 0.8×, Central 1.0×)
- Network topology: All-to-all (each can transfer to any other)

Product Characteristics (from actual data_generator.py output):

Product	Mean Demand	Std Dev	Min	Max
PROD_A	123.14	32.24	67.18	268.81
PROD_B	51.29	13.43	23.96	149.12
PROD_C	23.25	8.04	6.52	58.49

Demand Generation (data_generator.py):

- Base: Product-specific Poisson distribution
- Trend: 0.1% daily (PROD_A), -0.05% (PROD_B), 0% (PROD_C)
- Weekly seasonality: Weekends +20% demand
- Monthly seasonality: End of month +7.5%
- Promotions: 5% probability of 2× spike
- Random noise: Normal distribution

Lead Times (product.py):

- PROD_A: 3 days
- PROD_B: 5 days
- PROD_C: 2 days
- Disruption mode: 2× lead time during supplier crisis

Episode Structure:

- Length: 180 days (6 months simulation)
- Training episodes: 500-1,110 depending on approach
- Warm-up: None (start learning from day 1)

4.3 Transition Dynamics

Daily Simulation Loop (supply_chain_env.py: step()):

1. Receive Orders

- Check pending orders for arrivals today
- Add to inventory if capacity available
- Track capacity violations if rejected

2. Fulfill Demand

- Generate daily demand per product
- Fulfill from available inventory
- Record stockouts if insufficient

3. Execute Transfers (multi-warehouse only)

- Proactive balancing (every 7 days)
- Emergency transfers (during stockouts)
- Track transfer count and costs

4. Place New Orders

- Agent selects order quantities
- Create pending orders with arrival day = current + lead_time
- Calculate ordering costs

5. Calculate Costs

- Holding: inventory × holding_cost
- Stockout: unfulfilled × stockout_cost
- Order: order_cost + (quantity × unit_cost)
- Transfer: units_transferred × $5

6. Compute Reward

- reward = -total_cost
- Return to agent for learning

7. Advance Day

- current_day += 1
- Check if episode complete (day >= 180)

4.4 Test Scenarios

5 Disruption Scenarios (implemented in scenario_testing.py):

1. Normal Operations

- Standard demand patterns
- No modifications
- Baseline performance

2. High Demand

- 1.5× demand multiplier
- Tests adaptation to sustained increase
- Agents must increase safety stock

3. Supplier Crisis

- 2× lead time multiplier
- Tests pipeline management
- Requires earlier ordering

4. Demand Shock

- 3× demand spike
- Extreme volatility
- Tests robustness limits

5. Capacity Crisis

- 0.5× capacity reduction
- Tests resource constraints
- Forces efficient inventory management

Scenario Modification Process:

- Load base environment
- Apply multipliers to demand/lead time/capacity
- Run 5 episodes per scenario
- Record performance metrics

4.5 Performance Metrics

Primary Metrics (tracked in all environments):

1. **Total Cost**: Sum of all costs over 180-day episode
2. **Holding Cost**: Inventory × holding_cost_per_day
3. **Stockout Cost**: Unfulfilled_demand × stockout_cost
4. **Order Cost**: Number_of_orders × order_cost + units × unit_cost
5. **Transfer Cost**: Transfers × $5/unit (multi-warehouse)
6. **Transfer Count**: Number of inventory movements

Derived Metrics:

- Average daily cost: Total / 180
- Capacity utilization: inventory / capacity
- Fill rate: fulfilled_demand / total_demand (not explicitly tracked but calculable)

Statistical Measures (statistical_analysis.py):

- Mean ± Standard Deviation across episodes
- 95% Confidence Intervals
- Coefficient of Variation (stability measure)

4.6 Testing Framework

Unit Tests:

test_environment.py:

- Environment creation and initialization
- State space bounds checking
- Action space validation
- Reward calculation correctness
- Episode termination logic

test_multi_warehouse.py:

- Multi-warehouse initialization
- Regional demand variation (1.3×, 0.8×, 1.0×)
- Transfer mechanism execution
- Coordination feature validation

test_data_generator.py:

- Demand generation (Poisson properties)
- Disruption generation (types and frequencies)
- Multi-product demand creation
- Visualization function testing

test_entities.py:

- Product class methods (demand generation, validation)
- Warehouse class methods (inventory add/remove, orders, capacity)

Integration Tests:

- Full episode simulation with random policy
- Model save/load functionality
- Multi-agent coordination workflow

All tests ensure:

- Reproducibility (fixed seeds)

- Correctness (validation checks)
- Gymnasium API compliance

4.7 Reproducibility

Random Seed Control:

- Environment reset: seed parameter
- NumPy operations: np.random.seed()
- PyTorch: torch.manual_seed()
- Evaluation episodes: seeds 42, 43, 44, ... 51

Deterministic Operations:

```
python
torch.use_deterministic_algorithms(True)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

Fixed Configuration:

- All hyperparameters in code (no external config files modified)
- Episode length: Always 180 days
- Products: Always PROD_A, PROD_B, PROD_C with same parameters
- Action quantities: Always [0, 100, 200, 500]

SUMMARY

This document provides complete documentation of AdaptiveChain's implementation:

Section 1: Code organization across 3 architectural layers with 22 source files, comprehensive directory structure, and clear module responsibilities.

Section 2: Complete RL approach documentation including MDP formulation, DQN algorithm, multi-agent coordination protocol, and why coordination failed (6.3× excessive transfers, $163K overhead).

Section 3: Step-by-step installation (4 steps), dependency management, quick start guide, configuration options, and full reproduction pipeline (~2.5 hours).

Section 4: Detailed test environment specification including architecture, parameters, transition dynamics, 5 test scenarios, comprehensive metrics, testing framework, and reproducibility measures. **All implementation details reference actual code files and experimental results.**