Complete Project Guide: Understanding Every Component

🎯 Project Overview

Your project is a **modular RL research framework** designed to test different combinations of algorithms (PPO, Domain Randomization, SR2L) on a quadruped robot. Think of it as a scientific experiment where you're testing 4 different "recipes" to see which makes the most robust robot.

```
Your 4 Experiments (Ablations):
1. PPO only (baseline)
2. PPO + Domain Randomization (DR)
3. PPO + Smooth Regularization (SR2L)
4. PPO + DR + SR2L (full method)
```

📁 Directory Structure Explained

1. configs/ - The Brain of Your Experiments

```
configs/
├── experiments/      # Your 4 main experiments
│   ├── ppo_baseline.yaml     # Experiment 1: Just PPO
│   ├── ppo_dr.yaml           # Experiment 2: PPO + DR
│   ├── ppo_sr2l.yaml         # Experiment 3: PPO + SR2L
│   └── ppo_dr_sr2l.yaml      # Experiment 4: Everything
├── env/              # Environment settings
│   └── realant.yaml          # Robot configuration
├── train/            # Training settings
│   └── default.yaml          # Base hyperparameters
└── eval/             # Evaluation settings
    └── default.yaml          # How to test your robot
```

What you'll modify:

- Hyperparameters in `train/default.yaml`
- Fault settings in experiment files
- Environment rewards in `env/realant.yaml`

2. src/ - The Code That Runs Everything

```
src/
├── agents/          # RL algorithms
│   ├── ppo.py           # Standard PPO implementation
│   └── sr2l_ppo.py      # PPO + SR2L (smooth regularization)
├── envs/            # Robot environment
│   ├── realant_env.py     # Main robot environment
│   ├── fault_injection.py   # Breaks joints, adds noise
│   └── curriculum.py      # Gradually increases difficulty
├── utils/           # Helper tools
│   ├── config.py         # Loads YAML configs
│   ├── logger.py         # Logs to W&B, TensorBoard
│   └── metrics.py        # Calculates success rates
└── train.py          # Main script you'll run
```

What you'll modify:

- Add SR2L logic to `sr2l_ppo.py`
- Implement fault scenarios in `fault_injection.py`
- Define curriculum phases in `curriculum.py`

3. experiments/ - Where Results Live

```
experiments/
└── ppo_dr_sr2l_2025_07_30_143052/  # Auto-generated folder
    ├── logs/                  # Training logs
    │   ├── progress.csv          # Raw numbers
    │   └── tensorboard/          # TB files
    ├── models/                # Saved neural networks
    │   ├── checkpoint_1000.pt      # Early model
    │   └── best_model.pt         # Best performance
    ├── videos/                # Robot recordings
    └── metrics/               # Evaluation results
        └── evaluation_results.json  # Final scores
```

What happens here:

- Automatically created for each run
- All results saved with timestamps
- Easy to compare experiments

🔄 Complete Workflow

Phase 1: Local Development (Your MacBook)

```mermaid
graph LR
    A[Write Code] --> B[Test Locally]
    B --> C{Works?}
    C -->|No| A
    C -->|Yes| D[Sync to Cluster]
```

1. **Edit code** in `src/`
2. **Run quick test**:

```bash
python src/train.py --config configs/experiments/ppo_baseline.yaml \
                --override train.total_timesteps=1000 \
                --override train.num_envs=1
```

3. **Check it doesn't crash** (takes ~1 minute)

Phase 2: Cluster Training (Full Experiments)

```mermaid
graph LR
    A[Sync Code] --> B[Submit Job]
    B --> C[Train 10M Steps]
    C --> D[Monitor W&B]
    D --> E[Download Results]
```

1. **Sync to cluster**:

```bash
./scripts/sync_to_cluster.sh
```

2. **Submit SLURM job**:

```bash
sbatch scripts/train_cluster.sh ppo_dr_sr2l
```

3. **Monitor on Weights & Biases** (real-time)

📊 Weights & Biases Setup

What is W&B?

- Cloud-based experiment tracking
- Real-time training curves
- Hyperparameter comparison
- Video recordings of your robot

Setup Steps:

1. **Create account** at wandb.ai
2. **Install and login**:

```bash
pip install wandb
wandb login  # Enter your API key
```

3. **Enable in config**:

```yaml
# In configs/train/default.yaml
logging:
  wandb: true  # Change from false
  wandb_project: "robust-quadruped"
  wandb_entity: "your-username"
```

What W&B Tracks:

- Learning curves (reward over time)
- Success rates
- Videos of robot walking
- All hyperparameters
- System metrics (GPU usage)

🧪 Your Experimental Pipeline

Step 1: Baseline Test (PPO Only)

```bash
# Local quick test
python src/train.py --config configs/experiments/ppo_baseline.yaml \
    --override train.total_timesteps=10000

# Cluster full run
sbatch scripts/train_cluster.sh ppo_baseline
```

Step 2: Add Domain Randomization

```bash
# This config enables joint failures and sensor noise
sbatch scripts/train_cluster.sh ppo_dr
```

Step 3: Add Smooth Regularization

```bash
# This config enables SR2L for smooth actions
sbatch scripts/train_cluster.sh ppo_sr2l
```

Step 4: Full Method

```bash
# Everything combined
sbatch scripts/train_cluster.sh ppo_dr_sr2l
```

Step 5: Evaluation

```bash
# Test all trained models
python scripts/evaluate_all.py --fault_scenarios all
```

🔧 Key Files You'll Work With

1. src/train.py - Main Training Loop

```python
# Pseudocode of what happens
def main():
  # 1. Load config
  config = load_config(args.config)

  # 2. Create environment
  env = create_env(config)
  if config.domain_randomization.enabled:
    env = add_fault_wrapper(env)

  # 3. Create agent
  if config.sr2l.enabled:
    agent = SR2L_PPO(env, config)
  else:
    agent = PPO(env, config)

  # 4. Train
  for step in range(config.total_timesteps):
    agent.train()
    if step % config.eval_freq == 0:
      evaluate_and_log()
```

2. src/envs/fault_injection.py - Breaking the Robot

```python
```

```python
class FaultInjectionWrapper:
    def __init__(self, env, config):
        self.fault_config = config.faults

    def reset(self):
        # Randomly select which joints to break
        self.failed_joints = random.sample(
            range(8),
            k=random.randint(0, self.max_failures)
        )

    def step(self, action):
        # Override actions for broken joints
        action[self.failed_joints] = 0

        # Add sensor noise
        obs += np.random.normal(0, self.sensor_noise_std)
```

3. src/agents/sr2l_ppo.py - Smooth Regularization

```python
class SR2L_PPO(PPO):
    def compute_loss(self, obs, actions):
        # Standard PPO loss
        ppo_loss = super().compute_loss(obs, actions)

        # SR2L: Penalize non-smooth policies
        perturbed_obs = obs + small_noise
        smooth_loss = MSE(
            self.policy(obs),
            self.policy(perturbed_obs)
        )

        return ppo_loss + self.sr2l_lambda * smooth_loss
```

📈 Understanding Your Results

Training Curves (W&B)

- **Episode Reward**: Should increase over time
- **Success Rate**: % of episodes robot walks successfully

- **Policy Loss**: Should decrease and stabilize
- **Value Loss**: Indicates learning quality

Evaluation Metrics

json

```json
{
  "clean_performance": {
    "success_rate": 0.95,
    "avg_distance": 4.8,
    "avg_reward": 245.6
  },
  "single_joint_failure": {
    "success_rate": 0.78,
    "recovery_time": 1.2
  },
  "multiple_failures": {
    "success_rate": 0.45,
    "failure_modes": ["fall", "spin", "stuck"]
  }
}
```

🎮 Typical Development Cycle

Week 1-2: Setup & Baseline

1. Get PPO baseline working
2. Verify environment runs correctly
3. Establish baseline metrics

Week 3-4: Implement DR

1. Add fault injection wrapper
2. Implement curriculum learning
3. Test with increasing fault rates

Week 5-6: Implement SR2L

1. Modify PPO loss function
2. Tune smoothness parameter
3. Verify it doesn't hurt performance

Week 7-8: Full Experiments

1. Run all 4 ablations on cluster
2. Each takes ~24-48 hours
3. Monitor via W&B

Week 9-10: Analysis

1. Statistical comparison
2. Generate plots
3. Failure mode analysis

🚀 Quick Reference Commands

```bash
# Local testing (1 min)
python src/train.py --config configs/experiments/ppo_baseline.yaml \
                --override train.total_timesteps=1000

# View tensorboard locally
tensorboard --logdir experiments/

# Sync to cluster
rsync -av --exclude='experiments/' --exclude='venv/' \
    ./ cluster:~/projects/robust-quadruped/

# Submit cluster job
ssh cluster
cd projects/robust-quadruped
sbatch scripts/train_cluster.sh ppo_dr_sr2l

# Monitor W&B
# Just go to wandb.ai/your-username/robust-quadruped

# Download results
rsync -av cluster:~/projects/robust-quadruped/experiments/ ./experiments/

# Run evaluation
python scripts/evaluate_all.py --models experiments/*/models/best_model.pt
```

❓ Common Issues & Solutions

"Environment not found"

- Make sure MuJoCo is installed
- Check RealAnt package is installed

"CUDA out of memory"

- Reduce `num_envs` in config
- Reduce `batch_size`

"Training unstable"

- Lower learning rate
- Increase `n_epochs`
- Check reward scale

"W&B not logging"

- Check you're logged in: `wandb login`
- Verify `wandb: true` in config
- Check internet connection

📚 What to Learn

1. **Hydra Configuration**
   - Tutorial
   - Handles YAML loading and overrides
2. **Stable-Baselines3**
   - Docs
   - Your PPO implementation base
3. **Weights & Biases**
   - Quickstart
   - Focus on: logging metrics, saving videos
4. **MuJoCo Basics**
   - Understanding observation/action spaces
   - Coordinate systems

🎯 Success Criteria

Your project succeeds when:

1. **All 4 ablations train successfully**
2. **Clear performance ordering**: Full > DR/SR2L > Baseline
3. **Robust to failures**: >70% success with 1 joint failure
4. **Smooth behaviors**: No jerky movements with SR2L
5. **Reproducible**: Same config = similar results

💡 Pro Tips

1. **Start simple**: Get baseline working first
2. **Version control**: Commit before each experiment
3. **Document bugs**: Keep a log of issues/solutions
4. **Visualize early**: Watch videos of robot behavior
5. **Incremental testing**: Test each component separately
6. **Use W&B**: It's much easier than managing files manually
7. **Ask for help**: When stuck for >2 hours

This is your complete roadmap! Start with the baseline, gradually add complexity, and always test locally before cluster runs.