Lab Simulator Project - Improvement Plan & Implementation Roadmap

Executive Summary

The Lab Simulator project has excellent architectural design and solid foundational components. The main gap is connecting the well-designed configuration system and visualization tools to actual SimPy simulation execution. This document provides a prioritized roadmap for completing the implementation.

Current State Assessment

✅ Completed Components

- Core simulation, resource, and process class structures
- YAML configuration loading and workflow CSV parsing
- CLI and GUI configuration editors
- Basic metrics collection and Matplotlib visualization
- Interactive Dash Cytoscape network diagrams with critical path highlighting
- Comprehensive design documentation and configuration templates

⚠️ Gaps Requiring Immediate Attention

- **Missing SimPy Integration**: Core simulation processes not implemented
- **No Resource-Constrained Scheduling**: Critical path analysis exists but no resource allocation
- **Limited Metrics Collection**: Basic tracking but missing comprehensive performance data
- **Configuration Validation**: No validation of YAML configuration integrity
- **End-to-End Testing**: No complete workflow simulation testing

Priority Implementation Plan

Phase 1: Core SimPy Integration (Weeks 1-2)

1.1 Implement Basic SimPy Processes

python

```python
# lab_simulation/processes/sample_processor.py
def sample_preparation_process(env, sample, equipment, staff, metrics):
    """SimPy process for sample preparation"""
    with equipment.request() as eq_req, staff.request() as staff_req:
        yield eq_req & staff_req

        start_time = env.now

        # Setup phase
        yield env.timeout(equipment.setup_time)
        # Processing phase
        yield env.timeout(equipment.run_time)
        # Cleanup phase
        yield env.timeout(equipment.cleanup_time)

        metrics.record_equipment_usage(equipment.id, start_time, env.now, 'sample_prep')
```

1.2 Add Equipment State Management

python

```python
# lab_simulation/resources/equipment.py
class Equipment:
    def __init__(self, env, config):
        self.env = env
        self.config = config
        self.resource = simpy.Resource(env, capacity=1)
        self.state = 'idle'  # idle, running, maintenance, broken
        self.total_runtime = 0

    def process_sample(self, sample):
        """SimPy process for processing a sample"""
        with self.resource.request() as req:
            yield req

            self.state = 'running'
            start_time = self.env.now

            # Setup → Run → Cleanup with realistic timing
            yield self.env.timeout(self.config.setup_time.most_likely)

            run_time = random.triangular(
                self.config.sample_run_time.min,
                self.config.sample_run_time.max,
                self.config.sample_run_time.most_likely
            )
            yield self.env.timeout(run_time)

            yield self.env.timeout(self.config.cleanup_time.most_likely)

            self.total_runtime += self.env.now - start_time
            self.state = 'idle'
```

1.3 Configuration Validation System

python

```python
# lab_simulation/config/validator.py
from pydantic import BaseModel, validator
from typing import List, Dict, Optional

class TimingConfig(BaseModel):
    min: float
    most_likely: float
    max: float
    unit: str = "minutes"

    @validator('most_likely')
    def validate_timing(cls, v, values):
        if 'min' in values and 'max' in values:
            if not values['min'] <= v <= values['max']:
                raise ValueError('most_likely must be between min and max')
        return v

class EquipmentConfig(BaseModel):
    name: str
    make: str
    model: str
    setup_time: TimingConfig
    sample_run_time: TimingConfig
    cleanup_time: TimingConfig

class ConfigValidator:
    def validate_equipment_config(self, config: dict) -> List[str]:
        """Validate equipment configuration and return errors"""
        errors = []
        required_fields = ['name', 'make', 'model', 'setup_time', 'sample_run_time']
        for field in required_fields:
            if field not in config:
                errors.append(f"Missing required field: {field}")
        return errors
```

Phase 2: Enhanced Metrics & Scheduling (Weeks 3-4)

2.1 Comprehensive Metrics Collection

python

```python
# lab_simulation/tracking/metrics_collector.py
class MetricsCollector:
    def __init__(self, env):
        self.env = env
        self.equipment_usage = {}
        self.staff_utilization = {}
        self.sample_progress = []
        self.bottleneck_events = []

    def record_equipment_usage(self, equipment_id, start_time, end_time, operation_type):
        """Record equipment usage for utilization analysis"""
        if equipment_id not in self.equipment_usage:
            self.equipment_usage[equipment_id] = []
        self.equipment_usage[equipment_id].append({
            'start': start_time,
            'end': end_time,
            'duration': end_time - start_time,
            'operation': operation_type
        })

    def calculate_utilization_rates(self):
        """Calculate utilization percentages for all resources"""
        total_sim_time = self.env.now
        utilizations = {}
        for eq_id, usage_events in self.equipment_usage.items():
            total_usage = sum(event['duration'] for event in usage_events)
            utilizations[eq_id] = (total_usage / total_sim_time) * 100
        return utilizations

    def identify_bottlenecks(self):
        """Identify resource bottlenecks from utilization data"""
        utilizations = self.calculate_utilization_rates()
        bottlenecks = []
        for resource, util_pct in utilizations.items():
            if util_pct > 85:  # High utilization threshold
                bottlenecks.append({
                    'resource': resource,
                    'utilization': util_pct,
                    'severity': 'high' if util_pct > 95 else 'medium'
                })
        return bottlenecks
```

2.2 Resource-Constrained Scheduling

python

```python
# lab_simulation/scheduling/resource_scheduler.py
class ResourceConstrainedScheduler:
    def __init__(self, workflow, resources):
        self.workflow = workflow
        self.resources = resources

    def generate_schedule(self, sample_count):
        """Generate realistic schedule considering resource availability"""
        # Critical path analysis
        critical_path = self.calculate_critical_path()

        # Resource allocation with constraints
        schedule = self.allocate_resources(critical_path, sample_count)

        return schedule

    def calculate_critical_path(self):
        """CPM algorithm implementation"""
        # Forward pass - calculate earliest start/finish
        # Backward pass - calculate latest start/finish
        # Identify critical path (float = 0)
        pass

    def allocate_resources(self, critical_path, sample_count):
        """Allocate limited resources across workflow steps"""
        # Priority queue of ready tasks
        # Resource availability checking
        # Schedule generation with realistic start times
        pass
```

2.3 Real-Time Burn Chart Tracking

python

```python
# lab_simulation/tracking/burn_chart.py
class BurnChartTracker:
    def __init__(self, env, total_samples):
        self.env = env
        self.total_samples = total_samples
        self.planned_curve = []
        self.actual_completions = {}

    def record_sample_completion(self, day: int, sample_id: str, step_completed: str):
        """Record actual progress during simulation"""
        if day not in self.actual_completions:
            self.actual_completions[day] = []
        self.actual_completions[day].append((sample_id, step_completed))

    def get_current_data(self):
        """Generate current burn chart data for visualization"""
        current_day = int(self.env.now // (8 * 60))  # Convert sim time to days

        return {
            'planned_curve': self.planned_curve,
            'actual_curve': self.get_actual_curve(current_day),
            'current_day': current_day,
            'variance': self.calculate_variance(current_day)
        }

    def calculate_variance(self, current_day):
        """Calculate planned vs actual variance"""
        # Compare planned vs actual completion rates
        # Generate early warning indicators
        pass
```

Phase 3: Integration & Optimization (Weeks 5-6)

3.1 Main Simulation Runner

```
python
```

```python
# lab_simulation/simulation_runner.py
class SimulationRunner:
    def __init__(self, config_path: str):
        self.config = self.load_configuration(config_path)
        self.metrics_collector = None
        self.burn_chart_tracker = None

    def run_scenario(self, scenario_name: str, sample_count: int):
        """Run a complete simulation scenario"""
        # Setup simulation environment
        env = simpy.Environment()
        self.metrics_collector = MetricsCollector(env)
        self.burn_chart_tracker = BurnChartTracker(env, sample_count)

        # Create lab resources from config
        lab = self.create_lab_from_config(env, self.config)

        # Create project and workflow
        project = self.create_project(env, sample_count)

        # Run simulation
        env.run()

        # Generate comprehensive results
        results = SimulationResults(
            duration=env.now,
            utilization=self.metrics_collector.calculate_utilization_rates(),
            burn_chart=self.burn_chart_tracker.get_final_data(),
            bottlenecks=self.metrics_collector.identify_bottlenecks()
        )

        return results

    def compare_scenarios(self, scenarios: List[Dict]):
        """Run multiple scenarios and compare results"""
        results = {}
        for scenario in scenarios:
            results[scenario['name']] = self.run_scenario(
                scenario['name'],
                scenario['sample_count']
            )
        return results
```

3.2 Live Dashboard Integration

python

```python
# Update workflow_dash_app.py
@app.callback(
    Output('burn-chart', 'figure'),
    Input('interval-component', 'n_intervals')
)
def update_burn_chart(n):
    """Update burn chart with latest simulation data"""
    if simulation_running:
        current_data = burn_chart_tracker.get_current_data()
        return create_burn_chart_figure(current_data)
    return {}

def create_burn_chart_figure(data):
    """Create Plotly figure for burn chart"""
    fig = go.Figure()

    # Planned curve
    fig.add_trace(go.Scatter(
        x=[d[0] for d in data['planned_curve']],
        y=[d[1] for d in data['planned_curve']],
        name='Planned',
        line=dict(color='blue', dash='dash')
    ))

    # Actual curve
    fig.add_trace(go.Scatter(
        x=[d[0] for d in data['actual_curve']],
        y=[d[1] for d in data['actual_curve']],
        name='Actual',
        line=dict(color='green')
    ))

    return fig
```

Critical Missing Components to Add

1. New Directory Structure

```
lab_simulation/
├── config/
│   ├── __init__.py
│   ├── loader.py           # YAML/CSV loading (enhance existing)
│   ├── validator.py        # NEW: Configuration validation
│   └── schema.py          # NEW: Pydantic models for type safety
├── scheduling/             # NEW: Missing from current structure
│   ├── __init__.py
│   ├── critical_path.py   # CPM/PERT algorithms
│   ├── resource_scheduler.py
│   └── optimization.py    # Resource optimization algorithms
├── tracking/              # NEW: Enhanced version
│   ├── __init__.py
│   ├── metrics_collector.py  # NEW: Comprehensive metrics
│   ├── burn_chart.py         # NEW: Real-time progress tracking
│   └── performance_monitor.py # NEW: Performance analysis
└── simulation_runner.py    # NEW: Main orchestration class
```

2. Enhanced Visualization

Improve your existing `workflow_dash_app.py`:

python

```python
# Better layout algorithm
layout={
    'name': 'dagre',  # Better for directed graphs than breadthfirst
    'directed': True,
    'padding': 30,
    'spacingFactor': 1.2,
    'rankDir': 'TB'  # Top to bottom layout
}

# Add tooltips and interactivity
stylesheet=[
    {
        'selector': 'node',
        'style': {
            'label': 'data(label)',
            'text-valign': 'center',
            'text-halign': 'center',
            'shape': 'rectangle',
            'width': 150,  # Increased width
            'height': 80,  # Increased height
            'font-size': 12,
            'text-wrap': 'wrap',
            'text-max-width': 140
        }
    }
]
```

3. Comprehensive Testing Framework

python

```python
# tests/test_simulation_integration.py
def test_simple_workflow():
    """Test basic workflow with minimal resources"""
    config = load_test_config('simple_lab.yaml')
    runner = SimulationRunner(config)

    results = runner.run_scenario('test', sample_count=5)

    assert results.duration > 0
    assert len(results.utilization) > 0
    assert 'rock_saw' in results.utilization

def test_resource_constraints():
    """Test that resource constraints are respected"""
    # Ensure only one sample can use equipment at a time
    # Verify staff skill requirements are enforced
    pass

def test_critical_path_calculation():
    """Test that critical path is correctly identified"""
    # Verify critical path matches manual calculation
    pass

def test_burn_chart_accuracy():
    """Test burn chart tracking accuracy"""
    # Run simulation and verify progress tracking
    pass

# tests/test_configuration_validation.py
def test_config_validation():
    """Test configuration validation catches errors"""
    invalid_config = {'equipment': {'missing_fields': {}}}
    validator = ConfigValidator()
    errors = validator.validate_equipment_config(invalid_config)
    assert len(errors) > 0

def test_timing_validation():
    """Test timing parameter validation"""
    invalid_timing = {'min': 10, 'most_likely': 5, 'max': 15}  # Invalid order
    with pytest.raises(ValueError):
        TimingConfig(**invalid_timing)
```

Implementation Priorities

Week 1-2: Foundation

1. ✅ Implement basic SimPy equipment processes
2. ✅ Add configuration validation with Pydantic
3. ✅ Create simple end-to-end test (5 samples)
4. ✅ Fix equipment state management

Week 3-4: Core Features

1. ✅ Enhanced metrics collection system
2. ✅ Resource-constrained scheduling implementation
3. ✅ Real-time burn chart tracking
4. ✅ Integration testing with larger sample sets

Week 5-6: Polish & Optimization

1. ✅ Scenario comparison capabilities
2. ✅ Interactive dashboard enhancements
3. ✅ Performance optimization for 1000+ samples
4. ✅ Documentation and examples

Success Metrics

Phase 1 Success Criteria:

- ☐ 5-sample workflow completes successfully
- ☐ Equipment utilization metrics generated
- ☐ Configuration validation prevents invalid configs
- ☐ Basic unit tests pass

Phase 2 Success Criteria:

- ☐ 100-sample scenario runs in reasonable time (<5 minutes)
- ☐ Burn chart shows planned vs actual progress
- ☐ Bottlenecks automatically identified
- ☐ Resource utilization >80% detected and flagged

Phase 3 Success Criteria:

- □ Multiple scenarios can be compared
- □ Live dashboard updates during simulation
- □ 1000+ sample simulations complete successfully
- □ Optimization recommendations generated

Quick Wins (Implement First)

1. **Configuration Validation** - Prevent runtime errors from bad configs
2. **Basic SimPy Equipment Process** - Get one piece of equipment working end-to-end
3. **Simple Metrics Collection** - Track equipment usage and utilization
4. **End-to-End Test** - Prove the simulation actually works

Technical Debt to Address

1. **Inconsistent Naming**: Standardize naming conventions across modules
2. **Missing Error Handling**: Add try/catch blocks and meaningful error messages
3. **No Logging**: Add comprehensive logging for debugging
4. **Hard-coded Values**: Move remaining hard-coded values to configuration
5. **Memory Usage**: Optimize for large simulations (1000+ samples)

Conclusion

Your project has excellent architectural design and solid visualization components. The main focus should be connecting these components with working SimPy simulation processes. By following this phased approach, you'll have a fully functional laboratory simulation system within 6 weeks.

Immediate Next Step: Start with Phase 1.1 - implement one basic SimPy equipment process and get a single sample flowing through the rock cutting workflow. This will prove the core concept and provide a foundation for building the remaining featur