Quick Start: Dynamic Self-Configuration

Get started with autonomous runtime parameter adjustment in 5 minutes!

🚀 What is Dynamic Self-Configuration?

The system can **automatically adjust its own runtime parameters** based on performance metrics. For example:

- High latency detected → Reduces planner search depth for faster responses
- High error rate → Increases retry attempts
- High memory usage → Reduces batch size
- Excellent performance → Increases search depth for better quality

No manual intervention required!

📋 Prerequisites

Ensure you have:
- ✅ Powerhouse Multi-Agent Platform installed
- ✅ Python 3.8+
- ✅ All dependencies installed (`pip install -r requirements.txt`)

⚡ Quick Start (3 Steps)

Step 1: Import and Initialize

```python
from core.adaptive_orchestrator import AdaptiveOrchestrator

# Create orchestrator with dynamic config enabled
orchestrator = AdaptiveOrchestrator(
    agent_names=["planning", "react", "chain_of_thought"],
    enable_adaptation=True  # This enables self-configuration
)
```

Step 2: Run Tasks

```python
# Run tasks normally - system adapts automatically!
result = orchestrator.run("Analyze market trends and generate report")

# The system automatically:
# - Monitors performance (latency, success rate, errors)
# - Evaluates adjustment rules every 60 seconds
# - Adjusts parameters if needed
# - Rolls back if performance degrades
```

Step 3: Check Status

```python
# Get performance and configuration summary
summary = orchestrator.get_performance_summary()

print(f"Health Score: {summary['health_score']}/100")
print(f"Success Rate: {summary['performance']['success_rate']*100:.1f}%")
print(f"Current Parameters: {summary['configuration']['current_parameters']}")
print(f"Total Changes: {summary['configuration']['statistics']['total_changes']}")
```

That's it! Your system is now self-configuring.

🎯 Common Use Cases

Use Case 1: Handle Sudden Traffic Spike

```
# System detects high latency from increased load
# Automatically reduces search depth from 5 → 3
# Response time improves from 8s → 3s
# Later, when traffic normalizes, depth increases back to 5
```

Use Case 2: Adapt to Network Issues

```
# System detects high error rate (network timeouts)
# Automatically increases retries from 3 → 4
# Increases timeout from 60s → 72s
# Success rate improves from 75% → 90%
```

Use Case 3: Optimize During Idle Time

```
# System detects excellent performance (95%+ success)
# Automatically increases search depth from 5 → 6
# Quality improves with negligible latency increase
```

🎨 Adjustment Strategies

Choose your strategy based on your needs:

```python
from core.dynamic_config_manager import AdjustmentStrategy

# For production (safe, slow)
orchestrator = AdaptiveOrchestrator(
    agent_names=["planning"],
    enable_adaptation=True
)
# Uses BALANCED strategy by default

# For aggressive optimization (development)
# Change strategy in config_manager initialization
```

Strategies:
- **Conservative**: 50% of adjustments (very safe)
- **Balanced**: 100% of adjustments (recommended)
- **Aggressive**: 150% of adjustments (fast optimization)
- **Gradual**: 25% of adjustments (very slow)

🔧 Manual Control

Get Current Parameters

```
from core.dynamic_config_manager import get_config_manager

config_manager = get_config_manager()

# Get specific parameter
depth = config_manager.get_parameter("planner_search_depth")
print(f"Current search depth: {depth}")

# Get all parameters
snapshot = config_manager.get_configuration_snapshot()
print(snapshot['parameters'])
```

Override Parameters

```
# Set parameter manually
orchestrator.set_parameter(
    "planner_search_depth",
    8,
    reason="Optimizing for quality"
)

# System will use this value but can still adjust later
# To prevent auto-adjustment, disable the rule
```

Disable Specific Rules

```
# Disable a specific adjustment rule
config_manager.adjustment_rules["reduce_depth_on_high_latency"].enabled = False

# Re-enable later
config_manager.adjustment_rules["reduce_depth_on_high_latency"].enabled = True
```

Reset to Defaults

```
# Reset all parameters to defaults
orchestrator.reset_configuration()
```

📊 Monitoring

View Configuration Changes

```python
# Get recent changes
snapshot = config_manager.get_configuration_snapshot()

for change in snapshot['recent_changes']:
    print(f"{change['timestamp']}: {change['parameter']}")
    print(f"  {change['old_value']} → {change['new_value']}")
    print(f"  Reason: {change['reason']}")
```

Check Statistics

```python
# Get statistics
stats = config_manager.get_statistics()

print(f"Total changes: {stats['total_changes']}")
print(f"Rollbacks: {stats['rollbacks']}")
print(f"Changes per hour: {stats['avg_changes_per_hour']:.2f}")

# Changes by parameter
for param, count in stats['changes_by_parameter'].items():
    print(f"  {param}: {count} changes")
```

View Active Rules

```python
# Get all rules
snapshot = config_manager.get_configuration_snapshot()

for name, info in snapshot['active_rules'].items():
    status = "✓" if info['enabled'] else "✗"
    print(f"{status} {name}")
    if info['trigger_count'] > 0:
        print(f"  Triggered {info['trigger_count']} times")
```

🌐 REST API Usage

Get Configuration

```bash
curl -X GET http://localhost:8000/api/v1/config/snapshot \
  -H "Authorization: Bearer YOUR_TOKEN"
```

Update Parameter

```
curl -X POST http://localhost:8000/api/v1/config/parameters/planner_search_depth \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "parameter_name": "planner_search_depth",
    "value": 7,
    "reason": "Manual optimization"
  }'
```

Get System Health

```
curl -X GET http://localhost:8000/api/v1/config/health \
  -H "Authorization: Bearer YOUR_TOKEN"
```

Force Evaluation

```
curl -X POST http://localhost:8000/api/v1/config/force-evaluation \
  -H "Authorization: Bearer YOUR_TOKEN"
```

🧪 Testing

Run Unit Tests

```
cd /home/ubuntu/powerhouse_b2b_platform/backend
python -m pytest tests/test_dynamic_config_manager.py -v
```

Run Integration Tests

```
python -m pytest tests/test_integration_dynamic_config.py -v
```

Run Examples

```
# Run basic example
python examples/dynamic_config_example.py

# Run specific example
python -c "
from examples.dynamic_config_example import example_2_high_latency_adaptation
example_2_high_latency_adaptation()
"
```

🎓 Understanding Adjustment Rules

Default Rules

Rule	Trigger	Action	Priority
Reduce depth on high latency	Latency > 5s	Decrease search depth	8
Increase retries on errors	Error rate > 15%	Increase max retries	7
Increase timeout on timeouts	Latency > 50s	Multiply timeout by 1.2	6
Reduce batch on memory	Memory > 500MB	Decrease batch size	7
Increase depth on good perf	Success > 95%	Increase search depth	4
Lower threshold on low success	Success < 70%	Decrease quality threshold	5

Rule Components

Each rule has:
- **Trigger Metric**: What to monitor (latency, error_rate, etc.)
- **Threshold**: When to trigger (e.g., > 5000ms)
- **Target Parameter**: What to adjust
- **Adjustment**: How much to change
- **Cooldown**: Minimum time between triggers
- **Rate Limit**: Maximum triggers per hour

🔒 Safety Features

1. Parameter Bounds

All parameters have min/max limits:

```
planner_search_depth: 1-10 (default: 5)
max_retries: 0-5 (default: 3)
timeout_seconds: 5-300 (default: 60)
```

2. Automatic Rollback

If performance degrades after a change:
- System waits 5 minutes (configurable)
- Compares performance to baseline
- If worse, automatically reverts

3. Cooldown Periods

Rules can’t trigger too frequently:
- Prevents oscillation
- Default: 60-300 seconds between triggers

4. Rate Limiting

Maximum adjustments per hour:
- Prevents runaway adjustments
- Default: 2-10 per hour depending on rule

🐛 Troubleshooting

Parameters Not Changing?

Check if rules are enabled:

```
for name, rule in config_manager.adjustment_rules.items():
    print(f"{name}: {'enabled' if rule.enabled else 'disabled'}")
```

Check if cooldown expired:

```
rule = config_manager.adjustment_rules["reduce_depth_on_high_latency"]
if rule.last_triggered:
    elapsed = (datetime.now() - rule.last_triggered).total_seconds()
    print(f"Last triggered {elapsed}s ago (cooldown: {rule.cooldown_seconds}s)")
```

Too Many Adjustments?

Increase cooldown:

```
rule.cooldown_seconds = 300  # 5 minutes
```

Reduce rate limit:

```
rule.max_adjustments_per_hour = 2
```

Rollback Not Working?

Ensure enabled:

```
config_manager.enable_auto_rollback = True
```

Check degradation threshold:

```
# Rollback triggers if degradation score ≥ 3
# Factors: success rate drop, latency increase, error increase, etc.
```

📚 Next Steps

1. **Read Full Documentation**: `DYNAMIC_SELF_CONFIGURATION_README.md`
2. **Review Examples**: `examples/dynamic_config_example.py`
3. **Check Implementation**: `DYNAMIC_CONFIG_IMPLEMENTATION_SUMMARY.md`
4. **Explore API**: Try the REST endpoints
5. **Run Tests**: Verify everything works

💡 Pro Tips

Tip 1: Start Conservative

Begin with default (Balanced) strategy, monitor for a few days, then consider Aggressive if needed.

Tip 2: Monitor Changes

Regularly check configuration changes:

```python
snapshot = config_manager.get_configuration_snapshot()
print(f"Recent changes: {len(snapshot['recent_changes'])}")
```

Tip 3: Use Health Score

Health score (0-100) gives quick system status:
- 80-100: Excellent
- 60-79: Good
- 40-59: Fair
- 0-39: Poor

Tip 4: Custom Rules

Add your own rules for specific scenarios:

```python
from core.dynamic_config_manager import AdjustmentRule, MetricType

custom_rule = AdjustmentRule(
    name="my_custom_rule",
    description="My custom adaptation",
    trigger_metric=MetricType.COST,
    trigger_threshold=1.0,
    trigger_operator="gt",
    target_parameter="batch_size",
    adjustment_value=-5,
    adjustment_type="relative",
    scope=ConfigurationScope.GLOBAL,
    priority=6
)

config_manager.add_adjustment_rule(custom_rule)
```

Tip 5: Disable in Critical Situations

If you need absolute predictability:

```
orchestrator = AdaptiveOrchestrator(
    agent_names=["planning"],
    enable_adaptation=False  # Disable for critical operations
)
```

✅ Verification Checklist

After setup, verify:

- [ ] Orchestrator created with `enable_adaptation=True`
- [ ] Initial parameters loaded correctly
- [ ] Task execution works normally
- [ ] Performance metrics being recorded
- [ ] Configuration snapshot accessible
- [ ] Rules are enabled
- [ ] API endpoints responding (if using API)

🎉 You're Ready!

Your agent system can now:
- ✅ Monitor its own performance
- ✅ Detect degradation or opportunities
- ✅ Adjust parameters automatically
- ✅ Roll back if needed
- ✅ Provide complete observability

Happy autonomous optimization! 🚀

For detailed information, see:
- **Full Documentation**: `DYNAMIC_SELF_CONFIGURATION_README.md`
- **Implementation Summary**: `DYNAMIC_CONFIG_IMPLEMENTATION_SUMMARY.md`
- **Examples**: `examples/dynamic_config_example.py`