TaskFactory Integration Guide

Overview

We've built a sophisticated reasoning effort assessment system for multi-agent task management. This document provides implementation details and best practices for integrating the enhanced TaskFactory into your existing architecture.

Key Components

1. Enhanced TaskFactory

- **Dynamic Reasoning Effort Assessment**: Intelligent classification of tasks as LOW, MEDIUM, or HIGH effort
- **Weighted Category Scoring**: Analytical, comparative, creative, and complex keyword categories
- **Contextual Adjustments**: Event type, intent, confidence level, and deadline pressure
- **Auto-tuning System**: Learning loop that adjusts weights based on actual task performance
- **Detailed Diagnostics**: Comprehensive metrics for dashboard visualization

2. WebSocket Server Integration

- Task creation with automated reasoning effort assessment
- Real-time status updates for agents and tasks
- Client connection management with proper error handling
- Redis pub/sub integration for agent communication

3. Dashboard Components

- Task metrics and statistics visualization
- Complexity analysis with scatter plots
- Effort distribution breakdown
- Category impact and adjustment factors tracking

Integration Steps

1. Add the Enhanced TaskFactory

bash Copy

```bash
# Copy the TaskFactory implementation to your project
cp enhanced-task-factory.py /path/to/your/project/backend/factories/task_factory.py
```

Ensure the necessary imports are updated in your models:

python Copy

```python
# In your models.py or equivalent
from enum import Enum

class ReasoningEffort(str, Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"

class ReasoningStrategy(str, Enum):
    DIRECT = "direct_answer"
    COT = "chain-of-thought"
    COD = "chain-of-draft"

def get_reasoning_strategy(effort: ReasoningEffort) -> str:
    """
    Maps reasoning effort to a cognitive strategy the agent should use.
    """
    if effort == ReasoningEffort.LOW:
        return "direct_answer"
    elif effort == ReasoningEffort.MEDIUM:
        return "chain-of-thought"
    elif effort == ReasoningEffort.HIGH:
        return "chain-of-draft"
    return "unknown"
```

2. Update Your WebSocket Server

Replace your existing WebSocket server implementation with our enhanced version that integrates the TaskFactory:

bash Copy

```bash
cp websocket-server.py /path/to/your/project/backend/server/websocket_server.py
```

3. Add the Dashboard Components

bash Copy

```bash
# Copy the React dashboard component to your frontend
cp task-dashboard.jsx /path/to/your/frontend/components/TaskDashboard.jsx

# Add it to your page or app
```

In your main page component:

jsx Copy

```jsx
import TaskDashboard from '@/components/TaskDashboard';

// In your component:
return (
  <div className="flex flex-col">
    <Header />
    <main className="flex-1">
      <TaskDashboard />
    </main>
  </div>
);
```

4. Configure Environment Variables

Add these to your `.env` file:

Copy

```
# TaskFactory Configuration
TASK_FACTORY_AUTOTUNE=true
TASK_FACTORY_RETAIN_HISTORY=true
TASK_FACTORY_HISTORY_LIMIT=1000
TASK_FACTORY_MIN_SAMPLES=10

# Agent Configuration
REQUIRED_AGENTS=Grok,Claude,GPT
DEFAULT_AGENT=Grok

# Redis Configuration
REDIS_HOST=localhost
REDIS_PORT=6379
```

5. Update Agent Code to Handle Reasoning Strategy

Modify your agent implementations to handle the reasoning strategy specified by the TaskFactory:

python Copy

```python
# In your agent class
async def handle_task(self, task: Task):
    # Use the assigned reasoning strategy to guide processing
    strategy = task.reasoning_strategy.value
    logger.info(f"Processing task with strategy: {strategy}")

    if strategy == "direct_answer":
        # Use straightforward, minimal processing
        # ...
    elif strategy == "chain-of-thought":
        # Use step-by-step reasoning
        # ...
    elif strategy == "chain-of-draft":
        # Use iterative drafting and refinement
        # ...

    # Continue with task processing
    # ...
```

Best Practices

Performance Optimization

1. **Redis Connection Pooling**: Ensure proper connection pooling for Redis to prevent connection exhaustion.
2. **Task Batching**: For high-volume systems, consider implementing task batching to reduce processing overhead.
3. **Selective Streaming**: Only use streaming responses for HIGH effort tasks to reduce bandwidth usage.

Maintaining the Learning Loop

1. **Record Task Outcomes**: Always record the actual outcome, duration, and success of tasks:

python Copy

```python
# After task completion
TaskFactory.record_task_outcome(
    task_id=task.task_id,
    diagnostics=task_diagnostics,
    actual_duration=duration_seconds,
    success=was_successful,
    feedback=optional_feedback
)
```

2. **Regular Analysis**: Set up a periodic job to analyze outcomes if you have low task volume:

python Copy

```python
# Run this periodically if you don't reach the automatic threshold
if len(TaskFactory.outcome_history) > 0:
    TaskFactory.analyze_outcomes()
```

3. **Monitor Tuning Recommendations**: Check `task_factory_analysis_results.json` regularly for insights.

Error Handling

1. **Graceful Degradation**: If TaskFactory encounters an error, default to MEDIUM effort:

python Copy

```
try:
    reasoning_effort, diagnostics = TaskFactory.estimate_reasoning_effort(content, event, inter
except Exception as e:
    logger.error(f"Error estimating reasoning effort: {e}")
    reasoning_effort = ReasoningEffort.MEDIUM
    diagnostics = {"error": str(e)}
```

2. **Validation**: Always validate task content before passing to TaskFactory:

python Copy

```
if not content or not isinstance(content, str):
    content = "Empty or invalid task content"
    # Handle appropriately
```

Monitoring & Alerts

Set up monitoring for these key metrics:

1. **Effort Distribution Drift**: Alert if the distribution changes significantly (e.g., sudden increase in HIGH effort tasks)
2. **Task Duration Anomalies**: Monitor for tasks taking much longer than expected for their effort level
3. **Success Rate by Effort**: Track success rates for each effort level to identify potential misclassifications
4. **Agent Response Times**: Ensure agents respond appropriately to different reasoning strategies

Future Enhancements

1. **ML-based Effort Prediction**: Replace keyword-based classification with a trained ML model
2. **Agent-specific Calibration**: Calibrate effort levels per agent based on their specific capabilities
3. **User Feedback Integration**: Incorporate direct user feedback on task difficulty
4. **A/B Testing**: Implement A/B testing of different category weights and thresholds

Support & Troubleshooting

For issues with the TaskFactory integration:

1. Check the logs for detailed diagnostics
2. Ensure Redis is properly configured and accessible

3. Verify that agents are handling the reasoning strategies correctly
4. Check the task_factory_analysis_results.json file for insights

Conclusion

The enhanced TaskFactory provides a significant upgrade to multi-agent task coordination by dynamically assessing and adapting to task complexity. By integrating this system, your agents will be able to allocate appropriate cognitive resources to tasks based on their complexity and learn from their performance over time.

Let's kick some technological ass! 🚀