Integration Testing Framework

This document describes the integration testing framework for the agent-orchestration-ops repository.

Overview

The integration testing framework provides comprehensive validation of the routing infrastructure, including:

- **Router health and basic functionality** (smoke tests)
- **Provider fallback behavior** (resilience tests)
- **Cost tracking and attribution** (ledger tests)

All tests run against a local Docker Compose environment that mirrors production infrastructure.

Architecture

Test Environment

The test environment is defined in `tests/compose.int.yml` and includes:

- **Redis** (7-alpine): Caching and state management
- **LiteLLM**: Router service with provider fallback logic
- **vLLM**: Local LLM inference engine (GPU-enabled)

All services include health checks and proper dependency ordering.

Test Suites

1. Smoke Tests (`test_router_smoke.py`)

Basic functionality validation:
- Health endpoint availability
- Simple chat completion requests
- Response format validation
- Model listing
- Metadata handling

Key function: `wait_health()` - Waits for router to become healthy before running tests.

2. Fallback Tests (`test_fallbacks.py`)

Provider resilience validation:
- Timeout handling and fallback routing
- Rate limit handling
- Fallback chain exhaustion
- Metadata preservation through fallbacks
- Latency impact of fallback routing

3. Cost Ledger Tests (`test_cost_ledgers.py`)

Cost tracking validation:
- Usage metrics structure (prompt_tokens, completion_tokens, total_tokens)
- Multi-tenant cost attribution

- Cost accumulation across requests
- Metadata preservation for attribution
- Streaming response cost tracking
- Edge cases (minimal tokens, consistency)

Running Tests Locally

Prerequisites

- Docker and Docker Compose installed
- Python 3.11+ with pip
- (Optional) NVIDIA GPU with Docker runtime for vLLM

Quick Start

1. **Install test dependencies**:
   ```
   bash
      pip install pytest requests
   ```
2. **Start test environment**:
   ```
   bash
      cd tests
      docker compose -f compose.int.yml up -d
   ```
3. **Wait for services to be ready** (30-60 seconds):
   ```
   bash
      docker compose -f compose.int.yml ps
      docker compose -f compose.int.yml logs -f litellm
   ```
4. **Run all tests**:
   ```
   bash
      pytest tests/ -v
   ```
5. **Run specific test suite**:
   ```
   bash
      pytest tests/test_router_smoke.py -v
      pytest tests/test_fallbacks.py -v
      pytest tests/test_cost_ledgers.py -v
   ```
6. **Cleanup**:
   ```
   bash
      docker compose -f tests/compose.int.yml down -v
   ```

Environment Variables

- `ROUTER_URL` : Router base URL (default: `http://localhost:4000`)
- `HEALTH_TIMEOUT` : Seconds to wait for health (default: `60`)

Example:

```
ROUTER_URL=http://localhost:4000 HEALTH_TIMEOUT=120 pytest tests/ -v
```

CI/CD Integration

GitHub Actions Workflow

The integration tests run automatically on:
- Push to `main`, `develop`, or `feature/**` branches
- Pull requests to `main` or `develop`

Workflow file: `.github/workflows/integration.yml`

Workflow Steps

1. **Checkout code**
2. **Set up Docker Buildx**
3. **Set up Python 3.11**
4. **Install pytest and requests**
5. **Start test environment** (docker compose up)
6. **Wait for services** (30s + health checks)
7. **Run smoke tests**
8. **Run fallback tests**
9. **Run cost ledger tests**
10. **Capture logs on failure** (redis, litellm, vllm)
11. **Upload logs as artifacts** (7-day retention)
12. **Cleanup** (docker compose down)

Viewing Test Results

- **GitHub Actions UI**: Check the “Actions” tab in the repository
- **Pull Request checks**: Tests must pass before merge
- **Logs on failure**: Download artifacts from failed runs

Test Development Guidelines

Adding New Tests

1. **Choose appropriate test file**:
   - `test_router_smoke.py`: Basic functionality
   - `test_fallbacks.py`: Resilience and error handling
   - `test_cost_ledgers.py`: Cost tracking and attribution

2. **Use the `wait_health()` fixture**:
   `python`
   ```
     @pytest.fixture(scope="module", autouse=True)
     def ensure_router_ready():
         wait_health()
   ```

3. **Include descriptive docstrings**:
   ```python
   def test_new_feature():
   “””
   Test description here.
   ```

Verifies that:
1. First behavior
2. Second behavior
"""
```

4. **Use proper assertions**:
   `python`
   `assert response.status_code == 200, f"Request failed: {response.text}"`

5. **Clean up resources**:
   - Tests should be idempotent
   - No persistent state between tests
   - Use unique trace_ids for tracking

### Best Practices

- **Timeouts**: Always set reasonable timeouts on requests
- **Retries**: Use `wait_health()` pattern for flaky operations
- **Metadata**: Include tenant, trace_id, user_id for tracking
- **Assertions**: Be specific about what you're testing
- **Error messages**: Include context in assertion messages
- **Isolation**: Tests should not depend on each other

## Troubleshooting

### Common Issues

#### Services not starting

**Symptom**: Tests fail with connection errors

**Solution**:

```
# Check service status
docker compose -f tests/compose.int.yml ps

# View logs
docker compose -f tests/compose.int.yml logs litellm
docker compose -f tests/compose.int.yml logs redis

# Restart services
docker compose -f tests/compose.int.yml restart
```

Health check timeouts

Symptom: `TimeoutError: Router not healthy after 60s`

Solution:
- Increase `HEALTH_TIMEOUT` environment variable
- Check if services are actually running
- Verify network connectivity between containers

GPU not available for vLLM

Symptom: vLLM fails to start or falls back to CPU
```

Solution:
- Ensure NVIDIA Docker runtime is installed
- Check GPU availability: `nvidia-smi`
- Modify `compose.int.yml` to remove GPU requirement for testing

Port conflicts

Symptom: `Error: port already in use`

Solution:

```
# Find process using port
lsof -i :4000
lsof -i :6379
lsof -i :8000

# Kill process or change ports in compose.int.yml
```

Debug Mode

Run tests with verbose output:

```
pytest tests/ -v -s --tb=long
```

View real-time logs:

```
docker compose -f tests/compose.int.yml logs -f
```

Maintenance

Updating Test Environment

When infrastructure changes:

1. Update `tests/compose.int.yml` to match production config
2. Update test assertions if API contracts change
3. Add new tests for new features
4. Update this documentation

Performance Considerations

- Tests run in ~5-10 minutes in CI
- Local runs may be faster (cached images)
- GPU tests require appropriate hardware
- Consider test parallelization for large suites

Future Enhancements

Planned improvements:

- [ ] Load testing with k6 or locust
- [ ] Performance regression detection
- [ ] Multi-region failover tests

- [ ] Chaos engineering scenarios
- [ ] Integration with monitoring/alerting
- [ ] Test data generation and fixtures
- [ ] Parallel test execution
- [ ] Test coverage reporting

Support

For issues or questions:

1. Check this documentation
2. Review test logs and error messages
3. Check GitHub Issues for similar problems
4. Create a new issue with:
   - Test output
   - Service logs
   - Environment details
   - Steps to reproduce

Last Updated: 2025-09-30
Maintainer: Empire325Marketing DevOps Team