Builder Project - AI Model Testing Framework

Comprehensive AI Model Testing and Automated Issue Resolution System

Overview

The Builder project includes a sophisticated testing framework that automatically tests all installed AI models, detects issues, and applies intelligent fixes to ensure system reliability. The framework implements a **test-fix-retest loop** that continues until all issues are resolved or maximum iterations are reached.

Quick Start

Basic Testing

```
# Test all models (fail-fast mode)
./test.sh

# Test with automatic fixing enabled
./test.sh --auto-fix

# Test with custom date
./test.sh --date=2025-09-15

# Show help
./test.sh --help
```

Installation Testing

```
# Install and test General models
./Scripts/install.sh General
./test.sh --auto-fix

# Install and test Coder models
./Scripts/install.sh Coder
./test.sh --auto-fix

# Install and test Audio models
./Scripts/install.sh Generative/Audio
./test.sh --auto-fix
```

Test Framework Architecture

Core Components

1. **Model Testing Engine**

- Tests individual AI models with configurable timeouts
- Validates responses against expected patterns
- Generates both raw and cleaned output files
- Creates comprehensive test reports

2. **Issue Detection System**
   - Categorizes failures: `MODEL_NOT_AVAILABLE`, `TIMEOUT`, `UNEXPECTED_RESPONSE`, `NO_OUTPUT`
   - Creates detailed issue documentation with fix recommendations
   - Provides diagnostic information for each failure

3. **Automated Fixing Engine**
   - Scans project files for common issues
   - Applies targeted fixes to detected problems
   - Validates script permissions and dependencies
   - Checks system services and resources

4. **Test-Fix-Retest Loop**
   - Orchestrates iterative testing with automatic recovery
   - Tracks fix success rates and iteration counts
   - Provides comprehensive final reporting

Command Line Interface

Usage

```
./test.sh [OPTIONS]
```

Options

- `--auto-fix`: Enable automatic fixing of detected issues (default: false)
- `--date=YYYY-MM-DD`: Use specific date for test results (default: today)
- `--help`: Show help message

Environment Variables

```
# Enable auto-fix via environment variable
AUTO_FIX=true ./test.sh
```

Exit Codes

- `0`: All tests passed successfully
- `1`: Some tests failed (check reports for details)

Test Results Organization

Directory Structure

```
Tests/{YYYY-MM-DD}/
├── COMPREHENSIVE_TEST_REPORT.md    # Overall test results
├── model_size.txt                  # Selected model category
(7B/13B/34B/70B)
├── system_info.json               # System specifications
└── {MODEL_NAME}/
    ├── Generated/                  # Model outputs and generated content
    │   ├── test_response.txt      # Cleaned model response
    │   └── test_response_raw.txt  # Original response with control chars
    ├── Issues/                     # Detailed issue documentation
    │   ├── MODEL_PULL_FAILED.md  # Installation issues
    │   ├── TIMEOUT_OR_ERROR.md   # Runtime issues
    │   └── UNEXPECTED_RESPONSE.md # Response validation issues
    ├── Report.md                   # Individual model test report
    └── test_status.txt             # PASSED/FAILED status
```

Sample Results

```
Tests/2025-09-15/
├── COMPREHENSIVE_TEST_REPORT.md
├── qwen3:8b/
│   ├── Generated/
│   │   ├── test_response.txt (1443 characters)
│   │   └── test_response_raw.txt
│   ├── Report.md (✅ PASSED)
│   └── test_status.txt
├── deepseek-r1:7b/ (✅ PASSED)
├── llama3:8b/ (✅ PASSED)
├── mistral:7b/ (✅ PASSED)
├── openthinker:7b/ (❌ FAILED)
├── qwen2.5-coder:7b/ (✅ PASSED)
└── deepseek-coder:6.7b/ (✅ PASSED)
```

Model Categories Tested

Automatic Model Selection

The framework automatically selects appropriate model sizes based on available VRAM:

VRAM Available	Models Selected	Example Models
< 8GB	7B models	qwen3:8b, mistral:7b, codellama:7b
8GB+	13B models	qwen3:14b, mistral:12b, codellama:13b

VRAM Available	Models Selected	Example Models
12GB+	34B models	qwen3:32b, mixtral:8x22b, codellama:34b
24GB+	70B models	qwen3:72b, mixtral:8x70b, codellama:70b

Supported Categories

1. **General Models** - General-purpose conversation
    - Test: "What is 2+2? Answer briefly."
    - Expected: Response contains "4"
2. **Coder Models** - Code generation and debugging
    - Test: "Write a Python hello function. Show only code."
    - Expected: Response contains "def" and "hello"
3. **Tester Models** - Test generation and analysis
    - Test: "Write a unit test for a function that adds two numbers."
    - Expected: Response contains "test", "def", "assert"
4. **Translation Models** - Multilingual translation
    - Test: "Translate 'hello world' to Spanish."
    - Expected: Response contains "hola" and "mundo"
5. **Vision Models** (PNG/JPEG) - Visual content analysis
    - Test: "Describe what you see in this image: a red apple."
    - Expected: Response contains "apple" and "red"
6. **Generative Models** (SVG/Animation) - Code-based generation
    - Test: SVG/CSS generation prompts
    - Expected: Appropriate code structures
7. **Audio Models** - Music/speech generation (specialized framework)
    - Test: Audio generation and synthesis
    - Expected: Valid audio file output

Test-Fix-Retest Loop

Loop Logic

```
for iteration in 1..MAX_ITERATIONS:
    1. Run comprehensive model tests
    2. Identify failures and document issues
    3. If AUTO_FIX enabled:
       a. Apply codebase fixes
```

```
        b. Install missing models
        c. Adjust system configuration
        d. Increase timeouts if needed
    4. If fixes applied, re-run tests
    5. Continue until all pass or max iterations
```

Automatic Fixes Applied

Codebase Fixes

- **Script Typos**: Corrects common spelling errors
- **Dependency Checks**: Adds missing command validations
- **Permissions**: Makes scripts executable
- **Syntax Errors**: Fixes bash script syntax issues

System Fixes

- **Missing Models**: Automatically installs via `ollama pull`
- **Service Issues**: Starts Ollama service if stopped
- **Resource Limits**: Increases timeouts for slow models
- **Storage**: Checks available disk space

Configuration Fixes

- **Model Recipes**: Validates model file formats
- **Path Issues**: Corrects file and directory references
- **Environment**: Sets proper environment variables

Performance and Monitoring

Test Metrics

- **Success Rate**: Percentage of models passing tests
- **Response Time**: Average time per model test
- **Fix Effectiveness**: Percentage of issues automatically resolved
- **Iteration Count**: Number of test-fix cycles required

Sample Performance

```
╔═══════════════════════════════════════╗
║           Final Results               ║
╚═══════════════════════════════════════╝
Total Iterations: 1
Total Models Tested: 7
Passed: 6
Failed: 1
Auto-Fixed: 2
Success Rate: 86%
```

Resource Usage

- **CPU**: Minimal overhead, primarily model execution
- **Memory**: Uses existing model memory allocation
- **Storage**: Test results stored in `Tests/` directory
- **Network**: Required for model downloads during fixes

Integration with Builder System

Model Recipe Integration

The test framework automatically discovers models from the existing recipe system:

```
Scripts/Recipes/Models/
├── General/7B          # General conversation models
├── Coder/7B           # Code generation models
├── Translation/7B     # Translation models
└── Generative/        # Creative generation models
    ├── Audio/7B       # Audio generation models
    ├── PNG/7B         # Image analysis models
    └── SVG/7B         # SVG generation models
```

Installation Script Compatibility

- **install.sh**: Main installation entry point
- **install_ollama_models.sh**: Ollama model installation
- **install_audio_models.sh**: Specialized audio model installation
- **VRAM Detection**: Automatic hardware capability detection

Zero Breaking Changes

- All existing functionality preserved
- Installation workflows unchanged
- Model recipes format maintained
- Backward compatibility ensured

Troubleshooting

Common Issues

No Models Found

```
# Install models first
./Scripts/install.sh General
./test.sh
```

Permission Denied

```
# Make script executable
chmod +x test.sh
./test.sh
```

Ollama Service Issues

```
# Check service status
sudo systemctl status ollama

# Start service
sudo systemctl start ollama
./test.sh --auto-fix
```

Low Success Rate

```
# Run with auto-fix to resolve issues
./test.sh --auto-fix

# Check individual model reports
cat Tests/$(date +%Y-%m-%d)/*/Report.md
```

Disk Space Issues

```
# Check available space
df -h

# Clean old test results
rm -rf Tests/2025-01-*  # Remove old test data

# Run test with monitoring
./test.sh --auto-fix
```

Debug Mode

```
# Run with debug output
bash -x ./test.sh --auto-fix

# Check specific model issues
cat Tests/$(date +%Y-%m-%d)/model-name/Issues/*.md
```

Log Analysis

```
# View test progress
tail -f Tests/$(date +%Y-%m-%d)/COMPREHENSIVE_TEST_REPORT.md

# Check system info
cat Tests/$(date +%Y-%m-%d)/system_info.json

# Analyze model responses
ls Tests/$(date +%Y-%m-%d)/*/Generated/
```

Security Considerations

Data Privacy

- **Local Execution**: All tests run locally, no external API calls
- **Content Isolation**: Generated content stored in isolated directories
- **Temporary Files**: Test data organized by date for easy cleanup

Permission Model

- **Script Permissions**: Minimal permissions required
- **System Access**: Optional sudo for service management
- **File Access**: Read/write only to project directories

Safe Defaults

- **Fail-Safe**: Stops on first error without auto-fix
- **Timeout Limits**: Prevents resource exhaustion
- **Iteration Limits**: Maximum 5 fix iterations to prevent loops

Advanced Usage

Custom Test Patterns

```
# Modify test patterns in test.sh
# Example: Change expected pattern for General models
test_single_model "$model_name" "Custom prompt" "custom.*pattern"
```

Extended Model Testing

```
# Test specific categories only
# Edit run_test_iteration() function to customize
```

Integration with CI/CD

```
# Add to build pipeline
./test.sh || exit 1  # Fail build if tests fail

# Generate reports for artifacts
cp -r Tests/$(date +%Y-%m-%d) build-artifacts/
```

Monitoring and Alerting

```
# Check success rate
SUCCESS_RATE=$(./test.sh --auto-fix | grep "Success Rate" | awk '{print
$3}' | sed 's/%//')
if [ $SUCCESS_RATE -lt 80 ]; then
    echo "WARNING: Low success rate: $SUCCESS_RATE%"
fi
```

Future Enhancements

Planned Features

- **Parallel Testing**: Multi-threaded model testing for speed
- **Performance Benchmarks**: Response time and quality metrics
- **Custom Test Suites**: User-configurable test scenarios
- **Web Dashboard**: Browser-based results viewing
- **API Integration**: REST API for programmatic access

Extension Points

- **Custom Fixers**: Add specialized fix logic for new issue types
- **Test Plugins**: Modular test additions for new model categories
- **Report Formats**: JSON, XML, CSV output options
- **Notification Systems**: Email, Slack, webhook integrations

Contributing

Adding New Model Categories

1. Create model recipe files in `Scripts/Recipes/Models/NewCategory/`
2. Add test logic in `run_test_iteration()` function
3. Define appropriate test prompts and expected patterns
4. Test the new category thoroughly

Adding New Fix Types

1. Implement fix logic in `apply_codebase_fixes()` function
2. Add issue detection in `document_issue()` function
3. Create comprehensive fix documentation

4. Test fix effectiveness across different scenarios

Reporting Issues

- Document unexpected behavior with test results
- Include system information and model details
- Provide reproduction steps and error messages
- Suggest potential fixes or improvements

License and Support

This testing framework is part of the Builder project and follows the same licensing terms. For support, documentation updates, or feature requests, please refer to the main project repository.

The AI Model Testing Framework provides enterprise-grade quality assurance for your AI model infrastructure with automated testing, intelligent fixing, and comprehensive reporting capabilities.