🧪 Test Framework Documentation

Table of Contents

Overview

The Test Framework is a comprehensive testing system for AI models that implements an intelligent test-fix-retest loop. It automatically detects issues, applies fixes using AI providers, and continues testing until all models pass or maximum iterations are reached.

Key Features

- 🔄 **Test-Fix-Retest Loop**: Automated cycle until success
- 🤖 **AI-Powered Fixing**: Integration with multiple AI providers
- 📊 **Comprehensive Reporting**: Detailed test results and statistics
- 🎯 **Automatic Verification**: Confirms fixes actually work
- 🔧 **Multiple Fix Strategies**: AI-powered, rule-based, and manual fixes
- 📈 **Progress Tracking**: Real-time monitoring and iteration management

Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                    Test Framework Architecture                  │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│   ┌──────────────┐    ┌──────────────┐    ┌──────────────┐     │
│   │   Model      │──▶│   Execute    │──▶│   Analyze    │     │
│   │ Discovery    │    │   Tests      │    │  Results     │     │
│   └──────────────┘    └──────────────┘    └──────────────┘     │
│                                                  │              │
│                                                  ▼              │
│   ┌──────────────┐    ┌──────────────┐    ┌──────────────┐     │
│   │  Continue    │◀──│   Apply      │◀──│   Detect     │     │
│   │  Testing     │    │   Fixes      │    │   Issues     │     │
│   └──────────────┘    └──────────────┘    └──────────────┘     │
│                                                                 │
```

```
├──────────────────────────────────────────────────────────────────────┤
│                        Fix Strategies                                 │
├──────────────────────────────────────────────────────────────────────┤
│   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐            │
│   │ AI-Powered   │   │ Rule-Based   │   │ Codebase     │            │
│   │   Fixes      │   │   Fixes      │   │   Fixes      │            │
│   └──────────────┘   └──────────────┘   └──────────────┘            │
├──────────────────────────────────────────────────────────────────────┤
│                       Result Tracking                                 │
├──────────────────────────────────────────────────────────────────────┤
│   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐            │
│   │   Test       │   │  Progress    │   │   Final      │            │
│   │  Results     │   │  Tracking    │   │  Reports     │            │
│   └──────────────┘   └──────────────┘   └──────────────┘            │
└──────────────────────────────────────────────────────────────────────┘
```

Test-Fix-Retest Loop

Loop Overview

The framework implements a sophisticated loop that continues until all models pass or maximum iterations are reached:

```
1. Discovery Phase
   ├── Detect available models
   ├── Check system requirements
   └── Initialize test environment

2. Testing Phase
   ├── Execute tests on all models
   ├── Capture outputs and errors
   └── Classify issues

3. Analysis Phase
   ├── Identify failed models
   ├── Categorize issue types
   └── Determine fix strategies

4. Fixing Phase
   ├── Apply codebase fixes
   ├── Use AI-powered fixes
   ├── Apply model-specific fixes
   └── Verify fixes work

5. Validation Phase
   ├── Re-test fixed models
   ├── Confirm all models pass
   └── Generate final report
```

Loop Control Logic

```
# Main loop implementation
for iteration in 1..$MAX_ITERATIONS; do
    echo "🔄 Starting iteration $iteration"

    # Run tests
    run_all_tests

    # Check results
    if all_tests_passed; then
        echo "🎉 All tests passed!"
        break
    fi

    # Apply fixes if enabled
    if [ "$AUTO_FIX" = "true" ]; then
        apply_fixes
        if fixes_were_applied; then
            echo "🔧 Fixes applied, continuing to next iteration"
            continue
        fi
    fi

    echo "❌ No more fixes available"
    break
done

# Final confirmation
run_confirmation_tests
generate_final_report
```

Usage Guide

Basic Usage

1. Simple Test Run

```
# Run tests without auto-fix
./test.sh
```

2. Test with Auto-Fix (Default: Qwen)

```
# Enable automatic fixing with default AI provider
./test.sh --auto-fix
```

3. Test with Specific AI Provider

```
# Use Claude for advanced analysis
./test.sh --auto-fix --fixer=claude

# Use Qwen for fast local fixing
./test.sh --auto-fix --fixer=qwen
```

4. Test with Custom Date

```
# Use specific date for test results
./test.sh --auto-fix --date=2025-01-15
```

Command Line Options

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

Options:
  --auto-fix              Enable automatic fixing of detected issues
  --fixer=TYPE           Choose AI fixer (claude, qwen) - default: qwen
  --date=YYYY-MM-DD      Use specific date for test results
  --help                 Show help and available fixers

Examples:
  ./test.sh                                    # Basic test run
  ./test.sh --auto-fix                         # Auto-fix with Qwen
  ./test.sh --auto-fix --fixer=claude          # Auto-fix with Claude
  ./test.sh --auto-fix --date=2025-01-15       # Custom date
```

Environment Variables

```
# Auto-fix configuration
export AUTO_FIX=true                    # Enable auto-fix by default
export FIXER_TYPE=claude                # Default AI provider

# Test configuration
export MAX_ITERATIONS=5                 # Maximum test-fix cycles
export TIMEOUT_DURATION=30              # Model response timeout (seconds)

# Debug mode
export CLAUDE_DEBUG=1                   # Enable detailed logging
```

Configuration

System Configuration

1. Hardware Detection

The framework automatically detects system capabilities and adjusts model selection:

```
# VRAM detection for optimal model size
if command -v nvidia-smi &> /dev/null; then
    vram_mib=$(nvidia-smi --query-gpu=memory.total --
format=csv,noheader,nounits | head -1)
    vram_gb=$(echo "scale=1; $vram_mib / 1024" | bc -l)

    if (( $(echo "$vram_gb >= 24" | bc -l) )); then
        MODEL_SIZE="70B"
    elif (( $(echo "$vram_gb >= 12" | bc -l) )); then
        MODEL_SIZE="34B"
    elif (( $(echo "$vram_gb >= 8" | bc -l) )); then
        MODEL_SIZE="13B"
    else
        MODEL_SIZE="7B"
    fi
fi
```

2. Model Discovery

```
# Available models based on system capacity
discover_models() {
    local models=()

    case "$MODEL_SIZE" in
        "7B")
            models=("qwen3:8b" "mistral:7b" "llama3:8b")
            ;;
        "13B")
            models=("qwen3:8b" "mistral:7b" "llama3:8b" "codellama:13b")
            ;;
        "34B")
            models=("qwen3:8b" "mistral:7b" "llama3:8b" "codellama:34b")
            ;;
        "70B")
            models=("qwen3:8b" "mistral:7b" "llama3:8b" "llama3:70b")
            ;;
    esac

    echo "${models[@]}"
}
```

Test Configuration

1. Test Parameters

```
# Default test configuration
TEST_PROMPT="What is 2+2? Answer briefly."
EXPECTED_PATTERN=".*4.*"
TIMEOUT_DURATION=30
MAX_RESPONSE_LENGTH=1000
```

2. Custom Test Prompts

```
# Model-specific test prompts
get_test_prompt() {
    local model_name="$1"

    case "$model_name" in
        *"coder"*)
            echo "Write a Python function to add two numbers. Keep it
simple."
            ;;
        *"math"*)
            echo "Calculate 15 * 7 and show your work."
            ;;
        *)
            echo "What is 2+2? Answer briefly."
            ;;
    esac
}
```

Directory Structure

```
Tests/
└── YYYY-MM-DD/           # Date-based organization
    ├── model-name-1/
    │   ├── test_status.txt      # PASSED/FAILED
    │   ├── test_output.txt      # Model response
    │   ├── test_error.txt       # Error messages
    │   ├── ai_issue.json        # AI analysis data
    │   └── Issues/              # Issue documentation
    │       ├── TIMEOUT.md
    │       └── MODEL_NOT_AVAILABLE.md
    └── model-name-2/
        └── ...
```

Test Results

Result Classification

1. Test Status

Each model test results in one of these statuses:

- **PASSED**: Model responded correctly within timeout
- **FAILED**: Model failed due to various issues
- **SKIPPED**: Model not available for testing

2. Issue Types

Failed tests are classified into specific categories:

MODEL_NOT_AVAILABLE

```
# Model Not Available

**Issue**: The model 'model-name' is not available in Ollama.

**Symptoms**:
- Model not found in `ollama list`
- Error: "model 'model-name' not found"

**Automatic Fix**:
- Install model using `ollama pull model-name`
- Verify installation and retry
```

TIMEOUT

```
# Response Timeout

**Issue**: Model took longer than 30 seconds to respond.

**Symptoms**:
- Process killed after timeout
- Partial or no response received

**Automatic Fix**:
- Increase timeout duration
- Check system resources
- Consider using smaller model variant
```

UNEXPECTED_RESPONSE

```
# Unexpected Response Pattern

**Issue**: Model response doesn't match expected pattern.

**Expected**: Pattern matching ".*4.*"
```

```
**Actual**: "The answer is four"

**Automatic Fix**:
- Adjust prompt for clarity
- Update expected pattern
- Analyze response semantics
```

NO_OUTPUT

```
# No Output Received

**Issue**: Model produced no response.

**Symptoms**:
- Empty output file
- Model appears to run but produces no text

**Automatic Fix**:
- Restart Ollama service
- Check model integrity
- Verify system resources
```

Report Generation

1. Real-time Progress

```
# During test execution
[INFO] Starting test-fix-retest loop
[INFO] Auto-fix mode: ENABLED
[INFO] AI Fixer: deepseek
[INFO] Setting up test environment for 2025-01-15
[INFO] Discovered 8 models for testing
[INFO] Testing model 1/8: qwen3:8b
[SUCCESS] ✅ qwen3:8b responded correctly
[INFO] Testing model 2/8: mistral:7b
[ERROR] ❌ mistral:7b timed out after 30 seconds
```

2. Iteration Summary

```
# After each iteration
╔═══════════════════════════════════════╗
║           Iteration 1 Summary         ║
╚═══════════════════════════════════════╝

📊 Results:
  • Total Models: 8
```

```
  • Passed: 6 (75.0%)
  • Failed: 2 (25.0%)
  • Success Rate: 75.0%

🔧 Auto-fix enabled (using: qwen)
🧠 Memory: 15 fixes tried, 87.2% success rate

❌ Failed Models:
  • mistral:7b (TIMEOUT)
  • codellama:13b (MODEL_NOT_AVAILABLE)

🔄 Applying fixes and retesting...
```

3. Final Report

```
# AI Model Testing Report

**Date**: 2025-01-15
**Total Iterations**: 3
**Auto-Fix Enabled**: true (using: qwen)

## Final Results

| Metric | Count | Percentage |
|--------|-------|------------|
| Total Models | 8 | 100.0% |
| Passed Models | 8 | 100.0% |
| Failed Models | 0 | 0.0% |
| Fixed Models | 2 | 25.0% |

## Performance Summary

- **Overall Success Rate**: 100.0%
- **Average Response Time**: 2.3 seconds
- **Total Test Duration**: 45 minutes
- **Fixes Applied**: 2 successful

## Iteration Details

### Iteration 1
- **Models Tested**: 8
- **Passed**: 6
- **Failed**: 2 (mistral:7b, codellama:13b)

### Iteration 2
- **Models Tested**: 2 (retesting fixed models)
- **Passed**: 2
- **Failed**: 0

### Iteration 3 (Confirmation)
- **Models Tested**: 8 (all models)
```

```
- **Passed**: 8
- **Failed**: 0

## Fix Summary

### mistral:7b
- **Issue**: TIMEOUT
- **Fix Applied**: Increased timeout to 60 seconds
- **Fixer Used**: qwen
- **Result**: ✅ Fixed successfully

### codellama:13b
- **Issue**: MODEL_NOT_AVAILABLE
- **Fix Applied**: `ollama pull codellama:13b`
- **Fixer Used**: Rule-based
- **Result**: ✅ Fixed successfully

## System Information

- **OS**: Linux 6.14.0-29-generic
- **GPU**: NVIDIA RTX 4090 (24GB VRAM)
- **RAM**: 32GB
- **Ollama Version**: 0.1.17
- **Model Size Category**: 70B

---
**Report generated automatically by AI Model Testing Framework**
```

Troubleshooting

Common Issues

1. Framework Issues

Problem: "No models found for testing"

```
# Check Ollama installation
ollama --version

# List available models
ollama list

# Install basic models
ollama pull qwen3:8b
ollama pull mistral:7b
```

Problem: "Permission denied" errors

```
```

```
# Fix script permissions
chmod +x Scripts/test.sh
chmod +x Scripts/AutoFixers/*.py
chmod +x ai_fixers.sh
chmod +x memory.sh
```

Problem: "Tests directory not found"

```
# Framework creates directories automatically
# Ensure you're running from project root
pwd  # Should be in Builder directory
./Scripts/test.sh --auto-fix
```

2. Model Issues

Problem: Models consistently timing out

```
# Increase timeout globally
export TIMEOUT_DURATION=60

# Check system resources
free -h
nvidia-smi  # For GPU memory

# Use smaller models
export MODEL_SIZE="7B"
```

Problem: Models giving unexpected responses

```
# Check model integrity
ollama run model-name "Test prompt"

# Update model
ollama pull model-name

# Verify expected patterns are correct
grep -n "EXPECTED_PATTERN" Scripts/test.sh
```

3. Auto-Fix Issues

Problem: Auto-fix not working

```
# Check AI fixer availability
./ai_fixers.sh list

# Set up at least one fixer
# For Qwen: ollama pull qwen2.5-coder:7b
# For Claude: export ANTHROPIC_API_KEY="your-key"

# Enable debug mode
export CLAUDE_DEBUG=1
./Scripts/test.sh --auto-fix
```

Debug Mode

Enable comprehensive debugging:

```
export CLAUDE_DEBUG=1
./Scripts/test.sh --auto-fix --fixer=qwen
```

Debug Output Includes:

- Detailed command execution
- AI analysis JSON
- Fix command traces
- Memory system operations
- Verification steps

Log Files

Check these locations for detailed information:

```
# Test results
ls -la Tests/$(date +%Y-%m-%d)/

# AI analysis data
cat Tests/$(date +%Y-%m-%d)/*/ai_issue.json

# Memory system logs
sqlite3 ClaudeMemory/claude_memory.db ".schema"
```

Advanced Features

1. Custom Test Scenarios

Creating Custom Tests

```
# Create custom test function
run_custom_test() {
    local model_name="$1"
    local custom_prompt="$2"
    local expected_pattern="$3"

    # Implementation...
}

# Use in main test loop
for model in "${MODELS[@]}"; do
    run_custom_test "$model" "Custom prompt" "Custom pattern"
done
```

Multi-Phase Testing

```
# Phase 1: Basic functionality
run_basic_tests

# Phase 2: Performance tests
run_performance_tests

# Phase 3: Stress tests
run_stress_tests
```

2. Integration with CI/CD

GitHub Actions Example

```
name: AI Model Testing

on:
  schedule:
    - cron: '0 2 * * *'  # Daily at 2 AM
  push:
    branches: [ main ]

jobs:
  test-models:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Setup Ollama
      run: |
        curl -fsSL https://ollama.ai/install.sh | sh
        ollama pull qwen2.5-coder:7b
```

```yaml
    - name: Run AI Model Tests
      env:
        ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
      run: |
        ./Scripts/test.sh --auto-fix --fixer=claude

    - name: Upload Results
      uses: actions/upload-artifact@v3
      with:
        name: test-results
        path: Tests/
```

3. Parallel Testing

Concurrent Model Testing

```bash
# Test multiple models in parallel
parallel_test() {
    local models=("$@")
    local pids=()

    for model in "${models[@]}"; do
        test_single_model "$model" &
        pids+=($!)
    done

    # Wait for all tests to complete
    for pid in "${pids[@]}"; do
        wait "$pid"
    done
}
```

4. Custom Reporting

JSON Output

```bash
# Generate JSON report
generate_json_report() {
    local output_file="$1"

    cat > "$output_file" << EOF
{
    "test_date": "$(date -Iseconds)",
    "total_models": $TOTAL_MODELS,
    "passed_models": $PASSED_MODELS,
    "failed_models": $FAILED_MODELS,
    "success_rate": $(echo "scale=4; $PASSED_MODELS / $TOTAL_MODELS" | bc),
```

```
    "iterations": $CURRENT_ITERATION,
    "auto_fix_enabled": $AUTO_FIX,
    "fixer_type": "$FIXER_TYPE",
    "results": [...]
}
EOF
}
```

Customization

1. Adding New Model Categories

```
# Add new model type detection
detect_model_type() {
    local model_name="$1"

    case "$model_name" in
        *"vision"*|*"llava"*|*"minicpm-v"*)
            echo "vision"
            ;;
        *"coder"*|*"code"*|*"starcoder"*)
            echo "coding"
            ;;
        *"math"*|*"calculator"*)
            echo "mathematical"
            ;;
        *)
            echo "general"
            ;;
    esac
}

# Custom test prompts by type
get_test_prompt_by_type() {
    local model_type="$1"

    case "$model_type" in
        "vision")
            echo "Describe what you see in this image: [placeholder]"
            ;;
        "coding")
            echo "Write a Python function to calculate factorial of n"
            ;;
        "mathematical")
            echo "Solve: 2x + 5 = 17. What is x?"
            ;;
        *)
            echo "What is 2+2? Answer briefly."
            ;;
    esac
}
```

2. Custom Fix Strategies

```
# Add custom fix type
apply_custom_fixes() {
    local issue_type="$1"
    local model_name="$2"

    case "$issue_type" in
        "MEMORY_ERROR")
            # Custom memory optimization
            optimize_memory_for_model "$model_name"
            ;;
        "VERSION_MISMATCH")
            # Custom version handling
            update_model_version "$model_name"
            ;;
        *)
            return 1  # Fallback to standard fixes
            ;;
    esac
}
```

3. Custom Metrics

```
# Add performance metrics
collect_performance_metrics() {
    local model_name="$1"
    local start_time="$2"
    local end_time="$3"

    local response_time=$((end_time - start_time))
    local memory_usage=$(get_memory_usage)
    local cpu_usage=$(get_cpu_usage)

    # Store metrics
    echo "{
        \"model\": \"$model_name\",
        \"response_time\": $response_time,
        \"memory_usage\": $memory_usage,
        \"cpu_usage\": $cpu_usage
    }" >> "performance_metrics.json"
}
```

API Reference

Core Functions

test_model()

```
test_model() {
    local model_name="$1"
    local test_prompt="$2"
    local expected_pattern="$3"
    local timeout="$4"

    # Test implementation
    # Returns: 0 = success, 1 = failure
}
```

apply_fixes()

```
apply_fixes() {
    # Apply all available fix strategies
    # Returns: number of fixes applied

    local fixes_applied=0

    # Codebase fixes
    apply_codebase_fixes
    fixes_applied=$((fixes_applied + $?))

    # AI-powered fixes
    if [ "$AUTO_FIX" = "true" ]; then
        apply_ai_fixes
        fixes_applied=$((fixes_applied + $?))
    fi

    # Model-specific fixes
    apply_model_fixes
    fixes_applied=$((fixes_applied + $?))

    return $fixes_applied
}
```

generate_report()

```
generate_report() {
    local output_format="$1"  # markdown, json, csv
    local output_file="$2"

    case "$output_format" in
        "markdown")
            generate_markdown_report "$output_file"
            ;;
```

```
        "json")
            generate_json_report "$output_file"
            ;;
        "csv")
            generate_csv_report "$output_file"
            ;;
    esac
}
```

Configuration Variables

```
# Test configuration
TIMEOUT_DURATION=30           # Model response timeout
MAX_ITERATIONS=5              # Maximum test-fix cycles
MODEL_SIZE="auto"             # Auto-detect or specify
TEST_DATE=$(date +%Y-%m-%d)   # Test result date

# Auto-fix configuration
AUTO_FIX=false               # Enable automatic fixing
FIXER_TYPE="qwen"            # Default AI provider

# Output configuration
TESTS_DIR="Tests/$TEST_DATE"  # Results directory
VERBOSE=false                 # Detailed output
DEBUG=false                   # Debug mode
```

Exit Codes

```
# Test framework exit codes
EXIT_SUCCESS=0               # All tests passed
EXIT_SOME_FAILED=1          # Some tests failed, no fixes applied
EXIT_ALL_FAILED=2           # All tests failed
EXIT_MAX_ITERATIONS=3       # Reached maximum iterations
EXIT_SETUP_ERROR=4          # Setup/configuration error
EXIT_SYSTEM_ERROR=5         # System/dependency error
```

🧪 The Test Framework provides a robust foundation for continuous AI model testing with intelligent failure recovery and comprehensive reporting.