CI/DI_agent.py

```
import os
import re
import ast
import json
import shutil
import subprocess
from pathlib import Path
from typing import Dict, List, Tuple, Optional, Any
from datetime import datetime
import time
import yaml  # For CI/CD configuration
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain.chains import RetrievalQA
from langchain_community.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.schema import Document

class CI_CDHandler(FileSystemEventHandler):
    """Monitor file changes and trigger CI/CD pipeline"""
    def __init__(self, agent: Any, patterns: List[str]):
        self.agent = agent
        self.patterns = patterns
        self.last_trigger = 0
        self.cooldown = 5  # seconds between triggers

    def on_modified(self, event):
        if time.time() - self.last_trigger < self.cooldown:
            return

        if any(event.src_path.endswith(p) for p in self.patterns):
            self.last_trigger = time.time()
            print(f"\nCI/CD Triggered by: {event.src_path}")
            self.agent.run_ci_cd_pipeline()

class SelfImprovingCodingAgent:
    # Existing constants and __init__...

    def __init__(self,
                 workspace: str = "coding_agent_workspace",
```

```
            model_name: str = "gpt-4-turbo",
            research_agent: Optional[object] = None,
            user_project_dir: str = None,
            enable_ci_cd: bool = True):
    # Existing initialization...

    self.ci_cd_dir = self.workspace / "ci_cd"
    self.test_results_dir = self.ci_cd_dir / "results"
    self.ci_config_file = self.ci_cd_dir / "config.yaml"

    self.create_workspace()
    self.setup_ci_cd()

    # CI/CD monitoring
    self.ci_cd_enabled = enable_ci_cd
    if enable_ci_cd:
        self.start_ci_cd_monitor()

def create_workspace(self) -> None:
    """Create the workspace directory structure with necessary permissions"""
    # Existing directories...

    # Add CI/CD directories
    self.ci_cd_dir.mkdir(exist_ok=True)
    self.test_results_dir.mkdir(exist_ok=True)

    # Set permissions
    os.chmod(self.ci_cd_dir, 0o755)
    os.chmod(self.test_results_dir, 0o755)

    print(f"  - CI/CD Configuration: {self.ci_cd_dir}")
    print(f"  - Test Results: {self.test_results_dir}")

def setup_ci_cd(self) -> None:
    """Initialize CI/CD configuration"""
    if not self.ci_config_file.exists():
        default_config = {
            'version': '1.0',
            'pipelines': {
                'on_push': {
                    'trigger': ['*.py', '*.rs', '*.java', '*.c', '*.cpp', '*.h'],
                    'actions': [
                        {'name': 'run_tests', 'type': 'test'},
                        {'name': 'static_analysis', 'type': 'analysis'},
```

```
                    {'name': 'security_scan', 'type': 'security'}
                ]
            },
            'on_schedule': {
                'cron': '0 0 * * *',  # Daily at midnight
                'actions': [
                    {'name': 'full_test_suite', 'type': 'test'},
                    {'name': 'generate_reports', 'type': 'report'}
                ]
            }
        },
        'language_configs': {
            'python': {
                'test_command': 'pytest --junitxml={results_dir}/results.xml',
                'coverage_command': 'coverage run -m pytest && coverage xml -o
{results_dir}/coverage.xml',
                'linter_command': 'flake8 --format=pylint {file} > {results_dir}/lint.txt',
                'security_command': 'bandit -r {path} -f json -o {results_dir}/security.json'
            },
            'rust': {
                'test_command': 'cargo test -- --test-threads=1 --format=json >
{results_dir}/results.json',
                'coverage_command': 'cargo tarpaulin --out Xml --output-dir {results_dir}',
                'linter_command': 'cargo clippy --message-format=json > {results_dir}/lint.json',
                'security_command': 'cargo audit --json > {results_dir}/security.json'
            },
            # Configs for other languages...
        }
    }

    with open(self.ci_config_file, 'w') as f:
        yaml.dump(default_config, f)

def start_ci_cd_monitor(self) -> None:
    """Start monitoring for file changes to trigger CI/CD"""
    event_handler = CI_CDHandler(
        self,
        patterns=['*.py', '*.rs', '*.java', '*.c', '*.cpp', '*.h']
    )

    observer = Observer()
    observer.schedule(event_handler, str(self.user_project_dir), recursive=True)
    observer.schedule(event_handler, str(self.generated_code_dir), recursive=True)
    observer.schedule(event_handler, str(self.self_improvement_dir), recursive=True)
```

```
    observer.start()

    print(f"Started CI/CD monitoring on directories:")
    print(f"  - {self.user_project_dir}")
    print(f"  - {self.generated_code_dir}")
    print(f"  - {self.self_improvement_dir}")

def run_ci_cd_pipeline(self, scope: str = "all") -> Dict:
    """Run the full CI/CD pipeline"""
    print("\n" + "="*50)
    print("Starting CI/CD Pipeline")
    print("="*50)

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    results = {
        "timestamp": timestamp,
        "tests": {},
        "coverage": {},
        "lints": {},
        "security": {},
        "success": True
    }

    # Load CI/CD configuration
    with open(self.ci_config_file, 'r') as f:
        config = yaml.safe_load(f)

    # Determine scope of files to test
    test_files = []
    if scope == "all":
        test_files.extend(self.get_code_files(self.user_project_dir))
        test_files.extend(self.get_code_files(self.generated_code_dir))
        test_files.extend(self.get_code_files(self.self_improvement_dir))
    else:
        test_files.extend(self.get_code_files(Path(scope)))

    # Process each code file
    for file_path in test_files:
        print(f"\nProcessing: {file_path}")
        language = self.detect_language_from_extension(file_path.suffix)

        if not language or language not in config['language_configs']:
            print(f"  Skipping - Unsupported language for CI/CD")
            continue
```

```
            # Create language-specific results directory
            lang_results_dir = self.test_results_dir / language / timestamp
            lang_results_dir.mkdir(parents=True, exist_ok=True)

            # Run tests
            test_result = self.run_tests(file_path, language, lang_results_dir, config)
            results["tests"][str(file_path)] = test_result
            if not test_result["success"]:
                results["success"] = False

            # Run code coverage
            coverage_result = self.run_coverage(file_path, language, lang_results_dir, config)
            results["coverage"][str(file_path)] = coverage_result

            # Run linter
            lint_result = self.run_linter(file_path, language, lang_results_dir, config)
            results["lints"][str(file_path)] = lint_result

            # Run security scan
            security_result = self.run_security_scan(file_path, language, lang_results_dir, config)
            results["security"][str(file_path)] = security_result

        # Generate overall report
        report_path = self.generate_ci_report(results, timestamp)
        print(f"\nCI/CD Pipeline Complete")
        print(f"  Full Report: {report_path}")

        # If tests failed, trigger self-improvement
        if not results["success"]:
            print("\nCI/CD Failures Detected - Triggering Self-Improvement")
            self.auto_self_improve()

        return results

    def get_code_files(self, directory: Path) -> List[Path]:
        """Get all code files in a directory"""
        code_files = []
        for ext in [ext for exts in self.LANGUAGE_EXTENSIONS.values() for ext in exts]:
            code_files.extend(directory.rglob(f"*{ext}"))
        return code_files

    def run_tests(self, file_path: Path, language: str, results_dir: Path, config: Dict) -> Dict:
        """Run tests for a code file"""
```

```
    lang_config = config['language_configs'][language]
    test_cmd = lang_config['test_command'].format(
        file=file_path.name,
        path=file_path.parent,
        results_dir=results_dir
    )

    print(f"  Running Tests: {test_cmd}")
    start_time = time.time()

    try:
        result = subprocess.run(
            test_cmd.split(),
            cwd=file_path.parent,
            capture_output=True,
            text=True
        )

        # Parse test results
        test_report = {
            "command": test_cmd,
            "exit_code": result.returncode,
            "stdout": result.stdout,
            "stderr": result.stderr,
            "success": result.returncode == 0,
            "duration": time.time() - start_time
        }

        # Save raw output
        with open(results_dir / "test_output.txt", "w") as f:
            f.write(f"Exit Code: {result.returncode}\n")
            f.write(f"Stdout:\n{result.stdout}\n")
            f.write(f"Stderr:\n{result.stderr}\n")

        return test_report
    except Exception as e:
        return {
            "success": False,
            "error": str(e),
            "duration": time.time() - start_time
        }

def run_coverage(self, file_path: Path, language: str, results_dir: Path, config: Dict) -> Dict:
    """Run code coverage analysis"""
```

```
    lang_config = config['language_configs'][language]
    if 'coverage_command' not in lang_config:
        return {"success": True, "message": "Coverage not configured"}

    coverage_cmd = lang_config['coverage_command'].format(
        file=file_path.name,
        path=file_path.parent,
        results_dir=results_dir
    )

    print(f"  Running Coverage: {coverage_cmd}")
    start_time = time.time()

    try:
        result = subprocess.run(
            coverage_cmd.split(),
            cwd=file_path.parent,
            capture_output=True,
            text=True
        )

        coverage_report = {
            "command": coverage_cmd,
            "exit_code": result.returncode,
            "stdout": result.stdout,
            "stderr": result.stderr,
            "success": result.returncode == 0,
            "duration": time.time() - start_time
        }

        # Parse coverage report if available
        if (results_dir / "coverage.xml").exists():
            # In real implementation, parse XML to get coverage percentage
            coverage_report["coverage"] = "Available in coverage.xml"

        return coverage_report
    except Exception as e:
        return {
            "success": False,
            "error": str(e),
            "duration": time.time() - start_time
        }

def run_linter(self, file_path: Path, language: str, results_dir: Path, config: Dict) -> Dict:
```

```
        """Run static code analysis"""
        lang_config = config['language_configs'][language]
        if 'linter_command' not in lang_config:
            return {"success": True, "message": "Linter not configured"}

        lint_cmd = lang_config['linter_command'].format(
            file=file_path.name,
            path=file_path.parent,
            results_dir=results_dir
        )

        print(f"  Running Linter: {lint_cmd}")
        start_time = time.time()

        try:
            result = subprocess.run(
                lint_cmd,
                shell=True,  # Some commands need shell
                cwd=file_path.parent,
                capture_output=True,
                text=True
            )

            lint_report = {
                "command": lint_cmd,
                "exit_code": result.returncode,
                "stdout": result.stdout,
                "stderr": result.stderr,
                "success": result.returncode == 0,
                "duration": time.time() - start_time
            }

            return lint_report
        except Exception as e:
            return {
                "success": False,
                "error": str(e),
                "duration": time.time() - start_time
            }

    def run_security_scan(self, file_path: Path, language: str, results_dir: Path, config: Dict) ->
Dict:
        """Run security vulnerability scan"""
        lang_config = config['language_configs'][language]
```

```
if 'security_command' not in lang_config:
    return {"success": True, "message": "Security scan not configured"}

security_cmd = lang_config['security_command'].format(
    file=file_path.name,
    path=file_path.parent,
    results_dir=results_dir
)

print(f"  Running Security Scan: {security_cmd}")
start_time = time.time()

try:
    result = subprocess.run(
        security_cmd,
        shell=True,
        cwd=file_path.parent,
        capture_output=True,
        text=True
    )

    security_report = {
        "command": security_cmd,
        "exit_code": result.returncode,
        "stdout": result.stdout,
        "stderr": result.stderr,
        "success": result.returncode == 0,
        "duration": time.time() - start_time
    }

    # Parse security report if available
    if (results_dir / "security.json").exists():
        try:
            with open(results_dir / "security.json", "r") as f:
                security_data = json.load(f)
            security_report["issues"] = len(security_data.get("vulnerabilities", []))
        except:
            pass

    return security_report
except Exception as e:
    return {
        "success": False,
        "error": str(e),
```

```
            "duration": time.time() - start_time
        }

    def generate_ci_report(self, results: Dict, timestamp: str) -> Path:
        """Generate a CI/CD pipeline report"""
        report_path = self.test_results_dir / f"ci_report_{timestamp}.json"

        with open(report_path, "w") as f:
            json.dump(results, f, indent=2)

        # Generate summary markdown
        summary_path = self.test_results_dir / f"summary_{timestamp}.md"
        with open(summary_path, "w") as f:
            f.write(f"# CI/CD Pipeline Report - {timestamp}\n\n")
            f.write(f"**Overall Status:** {'✅ SUCCESS' if results['success'] else '❌ FAILURE'}\n\n")

            f.write("## Test Results\n")
            for file, test in results["tests"].items():
                status = "✅ PASSED" if test["success"] else "❌ FAILED"
                f.write(f"- `{file}`: {status} ({test['duration']:.2f}s)\n")

            f.write("\n## Security Scan Summary\n")
            for file, scan in results["security"].items():
                issues = scan.get("issues", "N/A")
                f.write(f"- `{file}`: {issues} issues found\n")

            f.write("\n## Next Steps\n")
            if results["success"]:
                f.write("- All tests passed successfully\n")
            else:
                f.write("- Investigate failed tests\n")
                f.write("- Run self-improvement process\n")
                f.write("- Review detailed reports\n")

        return summary_path

    def generate_test_cases(self, file_path: Path) -> Path:
        """Generate test cases for a code file"""
        with open(file_path, "r") as f:
            code = f.read()

        language = self.detect_language_from_extension(file_path.suffix)

        prompt = f"""
```

```
        You are a senior test engineer. Generate comprehensive test cases for the following {language} code.
        Include positive, negative, edge case, and performance tests.
        Provide the tests in the appropriate testing framework for the language.

        Code:
        ```{language}
        {code}
        ```

        Respond with the complete test file in a single code block.
        """

        # Generate tests using LLM
        test_code = self.llm(prompt)

        # Extract code block
        if "```" in test_code:
            test_code = test_code.split("```")[1]
            if test_code.startswith(language):
                test_code = test_code[len(language):].strip()

        # Determine test file path
        test_dir = file_path.parent / "tests"
        test_dir.mkdir(exist_ok=True)

        test_filename = f"test_{file_path.stem}.{self.language_config[language]['extension']}"
        test_path = test_dir / test_filename

        with open(test_path, "w") as f:
            f.write(test_code)

        print(f"Generated tests for {file_path.name} at {test_path}")
        return test_path

    # Update existing methods to integrate CI/CD

    def generate_code(self, task: str, language: str, filename: str = None) -> Dict:
        """Generate code with CI/CD integration"""
        # Existing generation logic...

        # Generate tests if they don't exist
        test_path = self.generate_test_cases(Path(result["agent_path"]))
```

```
        # Run initial CI/CD
        if self.ci_cd_enabled:
            ci_results = self.run_ci_cd_pipeline(scope=str(Path(result["agent_path"]).parent))
            result["ci_results"] = ci_results

            # If tests failed, immediately try to improve
            if not ci_results["success"]:
                print("Initial CI/CD failed - triggering self-improvement")
                self.trigger_improvement_from_ci(result["agent_path"], ci_results)

        return result

    def trigger_improvement_from_ci(self, file_path: str, ci_results: Dict) -> None:
        """Trigger self-improvement based on CI/CD results"""
        with open(file_path, "r") as f:
            code = f.read()

        # Create improvement prompt from CI results
        failures = []
        for test_file, test_result in ci_results["tests"].items():
            if not test_result["success"]:
                failures.append(f"- {test_file}: Exit code
{test_result['exit_code']}\n{test_result['stderr']}")

        prompt = f"""
        Our code failed CI/CD tests. Here are the details:

        Failed Tests:
        {chr(10).join(failures)}

        Original Code:
        ```{self.detect_language_from_extension(Path(file_path).suffix)}
        {code}
        ```

        Please fix the code to pass all tests.
        Respond with the complete fixed code in a single code block.
        """

        # Generate improved code
        improved_code = self.llm(prompt)

        # Extract code block
        if "```" in improved_code:
```

```
            improved_code = improved_code.split("```")[1]
            if improved_code.startswith(language):
                improved_code = improved_code[len(language):].strip()

        # Save improved version
        orig_path = Path(file_path)
        improved_filename = f"improved_{orig_path.name}"
        improved_path = self.self_improvement_dir / improved_filename

        with open(improved_path, "w") as f:
            f.write(improved_code)

        print(f"Saved CI/CD-triggered improvement to: {improved_path}")

        # Re-run CI/CD on improved version
        if self.ci_cd_enabled:
            self.run_ci_cd_pipeline(scope=str(improved_path.parent))

    # Update auto_self_improve to include CI/CD
    def auto_self_improve(self) -> List:
        """Automated self-improvement with CI/CD validation"""
        improvements = []

        # Find code to improve (same as before)...

        for code_file in code_files:
            # Run CI/CD first
            if self.ci_cd_enabled:
                ci_results = self.run_ci_cd_pipeline(scope=str(code_file.parent))

                # Only improve if tests fail
                if ci_results["success"]:
                    print(f"Code passed CI/CD: {code_file} - skipping improvement")
                    continue

            # Existing improvement logic...

        return improvements

# Example Usage
if __name__ == "__main__":
    # Create coding agent with CI/CD enabled
    coding_agent = SelfImprovingCodingAgent(
        workspace="coding_agent_workspace",
```

```
    model_name="gpt-4-turbo",
    user_project_dir="user_project",
    enable_ci_cd=True
)

# Generate code - will automatically trigger CI/CD
python_task = "Implement a function to calculate Fibonacci sequence"
python_result = coding_agent.generate_code(python_task, "python", "fibonacci.py")

# Manually modify code to trigger CI/CD
with open("user_project/fibonacci.py", "a") as f:
    f.write("\n# Introducing an error\ndef error_func():\n    return undefined_var\n")

# CI/CD will automatically trigger and detect error
# After detection, self-improvement will be triggered

# Run full pipeline manually
print("\nRunning full CI/CD pipeline")
coding_agent.run_ci_cd_pipeline()

# Schedule CI/CD (in production would use cron or scheduler)
import schedule
import time

def scheduled_ci_cd():
    print("\nRunning scheduled CI/CD pipeline")
    coding_agent.run_ci_cd_pipeline()

schedule.every().day.at("00:00").do(scheduled_ci_cd)

print("Scheduled CI/CD to run daily at midnight")
while True:
    schedule.run_pending()
    time.sleep(1)
```