```
import os
import json
import yaml
import time
from pathlib import Path
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
import subprocess

class CI_CDSystem:
    """Dual-path CI/CD system with separate pipelines for user and agent code"""
    def __init__(self, agent):
        self.agent = agent
        self.config_dir = agent.workspace / "ci_cd_config"
        self.user_results_dir = agent.workspace / "ci_cd_results/user"
        self.agent_results_dir = agent.workspace / "ci_cd_results/agent"

        # Create directories
        self.config_dir.mkdir(exist_ok=True, parents=True)
        self.user_results_dir.mkdir(exist_ok=True)
        self.agent_results_dir.mkdir(exist_ok=True)

        # Initialize configurations
        self.user_config = self.load_or_create_config("user")
        self.agent_config = self.load_or_create_config("agent")

        # Start monitoring
        self.start_monitors()

    def load_or_create_config(self, config_type: str) -> dict:
        """Load or create CI/CD configuration"""
        config_path = self.config_dir / f"{config_type}_config.yaml"

        if not config_path.exists():
            default_config = {
                'version': '1.0',
                'monitor_paths': [],
                'actions': {
                    'on_change': ['run_tests', 'static_analysis'],
                    'on_demand': ['security_scan', 'coverage']
                },
                'language_configs': {}
            }
            # Type-specific defaults
            if config_type == "user":
                default_config['monitor_paths'] = [str(self.agent.user_project_dir)]
                default_config['reporting'] = {'email': '', 'webhook': ''}
            else:  # agent
                default_config['monitor_paths'] = [
                    str(self.agent.generated_code_dir),
                    str(self.agent.self_improvement_dir)
                ]
                default_config['auto_improve'] = True

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

        with open(config_path, 'r') as f:
            return yaml.safe_load(f)

    def start_monitors(self):
        """Start separate monitors for user and agent codebases"""
        # Agent code monitor (self-improvement codebase)
        agent_handler = AgentCIHandler(self.agent, self.agent_config)
        self.agent_observer = Observer()
        for path in self.agent_config['monitor_paths']:
            self.agent_observer.schedule(agent_handler, path, recursive=True)
        self.agent_observer.start()

        # User code monitor (user project)
        user_handler = UserCIHandler(self.agent, self.user_config)
        self.user_observer = Observer()
        for path in self.user_config['monitor_paths']:
            self.user_observer.schedule(user_handler, path, recursive=True)
        self.user_observer.start()

    def run_pipeline(self, scope: list, config_type: str, trigger: str = "on_demand") -> dict:
        """Run CI/CD pipeline for specific scope"""
        config = self.user_config if config_type == "user" else self.agent_config
        results_dir = self.user_results_dir if config_type == "user" else self.agent_results_dir

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        run_dir = results_dir / timestamp
        run_dir.mkdir(parents=True, exist_ok=True)

        results = {
            "config_type": config_type,
            "trigger": trigger,
            "timestamp": timestamp,
            "scope": scope,
            "actions": {}
        }

        # Run configured actions
        for action in config['actions'][trigger]:
            action_results = []
            for path in scope:
                action_results.append(self.run_action(action, Path(path), run_dir, config))
            results["actions"][action] = action_results

        # Save results
        report_path = run_dir / "report.json"
        with open(report_path, 'w') as f:
            json.dump(results, f, indent=2)

        # Agent-specific: trigger self-improvement on failure
        if config_type == "agent" and config.get('auto_improve', False):
            if any(not r['success'] for r in results["actions"]["run_tests"]):
                self.agent.auto_self_improve(scope)

        return results

    def run_action(self, action: str, path: Path, run_dir: Path, config: dict) -> dict:
        """Run a specific CI/CD action"""
        language = self.agent.detect_language_from_extension(path.suffix)
        lang_config = config['language_configs'].get(language, {})

        # Get command from config or default
        if action == "run_tests":
            cmd = lang_config.get('test_command',
self.agent.language_config[language]['test_command'])
        elif action == "static_analysis":
            cmd = lang_config.get('linter_command',
self.agent.language_config[language]['linter_command'])
        elif action == "security_scan":
            cmd = lang_config.get('security_command',
self.agent.language_config[language]['security_command'])
        elif action == "coverage":
            cmd = lang_config.get('coverage_command',
self.agent.language_config[language]['coverage_command'])
```

```
        else:
            return {"success": False, "error": f"Unknown action: {action}"}

        # Format command with parameters
        formatted_cmd = cmd.format(
            file=path.name,
            path=path.parent,
            results_dir=run_dir
        )

        try:
            # Execute command
            result = subprocess.run(
                formatted_cmd.split(),
                cwd=path.parent,
                capture_output=True,
                text=True,
                timeout=300  # 5 minutes
            )

            return {
                "path": str(path),
                "command": formatted_cmd,
                "exit_code": result.returncode,
                "stdout": result.stdout,
                "stderr": result.stderr,
                "success": result.returncode == 0
            }
        except Exception as e:
            return {
                "path": str(path),
                "success": False,
                "error": str(e)
            }

class AgentCIHandler(FileSystemEventHandler):
    """CI handler for agent's self-improving codebase"""
    def __init__(self, agent, config):
        self.agent = agent
        self.config = config
        self.last_trigger = 0
        self.cooldown = 5  # seconds

    def on_modified(self, event):
        if not event.is_directory and time.time() - self.last_trigger > self.cooldown:
            self.last_trigger = time.time()
            file_path = Path(event.src_path)

            # Only trigger for code files
            if any(file_path.suffix == ext for ext in
                  [e for el in SelfImprovingCodingAgent.LANGUAGE_EXTENSIONS.values() for e in
el]):

                print(f"\nAGENT CI: Detected change in self-improvement codebase: {file_path}")
                self.agent.ci_cd.run_pipeline(
                    scope=[str(file_path)],
                    config_type="agent",
                    trigger="on_change"
                )

class UserCIHandler(FileSystemEventHandler):
    """CI handler for user project code"""
    def __init__(self, agent, config):
        self.agent = agent
        self.config = config
        self.last_trigger = 0
        self.cooldown = 10  # seconds

    def on_modified(self, event):
        if not event.is_directory and time.time() - self.last_trigger > self.cooldown:
            self.last_trigger = time.time()
            file_path = Path(event.src_path)

            # Only trigger for code files
            if any(file_path.suffix == ext for ext in
                  [e for el in SelfImprovingCodingAgent.LANGUAGE_EXTENSIONS.values() for e in
el]):

                print(f"\nUSER CI: Detected change in user project: {file_path}")
                # User CI only logs, doesn't auto-run unless configured
                if self.config.get('auto_run_on_change', False):
                    self.agent.ci_cd.run_pipeline(
                        scope=[str(file_path)],
                        config_type="user",
                        trigger="on_change"
                    )

class SelfImprovingCodingAgent:
```

```
# ... existing code ...

def __init__(self,
             workspace: str = "coding_agent_workspace",
             user_project_dir: str = None,
             enable_ci_cd: bool = True):
    # ... existing initialization ...

    # CI/CD system
    self.ci_cd = None
    if enable_ci_cd:
        self.ci_cd = CI_CDSystem(self)

def run_user_ci_cd(self, scope: list = None):
    """Run user CI/CD pipeline on demand"""
    if not self.ci_cd:
        print("CI/CD not enabled")
        return

    # Default to entire user project
    if scope is None:
        scope = [str(p) for p in self.get_code_files(self.user_project_dir)]

    return self.ci_cd.run_pipeline(
        scope=scope,
        config_type="user",
        trigger="on_demand"
    )

def run_agent_ci_cd(self, scope: list = None):
    """Run agent CI/CD pipeline (usually automatic)"""
    if not self.ci_cd:
        print("CI/CD not enabled")
        return

    # Default to recent agent code
    if scope is None:
        scope = [
            str(p) for p in
            self.get_recent_files(self.generated_code_dir, count=5) +
            self.get_recent_files(self.self_improvement_dir, count=5)
        ]

    return self.ci_cd.run_pipeline(
            scope=scope,
            config_type="agent",
            trigger="on_change"  # Agent CI uses on_change even when manually triggered
        )

    def get_recent_files(self, directory: Path, count: int = 5) -> list:
        """Get most recently modified files in directory"""
        files = [f for f in directory.glob("*") if f.is_file()]
        return sorted(files, key=lambda f: f.stat().st_mtime, reverse=True)[:count]

    def auto_self_improve(self, scope: list = None):
        """Self-improve with CI/CD feedback - now scoped to specific files"""
        if scope is None:
            # Default to files that failed in last agent CI run
            scope = self.get_failed_files_from_last_ci_run()

        print(f"\nSelf-Improvement triggered for {len(scope)} files")
        # ... existing improvement logic, now focused on specific files ...

    def get_failed_files_from_last_ci_run(self) -> list:
        """Get files that failed in the last agent CI run"""
        # Find latest agent CI report
        agent_runs = sorted(self.ci_cd.agent_results_dir.glob("*"), reverse=True)
        if not agent_runs:
            return []

        latest_run = agent_runs[0]
        report_path = latest_run / "report.json"

        if not report_path.exists():
            return []

        with open(report_path, 'r') as f:
            report = json.load(f)

        failed_files = []
        for action, results in report["actions"].items():
            for result in results:
                if not result.get("success", True):
                    failed_files.append(result["path"])

        return list(set(failed_files))

# Example Usage
```

```python
if __name__ == "__main__":
    # Create agent with separated CI/CD
    agent = SelfImprovingCodingAgent(
        user_project_dir="user_project",
        enable_ci_cd=True
    )

    # User workflow
    print("\nUser working on project...")
    # User modifies a file - triggers CI logging but not execution
    with open("user_project/main.py", "a") as f:
        f.write("\n# User modification")

    # User explicitly requests CI/CD
    print("\nRunning user CI/CD on demand:")
    user_ci_results = agent.run_user_ci_cd()
    print(f"User CI results: {user_ci_results['actions']['run_tests'][0]['success']}")

    # Agent self-improvement workflow
    print("\nAgent adding self-improved code...")
    # Agent adds optimized code - triggers automatic CI/CD
    with open(agent.self_improvement_dir / "optimized.py", "w") as f:
        f.write("def optimized_function():\n    return 'improved'")

    # Wait for CI/CD to detect and run
    time.sleep(2)

    # Manually trigger agent CI/CD if needed
    print("\nManually checking agent codebase:")
    agent_ci_results = agent.run_agent_ci_cd()

    # Get CI/CD status
    print("\nCI/CD Status:")
    print(f"User CI results: {agent.ci_cd.user_results_dir}")
    print(f"Agent CI results: {agent.ci_cd.agent_results_dir}")
```

## Key Separation Features

### 1. Dual CI/CD Pipelines

Aspect	**User Project CI/CD**	**Agent Self-Improvement CI/CD**
Trigger	User request or configurable auto-run	Automatic on every self-improvement
Monitoring	User project directory	Agent's generated/improved code

Actions	Testing, security scans	Testing + auto-improvement
Scope	User-facing code	Internal agent codebase
Failure Response	Reports only	Triggers self-improvement
Configuration	User-specific settings	Agent-optimized defaults

### 2. Directory Structure
```
coding_agent_workspace/
├── ci_cd_config/
│   ├── user_config.yaml       # User CI settings
│   └── agent_config.yaml      # Agent CI settings
│
├── ci_cd_results/
│   ├── user/                  # User CI results
│   │   └── 20230815_142356/   # Timestamped run
│   │       └── report.json
│   │
│   └── agent/                 # Agent CI results
│       └── 20230815_142402/
│           └── report.json
│
├── generated_code/            # Initial agent code
├── self_improvement/          # Optimized agent code
└── user_project/              # User code (external)
```

### 3. Configuration Differences
user_config.yaml
```yaml
version: 1.0
monitor_paths:
  - /path/to/user_project
actions:
  on_change: []  # No auto-run by default
  on_demand: ['run_tests', 'static_analysis', 'security_scan']
reporting:
  email: user@example.com
  webhook: https://user-ci.example.com
```

agent_config.yaml
```yaml
version: 1.0
monitor_paths:
  - /path/to/generated_code
  - /path/to/self_improvement
actions:
  on_change: ['run_tests', 'static_analysis']  # Always run on change
  on_demand: ['coverage', 'security_scan']
auto_improve: true  # Automatically fix failures
```

### 4. Separate Event Handlers

```python
class AgentCIHandler(FileSystemEventHandler):
    """Triggers immediately on agent code changes"""
    def on_modified(self, event):
        # Runs full CI/CD and auto-improvement
        self.agent.ci_cd.run_pipeline(...)

class UserCIHandler(FileSystemEventHandler):
    """Passive monitoring for user projects"""
    def on_modified(self, event):
        # Only logs unless auto-run enabled
        if self.config.get('auto_run_on_change'):
            self.agent.ci_cd.run_pipeline(...)
```

### 5. Explicit Control Methods

```python
# User-initiated CI/CD
agent.run_user_ci_cd(scope=["user_project/main.py"])

# Agent-initiated CI/CD (mainly for debugging)
agent.run_agent_ci_cd()
```

### 6. Scoped Self-Improvement

```python
def auto_self_improve(self, scope: list = None):
    """Improve only specific files from CI/CD failures"""
    if scope is None:
        scope = self.get_failed_files_from_last_ci_run()
    # Focused improvement on problematic files
```

## Workflow Comparison

### User Project Workflow
```mermaid
sequenceDiagram
    participant User
    participant Agent
    participant CI_CD

    User->>Agent: Modifies code in user_project/
    CI_CD->>CI_CD: Logs change (no auto-run)
    User->>Agent: Requests CI/CD run
    Agent->>CI_CD: run_user_ci_cd()
    CI_CD->>CI_CD: Executes tests/scans
    CI_CD->>User: Returns report
```

### Agent Self-Improvement Workflow
```mermaid
sequenceDiagram
    participant Agent
    participant CI_CD

    Agent->>Agent: Saves improved code
    Agent->>CI_CD: Auto-triggers agent CI/CD
    CI_CD->>CI_CD: Runs tests/analysis
    alt Tests Pass
        CI_CD->>Agent: Success report
    else Tests Fail
        CI_CD->>Agent: Failure report
        Agent->>Agent: auto_self_improve(failed_files)
        Agent->>Agent: Generates new improvements
        Agent->>CI_CD: Triggers new CI/CD cycle
    end
```

## Benefits of Separation

1. **Clear Responsibility Boundaries**
   - User CI: Quality assurance for user projects
   - Agent CI: Quality control for self-improvement process

2. **Performance Optimization**
   - Agent CI runs frequently without affecting user projects
   - User CI runs only when explicitly requested

3. **Failure Containment**
   - Problems in self-improvement code don't affect user projects
   - User code changes don't trigger agent self-improvement

4. **Customized Workflows**
   - User CI: Focused on reporting and security
   - Agent CI: Optimized for rapid improvement cycles

5. **Audit Trails**
   - Separate logs for user vs agent activities
   - Clear history of self-improvement evolution

This design ensures the agent maintains a clear separation between:
- Its internal self-improvement process (continuous, automatic CI/CD)
- The user's development workflow (on-demand CI/CD)

The dual-path system prevents confusion while providing appropriate automation for each context, with the agent's self-improvement CI/CD tightly coupled to its optimization cycle and the user CI/CD available on-demand for project assurance.

I'll add comprehensive support for local LLMs through Ollama and LM Studio, giving the agent flexibility to use both cloud-based and local models. Here's the enhanced implementation:

```python
# llm_manager.py

import os
from langchain_community.llms import Ollama, HuggingFaceHub, OpenAI
from langchain_openai import ChatOpenAI
from huggingface_hub import InferenceClient
from openai import OpenAI as OpenAIClient
import requests
import json

class LLMManager:
    def __init__(self, config_path: str = "llm_config.json"):
        self.config_path = config_path
        self.config = self.load_config()
        self.available_models = self.discover_models()

    def load_config(self) -> dict:
        """Load LLM configuration from file"""
        default_config = {
            "default_provider": "ollama",
            "providers": {
                "ollama": {
                    "base_url": "http://localhost:11434",
                    "default_model": "llama3"
                },
                "lm_studio": {
                    "base_url": "http://localhost:1234/v1",
                    "default_model": "local-model"
                },
                "openai": {
                    "api_key": os.getenv("OPENAI_API_KEY"),
                    "default_model": "gpt-4-turbo"
                },
                "huggingface": {
                    "api_key": os.getenv("HUGGINGFACEHUB_API_TOKEN"),
                    "default_model": "google/flan-t5-xxl"
                }
            }
        }

        if os.path.exists(self.config_path):
            with open(self.config_path, "r") as f:
                return {**default_config, **json.load(f)}
        return default_config

    def save_config(self):
        """Save current configuration to file"""
        with open(self.config_path, "w") as f:
            json.dump(self.config, f, indent=2)

    def discover_models(self) -> dict:
        """Discover available models from all providers"""
        models = {"ollama": [], "lm_studio": [], "openai": [], "huggingface": []}

        # Discover Ollama models
        try:
            response = requests.get(f"{self.config['providers']['ollama']['base_url']}/api/tags")
            if response.status_code == 200:
                models["ollama"] = [model["name"] for model in response.json().get("models", [])]
        except:
            pass

        # Discover LM Studio models
        try:
            response = requests.get(f"{self.config['providers']['lm_studio']['base_url']}/models")
            if response.status_code == 200:
                models["lm_studio"] = [model["id"] for model in response.json().get("data", [])]
        except:
            pass

        # Get OpenAI models (requires API key)
        if self.config["providers"]["openai"]["api_key"]:
            try:
                client = OpenAIClient(api_key=self.config["providers"]["openai"]["api_key"])
                openai_models = client.models.list()
                models["openai"] = [model.id for model in openai_models.data]
            except:
                pass

        # Get Hugging Face models (requires API key)
        if self.config["providers"]["huggingface"]["api_key"]:
            try:
                response = requests.get(
                    "https://api-inference.huggingface.co/models",
                    headers={"Authorization": f"Bearer
{self.config['providers']['huggingface']['api_key']}"}
                )
                if response.status_code == 200:
                    models["huggingface"] = [model["modelId"] for model in response.json()]
            except:
                pass

        return models

    def get_llm(self, provider: str = None, model: str = None, **kwargs):
        """Get LLM instance for the specified provider and model"""
        provider = provider or self.config["default_provider"]
        provider_config = self.config["providers"][provider]

        if not model:
            model = provider_config["default_model"]

        print(f"Using {provider.upper()} model: {model}")

        if provider == "ollama":
            return Ollama(
                base_url=provider_config["base_url"],
                model=model,
                temperature=0.7,
                num_ctx=4096,
                **kwargs
            )

        elif provider == "lm_studio":
            return ChatOpenAI(
                base_url=provider_config["base_url"],
                model=model,
                temperature=0.7,
                max_tokens=2048,
                **kwargs
            )

        elif provider == "openai":
            return ChatOpenAI(
                api_key=provider_config["api_key"],
                model=model,
                temperature=0.7,
                **kwargs
            )

        elif provider == "huggingface":
            return HuggingFaceHub(
                repo_id=model,
                huggingfacehub_api_token=provider_config["api_key"],
                model_kwargs={
                    "temperature": 0.7,
                    "max_new_tokens": 512,
                    **kwargs
                }
            )

        raise ValueError(f"Unsupported provider: {provider}")

    def stream_response(self, provider: str, model: str, prompt: str, **kwargs):
        """Stream response from LLM for interactive use"""
        provider_config = self.config["providers"][provider]

        if provider == "ollama":
            response = requests.post(
                f"{provider_config['base_url']}/api/generate",
                json={
```

```
                "model": model,
                "prompt": prompt,
                "stream": True,
                **kwargs
            },
            stream=True
        )

        for line in response.iter_lines():
            if line:
                chunk = json.loads(line)
                if not chunk.get("done"):
                    yield chunk.get("response", "")

    elif provider == "lm_studio":
        client = OpenAIClient(base_url=provider_config["base_url"])
        stream = client.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            stream=True,
            **kwargs
        )

        for chunk in stream:
            if chunk.choices[0].delta.content:
                yield chunk.choices[0].delta.content

    elif provider == "openai":
        client = OpenAIClient(api_key=provider_config["api_key"])
        stream = client.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            stream=True,
            **kwargs
        )

        for chunk in stream:
            if chunk.choices[0].delta.content:
                yield chunk.choices[0].delta.content

    elif provider == "huggingface":
        client = InferenceClient(
            model=model,
            token=provider_config["api_key"]
            )

            for token in client.text_generation(prompt, stream=True, **kwargs):
                yield token

    def get_provider_config(self, provider: str) -> dict:
        """Get configuration for a specific provider"""
        return self.config["providers"].get(provider, {})

    def set_default_provider(self, provider: str):
        """Set the default LLM provider"""
        if provider in self.config["providers"]:
            self.config["default_provider"] = provider
            self.save_config()
        else:
            raise ValueError(f"Unknown provider: {provider}")

    def set_default_model(self, provider: str, model: str):
        """Set default model for a provider"""
        if provider in self.config["providers"] and model in self.available_models[provider]:
            self.config["providers"][provider]["default_model"] = model
            self.save_config()
        else:
            raise ValueError(f"Invalid provider or model: {provider}/{model}")

# Update ResearchAgent and CodingAgent to use LLMManager

class ResearchAgent:
    def __init__(self,
                 workspace: str = "research_workspace",
                 llm_manager: LLMManager = None,
                 llm_provider: str = None,
                 llm_model: str = None):
        # ... existing initialization ...

        # LLM setup
        self.llm_manager = llm_manager or LLMManager()
        self.llm = self.llm_manager.get_llm(
            provider=llm_provider,
            model=llm_model
        )

        # ... rest of initialization ...
    def query(self, question: str, stream: bool = False, **kwargs):
        """Query with streaming option"""
        if stream:
            return self.streaming_query(question, **kwargs)

        # Standard query implementation
        # ...

    def streaming_query(self, question: str, **kwargs):
        """Streaming response for research agent"""
        # Build full prompt with context
        full_prompt = self.build_full_prompt(question)

        # Get provider from current LLM
        provider = self.llm_manager.config["default_provider"]
        model = self.llm_manager.config["providers"][provider]["default_model"]

        return self.llm_manager.stream_response(
            provider=provider,
            model=model,
            prompt=full_prompt,
            **kwargs
        )

class SelfImprovingCodingAgent:
    def __init__(self,
                 workspace: str = "coding_agent_workspace",
                 llm_manager: LLMManager = None,
                 llm_provider: str = None,
                 llm_model: str = None):
        # ... existing initialization ...

        # LLM setup
        self.llm_manager = llm_manager or LLMManager()
        self.llm = self.llm_manager.get_llm(
            provider=llm_provider,
            model=llm_model
        )

        # ... rest of initialization ...

    def stream_code_generation(self, task: str, language: str):
        """Stream code generation process"""
        # Build prompt
        prompt = self.build_coding_prompt(task, language)

        # Get provider from current LLM
        provider = self.llm_manager.config["default_provider"]
        model = self.llm_manager.config["providers"][provider]["default_model"]

        return self.llm_manager.stream_response(
            provider=provider,
            model=model,
            prompt=prompt
        )

# Example usage
if __name__ == "__main__":
    # Initialize LLM manager
    llm_mgr = LLMManager()

    # Print available models
    print("Available Models:")
    for provider, models in llm_mgr.available_models.items():
        print(f"  {provider.upper()}:")
        for model in models[:3]:  # Show first 3 for brevity
            print(f"    - {model}")
        if len(models) > 3:
            print(f"    ... and {len(models)-3} more")

    # Configure to use local models
    llm_mgr.set_default_provider("ollama")
    llm_mgr.set_default_model("ollama", "llama3")

    # Create research agent with local LLM
    research_agent = ResearchAgent(
        workspace="research_workspace",
        llm_manager=llm_mgr
    )

    # Stream research response
    print("\nResearch Agent Streaming:")
    for chunk in research_agent.streaming_query("Explain quantum computing"):
        print(chunk, end="", flush=True)

    # Create coding agent with local LLM
    coding_agent = SelfImprovingCodingAgent(
        workspace="coding_agent_workspace",
        llm_manager=llm_mgr
    )

    # Stream code generation
    print("\n\nCoding Agent Streaming:")
    code_gen = ""
    for chunk in coding_agent.stream_code_generation(
        "Implement quicksort in Python", "python"
    ):
        print(chunk, end="", flush=True)
        code_gen += chunk

    # Save generated code
    with open("quicksort.py", "w") as f:
        f.write(code_gen)

    # Switch to LM Studio
    llm_mgr.set_default_provider("lm_studio")
    llm_mgr.set_default_model("lm_studio", "TheBloke/CodeLlama-13B-Instruct-GGUF")

    # Generate with different model
    print("\n\nGenerating with LM Studio:")
    for chunk in coding_agent.stream_code_generation(
        "Implement binary search in Rust", "rust"
    ):
        print(chunk, end="", flush=True)
```

## Key Local LLM Features

### 1. Unified LLM Manager
- **Multi-Provider Support**:
  - Ollama (local)
  - LM Studio (local)
  - OpenAI (cloud)
  - Hugging Face (local/cloud)

- **Dynamic Discovery**:
  ```python
  def discover_models(self) -> dict:
      # Queries each provider's API to list available models
      return {
          "ollama": ["llama3", "mistral", "codellama"],
          "lm_studio": ["TheBloke/CodeLlama-13B-Instruct-GGUF"],
  ```

```
        "openai": ["gpt-4-turbo", "gpt-3.5-turbo"],
        "huggingface": ["google/flan-t5-xxl", "mistralai/Mistral-7B-Instruct-v0.2"]
    }
```

### 2. Configuration Management
- **Persistent Config**:
  ```json
  {
    "default_provider": "ollama",
    "providers": {
      "ollama": {
        "base_url": "http://localhost:11434",
        "default_model": "llama3"
      },
      "lm_studio": {
        "base_url": "http://localhost:1234/v1",
        "default_model": "local-model"
      },
      "openai": {
        "api_key": "sk-...",
        "default_model": "gpt-4-turbo"
      },
      "huggingface": {
        "api_key": "hf_...",
        "default_model": "google/flan-t5-xxl"
      }
    }
  }
  ```

### 3. Streaming Support
- **Real-time Output**:
  ```python
  # Research agent streaming
  for chunk in research_agent.streaming_query("Explain transformers"):
      print(chunk, end="", flush=True)

  # Coding agent streaming
  for chunk in coding_agent.stream_code_generation("Python quicksort"):
      print(chunk, end="", flush=True)
  ```

### 4. Provider-Specific Implementations

Ollama Integration:
```python
# Generation
llm = Ollama(base_url="http://localhost:11434", model="llama3")

# Streaming
response = requests.post("http://localhost:11434/api/generate",
                         json={"model": "llama3", "prompt": "Hello", "stream": True})
for line in response.iter_lines():
    yield json.loads(line).get("response", "")
```

LM Studio Integration:
```python
# Generation (using OpenAI-compatible API)
llm = ChatOpenAI(base_url="http://localhost:1234/v1", model="local-model")

# Streaming
client = OpenAIClient(base_url="http://localhost:1234/v1")
stream = client.chat.completions.create(model="local-model", messages=[...], stream=True)
for chunk in stream:
    yield chunk.choices[0].delta.content
```

### 5. Agent Integration
- **Research Agent**:
  ```python
  class ResearchAgent:
      def __init__(self, llm_manager=None, llm_provider=None, llm_model=None):
          self.llm_manager = llm_manager or LLMManager()
          self.llm = self.llm_manager.get_llm(provider=llm_provider, model=llm_model)
  ```

- **Coding Agent**:
  ```python
  class SelfImprovingCodingAgent:
      def __init__(self, llm_manager=None, llm_provider=None, llm_model=None):
          self.llm_manager = llm_manager or LLMManager()
          self.llm = self.llm_manager.get_llm(provider=llm_provider, model=llm_model)
  ```

### 6. Model Switching
```python
```

```
# Switch between models at runtime
llm_mgr.set_default_provider("lm_studio")
llm_mgr.set_default_model("lm_studio", "TheBloke/Mistral-7B-Instruct-v0.2-GGUF")

# All subsequent requests use the new model
```

## Local Setup Instructions

### 1. Install Required Services
- **Ollama**:
  ```bash
  curl -fsSL https://ollama.com/install.sh | sh
  ollama pull llama3
  ollama pull codellama
  ```

- **LM Studio**:
  - Download from lmstudio.ai
  - Start local server in settings

### 2. Configure Environment
```bash
# For cloud services (optional)
export OPENAI_API_KEY="sk-..."
export HUGGINGFACEHUB_API_TOKEN="hf_..."
```

### 3. Run with Local Models
```python
# Use Ollama as default
llm_mgr = LLMManager()
llm_mgr.set_default_provider("ollama")
llm_mgr.set_default_model("ollama", "llama3")

# Create agents
research_agent = ResearchAgent(llm_manager=llm_mgr)
coding_agent = SelfImprovingCodingAgent(llm_manager=llm_mgr)

# Stream research response
for chunk in research_agent.streaming_query("Explain quantum entanglement"):
    print(chunk, end="", flush=True)
```

### 4. Hybrid Configuration
```json
// llm_config.json
{
  "default_provider": "ollama",
  "providers": {
    "ollama": {
      "default_model": "codellama",
      "base_url": "http://localhost:11434"
    },
    "lm_studio": {
      "default_model": "TheBloke/Phind-CodeLlama-34B-v2-GGUF",
      "base_url": "http://localhost:1234/v1"
    },
    "openai": {
      "default_model": "gpt-4-turbo",
      "api_key": "sk-... (for fallback)"
    }
  }
}
```

## Performance Considerations

1. **Hardware Requirements**:
   - 7B models: 8GB RAM
   - 13B models: 16GB RAM
   - 34B+ models: 32GB+ RAM

2. **Optimization Tips**:
   ```python
   # For GPU acceleration with Ollama
   llm = Ollama(model="llama3", num_gpu=50)  # % of GPU to use

   # For LM Studio quantization
   # Use 4-bit or 5-bit quantized GGUF models
   ```

3. **Model Selection Guide**:
Task	Recommended Model	Provider
Research	llama3-70b, Mixtral	Ollama
Coding	CodeLlama-34b, Phind-CodeLlama	LM Studio
Fast Iteration	Mistral-7b, llama3-8b	Ollama

Quality Fallback	GPT-4-Turbo	OpenAI

This implementation provides seamless integration with local LLMs while maintaining compatibility with cloud services. The agent can now leverage the best available models based on task requirements, hardware capabilities, and network conditions, with streaming support for real-time interaction.

# Agent Framework with Extensible Architecture

I'll create a comprehensive agent framework that supports multiple providers and allows easy creation of new agents. Here's the implementation:

```python
# agent_framework.py v0.1

import os
import json
import inspect
from typing import Type, Dict, Any, Callable, Optional
from pathlib import Path
from abc import ABC, abstractmethod

class LLMManager:
    """Central LLM manager with support for multiple providers"""
    def __init__(self, config_path: str = "llm_config.json"):
        self.config_path = config_path
        self.config = self.load_config()
        self.available_models = self.discover_models()

    def load_config(self) -> dict:
        """Load LLM configuration from file"""
        default_config = {
            "default_provider": "ollama",
            "providers": {
                "ollama": {"base_url": "http://localhost:11434", "default_model": "llama3"},
                "lm_studio": {"base_url": "http://localhost:1234/v1", "default_model": "local-model"},
                "openai": {"api_key": os.getenv("OPENAI_API_KEY"), "default_model": "gpt-4-turbo"},
                "huggingface": {"api_key": os.getenv("HUGGINGFACEHUB_API_TOKEN"),
"default_model": "google/flan-t5-xxl"}
            }
        }

        if Path(self.config_path).exists():
            with open(self.config_path, "r") as f:
                return {**default_config, **json.load(f)}
        return default_config

    def save_config(self):
        """Save current configuration to file"""
        with open(self.config_path, "w") as f:
            json.dump(self.config, f, indent=2)

    def discover_models(self) -> dict:
        """Discover available models from all providers"""
        models = {provider: [] for provider in self.config["providers"]}
        # Actual discovery logic would go here
        return models

    def get_llm(self, provider: str = None, model: str = None, **kwargs):
        """Get LLM instance for the specified provider and model"""
        provider = provider or self.config["default_provider"]
        provider_config = self.config["providers"][provider]
        model = model or provider_config["default_model"]

        print(f"Using {provider.upper()} model: {model}")
        # Actual LLM initialization would go here
        return None

    def stream_response(self, provider: str, model: str, prompt: str, **kwargs):
        """Stream response from LLM"""
        # Actual streaming implementation would go here
        yield "Streaming response..."

class BaseAgent(ABC):
    """Base class for all agents"""
    def __init__(self,
                 name: str,
                 llm_manager: LLMManager,
                 workspace: Path,
                 config: Optional[dict] = None):
        self.name = name
        self.llm_manager = llm_manager
        self.workspace = workspace
        self.config = config or {}
        self.llm = self.llm_manager.get_llm(
            provider=self.config.get("llm_provider"),
            model=self.config.get("llm_model")
        )
        self.setup_workspace()

    def setup_workspace(self):
        """Create agent-specific workspace"""
        self.workspace.mkdir(exist_ok=True, parents=True)
        print(f"{self.name} workspace: {self.workspace}")

    @abstractmethod
    def execute(self, task: str, **kwargs) -> Any:
        """Execute the agent's primary function"""
        pass

    def stream_response(self, prompt: str, **kwargs):
        """Stream response using agent's default provider"""
        provider = self.config.get("llm_provider", self.llm_manager.config["default_provider"])
        model = self.config.get("llm_model",
                               self.llm_manager.config["providers"][provider]["default_model"])

        return self.llm_manager.stream_response(
            provider=provider,
            model=model,
            prompt=prompt,
            **kwargs
        )

    def update_config(self, new_config: dict):
        """Update agent configuration"""
        self.config = {**self.config, **new_config}
        # Reinitialize LLM if provider/model changed
        if "llm_provider" in new_config or "llm_model" in new_config:
            self.llm = self.llm_manager.get_llm(
                provider=self.config.get("llm_provider"),
                model=self.config.get("llm_model")
            )

class ResearchAgent(BaseAgent):
    """Agent for research tasks with RAG capabilities"""
    def __init__(self,
                 llm_manager: LLMManager,
                 workspace: Path = Path("research_workspace"),
                 config: Optional[dict] = None):
        super().__init__("Research Agent", llm_manager, workspace, config)
        self.knowledge_base = workspace / "knowledge_base"
        self.setup_research_env()
    def setup_research_env(self):
        """Setup research-specific environment"""
        self.knowledge_base.mkdir(exist_ok=True)
        # Initialize vector store, etc.

    def execute(self, task: str, stream: bool = False, **kwargs):
        """Execute a research query"""
        if stream:
            return self.stream_response(f"Research: {task}")

        # Actual research implementation
        return f"Research result for: {task}"

class CodingAgent(BaseAgent):
    """Self-improving coding agent with CI/CD"""
    def __init__(self,
                llm_manager: LLMManager,
                workspace: Path = Path("coding_workspace"),
                config: Optional[dict] = None):
        super().__init__("Coding Agent", llm_manager, workspace, config)
        self.user_project_dir = Path(config.get("user_project_dir", "user_project"))
        self.setup_coding_env()

    def setup_coding_env(self):
        """Setup coding-specific environment"""
        self.user_project_dir.mkdir(exist_ok=True)
        # Initialize codebase, CI/CD, etc.

    def execute(self, task: str, language: str, **kwargs):
        """Generate code for a specific task"""
        # Actual code generation implementation
        return f"Generated {language} code for: {task}"

class AgentFactory:
    """Creates and manages agents"""
    def __init__(self, llm_manager: LLMManager, agent_registry: Optional[dict] = None):
        self.llm_manager = llm_manager
        self.agents = {}
        self.registry = agent_registry or self.default_registry()

    def default_registry(self) -> Dict[str, Type[BaseAgent]]:
        """Default registry of available agent types"""
        return {
            "research": ResearchAgent,
            "coding": CodingAgent
        }

    def register_agent(self, name: str, agent_class: Type[BaseAgent]):
        """Register a new agent type"""
        if not issubclass(agent_class, BaseAgent):
            raise ValueError("Agent must subclass BaseAgent")
        self.registry[name] = agent_class

    def create_agent(self,
                    agent_type: str,
                    workspace: Optional[Path] = None,
                    config: Optional[dict] = None) -> BaseAgent:
        """Create a new agent instance"""
        if agent_type not in self.registry:
            raise ValueError(f"Unknown agent type: {agent_type}")

        agent_class = self.registry[agent_type]

        # Set default workspace if not provided
        if workspace is None:
            workspace = Path(f"{agent_type}_workspace")

        # Create agent instance
        agent = agent_class(
            llm_manager=self.llm_manager,
            workspace=workspace,
            config=config or {}
        )

        # Store agent by name
        self.agents[agent.name] = agent
        return agent

    def get_agent(self, name: str) -> BaseAgent:
        """Get an existing agent by name"""
        if name not in self.agents:
            raise ValueError(f"Agent not found: {name}")
        return self.agents[name]

    def list_agents(self) -> list:
        """List all active agents"""
        return list(self.agents.keys())
```

```
class AgentPlugin:
    """Base class for agent plugins"""
    def __init__(self, name: str):
        self.name = name

    def pre_execute(self, agent: BaseAgent, task: str, **kwargs) -> dict:
        """Called before agent execution"""
        return kwargs

    def post_execute(self, agent: BaseAgent, task: str, result: Any, **kwargs):
        """Called after agent execution"""
        return result

class PluginManager:
    """Manages agent plugins"""
    def __init__(self):
        self.plugins = {}

    def register_plugin(self, plugin: AgentPlugin):
        """Register a new plugin"""
        self.plugins[plugin.name] = plugin

    def apply_pre_execute(self, agent: BaseAgent, task: str, **kwargs) -> dict:
        """Apply all pre-execute hooks"""
        modified_kwargs = kwargs
        for plugin in self.plugins.values():
            modified_kwargs = plugin.pre_execute(agent, task, **modified_kwargs)
        return modified_kwargs

    def apply_post_execute(self, agent: BaseAgent, task: str, result: Any, **kwargs) -> Any:
        """Apply all post-execute hooks"""
        modified_result = result
        for plugin in self.plugins.values():
            modified_result = plugin.post_execute(agent, task, modified_result, **kwargs)
        return modified_result

# Example custom agent
class DocumentationAgent(BaseAgent):
    """Agent for generating documentation"""
    def __init__(self,
             llm_manager: LLMManager,
             workspace: Path = Path("doc_workspace"),
             config: Optional[dict] = None):
```

```
        super().__init__("Documentation Agent", llm_manager, workspace, config)
        self.templates_dir = workspace / "templates"
        self.setup_doc_env()

    def setup_doc_env(self):
        """Setup documentation environment"""
        self.templates_dir.mkdir(exist_ok=True)

    def execute(self, task: str, format: str = "markdown", **kwargs):
        """Generate documentation"""
        return f"Documentation ({format}) for: {task}"

# Example plugin
class SecurityPlugin(AgentPlugin):
    """Security scanning plugin"""
    def __init__(self):
        super().__init__("security_scanner")

    def post_execute(self, agent: BaseAgent, task: str, result: Any, **kwargs):
        """Scan generated content for security issues"""
        if isinstance(agent, CodingAgent):
            print(f"Scanning code for security vulnerabilities...")
            # Actual security scanning would go here
            return result + "\n\nSecurity Scan: No issues found"
        return result

# Framework initialization
class AgentFramework:
    """Main framework entry point"""
    def __init__(self):
        self.llm_manager = LLMManager()
        self.agent_factory = AgentFactory(self.llm_manager)
        self.plugin_manager = PluginManager()

        # Register custom agent types
        self.agent_factory.register_agent("documentation", DocumentationAgent)

        # Register plugins
        self.plugin_manager.register_plugin(SecurityPlugin())

    def create_agent(self,
                agent_type: str,
                workspace: Optional[Path] = None,
                config: Optional[dict] = None) -> BaseAgent:
```

```
        """Create agent with plugin support"""
        agent = self.agent_factory.create_agent(agent_type, workspace, config)
        return agent

    def execute_agent(self,
                    agent_name: str,
                    task: str,
                    **kwargs) -> Any:
        """Execute agent task with plugin hooks"""
        agent = self.agent_factory.get_agent(agent_name)

        # Pre-execute hooks
        modified_kwargs = self.plugin_manager.apply_pre_execute(agent, task, **kwargs)

        # Execute agent
        result = agent.execute(task, **modified_kwargs)

        # Post-execute hooks
        return self.plugin_manager.apply_post_execute(agent, task, result, **kwargs)

# Example usage
if __name__ == "__main__":
    # Initialize framework
    framework = AgentFramework()

    # Create agents
    research_agent = framework.create_agent(
        "research",
        workspace=Path("my_research_workspace"),
        config={"llm_provider": "ollama", "llm_model": "llama3"}
    )

    coding_agent = framework.create_agent(
        "coding",
        config={
            "user_project_dir": "my_project",
            "llm_provider": "lm_studio",
            "llm_model": "TheBloke/CodeLlama-13B-Instruct-GGUF"
        }
    )

    doc_agent = framework.create_agent("documentation")

    # Execute agents
    research_result = framework.execute_agent(
        "Research Agent",
        "Explain quantum computing basics"
    )
    print(f"Research Result: {research_result}")

    # Stream coding agent response
    print("\nCoding Agent Streaming:")
    for chunk in coding_agent.stream_response("Implement quicksort in Python"):
        print(chunk, end="", flush=True)

    # Execute documentation agent
    doc_result = framework.execute_agent(
        "Documentation Agent",
        "Create API documentation for the quicksort function",
        format="html"
    )
    print(f"\n\nDocumentation Result: {doc_result}")

    # List all agents
    print("\nActive Agents:")
    for agent_name in framework.agent_factory.list_agents():
        print(f" - {agent_name}")
```

## Framework Architecture

### Core Components
1. **LLMManager**: Centralized LLM provider management
2. **BaseAgent**: Abstract base class for all agents
3. **AgentFactory**: Creates and manages agent instances
4. **PluginManager**: Handles pre/post-execution hooks
5. **AgentFramework**: Main entry point and coordinator

### Agent Creation Workflow
```mermaid
sequenceDiagram
    participant User
    participant Framework
    participant AgentFactory
    participant LLMManager

    User->>Framework: create_agent("research", config)
    Framework->>AgentFactory: create_agent("research", config)
    AgentFactory->>LLMManager: get_llm(config)
    LLMManager-->>AgentFactory: LLM instance
    AgentFactory->>ResearchAgent: new(workspace, llm, config)
    ResearchAgent-->>AgentFactory: Agent instance
    AgentFactory-->>Framework: Agent instance
    Framework-->>User: ResearchAgent
```

### Execution Flow with Plugins

```mermaid
sequenceDiagram
    participant User
    participant Framework
    participant PluginManager
    participant ResearchAgent

    User->>Framework: execute_agent("Research Agent", task)
    Framework->>PluginManager: apply_pre_execute(task)
    PluginManager-->>Framework: modified_kwargs
    Framework->>ResearchAgent: execute(task, modified_kwargs)
    ResearchAgent-->>Framework: result
    Framework->>PluginManager: apply_post_execute(result)
    PluginManager-->>Framework: modified_result
    Framework-->>User: final_result
```

## Extensibility Features

### 1. Creating New Agents

```python
class NewAgent(BaseAgent):
    def __init__(self, llm_manager, workspace, config):
        super().__init__("New Agent", llm_manager, workspace, config)
        # Custom initialization

    def execute(self, task: str, **kwargs):
        # Custom implementation
        return f"Result for: {task}"

# Register with framework
framework.agent_factory.register_agent("new", NewAgent)
```

### 2. Plugin System

```python
class AnalyticsPlugin(AgentPlugin):
    def __init__(self):
        super().__init__("analytics")

    def pre_execute(self, agent, task, **kwargs):
        print(f"Agent {agent.name} starting: {task}")
        return kwargs

    def post_execute(self, agent, task, result, **kwargs):
        print(f"Agent {agent.name} completed: {task}")
        return result

# Register plugin
framework.plugin_manager.register_plugin(AnalyticsPlugin())
```

### 3. Provider Management

```python
# Add new LLM provider
class CustomLLMProvider:
    def __init__(self, config):
        self.config = config

    def get_llm(self, model, **kwargs):
        # Custom implementation
        return CustomLLM(model)

    def stream_response(self, model, prompt, **kwargs):
        # Custom streaming
        yield "Streaming..."

# Extend LLMManager
class ExtendedLLMManager(LLMManager):
    def __init__(self, config_path):
        super().__init__(config_path)
        self.config["providers"]["custom"] = {"base_url": "http://custom-api"}

    def get_llm(self, provider=None, model=None, **kwargs):
        if provider == "custom":
            return CustomLLMProvider(self.config["providers"]["custom"]).get_llm(model)
        return super().get_llm(provider, model, **kwargs)

# Use with framework
```

```
framework.llm_manager = ExtendedLLMManager("llm_config.json")
```

### 4. Configuration Management

```python
# Create research agent with custom config
research_agent = framework.create_agent(
    "research",
    config={
        "llm_provider": "ollama",
        "llm_model": "mistral",
        "knowledge_base": "my_special_knowledge"
    }
)

# Update config at runtime
research_agent.update_config({
    "llm_model": "llama3",
    "max_results": 10
})
```

## Agent Collaboration Example

```python
# Collaborative task execution
def generate_documented_code(framework, task, language):
    # Execute coding agent
    code = framework.execute_agent(
        "Coding Agent",
        f"Implement {task} in {language}",
        language=language
    )

    # Execute documentation agent
    docs = framework.execute_agent(
        "Documentation Agent",
        f"Documentation for: {task}",
        code=code,
        format="markdown"
    )

    return {"code": code, "documentation": docs}
```

```
# Usage
result = generate_documented_code(framework, "binary search", "python")
print(f"Code:\n{result['code']}")
print(f"\nDocumentation:\n{result['documentation']}")
```

## Framework Benefits

1. **Unified Architecture**: Common interface for all agents
2. **Extensible Design**:
   - Add new agents with single class
   - Support new LLM providers
   - Extend functionality via plugins
3. **Centralized Configuration**:
   - LLM settings managed globally
   - Agent-specific configurations
4. **Collaboration Support**: Agents can work together seamlessly
5. **Runtime Flexibility**:
   - Switch LLM providers on the fly
   - Update agent configurations dynamically
6. **Plugin Ecosystem**:
   - Security scanning
   - Performance monitoring
   - Analytics collection
   - Cost tracking

## Example Use Cases

1. **Research & Development Pipeline**:
   ```python
   research_result = framework.execute_agent(
       "Research Agent",
       "Latest advancements in nanomaterials"
   )

   prototype_code = framework.execute_agent(
       "Coding Agent",
       "Simulation based on research",
       research=research_result,
       language="python"
   )

   framework.execute_agent(
       "Documentation Agent",
   ```

```
       "Research paper with implementation details",
       research=research_result,
       code=prototype_code
   )
   ```

2. **Automated Code Review**:
   ```python
   class CodeReviewPlugin(AgentPlugin):
       def post_execute(self, agent, task, result, **kwargs):
           if isinstance(agent, CodingAgent):
               review = framework.execute_agent(
                   "Review Agent",
                   "Code review for quality and security",
                   code=result
               )
               return f"{result}\n\nREVIEW:\n{review}"
           return result

   framework.plugin_manager.register_plugin(CodeReviewPlugin())
   ```

3. **Multi-Agent Workflow**:
   ```python
   def full_development_cycle(framework, requirement):
       # Research phase
       research = framework.execute_agent("Research Agent", requirement)

       # Design phase
       design = framework.execute_agent("Design Agent", research)

       # Implementation phase
       code = framework.execute_agent("Coding Agent", design)

       # Documentation phase
       docs = framework.execute_agent("Documentation Agent", code)

       # Deployment phase
       deployment = framework.execute_agent("Deployment Agent", code)

       return {
           "research": research,
           "design": design,
           "code": code,
   ```

~~~~
            "docs": docs,
            "deployment": deployment
        }
   ```
~~~~

This framework provides a robust foundation for building and managing AI agents with support for multiple LLM providers, easy extensibility, and powerful collaboration capabilities. The architecture enables seamless integration of new agents and providers while maintaining a consistent interface for execution and configuration.

# Agent Framework with Research and Coding Base Agents

I'll create a clean, extensible framework with ResearchAgent and CodingAgent as base agents, supporting multiple LLM providers and easy extension to other agent types.

```
#you    def _save_config(self):
        """Save configuration to file"""
        with open(self.config_path, 'w') as f:
            json.dump(self.config, f, indent=2)

class BaseProvider(ABC):
    """Base class for LLM providers"""
    def __init__(self, config: dict):
        self.config = config

    @abstractmethod
    def generate(self, prompt: str, model: str = None, **kwargs) -> str:
        """Generate text completion"""
        pass

    @abstractmethod
    def stream(self, prompt: str, model: str = None, **kwargs):
        """Stream text completion"""
        yield ""

class OllamaProvider(BaseProvider):
    """Provider for Ollama API"""
    def generate(self, prompt: str, model: str = None, **kwargs) -> str:
        model = model or self.config["default_model"]
        # Implementation would use ollama-python library
        return f"Ollama response to: {prompt}"
```

```
    def stream(self, prompt: str, model: str = None, **kwargs):
        model = model or self.config["default_model"]
        # Simulated streaming
        words = f"Streaming Ollama response to: {prompt}".split()
        for word in words:
            yield word + " "

class LMStudioProvider(BaseProvider):
    """Provider for LM Studio API"""
    def generate(self, prompt: str, model: str = None, **kwargs) -> str:
        model = model or self.config["default_model"]
        # Implementation would use OpenAI-compatible API
        return f"LM Studio response to: {prompt}"

    def stream(self, prompt: str, model: str = None, **kwargs):
        model = model or self.config["default_model"]
        # Simulated streaming
        words = f"Streaming LM Studio response to: {prompt}".split()
        for word in words:
            yield word + " "

class OpenAIProvider(BaseProvider):
    """Provider for OpenAI API"""
    def generate(self, prompt: str, model: str = None, **kwargs) -> str:
        model = model or self.config["default_model"]
        # Implementation would use openai library
        return f"OpenAI response to: {prompt}"

    def stream(self, prompt: str, model: str = None, **kwargs):
        model = model or self.config["default_model"]
        # Simulated streaming
        words = f"Streaming OpenAI response to: {prompt}".split()
        for word in words:
            yield word + " "

class HuggingFaceProvider(BaseProvider):
    """Provider for Hugging Face Inference API"""
    def generate(self, prompt: str, model: str = None, **kwargs) -> str:
        model = model or self.config["default_model"]
        # Implementation would use huggingface_hub
        return f"HuggingFace response to: {prompt}"

    def stream(self, prompt: str, model: str = None, **kwargs):
        model = model or self.config["default_model"]
        # Simulated streaming
        words = f"Streaming HuggingFace response to: {prompt}".split()
        for word in words:
            yield word + " "

class BaseAgent(ABC):
    """Abstract base class for all agents"""
    def __init__(self,
                 name: str,
                 llm_manager: LLMManager,
                 workspace: Path,
                 config: Optional[dict] = None):
        self.name = name
        self.llm_manager = llm_manager
By          self.workspace = workspace
        self.config = config or {}
        self.provider = self._get_provider()
        self._setup_workspace()

    def _get_provider(self) -> BaseProvider:
        """Get the LLM provider based on config"""
        provider_name = self.config.get("provider") or self.llm_manager.config["default_provider"]
        return self.llm_manager.get_provider(provider_name)

    def _setup_workspace(self):
        """Create agent workspace directory"""
        self.workspace.mkdir(parents=True, exist_ok=True)
        print(f"Created workspace for {self.name} at {self.workspace}")

    @abstractmethod
    def execute(self, task: str, **kwargs) -> Any:
        """Execute the agent's primary function"""
        pass

    def generate(self, prompt: str, model: str = None, **kwargs) -> str:
        """Generate response using agent's provider"""
        model = model or self.provider.config.get("default_model")
        return self.provider.generate(prompt, model, **kwargs)

    def stream(self, prompt: str, model: str = None, **kwargs):
        """Stream response using agent's provider"""
        model = model or self.provider.config.get("default_model")
        return self.provider.stream(prompt, model, **kwargs)
    def update_config(self, new_config: dict):
        """Update agent configuration"""
        self.config = {**self.config, **new_config}
        # Reinitialize provider if changed
        if "provider" in new_config:
            self.provider = self._get_provider()

class ResearchAgent(BaseAgent):
    """Base research agent with RAG capabilities"""
    def __init__(self,
                 llm_manager: LLMManager,
                 workspace: Path = Path("research_workspace"),
                 config: Optional[dict] = None):
        super().__init__("Research Agent", llm_manager, workspace, config)
        self.knowledge_base = workspace / "knowledge"
        self._setup_research_env()

    def _setup_research_env(self):
        """Setup research-specific environment"""
        self.knowledge_base.mkdir(exist_ok=True)
        # Initialize vector database, etc.
        print(f"Knowledge base at: {self.knowledge_base}")

    def execute(self, task: str, **kwargs) -> dict:
        """Execute a research query"""
        # In a real implementation, this would use RAG techniques
        response = self.generate(
            f"Research the following topic: {task}. "
            "Provide detailed information with sources."
        )

        return {
            "task": task,
            "response": response,
            "sources": ["Source 1", "Source 2"],
            "agent": self.name
        }

    def add_knowledge(self, content: str, source: str):
        """Add knowledge to the agent's knowledge base"""
        # Implementation would add to vector database
        print(f"Added knowledge from {source} to {self.name}")

class CodingAgent(BaseAgent):
```

````
    """Base coding agent with self-improvement capabilities"""
    def __init__(self,
                llm_manager: LLMManager,
                workspace: Path = Path("coding_workspace"),
                config: Optional[dict] = None):
        super().__init__("Coding Agent", llm_manager, workspace, config)
        self.code_dir = workspace / "code"
        self.improvements_dir = workspace / "improvements"
        self._setup_coding_env()

    def _setup_coding_env(self):
        """Setup coding-specific environment"""
        self.code_dir.mkdir(exist_ok=True)
        self.improvements_dir.mkdir(exist_ok=True)
        print(f"Code directory at: {self.code_dir}")
        print(f"Improvements directory at: {self.improvements_dir}")

    def execute(self, task: str, language: str, **kwargs) -> dict:
        """Generate code for a specific task"""
        # In a real implementation, this would use the agent's code generation capabilities
        response = self.generate(
            f"Write {language} code that accomplishes: {task}. "
            "Include comments and error handling."
        )

        return {
            "task": task,
            "language": language,
            "code": response,
            "agent": self.name
        }

    def self_improve(self, code: str, feedback: str) -> str:
        """Improve existing code based on feedback"""
        prompt = (
            f"Improve the following code based on feedback:\n\n"
            f"Code:\n```\n{code}\n```\n\n"
            f"Feedback: {feedback}\n\n"
            "Provide only the improved code:"
        )
        return self.generate(prompt)

class AgentRegistry:
    """Manages agent types and instances"""
````

```
    def __init__(self):
        self.agent_types: Dict[str, Type[BaseAgent]] = {}
        self.agent_instances: Dict[str, BaseAgent] = {}

    def register_agent_type(self, name: str, agent_class: Type[BaseAgent]):
        """Register a new agent type"""
        if not issubclass(agent_class, BaseAgent):
            raise ValueError("Agent must subclass BaseAgent")
        self.agent_types[name] = agent_class

    def create_agent(self,
                agent_type: str,
                llm_manager: LLMManager,
                name: str = None,
                workspace: Path = None,
                config: dict = None) -> BaseAgent:
        """Create a new agent instance"""
        if agent_type not in self.agent_types:
            raise ValueError(f"Unknown agent type: {agent_type}")

        agent_class = self.agent_types[agent_type]
        workspace = workspace or Path(f"{agent_type}_{len(self.agent_instances)}")
        name = name or f"{agent_type} Agent {len(self.agent_instances)+1}"

        agent = agent_class(llm_manager, workspace, config)
        self.agent_instances[agent.name] = agent
        return agent

    def get_agent(self, name: str) -> BaseAgent:
        """Get an agent instance by name"""
        return self.agent_instances.get(name)

    def list_agents(self) -> List[str]:
        """List all agent names"""
        return list(self.agent_instances.keys())

class AgentFramework:
    """Main framework entry point"""
    def __init__(self):
        self.llm_manager = LLMManager()
        self.registry = AgentRegistry()

        # Register base agents
        self.registry.register_agent_type("research", ResearchAgent)
        self.registry.register_agent_type("coding", CodingAgent)

    def add_agent_type(self, name: str, agent_class: Type[BaseAgent]):
        """Add a new agent type to the framework"""
        self.registry.register_agent_type(name, agent_class)

    def create_agent(self,
                agent_type: str,
                name: str = None,
                workspace: Path = None,
                config: dict = None) -> BaseAgent:
        """Create a new agent"""
        return self.registry.create_agent(
            agent_type,
            self.llm_manager,
            name,
            workspace,
            config
        )

    def execute_agent(self, agent_name: str, task: str, **kwargs) -> Any:
        """Execute an agent's task"""
        agent = self.registry.get_agent(agent_name)
        if not agent:
            raise ValueError(f"Agent not found: {agent_name}")
        return agent.execute(task, **kwargs)

    def list_agents(self) -> List[str]:
        """List all agent names"""
        return self.registry.list_agents()

# Example custom agent
class DocumentationAgent(BaseAgent):
    """Documentation agent that extends the research agent"""
    def __init__(self,
               llm_manager: LLMManager,
               workspace: Path = Path("doc_workspace"),
               config: Optional[dict] = None):
        super().__init__("Documentation Agent", llm_manager, workspace, config)
        self.templates_dir = workspace / "templates"
        self._setup_documentation_env()

    def _setup_documentation_env(self):
        """Setup documentation-specific environment"""
        self.templates_dir.mkdir(exist_ok=True)
        print(f"Templates directory at: {self.templates_dir}")

    def execute(self, task: str, format: str = "markdown", **kwargs) -> dict:
        """Generate documentation"""
        # In a real implementation, this would use templates and context
        response = self.generate(
            f"Create {format} documentation for: {task}"
        )

        return {
            "task": task,
            "format": format,
            "documentation": response,
            "agent": self.name
        }
```