Automatic Code Review Crew

A multi-agent AI system built with **CrewAI** that automatically reviews pull requests for code quality and security vulnerabilities. This application demonstrates advanced CrewAI features including memory, guardrails, and execution hooks.

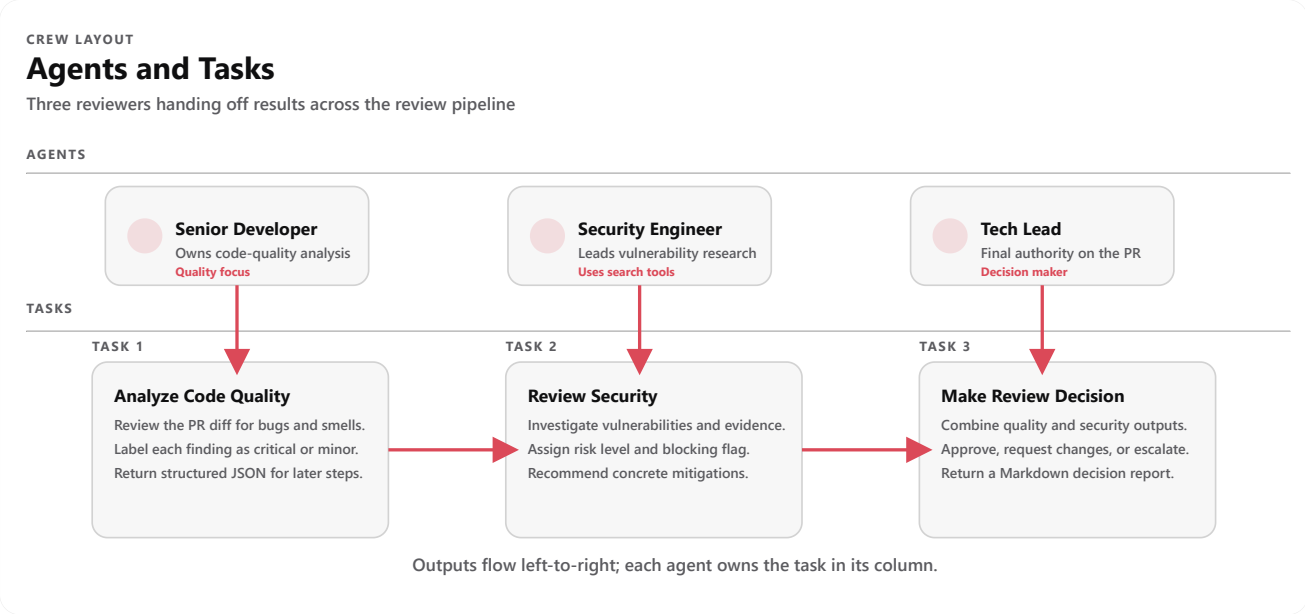
Table of Contents

Overview

The Automatic Code Review Crew is designed to automate the initial code review process in a CI/CD pipeline. It uses three specialized AI agents working collaboratively to:

- **Analyze code quality** - Identify bugs, style issues, and maintainability concerns
- **Review security** - Detect vulnerabilities and assess risk levels
- **Make decisions** - Approve changes, request fixes, or escalate to human reviewers

Agents and Tasks Diagram



CrewAI Framework Fundamentals

What is CrewAI?

CrewAI is an open-source Python framework for orchestrating autonomous AI agents. It provides a structured approach to creating teams of specialized agents that collaborate to solve complex tasks. The framework emphasizes:

- **Role-based coordination** - Agents are defined with specific roles, goals, and backstories
- **Task delegation** - Built-in mechanisms for assigning tasks based on agent capabilities
- **Agent collaboration** - Framework for inter-agent communication and knowledge sharing

Official Documentation: https://docs.crewai.com/

Core Components

CrewAI is built around six key pillars:

Component	Description
Roles	Define what each agent specializes in
Focus	Keep agents concentrated on their assigned tasks
Tools	Equipment for data retrieval, processing, and interaction
Cooperation	Enable agents to collaborate and delegate
Guardrails	Safety measures to ensure reliable outputs
Memory	Store and recall past interactions for better decisions

---

Agents

An **Agent** is an autonomous unit programmed to perform tasks, make decisions, and communicate with other agents. Think of an agent as a team member with specific skills and a particular job.

Agent Attributes

```
from crewai import Agent

agent = Agent(
    role="Data Analyst",                    # Job title/function
    goal="Extract actionable insights",     # What they aim to achieve
    backstory="You're an expert analyst...", # Experience and expertise
    verbose=True,                           # Show detailed output
    tools=[search_tool],                    # Available tools
    allow_delegation=False,                 # Can delegate to others
    max_iter=20,                           # Max iterations before answering
    max_rpm=10,                            # Rate limiting
)
```

Key Agent Parameters

Parameter	Type	Description
`role`	`str`	Defines the agent's function within the crew
`goal`	`str`	The individual objective the agent aims to achieve
`backstory`	`str`	Provides context for the agent's persona and expertise
`tools`	`list`	Set of tools the agent can use for tasks
`verbose`	`bool`	Enable detailed execution logs
`allow_delegation`	`bool`	Allow delegating tasks to other agents
`max_iter`	`int`	Maximum iterations before forcing a response
`max_rpm`	`int`	Maximum requests per minute (rate limiting)

Tasks

A **Task** represents a specific assignment to be completed by an agent. Tasks include detailed instructions and define what successful completion looks like.

Task Attributes

```
from crewai import Task

task = Task(
```

```
    description="Analyze the code for security issues...",
    expected_output="A JSON report with vulnerabilities...",
    agent=security_agent,
    output_json=SecurityReportSchema,    # Pydantic model for structured output
    guardrails=[validation_function],     # Output validation
    context=[previous_task],              # Results from other tasks
    markdown=True,                        # Output in Markdown format
)
```

Key Task Parameters

Parameter	Type	Description
`description`	`str`	Clear explanation of what the task involves
`expected_output`	`str`	Description of the expected result format
`agent`	`Agent`	The agent responsible for this task
`output_json`	`BaseModel`	Pydantic model for structured JSON output
`guardrails`	`list`	Validation functions for output
`context`	`list[Task]`	Previous tasks whose outputs provide context
`markdown`	`bool`	Format output as Markdown
`async_execution`	`bool`	Run task asynchronously

Crews

A **Crew** is a collaborative group of agents working together to achieve a set of tasks. It orchestrates the workflow, manages agent collaboration, and handles execution.

Crew Attributes

```
from crewai import Crew

crew = Crew(
    agents=[agent1, agent2, agent3],
    tasks=[task1, task2, task3],
    memory=True,                          # Enable memory
    verbose=True,
    process=Process.sequential,           # Task execution order
    before_kickoff_callbacks=[read_hook],  # Pre-execution hooks
    after_kickoff_callbacks=[save_hook],   # Post-execution hooks
)

# Execute the crew
result = crew.kickoff(inputs={"topic": "AI"})
```

Key Crew Parameters

Parameter	Type	Description
`agents`	`list[Agent]`	List of agents in the crew
`tasks`	`list[Task]`	List of tasks to be executed
`memory`	`bool`	Enable memory for the crew
`process`	`Process`	Execution process type (sequential/hierarchical)
`before_kickoff_callbacks`	`list`	Functions to run before execution
`after_kickoff_callbacks`	`list`	Functions to run after execution
`max_rpm`	`int`	Global rate limit for the crew

Kickoff Methods

```
# Standard execution
result = crew.kickoff(inputs={"topic": "AI"})

# Execute for multiple inputs
results = crew.kickoff_for_each(inputs=[
    {"topic": "AI"},
    {"topic": "ML"}
])

# Asynchronous execution
result = await crew.kickoff_async(inputs={"topic": "AI"})
```

Tools

Tools are capabilities that agents can use to interact with external services, databases, or APIs.

Built-in Tools

CrewAI provides several built-in tools through `crewai_tools`:

```
from crewai_tools import SerperDevTool, ScrapeWebsiteTool

# Web search tool
search_tool = SerperDevTool(search_url="https://owasp.org")

# Website scraping tool
scrape_tool = ScrapeWebsiteTool()

# Assign to agent
agent = Agent(
```

```
    role="Researcher",
    tools=[search_tool, scrape_tool],
    ...
)
```

Common Built-in Tools

Tool	Description
`SerperDevTool`	Search the web via Serper API
`ScrapeWebsiteTool`	Scrape content from websites
`FileReadTool`	Read files from the filesystem
`DirectoryReadTool`	Read directory contents
`PDFSearchTool`	Search within PDF documents
`WebsiteSearchTool`	Search specific websites

Memory

Memory allows agents to remember and learn from past interactions, improving decision-making over time.

Memory Types

Type	Description
Short-term Memory	Stores information from the current execution
Long-term Memory	Retains data across multiple executions
Entity Memory	Remembers key entities encountered
Contextual Memory	Maintains context between interactions

Enabling Memory

```
crew = Crew(
    agents=[...],
    tasks=[...],
    memory=True,  # Enable memory
)
```

When memory is enabled, agents can:

- Remember previously identified patterns
- Recognize recurring issues
- Build upon past experiences

- Make more consistent decisions

Guardrails

Guardrails validate and transform task outputs before they're passed to the next task, ensuring data quality and reliability.

Types of Guardrails

1. **Function-based Guardrails**: Python functions with custom validation logic
2. **LLM-based Guardrails**: String descriptions for natural language validation

Function-based Guardrail Example

```
from typing import Tuple, Any

def validate_output(output) -> Tuple[bool, Any]:
    """
    Guardrail function to validate task output.

    Returns:
        (True, validated_result) on success
        (False, "Error message") on failure
    """
    try:
        json_output = output.json_dict

        # Validate required fields
        if 'required_field' not in json_output:
            return (False, "Missing required field")

        # Validation passed
        return (True, json_output)

    except Exception as e:
        return (False, f"Validation error: {str(e)}")

# Apply to task
task = Task(
    description="...",
    guardrails=[validate_output],
    ...
)
```

Multiple Guardrails

```
task = Task(
    description="...",
```

```
    guardrails=[
        validate_word_count,    # Function-based
        validate_format,        # Function-based
        "Must be professional"  # LLM-based
    ],
    guardrail_max_retries=3,   # Retry on failure
    ...
)
```

Execution Hooks

Execution Hooks are callbacks that run before or after crew execution, allowing you to inject custom logic.

Before Kickoff Hook

Runs before agents start their work. Useful for preprocessing inputs.

```
def read_file_hook(inputs: dict) -> dict:
    """Read a file and add contents to inputs."""
    filename = inputs.get("file_path")

    with open(filename, "r") as f:
        inputs["file_content"] = f.read()

    return inputs

crew = Crew(
    agents=[...],
    tasks=[...],
    before_kickoff_callbacks=[read_file_hook],
)
```

After Kickoff Hook

Runs after all tasks complete. Useful for saving results or cleanup.

```
def save_result_hook(result):
    """Save the final result to a file."""
    if hasattr(result, 'tasks_output'):
        content = result.tasks_output[-1].raw

        with open("output.md", "w") as f:
            f.write(content)

    return result

crew = Crew(
    agents=[...],
```

```
    tasks=[...],
    after_kickoff_callbacks=[save_result_hook],
)
```

Process Types

CrewAI supports different execution processes:

Sequential Process

Tasks are executed one after another in order. Each task's output can be used as context for the next.

```
crew = Crew(
    agents=[agent1, agent2],
    tasks=[task1, task2],
    process=Process.sequential,  # Default
)
```

Hierarchical Process

A manager agent coordinates the crew, delegating tasks and validating outcomes.

```
from crewai import Process

crew = Crew(
    agents=[agent1, agent2],
    tasks=[task1, task2],
    process=Process.hierarchical,
    manager_llm=ChatOpenAI(model="gpt-4"),  # Required
)
```

Application Architecture

The pipeline mirrors the visual in the Agents and Tasks Diagram: three agents own one task each, handing results left-to-right. In order:

- **Senior Developer** → **Analyze Code Quality** produces structured findings (critical/minor) from the PR diff.
- **Security Engineer** → **Review Security** investigates vulnerabilities, assigns risk levels, and marks blocking vs. non-blocking.
- **Tech Lead** → **Make Review Decision** combines both outputs to approve, request changes, or escalate, returning the final Markdown report.

Memory remains enabled across runs for pattern recognition, and guardrails validate security outputs and the final decision format.

OWASP Top 10 Coverage

The `review_security` task now maps findings to the OWASP Top 10 (A01–A10). Its output includes an `owasp_top_10` list where each entry has `id` (A01-A10), `name`, `status` (`not_observed`, `needs_attention`, `confirmed`), and supporting `evidence` or a `not_applicable` reason. Each vulnerability in `security_vulnerabilities` should also reference the relevant OWASP category.

Recommendation Report Task

The `write_recommendation_report` task produces a Markdown summary at report/recommendations.md. It pulls from Analyze Code Quality, Review Security (including OWASP mapping), and Make Review Decision to list critical/minor issues, security risks, the final decision, and prioritized fixes.

Installation

Prerequisites

- Python 3.10 or higher
- OpenAI API key
- Serper API key (optional, for web search)

Setup

1. **Clone or download the project:**

```
cd automatic_code_review_crew
```

2. **Create and activate a virtual environment:**

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install dependencies:**

```
pip install crewai[tools]==1.3.0
pip install pydantic pyyaml
```

4. **Set environment variables:**

```
export OPENAI_API_KEY="your-openai-api-key"
export SERPER_API_KEY="your-serper-api-key"  # Optional
```

```
export MODEL="gpt-4o-mini"                    # Optional, defaults to gpt-4o-mini
```

Configuration

Agent Configuration (`config/agents.yaml`)

Define agents with their roles, goals, and backstories:

```
senior_developer:
  role: Senior Developer
  goal: Evaluate code changes for quality issues
  backstory: You're an experienced software engineer...
  allow_delegation: false
```

Task Configuration (`config/tasks.yaml`)

Define tasks with descriptions and expected outputs:

```
analyze_code_quality:
  description: >
    1. Review the code changes in the pull request.
    2. Identify style issues, bugs, or maintainability concerns.
    PR content: {file_content}
  expected_output: >
    A JSON with keys: 'critical_issues', 'minor_issues', 'reasoning'
  name: Analyze Code Quality
```

Usage

Running the Application

1. **Place your PR diff in `code_changes.txt`:**

```
diff --git a/app/auth.py b/app/auth.py
--- a/app/auth.py
+++ b/app/auth.py
@@ -1,5 +1,10 @@
 def login(username, password):
+    # Vulnerable to SQL injection
     user = db.query(f"SELECT * FROM users WHERE name='{username}'")
```

2. **Run the application:**

```
python main.py
```

3. **View the output:**

The crew will analyze the code and provide a decision:

- ☑ **Approve** - Code meets quality standards
- 🔄 **Request Changes** - Issues need to be fixed
- ⚠ **Escalate** - Human review required

Programmatic Usage

```python
from main import AutomaticCodeReviewCrew

# Initialize the crew
crew = AutomaticCodeReviewCrew(verbose=True)

# Run a review
result = crew.review("path/to/code_changes.txt")

# Access the final decision
print(result.tasks_output[-1].raw)
```

Additional Libraries

This application uses the following Python libraries:

Core Dependencies

Library	Version	Purpose
`crewai[tools]`	1.3.0	Multi-agent AI framework
`pydantic`	2.x	Data validation using Python type hints
`pyyaml`	6.x	YAML configuration file parsing

CrewAI Tools

Tool	Purpose
`SerperDevTool`	Web search via Serper API for security research
`ScrapeWebsiteTool`	Extract content from websites (e.g., OWASP)

Installation Command

```
```

```
pip install "crewai[tools]==1.3.0" pydantic pyyaml
```

Project Structure

```
automatic_code_review_crew/
├── main.py                    # Main application entry point
├── config/
│   ├── agents.yaml           # Agent configurations
│   └── tasks.yaml            # Task configurations
├── code_changes.txt          # Sample PR diff for testing
├── report/
│   └── recommendations.md    # Generated recommendation report (Markdown)
└── README.md                 # This documentation
```

File Descriptions

File	Description
`main.py`	Complete Python application with crew, agents, tasks, guardrails, and hooks
`config/agents.yaml`	YAML configuration defining agent roles, goals, and backstories
`config/tasks.yaml`	YAML configuration defining task descriptions and expected outputs
`code_changes.txt`	Sample pull request diff file for testing
`README.md`	Comprehensive documentation

Implementation Guide

Step 1: Set Up Environment

```
# Create virtual environment
python -m venv venv
source venv/bin/activate

# Install dependencies
pip install "crewai[tools]==1.3.0" pydantic pyyaml

# Set API keys
export OPENAI_API_KEY="sk-..."
export SERPER_API_KEY="..."  # Optional
```

Step 2: Configure Agents and Tasks

Edit `config/agents.yaml` and `config/tasks.yaml` to customize:

- Agent roles and expertise
- Task descriptions and expected outputs

Step 3: Add Your Code Changes

Replace `code_changes.txt` with your actual PR diff:

```
git diff main..feature-branch > code_changes.txt
```

Step 4: Run the Review

```
python main.py
```

Step 5: Interpret Results

The final decision will be one of:

1. **Approve** - No blocking issues found
2. **Request Changes** - Issues must be fixed before merge
3. **Escalate** - Complex issues require human judgment

References

- CrewAI Documentation
- CrewAI GitHub Repository
- Pydantic Documentation
- OWASP Security Guidelines

License

This project is provided for educational purposes. See the CrewAI license for framework usage terms.