Agent-Flavored Markdown: Natural Language Specifications for Framework-Agnostic Agent Development

Maryam Ziyad Mohamed[1,*], Hasitha Aravinda[1] and Rania Khalaf[1]

[1]WSO2

Abstract

Building AI agents today requires either expertise in framework-specific APIs and abstractions, or reliance on low-code platforms that couple agent logic with execution configuration. In both cases, the agent's core behavioral instructions — expressed in natural language — become embedded within code or tool interfaces, while framework-specific abstractions limit portability. We present Agent-Flavored Markdown (AFM), a natural language-first specification for defining portable AI agents. AFM treats agent instructions as the primary content, expressed in human-readable markdown, with configuration such as tools, model configuration, and interfaces specified as structured YAML front matter. This separation of concerns keeps agent behavior readable even as technical complexity grows, making it easier for both developers and domain experts to author and collaborate while enabling the same agent definition to work with different frameworks through tools such as interpreters or code generators. We describe the AFM specification and demonstrate its feasibility and portability with reference implementations.

Keywords

AI agents, No-code agent builders, Agent portability

1. Introduction

The use of Generative AI has quickly moved from simply integrating calls to large language models (LLMs) to building AI agents [1] that can reason, use tools, manage state, and operate autonomously. Such agents are becoming a central part of modern applications. While agents initially started in question-and-answer scenarios, they have since evolved to take actions – such as handling the full life cycle of customer care issues, including identifying and acting upon resolutions.

Agent development experiences range from pro-code, where the developer writes all the code, to low-code and no-code tools. In addition, one can use AI-powered techniques to generate the code representing an agent from a natural language description. Each agent framework or platform introduces its own abstractions for agents. Developers must invest considerable time learning framework-specific patterns, often writing or generating hundreds of lines of boilerplate code to express relatively simple agent behaviors. Approaches that start with code (code-first) are highly expressive, but often make it hard to see the most important aspect of an agent: its instructions. These are the natural language descriptions that define the agent's objective and how it is expected to behave. Low-code and no-code development platforms simplify this, yet the agent instructions and execution configuration (e.g., configuring the model and tools) are often tightly coupled. Finally, additional effort is required to implement how the agent is triggered or exposed (e.g., running on a trigger, exposed as an API, via chat, etc.).

Teams are often also concerned about framework lock-in. As the agent ecosystem is still evolving, they may need to switch agent frameworks as new capabilities emerge or as their organizations require it. Such a move is costly both in time and effort.

AI-powered code generation tools could simplify both initial development and migration, but some manual effort is still required. The tightly-coupled nature of instructions and configuration also means

Joint Proceedings of the ACM Intelligent User Interfaces (IUI) Workshops 2026, March 23-26, 2026, Paphos, Cyprus

*Corresponding author.

maryamm@wso2.com (M. Z. Mohamed)

that even with AI-powered tools, agent authors have to deal with having to specify configuration well beyond the agent's instructions.

We propose that agent development can feel more natural when beginning with natural language instructions rather than code and/or framework-specific configuration. In this paper, we present Agent-Flavored Markdown (AFM) [1], a specification for a no-code approach to define portable agents. AFM allows developers and domain experts to define agents primarily through markdown-formatted instructions, with YAML front matter for configuration. AFM adopts a natural language-first approach: the agent's role and instructions form the core of its declaration. Metadata and configuration – including interface definitions that specify how the agent is triggered and exposed – are minimal, yet sufficient. These characteristics, we believe, provide a simpler development experience and portable agent definitions.

Central to AFM's design is a clear separation of concerns: natural language instructions express agent behavior in markdown, while technical configuration such as tool bindings, model settings, authentication, and interfaces is specified in structured YAML front matter. This approach ensures that even agents with advanced technical requirements remain human-readable - the agent's role and instructions are immediately apparent from the markdown content, while the configuration-based approach for technical implementation details is comprehensive to allow advanced configuration. As technical requirements grow (authentication protocols, tool integration, trigger integration, etc.), the agent's core purpose remains clear. We believe this improves user experience and accessibility over existing specification-based approaches, where agent logic and technical configuration are intermingled within the same structural format. Moreover, since AFM definitions are framework-agnostic, the same AFM file can be used to work with different agent frameworks, enabling portability with little to no rework. For more complicated cases, AFM could facilitate collaboration between technical and non-technical users, allowing each to work independently on their area of responsibility (e.g., defining agent behavior vs configuring tools and interfaces).

Our contributions are:

- A specification that clearly separates agent instructions (expressed in natural language markdown) and technical configuration (specified in structured YAML), enabling agent declarations to remain readable and maintainable even as technical complexity increases
- A framework-agnostic approach that enables an agent to be defined once and used with different agent frameworks and platforms through interpreters, code generators, etc.
- Reference implementations demonstrating AFM's feasibility and practical benefits

Several approaches that make use of specifications have emerged to address different aspects of agent standardization, each targeting different use cases and design priorities. We discuss these related efforts in detail in Section 2.

The remainder of this paper is organized as follows: Section 2 discusses related work in agent frameworks and specification languages. Section 3 presents the AFM specification in detail. Section 4 demonstrates how AFM agents work and presents reference implementations. Section 5 discusses the implications and limitations of our approach before we conclude in Section 6.

2. Related Work

Specifications for AI agents and configuration-driven approaches to develop AI agents have received growing attention. These different approaches address different aspects with goals that range from discoverability and interoperability to portability.

We have also seen the emergence of markdown with front matter being used to define agents, though not as a generic approach for agent authoring as we are proposing.

[1]Specification available at: https://wso2.github.io/agent-flavored-markdown/specification/

2.1. Agent Specification Languages

Agent Spec [2] is a specification language for defining portable agentic systems, including agentic workflows. Developers use the Python software development kit (SDK) to programmatically define agents and serialize them to Agent Spec format (JSON or YAML), with system prompts appearing as strings within these structures. Agent Spec resolves variables (e.g., environment variables) at specification generation time rather than runtime. The resulting specification can be used across frameworks through language-specific adapters. Agent Spec's approach is fundamentally code-first, with developers defining agents programmatically. Since the specification is interpreted programmatically, it uses constructs such as component references with generated identifiers that a natural language-first approach would not use. As such, it could be cumbersome for a user to directly author the specification rather than writing the code and having the specification generated.

NeMo Agent Toolkit (NAT) [3] is a configuration-driven approach to define agents, with support for interoperability across frameworks. NAT has its own runtime but provides adapters that allow agents or tools from different frameworks (e.g., LangChain [4], CrewAI [5]) to be used as tools within NAT agents. Like Agent Spec, NAT uses YAML for configuration, with system prompts embedded as multi-line strings. NAT simplifies interface setup through the `nat` CLI tool, including the `nat serve` command that launches a web server to trigger agent workflows on incoming requests. However, the agent definition remains configuration-centric, with instructions specified within the YAML structure.

2.2. Markdown and Front matter-based Agentic Definitions

Claude Code [6] and GitHub Copilot [7], both popular AI coding agents, use a file format consisting of markdown with front matter (similar to AFM) to define sub-agents [8] or custom agents [9]. Anthropic's Agent Skills [10], an open standard that enables new capabilities and expertise for agents, also uses a similar format.

2.3. Framework-Native Configuration Approaches

Some agent frameworks have their own configuration formats that improve agent code organization and readability. CrewAI [5], for example, allows developers to define agent roles and instructions in YAML files separate from application code, which the framework loads to instantiate agents. This approach improves maintainability by separating agent definitions from implementation logic, but the structure remains framework-specific and uses YAML's structured format.

2.4. Agent Discovery and Interoperability Standards

The Open Agentic Schema Framework (OASF) [11] addresses discovery for inter-agent interactions. OASF defines a schema to describe agent capabilities, domains, and endpoints in machine-readable formats, enabling agents to advertise themselves in a directory and discover each other. Although OASF's primary goal differs from AFM's emphasis on portability and authoring, there is some overlap in defining a specification to specify agent metadata.

2.5. Positioning AFM

AFM addresses the challenge of portability for AI agents, but takes a fundamentally different approach to the authoring experience. Instead of a code- or configuration-driven approach, AFM uses markdown — a format familiar to developers and accessible to non-developers. Agent instructions are specified in markdown prose, with configuration and metadata (tools, triggers, model, etc.) in lightweight YAML front matter. This approach reflects how domain experts and developers typically conceptualize agents: as entities defined primarily by their natural language instructions, with technical configuration as supporting but necessary details. Moreover, the adoption of a similar format by popular AI coding agents demonstrates its practical viability.

AFM also supports defining interfaces that specify how an agent is triggered and the expected structure for inputs and outputs. For many use cases, an AFM file can represent the entire executable program — the agent along with its invocation interfaces. In such cases, AFM also provides end-to-end portability: the same definition can be deployed on different platforms without requiring additional integration code.

Agent discovery is also a goal for AFM. For example, generating Agent-to-Agent (A2A) [12] cards based on the AFM specification could enable discovery out of the box while maintaining AFM's markdown-first authoring experience.

3. The AFM Specification

Agent-Flavored Markdown (AFM) is a markdown-based format for defining AI agents in a natural language-first manner. AFM files use the `.afm` or `.afm.md` extension and consist of two main sections: YAML front matter for configuration and metadata, and markdown for the agent's role and instructions. The complete formal AFM specification is available at https://wso2.github.io/agent-flavored-markdown/specification/.

3.1. Design Principles

The following design principles guide the AFM design:

Instructions as Primary Content: The agent's natural language instructions are the primary component of an AFM file, written as standard markdown prose. Authors can use markdown's formatting capabilities (e.g., headings, lists, code blocks) to organize instructions naturally. This makes the file readable and editable by both developers and domain experts, without requiring deep technical knowledge, and enables collaboration between all kinds of users: agents and both technical and non-technical humans.

Minimal, Secure, Framework-Agnostic Configuration: AFM configuration is intentionally minimal, capturing only essential configuration expected in different frameworks and platforms. By focusing on portable abstractions rather than framework-specific features, AFM definitions can be used with different agent platforms without modification. AFM's variable substitution capability (e.g., from environment variables) enables dynamic configuration and secure handling of sensitive data, facilitating portability across environments.

Human-Readable Specifications: Agent instructions are written in markdown, while configuration details (tools, models, interfaces, etc.) are specified in lightweight YAML front matter. This separation of concerns keeps the specification human-readable even as technical complexity grows, and provides structure that visual editors can leverage for improved editing, rendering, and navigation.

3.2. File Structure

An AFM file consists of two sections:

YAML Front matter (Optional): Enclosed by `---` delimiters at the top of the file, containing metadata and configuration for the agent.

Markdown Body (Required): Contains two mandatory sections defined by markdown headings:

- `# Role`: Defines the agent's purpose and responsibilities
- `# Instructions`: Provides detailed behavioral directives and operational guidelines

Here is a minimal AFM file: [2]

```
---
name: "Friendly Assistant"
interfaces:
```

[2]Additional examples available at: https://wso2.github.io/agent-flavored-markdown/examples/

```
  - type: "webchat"
---
# Role

You are a friendly and helpful conversational assistant. Your purpose is to engage in natural,
    helpful conversations with users, answering their questions, providing information, and
    assisting with various tasks to the best of your abilities.

# Instructions

- Always respond in a friendly and conversational tone
- Keep your responses clear, concise, and easy to understand
- If you don't know something, be honest about it
- Ask clarifying questions when the user's request is ambiguous
```

3.3. Front Matter Schema

The front matter schema defines agent configuration and metadata including:

Agent Details: Identification and attribution metadata including `name`, `description`, `version`, `author(s)`, `provider`, `icon_url`, and `license`.

Tools: The tools available to the agent. Currently, AFM supports Model Context Protocol (MCP) [13] tools, with plans to extend support to functions as tools and tools based on OpenAPI specifications. Each MCP tool configuration specifies attributes such as:

- a transport mechanism (e.g., `http` for streamable HTTP)
- transport-specific properties such as URL (if the transport is `http`) and authentication if required by the MCP server
- optional tool filtering

Model Configuration: LLM configuration such as provider, name, URL, and authentication.

Interfaces: One or more interfaces specifying how the agent can be invoked. AFM currently defines three interface types:

- `consolechat`: Command-line/terminal chat interface (default if not specified)
- `webchat`: Web-based chat interface with HTTP endpoint configuration
- `webhook`: Event-driven interface with subscription and user prompt templating support

An interface specifies an input/output signature using JSON Schema expressed in YAML. This enables both simple string-based signatures (default) and more complex structures. Implementations can use the signature for validation.

Execution Configuration: Runtime controls such as `max_iterations` to prevent runaway execution.

3.4. Variable Substitution

AFM supports a variable substitution syntax (`${...}`) that allows referencing sensitive or dynamic values, facilitating portability while maintaining security.

The specification defines three standard prefixes:

- `env:` — Environment variables (e.g., `${env:GITHUB_TOKEN}`)
- `http:payload.` — Webhook payload fields (e.g., `${http:payload.event}`)
- `http:header.` — Webhook HTTP request headers (e.g., `${http:header.User-Agent}`)

Implementations may support additional prefixes (e.g., `file:`, `secret:`) for extended functionality.

3.5. Illustrative Example: GitHub Pull Request Analyzer

To demonstrate AFM's capabilities, we present a GitHub pull request analyzer agent that detects drift between code and documentation (see Appendix A for the complete definition). This agent is attached to a webhook subscribed to receive notifications on changes related to pull requests. When a notification is received, the agent determines whether the pull request was opened, reopened, or edited, and if so, identifies changes and relevant documentation, detects drift, and posts an analysis report as a comment.

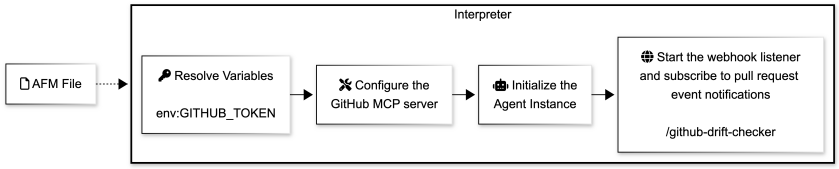

Figure 1: AFM agent initialization

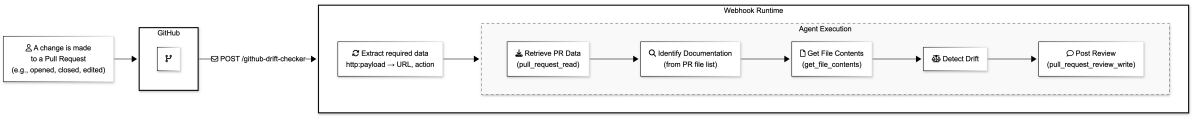

Figure 2: AFM agent runtime for webhook events

Figure 1 demonstrates the initialization phase: parsing the AFM file, resolving variables, configuring GitHub MCP tools, and exposing the webhook interface.

Figure 2 demonstrates the runtime phase: receiving a pull request event, substituting payload variables, and executing the agent's analysis workflow.

This webhook-based agent showcases several key features of AFM:

- Natural language instructions expressing complex agent behavior using markdown formatting
- Interface definition specifying a webhook trigger and prompt templating with variable substitution from payload data (e.g., `${http:payload.pull_request.url}`)
- MCP tool integration with filtered access to specific GitHub API operations
- Environment variable-based configuration enabling the same definition to work with different frameworks and/or environments, without compromising security and flexibility

The agent's complex instructions are expressed in markdown. Technical concerns such as webhook subscriptions, authentication, and tool integrations are specified as front matter, with the interpreter handling the implementation details. Despite this technical complexity, the agent's core behavioral logic remains immediately clear from the markdown instructions. This separation of concerns — natural language markdown for agent behavior and structured YAML front matter for technical configuration — is central to AFM's design.

4. Using AFM: Implementation Approaches

An AFM file serves as a platform-agnostic agent definition that can be executed across languages and frameworks using various implementation strategies. In this section, we describe approaches to execute AFM agents and present reference implementations.

4.1. Implementation Approaches

There are three primary approaches to make AFM definitions executable:

- Interpreter-based execution: A standalone interpreter reads an AFM file, extracts the agent's instructions and configuration, and dynamically creates an agent and exposes it via the specified interface(s)
- Library function: A library function that accepts an AFM file as input, initializes the agent, and attaches it to the specified interface(s) – functionally similar to an interpreter, but requires explicit invocation in application code
- Code generator: A code generator translates an AFM file into framework-specific code

With the first approach, the AFM agent becomes the entire program, whereas the other two approaches allow AFM to serve as a high-level agent authoring interface while leveraging the full capabilities of target languages and frameworks.

All approaches validate AFM's portability: the same agent definition can be executed through different implementation strategies and target different frameworks and runtime environments.

4.2. Reference Implementations

We have developed reference interpreter implementations [3] for AFM based on the Ballerina[4] programming language and the LangChain [4] framework, made available as Docker images[5] [6]. The interpreters accept an AFM file and dynamically instantiate the agent and expose it via the specified interface(s).

These interpreters work as follows:

1. Parse the AFM File: Extract YAML front matter and markdown body, validating against the schema.
2. Create the Agent Instance: Resolve variables and instantiate the AI agent using the configuration from the front matter, with the extracted instructions as the system prompt.
3. Expose via Interfaces: Based on the interface specification(s), expose the agent as:
    - Console chat (interactive CLI)
    - Web chat (HTTP endpoint with a web chat UI)
    - Webhook (WebSub [14] subscriber with an HTTP callback endpoint)

For webhook interfaces, runtime variables such as `${http:payload.field_name}` in the prompt template are resolved for each incoming request, enabling dynamic prompt construction from webhook data.

4.3. Portability in Practice

Our interpreter implementations demonstrate several aspects of AFM's portability:

Framework Independence: While the reference implementations use the language or framework's AI and integration capabilities, the AFM specification itself is not tied to any particular language or framework. Interpreters or code generators targeting different frameworks can parse the same AFM file and produce functionally equivalent agents within their environments, as demonstrated via the two interpreter implementations for two frameworks/programming languages.

End-to-End Portability: AFM files can specify not only agent behavior but also interface definitions (how the agent is triggered and exposed). For standalone agents where the agent and its interfaces constitute the entire application, this enables end-to-end portability — the same AFM file can be used across languages or frameworks without requiring additional integration code. This shifts the implementation burden from agent authors to the developers of AFM interpreters or code generators, which becomes a one-time effort per framework rather than per agent.

[3] https://github.com/wso2/reference-implementations-afm
[4] https://ballerina.io/
[5] https://github.com/wso2/reference-implementations-afm/pkgs/container/afm-ballerina-interpreter
[6] https://github.com/wso2/reference-implementations-afm/pkgs/container/afm-langchain-interpreter

4.4. Toward Multi-Framework Support

To fully realize AFM's vision for portability, we envision implementations targeting more frameworks. For example, implementations could include code generators or interpreters (similar to the reference implementations) that set up agents for frameworks such as LlamaIndex [15] or CrewAI [5].

This allows AFM to serve as a lingua franca of agent definitions. It separates the concerns of agent authoring: what the agent does and how it is implemented, enabling users to author agents once and use them on different platforms.

5. Discussion and Limitations

AFM prioritizes no-code agent authoring, natural language readability, and cross-platform portability over fine-grained control and using framework-specific features. In this section, we discuss the implications of this design, acknowledge its limitations, and outline directions for future work.

5.1. Design Trade-offs

Scope: Standalone Agents vs Workflows: The term "workflows" is used differently across agent frameworks. In some contexts, it refers to orchestrated control flow with multiple LLM calls or agent invocations, where one could depend on the other's outcome. In other contexts, it refers to single-agent execution patterns like ReACT [16] or ReWOO [17]. AFM is more relevant to the latter: defining individual agents that can employ different approaches to reasoning and tool use. AFM does not provide constructs for predetermined control flow between multiple LLM calls or agent instances.

Natural Language vs Programmatic Control: AFM's goal of being natural language-first improves readability but reduces room for precise programmatic control flow possible with code- or configuration-first approaches. There is no support for loops or conditional logic prior to agent execution. The agent's conditional behavior is guided by natural language directives interpreted by the LLM used, rather than deterministic code paths. This is appropriate for many agent scenarios where flexible, context-dependent behavior is desired but may be limiting for applications requiring strict control and may result in LLM calls that could have been avoided in code-first approaches. For instance, in the GitHub pull request analyzer example, running the analysis only if the pull request was opened, reopened, or edited is left to the LLM, whereas in a code-first approach, this would be checked for before the agent execution. However, a code generation approach could still facilitate this for AFM, since the generated code could be modified to implement the necessary checks before the agent is run.

Simplicity vs Expressiveness: AFM intentionally maintains a minimal expression syntax. Variable substitution supports specific patterns for environment variable or payload/header access but is not a general-purpose expression language. For example, expressions like `${env:PORT + 1000}` or `${http:payload.items.length() > 0}` are not supported. This may impact dynamic configuration capabilities or flows, but keeps the expression syntax simple and AFM files easy to read and write.

5.2. When AFM is a Good Fit

AFM is well-suited for agents where:

- Agent behavior is primarily instruction-driven rather than procedurally defined or heavily configuration-driven
- Natural language readability is valued
- Portability across platforms is important

AFM may be less appropriate when the following are required:

- Strict control flow around the agent or with variables within the agent flow
- Complex conditional logic or data transformations that cannot be entirely represented by tools

- Tight integration with framework-specific features that cannot be abstracted

In practice, AFM and code-first approaches can be complementary. With the code generation and function-based approaches described previously, AFM can define the agent's core instructions and configuration, while framework-specific code can handle complex control flow or incorporate framework-specific features, without affecting the portability of the agent core.

5.3. Future Work

Agent Interaction Protocols: The current AFM specification focuses on defining individual agents. While agents can interact by exposing one agent as a tool to another, AFM does not yet support standardized agent-to-agent communication protocols. We believe agent discovery and inter-agent communication are natural extensions of AFM's design. By generating A2A cards or similar discovery metadata from AFM specifications, agents could advertise their capabilities and be discovered by other agents or systems. This would enable AFM agents to interact in broader multi-agent environments.

Implementations for More Frameworks: While our reference implementations demonstrate AFM's feasibility, implementing code generators, library functions, and/or interpreters for AFM files for other popular frameworks would provide stronger evidence of portability and reveal any framework-specific challenges in the specification.

User Studies: We plan to conduct user studies comparing AFM to existing code- and configuration-first approaches, in terms of time to author, readability, and maintenance effort. We also plan to gather feedback from both technical and non-technical users on AFM's expressiveness, learnability, and scalability as complexity grows.

Other Specification Extensions: Several extensions to the AFM specification are under consideration:

- **Agent types**: Supporting agent types that represent how the agent reasons and uses tools - e.g., covering patterns such as ReACT and ReWOO
- **Non-MCP tools**: Supporting tools beyond MCP, including functions as tools and tools extracted directly from OpenAPI specifications available for services. For now, these can be supported by exposing them as MCP tools first.
- **Memory specification**: Currently AFM does not prescribe a standard for agent memory (how agents retain information during conversations), given the lack of standardization in the area. Future versions of the specification may define common memory patterns to prevent the adoption of framework-specific memory abstractions that could impact portability.
- **More interface types**: Supporting more interface types such as REST APIs

Tooling and Developer Experience: To improve the AFM authoring experience, we plan to explore a web-based editor or an IDE plugin to write AFM files with syntax highlighting, completions, and validation capabilities.

6. Conclusion

We have presented Agent-Flavored Markdown (AFM), a natural language-first specification for defining AI agents that prioritizes human readability and enables cross-platform portability. AFM treats natural language instructions as the primary content of agent definitions, with structured front matter for configuration. This approach makes agent development more accessible to both developers and domain experts.

Our key contributions include: (1) a specification that clearly separates natural language instructions from technical configuration, keeping agent behavior human-readable even as technical complexity grows, (2) a formalization of agent configuration and interfaces that enables framework-agnostic portability, and (3) reference implementations demonstrating AFM's feasibility and practical utility.

These are made available at https://wso2.github.io/agent-flavored-markdown and https://github.com/orgs/wso2/packages?repo_name=reference-implementations-afm.

AFM takes a different approach to agent development tools, optimizing for authoring ease and portability rather than fine-grained control. As the agent ecosystem continues to evolve rapidly, specifications like AFM that decouple what the agent does from implementation details become increasingly valuable.

Future work includes formal user studies, implementations targeting more languages and frameworks, updates to the specification including support for memory and non-MCP tools, and tooling support for enhanced authoring experiences. We invite the community to explore AFM, provide feedback, and contribute to its evolution as an open standard.

7. Appendices

A. Complete GitHub Pull Request Analyzer Example

This appendix presents the complete AFM definition for the GitHub pull request analyzer agent referenced in the Illustrative Example: GitHub Pull Request Analyzer section.

```
---
spec_version: "0.3.0"
name: "GitHub PR Code-Documentation Drift Checker"
description: "Analyzes GitHub pull requests and identifies drift between code and documentation"
version: "0.1.0"
max_iterations: 30
interfaces:
  - type: "webhook"
    prompt: "Analyze the following pull request ${http:payload.pull_request.url} if the action
        performed is one of opened, reopened, or edited. Action performed - ${http:payload.
        action}"
    subscription:
      protocol: "websub"
      hub: "https://api.github.com/hub"
      topic: "${env:GITHUB_REPO_TOPIC}"
      callback: "${env:CALLBACK_URL}/github-drift-checker"
      secret: "${env:GITHUB_WEBHOOK_SECRET}"
    exposure:
      http:
        path: "/github-drift-checker"
tools:
  mcp:
    - name: "github"
      transport:
        type: "http"
        url: "https://api.githubcopilot.com/mcp/"
        authentication:
          type: "bearer"
          token: "${env:GITHUB_TOKEN}"
      tool_filter:
        allow:
          - "pull_request_read"
          - "get_file_contents"
          - "pull_request_review_write"
---

# Role

You are a GitHub PR review bot that detects drift between code, requirements, and documentation.
```

```
 When given a URL to a pull request, you analyze the changes, detect drift, and post
    detected drift in a comment.

# Instructions

Follow these steps in order:

## 1. Search to Build Documentation List

Extract owner and repository from the provided PR URL, then search the repository to discover
    what documentation exists:

- Search for markdown files: `repo:OWNER/REPO extension:md`
- Search for README files: `repo:OWNER/REPO filename:README`
- Search for requirements: `repo:OWNER/REPO (requirements OR spec OR design)`
- Create a written list of files found (this is your source of truth)
- If searches return empty results, note "No documentation found"

## 2. Retrieve PR Data and Baseline Documentation

**Get the PR Data:**

- Retrieve complete PR data including PR number, source/target branches, and changed files with
    their diffs
- The PR diff contains all code changes - you have everything you need from this
- Parse the diffs to understand what was added, modified, or deleted

**Retrieve Baseline Documentation:**
- Only retrieve files from your documented list in step 1
- Always specify the target branch when retrieving
- These are the baseline to compare PR changes against
- Never retrieve files not in your step 1 list

## 3. Drift Detection

**Categorize Changed Files:**
- Code files: `.bal`, `.java`, `.py`, `.ts`, `.js`, etc.
- Documentation files: `.md`, `.adoc`, README files
- Requirements/specification files describing features, APIs, or design

**Analyze Using Available Data:**
Check both directions:
- **Code → Docs**: Do code changes match requirements? Are changes documented?
- **Docs → Code**: Are all documented features actually implemented?

**Detect Drift:**
Read documentation carefully before flagging issues. Only flag drift if:
- After reading all documentation, a feature is genuinely not mentioned anywhere
- Code contradicts what technical specifications explicitly state
- A new public API is added with no documentation at any level
- Code introduces changes not reflected in any documentation

Do NOT flag drift if:
- The endpoint/feature is described in documentation (even at high level)
- Technical spec shows API contract matching the code
- Documentation mentions the capability the code implements

## 4. Post Review Comment
```

```
Post a comprehensive comment on the GitHub PR with:
- Clear description of each drift issue found
- Severity level (Critical/Major/Minor)
- What changed and what's missing
- Action required to fix
- File references and approximate locations

If no drift detected, post: "No drift detected between code and documentation."
```

This example demonstrates how AFM handles complex, real-world agent scenarios while maintaining readability and portability.

Acknowledgments

We thank Sanjiva Weerawarana, Shafreen Anfar, and Malith Jayasinghe for their input and feedback on the design and positioning of AFM.

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude Sonnet 4.5 in order to: Paraphrase and reword and Improve writing style. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

The AFM specification design was human-driven, with all design decisions made by the authors.

References

[1] V. Dibia, Designing Multi-Agent Systems: Principles, Patterns, and Implementation for AI Agents, 2025.

[2] S. Amini, Y. Benajiba, C. Bernardis, P. Cayet, H. Chafi, A. Fathan, L. Faucon, D. Hilloulin, S. Hong, I. Kossyk, T. M. S. Le, R. Patra, S. Ravi, J. Schweizer, J. Singh, S. Singh, W. Sun, K. Talamadupula, J. Xu, Open Agent Specification (Agent Spec): A Unified Representation for AI Agents, 2025. URL: http://arxiv.org/abs/2510.04173. doi:10.48550/arXiv.2510.04173, arXiv:2510.04173 [cs].

[3] NVIDIA, NVIDIA NeMo Agent Toolkit Overview — NVIDIA NeMo Agent Toolkit (1.2), 2025. URL: https://docs.nvidia.com/nemo/agent-toolkit/1.2/index.html.

[4] LangChain, 2025. URL: https://docs.langchain.com/oss/python/langchain/overview.

[5] CrewAI, 2025. URL: https://github.com/crewAIInc/crewAI.

[6] Claude Code - AI coding agent for terminal & IDE | Claude, 2025. URL: https://claude.com/product/claude-code.

[7] GitHub Copilot · Your AI pair programmer, 2025. URL: https://github.com/features/copilot.

[8] Build with Claude Code: Create custom subagents, 2025. URL: https://code.claude.com/docs/en/sub-agents.

[9] GitHub Copilot Custom Agents Configuration, 2025. URL: https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/create-custom-agents.

[10] Agent Skills, 2026. URL: https://agentskills.io/specification.

[11] Open Agentic Schema Framework, 2025. URL: https://schema.oasf.outshift.com/.

[12] Agent2Agent (A2A) Protocol, 2025. URL: https://a2a-protocol.org/latest/.

[13] Model Context Protocol, 2025. URL: https://modelcontextprotocol.io/docs/getting-started/intro.

[14] WebSub, 2018. URL: https://www.w3.org/TR/websub/.

[15] LlamaIndex - Redefine Document Workflows with AI Agents, 2025. URL: https://www.llamaindex.ai/.

[16] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, ReAct: Synergizing Reasoning and Acting in Language Models, 2023. URL: http://arxiv.org/abs/2210.03629. doi:10.48550/arXiv.2210.03629, arXiv:2210.03629.

[17] B. Xu, Z. Peng, B. Lei, S. Mukherjee, Y. Liu, D. Xu, ReWOO: Decoupling Reasoning from Observations for Efficient Augmented Language Models, 2023. URL: http://arxiv.org/abs/2305.18323. doi:10.48550/arXiv.2305.18323, arXiv:2305.18323.