Agencia: A Prompt-Centric Platform for Agentic Programming

1. Introduction

An agentic programming model is just an AI callback function that happens to call AI. After studying AutoGen, LangGraph, and CrewAI, the biggest problem is that they hide the prompt. When you look at agents in any of these systems, they use a function to compose the prompt. You can't see the prompt unless you use an unnamed function. All you can see is the agent's name, description, and which function is called.

I wanted a more declarative approach where you can see the prompt and it's composition. There's nothing better than Go templates for this purpose. We use a go-template that composes the prompt for the agent. Go templates are Turing complete, so we have all the power we need to compose the prompt. And if the template evaluates to blank, it skips the agent, allowing flow control.

There are two primary template calls you can make inside a prompt: Input and Get. Input is the input from the user. Get is a call to another agent, replacing its results into the prompt. Input is the user input replaced in the prompt.

In Agencia, there is only one functional element: the agent. You can call other agents in two ways: directly using Get inside the template, or indirectly using listeners. A list of agents, called listeners, that behave like functions (or tools). They must have a description so that AI can recognize the pattern and call them.

The benefit of this approach is that it is an entirely declarative style for building agentic systems. An agent behaves as a pure function with side effects only to save (or memoize) facts. And yes, it can even describe monads.

2. Agents

Here is a simple agent:

```
agents:
  greet:
    description: Greet the user
    prompt: |
      Say hello to {{ .Input }}.
```

This agent calls AI on the result of the prompt template. There is also a simpler form of agent that does not call AI. It simply evaluates the template and returns the result. This is a template agent.

```
agents:
  greet:
    description: Greet the user
    template: |
      Hello, {{ .Input }}!
```

Notice the only difference is that the prompt keyword becomes a template. You cannot use both at once. This is a case where we don't need AI to determine how to say hello. A template agent answers the greeting more directly. But that doesn't mean that a template can't call an agent.

```
agents:
  greeting:
    description: Greet the user
    prompt: |
      Say hello to {{ .Input }}.
  intro:
    Description: Introduce our service to the user
    template: |
      {{ Get "greeting" }}
      Welcome to our service!
```

The intro agent uses Get to call the greeting agent, which calls AI to generate the greeting. Of course, a template agent cannot use listeners because AI is required to call them.

A third agent type is an alias agent. This is a simple agent that just calls another agent.

```
agents:
  greeting:
    description: Greet the user
    prompt: |
      Say hello to {{ .Input }}.
  greet:
    description: Greet the pilot
    input_prompt:
      callsign:
        description: The pilot's call sign
    alias: "greeting"
```

An alias agent allows changing the input_prompt, facts, and description of another agent. This can be important if you want the same agent behavior but in a different situation.

There are a few more features of agents that we will cover. But this is all you need to understand to see the simplicity of using agents.

3. Structured vs Unstructured Input

When AI is able to call a function, that is the unstructured AI/User world calling the structured functional world. So we need a way to convert between the two worlds. The lowest level agent is a function agent, which calls a function from the coding library. Like templates, these functions return a string. However, the inputs to the function are structured, so we need a way to define the arguments.

An agent can take an input_prompt, which is a map of names and descriptions of what goes into that value. This is just like the arguments for any AI function (aka tool). This is required for function agents and listeners because they both take structured arguments.

But you can define an input_prompt for any agent, even a template agent. This calls AI on the Input as needed to generate the arguments for the agent before it is called. Once we have those inputs, the template or prompt can reference those values using Args for all the arguments or Arg to get a specific argument.

```yaml
agents:
  greet:
    description: Greet the user
    input_prompt:
      name:
        description: The name of the person to greet
    prompt: |
      Say hello to {{ .Arg "name" }}.
```

If the user wrote something like: "Hi, my name is Mary and I love warm hugs" Just calling Input would return the entire string. But calling Arg "name" would return "Mary". This intelligent deconstruction is useful even when not calling an external function.

A prompt and a template are string-to-string pure functions. So the structure produced by the input_prompt is not passed. Instead, it is for use in the template or prompt.

4. Agent Libraries

Function agents must be declared in code. These can be organized into a library of agents. Libraries can be used in other libraries or agents. The library name references the agent using dot syntax: 'libname.agentname'. For example, you might use a time library agent like this: 'time.current'. Below is an example

```yaml
agents:
  greet:
    description: Greet the user
    input_prompt:
      name:
        description: The name of the person to greet
    prompt: |
      Welcome, it is {{ .Get "time.current"}}
      Say hello to {{ .Arg "name" }}.
```

The Go code convention used for Agencia is to declare a package variable called Agents. This is a map of agent names to function agents. The function agent takes a context arg and a map of structured input and returns a string.

Context is passed down the Go calling tree, allowing access to other configuration objects stored in the context. But if you do that, these are no longer pure functions.

5. Remembering Facts in Chat

When using Agencia chat, a session object keeps a set of structured facts stored by the agents. Facts are declared on the agent using the facts keyword. Facts are similar to input_prompt, in that they are descriptions of values filled in by AI. But in this case when you declare a fact, it is stored in the Chat object by agent name. So to reference a fact in another template, you use the agent name as the key. For example, if you have an agent called 'greet' that declares a fact called 'name', you can refer to it in another template using {{ .Fact "greet.name" }}.

```yaml
agents:
  greet:
    description: Greet the user
    facts:
      name:
        description: The name of the person to greet
    prompt: |
      Say hello to {{ .Input }}.

  intro:
    description: Introduce our service to the user
    template: |
      {{ .Get "greet" }}
      {{ .Fact "greet.name" }}
      Welcome to our service!
```

Once an agent stores the fact, it can be accessed by any agent in the chat and is saved for the next time you use the same chat ID.

The input prompt is filled in by AI using the user input only. But the facts are filled in using both the input and the result of the template or prompt. So you can't use your own facts in the input prompt or template. However, you can reference your own facts, if what you want is the prior value.

6. Procedures

An agent can also declare a procedure, which is a list of agents to call in order, and keeps all the outputs from prior agents as the context for future agents. Below is an example:

```
agents:
  checkout:
    description: Checkout a library book
    input_prompt:
      title:
        description: The title of the book to check out
    procedure:
      - check_book_availability
      - get_library_card
      - check_out_book
    template: |
      Checking out book: {{ .Arg "title" }}
      You will be notified when it is done.

  intro:
    description: Introduce our service to the user
    template: |
      {{ .Get "checkout" }}
      Welcome to our service!
```

Procedures are asynchronous; they run in the background and notify the user with their result. However, they also return a message immediately, informing the user that the job has started.

The agents in a procedure may save ephemeral facts, which are only available to the procedure. This is done using the scope keyword on a fact. Scope defaults to global, which is how we described facts above. The local scope is saved in an ephemeral context used only inside the procedure.

```
agents:
  checkout:
    description: Checkout a library book
    facts:
      library:
        description: The library where the book resides.
        scope: local
    input_prompt:
      title:
        description: The title of the book to check out
    procedure:
      - check_book_availability
      - get_library_card
      - check_out_book
    template: |
      Checking out book: {{ .Arg "title" }}
      Job ID: {{ .JobID }}
      You will be notified when it is done.

  intro:
    description: Introduce our service to the user
    template: |
      {{ .Get "checkout" }}
      Welcome to our service!
```

A procedure agent uses the template or prompt to return the starting message to the user. If no prompt or template is provided, then the procedure returns a standard message: Running procedure: and .

The user can cancel, pause, or ask about the status of the job. If they have forgotten the JobID, They can ask about the status of all jobs or refer to them by the procedure name.

7. Using Agencia

Agencia is a web service you can find here: https://agencia.dev

The code is open-source and resides here: https://github.com/robbyriverside/agencia

8. License

MIT

9. Contact

Rob Farrow robbyriverside@gmail.com