Template

- Basic Template Structure
- Adding templates to your model
- Variables
- Tips and Best Practices
- Examples
  - Example Messages
    - ChatML
  - Example Tools
    - Mistral
  - Example Fill-in-Middle
    - CodeLlama
    - Codestral

Ollama provides a powerful templating engine backed by Go's built-in templating engine to construct prompts for your large language model. This feature is a valuable tool to get the most out of your models.

Basic Template Structure

A basic Go template consists of three main parts:

- **Layout**: The overall structure of the template.
- **Variables**: Placeholders for dynamic data that will be replaced with actual values when the template is rendered.
- **Functions**: Custom functions or logic that can be used to manipulate the template's content.

Here's an example of a simple chat template:

```
{{- range .Messages }}
{{ .Role }}: {{ .Content }}
```

```
{{- end }}
```

In this example, we have:

- A basic messages structure (layout)
- Three variables: `Messages`, `Role`, and `Content` (variables)
- A custom function (action) that iterates over an array of items (`range .Messages`) and displays each item

Adding templates to your model

By default, models imported into Ollama have a default template of `{{ .Prompt }}`, i.e. user inputs are sent verbatim to the LLM. This is appropriate for text or code completion models but lacks essential markers for chat or instruction models.

Omitting a template in these models puts the responsibility of correctly templating input onto the user. Adding a template allows users to easily get the best results from the model.

To add templates in your model, you'll need to add a `TEMPLATE` command to the Modelfile. Here's an example using Meta's Llama 3.

```
FROM llama3.2

TEMPLATE """{{- if .System }}<|start_header_id|>system<|end_header_id|>

{{ .System }}<|eot_id|>
{{- end }}
{{- range .Messages }}<|start_header_id|>{{ .Role }}<|end_header_id|>

{{ .Content }}<|eot_id|>
{{- end }}<|start_header_id|>assistant<|end_header_id|>

"""
```

Variables

`System` (string): system prompt

`Prompt` (string): user prompt

`Response` (string): assistant response

`Suffix` (string): text inserted after the assistant's response

`Messages` (list): list of messages

`Messages[].Role` (string): role which can be one of `system`, `user`, `assistant`, or `tool`

`Messages[].Content` (string): message content

`Messages[].ToolCalls` (list): list of tools the model wants to call

`Messages[].ToolCalls[].Function` (object): function to call

`Messages[].ToolCalls[].Function.Name` (string): function name

`Messages[].ToolCalls[].Function.Arguments` (map): mapping of argument name to argument value

`Tools` (list): list of tools the model can access

`Tools[].Type` (string): schema type. `type` is always `function`

`Tools[].Function` (object): function definition

`Tools[].Function.Name` (string): function name

`Tools[].Function.Description` (string): function description

`Tools[].Function.Parameters` (object): function parameters

`Tools[].Function.Parameters.Type` (string): schema type. `type` is always `object`

`Tools[].Function.Parameters.Required` (list): list of required properties

`Tools[].Function.Parameters.Properties` (map): mapping of property name to property definition

`Tools[].Function.Parameters.Properties[].Type` (string): property type

`Tools[].Function.Parameters.Properties[].Description` (string): property description

`Tools[].Function.Parameters.Properties[].Enum` (list): list of valid values

Tips and Best Practices

Keep the following tips and best practices in mind when working with Go templates:

- **Be mindful of dot**: Control flow structures like `range` and `with` changes the value `.`
- **Out-of-scope variables**: Use `$.` to reference variables not currently in scope, starting from the root
- **Whitespace control**: Use `-` to trim leading (`{{-`) and trailing (`-}}`) whitespace

Examples

Example Messages

ChatML

ChatML is a popular template format. It can be used for models such as Databrick's DBRX, Intel's Neural Chat, and Microsoft's Orca 2.

```
{{- range .Messages }}<|im_start|>{{ .Role }}
{{ .Content }}<|im_end|>
{{ end }}<|im_start|>assistant
```

Example Tools

Tools support can be added to a model by adding a `{{ .Tools }}` node to the template. This feature is useful for models trained to call external tools and can a powerful tool for retrieving real-time data or performing complex tasks.

Mistral

Mistral v0.3 and Mixtral 8x22B supports tool calling.

```
{{- range $index, $_ := .Messages }}
{{- if eq .Role "user" }}
{{- if and (le (len (slice $.Messages $index)) 2) $.Tools }}[AVAILABLE_TOOLS] {{ json $.Tools
}}[/AVAILABLE_TOOLS]
{{- end }}[INST] {{ if and (eq (len (slice $.Messages $index)) 1) $.System }}{{ $.System }}

{{ end }}{{ .Content }}[/INST]
{{- else if eq .Role "assistant" }}
{{- if .Content }} {{ .Content }}</s>
{{- else if .ToolCalls }}[TOOL_CALLS] [
```

```
{{- range .ToolCalls }}{"name": "{{ .Function.Name }}", "arguments": {{ json
.Function.Arguments }}}
{{- end }}]</s>
{{- end }}
{{- else if eq .Role "tool" }}[TOOL_RESULTS] {"content": {{ .Content }}}[/TOOL_RESULTS]
{{- end }}
{{- end }}
```

Example Fill-in-Middle

Fill-in-middle support can be added to a model by adding a `{{ .Suffix }}` node to the template. This feature is useful for models that are trained to generate text in the middle of user input, such as code completion models.

CodeLlama

CodeLlama 7B and 13B code completion models support fill-in-middle.

```
<PRE> {{ .Prompt }} <SUF>{{ .Suffix }} <MID>
```

CodeLlama 34B and 70B code completion and all instruct and Python fine-tuned models do not support fill-in-middle.

Codestral

Codestral 22B supports fill-in-middle.

```
[SUFFIX]{{ .Suffix }}[PREFIX] {{ .Prompt }}
```

□□□ #1
□ □□□ □□□ 2026-10-31 13:18:35 UTC
□ □□□ □□□ 2026-10-31 13:18:35 UTC