ollama/ollama

Documentation

Branch: main | Path: docs

Generated on 2025-07-29 18:57:29

Table of Contents

README.md

Documentation

Getting Started

- Quickstart
- Examples
- Importing models
- MacOS Documentation
- Linux Documentation
- Windows Documentation
- Docker Documentation

Reference

- API Reference
- Modelfile Reference
- OpenAI Compatibility

Resources

- Troubleshooting Guide
- FAQ
- Development guide

api.md

API

Endpoints

Conventions

Model names

Model names follow a `model:tag` format, where `model` can have an optional namespace such as `example/model`. Some examples are `orca-mini:3b-q8_0` and `llama3:70b`. The tag is optional and, if not provided, will default to `latest`. The tag is used to identify a specific version.

Durations

All durations are returned in nanoseconds.

Streaming responses

Certain endpoints stream responses as JSON objects. Streaming can be disabled by providing `{"stream": false}` for these endpoints.

Generate a completion

```
POST /api/generate
```

Generate a response for a given prompt with a provided model. This is a streaming endpoint, so there will be a series of responses. The final response object will include statistics and additional data from the request.

Parameters

- `model` : (required) the model name
- `prompt` : the prompt to generate a response for
- `suffix` : the text after the model response
- `images` : (optional) a list of base64-encoded images (for multimodal models such as `llava`)
- `think` : (for thinking models) should the model think before responding?

Advanced parameters (optional):

- `format` : the format to return a response in. Format can be `json` or a JSON schema
- `options` : additional model parameters listed in the documentation for the Modelfile such as `temperature`
- `system` : system message to (overrides what is defined in the `Modelfile`)
- `template` : the prompt template to use (overrides what is defined in the `Modelfile`)
- `stream` : if `false` the response will be returned as a single response object, rather than a stream of objects

- `raw`: if `true` no formatting will be applied to the prompt. You may choose to use the `raw` parameter if you are specifying a full templated prompt in your request to the API
- `keep_alive`: controls how long the model will stay loaded into memory following the request (default: `5m`)
- `context` (deprecated): the context parameter returned from a previous request to `/generate`, this can be used to keep a short conversational memory

Structured outputs

Structured outputs are supported by providing a JSON schema in the `format` parameter. The model will generate a response that matches the schema. See the structured outputs example below.

JSON mode

Enable JSON mode by setting the `format` parameter to `json`. This will structure the response as a valid JSON object. See the JSON mode example below.

> [!IMPORTANT] It's important to instruct the model to use JSON in the `prompt`. Otherwise, the model may generate large amounts whitespace.

Examples

Generate request (Streaming)

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "Why is the sky blue?"
}'
```

Response

A stream of JSON objects is returned:

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T08:52:19.385406455-07:00",
  "response": "The",
  "done": false
}
```

The final response in the stream also includes additional data about the generation:

- `total_duration`: time spent generating the response
- `load_duration`: time spent in nanoseconds loading the model
- `prompt_eval_count`: number of tokens in the prompt
- `prompt_eval_duration`: time spent in nanoseconds evaluating the prompt
- `eval_count`: number of tokens in the response
- `eval_duration`: time in nanoseconds spent generating the response
- `context`: an encoding of the conversation used in this response, this can be sent in the next request to keep a conversational memory
- `response`: empty if the response was streamed, if not streamed, this will contain the full response

To calculate how fast the response is generated in tokens per second (token/s), divide `eval_count` / `eval_duration` * `10^9`.

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T19:22:45.499127Z",
  "response": "",
  "done": true,
  "context": [1, 2, 3],
  "total_duration": 10706818083,
  "load_duration": 6338219291,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 130079000,
  "eval_count": 259,
  "eval_duration": 4232710000
}
```

Request (No streaming)

Request

A response can be received in one reply when streaming is off.

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "Why is the sky blue?",
  "stream": false
}'
```

Response

If `stream` is set to `false`, the response will be a single JSON object:

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T19:22:45.499127Z",
  "response": "The sky is blue because it is the color of
  "done": true,
  "context": [1, 2, 3],
  "total_duration": 5043500667,
  "load_duration": 5025959,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 325953000,
  "eval_count": 290,
  "eval_duration": 4709213000
}
```

Request (with suffix)

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "codellama:code",
  "prompt": "def compute_gcd(a, b):",
  "suffix": "    return result",
  "options": {
    "temperature": 0
  },
  "stream": false
}'
```

Response

```
{
  "model": "codellama:code",
  "created_at": "2024-07-22T20:47:51.147561Z",
  "response": "\n  if a == 0:\n    return b\n  else:\n
  "done": true,
  "done_reason": "stop",
  "context": [...],
  "total_duration": 1162761250,
  "load_duration": 6683708,
  "prompt_eval_count": 17,
  "prompt_eval_duration": 201222000,
  "eval_count": 63,
  "eval_duration": 953997000
}
```

Request (Structured outputs)

Request

```
curl -X POST http://localhost:11434/api/generate -H "Conte
  "model": "llama3.1:8b",
  "prompt": "Ollama is 22 years old and is busy saving the
  "stream": false,
  "format": {
    "type": "object",
    "properties": {
      "age": {
        "type": "integer"
      },
      "available": {
        "type": "boolean"
      }
    },
    "required": [
      "age",
      "available"
    ]
  }
}'
```

Response

```
{
  "model": "llama3.1:8b",
  "created_at": "2024-12-06T00:48:09.983619Z",
  "response": "{\n  \"age\": 22,\n  \"available\": true\n}
  "done": true,
  "done_reason": "stop",
  "context": [1, 2, 3],
  "total_duration": 1075509083,
  "load_duration": 567678166,
  "prompt_eval_count": 28,
  "prompt_eval_duration": 236000000,
  "eval_count": 16,
  "eval_duration": 269000000
}
```

Request (JSON mode)

> [!IMPORTANT] When `format` is set to `json`, the output will always be a well-formed JSON object. It's important to also instruct the model to respond in JSON.

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "What color is the sky at different times of t
  "format": "json",
  "stream": false
}'
```

Response

```
{
  "model": "llama3.2",
  "created_at": "2023-11-09T21:07:55.186497Z",
  "response": "{\n\"morning\": {\n\"color\": \"blue\"\n},\
  "done": true,
  "context": [1, 2, 3],
  "total_duration": 4648158584,
  "load_duration": 4071084,
  "prompt_eval_count": 36,
  "prompt_eval_duration": 439038000,
  "eval_count": 180,
  "eval_duration": 4196918000
}
```

The value of `response` will be a string containing JSON similar to:

```
 {
  "morning": {
    "color": "blue"
  },
  "noon": {
    "color": "blue-gray"
  },
  "afternoon": {
    "color": "warm gray"
  },
  "evening": {
    "color": "orange"
  }
}
```

Request (with images)

To submit images to multimodal models such as `llava` or `bakllava`, provide a list of base64-encoded `images`:

Request

```
 curl http://localhost:11434/api/generate -d '{
  "model": "llava",
  "prompt":"What is in this picture?",
  "stream": false,
  "images": ["iVBORw0KGgoAAAANSUhEUgAAAG0AAABmCAYAAADBPx+V
}'
```

Response

```
{
  "model": "llava",
  "created_at": "2023-11-03T15:36:02.583064Z",
  "response": "A happy cartoon character, which is cute an
  "done": true,
  "context": [1, 2, 3],
  "total_duration": 2938432250,
  "load_duration": 2559292,
  "prompt_eval_count": 1,
  "prompt_eval_duration": 2195557000,
  "eval_count": 44,
  "eval_duration": 736432000
}
```

Request (Raw Mode)

In some cases, you may wish to bypass the templating system and provide a full prompt. In this case, you can use the `raw` parameter to disable templating. Also note that raw mode will not return a context.

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "mistral",
  "prompt": "[INST] why is the sky blue? [/INST]",
  "raw": true,
  "stream": false
}'
```

Request (Reproducible outputs)

For reproducible outputs, set `seed` to a number:

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "mistral",
  "prompt": "Why is the sky blue?",
  "options": {
    "seed": 123
  }
}'
```

Response

```
{
  "model": "mistral",
  "created_at": "2023-11-03T15:36:02.583064Z",
  "response": " The sky appears blue because of a phenomen
  "done": true,
  "total_duration": 8493852375,
  "load_duration": 6589624375,
  "prompt_eval_count": 14,
  "prompt_eval_duration": 119039000,
  "eval_count": 110,
  "eval_duration": 1779061000
}
```

Generate request (With options)

If you want to set custom options for the model at runtime rather than in the Modelfile, you can do so with the `options` parameter. This example sets every available option, but you can set any of them individually and omit the ones you do not want to override.

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "Why is the sky blue?",
  "stream": false,
  "options": {
    "num_keep": 5,
    "seed": 42,
    "num_predict": 100,
    "top_k": 20,
    "top_p": 0.9,
    "min_p": 0.0,
    "typical_p": 0.7,
    "repeat_last_n": 33,
    "temperature": 0.8,
    "repeat_penalty": 1.2,
    "presence_penalty": 1.5,
    "frequency_penalty": 1.0,
    "penalize_newline": true,
    "stop": ["\n", "user:"],
    "numa": false,
    "num_ctx": 1024,
    "num_batch": 2,
    "num_gpu": 1,
    "main_gpu": 0,
    "use_mmap": true,
    "num_thread": 8
  }
}'
```

Response

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T19:22:45.499127Z",
  "response": "The sky is blue because it is the color of
  "done": true,
  "context": [1, 2, 3],
  "total_duration": 4935886791,
  "load_duration": 534986708,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 107345000,
  "eval_count": 237,
  "eval_duration": 4289432000
}
```

Load a model

If an empty prompt is provided, the model will be loaded into memory.

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2"
}'
```

Response

A single JSON object is returned:

```
{
  "model": "llama3.2",
  "created_at": "2023-12-18T19:52:07.071755Z",
  "response": "",
  "done": true
}
```

Unload a model

If an empty prompt is provided and the `keep_alive` parameter is set to `0`, a model will be unloaded from memory.

Request

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "keep_alive": 0
}'
```

Response

A single JSON object is returned:

```
{
  "model": "llama3.2",
  "created_at": "2024-09-12T03:54:03.516566Z",
  "response": "",
  "done": true,
  "done_reason": "unload"
}
```

Generate a chat completion

```
POST /api/chat
```

Generate the next message in a chat with a provided model. This is a streaming endpoint, so there will be a series of responses. Streaming can be disabled using `"stream": false`. The final response object will include statistics and additional data from the request.

Parameters

- `model`: (required) the model name
- `messages`: the messages of the chat, this can be used to keep a chat memory
- `tools`: list of tools in JSON for the model to use if supported
- `think`: (for thinking models) should the model think before responding?

The `message` object has the following fields:

- `role`: the role of the message, either `system`, `user`, `assistant`, or `tool`
- `content`: the content of the message
- `thinking`: (for thinking models) the model's thinking process
- `images` (optional): a list of images to include in the message (for multimodal models such as `llava`)
- `tool_calls` (optional): a list of tools in JSON that the model wants to use
- `tool_name` (optional): add the name of the tool that was executed to inform the model of the result

Advanced parameters (optional):

- `format`: the format to return a response in. Format can be `json` or a JSON schema.
- `options`: additional model parameters listed in the documentation for the Modelfile such as `temperature`
- `stream`: if `false` the response will be returned as a single response object, rather than a stream of objects
- `keep_alive`: controls how long the model will stay loaded into memory following the request (default: `5m`)

Tool calling

Tool calling is supported by providing a list of tools in the `tools` parameter. The model will generate a response that includes a list of tool calls. See the Chat request (Streaming with tools) example below.

Models can also explain the result of the tool call in the response. See the Chat request (With history, with tools) example below.

See models with tool calling capabilities.

Structured outputs

Structured outputs are supported by providing a JSON schema in the `format` parameter. The model will generate a response that matches the schema. See the Chat request (Structured outputs) example below.

Examples

Chat request (Streaming)

Request

Send a chat message with a streaming response.

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "why is the sky blue?"
    }
  ]
}'
```

Response

A stream of JSON objects is returned:

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T08:52:19.385406455-07:00",
  "message": {
    "role": "assistant",
    "content": "The",
    "images": null
  },
  "done": false
}
```

Final response:

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T19:22:45.499127Z",
  "message": {
    "role": "assistant",
    "content": ""
  },
  "done": true,
  "total_duration": 4883583458,
  "load_duration": 1334875,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 342546000,
  "eval_count": 282,
  "eval_duration": 4535599000
}
```

Chat request (Streaming with tools)

Request

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "what is the weather in tokyo?"
    }
  ],
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Get the weather in a given city",
        "parameters": {
          "type": "object",
          "properties": {
            "city": {
              "type": "string",
              "description": "The city to get the weather
            }
          },
          "required": ["city"]
        }
      }
    }
  ],
  "stream": true
}'
```

Response

A stream of JSON objects is returned:

```
{
    "model": "llama3.2",
    "created_at": "2025-07-07T20:22:19.184789Z",
    "message": {
        "role": "assistant",
        "content": "",
        "tool_calls": [
            {
                "function": {
                    "name": "get_weather",
                    "arguments": {
                        "city": "Tokyo"
                    }
                },
            }
        ]
    },
    "done": false
}
```

Final response:

```
 {
  "model":"llama3.2",
  "created_at":"2025-07-07T20:22:19.19314Z",
  "message": {
    "role": "assistant",
    "content": ""
  },
  "done_reason": "stop",
  "done": true,
  "total_duration": 182242375,
  "load_duration": 41295167,
  "prompt_eval_count": 169,
  "prompt_eval_duration": 24573166,
  "eval_count": 15,
  "eval_duration": 115959084
}
```

Chat request (No streaming)

Request

```
 curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "why is the sky blue?"
    }
  ],
  "stream": false
}'
```

Response

```
{
  "model": "llama3.2",
  "created_at": "2023-12-12T14:13:43.416799Z",
  "message": {
    "role": "assistant",
    "content": "Hello! How are you today?"
  },
  "done": true,
  "total_duration": 5191566416,
  "load_duration": 2154458,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 383809000,
  "eval_count": 298,
  "eval_duration": 4799921000
}
```

Chat request (No streaming, with tools)

Request

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "what is the weather in tokyo?"
    }
  ],
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Get the weather in a given city",
        "parameters": {
          "type": "object",
          "properties": {
            "city": {
              "type": "string",
              "description": "The city to get the weather
            }
          },
          "required": ["city"]
        }
      }
    }
  ],
  "stream": false
}'
```

Response

```
{
  "model": "llama3.2",
  "created_at": "2025-07-07T20:32:53.844124Z",
  "message": {
    "role": "assistant",
    "content": "",
    "tool_calls": [
      {
        "function": {
          "name": "get_weather",
          "arguments": {
            "city": "Tokyo"
          }
        },
      }
    ]
  },
  "done_reason": "stop",
  "done": true,
  "total_duration": 3244883583,
  "load_duration": 2969184542,
  "prompt_eval_count": 169,
  "prompt_eval_duration": 141656333,
  "eval_count": 18,
  "eval_duration": 133293625
}
```

Chat request (Structured outputs)

Request

```
 curl -X POST http://localhost:11434/api/chat -H "Content-
  "model": "llama3.1",
  "messages": [{"role": "user", "content": "Ollama is 22 y
  "stream": false,
  "format": {
    "type": "object",
    "properties": {
      "age": {
        "type": "integer"
      },
      "available": {
        "type": "boolean"
      }
    },
    "required": [
      "age",
      "available"
    ]
  },
  "options": {
    "temperature": 0
  }
}'
```

Response

```
{
  "model": "llama3.1",
  "created_at": "2024-12-06T00:46:58.265747Z",
  "message": { "role": "assistant", "content": "{\"age\":
  "done_reason": "stop",
  "done": true,
  "total_duration": 2254970291,
  "load_duration": 574751416,
  "prompt_eval_count": 34,
  "prompt_eval_duration": 1502000000,
  "eval_count": 12,
  "eval_duration": 175000000
}
```

Chat request (With History)

Send a chat message with a conversation history. You can use this same approach to start the conversation using multi-shot or chain-of-thought prompting.

Request

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "why is the sky blue?"
    },
    {
      "role": "assistant",
      "content": "due to rayleigh scattering."
    },
    {
      "role": "user",
      "content": "how is that different than mie scatterin
    }
  ]
}'
```

Response

A stream of JSON objects is returned:

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T08:52:19.385406455-07:00",
  "message": {
    "role": "assistant",
    "content": "The"
  },
  "done": false
}
```

Final response:

```
{
  "model": "llama3.2",
  "created_at": "2023-08-04T19:22:45.499127Z",
  "done": true,
  "total_duration": 8113331500,
  "load_duration": 6396458,
  "prompt_eval_count": 61,
  "prompt_eval_duration": 398801000,
  "eval_count": 468,
  "eval_duration": 7701267000
}
```

Chat request (With history, with tools)

Request

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "what is the weather in Toronto?"
    },
    // the message from the model appended to history
    {
      "role": "assistant",
      "content": "",
      "tool_calls": [
        {
          "function": {
            "name": "get_temperature",
            "arguments": {
              "city": "Toronto"
            }
          },
        }
      ]
    },
    // the tool call result appended to history
    {
      "role": "tool",
      "content": "11 degrees celsius",
      "tool_name": "get_temperature",
    }
  ],
  "stream": false,
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Get the weather in a given city",
        "parameters": {
          "type": "object",
          "properties": {
            "city": {
              "type": "string",
              "description": "The city to get the weather
            }
          },
          "required": ["city"]
        }
      }
    }
  ]
}'
```

Response

```
 {
  "model": "llama3.2",
  "created_at": "2025-07-07T20:43:37.688511Z",
  "message": {
    "role": "assistant",
    "content": "The current temperature in Toronto is 11°C
  },
  "done_reason": "stop",
  "done": true,
  "total_duration": 890771750,
  "load_duration": 707634750,
  "prompt_eval_count": 94,
  "prompt_eval_duration": 91703208,
  "eval_count": 11,
  "eval_duration": 90282125
}
```

Chat request (with images)

Request

Send a chat message with images. The images should be provided as an array, with the individual images encoded in Base64.

```
curl http://localhost:11434/api/chat -d '{
  "model": "llava",
  "messages": [
    {
      "role": "user",
      "content": "what is in this image?",
      "images": ["iVBORw0KGgoAAAANSUhEUgAAAG0AAABmCAYAAADB
    }
  ]
}'
```

Response

```
{
  "model": "llava",
  "created_at": "2023-12-13T22:42:50.203334Z",
  "message": {
    "role": "assistant",
    "content": " The image features a cute, little pig wit
    "images": null
  },
  "done": true,
  "total_duration": 1668506709,
  "load_duration": 1986209,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 359682000,
  "eval_count": 83,
  "eval_duration": 1303285000
}
```

Chat request (Reproducible outputs)

Request

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    }
  ],
  "options": {
    "seed": 101,
    "temperature": 0
  }
}'
```

Response

```
{
  "model": "llama3.2",
  "created_at": "2023-12-12T14:13:43.416799Z",
  "message": {
    "role": "assistant",
    "content": "Hello! How are you today?"
  },
  "done": true,
  "total_duration": 5191566416,
  "load_duration": 2154458,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 383809000,
  "eval_count": 298,
  "eval_duration": 4799921000
}
```

Chat request (with tools)

Request

```
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "What is the weather today in Paris?"
    }
  ],
  "stream": false,
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_current_weather",
        "description": "Get the current weather for a loca
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The location to get the weat
            },
            "format": {
              "type": "string",
              "description": "The format to return the wea
              "enum": ["celsius", "fahrenheit"]
            }
          },
          "required": ["location", "format"]
        }
      }
    }
  ]
}'
```

Response

```json
{
  "model": "llama3.2",
  "created_at": "2024-07-22T20:33:28.123648Z",
  "message": {
    "role": "assistant",
    "content": "",
    "tool_calls": [
      {
        "function": {
          "name": "get_current_weather",
          "arguments": {
            "format": "celsius",
            "location": "Paris, FR"
          }
        }
      }
    ]
  },
  "done_reason": "stop",
  "done": true,
  "total_duration": 885095291,
  "load_duration": 3753500,
  "prompt_eval_count": 122,
  "prompt_eval_duration": 328493000,
  "eval_count": 33,
  "eval_duration": 552222000
}
```

Load a model

If the messages array is empty, the model will be loaded into memory.

Request

```
 curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": []
}'
```

Response

```
 {
  "model": "llama3.2",
  "created_at":"2024-09-12T21:17:29.110811Z",
  "message": {
    "role": "assistant",
    "content": ""
  },
  "done_reason": "load",
  "done": true
}
```

Unload a model

If the messages array is empty and the `keep_alive` parameter is set to `0`, a model will be unloaded from memory.

Request

```
 curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [],
  "keep_alive": 0
}'
```

Response

A single JSON object is returned:

```
{
  "model": "llama3.2",
  "created_at":"2024-09-12T21:33:17.547535Z",
  "message": {
    "role": "assistant",
    "content": ""
  },
  "done_reason": "unload",
  "done": true
}
```

Create a Model

```
POST /api/create
```

Create a model from: * another model; * a safetensors directory; or * a GGUF file.

If you are creating a model from a safetensors directory or from a GGUF file, you must [create a blob](#) for each of the files and then use the file name and SHA256 digest associated with each blob in the `files` field.

Parameters

- `model`: name of the model to create
- `from`: (optional) name of an existing model to create the new model from

- `files`: (optional) a dictionary of file names to SHA256 digests of blobs to create the model from
- `adapters`: (optional) a dictionary of file names to SHA256 digests of blobs for LORA adapters
- `template`: (optional) the prompt template for the model
- `license`: (optional) a string or list of strings containing the license or licenses for the model
- `system`: (optional) a string containing the system prompt for the model
- `parameters`: (optional) a dictionary of parameters for the model (see Modelfile for a list of parameters)
- `messages`: (optional) a list of message objects used to create a conversation
- `stream`: (optional) if `false` the response will be returned as a single response object, rather than a stream of objects
- `quantize` (optional): quantize a non-quantized (e.g. float16) model

Quantization types

Type	Recommended
q4_K_M	*
q4_K_S	
q8_0	*

Examples

Create a new model

Create a new model from an existing model.

Request

```
curl http://localhost:11434/api/create -d '{
  "model": "mario",
  "from": "llama3.2",
  "system": "You are Mario from Super Mario Bros."
}'
```

Response

A stream of JSON objects is returned:

```
{"status":"reading model metadata"}
{"status":"creating system layer"}
{"status":"using already created layer sha256:22f7f8ef5f4c
{"status":"using already created layer sha256:8c17c2ebb0ea
{"status":"using already created layer sha256:7c23fb36d801
{"status":"using already created layer sha256:2e0493f67d0c
{"status":"using already created layer sha256:2759286baa87
{"status":"writing layer sha256:df30045fe90f0d750db82a0581
{"status":"writing layer sha256:f18a68eb09bf925bb1b6694904
{"status":"writing manifest"}
{"status":"success"}
```

Quantize a model

Quantize a non-quantized model.

Request

```
curl http://localhost:11434/api/create -d '{
  "model": "llama3.2:quantized",
  "from": "llama3.2:3b-instruct-fp16",
  "quantize": "q4_K_M"
}'
```

Response

A stream of JSON objects is returned:

```
{"status":"quantizing F16 model to Q4_K_M","digest":"0","
{"status":"quantizing F16 model to Q4_K_M","digest":"0","t
{"status":"verifying conversion"}
{"status":"creating new layer sha256:fb7f4f211b89c6c4928ff
{"status":"using existing layer sha256:966de95ca8a62200913
{"status":"using existing layer sha256:fcc5a6bec9daf9b561a
{"status":"using existing layer sha256:a70ff7e570d97baaf4e
{"status":"using existing layer sha256:56bb8bd477a519ffa69
{"status":"writing manifest"}
{"status":"success"}
```

Create a model from GGUF

Create a model from a GGUF file. The `files` parameter should be filled out with the file name and SHA256 digest of the GGUF file you wish to use. Use /api/blobs/:digest to push the GGUF file to the server before calling this API.

Request

```
curl http://localhost:11434/api/create -d '{
  "model": "my-gguf-model",
  "files": {
    "test.gguf": "sha256:432f310a77f4650a88d0fd59ecdd7cebe
  }
}'
```

Response

A stream of JSON objects is returned:

```
{"status":"parsing GGUF"}
{"status":"using existing layer sha256:432f310a77f4650a88d
{"status":"writing manifest"}
{"status":"success"}
```

Create a model from a Safetensors directory

The `files` parameter should include a dictionary of files for the safetensors model which includes the file names and SHA256 digest of each file. Use [/api/blobs/:digest]() to first push each of the files to the server before calling this API. Files will remain in the cache until the Ollama server is restarted.

Request

```
curl http://localhost:11434/api/create -d '{
  "model": "fred",
  "files": {
    "config.json": "sha256:dd3443e529fb2290423a0c65c2d633e
    "generation_config.json": "sha256:88effbb63300dbbc7390
    "special_tokens_map.json": "sha256:b7455f0e8f005391088
    "tokenizer.json": "sha256:bbc1904d35169c542dffbe1f7589
    "tokenizer_config.json": "sha256:24e8a6dc2547164b7002e
    "model.safetensors": "sha256:1ff795ff6a07e6a68085d206f
  }
}'
```

Response

A stream of JSON objects is returned:

```
{"status":"converting model"}
{"status":"creating new layer sha256:05ca5b813af4a53d2c292
{"status":"using autodetected template llama3-instruct"}
{"status":"using existing layer sha256:56bb8bd477a519ffa69
{"status":"writing manifest"}
{"status":"success"}
```

Check if a Blob Exists

```
HEAD /api/blobs/:digest
```

Ensures that the file blob (Binary Large Object) used with create a model exists on the server. This checks your Ollama server and not ollama.com.

Query Parameters

- `digest`: the SHA256 digest of the blob

Examples

Request

```
curl -I http://localhost:11434/api/blobs/sha256:29fdb92e5
```

Response

Return 200 OK if the blob exists, 404 Not Found if it does not.

Push a Blob

```
POST /api/blobs/:digest
```

Push a file to the Ollama server to create a "blob" (Binary Large Object).

Query Parameters

- `digest`: the expected SHA256 digest of the file

Examples

Request

```
curl -T model.gguf -X POST http://localhost:11434/api/blol
```

Response

Return 201 Created if the blob was successfully created, 400 Bad Request if the digest used is not expected.

List Local Models

```
GET /api/tags
```

List models that are available locally.

Examples

Request

```
curl http://localhost:11434/api/tags
```

Response

A single JSON object will be returned.

```
{
  "models": [
    {
      "name": "deepseek-r1:latest",
      "model": "deepseek-r1:latest",
      "modified_at": "2025-05-10T08:06:48.639712648-07:00"
      "size": 4683075271,
      "digest": "0a8c266910232fd3291e71e5ba1e058cc5af9d411
      "details": {
        "parent_model": "",
        "format": "gguf",
        "family": "qwen2",
        "families": [
          "qwen2"
        ],
        "parameter_size": "7.6B",
        "quantization_level": "Q4_K_M"
      }
    },
    {
      "name": "llama3.2:latest",
      "model": "llama3.2:latest",
      "modified_at": "2025-05-04T17:37:44.706015396-07:00"
      "size": 2019393189,
      "digest": "a80c4f17acd55265feec403c7aef86be0c25983ab
      "details": {
        "parent_model": "",
        "format": "gguf",
        "family": "llama",
        "families": [
          "llama"
        ],
        "parameter_size": "3.2B",
        "quantization_level": "Q4_K_M"
      }
    }
  ]
}
```

Show Model Information

```
POST /api/show
```

Show information about a model including details, modelfile, template, parameters, license, system prompt.

Parameters

- `model` : name of the model to show
- `verbose` : (optional) if set to `true` , returns full data for verbose response fields

Examples

Request

```
curl http://localhost:11434/api/show -d '{
  "model": "llava"
}'
```

Response

```
{
  "modelfile": "# Modelfile generated by \"ollama show\"\n
  "parameters": "num_keep                        24\nstop
  "template": "{{ if .System }}<|start_header_id|>system<|
  "details": {
    "parent_model": "",
    "format": "gguf",
    "family": "llama",
    "families": [
      "llama"
    ],
    "parameter_size": "8.0B",
    "quantization_level": "Q4_0"
  },
  "model_info": {
    "general.architecture": "llama",
    "general.file_type": 2,
    "general.parameter_count": 8030261248,
    "general.quantization_version": 2,
    "llama.attention.head_count": 32,
    "llama.attention.head_count_kv": 8,
    "llama.attention.layer_norm_rms_epsilon": 0.00001,
    "llama.block_count": 32,
    "llama.context_length": 8192,
    "llama.embedding_length": 4096,
    "llama.feed_forward_length": 14336,
    "llama.rope.dimension_count": 128,
    "llama.rope.freq_base": 500000,
    "llama.vocab_size": 128256,
    "tokenizer.ggml.bos_token_id": 128000,
    "tokenizer.ggml.eos_token_id": 128009,
    "tokenizer.ggml.merges": [],             // populates i
    "tokenizer.ggml.model": "gpt2",
    "tokenizer.ggml.pre": "llama-bpe",
    "tokenizer.ggml.token_type": [],         // populates i
    "tokenizer.ggml.tokens": []              // populates i
  },
  "capabilities": [
    "completion",
    "vision"
  ],
}
```

Copy a Model

```
POST /api/copy
```

Copy a model. Creates a model with another name from an existing model.

Examples

Request

```
 curl http://localhost:11434/api/copy -d '{
  "source": "llama3.2",
  "destination": "llama3-backup"
}'
```

Response

Returns a 200 OK if successful, or a 404 Not Found if the source model doesn't exist.

Delete a Model

```
DELETE /api/delete
```

Delete a model and its data.

Parameters

- `model`: model name to delete

Examples

Request

```
 curl -X DELETE http://localhost:11434/api/delete -d '{
  "model": "llama3:13b"
}'
```

Response

Returns a 200 OK if successful, 404 Not Found if the model to be deleted doesn't exist.

Pull a Model

```
POST /api/pull
```

Download a model from the ollama library. Cancelled pulls are resumed from where they left off, and multiple calls will share the same download progress.

Parameters

- `model` : name of the model to pull
- `insecure` : (optional) allow insecure connections to the library. Only use this if you are pulling from your own library during development.
- `stream` : (optional) if `false` the response will be returned as a single response object, rather than a stream of objects

Examples

Request

```
curl http://localhost:11434/api/pull -d '{
  "model": "llama3.2"
}'
```

Response

If `stream` is not specified, or set to `true` , a stream of JSON objects is returned:

The first object is the manifest:

```
 {
  "status": "pulling manifest"
}
```

Then there is a series of downloading responses. Until any of the download is completed, the `completed` key may not be included. The number of files to be downloaded depends on the number of layers specified in the manifest.

```
 {
  "status": "downloading digestname",
  "digest": "digestname",
  "total": 2142590208,
  "completed": 241970
}
```

After all the files are downloaded, the final responses are:

```
 {
    "status": "verifying sha256 digest"
}
{
    "status": "writing manifest"
}
{
    "status": "removing any unused layers"
}
{
    "status": "success"
}
```

if `stream` is set to false, then the response is a single JSON object:

```
{
  "status": "success"
}
```

Push a Model

```
POST /api/push
```

Upload a model to a model library. Requires registering for ollama.ai and adding a public key first.

Parameters

- `model`: name of the model to push in the form of `<namespace>/<model>:<tag>`
- `insecure`: (optional) allow insecure connections to the library. Only use this if you are pushing to your library during development.
- `stream`: (optional) if `false` the response will be returned as a single response object, rather than a stream of objects

Examples

Request

```
curl http://localhost:11434/api/push -d '{
  "model": "mattw/pygmalion:latest"
}'
```

Response

If `stream` is not specified, or set to `true`, a stream of JSON objects is returned:

```
{ "status": "retrieving manifest" }
```

and then:

```
{
  "status": "starting upload",
  "digest": "sha256:bc07c81de745696fdf5afca05e065818a8149f
  "total": 1928429856
}
```

Then there is a series of uploading responses:

```
{
  "status": "starting upload",
  "digest": "sha256:bc07c81de745696fdf5afca05e065818a8149f
  "total": 1928429856
}
```

Finally, when the upload is complete:

```
{"status":"pushing manifest"}
{"status":"success"}
```

If `stream` is set to `false`, then the response is a single JSON object:

```
{ "status": "success" }
```

Generate Embeddings

```
POST /api/embed
```

Generate embeddings from a model

Parameters

- `model` : name of model to generate embeddings from
- `input` : text or list of text to generate embeddings for

Advanced parameters:

- `truncate` : truncates the end of each input to fit within context length. Returns error if `false` and context length is exceeded. Defaults to `true`
- `options` : additional model parameters listed in the documentation for the Modelfile such as `temperature`
- `keep_alive` : controls how long the model will stay loaded into memory following the request (default: `5m`)

Examples

Request

```
curl http://localhost:11434/api/embed -d '{
  "model": "all-minilm",
  "input": "Why is the sky blue?"
}'
```

Response

```
{
  "model": "all-minilm",
  "embeddings": [[
    0.010071029, -0.0017594862, 0.05007221, 0.04692972, 0.
    0.008599704, 0.105441414, -0.025878139, 0.12958129, 0.
  ]],
  "total_duration": 14143917,
  "load_duration": 1019500,
  "prompt_eval_count": 8
}
```

Request (Multiple input)

```
curl http://localhost:11434/api/embed -d '{
  "model": "all-minilm",
  "input": ["Why is the sky blue?", "Why is the grass gree
}'
```

Response

```
{
  "model": "all-minilm",
  "embeddings": [[
    0.010071029, -0.0017594862, 0.05007221, 0.04692972, 0.
    0.008599704, 0.105441414, -0.025878139, 0.12958129, 0.
  ],[
    -0.0098027075, 0.06042469, 0.025257962, -0.006364387,
    0.017194884, 0.09032035, -0.051705178, 0.09951512, 0.0
  ]]
}
```

List Running Models

```
GET /api/ps
```

List models that are currently loaded into memory.

Examples

Request

```
curl http://localhost:11434/api/ps
```

Response

A single JSON object will be returned.

```
{
  "models": [
    {
      "name": "mistral:latest",
      "model": "mistral:latest",
      "size": 5137025024,
      "digest": "2ae6f6dd7a3dd734790bbbf58b8909a606e0e7e97
      "details": {
        "parent_model": "",
        "format": "gguf",
        "family": "llama",
        "families": [
          "llama"
        ],
        "parameter_size": "7.2B",
        "quantization_level": "Q4_0"
      },
      "expires_at": "2024-06-04T14:38:31.83753-07:00",
      "size_vram": 5137025024
    }
  ]
}
```

Generate Embedding

> Note: this endpoint has been superseded by `/api/embed`

```
POST /api/embeddings
```

Generate embeddings from a model

Parameters

- `model` : name of model to generate embeddings from
- `prompt` : text to generate embeddings for

Advanced parameters:

- `options` : additional model parameters listed in the documentation for the Modelfile such as `temperature`
- `keep_alive` : controls how long the model will stay loaded into memory following the request (default: `5m`)

Examples

Request

```
curl http://localhost:11434/api/embeddings -d '{
  "model": "all-minilm",
  "prompt": "Here is an article about llamas..."
}'
```

Response

```
{
  "embedding": [
    0.5670403838157654, 0.009260174818336964, 0.2317874431
    0.8785552978515625, -0.34576427936553955, 0.5742510557
  ]
}
```

Version

```
GET /api/version
```

Retrieve the Ollama version

Examples

Request

```
curl http://localhost:11434/api/version
```

Response

```
{
  "version": "0.5.1"
}
```

development.md

Development

Install prerequisites:

- Go
- C/C++ Compiler e.g. Clang on macOS, TDM-GCC (Windows amd64) or llvm-mingw (Windows arm64), GCC/Clang on Linux.

Then build and run Ollama from the root directory of the repository:

```
go run . serve
```

macOS (Apple Silicon)

macOS Apple Silicon supports Metal which is built-in to the Ollama binary. No additional steps are required.

macOS (Intel)

Install prerequisites:

- CMake or `brew install cmake`

Then, configure and build the project:

```
 cmake -B build
cmake --build build
```

Lastly, run Ollama:

```
go run . serve
```

Windows

Install prerequisites:

- CMake
- Visual Studio 2022 including the Native Desktop Workload
- (Optional) AMD GPU support
    - ROCm
    - Ninja
- (Optional) NVIDIA GPU support
    - CUDA SDK

Then, configure and build the project:

```
 cmake -B build
cmake --build build --config Release
```

> [!IMPORTANT] Building for ROCm requires additional flags: `cmake -B build -G Ninja -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ cmake --build build --config Release`

Lastly, run Ollama:

```
go run . serve
```

Windows (ARM)

Windows ARM does not support additional acceleration libraries at this time. Do not use cmake, simply `go run` or `go build`.

Linux

Install prerequisites:

- CMake or `sudo apt install cmake` or `sudo dnf install cmake`

- (Optional) AMD GPU support
  - [ROCm]
- (Optional) NVIDIA GPU support
  - [CUDA SDK]

> [!IMPORTANT] Ensure prerequisites are in `PATH` before running CMake.

Then, configure and build the project:

```
 cmake -B build
cmake --build build
```

Lastly, run Ollama:

```
go run . serve
```

Docker

```
docker build .
```

ROCm

```
docker build --build-arg FLAVOR=rocm .
```

Running tests

To run tests, use `go test`:

```
go test ./...
```

> NOTE: In rare circumstances, you may need to change a package using the new "synctest" package in go1.24.

> If you do not have the "synctest" package enabled, you will not see build or test failures resulting from your change(s), if any, locally, but CI will break.
>
> If you see failures in CI, you can either keep pushing changes to see if the CI build passes, or you can enable the "synctest" package locally to see the failures before pushing.
>
> To enable the "synctest" package for testing, run the following command:
>
> `shell GOEXPERIMENT=synctest go test ./...`
>
> If you wish to enable synctest for all go commands, you can set the `GOEXPERIMENT` environment variable in your shell profile or by using:
>
> `shell go env -w GOEXPERIMENT=synctest`
>
> Which will enable the "synctest" package for all go commands without needing to set it for all shell sessions.
>
> The synctest package is not required for production builds.

Library detection

Ollama looks for acceleration libraries in the following paths relative to the `ollama` executable:

- `./lib/ollama` (Windows)
- `../lib/ollama` (Linux)
- `.` (macOS)
- `build/lib/ollama` (for development)

If the libraries are not found, Ollama will not run with any acceleration libraries.

---

docker.md

Ollama Docker image

CPU only

```
docker run -d -v ollama:/root/.ollama -p 11434:11434 --nar
```

Nvidia GPU

Install the NVIDIA Container Toolkit.

Install with Apt

1. Configure the repository

   `shell curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey \ | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list \ | sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' \ | sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list sudo apt-get update`

2. Install the NVIDIA Container Toolkit packages

   `shell sudo apt-get install -y nvidia-container-toolkit`

Install with Yum or Dnf

1. Configure the repository

   `shell curl -s -L https://nvidia.github.io/libnvidia-container/stable/rpm/nvidia-container-toolkit.repo \ |`

`sudo tee /etc/yum.repos.d/nvidia-container-toolkit.repo`

2. Install the NVIDIA Container Toolkit packages

   `shell sudo yum install -y nvidia-container-toolkit`

Configure Docker to use Nvidia driver

```
 sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker
```

Start the container

```
docker run -d --gpus=all -v ollama:/root/.ollama -p 11434
```

> [!NOTE]
> If you're running on an NVIDIA JetPack system, Ollama can't automatically discover the correct JetPack version. Pass the environment variable JETSON_JETPACK=5 or JETSON_JETPACK=6 to the container to select version 5 or 6.

AMD GPU

To run Ollama using Docker with AMD GPUs, use the `rocm` tag and the following command:

```
docker run -d --device /dev/kfd --device /dev/dri -v ollar
```

Run model locally

Now you can run a model:

```
docker exec -it ollama ollama run llama3.2
```

Try different models

More models can be found on the Ollama library.

examples.md

Examples

This directory contains different examples of using Ollama.

Python examples

Ollama Python examples at ollama-python/examples

JavaScript examples

Ollama JavaScript examples at ollama-js/examples

OpenAI compatibility examples

Ollama OpenAI compatibility examples at ollama/examples/openai

Community examples

- LangChain Ollama Python
- LangChain Ollama JS

faq.md

FAQ

How can I upgrade Ollama?

Ollama on macOS and Windows will automatically download updates. Click on the taskbar or menubar item and then click "Restart to update" to apply the update. Updates can also be installed by downloading the latest version manually.

On Linux, re-run the install script:

```
curl -fsSL https://ollama.com/install.sh | sh
```

How can I view the logs?

Review the Troubleshooting docs for more about using logs.

Is my GPU compatible with Ollama?

Please refer to the GPU docs.

How can I specify the context window size?

By default, Ollama uses a context window size of 4096 tokens.

This can be overridden with the `OLLAMA_CONTEXT_LENGTH` environment variable. For example, to set the default context window to 8K, use:

```
OLLAMA_CONTEXT_LENGTH=8192 ollama serve
```

To change this when using `ollama run`, use `/set parameter`:

```
/set parameter num_ctx 4096
```

When using the API, specify the `num_ctx` parameter:

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "Why is the sky blue?",
  "options": {
    "num_ctx": 4096
  }
}'
```

How can I tell if my model was loaded onto the GPU?

Use the `ollama ps` command to see what models are currently loaded into memory.

```
ollama ps
```

> **Output**:
>
> `NAME ID SIZE PROCESSOR UNTIL llama3:70b bcfb190ca3a7 42 GB 100% GPU 4 minutes from now`

The `Processor` column will show which memory the model was loaded in to: * `100% GPU` means the model was loaded entirely into the GPU * `100% CPU` means the model was loaded entirely in system memory * `48%/52% CPU/GPU` means the model was loaded partially onto both the GPU and into system memory

How do I configure Ollama server?

Ollama server can be configured with environment variables.

Setting environment variables on Mac

If Ollama is run as a macOS application, environment variables should be set using `launchctl`:

1. For each environment variable, call `launchctl setenv`.

   `bash launchctl setenv OLLAMA_HOST "0.0.0.0:11434"`

2. Restart Ollama application.

Setting environment variables on Linux

If Ollama is run as a systemd service, environment variables should be set using `systemctl`:

1. Edit the systemd service by calling `systemctl edit ollama.service`. This will open an editor.

2. For each environment variable, add a line `Environment` under section `[Service]`:

   `ini [Service] Environment="OLLAMA_HOST=0.0.0.0:11434"`

3. Save and exit.

4. Reload `systemd` and restart Ollama:

`shell systemctl daemon-reload systemctl restart ollama`

Setting environment variables on Windows

On Windows, Ollama inherits your user and system environment variables.

1. First Quit Ollama by clicking on it in the task bar.

2. Start the Settings (Windows 11) or Control Panel (Windows 10) application and search for *environment variables*.

3. Click on *Edit environment variables for your account*.

4. Edit or create a new variable for your user account for `OLLAMA_HOST`, `OLLAMA_MODELS`, etc.

5. Click OK/Apply to save.

6. Start the Ollama application from the Windows Start menu.

How do I use Ollama behind a proxy?

Ollama pulls models from the Internet and may require a proxy server to access the models. Use `HTTPS_PROXY` to redirect outbound requests through the proxy. Ensure the proxy certificate is installed as a system certificate. Refer to the section above for how to use environment variables on your platform.

> [!NOTE] Avoid setting `HTTP_PROXY`. Ollama does not use HTTP for model pulls, only HTTPS. Setting `HTTP_PROXY` may interrupt client connections to the server.

How do I use Ollama behind a proxy in Docker?

The Ollama Docker container image can be configured to use a proxy by passing `-e HTTPS_PROXY=https://proxy.example.com` when starting the container.

Alternatively, the Docker daemon can be configured to use a proxy. Instructions are available for Docker Desktop on macOS, Windows, and Linux, and Docker daemon with systemd.

Ensure the certificate is installed as a system certificate when using HTTPS. This may require a new Docker image when using a self-signed certificate.

```
 FROM ollama/ollama
COPY my-ca.pem /usr/local/share/ca-certificates/my-ca.crt
RUN update-ca-certificates
```

Build and run this image:

```
 docker build -t ollama-with-ca .
docker run -d -e HTTPS_PROXY=https://my.proxy.example.com
```

Does Ollama send my prompts and answers back to ollama.com?

No. Ollama runs locally, and conversation data does not leave your machine.

How can I expose Ollama on my network?

Ollama binds 127.0.0.1 port 11434 by default. Change the bind address with the `OLLAMA_HOST` environment variable.

Refer to the section above for how to set environment variables on your platform.

How can I use Ollama with a proxy server?

Ollama runs an HTTP server and can be exposed using a proxy server such as Nginx. To do so, configure the proxy to forward requests and optionally set required headers (if not exposing Ollama on the network). For example, with Nginx:

```
 server {
    listen 80;
    server_name example.com;  # Replace with your domain o
    location / {
        proxy_pass http://localhost:11434;
        proxy_set_header Host localhost:11434;
    }
}
```

How can I use Ollama with ngrok?

Ollama can be accessed using a range of tools for tunneling tools. For example with Ngrok:

```
ngrok http 11434 --host-header="localhost:11434"
```

How can I use Ollama with Cloudflare Tunnel?

To use Ollama with Cloudflare Tunnel, use the `--url` and `--http-host-header` flags:

```
cloudflared tunnel --url http://localhost:11434 --http-hos
```

How can I allow additional web origins to access Ollama?

Ollama allows cross-origin requests from `127.0.0.1` and `0.0.0.0` by default. Additional origins can be configured with `OLLAMA_ORIGINS`.

For browser extensions, you'll need to explicitly allow the extension's origin pattern. Set `OLLAMA_ORIGINS` to include `chrome-extension://*`, `moz-extension://*`, and `safari-web-extension://*` if you wish to allow all browser extensions access, or specific extensions as needed:

```
# Allow all Chrome, Firefox, and Safari extensions
OLLAMA_ORIGINS=chrome-extension://*,moz-extension://*,safa
```

Refer to the section above for how to set environment variables on your platform.

Where are models stored?

- macOS: `~/.ollama/models`
- Linux: `/usr/share/ollama/.ollama/models`
- Windows: `C:\Users\%username%\.ollama\models`

How do I set them to a different location?

If a different directory needs to be used, set the environment variable `OLLAMA_MODELS` to the chosen directory.

> Note: on Linux using the standard installer, the `ollama` user needs read and write access to the specified directory. To assign the directory to the `ollama` user run `sudo chown -R ollama:ollama <directory>`.

Refer to the section above for how to set environment variables on your platform.

How can I use Ollama in Visual Studio Code?

There is already a large collection of plugins available for VSCode as well as other editors that leverage Ollama. See the list of extensions & plugins at the bottom of the main repository readme.

How do I use Ollama with GPU acceleration in Docker?

The Ollama Docker container can be configured with GPU acceleration in Linux or Windows (with WSL2). This requires the nvidia-container-toolkit. See ollama/ollama for more details.

GPU acceleration is not available for Docker Desktop in macOS due to the lack of GPU passthrough and emulation.

Why is networking slow in WSL2 on Windows 10?

This can impact both installing Ollama, as well as downloading models.

Open `Control Panel > Networking and Internet > View network status and tasks` and click on `Change adapter settings` on the left panel. Find the `vEthernel (WSL)` adapter, right click and select `Properties`. Click on `Configure` and open the `Advanced` tab. Search through each of the properties until you find `Large Send Offload Version 2 (IPv4)` and `Large Send Offload Version 2 (IPv6)`. *Disable* both of these properties.

How can I preload a model into Ollama to get faster response times?

If you are using the API you can preload a model by sending the Ollama server an empty request. This works with both the `/api/generate` and `/api/chat` API endpoints.

To preload the mistral model using the generate endpoint, use:

```
curl http://localhost:11434/api/generate -d '{"model": "mi
```

To use the chat completions endpoint, use:

```
curl http://localhost:11434/api/chat -d '{"model": "mistra
```

To preload a model using the CLI, use the command:

```
ollama run llama3.2 ""
```

How do I keep a model loaded in memory or make it unload immediately?

By default models are kept in memory for 5 minutes before being unloaded. This allows for quicker response times if you're making numerous requests to the LLM. If you want to immediately unload a model from memory, use the `ollama stop` command:

```
ollama stop llama3.2
```

If you're using the API, use the `keep_alive` parameter with the `/api/generate` and `/api/chat` endpoints to set the amount of time that a model stays in memory. The `keep_alive` parameter can be set to: * a duration string (such as "10m" or "24h") * a number in seconds (such as 3600) * any negative number which will keep the model loaded in memory (e.g. -1 or "-1m") * '0' which will unload the model immediately after generating a response

For example, to preload a model and leave it in memory use:

```
curl http://localhost:11434/api/generate -d '{"model": "ll
```

To unload the model and free up memory use:

```
curl http://localhost:11434/api/generate -d '{"model": "ll
```

Alternatively, you can change the amount of time all models are loaded into memory by setting the `OLLAMA_KEEP_ALIVE` environment variable when starting the Ollama server. The `OLLAMA_KEEP_ALIVE` variable uses the same parameter types as the `keep_alive` parameter types mentioned above. Refer to the section explaining how to configure the Ollama server to correctly set the environment variable.

The `keep_alive` API parameter with the `/api/generate` and `/api/chat` API endpoints will override the `OLLAMA_KEEP_ALIVE` setting.

How do I manage the maximum number of requests the Ollama server can queue?

If too many requests are sent to the server, it will respond with a 503 error indicating the server is overloaded. You can adjust how many requests may be queue by setting `OLLAMA_MAX_QUEUE`.

How does Ollama handle concurrent requests?

Ollama supports two levels of concurrent processing. If your system has sufficient available memory (system memory when using CPU inference, or VRAM for GPU inference) then multiple models can be loaded at the same time. For a given model, if there is sufficient available memory when the model is loaded, it can be configured to allow parallel request processing.

If there is insufficient available memory to load a new model request while one or more models are already loaded, all new requests will be queued until the new model can be loaded. As prior models become idle, one or more will be unloaded to make room for the new model. Queued requests will be processed in order. When using GPU inference new models must be able to completely fit in VRAM to allow concurrent model loads.

Parallel request processing for a given model results in increasing the context size by the number of parallel requests. For example, a 2K context with 4 parallel requests will result in an 8K context and additional memory allocation.

The following server settings may be used to adjust how Ollama handles concurrent requests on most platforms:

- `OLLAMA_MAX_LOADED_MODELS` - The maximum number of models that can be loaded concurrently provided they fit in available memory. The default is 3 * the number of GPUs or 3 for CPU inference.
- `OLLAMA_NUM_PARALLEL` - The maximum number of parallel requests each model will process at the same time. The default is 1, and will handle 1 request per model at a time.
- `OLLAMA_MAX_QUEUE` - The maximum number of requests Ollama will queue when busy before rejecting additional requests. The default is 512

Note: Windows with Radeon GPUs currently default to 1 model maximum due to limitations in ROCm v5.7 for available VRAM reporting. Once ROCm v6.2 is available, Windows Radeon will follow the defaults above. You may enable concurrent model loads on Radeon on Windows, but ensure you don't load more models than will fit into your GPUs VRAM.

How does Ollama load models on multiple GPUs?

When loading a new model, Ollama evaluates the required VRAM for the model against what is currently available. If the model will entirely fit on any single GPU, Ollama will load the model on that GPU. This typically provides the best performance as it reduces the amount of data transferring across the PCI bus during inference. If the model does not fit entirely on one GPU, then it will be spread across all the available GPUs.

How can I enable Flash Attention?

Flash Attention is a feature of most modern models that can significantly reduce memory usage as the context size grows. To enable Flash Attention, set the `OLLAMA_FLASH_ATTENTION` environment variable to `1` when starting the Ollama server.

How can I set the quantization type for the K/V cache?

The K/V context cache can be quantized to significantly reduce memory usage when Flash Attention is enabled.

To use quantized K/V cache with Ollama you can set the following environment variable:

- `OLLAMA_KV_CACHE_TYPE` - The quantization type for the K/V cache. Default is `f16`.

> Note: Currently this is a global option - meaning all models will run with the specified quantization type.

The currently available K/V cache quantization types are:

- `f16` - high precision and memory usage (default).
- `q8_0` - 8-bit quantization, uses approximately 1/2 the memory of `f16` with a very small loss in precision, this usually has no noticeable impact on the model's quality (recommended if not using f16).
- `q4_0` - 4-bit quantization, uses approximately 1/4 the memory of `f16` with a small-medium loss in precision that may be more noticeable at higher context sizes.

How much the cache quantization impacts the model's response quality will depend on the model and the task. Models that have a high GQA count (e.g. Qwen2) may see a larger impact on precision from quantization than models with a low GQA count.

You may need to experiment with different quantization types to find the best balance between memory usage and quality.

How can I stop Ollama from starting when I login to my computer

Ollama for Windows and macOS register as a login item during installation. You can disable this if you prefer not to have Ollama automatically start. Ollama will respect this setting across upgrades, unless you uninstall the application.

Windows - Remove `%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\Ollama.lnk`

MacOS Monterey (v12) - Open `Settings` -> `Users & Groups` -> `Login Items` and find the `Ollama` entry, then click the `-` (minus) to remove

MacOS Ventura (v13) and later - Open `Settings` and search for "Login Items", find the `Ollama` entry under "Allow in the Background`, then click the slider to disable.

---

gpu.md

GPU

Nvidia

Ollama supports Nvidia GPUs with compute capability 5.0+ and driver version 531 and newer.

Check your compute compatibility to see if your card is supported: https://developer.nvidia.com/cuda-gpus

Compute Capability	Family	Cards
12.0	GeForce RTX 50xx	`RTX 5060` `RTX 5060 Ti` `RTX 5070` `RTX 5070 Ti` `RTX 5080` `RTX 5090`
	NVIDIA Professioal	`RTX PRO 4000 Blackwell` `RTX PRO 4500 Blackwell` `RTX PRO 5000 Blackwell` `RTX PRO 6000 Blackwell`
9.0	NVIDIA	`H200` `H100`
8.9	GeForce RTX 40xx	`RTX 4090` `RTX 4080 SUPER` `RTX 4080` `RTX 4070 Ti SUPER` `RTX 4070 Ti` `RTX 4070 SUPER` `RTX 4070` `RTX 4060 Ti` `RTX 4060`
	NVIDIA Professional	`L4` `L40` `RTX 6000`
8.6	GeForce RTX 30xx	`RTX 3090 Ti` `RTX 3090` `RTX 3080 Ti` `RTX 3080` `RTX 3070 Ti` `RTX 3070` `RTX 3060 Ti`

Compute Capability	Family	Cards
		`RTX 3060` `RTX 3050 Ti` `RTX 3050`
	NVIDIA Professional	`A40` `RTX A6000` `RTX A5000` `RTX A4000` `RTX A3000` `RTX A2000` `A10` `A16` `A2`
8.0	NVIDIA	`A100` `A30`
7.5	GeForce GTX/RTX	`GTX 1650 Ti` `TITAN RTX` `RTX 2080 Ti` `RTX 2080` `RTX 2070` `RTX 2060`
	NVIDIA Professional	`T4` `RTX 5000` `RTX 4000` `RTX 3000` `T2000` `T1200` `T1000` `T600` `T500`
	Quadro	`RTX 8000` `RTX 6000` `RTX 5000` `RTX 4000`
7.0	NVIDIA	`TITAN V` `V100` `Quadro GV100`
6.1	NVIDIA TITAN	`TITAN Xp` `TITAN X`
	GeForce GTX	`GTX 1080 Ti` `GTX 1080` `GTX 1070 Ti` `GTX 1070` `GTX 1060` `GTX 1050 Ti` `GTX 1050`
	Quadro	`P6000` `P5200` `P4200` `P3200` `P5000` `P4000` `P3000` `P2200` `P2000` `P1000` `P620` `P600` `P500` `P520`
	Tesla	`P40` `P4`
6.0	NVIDIA	`Tesla P100` `Quadro GP100`
5.2	GeForce GTX	

Compute Capability	Family	Cards
		`GTX TITAN X` `GTX 980 Ti` `GTX 980` `GTX 970` `GTX 960` `GTX 950`
	Quadro	`M6000 24GB` `M6000` `M5000` `M5500M` `M4000` `M2200` `M2000` `M620`
	Tesla	`M60` `M40`
5.0	GeForce GTX	`GTX 750 Ti` `GTX 750` `NVS 810`
	Quadro	`K2200` `K1200` `K620` `M1200` `M520` `M5000M` `M4000M` `M3000M` `M2000M` `M1000M` `K620M` `M600M` `M500M`

For building locally to support older GPUs, see developer.md

GPU Selection

If you have multiple NVIDIA GPUs in your system and want to limit Ollama to use a subset, you can set `CUDA_VISIBLE_DEVICES` to a comma separated list of GPUs. Numeric IDs may be used, however ordering may vary, so UUIDs are more reliable. You can discover the UUID of your GPUs by running `nvidia-smi -L` If you want to ignore the GPUs and force CPU usage, use an invalid GPU ID (e.g., "-1")

Linux Suspend Resume

On linux, after a suspend/resume cycle, sometimes Ollama will fail to discover your NVIDIA GPU, and fallback to running on the CPU. You can workaround this driver bug by reloading the NVIDIA UVM driver with `sudo rmmod nvidia_uvm && sudo modprobe nvidia_uvm`

AMD Radeon

Ollama supports the following AMD GPUs:

Linux Support

Family	Cards and accelerators
AMD Radeon RX	`7900 XTX` `7900 XT` `7900 GRE` `7800 XT` `7700 XT` `7600 XT` `7600` `6950 XT` `6900 XTX` `6900XT` `6800 XT` `6800` `Vega 64` `Vega 56`
AMD Radeon PRO	`W7900` `W7800` `W7700` `W7600` `W7500` `W6900X` `W6800X Duo` `W6800X` `W6800` `V620` `V420` `V340` `V320` `Vega II Duo` `Vega II` `VII` `SSG`
AMD Instinct	`MI300X` `MI300A` `MI300` `MI250X` `MI250` `MI210` `MI200` `MI100` `MI60` `MI50`

Windows Support

With ROCm v6.1, the following GPUs are supported on Windows.

Family	Cards and accelerators
AMD Radeon RX	`7900 XTX` `7900 XT` `7900 GRE` `7800 XT` `7700 XT` `7600 XT` `7600` `6950 XT` `6900 XTX` `6900XT` `6800 XT` `6800`
AMD Radeon PRO	`W7900` `W7800` `W7700` `W7600` `W7500` `W6900X` `W6800X Duo` `W6800X` `W6800` `V620`

Overrides on Linux

Ollama leverages the AMD ROCm library, which does not support all AMD GPUs. In some cases you can force the system to try to use a similar LLVM target that is close. For example The Radeon RX 5400 is `gfx1034` (also known as 10.3.4) however, ROCm does not currently support this target. The closest support is `gfx1030`. You can use the environment variable `HSA_OVERRIDE_GFX_VERSION` with `x.y.z` syntax. So for example, to force the system to run on the RX 5400, you would set `HSA_OVERRIDE_GFX_VERSION="10.3.0"` as an environment variable for the server. If you have an unsupported AMD GPU you can experiment using the list of supported types below.

If you have multiple GPUs with different GFX versions, append the numeric device number to the environment variable to set them individually. For example, `HSA_OVERRIDE_GFX_VERSION_0=10.3.0` and `HSA_OVERRIDE_GFX_VERSION_1=11.0.0`

At this time, the known supported GPU types on linux are the following LLVM Targets. This table shows some example GPUs that map to these LLVM targets: | **LLVM Target** | **An Example GPU** | |-----------------|---------------------| | gfx900 | Radeon RX Vega 56 | | gfx906 | Radeon Instinct MI50 | | gfx908 | Radeon Instinct MI100 | | gfx90a | Radeon Instinct MI210 | | gfx940 | Radeon Instinct MI300 | | gfx941 | | | gfx942 | | | gfx1030 | Radeon PRO V620 | | gfx1100 | Radeon PRO W7900 | | gfx1101 | Radeon PRO W7700 | | gfx1102 | Radeon RX 7600 |

AMD is working on enhancing ROCm v6 to broaden support for families of GPUs in a future release which should increase support for more GPUs.

Reach out on Discord or file an issue for additional help.

GPU Selection

If you have multiple AMD GPUs in your system and want to limit Ollama to use a subset, you can set `ROCR_VISIBLE_DEVICES` to a comma separated list of GPUs. You can see the list of devices with `rocminfo`. If you want to ignore the GPUs and force CPU usage, use an invalid GPU ID (e.g., "-1"). When available, use the `Uuid` to uniquely identify the device instead of numeric value.

Container Permission

In some Linux distributions, SELinux can prevent containers from accessing the AMD GPU devices. On the host system you can run `sudo setsebool container_use_devices=1` to allow containers to use devices.

Metal (Apple GPUs)

Ollama supports GPU acceleration on Apple devices via the Metal API.

---

import.md

Importing a model

Table of Contents

Importing a fine tuned adapter from Safetensors weights

First, create a `Modelfile` with a `FROM` command pointing at the base model you used for fine tuning, and an `ADAPTER` command which points to the directory with your Safetensors adapter:

```
 FROM <base model name>
ADAPTER /path/to/safetensors/adapter/directory
```

Make sure that you use the same base model in the `FROM` command as you used to create the adapter otherwise you will get erratic results. Most frameworks use different quantization methods, so it's best to use non-quantized (i.e. non-QLoRA) adapters. If your adapter is in the same directory as your `Modelfile`, use `ADAPTER .` to specify the adapter path.

Now run `ollama create` from the directory where the `Modelfile` was created:

```
ollama create my-model
```

Lastly, test the model:

```
ollama run my-model
```

Ollama supports importing adapters based on several different model architectures including:

- Llama (including Llama 2, Llama 3, Llama 3.1, and Llama 3.2);
- Mistral (including Mistral 1, Mistral 2, and Mixtral); and
- Gemma (including Gemma 1 and Gemma 2)

You can create the adapter using a fine tuning framework or tool which can output adapters in the Safetensors format, such as:

- Hugging Face fine tuning framework
- Unsloth
- MLX

Importing a model from Safetensors weights

First, create a `Modelfile` with a `FROM` command which points to the directory containing your Safetensors weights:

```
FROM /path/to/safetensors/directory
```

If you create the Modelfile in the same directory as the weights, you can use the command `FROM .` .

If you do not create the Modelfile, ollama will act as if there was a Modelfile with the command `FROM .` .

Now run the `ollama create` command from the directory where you created the `Modelfile` :

```
ollama create my-model
```

Lastly, test the model:

```
ollama run my-model
```

Ollama supports importing models for several different architectures including:

- Llama (including Llama 2, Llama 3, Llama 3.1, and Llama 3.2);
- Mistral (including Mistral 1, Mistral 2, and Mixtral);
- Gemma (including Gemma 1 and Gemma 2); and
- Phi3

This includes importing foundation models as well as any fine tuned models which have been *fused* with a foundation model.

Importing a GGUF based model or adapter

If you have a GGUF based model or adapter it is possible to import it into Ollama. You can obtain a GGUF model or adapter by:

- converting a Safetensors model with the `convert_hf_to_gguf.py` from Llama.cpp;
- converting a Safetensors adapter with the `convert_lora_to_gguf.py` from Llama.cpp; or
- downloading a model or adapter from a place such as HuggingFace

To import a GGUF model, create a `Modelfile` containing:

```
FROM /path/to/file.gguf
```

For a GGUF adapter, create the `Modelfile` with:

```
 FROM <model name>
ADAPTER /path/to/file.gguf
```

When importing a GGUF adapter, it's important to use the same base model as the base model that the adapter was created with. You can use:

- a model from Ollama
- a GGUF file
- a Safetensors based model

Once you have created your `Modelfile`, use the `ollama create` command to build the model.

```
ollama create my-model
```

Quantizing a Model

Quantizing a model allows you to run models faster and with less memory consumption but at reduced accuracy. This allows you to run a model on more modest hardware.

Ollama can quantize FP16 and FP32 based models into different quantization levels using the `-q/--quantize` flag with the `ollama create` command.

First, create a Modelfile with the FP16 or FP32 based model you wish to quantize.

```
FROM /path/to/my/gemma/f16/model
```

Use `ollama create` to then create the quantized model.

```
 $ ollama create --quantize q4_K_M mymodel
transferring model data
quantizing F16 model to Q4_K_M
creating new layer sha256:735e246cc1abfd06e9cdcf95504d6789
creating new layer sha256:0853f0ad24e5865173bbf9ffcc7b0f5d
```

```
writing manifest
success
```

Supported Quantizations

- `q8_0`

K-means Quantizations

- `q4_K_S`
- `q4_K_M`

Sharing your model on ollama.com

You can share any model you have created by pushing it to ollama.com so that other users can try it out.

First, use your browser to go to the Ollama Sign-Up page. If you already have an account, you can skip this step.

Sign-Up

The `Username` field will be used as part of your model's name (e.g. `jmorganca/mymodel`), so make sure you are comfortable with the username that you have selected.

Now that you have created an account and are signed-in, go to the Ollama Keys Settings page.

Follow the directions on the page to determine where your Ollama Public Key is located.

Ollama Keys

Click on the `Add Ollama Public Key` button, and copy and paste the contents of your Ollama Public Key into the text field.

To push a model to ollama.com, first make sure that it is named correctly with your username. You may have to use the `ollama cp` command to copy your model to give it the correct name. Once you're happy with your model's name, use the `ollama push` command to push it to ollama.com.

```
 ollama cp mymodel myuser/mymodel
ollama push myuser/mymodel
```

Once your model has been pushed, other users can pull and run it by using the command:

```
ollama run myuser/mymodel
```

---

linux.md

Linux

Install

To install Ollama, run the following command:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Manual install

> [!NOTE] If you are upgrading from a prior version, you should remove the old libraries with `sudo rm -rf /usr/lib/ollama` first.

Download and extract the package:

```
 curl -LO https://ollama.com/download/ollama-linux-amd64.t
sudo tar -C /usr -xzf ollama-linux-amd64.tgz
```

Start Ollama:

```
ollama serve
```

In another terminal, verify that Ollama is running:

```
ollama -v
```

AMD GPU install

If you have an AMD GPU, also download and extract the additional ROCm package:

```
 curl -L https://ollama.com/download/ollama-linux-amd64-ro
sudo tar -C /usr -xzf ollama-linux-amd64-rocm.tgz
```

ARM64 install

Download and extract the ARM64-specific package:

```
 curl -L https://ollama.com/download/ollama-linux-arm64.tg
sudo tar -C /usr -xzf ollama-linux-arm64.tgz
```

Adding Ollama as a startup service (recommended)

Create a user and group for Ollama:

```
 sudo useradd -r -s /bin/false -U -m -d /usr/share/ollama 
sudo usermod -a -G ollama $(whoami)
```

Create a service file in `/etc/systemd/system/ollama.service`:

```
 [Unit]
Description=Ollama Service
After=network-online.target

[Service]
ExecStart=/usr/bin/ollama serve
User=ollama
Group=ollama
Restart=always
RestartSec=3
Environment="PATH=$PATH"
```

```
[Install]
WantedBy=multi-user.target
```

Then start the service:

```
 sudo systemctl daemon-reload
sudo systemctl enable ollama
```

Install CUDA drivers (optional)

Download and install CUDA.

Verify that the drivers are installed by running the following command, which should print details about your GPU:

```
 nvidia-smi
```

Install AMD ROCm drivers (optional)

Download and Install ROCm v6.

Start Ollama

Start Ollama and verify it is running:

```
 sudo systemctl start ollama
sudo systemctl status ollama
```

> [!NOTE] While AMD has contributed the `amdgpu` driver upstream to the official linux kernel source, the version is older and may not support all ROCm features. We recommend you install the latest driver from AMD for best support of your Radeon GPU.

Customizing

To customize the installation of Ollama, you can edit the systemd service file or the environment variables by running:

```
sudo systemctl edit ollama
```

Alternatively, create an override file manually in `/etc/systemd/system/ollama.service.d/override.conf`:

```
 [Service]
Environment="OLLAMA_DEBUG=1"
```

Updating

Update Ollama by running the install script again:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Or by re-downloading Ollama:

```
 curl -L https://ollama.com/download/ollama-linux-amd64.tg
sudo tar -C /usr -xzf ollama-linux-amd64.tgz
```

Installing specific versions

Use `OLLAMA_VERSION` environment variable with the install script to install a specific version of Ollama, including pre-releases. You can find the version numbers in the releases page.

For example:

```
curl -fsSL https://ollama.com/install.sh | OLLAMA_VERSION
```

Viewing logs

To view logs of Ollama running as a startup service, run:

```
journalctl -e -u ollama
```

Uninstall

Remove the ollama service:

```
sudo systemctl stop ollama
sudo systemctl disable ollama
sudo rm /etc/systemd/system/ollama.service
```

Remove the ollama binary from your bin directory (either `/usr/local/bin`, `/usr/bin`, or `/bin`):

```
sudo rm $(which ollama)
```

Remove the downloaded models and Ollama service user and group:

```
sudo rm -r /usr/share/ollama
sudo userdel ollama
sudo groupdel ollama
```

Remove installed libraries:

```
sudo rm -rf /usr/local/lib/ollama
```

---

macos.md

Ollama for macOS

System Requirements

- MacOS Monterey (v12) or newer
- Apple M series (CPU and GPU support) or x86 (CPU only)

Filesystem Requirements

The preferred method of installation is to mount the `ollama.dmg` and drag-and-drop the Ollama application to the system-wide `Applications` folder. Upon startup, the Ollama app will verify the `ollama` CLI is present in your PATH, and if not detected, will prompt for permission to create a link in `/usr/local/bin`

Once you've installed Ollama, you'll need additional space for storing the Large Language models, which can be tens to hundreds of GB in size. If your home directory doesn't have enough space, you can change where the binaries are installed, and where the models are stored.

Changing Install Location

To install the Ollama application somewhere other than `Applications`, place the Ollama application in the desired location, and ensure the CLI `Ollama.app/Contents/Resources/ollama` or a sym-link to the CLI can be found in your path. Upon first start decline the "Move to Applications?" request.

Troubleshooting

Ollama on MacOS stores files in a few different locations. - `~/.ollama` contains models and configuration - `~/.ollama/logs` contains logs - *app.log* contains most recent logs from the GUI application - *server.log* contains the most recent server logs - `<install location>/Ollama.app/Contents/Resources/ollama` the CLI binary

Uninstall

To fully remove Ollama from your system, remove the following files and folders:

```
 sudo rm -rf /Applications/Ollama.app
sudo rm /usr/local/bin/ollama
rm -rf "~/Library/Application Support/Ollama"
rm -rf "~/Library/Saved Application State/com.electron.oll
rm -rf ~/Library/Caches/com.electron.ollama/
rm -rf ~/Library/Caches/ollama
rm -rf ~/Library/WebKit/com.electron.ollama
rm -rf ~/.ollama
```

modelfile.md

Ollama Model File

> [!NOTE] `Modelfile` syntax is in development

A model file is the blueprint to create and share models with Ollama.

Table of Contents

Format

The format of the `Modelfile`:

```
 # comment
INSTRUCTION arguments
```

Instruction	Description
`FROM` (required)	Defines the base model to use.
`PARAMETER`	Sets the parameters for how Ollama will run the model.
`TEMPLATE`	The full prompt template to be sent to the model.
`SYSTEM`	Specifies the system message that will be set in the template.
`ADAPTER`	Defines the (Q)LoRA adapters to apply to the model.
`LICENSE`	Specifies the legal license.
`MESSAGE`	Specify message history.

Examples

Basic `Modelfile`

An example of a `Modelfile` creating a mario blueprint:

```
 FROM llama3.2
# sets the temperature to 1 [higher is more creative, lowe
PARAMETER temperature 1
# sets the context window size to 4096, this controls how
PARAMETER num_ctx 4096

# sets a custom system message to specify the behavior of
SYSTEM You are Mario from super mario bros, acting as an a
```

To use this:

1. Save it as a file (e.g. `Modelfile`)

2. `ollama create choose-a-model-name -f <location of the file e.g. ./Modelfile>`
3. `ollama run choose-a-model-name`
4. Start using the model!

To view the Modelfile of a given model, use the `ollama show --modelfile` command.

```
ollama show --modelfile llama3.2
```

> **Output**:
>
> ```
>
> ## Modelfile generated by "ollama show"
>
> ## To build a new Modelfile based on this one, replace the FROM line with:
>
> ## FROM llama3.2:latest
>
> FROM /Users/pdevine/.ollama/models/blobs/
> sha256-00e1317cbf74d901080d7100f57580ba8dd8de57203072dc6f668324ba545f29
> TEMPLATE """{{ if .System }}<|start_header_id|>system<|
> end_header_id|>
>
> {{ .System }}<|eot_id|>{{ end }}{{ if .Prompt }}<|start_header_id|
> >user<|end_header_id|>
>
> {{ .Prompt }}<|eot_id|>{{ end }}<|start_header_id|>assistant<|
> end_header_id|>

> {{ .Response }}<|eot_id|>""" PARAMETER stop "<|start_header_id|>" PARAMETER stop "<|end_header_id|>" PARAMETER stop "<|eot_id|>" PARAMETER stop "<|reserved_special_token" ```

Instructions

FROM (Required)

The `FROM` instruction defines the base model to use when creating a model.

```
FROM <model name>:<tag>
```

Build from existing model

```
FROM llama3.2
```

A list of available base models: https://github.com/ollama/ollama#model-library Additional models can be found at: https://ollama.com/library

Build from a Safetensors model

```
FROM <model directory>
```

The model directory should contain the Safetensors weights for a supported architecture.

Currently supported model architectures: * Llama (including Llama 2, Llama 3, Llama 3.1, and Llama 3.2) * Mistral (including Mistral 1, Mistral 2, and Mixtral) * Gemma (including Gemma 1 and Gemma 2) * Phi3

Build from a GGUF file

```
FROM ./ollama-model.gguf
```

The GGUF file location should be specified as an absolute path or relative to the `Modelfile` location.

PARAMETER

The `PARAMETER` instruction defines a parameter that can be set when the model is run.

```
PARAMETER <parameter> <parametervalue>
```

Valid Parameters and Values

Parameter	Description	Value Type	Example Usage
num_ctx	Sets the size of the context window used to generate the next token. (Default: 4096)	int	num_ctx 4096
repeat_last_n	Sets how far back for the model to look back to prevent repetition. (Default: 64, 0 = disabled, -1 = num_ctx)	int	repeat_last_n 64
repeat_penalty	Sets how strongly to penalize repetitions. A higher value (e.g.,	float	repeat_penalty 1.1

Parameter	Description	Value Type	Example Usage
	1.5) will penalize repetitions more strongly, while a lower value (e.g., 0.9) will be more lenient. (Default: 1.1)		
temperature	The temperature of the model. Increasing the temperature will make the model answer more creatively. (Default: 0.8)	float	temperature 0.7
seed	Sets the random number seed to use for generation. Setting this to a specific number will make the model generate the same text for the same prompt. (Default: 0)	int	seed 42
stop	Sets the stop sequences to use. When this pattern is encountered the LLM will stop generating text	string	stop "AI assistant:"

Parameter	Description	Value Type	Example Usage
	and return. Multiple stop patterns may be set by specifying multiple separate `stop` parameters in a modelfile.		
num_predict	Maximum number of tokens to predict when generating text. (Default: -1, infinite generation)	int	num_predict 42
top_k	Reduces the probability of generating nonsense. A higher value (e.g. 100) will give more diverse answers, while a lower value (e.g. 10) will be more conservative. (Default: 40)	int	top_k 40
top_p	Works together with top-k. A higher value (e.g., 0.95) will lead to more diverse text, while a lower value (e.g., 0.5)	float	top_p 0.9

Parameter	Description	Value Type	Example Usage
	will generate more focused and conservative text. (Default: 0.9)		
min_p	Alternative to the top_p, and aims to ensure a balance of quality and variety. The parameter *p* represents the minimum probability for a token to be considered, relative to the probability of the most likely token. For example, with *p*=0.05 and the most likely token having a probability of 0.9, logits with a value less than 0.045 are filtered out. (Default: 0.0)	float	min_p 0.05

TEMPLATE

`TEMPLATE` of the full prompt template to be passed into the model. It may include (optionally) a system message, a user's message and the response from the model. Note: syntax may be model specific. Templates use Go template syntax.

Template Variables

Variable	Description
`{{ .System }}`	The system message used to specify custom behavior.
`{{ .Prompt }}`	The user prompt message.
`{{ .Response }}`	The response from the model. When generating a response, text after this variable is omitted.

```
TEMPLATE """{{ if .System }}<|im_start|>system
{{ .System }}<|im_end|>
{{ end }}{{ if .Prompt }}<|im_start|>user
{{ .Prompt }}<|im_end|>
{{ end }}<|im_start|>assistant
"""
```

SYSTEM

The `SYSTEM` instruction specifies the system message to be used in the template, if applicable.

```
SYSTEM """<system message>"""
```

ADAPTER

The `ADAPTER` instruction specifies a fine tuned LoRA adapter that should apply to the base model. The value of the adapter should be an absolute path or a path relative to the Modelfile. The base model should be specified with a `FROM` instruction. If the base model is not the same as the base model that the adapter was tuned from the behaviour will be erratic.

Safetensor adapter

```
ADAPTER <path to safetensor adapter>
```

Currently supported Safetensor adapters: * Llama (including Llama 2, Llama 3, and Llama 3.1) * Mistral (including Mistral 1, Mistral 2, and Mixtral) * Gemma (including Gemma 1 and Gemma 2)

GGUF adapter

```
ADAPTER ./ollama-lora.gguf
```

LICENSE

The `LICENSE` instruction allows you to specify the legal license under which the model used with this Modelfile is shared or distributed.

```
 LICENSE """
<license text>
"""
```

MESSAGE

The `MESSAGE` instruction allows you to specify a message history for the model to use when responding. Use multiple iterations of the MESSAGE command to build up a conversation which will guide the model to answer in a similar way.

```
MESSAGE <role> <message>
```

Valid roles

Role	Description
system	Alternate way of providing the SYSTEM message for the model.
user	An example message of what the user could have asked.
assistant	An example message of how the model should respond.

Example conversation

```
 MESSAGE user Is Toronto in Canada?
MESSAGE assistant yes
MESSAGE user Is Sacramento in Canada?
MESSAGE assistant no
MESSAGE user Is Ontario in Canada?
MESSAGE assistant yes
```

Notes

- the **`Modelfile` is not case sensitive**. In the examples, uppercase instructions are used to make it easier to distinguish it from arguments.
- Instructions can be in any order. In the examples, the `FROM` instruction is first to keep it easily readable.

openai.md

OpenAI compatibility

> [!NOTE] OpenAI compatibility is experimental and is subject to major adjustments including breaking changes. For fully-featured access to the Ollama API, see the Ollama Python library, JavaScript library and REST API.

Ollama provides experimental compatibility with parts of the OpenAI API to help connect existing applications to Ollama.

Usage

OpenAI Python library

```
 from openai import OpenAI

client = OpenAI(
    base_url='http://localhost:11434/v1/',

    # required but ignored
    api_key='ollama',
)

chat_completion = client.chat.completions.create(
    messages=[
        {
            'role': 'user',
            'content': 'Say this is a test',
        }
    ],
    model='llama3.2',
)
```

```
response = client.chat.completions.create(
    model="llava",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "What's in this i
                {
                    "type": "image_url",
                    "image_url": "data:image/png;base64,iV
                },
            ],
        }
    ],
    max_tokens=300,
)

completion = client.completions.create(
    model="llama3.2",
    prompt="Say this is a test",
)

list_completion = client.models.list()

model = client.models.retrieve("llama3.2")

embeddings = client.embeddings.create(
    model="all-minilm",
    input=["why is the sky blue?", "why is the grass green
)
```

Structured outputs

```
 from pydantic import BaseModel
from openai import OpenAI
```

```
client = OpenAI(base_url="http://localhost:11434/v1", api_

# Define the schema for the response
class FriendInfo(BaseModel):
    name: str
    age: int
    is_available: bool

class FriendList(BaseModel):
    friends: list[FriendInfo]

try:
    completion = client.beta.chat.completions.parse(
        temperature=0,
        model="llama3.1:8b",
        messages=[
            {"role": "user", "content": "I have two friend
        ],
        response_format=FriendList,
    )

    friends_response = completion.choices[0].message
    if friends_response.parsed:
        print(friends_response.parsed)
    elif friends_response.refusal:
        print(friends_response.refusal)
except Exception as e:
    print(f"Error: {e}")
```

OpenAI JavaScript library

```
 import OpenAI from 'openai'

const openai = new OpenAI({
  baseURL: 'http://localhost:11434/v1/',
  // required but ignored
  apiKey: 'ollama',
})

const chatCompletion = await openai.chat.completions.creat
    messages: [{ role: 'user', content: 'Say this is a tes
    model: 'llama3.2',
})

const response = await openai.chat.completions.create({
    model: "llava",
    messages: [
        {
        role: "user",
        content: [
            { type: "text", text: "What's in this image?"
            {
            type: "image_url",
            image_url: "data:image/png;base64,iVBORw0KGgoA
            },
        ],
        },
    ],
})

const completion = await openai.completions.create({
    model: "llama3.2",
    prompt: "Say this is a test.",
})

const listCompletion = await openai.models.list()

const model = await openai.models.retrieve("llama3.2")

const embedding = await openai.embeddings.create({
  model: "all-minilm",
  input: ["why is the sky blue?", "why is the grass green?
})
```

`curl`

```
 curl http://localhost:11434/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "llama3.2",
        "messages": [
            {
                "role": "system",
                "content": "You are a helpful assistant."
            },
            {
                "role": "user",
                "content": "Hello!"
            }
        ]
    }'

curl http://localhost:11434/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "llava",
    "messages": [
      {
        "role": "user",
        "content": [
          {
            "type": "text",
            "text": "What'\''s in this image?"
          },
          {
            "type": "image_url",
            "image_url": {
               "url": "data:image/png;base64,iVBORw0KGgoAA
```

```
              }
            }
          ]
        }
      ],
      "max_tokens": 300
    }'

curl http://localhost:11434/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "llama3.2",
        "prompt": "Say this is a test"
    }'

curl http://localhost:11434/v1/models

curl http://localhost:11434/v1/models/llama3.2

curl http://localhost:11434/v1/embeddings \
    -H "Content-Type: application/json" \
    -d '{
        "model": "all-minilm",
        "input": ["why is the sky blue?", "why is the gras
    }'
```

Endpoints

`/v1/chat/completions`

Supported features

- [x] Chat completions
- [x] Streaming
- [x] JSON mode
- [x] Reproducible outputs
- [x] Vision

- [x] Tools
- [ ] Logprobs

Supported request fields

- [x] `model`
- [x] `messages`
- [x] Text `content`
- [x] Image `content`
  - [x] Base64 encoded image
  - [ ] Image URL
- [x] Array of `content` parts
- [x] `frequency_penalty`
- [x] `presence_penalty`
- [x] `response_format`
- [x] `seed`
- [x] `stop`
- [x] `stream`
- [x] `stream_options`
- [x] `include_usage`
- [x] `temperature`
- [x] `top_p`
- [x] `max_tokens`
- [x] `tools`
- [ ] `tool_choice`
- [ ] `logit_bias`
- [ ] `user`
- [ ] `n`

`/v1/completions`

Supported features

- [x] Completions
- [x] Streaming
- [x] JSON mode
- [x] Reproducible outputs
- [ ] Logprobs

Supported request fields

- [x] `model`
- [x] `prompt`
- [x] `frequency_penalty`
- [x] `presence_penalty`
- [x] `seed`
- [x] `stop`
- [x] `stream`
- [x] `stream_options`
- [x] `include_usage`
- [x] `temperature`
- [x] `top_p`
- [x] `max_tokens`
- [x] `suffix`
- [ ] `best_of`
- [ ] `echo`
- [ ] `logit_bias`
- [ ] `user`
- [ ] `n`

Notes

- `prompt` currently only accepts a string

`/v1/models`

Notes

- `created` corresponds to when the model was last modified
- `owned_by` corresponds to the ollama username, defaulting to `"library"`

`/v1/models/{model}`

Notes

- `created` corresponds to when the model was last modified

- `owned_by` corresponds to the ollama username, defaulting to `"library"`

`/v1/embeddings`

Supported request fields

- [x] `model`
- [x] `input`
- [x] string
- [x] array of strings
- [ ] array of tokens
- [ ] array of token arrays
- [ ] `encoding format`
- [ ] `dimensions`
- [ ] `user`

Models

Before using a model, pull it locally `ollama pull`:

```
ollama pull llama3.2
```

Default model names

For tooling that relies on default OpenAI model names such as `gpt-3.5-turbo`, use `ollama cp` to copy an existing model name to a temporary name:

```
ollama cp llama3.2 gpt-3.5-turbo
```

Afterwards, this new model name can be specified the `model` field:

```
curl http://localhost:11434/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "gpt-3.5-turbo",
        "messages": [
            {
                "role": "user",
                "content": "Hello!"
            }
        ]
    }'
```

Setting the context size

The OpenAI API does not have a way of setting the context size for a model. If you need to change the context size, create a `Modelfile` which looks like:

```
 FROM <some model>
PARAMETER num_ctx <context size>
```

Use the `ollama create mymodel` command to create a new model with the updated context size. Call the API with the updated model name:

```
curl http://localhost:11434/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "mymodel",
        "messages": [
            {
                "role": "user",
                "content": "Hello!"
            }
        ]
    }'
```

template.md

Template

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<|en

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

{{ .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
{{- end }}[INST] {{ if and (eq (len (slice $.Messages $ind

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

> [!NOTE] 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 }}
```

---

troubleshooting.md

How to troubleshoot issues

Sometimes Ollama may not perform as expected. One of the best ways to figure out what happened is to take a look at the logs. Find the logs on **Mac** by running the command:

```
cat ~/.ollama/logs/server.log
```

On **Linux** systems with systemd, the logs can be found with this command:

```
journalctl -u ollama --no-pager --follow --pager-end
```

When you run Ollama in a **container**, the logs go to stdout/stderr in the container:

```
docker logs <container-name>
```

(Use `docker ps` to find the container name)

If manually running `ollama serve` in a terminal, the logs will be on that terminal.

When you run Ollama on **Windows**, there are a few different locations. You can view them in the explorer window by hitting `<cmd>+R` and type in: - `explorer %LOCALAPPDATA%\Ollama` to view logs. The most recent server logs will be in `server.log` and older logs will be in `server-#.log` - `explorer %LOCALAPPDATA%\Programs\Ollama` to browse the binaries (The installer adds this to your user PATH) - `explorer %HOMEPATH%\.ollama` to browse where models and configuration is stored

To enable additional debug logging to help troubleshoot problems, first **Quit the running app from the tray menu** then in a powershell terminal

```
 $env:OLLAMA_DEBUG="1"
& "ollama app.exe"
```

Join the Discord for help interpreting the logs.

LLM libraries

Ollama includes multiple LLM libraries compiled for different GPUs and CPU vector features. Ollama tries to pick the best one based on the capabilities of your system. If this autodetection has problems, or you run into other problems (e.g. crashes in your GPU) you can workaround this by forcing a specific LLM library. `cpu_avx2` will perform the best, followed by `cpu_avx` and the slowest but most compatible is `cpu`. Rosetta emulation under MacOS will work with the `cpu` library.

In the server log, you will see a message that looks something like this (varies from release to release):

```
Dynamic LLM libraries [rocm_v6 cpu cpu_avx cpu_avx2 cuda_
```

Experimental LLM Library Override

You can set OLLAMA_LLM_LIBRARY to any of the available LLM libraries to bypass autodetection, so for example, if you have a CUDA card, but want to force the CPU LLM library with AVX2 vector support, use:

```
OLLAMA_LLM_LIBRARY="cpu_avx2" ollama serve
```

You can see what features your CPU has with the following.

```
cat /proc/cpuinfo| grep flags | head -1
```

Installing older or pre-release versions on Linux

If you run into problems on Linux and want to install an older version, or you'd like to try out a pre-release before it's officially released, you can tell the install script which version to install.

```
curl -fsSL https://ollama.com/install.sh | OLLAMA_VERSION=
```

Linux docker

If Ollama initially works on the GPU in a docker container, but then switches to running on CPU after some period of time with errors in the server log reporting GPU discovery failures, this can be resolved by disabling systemd cgroup management in Docker. Edit `/etc/docker/daemon.json` on the host and add `"exec-opts": ["native.cgroupdriver=cgroupfs"]` to the docker configuration.

NVIDIA GPU Discovery

When Ollama starts up, it takes inventory of the GPUs present in the system to determine compatibility and how much VRAM is available. Sometimes this discovery can fail to find your GPUs. In general, running the latest driver will yield the best results.

Linux NVIDIA Troubleshooting

If you are using a container to run Ollama, make sure you've set up the container runtime first as described in docker.md

Sometimes the Ollama can have difficulties initializing the GPU. When you check the server logs, this can show up as various error codes, such as "3" (not initialized), "46" (device unavailable), "100" (no device), "999" (unknown), or others. The following troubleshooting techniques may help resolve the problem

- If you are using a container, is the container runtime working? Try `docker run --gpus all ubuntu nvidia-smi` - if this doesn't work, Ollama won't be able to see your NVIDIA GPU.
- Is the uvm driver loaded? `sudo nvidia-modprobe -u`
- Try reloading the nvidia_uvm driver - `sudo rmmod nvidia_uvm` then `sudo modprobe nvidia_uvm`
- Try rebooting
- Make sure you're running the latest nvidia drivers

If none of those resolve the problem, gather additional information and file an issue: - Set `CUDA_ERROR_LEVEL=50` and try again to get more diagnostic logs - Check dmesg for any errors `sudo dmesg | grep -i nvrm` and `sudo dmesg | grep -i nvidia`

AMD GPU Discovery

On linux, AMD GPU access typically requires `video` and/or `render` group membership to access the `/dev/kfd` device. If permissions are not set up correctly, Ollama will detect this and report an error in the server log.

When running in a container, in some Linux distributions and container runtimes, the ollama process may be unable to access the GPU. Use `ls -lnd /dev/kfd /dev/dri /dev/dri/*` on the host system to determine the **numeric** group IDs on your system, and pass additional `--group-add ...` arguments to the container so it can access the required devices. For example, in the following output `crw-rw---- 1 0 44 226, 0 Sep 16 16:55 /dev/dri/card0` the group ID column is `44`

If you are experiencing problems getting Ollama to correctly discover or use your GPU for inference, the following may help isolate the failure. - `AMD_LOG_LEVEL=3` Enable info log levels in the AMD HIP/ROCm libraries. This can help show more detailed error codes that can help troubleshoot problems - `OLLAMA_DEBUG=1` During GPU discovery additional information will be reported - Check dmesg for any errors from amdgpu or kfd drivers `sudo dmesg | grep -i amdgpu` and `sudo dmesg | grep -i kfd`

Multiple AMD GPUs

If you experience gibberish responses when models load across multiple AMD GPUs on Linux, see the following guide.

- https://rocm.docs.amd.com/projects/radeon/en/latest/docs/install/native_linux/mgpu.html#mgpu-known-issues-and-limitations

Windows Terminal Errors

Older versions of Windows 10 (e.g., 21H1) are known to have a bug where the standard terminal program does not display control characters correctly. This can result in a long string of strings like `←[?25h←[?25l` being displayed, sometimes erroring with `The parameter is incorrect` To resolve this problem, please update to Win 10 22H1 or newer.

---

windows.md

Ollama Windows

Welcome to Ollama for Windows.

No more WSL required!

Ollama now runs as a native Windows application, including NVIDIA and AMD Radeon GPU support. After installing Ollama for Windows, Ollama will run in the background and the `ollama` command line is available in `cmd`, `powershell` or your favorite terminal application. As usual the Ollama api will be served on `http://localhost:11434`.

System Requirements

- Windows 10 22H2 or newer, Home or Pro
- NVIDIA 452.39 or newer Drivers if you have an NVIDIA card
- AMD Radeon Driver https://www.amd.com/en/support if you have a Radeon card

Ollama uses unicode characters for progress indication, which may render as unknown squares in some older terminal fonts in Windows 10. If you see this, try changing your terminal font settings.

Filesystem Requirements

The Ollama install does not require Administrator, and installs in your home directory by default. You'll need at least 4GB of space for the binary install. Once you've installed Ollama, you'll need additional space for storing the Large Language models, which can be tens to hundreds of GB in size. If your home directory doesn't have enough space, you can change where the binaries are installed, and where the models are stored.

Changing Install Location

To install the Ollama application in a location different than your home directory, start the installer with the following flag

```
OllamaSetup.exe /DIR="d:\some\location"
```

API Access

Here's a quick example showing API access from `powershell`

```
(Invoke-WebRequest -method POST -Body '{"model":"llama3.2
```

Troubleshooting

Ollama on Windows stores files in a few different locations. You can view them in the explorer window by hitting `<Ctrl>+R` and type in: - `explorer %LOCALAPPDATA%\Ollama` contains logs, and downloaded updates - *app.log* contains most resent logs from the GUI application - *server.log* contains the most recent server logs - *upgrade.log* contains log output for upgrades - `explorer %LOCALAPPDATA%\Programs\Ollama` contains the binaries (The installer adds this to your user PATH) - `explorer %HOMEPATH%\.ollama` contains models and configuration

Uninstall

The Ollama Windows installer registers an Uninstaller application. Under `Add or remove programs` in Windows Settings, you can uninstall Ollama.

> [!NOTE] If you have changed the OLLAMA_MODELS location, the installer will not remove your downloaded models

Standalone CLI

The easiest way to install Ollama on Windows is to use the `OllamaSetup.exe` installer. It installs in your account without requiring Administrator rights. We update Ollama regularly to support the latest models, and this installer will help you keep up to date.

If you'd like to install or integrate Ollama as a service, a standalone `ollama-windows-amd64.zip` zip file is available containing only the Ollama CLI and GPU library dependencies for Nvidia. If you have an AMD GPU, also download and extract the additional ROCm package `ollama-windows-amd64-rocm.zip` into the same directory. This allows for embedding Ollama in existing applications, or running it as a system service via `ollama serve` with tools such as NSSM.

> [!NOTE]
> If you are upgrading from a prior version, you should remove the old directories first.
