Building Oasis: a Local Code Generator Tool Using Open Source Models and Microsoft's Guidance Library

Local LLamas in VSCode

Paolo Rechia · Follow
Published in Better Programming
14 min read · 3 days ago

Listen Share More

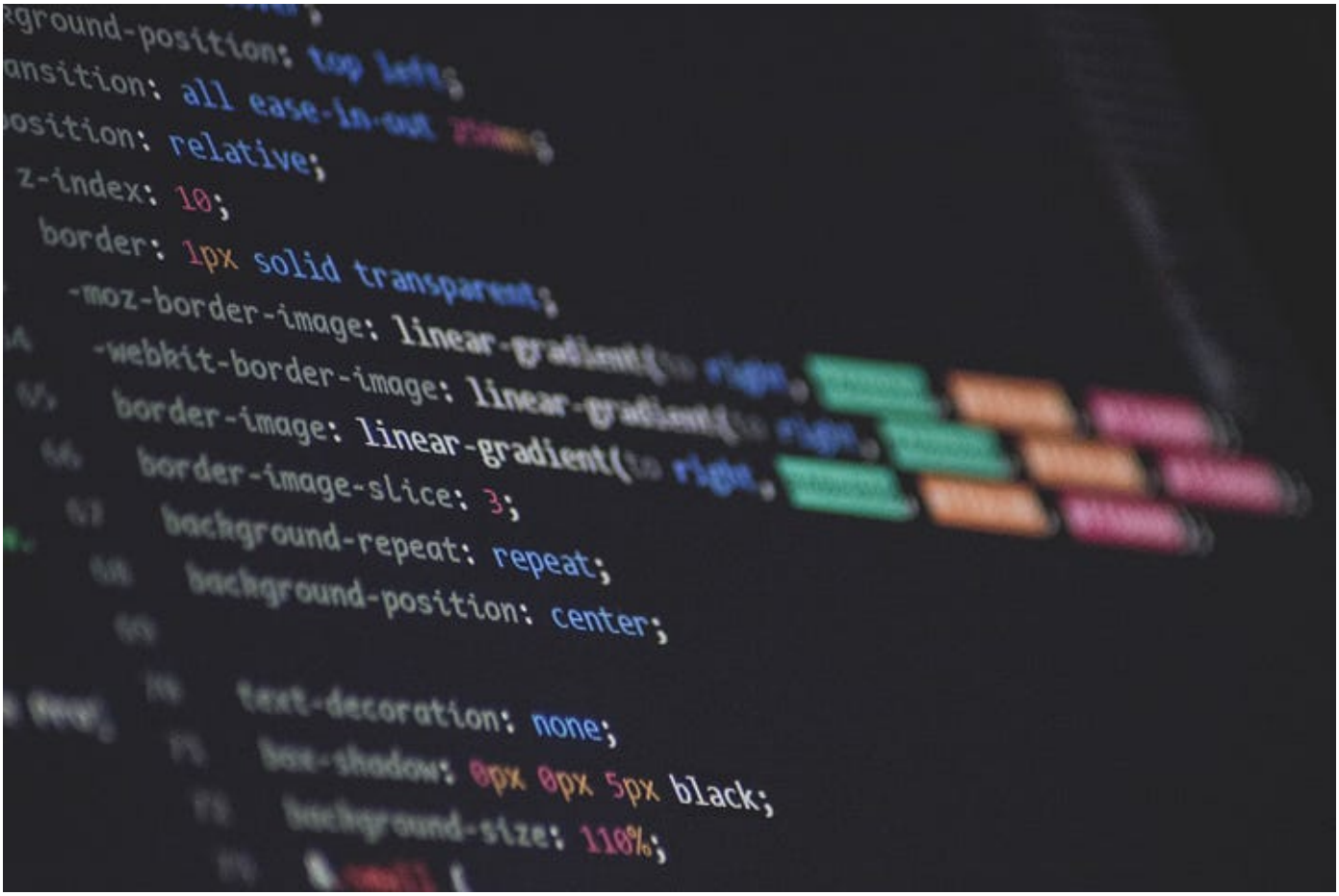


Photo by Maik Jonietz on Unsplash

A while ago, I wrote a VSCode plugin using the now-deprecated Codex API from OpenAI, and it worked fairly well to perform simple instructions on the selected code in VSCode. For instance, it could add a docstring to the function that I selected.

As I've been fiddling a lot with local LLMs (Large Language Models), the next step was naturally to see how I could build something similar that does not depend on OpenAI.

For this purpose, I started the oasis project. When I first started, the idea was super simple: just calling directly the `text-generation-webui` API, to generate docstrings for my own code.

It didn't take me long to reach similar results from my older plugin, but now using WizardLM instead of OpenAI. That was pretty exciting, however, this model is still limited in terms of license: it's only meant for research.

And this time I wanted something that I could use it for commercial purposes or for my own work routine, without having trouble explaining myself about the model license.

A while ago, I wrote about a quick PoC that I built using Salesforce Codegen model on CPU to generate code. Back then, I didn't bother connecting it to VSCode, mostly because this is a model suitable for completion, but not for instruction following.

As I spent more time writing prompts these days and became familiar with Microsoft's guidance library, it occurred to me I could create a viable product by combining these two things:

1. Salesforce Codegen for generating code
2. Microsoft's guidance library to steer the model into generating what I actually need, instead of random input

So I spent a few days building a new plugin version, that combines this approach, and I was quite happy with the initial results. Admittedly, the solution became much more complicated than I anticipated, but it works fairly well! And more importantly, we can use even the 350 million parameters model version and still generate correctly some docstrings!

Of course, it does not always work, and bigger models still tend to work much better — but using smaller models gives us a huge improvement in terms of accessibility and inference performance — almost anyone can spin up the 350m model version.

The current version of this plugin is available on my GitHub.

Today, we'll explore how this plugin was built:

1. Writing the VSCode extension.
2. Setting up an HTTP server with the guidance library.
3. Setting up a middleware prompt server to parse code and pick prompts dynamically.
4. The details of making a command like `add_docstring` work effectively with less powerful models. This will include how to write a guidance prompt, and make sure the input/output looks like expected. This last section will be further subdivided into:

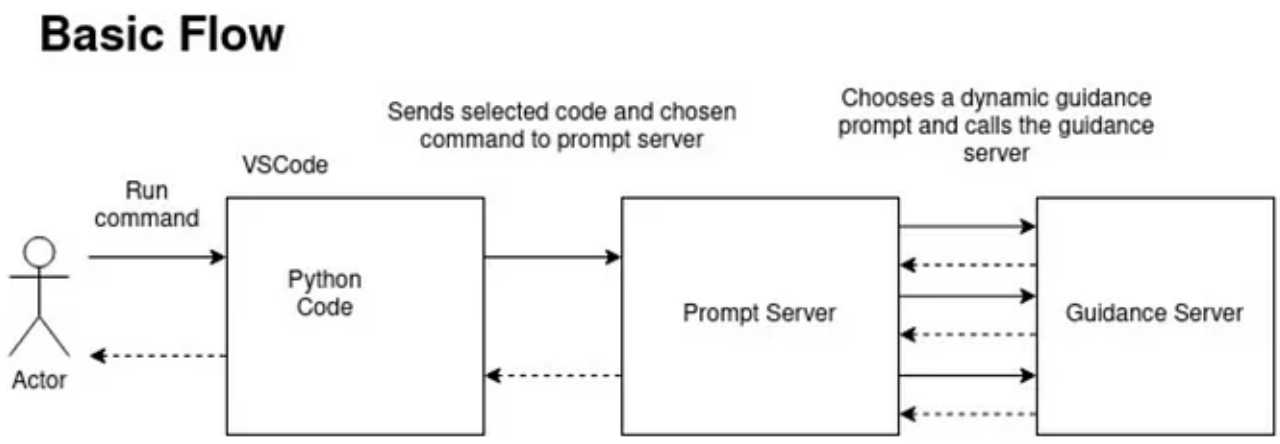
a) Parsing the Input

b) The Add Docstring Guidance Prompt

c) Building the Output

Note: the docstring format is somewhat simple. Nothing stops us of improving the generated format.

Here is a high-level diagram of what we're building:



Overview of the basic flow

The Basics Of Writing a VSCode Extension

The first step is to follow the official documentation hello world. That gives you a nice bootstrapped sample extension.

Then, we can modify our `extension.ts` to add the main logic. The first step is to define how we'll initialize our extension. Upon initialization, we'll register the existing commands. Here's the relevant code:

```
export function activate(context: vscode.ExtensionContext) {
 const commands = [
  ["addDocstring", "add_docstring"],
  ["addTypeHints", "add_type_hints"],
  ["addUnitTest", "add_unit_test"],
  ["fixSyntaxError", "fix_syntax_error"],
  ["customPrompt", "custom_prompt"],
 ];
 commands.forEach(tuple_ => {
  const [commandName, oasisCommand] = tuple_;
  const command = vscode.commands.registerCommand(`oasis.${commandName}`, () => {
   useoasis(oasisCommand);
  });
  context.subscriptions.push(command);
 });
}
```

This binds each expected command to a call to the main logic function, called `useoasis`. We'll go over this function later.

These commands must match what's defined in the project `package.json` —we will define a few blocks to declare the commands exposed by our extension — let's start by defining the `contributes.commands`:

```
[
      {
        "command": "oasis.addDocstring",
        "title": "Add Docstring to Selection"
      },
      {
        "command": "oasis.addTypeHints",
        "title": "Add type hints to selection"
      },
      {
        "command": "oasis.fixSyntaxError",
        "title": "Fix syntax error"
      },
      {
        "command": "oasis.customPrompt",
        "title": "Custom Prompt"
      }
]
```

We also want that the user is able to select a block of code with the mouse find this command with a right-click, so we use the configuration `contributes.menus.editor/context`, like this:

```
"editor/context": [
  {
    "command": "oasis.addDocstring",
    "when": "editorHasSelection",
    "group": "7_modification"
  },
  {
    "command": "oasis.addTypeHints",
    "when": "editorHasSelection",
    "group": "7_modification"
```

```
    },
    {
      "command": "oasis.fixSyntaxError",
      "when": "editorHasSelection",
      "group": "7_modification"
    },
    {
      "command": "oasis.customPrompt",
      "when": "editorHasSelection",
      "group": "7_modification"
    }
  ]
```

Alright, that's enough to get started with binding commands. Let's go back and look at our main extension function:

```
      body: requestBody,
      timeout: {
       request: 300000  // 5 minutes max
      }
     }).json();
    } catch (e: any) {
     vscode.window.showErrorMessage("Oasis Plugin: error calling the API")
     try {
      const apiStatusCode = `Error calling API: ${e.response.statusCode}`;
      vscode.window.showErrorMessage(apiStatusCode);
     } catch (error) {
      console.error("Error parsing error response code", error);
     }
    }

    if (response) {
     console.log("From got", response.text);
     const editedText = response.text;

     activeEditor.edit(editBuilder => {
      console.log("Edit builder", editBuilder);
      editBuilder.replace(selection, editedText);
     });
    }
   };
```

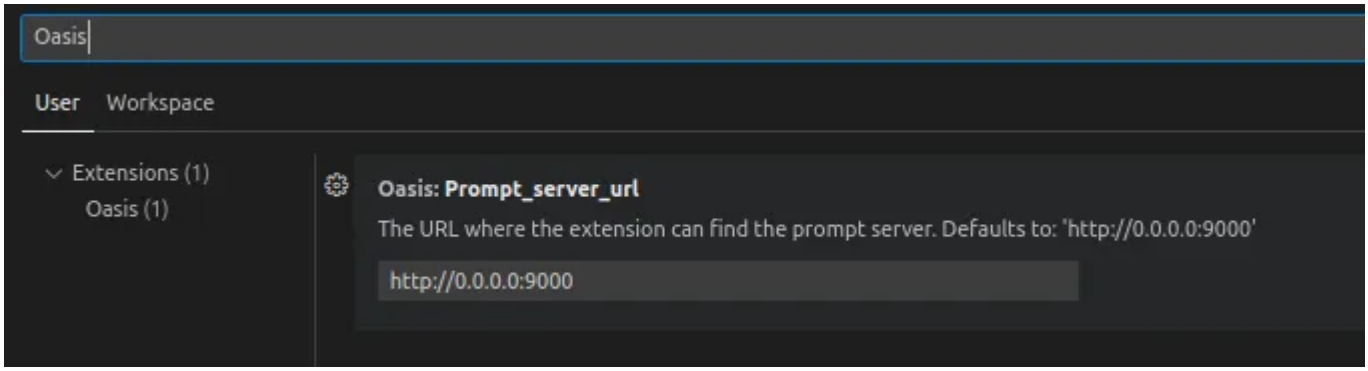
This is a bit longer, but still fairly simple. First, we check there's an active editor to avoid referencing a null. Here's a simple trick (slightly reformatted):

```
const oasisUrl = vscode.workspace
  .getConfiguration('oasis')
  .get("prompt_server_url")
```

This reads the workspace configuration to read where the prompt server is being served. So we must go back to the extension config (`package.json`), and add the following:

```
  "contributes": {
    "configuration": {
      "title": "Oasis",
      "properties": {
        "oasis.prompt_server_url": {
          "type": "string",
          "default": "http://0.0.0.0:9000",
          "description": "The URL where the extension can find the prompt server.
        }
      }
    },
```

This not only defines a default value, but also exposes the config so the user can override it, either through the UI or through the JSON editor. Here's a sample screenshot of the UI:



Let's go back to the extension code:

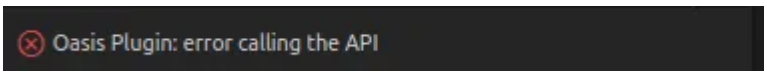
```
const text = document.getText(selection);

const requestBody = JSON.stringify({
data: text
});
const url = `${oasisUrl}/${command}`;

console.log("Calling API", url, "with body: ", requestBody);
```

Fairly simple, we read the selected text in the editor and then prepare the request. Recall that `command` comes as a closure parameter that is built in the `activate` function.

We use the `got` library here (for no good reason), and make the request:

```
let response: oasisResponse | undefined = undefined;
try {
 response = await got(url, {
  method: "POST",
  headers: {
   // eslint-disable-next-line @typescript-eslint/naming-convention
   "Content-Type": "application/json",
   // eslint-disable-next-line @typescript-eslint/naming-convention
  },
  body: requestBody,
  timeout: {
   request: 300000  // 5 minutes max
  }
 }).json();
} catch (e: any) {
 vscode.window.showErrorMessage("Oasis Plugin: error calling the API")
 try {
  const apiStatusCode = `Error calling API: ${e.response.statusCode}`;
  vscode.window.showErrorMessage(apiStatusCode);
 } catch (error) {
  console.error("Error parsing error response code", error);
 }
}
```

If the request fails, we display an error in the UI, which is displayed like this:



Not very informative, but at least the user knows that he shouldn't wait anymore. Then if everything works, we parse and then use the active editor to replace the selection with the response from the API:

```
  if (response) {
   console.log("From got", response.text);
   const editedText = response.text;

   activeEditor.edit(editBuilder => {
    console.log("Edit builder", editBuilder);
    editBuilder.replace(selection, editedText);
   });
  }
```

The Guidance Library: Setting Up an HTTP Server With the Guidance Library

Guidance is a very interesting library from Microsoft to work with LLMs. I recommend a quick read through its Readme.

The main advantage of using guidance is that you can perform specific tasks with models that are much less powerful at following instructions.

The downside, as we'll see later, is that it does add a significant development overhead since we resolve a lot of the logical steps for the model using a standard programming language.

When developing with guidance, it can be really cumbersome to reload the server, especially if the used model is heavy. For that reason, I wrapped the model loading and the guidance library inside a server that does not need to change often.

This server's source code is really simple, and fits into a single file:

```
model = "Salesforce/codegen-350m-mono"
# model = "Salesforce/codegen-2b-mono"
# model = "Salesforce/codegen-6b-mono"
# model = "Salesforce/codegen-16B-mono"

llama = guidance.llms.Transformers(model, quantization_config=nf4_config, revisio
print("Server loaded!")


@app.post("/")
def call_llama(request: Request):
    input_vars = request.input_vars
    kwargs = request.guidance_kwargs
    output_vars = request.output_vars

    guidance_program: Program = guidance(request.prompt_template)
    program_result = guidance_program(
        **kwargs,
        stream=False,
        async_mode=False,
        caching=False,
        **input_vars,
        llm=llama,
    )
    output = {}
    for output_var in output_vars:
        output[output_var] = program_result[output_var]
    return output
```

That's it! It takes the prompt template, with the input variables and expected output variables from an HTTP request, and routes it through the guidance library. Then it extracted the expected output variables and returns in the HTTP response.

If you're curious about the `quantization_config`, I'm using the latest 4-bit quantization technique released by Hugging Face.

We'll also need a client to use this server:

```
    if input_vars is None:
        input_vars = {}
    if guidance_kwargs is None:
        guidance_kwargs = {}

    data = {
        "prompt_template": prompt_template,
        "output_vars": output_vars,
        "guidance_kwargs": guidance_kwargs,
        "input_vars": input_vars,
    }

    response = requests.post(
        guidance_url,
        json=data
    )

    response.raise_for_status()

    return response.json()
```

Nothing too interesting to see here.

Note that this server does not communicate directly with the VScode extension.

Setting Up a Middleware Prompt Server to Parse Code and Pick Prompts Dynamically

Now let's see the middleware that actually receives the request from the VSCode. Let's jump into its main server code:

```
    logger.info("parsed output: '%s'", result)

    return {"text": result}
```

This may look simple, but it's hiding a lot of nasty details. The overall idea of the controller though, is indeed simple:

1. Take an input (command, input code) pair
2. Find an appropriate prompt
3. Parse the input into a specific format
4. Run through the guidance server
5. Parse the output into the right format
6. Reply back

The Details of the 'add_docstring' Command

Parsing the input

Alright, we've gone through the simple parts. Now let's look at the more complex parts. Let's expand each step above.

First, we find out which command we're requested to apply:

```
logger.info("Received command: '%s'", command)
logger.debug("Received data: '%s'", request)
received_code = request.data
try:
    command_to_apply = commands_mapping[command]
    logger.info("Loaded command: '%s'", command_to_apply)
```

This is a dictionary built with this call:

```
commands_mapping = build_command_mapping(prompts_module)
```

Here we define the available commands. Notice that only one command is available for now:

```
def build_command_mapping(prompt_module: PromptModuleInterface):
    add_docstring_command = DocStringCommand(
        prompt={
            "generic_prompt": prompt_module.doc_string_guidance_prompt,
            "function_prompt": prompt_module.function_doc_string_guidance_prompt
        }
    )
    commands_mapping = {
          "add_docstring": add_docstring_command,
          # "add_type_hints": ADD_TYPE_HINTS,
          # "fix_syntax_error": FIX_SYNTAX_ERROR,
          # "custom_prompt": CUSTOM_PROMPT
    }

  return commands_mapping
```

The class `DocStringCommand` implements the abstract `Command` class:

```
        def __init__(self, *args, **kwargs):
            super().__init__(*args,**kwargs)
```

So this class must hold a dictionary of guidance prompts, and implement two functions, so that it can choose the correct prompt and extract the output. The `GuidancePrompt` will look familiar if you recall how our guidance server is built:

```
@dataclass
class GuidancePrompt:
    prompt_template: str
    input_vars: Dict[str, str]
    output_vars: Dict[str, str]
    guidance_kwargs: Dict[str, str]
```

Let's see how we choose the prompt:

```
def prompt_picker(self, input_: str) -> Tuple[str, GuidancePrompt, Dict[str, str]]
    prompt_key = "None"
    try:
        function_header, function_body, leading_indentation, indentation_type = fu
        prompt_key = "function_prompt"
        return_value = prompt_key, self.prompt[prompt_key], {
            "function_header": function_header,
            "function_body": function_body,
            "leading_indentation": leading_indentation,
            "indentation_type": indentation_type
        }

    except (FailedToParseFunctionException):
        logger.warn("Failed to identify specific type of code block, falling back
        prompt_key = "generic_prompt"
        return_value = prompt_key, self.prompt[prompt_key], {"input": input_}

    logger.info("Chosen prompt: %s", prompt_key)
    return return_value
```

So if you noticed, there are two prompts here: `generic_prompt` and `function_prompt`. If we identify a function, we apply the later, specialized one — else we fallback into the generic.

The returned value is a tuple of the format `(prompt_key, prompt, input_dict)`. We'll use these values later on when calling the LLM with this prompt and when parsing the LLM output.

Going back a bit, if we look at how we identify if it's a function, we're calling the `function_parser`. Let's go over this rabbit hole:

```
def function_parser(input_code_str: str) -> Tuple[str, str, str]:
    leading_indentation = _get_leading_indentation(input_code_str)
    simple_indented_code = _remove_extra_indentation(input_code_str, leading_inde
    indentation_type = _get_indentation_type(input_code_str)

    parsed = ast.parse(simple_indented_code, filename="<string>")
    parsed
    try:
        first_node = parsed.body[0]
    except IndexError:
        raise FailedToParseFunctionException from IndexError

    if not isinstance(first_node, ast.FunctionDef):
        raise FailedToParseFunctionException(f"Parsed type is not a function: '{ty

    function_body = _extract_function_body(first_node, leading_indentation, inden
    function_header = _extract_function_header(first_node)
    return function_header, function_body, leading_indentation, indentation_type
```

So the basic idea here is to use the `ast.parse` (documentation) so we can look at the AST and find the function body/header. We also do some weird magic with indentation, why?

Well, if you try passing the following string to `ast.parse`:

```
"""
  def hello():
    print('hello!')
"""
```

You'll be greeted with a nice `IndentationError`. Why? I'm not completely sure, but it seems that this parser assumes it's always reading a file. This means that this string is interpreted as a module-level source code, and thus has the wrong indentation.

Working around this led me to several workarounds with the indentation, which I'll not cover in detail — you can always dive in the full source code if you're curious.

So let's backtrack a bit — this parser gave us the function header and the function body. With this, we can now look at the guidance prompt we have prepared, which we will use soon.

The Docstring Guidance Prompt

Before we start, let's use one of the functions from Oasis to see how this generation works. Given this function as input:

```
def _extract_function_header(fun_code: ast.FunctionDef) -> str:
    full = ast.unparse(fun_code)
    body = ast.unparse(fun_code.body)
    return full.split(body[0:USED_BODY_CHARS_TO_SPLIT])[0].strip()
```

This is the docstring generated by Oasis:

```
    """

    full = ast.unparse(fun_code)
    body = ast.unparse(fun_code.body)
    return full.split(body[0:USED_BODY_CHARS_TO_SPLIT])[0].strip()
```

So how is it generated? The prompt is somewhat long, so let's look at it in blocks. First, we start providing examples with the good and old few-shot prompting technique.

```
function_doc_string_guidance_prompt = GuidancePrompt(
    prompt_template="""
def print_hello_world():
    print("hello_world")

def print_hello_world():
# Docstring below
    \"\"\"This functions print the string 'hello_world' to the standard output.\"

def sum_2(x, y):
    return x + y

def sum_2(x, y):
# Docstring below
    \"\"\"This functions receives two parameters and returns the sum.

    Parameters:
        int: x - first number to sum
        int: y - second number to sum

    Returns:
        int: sum of the two given integers
    \"\"\"
    return x + y
```

Notice that we don't explicitly give an instruction to the model, unlike what we do with WizardLM models. Here we simply provide patterns so the model can autocomplete in a similar way. After a few examples, we define how the generation should be done:

```
    """
    (few shot examples above...)

    {{leading_indentation}}{{function_header}}
    {{function_body}}

    {{leading_indentation}}{{function_header}}
    # Docstring below
    {{leading_indentation}}\"\"\"{{gen 'description' temperature=0.1 max_tokens=128 s

    Parameters: {{gen 'parameters' temperature=0.1 max_tokens=128 stop='Returns:'}}
    Returns: {{gen 'returns' temperature=0.1 max_tokens=128 stop='\"\"\"'}}
    """,
        guidance_kwargs={},
        input_vars=["function_header", "function_body", "leading_indentation"],
        output_vars=["description", "parameters", "returns"],
    )
```

At the very end, we define the expected input and output variables. This is not strictly part of the template but helps us when interfacing with this prompt from a high-level call in the codebase. Let's look at the end of the actual prompt.

The first block adds the input source code like in the few shot examples:

```
    """
    {{leading_indentation}}{{function_header}}
    {{function_body}}
    """
```

The syntax might remind you of a template engine like Jinja2 and as you might expect this does represent variables that will be replaced before sending the input to the Large Language Model.

Then here we insert only the function header and the comment `# Docstring below`, just like in the few shot examples, to hopefully trigger the auto-completion as we expect from the model:

```
{{leading_indentation}}{{function_header}}
# Docstring below
```

We start then our first generation with this command (let's ignore the indentation for now):

```
\"\"\"{{gen 'description' temperature=0.1 max_tokens=128 stop='.'}}
```

So here we start the docstring and then ask the LLM to generate until it finds the '.' character. Let's see a sample generation. Notice it does not include the triple quotes in the output:

```
This function extracts the function header from the given function definition
```

Then the next generated variable:

```
Parameters: {{gen 'parameters' temperature=0.1 max_tokens=128 stop='Returns:'}}
```

Which is expanded into:

```
fun_code (ast.FunctionDef): The function definition to extract the function heade
```

And finally:

```
{{gen 'returns' temperature=0.1 max_tokens=128 stop='\"\"\"'}}
```

Becomes:

```
str: The function header.
```

As you probably noticed, we have a bunch of loose fragments and not a real docstring yet. The next part is building a proper docstring from these fragments.

Building the Output

We're now at the end of our middleware server controller:

```
result = call_guidance(
        prompt_template=prompt_to_apply.prompt_template,
        input_vars=extracted_input,
        output_vars=prompt_to_apply.output_vars,
        guidance_kwargs={}
    )
logger.info("LLM output: '%s'", result)

result = command_to_apply.output_extractor(prompt_key, extracted_input, result)
```

We use the prompt above with our guidance client, to generate a dictionary that will contain the generations from the previous section. It will come in the following format:

```
{
  "description": "Func description",
  "parameters": "a (int): a number to use in the sum",
```

```
    "returns": "(int): returns the sum"
}
```

Then we can provide it to the output extractor of the `DocStringCommand` — it's a long function, so let's see just some parts of it.

First, we decide how to parse the output depending on the used prompt.

```
def output_extractor(self, prompt_key, extracted_input, result: Dict[str, str]) ->
    if prompt_key == "generic_prompt":
        return result["output"]
    elif prompt_key == "function_prompt":
        ind = extracted_input["leading_indentation"]
```

Then, when we're parsing the `function_prompt` output, there's some logic to convert the generated parameters into part of a docstring:

```
# (boring indentation workaround black magic above...)
try:
    parameters = result["parameters"].strip().split("\n")
except (KeyError, ValueError):
    parameters = []

# (handles the same for the 'returns' part...)
parameters_string = ""
if parameters:
    parameters_string = body_indentation + "Parameters: \n"
    for param in parameters:
        parameters_string += f"{body_indentation}{indentation_type}{param.lstrip()

    parameters_string += "\n"
```

Later on, we use the function header and append the description and the parameters string we built before (this is still in the same function):

```
    (...)
    code_with_docstring = (header_indentation + extracted_input["function_header"] + '
    code_with_docstring += (body_indentation + '"""')
    code_with_docstring += (result["description"].lstrip().strip() + "\n\n")

    logger.info("Header with description: %s", code_with_docstring)
    if parameters_string:
        code_with_docstring += (parameters_string)
    if returns_string:
        code_with_docstring += (returns_string)


    code_with_docstring += (body_indentation + '"""\n\n')

    code_with_docstring += extracted_input["function_body"]
    logger.info("Generated code with docstring: %s", code_with_docstring)
    return code_with_docstring
```

Note how we are "manually" reconstructing the docstring with the generations from the LLM. This is *not* the only way, it's also possible to consume the entire string directly from guidance and avoid partially this gymnastic (we'd still need to identify the start of the docstring and modify the prompt a bit).

Conclusion and Next Steps

I find it very interesting how much back into software engineering we are when using the guidance library — it does add a lot of effort to build solutions that use it, as opposed to simply prompting a Large Language Model.

However, in some cases the extra effort does pay off: the increased performance from the prompts is definitely worth it when precision is a requirement.

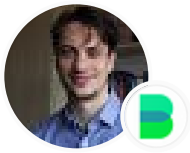
I was fairly disappointed with how hard it was using the `ast.parse` for my use case, in special it led me to two problems:

1. The annoying `IndentationError`, which led me to write a lot of extra code.
2. Stripping repeated newlines from the parsed code.

The second problem is a current source of bugs in Oasis, which means I might need to remove the `ast.parse` completely, and write my own parser.

Overall though, I think we obtained a nice starting point for writing guided prompts that assist with code generation.

Large Language Models | Prompt Engineering | Python | Vscode | Programming



Follow

Written by Paolo Rechia

200 Followers · Writer for Better Programming

Software Developer / Data Engineer - Connect with me on Linkedin: https://www.linkedin.com/in/paolo-rechia/

More from Paolo Rechia and Better Programming

Paolo Rechia in Better Programming

Building a Question-Answer Bot With Langchain, Vicuna, and Sentence Transformers

A Q/A bot with open source

10 min read · May 1

Emily Dresner in Better Programming

Why an Engineering Manager Should Not Review Code

When discussing team organization, I am often asked: "Why don't you have the tech lead manage the team?" My response is to hiss like a...

8 min read · May 11

1.7K 23

Martin Heinz in Better Programming

Real Multithreading is Coming to Python — Learn How You Can Use It Now

True multi-core concurrency is coming to Python in 3.12 release and here’s how you can use it right now using sub-interpreter API

· 6 min read · May 14

Paolo Rechia in Better Programming

Creating My First AI Agent With Vicuna and Langchain

Print jokes by prompting the Chuck Norris API

10 min read · May 1

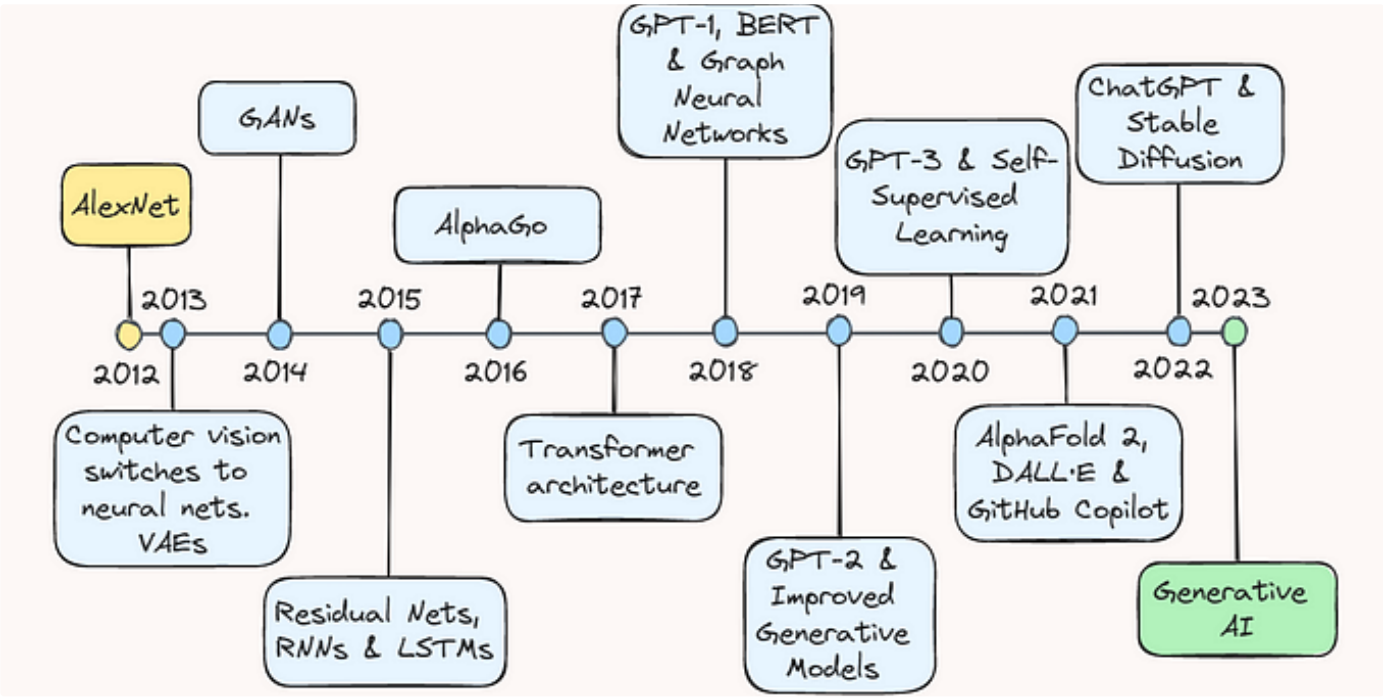
Recommended from Medium

Gabe Araujo, M.Sc. in Level Up Coding

How I Used Python to Make Everyday Tasks Easier

Hey there! As a busy person with a lot on my plate, I’m always looking for ways to make my life easier.

· 8 min read · May 1



Thomas A Dorfer in Towards Data Science

Ten Years of AI in Review

From image classification to chatbot therapy

✦ · 15 min read · May 23

1K 10

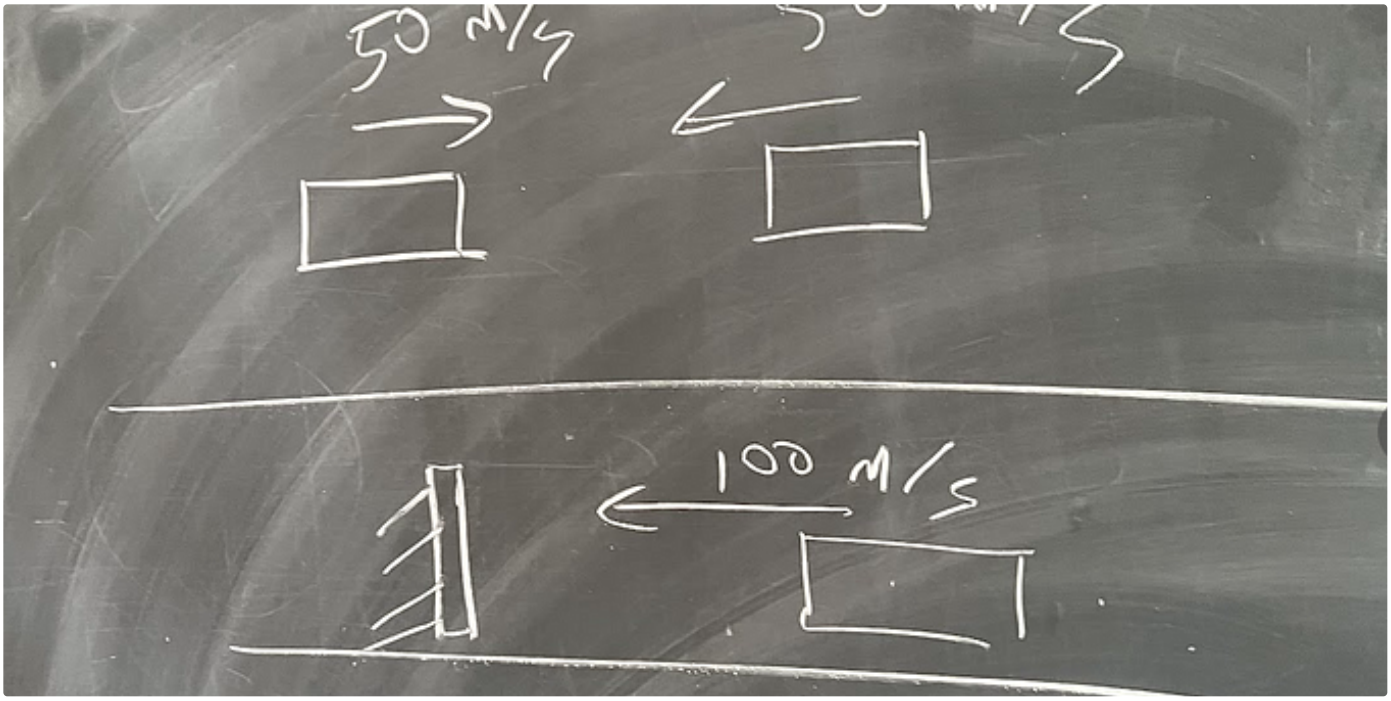
Lists

Stories to Help You Grow as a Software Developer
19 stories · 70 saves

Leadership
30 stories · 29 saves

How to Run More Meaningful 1:1 Meetings
11 stories · 35 saves

Stories to Help You Level-Up at Work
19 stories · 55 saves

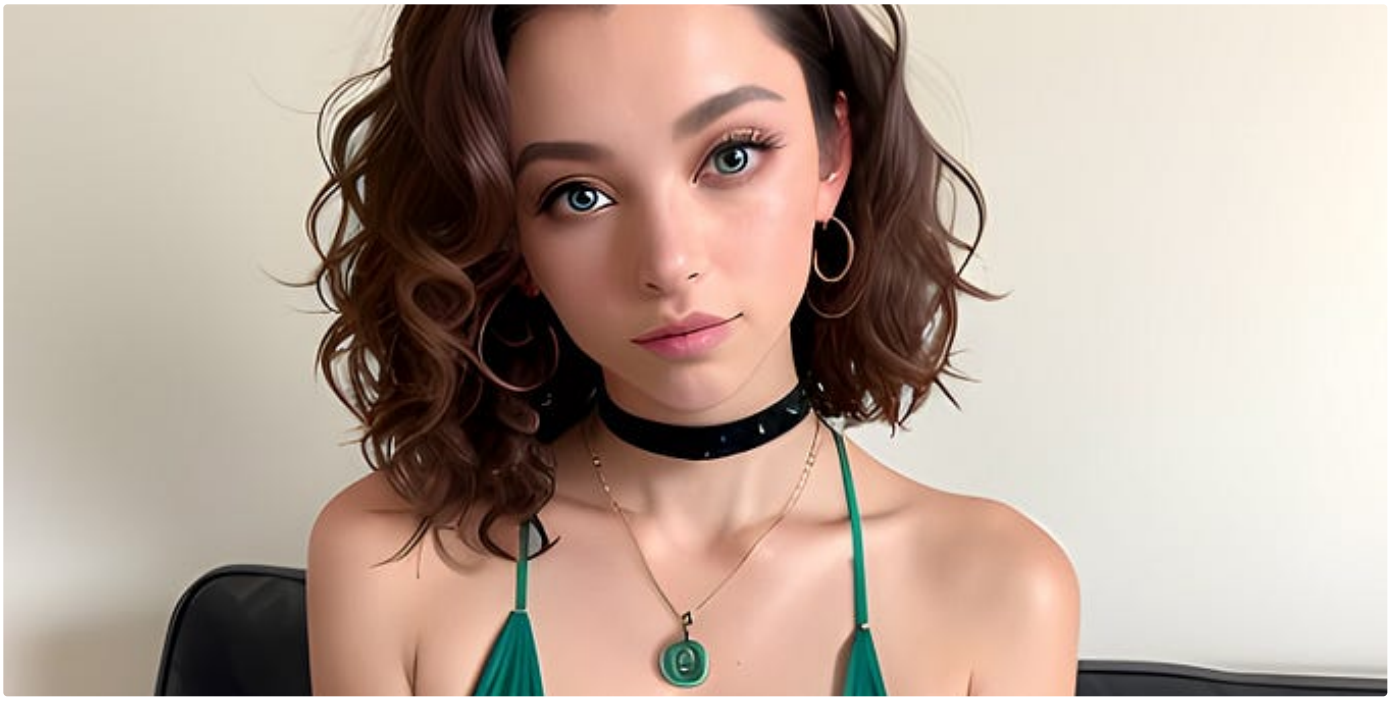


Rhett Allain

Is a 50 mph-50 mph Collision the Same as a 100–0 mph Collision?

A long time ago there was a MythBusters episode that looked at head to head crashing cars. Suppose you have two equal mass cars both...

· 9 min read · May 20



Rui Alves in The Generator

Snapchat Influencer Uses GPT-4 to Become a High-End AI Girlfriend

This is how she made $71,610 in a single month

· 4 min read · May 12

Bobby in Level Up Coding

6 Pythonic Ways to Replace if-else Statements

Avoid if-else the Pythonic Ways

✦ · 6 min read · Feb 27

424 13

Sam Westreich, PhD

Beware the Dead Sea Effect at Companies

"Salty" is a terrible adjective to apply to a job.

✦ · 7 min read · 6 days ago

1.2K 20

See more recommendations