Local AI Meets the Unix Pipeline With ShellGPT

June 18, 2026 / Gavin Jackson

ai shell-gpt ollama gemma local-ai unix bash terminal linux

I started using ShellGPT today, pointed at my locally hosted Gemma 4 26B quantized model through Ollama.

This is running on the same NVIDIA RTX PRO 4000 Blackwell workstation I wrote about in Running Modern Edge LLMs on an RTX PRO 4000 Blackwell. That setup already made local inference feel practical. ShellGPT adds a different layer on top: it puts the model directly into the terminal, where a lot of real system work already happens.

The result is interesting. Not because typing a question into a terminal is magically better than typing it into a chat window. That part was fine, but not revolutionary. It felt very similar to asking Ollama directly.

The useful part was composability.

ShellGPT becomes much more compelling when it is chained with commands, fed real output from the machine, and used to turn human language into shell commands or bash scripts.

That is where it started to click.

Note - to keep this article down to a reasonable length, none of my examples actually show output from the commands, but this should get a better idea of what is possible.

What ShellGPT Does

ShellGPT is a command-line productivity tool for talking to large language models from the shell. The project supports regular prompts, shell command generation, code generation, chat sessions, REPL mode, and shell integration.

The command is `sgpt`, so the basic shape is simple:

```
sgpt "Explain what the Linux load average means in plain English"
```

That is useful enough. It means I can ask a question without leaving the terminal, switching to a browser, opening another app, or copying context around.

But with a local model, the more important detail is that the question stays on the workstation. The prompt goes to Ollama on `localhost`, not to a cloud API.

That matters for the sort of environments I care about. A lot of shell work involves logs, hostnames, service names, paths, configuration snippets, and operational clues that should not be casually pasted into a public SaaS tool.

Asking Questions Was Fine

The first thing I tried was the obvious thing:

```
sgpt "What is the difference between a systemd service and a systemd target?"
```

That worked exactly as expected. It gave a reasonable explanation. I tried a few more:

```
sgpt "What does set -euo pipefail do in a bash script?"
```

```
sgpt "Explain the difference between a bind mount and a named Docker volume"
```

This is handy, but it is not very different from:

```
ollama run gemma4-26b "What does set -euo pipefail do in a bash script?"
```

For standalone questions, ShellGPT is mostly a nicer terminal wrapper around a model. That is not a criticism. It is still convenient. But if that was all it did, I probably would not write a whole post about it.

The real power showed up when I stopped treating it like chat and started treating it like another Unix tool.

Piping Real Output Into The Model

The shell is already good at collecting evidence. It can list files, stream logs, inspect services, query APIs, and format text. ShellGPT becomes useful when it can sit at the end of that pipeline and reason over the output.

The pattern is simple:

> *Use normal shell tools to collect the evidence, then pipe that evidence into `sgpt` and ask the model to interpret it.*

That feels much more useful than asking generic troubleshooting questions in isolation.

For example, instead of asking a generic question about disk usage, I can give it the actual disk usage:

```
df -h | sgpt "Which filesystem should I investigate first? Keep the answer short."
```

Or I can give it a size breakdown:

```
du -xh --max-depth=1 /var 2>/dev/null \
 | sort -h \
 | tail -20 \
 | sgpt "Summarise which directories are taking space and suggest the safest next checks."
```

That is a better workflow than asking "how do I troubleshoot disk space?" because the model has the current state in front of it.

Failed services are another nice fit:

```
systemctl --failed --no-pager \
 | sgpt "Explain these failed services and rank them by likely importance."
```

Logs are an obvious example:

```
journalctl -u ollama -n 120 --no-pager \
 | sgpt "Summarise the last failure and suggest the next three commands to run."
```

For Docker:

```
docker ps --format 'table {{.Names}}\t{{.Status}}\t{{.Ports}}' \
 | sgpt "Explain which services are exposed and whether anything looks surprising."
```

If a specific container is misbehaving, I can hand ShellGPT the recent logs directly:

```
docker logs --tail 120 my-container 2>&1 \
 | sgpt "Find the likely root cause of the error in these logs."
```

For Git:

```
git diff --stat \
 | sgpt "Summarise the scope of this change in one paragraph."
```

And for quick commit-message drafting:

```
git diff --cached \
 | sgpt "Write a concise git commit message for these staged changes."
```

For local AI work, GPU state is useful context:

```
nvidia-smi \
 | sgpt "Summarise GPU utilisation and whether this looks healthy for local LLM inference."
```

And because Ollama exposes a local API, I can query it and ask ShellGPT to make the output more readable:

```
curl -s http://localhost:11434/api/tags \
 | sgpt "List the available Ollama models and explain which one looks like the default candidate."
```

This is the part I liked most. The terminal already has the context. ShellGPT lets the local model read that context without me manually copying it somewhere else.

Generating Shell Commands From Human Language

ShellGPT has a shell command mode:

```
sgpt --shell "find all markdown files changed in the last 7 days and print their titles"
```

or with the short flag:

```
sgpt -s "show me the ten largest files under /var/log modified in the last week"
```

It returns a command and then gives an interactive choice to execute, describe, or abort.

That interaction is important. I do not want a local model blindly running system commands on my behalf. I want it to propose a command, let me inspect it, ask it to describe the command if needed, then decide whether to run it.

One detail I appreciated is that ShellGPT is smart enough to account for the shell you are using. Bash, Zsh, and Fish all have slightly different syntax and conventions, and the generated commands or scripts can reflect that instead of treating every terminal as generic POSIX `sh`. That matters once you get into arrays, globbing, completions, aliases, functions, and shell-specific quality-of-life features.

Some examples that fit my normal workflow:

```
sgpt -s "find markdown posts that contain the word ollama and sort them newest first"
```

```
sgpt -s "show listening TCP ports with process names on Ubuntu"
```

```
sgpt -s "compress all png files in this directory into a tar.gz archive named images-backup.tar.gz"
```

```
sgpt -s "use curl to check whether http://localhost:11434/api/tags is responding"
```

```
sgpt -s "create a one-line command to list the current GPU name, driver version, and used VRAM"
```

This is where ShellGPT feels different from a normal chatbot. I am not asking for a tutorial. I am asking for the exact command I need right now. I'm going to be using this a lot for tcpdump options - I always forget these!

That is a small shift, but it changes the feel of the workflow.

Building Bash Scripts From Plain English

The best use case today was using ShellGPT to draft more complex bash scripts.

For example, I can ask:

```
sgpt --code "Write a bash script called ollama-health.sh that:
- checks whether the Ollama API is listening on localhost:11434
- prints the installed Ollama models
- sends a short test prompt to gemma4-26b
- prints GPU memory usage with nvidia-smi
- exits non-zero if any check fails"
```

That is the sort of script I would normally write myself, but it takes a few minutes of looking up curl flags, remembering the Ollama API shape, and deciding how neat I want the error handling to be.

ShellGPT can produce a first draft immediately:

```
#!/usr/bin/env bash
set -euo pipefail

MODEL="${1:-gemma4-26b}"
OLLAMA_URL="${OLLAMA_URL:-http://127.0.0.1:11434}"

echo "Checking Ollama API..."
curl -fsS "$OLLAMA_URL/api/tags" >/dev/null

echo "Installed models:"
ollama list

echo "Testing model: $MODEL"
ollama run "$MODEL" "Reply with one short sentence confirming you are ready."

echo "GPU memory:"
nvidia-smi --query-gpu=name,memory.used,memory.total --format=csv
```

Would I run that blindly on a production host? No.

Would I use it as a starting point, review it, tighten it, and save myself some time? Absolutely.

Another useful prompt:

```
sgpt --code "Write a bash script that scans a directory of markdown blog posts, extracts YAML title
and date fields, and prints a sorted table of date, title, and filename."
```

Or:

```
sgpt --code "Write a bash script that checks all Docker containers, prints unhealthy ones, shows
their last 50 log lines, and exits 1 if any container is unhealthy."
```

Or:

```
sgpt --code "Write a bash script that backs up /etc/nginx and /etc/systemd/system to a timestamped
tar.gz file, writes a SHA256 checksum, and keeps only the last 10 backups."
```

This is the sweet spot for me. I know enough bash to review the result. I know enough Linux to catch dangerous assumptions. What I want is acceleration: a draft that gets the boring structure in place so I can focus on the intent.

Installation With Ollama

The upstream project is here:

ther1d/shell_gpt on GitHub

The basic install from the README is:

```
python3 -m pip install --upgrade shell-gpt
```

By default, ShellGPT uses OpenAI's API. To use Ollama, install ShellGPT with LiteLLM support:

```
python3 -m pip install --upgrade "shell-gpt[litellm]"
```

If Ollama is not already installed, the Linux installer is:

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

If you are using the official ShellGPT Ollama guide as a quick test, it uses:

```
ollama pull gemma4:e4b
```

In my case, I already had a larger Gemma 4 26B quantized model imported into Ollama from earlier testing. The important thing is the model name that appears in `ollama list` (**IMPORTANT**: don't forget to prefix this with ollama/ - see below). Use that exact name in the ShellGPT configuration.

Open the config file:

```
nano ~/.config/shell_gpt/.sgptrc
```

Set the Ollama-related values:

```
DEFAULT_MODEL=ollama/gemma4-26b:latest
OPENAI_USE_FUNCTIONS=false
USE_LITELLM=true
API_BASE_URL=http://localhost:11434
OPENAI_API_KEY=0
```

Replace `gemma4-26b:latest` with the exact model name from `ollama list`. If your model is called `gemma4:e4b`, use:

```
DEFAULT_MODEL=ollama/gemma4:e4b
```

Then test it:

```
sgpt "Hello Ollama. Reply in one short sentence."
```

If that works, try shell mode:

```
sgpt -s "print the current directory, git branch, and latest commit"
```

A Few Practical Notes

ShellGPT's own documentation notes that local models may not behave exactly like the default hosted OpenAI path. That matches my experience. Local models are good enough to be useful, but they still need supervision.

A few habits make the workflow safer:

- Use `--shell` when you want a command, because the interactive execute, describe, and abort flow is useful.
- Ask for a description before running anything destructive.
- Avoid piping `--no-interaction` directly into `bash`.
- Treat generated scripts as drafts.
- Keep Ollama bound to localhost unless you have a deliberate reason to expose it.
- Use the exact model name from `ollama list`.

The last point sounds mundane, but it matters. With Ollama, the name is the contract. If the model is registered as `gemma4-26b:latest`, that is what ShellGPT and LiteLLM need to see.

The Takeaway

My first impression is that ShellGPT is not most interesting as "chat in a terminal."

That is fine, but familiar.

It is more interesting as a shell-native bridge to a local model. I can pipe logs into it, ask it to interpret real command output, generate one-off commands, and draft bash scripts in the place where those scripts are going to run.

That makes local AI feel less like a separate destination and more like part of the operating environment.

For a local Gemma 4 model running through Ollama on the RTX PRO 4000 Blackwell, that is exactly the kind of workflow I wanted to test. The model does not need to be perfect. It needs to be nearby, private, fast enough, and useful in the flow of work.

Today, ShellGPT cleared that bar and will be part of my toolkit.

Links

- ther1d/shell_gpt on GitHub
- ShellGPT Ollama wiki page
- Ollama
- My earlier post: Running Modern Edge LLMs on an RTX PRO 4000 Blackwell