Project 4 Paper
Prompt Engineering Experiments for SPICE Netlist Generation: Follow-Up Study
Zoe Rochelle II
CSC575 Generative A.I. – Dr. Lin
Quinnipiac University
12/11/2025

Part 1: Technique Selection & Prompt Design

1.1 Background and Goal

In my original project, I used a small pretrained language model (Qwen/Qwen3-0.6B-Base) to generate ngspice-compatible SPICE netlists from natural-language circuit descriptions [1]. The core idea was to treat netlists as structured code and evaluate different prompt engineering strategies on a 20-case benchmark of SPICE tasks (DC, AC, transient, nonlinear devices, and subcircuits) [5].

For this follow-up assignment, I reused that experiment and layered additional prompt engineering techniques on top of my existing **few-shot** setup. The goal was to see which prompt changes actually move the metrics, and which ones are mostly neutral.

1.2 Techniques Selected

From the provided list, I focused on four techniques:

1. **Few-Shot Learning** (baseline from my previous work)
    a. Already implemented as P2 in my original project.
    b. Gives the model 3 complete ngspice netlists as examples before the new task.
    c. Serves as the reference configuration for this assignment.
2. **Instruction Refinement** (P4)
    a. Start from the P2 few-shot prompt and refine the instructions based on observed error patterns.
    b. Easy to implement: add a small “rules” section without changing the model or examples.
3. **Role Prompting** (P5)
    a. As the model to pretend it’s an “expert analog circuit designer and ngspice power user.”
    b. Very low engineering overhead: prepend a role paragraph to the existing P2 prompt.
4. **Output Format Control** (P6)
    a. Force the model to wrap the netlist inside [NETLIST] tags.
    b. Let's me robustly extract just the netlist portion before evaluation.

I chose these three new techniques because they are not too complicated to integrate into my workflow, and they do not require changing the evaluation code or training the model.

1.3 Prompt Templates

Below are simplified versions of the prompt templates that I used for each technique. The same three few-shot examples are used in all cases: a DC operating-point divider, an RC transient step response, and an RC AC low-pass filter.

1.3.1 Few-Shot Learning (P2)

```
### Example 1 (DC OP divider)
Request:
Create an ngspice netlist for a DC resistor divider...
Netlist:
V1 in 0 DC 10
R1 in out 1k
R2 out 0 2k
.op
.print DC V(out)
.end

### Example 2 (RC transient)
...

### Example 3 (RC AC low-pass)
...

### New Task
Request: {spec from benchmark JSON}
Netlist:
```

The new task's spec is taken from the JSON benchmark entry (e.g., "Create an ngspice-compatible netlist for a DC voltage divider…") [4], and the model is expected to output only the netlist lines.

1.3.2 Instruction Refinement (P4)

P4 uses the exact same three examples as P2, but inserts a "refined instructions" block before the new task:

```
### Example 1 ...
... (same as P2 examples)

### Additional generation rules
You are generating ngspice-compatible netlists. Follow these rules:
- Reuse the node, source, and subcircuit names that appear in the request.
  Do not invent new names or rename them (e.g., keep VIN as VIN, DIV as DIV).
- Always include at least one output directive: .print, .plot, or .meas.
- If the circuit uses diodes, BJTs, MOSFETs, or zeners, include appropriate
  .model lines and use those model names consistently.
- If you define a .subckt, also include a matching .ends <name> before .end.
- Always end the netlist with a single .end line.
```

```
### New Task
Request: {spec}
Netlist:
```

Nothing else changes: same model, same decoding settings, same examples.

1.3.3 Role Prompting (P5)

```
P5 prepends a short role paragraph before the few-shot examples:
You are an expert analog circuit designer and ngspice power user.
Your job is to translate natural-language circuit descriptions into
ngspice-compatible netlists that run without errors.
You strictly follow ngspice syntax, keep node and source names
consistent, and always end with .end.

### Example 1 ...
... (same P2 examples)

### New Task
Request: {spec}
Netlist:
```

The rest of the prompt (examples + structure) is identical to P2.

1.3.4 Output Format Control (P6)

```
P6 again uses the P2 few-shot block, but adds explicit instructions about
wrapping the netlist in tags:
### Example 1 ...
... (same P2 examples)

### Output Format Rules
Wrap the final ngspice netlist between [NETLIST] and [/NETLIST].
Inside [NETLIST]...[/NETLIST], write ONLY valid ngspice lines:
- No explanations, comments, or markdown.
- Use node 0 as ground when appropriate.
- Always include any required analysis and at least one .print/.plot/.meas.
- Always end the netlist with a single .end line.

Anything outside [NETLIST]...[/NETLIST] will be ignored.

### New Task
Request: {spec}
Answer:
```

In the Python script, I extract only the text between [NETLIST] and [/NETLIST] and then trim at .end before writing the .cir file.

Part 2: Experimental Implementation & Results

2.1 Test Cases and What They Evaluate

I reused my original **20-case SPICE benchmark**, stored in JSON, but here I highlight five representative cases to match the rubric:

1. **spice_001 – DC resistor divider (DC OP)**
    a. Tests basic syntax, .op analysis, and a .print or .meas directive.
    b. Checks exact lines like V1 in 0 DC 10, R1 in out 1k, R2 out 0 2k.
2. **spice_003 – Series RLC transient (TRAN)**
    a. Sine source driving R-L-C in series, .tran analysis, measurement at the output node.
    b. Stresses correct component naming (V1, R1, L1, C1) and transient syntax.
3. **spice_005 – AC high-pass filter (AC)**
    a. Tests .ac analysis and frequency-domain .print ac directives.
    b. Checks that the model correctly sets up source amplitude and AC analysis line.
4. **spice_013 – NMOS DC transfer curve (MOSFET)**
    a. Requires consistent naming across source (VIN), .dc directive, MOSFET instance (M1), and .model NMOS1 NMOS.
    b. Stresses cross-line consistency and device model syntax.
5. **spice_014 – Divider subcircuit (SUBCKT)**
    a. Uses .subckt DIV in out gnd, .ends DIV, and an instantiation X1 vin vout 0 DIV at the top level.
    b. Tests hierarchical netlist structure and correct pairing of .subckt / .ends.

The full benchmark covers DC, AC, transient, nonlinear devices (diodes, BJTs, MOSFETs), dependent sources, and subcircuits. For each case, the JSON file defines must_contain and must_contain_any lists that specify required substrings for evaluation.

2.2 Experimental Setup

For all variants, I used the same model and decoding configuration:

- Model: Qwen/Qwen3-0.6B-Base (pretrained base model, no fine-tuning). [1][2]
- Decoding:
    - do_sample = False
    - max_new_tokens = 200
    - repetition_penalty = 1.1
    - no_repeat_ngram_size = 6

For each technique (P2, P4, P5, P6), I:

1. Read the 20-case JSON benchmark.
2. Built prompts using the appropriate template.
3. Generated netlists and saved them to runs/Px/spice_XXX.cir.

4. Evaluated each .cir using three metrics:
    a. **Syntax validity**
    b. **Coverage** (fraction of must_contain and must_contain_any satisfied)
    c. **Exact-match** (strict substring matches for key strings).

2.3 Comparison of Results

The table below summarizes the aggregate performance for the four techniques on the 20-case benchmark:

Technique	Description	Syntax Pass Rate	Avg Coverage	Avg Exact-Match
P2 – Few-Shot	Baseline 3-example prompt	0.55	0.311	0.109
P4 – Instruction Refinement	P2 + extra rules block	0.55	0.311	0.109
P5 – Role Prompting	P2 + expert role paragraph	**0.65**	0.338	**0.134**
P6 – Output Format Control	P2 + [NETLIST]...[/NETLIST] wrapper	0.55	0.311	0.109

Key observations:

- **Instruction Refinement (P4)** and **Output Format Control (P6)** produced **identical metrics** to the base few-shot prompt (P2).
- **Role Prompting (P5)** showed a modest but consistent improvement over P2:
    - Syntax pass rate increased from 0.55 → 0.65.
    - Coverage rose from 0.311 → 0.338.
    - Exact-match improved from 0.109 → 0.134.

2.4 Example Outputs

As a concrete example, consider the **DC divider** case (spice_001):

Spec (this is simplified):

> *"Create an ngspice-compatible netlist for a DC voltage divider: V1 supplies 10 V DC from node in to ground. R1 is 1k from in to out. R2 is 2k from out to 0. Run an operating point analysis and print the DC voltage at node out."*

A representative output from each technique looked like:

P2 / P4 / P6 (typical):

```
V1 in 0 DC 10
R1 in out 1k
R2 out 0 2k
.op
.print DC V(out)
.end
```

P5 (Role Prompting, typical):

```
V1 in 0 DC 10
R1 in out 1k
R2 out 0 2k
.op
.print DC V(out)
.end
```

For easy cases like this, all techniques tend to produce nearly identical netlists. The differences show up more clearly in harder cases (e.g., MOSFET DC transfer and subcircuits), where P5 is slightly more likely to keep naming consistent and to include required directives, which is reflected in the higher syntax and exact-match scores.

Part 3: Analysis & Reflection

3.1 Which Techniques Performed Best and Why

Among the three new techniques, **Role Prompting (P5)** was the only one that clearly improved performance over the base few-shot prompt (P2). By casting the model as an "expert analog circuit designer and ngspice power user," the prompt appears to nudge the model toward more consistent and "careful" behavior. This showed up as:

- Higher syntax validity (0.65 vs. 0.55),
- Slightly higher coverage,
- Best exact-match score (0.134).

In contrast, **Instruction Refinement (P4)** and **Output Format Control (P6)** were essentially **neutral**: their metrics matched P2 exactly. In P4, the extra rules (don't rename signals, always include .model lines for devices, keep .subckt / .ends paired) did not change the average behavior. A likely reason is that the three few-shot examples already encode most of the desired behavior, so restating it as text rules doesn't add much.

For P6, the [NETLIST]...[/NETLIST] wrapper is mainly a convenience for parsing; my P2 outputs were already mostly clean and netlist-only, so adding tags didn't change how the model reasons about the circuit. It helped the post-processing code become more robust but did not affect the evaluation metrics.

3.2 What I Learned About Prompt Engineering

Few-shot examples dominate vague instructions - The examples in P2 carry a lot of information: how to format netlists, how to use .op, .tran, .ac, and .print, and how to end with .end. Simply adding more rules on top of those examples doesn't necessarily change behavior.

Role Prompting can help, even in code like domains - I expected role prompting to matter more in open-ended writing tasks, but it also helped here. The "expert ngspice user" framing seems to make the model a bit more conservative and consistent.

Output format control is valuable for tooling, not always for quality - The [NETLIST]...[/NETLIST] tags did not improve the netlists themselves, but they made my extraction and evaluation code simpler and safer. That's still valuable, just not directly visible in the metrics.

Hard failures are structural, not stylistic - The remaining problem cases (MOSFET DC transfer, subcircuits) involve tightly coupled lines and hierarchical structure. These are hard to fix with surface-level prompt changes. They likely require either more targeted examples, some SPICE-aware checking, or even light fine-tuning.

3.3 Applying These Findings to My Final Project

Unfortunately, I made the mistake of completing Assignment 4 after finishing our final project. But I'd imagine that if I'd instead done them in reverse, I might have used Few-Shot + Role Prompting as the default and behaved a bit more selectively when it came to the Instruction Refinement phase.

Citations

[1] Qwen Team, “Qwen3-0.6B-Base,” *Hugging Face Model Card*, Aug. 2025. [Online]. Available: https://huggingface.co/Qwen/Qwen3-0.6B-Base. Accessed: Dec. 11, 2025.

[2] Qwen Team, “Qwen3 – A Qwen Collection,” *Hugging Face Collection*, Aug. 2025. [Online]. Available: https://huggingface.co/collections/Qwen/qwen3. Accessed: Dec. 11, 2025.

[4] Ngspice Project, “Ngspice Documentation: Manual and Control Flow,” 2024. [Online]. Available: https://ngspice.sourceforge.io/docs.html. Accessed: Dec. 11, 2025.

[5] R. Lin, “Final Project: Semiconductor Simulation Code Generation – CSC 375/575,” *Course Website, Quinnipiac University*, 2025. [Online]. Available: https://rongyulin3.com/qu_genai/assignments/final_project_instructions.html. Accessed: Dec. 11, 2025.