

Prompt Engineering Experiments for SPICE Netlist Generation Using a $\leq 1\text{B}$ -Parameter Language Model

Zoe Rochelle II
CSC575 Generative A.I. – Dr. Lin
Quinnipiac University
12/11/2025

This report studies how different prompt engineering techniques affect the ability of a small language model to generate ngspice-compatible SPICE netlists from natural-language descriptions. I use Qwen/Qwen3-0.6B-Base (a 0.6B-parameter base model) [2] and compare four prompt setups: a simple instruction-only baseline, a structured-constraints prompt, a few-shot prompt with three worked examples, and a two-pass critique–revise pipeline. All experiments use the same decoding configuration and a custom 20-case benchmark [5] that checks syntax validity, coverage of required substrings, and strict exact matches of key lines. The baseline performs very poorly, while the best configuration reaches a syntax pass rate of 0.60, average coverage of 0.35, and exact-match of 0.13. These experiments show that prompt design alone, especially few-shot prompting plus a constrained revision step, can significantly improve structured code generation with a small non-instruction-tuned model, but also reveal remaining failure modes on more complex SPICE tasks.

I. Model Selection and Justification

The goal of this project is to generate **ngspice-compatible SPICE netlists** from natural-language specifications, under two constraints: the model must have **$\leq 1\text{B}$ parameters**, and the output must be **netlist-only** (no extra prose) [5]. SPICE netlists are treated as a form of **structured code**: each line follows a strict syntax (e.g., V1 in 0 DC 10), references must match across lines (VIN in both the source and the .dc statement), and small errors like missing .end cause the file to fail.

I chose **Qwen/Qwen3-0.6B-Base** [1] for three reasons. First, the 0.6B parameter size satisfies the $\leq 1\text{B}$ requirement. Second, the model is small enough to run locally on an RTX 3080 GPU, making it practical to generate outputs for many prompt variants and benchmark cases. Third, I deliberately used a **base** model rather than an instruction-tuned one. This makes the task harder, but it also makes it easier to see the impact of prompt design itself, because the model has not been heavily optimized for following instructions or producing code-like outputs.

No fine-tuning was performed in this work. All experiments are **inference-only**: the underlying

model weights are fixed, and only the prompts and the evaluation scripts change.

II. Training Methodology and Hyperparameters

Although my project did not utilize training in the usual sense, the experimental procedure and decoding hyperparameters still need to be controlled.

A. Experimental Setup

All experiments were run on a local machine with an NVIDIA RTX 3080. For each of the 20 benchmark tasks [5], I constructed a prompt using one of four prompt variants:

- P0: instruction-only baseline
- P1: structured constraints
- P2: few-shot with three examples
- P3: two-pass critique–revise on top of P2

For each variant and each task, the model generated a single completion, which was saved as a .cir file (e.g., runs/P2/spice_003.cir). These outputs were then scored by a separate evaluation script.

B. Decoding Hyperparameters

To ensure that differences in performance come from **prompting** rather than decoding randomness, I used the same decoding configuration for all conditions:

- `do_sample = False` (deterministic decoding)
- `max_new_tokens = 200`
- `repetition_penalty = 1.1`
- `no_repeat_ngram_size = 6`
- `pad_token_id` and `eos_token_id` taken from the tokenizer/model

This setup roughly corresponds to greedy decoding with some repetition control, and it is held fixed for P0–P3.

C. Prompt Variants

P0 – Baseline, Instruction-Only

The baseline prompt asks the model to “generate an ngspice-compatible SPICE netlist” given the task description. It enforces a few basic rules: the output should be **netlist-only**, use node 0 as ground, and terminate with `.end`. There are no examples and no explicit structure beyond this.

P1 – Structured Constraints

P1 keeps the same goal but adds a short “**Hard Constraints**” list to the prompt. The model is explicitly told to always use node 0 as ground, include at least one output directive (e.g., `.print`, `.plot`, or `.meas`), and end with `.end`. The prompt can also suggest an output structure with sections like TITLE, NETLIST, ANALYSIS, and OUTPUT. P1 still has no examples, but the expected shape of a “complete” answer is more clearly stated than in P0.

P2 – Few-Shot Prompting

P2 introduces **three worked examples** before the actual task. Each example is a complete ngspice netlist [3][4]:

1. A DC operating-point resistor divider with `.op` and `.print V(out)` [3].

2. An RC transient step response with `.tran` and `.print tran V(out)` [3].
3. An RC low-pass filter with `.ac` and `.print ac V(out)` [3].

After these examples, the prompt presents the new benchmark specification using a “Request: ... Netlist:” pattern. The idea is that the model can copy the structure, naming style, analysis directives, and final `.end` from the examples when generating a new netlist.

P3 – Critique–Revise, Two-Pass

P3 turns P2 into a **two-pass pipeline**:

- *Pass 1*: generate a draft netlist using the same few-shot prompt as P2.
- *Pass 2*: feed the draft back in with a **revision prompt** that includes a checklist derived from that task’s `must_contain` and `must_contain_any` strings.

The revision prompt is written with **patch-only rules**: it tells the model not to rename nodes or components, not to change values unless necessary, to keep `.end`, and to make only minimal edits to satisfy the checklist (for example, insert a missing `.print` line). In code, a simple coverage score is computed on the draft; Pass 2 is only run if that score is low, and afterwards the script keeps whichever version (draft or revised) has higher coverage.

III. Benchmark Design and Evaluation Metrics

A. Benchmark Design

To evaluate the prompts, I built a benchmark of **20 SPICE tasks** stored in a JSON file. Each entry includes:

- `id` (e.g., `spice_001`),
- `category` (e.g., DC resistive, AC filter, transient, diode, MOSFET, subcircuit),
- `spec`: a natural-language description of the circuit and required analysis,
- `must_contain`: a list of exact substrings that should appear in the netlist,

- `must_contain_any`: a list of “at least one of these” groups.

The `must_contain` field covers specific component and directive lines, such as V1 in 0 DC 10, R1 in out 1k, .op, .tran, .ac, .dc, and .end [3][4]. The `must_contain_any` field is standardized as `[["print", ".plot", ".meas"]]` for all cases, which enforces that each netlist includes at least one output directive.

Each prompt variant is run over the same 20 tasks. A generation script reads the JSON, formats the prompt from spec, calls the model, and writes the resulting netlist to disk. An evaluation script then scores each .cir file.

B. Evaluation Metrics

I used three metrics:

1. **Syntax validity**: A netlist passes if it (i) ends with .end, (ii) has at least one line that looks like a valid SPICE element or source (e.g., starts with R, C, L, V), and (iii) includes the required analysis directive (.op, .tran, .ac, or .dc) [3][4] if one is specified for that task.
2. **Coverage score**: For each case, the script counts how many `must_contain` strings appear and how many `must_contain_any` groups are satisfied (at least one string from the group is present). Coverage is the fraction of requirements satisfied, and ranges from 0 to 1.
3. **Exact-match score**: For a subset of key lines (for example, V1 in 0 DC 10 or .dc VIN 0 5 0.1), the script checks whether those substrings appear exactly, using case-insensitive matching. This metric is stricter than coverage and emphasizes token-level details like labels, node names, and value formatting.

IV. Results and Analysis

A. Aggregate Results

Across all 20 tasks, the four prompt variants perform as follows:

- **P0 (baseline)**: syntax pass rate 0.00, average coverage ≈ 0.05 , exact-match 0.00.
- **P1 (structured)**: syntax pass rate 0.20, coverage ≈ 0.11 , exact-match ≈ 0.02 .
- **P2 (few-shot)**: syntax pass rate 0.55, coverage ≈ 0.31 , exact-match ≈ 0.11 .
- **P3 (critique-revise)**: syntax pass rate 0.60, coverage ≈ 0.35 , exact-match ≈ 0.13 .

Because the model, benchmark, and decoding are kept fixed, these differences come from the prompt strategies.

B. Interpretation

The jump from P0 to P1 shows that simply listing constraints (“use node 0,” “include an output directive,” “end with .end”) is helpful but not sufficient. Many netlists are still incomplete or malformed, so syntax and coverage remain low.

The largest improvement comes from P2. The three worked examples give the model a clear pattern for what a valid ngspice netlist looks like [3][4]: component lines, analysis directive, at least one .print/.plot/.meas, and .end. This greatly reduces “format drift” and leads to much better syntax and coverage, and also a significant increase in exact-match.

P3 adds a smaller but consistent gain. The critique–revise step uses the benchmark’s own `must_contain` strings as a checklist to fix missing lines. The patch-only rules and the “keep best of draft or revised” logic are important: without them, an unconstrained second pass can actually break good drafts (for example, by renaming components or removing .end). With these safeguards, P3 acts as a targeted repair stage on top of P2.

V. Failure Case Analysis

To see where the model still fails, I examined three representative cases.

Series RLC Transient (spice_003).

This case describes a sine source driving a series R–L–C chain, plus a transient analysis and a measurement at the output node. In weaker prompts, the model sometimes produced lines that looked like an RLC circuit but missed the element designators (e.g., in `0 SIN(...)` instead of `V1 in 0 SIN(...)`), or it wrote bare node/value lines where SPICE expects `R1 in n1 10`. These outputs were “close” but failed syntax and `must_contain`. Few-shot prompting made proper `V1`, `R1`, `L1`, and `C1` lines much more common, and P3’s checklist helped add any still-missing substrings.

MOSFET DC Transfer (spice_013).

Here the benchmark requires a consistent input source and sweep directive (e.g., `VIN in 0 DC 0` and `.dc VIN 0 5 0.1`), a four-terminal MOSFET line, and a `.model NMOS1 NMOS` line. Early variants often defined `V1` but wrote `.dc VIN ...`, or omitted the model line, or got the MOSFET terminals wrong. From a circuit point of view, these outputs were understandable, but they failed the strict text checks. P2 improved consistency by showing complete examples, and P3 further encouraged matching names across lines, but this case remains sensitive to small naming mistakes.

Subcircuit Divider (spice_014).

This task asks for a `.subckt DIV in out gnd` definition with internal resistors, a closing `.ends`, an instance line like `X1 vin vout 0 DIV`, an analysis command, and `.end`. Many failures involved missing `.ends`, confusing `.end` with `.ends`, or using the wrong subcircuit name in the instance. Because none of the P2 examples use subcircuits, the model sometimes struggled with the hierarchical structure. The revision step in P3 could add missing lines when they were explicitly listed in `must_contain`, but this case shows that prompt engineering alone has limits when the language model must manage multiple scopes and a strict block structure.

V. Citations

[1] **Qwen Team**, “Qwen3 Technical Report,” *arXiv preprint* arXiv:2505.09388, 2025. [Online].

Available: <https://arxiv.org/abs/2505.09388>.

Accessed: Dec. 11, 2025.

[2] **Qwen**, “Qwen3-0.6B-Base,” *Hugging Face model card*, 2025. [Online]. Available:

<https://huggingface.co/Qwen/Qwen3-0.6B-Base>.

Accessed: Dec. 11, 2025.

[3] **H. Vogt, G. Atkinson, and P. Nenzi**, *Ngspice User’s Manual, Version 42*, Dec. 27, 2023. [Online].

Available:

<https://ngspice.sourceforge.io/docs/ngspice-42-manual.pdf>. Accessed: Dec. 11, 2025.

[4] **Ngspice Project**, “Ngspice User’s Manual (Version 45, HTML),” *ngspice Documentation*, 2024. [Online]. Available:

<https://ngspice.sourceforge.io/docs/ngspice-html-manual/manual.xhtml>. Accessed: Dec. 11, 2025.

[5] **R. Lin**, “Final Project: Semiconductor Simulation Code Generation – CSC 375/575,” *Course website, Quinnipiac University*, Fall 2025. [Online].

Available:

https://rongyulin3.com/qu_genai/assignments/final_project_instructions.html. Accessed: Dec. 11, 2025.