Fachhochschule Graubünden
University of Applied Sciences

Lab: Prompting in Coding

AISE501 – AI in Software Engineering

Spring Semester 2026

Model: `qwen3.5-35b-a3b` Server: `silicon.fhgr.ch:7080` Duration: 3 hours

Introduction

In this lab you practice the prompting techniques covered in the lecture by working directly with a large language model running on the faculty GPU server via an OpenAI-compatible API.

You will complete four exercises that build on each other:

1. **XML Structured Prompting** – structure a request using XML tags
2. **Persona, Task, and Data** – combine role, task, and file context
3. **Structured Output** – request and parse JSON/YAML responses
4. **Chain-of-Thought Pipeline** – build an iterative plan-then-execute pipeline with validation

Each exercise has a skeleton file (`ex0X_...py`) with `# TODO` markers and a solution file (`ex0X_..._solution.py`).

Hints & Tips

Before starting, open a terminal **in the folder that contains the exercise files** and run:

```
python test_connection.py
```

You should see `Models available: ['qwen3.5-35b-a3b']` followed by `Connection OK.`

All exercise scripts (`ex01_...py` etc.) must also be run from that same folder, because they import `server_utils` and read `analyze_me.py` from the current directory.

Key concept – Qwen3 think mode: Qwen3 has a built-in chain-of-thought mode that wraps responses in `<think>...</think>` blocks. In `server_utils.py` this is disabled so you get direct answers:

```
extra_body={"chat_template_kwargs": {"enable_thinking": False}}
```

In Exercise 4 you will implement your own explicit reasoning step to understand what think mode does internally.

Helper functions in `server_utils.py`

The shared utilities provide the following functions:

`chat(client, messages, ...)` – general-purpose call. Returns the raw response string. Use this when you want plain text or code output.

`chat_json(client, messages, ...)` – structured-output call. Adds

```
response_format={"type": "json_object"}
```

which instructs the vLLM server to constrain token sampling so the output is *always* a syntactically valid JSON object. Use this in Exercises 3 and 4 wherever you need to call `json.loads()` on the result.

Important: `response_format=json_object` requires the top-level output to be a JSON *object* (`{...}`), not a bare array (`[...]`). If you need a list, wrap it: `{"items": [...]}`.

`print_messages(messages)` – debugging helper. Call this *before* every `chat()` or `chat_json()` call to print the exact prompt hierarchy (system, user, and assistant turns) that the model receives. The skeleton files already include commented-out `print_messages()` calls at every API call site.

1 Exercise 1 – Basic XML Structured Prompting

File: `ex01_xml_prompting.py` **Time:** approx. 30–40 min

Learning goals: Connect to the local LLM. Understand the difference between unstructured and XML-structured prompts. Use a system prompt to set a global persona and response style.

Why XML tags?

A **structured prompt** uses named sections to separate concerns. XML tags are a natural choice:

- They are unambiguous – every section has a clear start and end
- LLMs are trained on vast amounts of XML/HTML and parse it reliably
- Tags can be nested for hierarchical information

Compare the same request, unstructured vs. structured:

```
-- Unstructured --
Explain list comprehensions, give a filter example, and list two
    beginner mistakes.

-- Structured --
<request>
  <topic>Python list comprehensions</topic>
  <example>Filter even numbers from a list</example>
  <focus>Syntax overview and two common beginner mistakes</focus>
</request>
```

Part A – Run the Unstructured Prompt (pre-filled)

Part A is already complete. Run the file and read the response before continuing.

Part B – XML-Structured Prompt (TODOs 1–3)

TODO 1 Fill in the three XML sections inside `structured_content`: `<topic>`, `<example>`, `<focus>`

TODO 2 Build the `structured_messages` list. A messages list is a `list[dict]`, each dict having keys `"role"` and `"content"`. Role is one of `"system"`, `"user"`, or `"assistant"`.

TODO 3 Call `chat(client, structured_messages)`, store the result, and print it. Compare with the output from Part A.

Part C – System Prompt (TODOs 4–5)

TODO 4 Write an XML-structured system prompt using tags: `<persona>`, `<style>`, `<constraints>`

TODO 5 Build a messages list with the system message *first*, followed by the user message from Part B. Call `chat()` and observe how the answer changes.

Hints & Tips

- The XML lives *inside* the `"content"` string of the user message – it is just a string passed to the API.
- Tag names are arbitrary. Choose names that describe the section's purpose.
- A system message must always be the **first** element of the messages list.
- Start with `temperature=0.7` (the default). Try 0.2 for more deterministic answers.

Common Mistakes

- Forgetting a closing tag (`</topic>`) – responses become less reliable.
- Putting the system message after the user message – the API will reject it.
- Using `print(chat(...))` directly – you lose the response for multi-turn follow-ups.

Reflection:

1. How did XML structure change the format and depth of the response?
2. What happens if you use inconsistent or missing closing tags?
3. How does the system prompt interact with the user message?

2 Exercise 2 – Persona, Task, and Data

File: `ex02_persona_task_data.py` **Time:** approx. 40–50 min

Learning goals: Separate three prompt concerns with XML: *who* the LLM is, *what* it must do, and the *data* it works with. Pass a real Python file as context inside a `<code>` tag. Use multi-turn conversation to refine output.

The three-tag pattern

This pattern is the foundation of code-review and RAG (Retrieval-Augmented Generation) pipelines:

```
<persona>  Who the LLM should be          </persona>
<task>     What it must do                </task>
<code>     The data it should work with   </code>
```

The file `analyze_me.py` contains a small statistics module with **seven intentional bugs**. Try to spot them yourself before asking the LLM – it helps you evaluate the model's output.

Part A – First Structured Review (TODOs 1–4)

TODO 1 Fill in `<persona>`: describe a senior Python engineer who values correctness and PEP-8.

TODO 2 Fill in `<task>`: ask the LLM to identify every bug with a short explanation.

TODO 3 The `<code>` block already contains the file content via an f-string – do not change it.

TODO 4 Build `messages_a` (user message only) and call `chat()`. How many bugs were found?

Part B – Prioritised Bug List (TODOs 5–6)

TODO 5 Extend `<task>` and add an `<output_format>` tag asking the model to group findings by severity (**Critical** / **Medium** / **Style**), including line numbers and a one-line fix hint for each.

TODO 6 Add a system prompt reinforcing the persona. Build `messages_b` and call `chat()`.

Part C – Request a Corrected Function (TODO 7)

Pick one buggy function (e.g. `calculate_statistics`). Build `messages_c` by extending `messages_b`: append the model's last response as an `"assistant"` message, then add a new `"user"` message asking for a corrected version with type hints and a docstring.

Hints & Tips

- Read the file at the top of the skeleton: `code_to_review = Path("analyze_me.py").read_text()`
- Embed the content in the `<code>` tag using an f-string: `f"<code>{code_to_review}</code>"`
- For multi-turn conversation (Part C), always append the previous response as an `"assistant"` message before adding the new `"user"` message.
- Add `language="python"` and `filename="analyze_me.py"` as attributes in the `<code>` tag to set context without extra prose.

The seven bugs in `analyze_me.py` – try to find them first:

1. `ZeroDivisionError` when `numbers` is empty in `calculate_statistics`
2. `IndexError` when `numbers` is empty in `calculate_statistics`
3. Population variance used instead of sample variance ($\div N$ vs. $\div(N-1)$)
4. No context manager for file I/O in `process_data`
5. `int()` crashes on floats and blank lines in `process_data`
6. Operator-precedence error in min-max normalisation in `normalize`
7. Silent failure with `print()` instead of `raise ValueError` in `normalize`

Reflection:

1. Did the LLM find all seven bugs? Which did it miss or hallucinate?
2. How did the `<output_format>` tag change the answer's structure?
3. What is the advantage of multi-turn conversation vs. starting fresh?

3 Exercise 3 – Structured Input and Structured Output

File: `ex03_structured_output.py` **Time:** approx. 40–50 min

Learning goals: Instruct the model to return machine-parseable JSON or YAML. Parse the response in Python and use it programmatically. Build a follow-up prompt dynamically from parsed data.

Why structured output?

When an LLM response needs to be processed by code, plain prose is not enough. Asking for **JSON** or **YAML** lets you parse, filter, and chain results without fragile string manipulation. Three techniques enforce it – use all three together:

1. **System prompt**: *"Respond only with valid JSON, nothing else."*
2. **Schema tag**: embed the exact JSON schema inside a `<schema>` tag in the user prompt.
3. `chat_json()`: use the server-level `response_format={"type": "json_object"}` constraint, which token-samples directly into valid JSON.

Target JSON schema for this exercise:

```
{
  "summary": "<one-sentence overview>",
  "bugs": [
    {
      "id": 1,
      "severity": "Critical|Medium|Style",
      "line": <int or null>,
      "function": "<function name>",
      "description": "<what is wrong>",
      "fix": "<one-sentence fix hint>"
    }
  ],
  "overall_quality": "Poor|Fair|Good|Excellent"
}
```

Part A – Request JSON Output (TODOs 1–4)

TODO 1 Write a system prompt that enforces pure JSON output (no markdown fences, no prose).

TODO 2 Write the user prompt with `<persona>`, `<task>`, `<schema>`, and `<code>` tags.

TODO 3 Build `messages_a` and call `chat_json()` with `temperature=0.2`. `chat_json()` adds the server-level JSON constraint on top of your system prompt.

TODO 4 Store the raw string in `raw_json_a` and print it.

Part B – Parse and Display (TODOs 5–6)

TODO 5 Parse `raw_json_a` with `json.loads()`. Wrap in `try/except json.JSONDecodeError`.

TODO 6 Print a formatted summary table using `ljust()` / `rjust()` for

alignment.

Part C – Dynamic Follow-Up Prompt (TODOs 7–8)

TODO 7 Filter `bugs` for `severity == "Critical"`. Build a follow-up user prompt that lists each critical bug by ID and description, and asks for corrected code wrapped in a JSON *object* (required by `chat_json`): `{"fixes": [{"bug_id": 1, "fixed_code": "..."}, ...]}`

TODO 8 Continue the conversation (multi-turn), call `chat_json()`, parse the result with `json.loads(raw)["fixes"]`, and print each fix.

Part D – YAML Output (TODO 9)

Repeat Part A requesting YAML instead of JSON. Parse with `yaml.safe_load()` (`pip install pyyaml`). Which format do you prefer for human-readable reports? For machine pipelines?

Hints & Tips

- Embed the JSON schema inside a `<schema>` tag. Use double braces `{{...}}` when the schema appears inside an f-string to escape the curly braces.
- Use `chat_json()` (not `chat()`) for Parts A and C – it adds server-level JSON enforcement so `json.loads()` is guaranteed to succeed.
- `response_format=json_object` requires a top-level JSON object. For Part C wrap the list: `{"fixes": [...]}` and extract it with `json.loads(raw)["fixes"]`.
- Use `temperature=0.2` for structured output – lower temperature means more predictable formatting.

Common Mistakes

- LLMs sometimes add a preamble before the JSON even when told not to. `chat_json()` prevents this, but always use `try/except` as a safety net.
- **Never** use `eval()` to parse LLM output – use `json.loads()` or `yaml.safe_load()` only.
- For optional fields like `line`, include `null` in your schema; otherwise the model may emit the string `"null"` or omit the key.
- Part D (YAML) uses `chat()`, not `chat_json()` – YAML is plain text, not a JSON object.

Reflection:

1. What can go wrong when asking an LLM to return JSON?
2. How did the `<schema>` tag influence the output?
3. Why is structured output essential for building LLM pipelines?

4 Exercise 4 – Build Your Own Chain-of-Thought Pipeline

File: `ex04_cot_pipeline.py` **Time:** approx. 50–60 min

Learning goals: Understand how built-in reasoning models (o1, DeepSeek-R1, Qwen3 think mode) work at a high level. Build an iterative plan-then-execute pipeline manually. Validate each step before proceeding. Compare CoT output with a direct single-shot response.

The iterative CoT pattern

Modern "reasoning" models generate an internal scratchpad before answering. In **Qwen3 think mode** this appears as `<think>...</think>` blocks; in **OpenAI o1** it runs silently. You can replicate this behaviour manually:

Call 1 – Planning	input →	JSON plan (steps with reasoning)
Call 2 – Step 1	buggy code + step 1 →	updated code (validated)
Call 3 – Step 2	code from step 1 + step 2 →	updated code (validated)
...	...→	...
Call N – Step N-1	code from step N-2 + step N-1 →	final code (validated)

Unlike a one-shot approach (dump the entire plan into a single prompt), this **iterative** pattern:

- Focuses the model's attention on *one change at a time*
- Validates each step (syntax check) before moving on
- Builds on *concrete code* from the previous step, not abstract intentions
- Isolates errors – if step 4 breaks, you know exactly where

The task: fully rewrite `analyze_me.py` to fix all bugs and add type hints and docstrings.

Part A – Planning Phase (TODOs 1–5)

The model should *only* produce a plan here – no code yet.

TODO 1 Write a system prompt that assigns the role of *software architect*, permits only planning (no coding), and enforces JSON-only output. **Explicitly forbid code snippets in all fields** – instruct the model to use plain English descriptions only. This prevents unescaped quote characters inside JSON string values from breaking the output.

TODO 2 Write the planning user prompt with tags: `<problem>`, `<code>`, `<task>`, `<schema>`. Each plan step should have: `step_id`, `title`, `reasoning`, `action` (plain English, no code), `depends_on`.

TODO 3 Build `messages_plan` and call `chat_json()` with `max_tokens=4096`. Use `chat_json()` for the same reason as Exercise 3: guaranteed valid JSON. Use `max_tokens=4096` because a multi-step plan easily exceeds the 2048 default, which would silently truncate the JSON.

TODO 4 Parse the plan JSON and print each step in a human-readable format.

TODO 5 (Optional) Inspect and edit the `plan` dict before passing it to Phase B.

Part B – Iterative Execution Phase (TODOs 6–10)

Now the model implements the plan **one step at a time**, with syntax validation after each step.

TODO 6 Write a system prompt for a *senior Python developer* who receives the current code plus a single step to apply. The model should return the complete updated module.

TODO 7 Complete the `validate_syntax()` function: write code to a temp file, run `py_compile`, return `(True, "")` or `(False, error_message)`.

TODO 8 Implement the step-by-step loop. For each step in `plan["steps"]`:

- Build a prompt with `<current_code>`, `<step>`, and `<task>` tags
- Call `chat()` and strip code fences from the response
- Validate syntax – if it passes, update `current_code`

TODO 9 Add retry logic: when a step produces a syntax error, send the error back to the model and let it fix itself (one retry attempt).

TODO 10 Save the final `current_code` to `analyze_me_fixed.py` and run it with `subprocess` as a final validation.

Part C – Baseline Comparison (TODO 11)

Send the same problem as a direct single-shot prompt *without* any plan or iteration. Save both outputs to files and compare: which is more complete and correct?

Hints & Tips

- **Each execution step is a fresh conversation.** The model receives only the system prompt, the current code, and the single step to apply – not the entire planning history.
- Ask the model to mark each change with a comment: `# Step 1 - Handle empty list`. This makes the connection between plan and code verifiable.
- Use `temperature=0.2` for all execution steps for deterministic output. Set `max_tokens=4096` for both planning and execution phases.
- Keep the `action` field values in the schema description to plain English. If the model includes Python code (e.g. `raise ValueError(f"...")`), the embedded quotes break the JSON – even with `chat_json()`.
- For `validate_syntax()`, use `subprocess.run([sys.executable, "-m", "py_compile", filename])`.

Common Mistakes

- **Do not continue the planning conversation into Phase B.** The system prompt says "only plan, no code", so the model will refuse to write code if that instruction is still active.
- Not setting `max_tokens` high enough causes outputs to be silently truncated, producing invalid JSON for Phase A or incomplete code for Phase B.
- `chat_json()` still requires a top-level JSON object – the plan schema already satisfies this because the top level is `{"goal": ..., "steps": [...]}`.
- Always call `strip_code_fences()` on the model's response before saving or

validating – the model often wraps code in ```` ```python ```` even when told not to.
- Clean up your temp file (`_tmp_validate.py`) after syntax checking with `unlink(missing_ok=True)`.

Reflection:

1. How did the iterative CoT output differ from the direct single-shot response?
2. Did the validation step catch any syntax errors? How were they resolved?
3. What would happen if you supplied a deliberately wrong plan?
4. How does this manual pipeline relate to built-in think modes in o1, DeepSeek-R1, and Qwen3?
5. What are the trade-offs of step-by-step iteration vs. one-shot execution?
6. How could you extend validation beyond syntax checking? (unit tests, linting, type checking)