Alright, let's translate all of that into "plain English" **without dropping any of the technical details**. I'll keep the same structure so you can map it back to the original.

---

1. Overall method: what you're doing and why

The task

You define the task as a pipeline:

text → π_1 → schema → π_2 → table rows (JSONL)

- **π_1**: schema induction model (Stage 1)
  It reads the document and decides what fields/columns should exist and how they're structured.
- **π_2**: table extraction model (Stage 2)
  It fills in those fields from the document and outputs rows as JSONL (one JSON object per line).

Your goals:

- The **schema** should be:
  - Compact,
  - Only based on evidence in the text,
  - No made-up (hallucinated) fields.
- The **tables** should:
  - Exactly follow that schema (names, order, structure),
  - Contain only facts that can be verified in the text.

Why two stages instead of one big model?

Two-stage pipeline (π_1 + π_2) vs. one end-to-end model

1. Decomposability & interpretability

- π_1 and π_2 do different jobs:
  - **π_1**: "What columns should the table have?" (names, structure)
  - **π_2**: "What data goes into these columns?"
- Splitting them means:
  - You can debug π_1 alone: is the schema reasonable?
  - You can debug π_2 alone: is it filling the table correctly?

    - You can measure errors:
        - At schema level (π_1),
        - At table level (π_2).

And it mirrors how humans work:

1. First define the table structure (columns).
2. Then fill it in.

2. Stronger control & constraints

- π_2 takes the schema as **input**.
- That means:
    - You can **force** the exact field names and ordering.
    - You can feed **different schemas later** (even from another source) without retraining π_2.
- Conceptually:
    - π_1 = “schema program”.
    - π_2 = “schema executor”.

3. Better RL later

Because of this separation:

- You can give π_1 a reward based on **how well its schemas help π_2** perform.
- You can give π_2 a reward based on **table accuracy** when using realistic schemas from π_1.
- Doing this kind of “pipeline RL” is much harder if you only had one black-box model spitting out tables.

How to say it to a professor:

> I decomposed the task into two models: one for schema induction (π_1) and one for table extraction (π_2). This separation makes the system more interpretable, allows independent evaluation and RL signals for each stage, and lets me enforce schema constraints explicitly when generating tables.

Why SFT first, then RL, then pipeline RL?

Your plan:

- **Phase A**: SFT on gold data
    - π_1^s: schema SFT model
    - π_2^s: table SFT model

- **Phase B**: RL per stage
    - π_1^r-local: π_1 with local RL
    - π_2^r-local: π_2 with local RL
- **Phase C**: Pipeline RL
    - π_1^r-pipe, π_2^r-pipe: both trained to work well together.

Why this order:

1. **SFT gives a strong starting point**
    - Uses exact gold targets (gold schema and gold table).
    - Low-noise gradients: the model sees the correct answer directly.
    - Standard pattern: **pretrain** → **SFT** → **RL** (same as RLHF pipelines).
2. **RL for non-differentiable metrics**
    - Your rewards include:
        - Schema F1,
        - JSON validity,
        - Table cell F1,
        - End-to-end pipeline accuracy.
    - These are not simple cross-entropy losses.
    - RL is the right tool to directly optimize these metrics.
3. **Pipeline RL only after the stages are decent**
    - If π_2 is still bad, using π_2 as an "executor" to train π_1 gives noisy, unhelpful rewards.
    - So first you make each stage competent (SFT + local RL), then connect them and do pipeline RL.

Professor version:

> I follow a pretrain → supervised fine-tuning → RL → pipeline RL sequence. SFT provides low-variance learning from gold labels, local RL optimizes non-differentiable objectives like F1 and JSON validity, and only after both stages are stable do I introduce pipeline RL so each model can be rewarded based on the combined behavior.

---

2. Stage-1 (π_1) SFT: schema model design

This includes:

- Prompt design,
- Data format,
- 3-stage training schedule,
- Hyperparameters (LoRA, learning rate, batch size, context length, etc.).

2.1 Why a separate schema SFT builder?

The schema SFT builder turns each training example into:

```
{
  "messages": [
    {"role": "system", "content": SYSTEM_PROMPT},
    {"role": "user", "content": "<|policy|>...[METADATA]...[DOCUMENT]...[TASK]"},
    {"role": "assistant", "content": "{...schema json...}"}
  ]
}
```

Why this is reasonable:

1. **Matches how you'll actually call the model**

At inference time you will also send:

- A **system** message that defines the model's job.
- A **user** message with:
    - [POLICY] block,
    - [METADATA],
    - [DOCUMENT],
    - [TASK].

So you **train** on the same structure you **serve** with. That reduces train–test mismatch.

2. **Explicit [POLICY] → more control**

The [POLICY] section spells out rules such as:

- Use `lower_snake_case` field names,
- Only include fields supported by evidence,
- Type constraints and required/optional rules,
- Enums, regex patterns, etc.,
- How to choose structure (kv_single vs flat_single_row).

Instead of hoping the model "implicitly learns" these from examples, you **state them explicitly** in text. This makes behavior:

- Easier to understand,
- Easier to adjust,
- More stable.

3. **[METADATA] supports domain generalization**

The METADATA JSON may contain:

- Language,
- Locale,
- Table hints,
- Missing-value tokens,
- Etc.

Even if you don't use all of it yet, it:

- Makes the prompt self-contained.
- Lets you later extend behavior (e.g., set numeric format for `de-DE`).

4. **Tags like `<|policy|>` and [DOCUMENT]**

These delimiters:

- Help humans read the prompt.
- Help tools and reward functions find specific parts of the prompt later (like the policy block).

How to phrase it:

> I designed the schema SFT prompts to mirror the final inference interface: a system role plus a structured user message with [POLICY], [METADATA], [DOCUMENT], and [TASK] sections. This explicit formatting makes the model's behavior more controllable and makes it easy to parse specific blocks in downstream analysis or reward models.

2.2 Why is `build_assistant_schema` designed this way?

`build_assistant_schema` turns the gold schema into a simplified object with:

- Snake_case field names (deduplicated),
- All types set to `"string"`,
- `required = False` for all fields,
- A deterministic `schema_id` hash,
- A heuristic choice of structure: `kv_single` vs `flat_single_row`.

Reasons:

1. **You only trust header information**

- The gold JSONL has both headers and row data.
- In this stage, you **deliberately ignore row data**.
- That means schema induction is based only on what's in the prompt (headers/text), not hidden information.

2. **Conservative typing (`"string"`)**

- Without stat analysis of values, guessing `"integer"`, `"number"`, `"boolean"` is risky.
- Using `"string"`:
    - Avoids incorrect type guesses,
    - Still teaches the model the structural form: fields list, constraints object, etc.

3. **Name normalization and collisions**

- Names are normalized to `lower_snake_case`.
- If two different source headers normalize to the same name, you add suffixes (`_2`, `_3`, …).
- This strictly follows the policy.
- The schema target always obeys your naming rules, giving the model a consistent pattern.

4. **Simple structure detection**

- Many real documents are:
    - Flat tables, or
    - Key-value pairs.
- `detect_structure()` just decides between those (e.g., `kv_single` vs. `flat_single_row`).
- More complex setups (multi-headers, ragged tables) can be handled later with more logic or RL. For SFT, this simple heuristic is enough.

2.3 Stage-1 SFT hyperparameters (π_1)

Context lengths per stage:

- **Stage 1**: 2048 tokens
- **Stage 2**: 4096 tokens
- **Stage 3**: 8192 tokens

Why:

1. **Curriculum over sequence length**

- Short sequences → faster, easier learning.
- Most documents are short; early training focuses on:
    - Output format (schema JSON),
    - Policy rules.
- Later stages expose the model to longer documents and metadata, which is harder.

2. **Fit hardware**

- 8k context on an 8B model is heavy.
- Using 2k → 4k → 8k gives a smooth ramp-up in memory and time.

How to describe it:

> I used a three-stage curriculum on context length (2k → 4k → 8k tokens), so the model first learns the schema format on shorter inputs and then gradually adapts to long documents while respecting GPU memory limits.

Time budgets:

- `stage1_hours = 4.0`
- `stage2_hours = 5.0`
- `stage3_hours = 5.0`
- Estimated `est_step_seconds = 22.0`

You compute how many steps to run per stage from:

> hours × 3600 / est_step_seconds

You also support a `--max_steps` override.

Why this setup:

- You care about **wall-clock time** (compute budget) more than about exact number of epochs.
- Stage 1 uses shorter sequences → more steps per hour, so it can have slightly fewer hours.
- Stages 2 and 3 use longer context → balanced time for long-context refinement.
- `--max_steps` gives reproducibility across machines with different speeds.

Explanation:

> I planned training by wall-clock hours per stage, using an estimated seconds-per-step to determine the number of updates. This matches practical compute constraints and `max_steps` allows reproducible runs.

Learning rates and warmup ratios:

- **Stage 1**:
  - `lr = 1.5e-4`
  - `warmup_ratio = 0.08`

- **Stage 2**:
  - `lr = 8.0e-5`
  - `warmup_ratio = 0.05`
- **Stage 3**:
  - `lr = 5.0e-5`
  - `warmup_ratio = 0.03`

Why:

1. **LR decreases over stages**
   - Stage 1: more aggressive LR to rapidly learn format and policy on shorter sequences.
   - Stage 3: smaller LR for careful adjustment on long contexts (to avoid overwriting earlier learning).
2. **Warmup shrinks over stages**
   - Early stages: larger warmup to gently start from the base model.
   - Later stages: LoRA weights are already adapted, so you need less warmup.

How to phrase:

> I use a decaying learning rate schedule across stages (1.5e−4 → 8e−5 → 5e−5) and shorter warmup ratios later. Early stages can use more aggressive updates, while long-context training needs more stability.

LoRA hyperparameters:

- `lora_rank = 128`
- `lora_alpha = 256`
- `lora_dropout = 0.05`
- Targets: `"q_proj,k_proj,v_proj,o_proj,gate_proj,up_proj,down_proj"`

Why these values:

1. **Rank 128 = mid-range**
   - Too low (8, 16) → may not capture enough nuance across domains.
   - Too high (256, 512) → heavier in memory.
   - Rank 128 is a common sweet spot for 7B–13B models.
2. **Alpha = 2 × rank**
   - Common rule of thumb.
   - Scales LoRA updates to a reasonable level.
3. **Dropout = 0.05**
   - Small but non-zero → helps prevent overfitting.
   - Avoids the LoRA layers memorizing specific schemas/documents.

4. **Targeted modules**
    - Q/K/V/O: attention projections (control what to attend to).
    - gate/up/down: MLP (controls transformations of hidden states).
    - This is a standard LoRA configuration.

Micro batch size & gradient accumulation (Stage 1):

- `stage1_micro_bsz = 8` (per GPU)
- `stage1_grad_accum = 4`
- Suppose you have 2 GPUs:
    - Global batch size = 8 × 4 × 2 = **64**

Later stages:

- Smaller micro batch (due to longer sequences),
- Larger `grad_accum` to maintain a reasonable global batch size.

Why:

- Fitting in GPU memory while still having a stable effective batch size.

Bucketing:

- Enable `--use_buckets` with:
    - `chars_per_token = 4.0`
    - `bucket_margin = 0.25`

What this does:

- Approximates token count from character length.
- Groups examples into buckets by length.
- Sends shorter examples to shorter-context stages, longer ones to longer-context stages.
- `bucket_margin = 0.25` makes the boundaries less sharp so examples near a boundary don't get weird behavior.

How to explain:

> I used approximate length bucketing based on character counts to assign examples to appropriate context-length stages. I estimated 4 characters per token and used a 0.25 margin to avoid hard boundary effects.

---

3. Stage-2 (π_2) SFT: table model design

Stage 2 is similar in spirit to Stage 1, but focused on **table extraction**.

3.1 Why separate table model & NDJSON output?

You train π_2 with:

- **system**: e.g., "You convert documents into tabular data under a schema…"
- **user**: [POLICY] + [SCHEMA] + [DOCUMENT] + [TASK]
- **assistant**: NDJSON (each line is one JSON row).

Why:

1. **Matches evaluation exactly**

- Your eval/reward pipeline:
  - Splits the model's output by newline,
  - Runs `json.loads` on each line,
  - Compares to gold table rows (cell-level, field-level).

Training the model to emit exactly that NDJSON format removes mismatch between training and evaluation.

2. **Schema-conditioned extraction**

- The [SCHEMA] block contains:
  - `schema_id`,
  - `structure`,
  - `fields` list.

This tells π_2:

- Which columns to output,
- In what order.

This is important for later RL when π_2 will:

- Take **predicted** schemas from π_1,
- Fill tables consistent with those schemas.

3. **Why JSONL/NDJSON vs arrays**

- NDJSON is:
  - Streamable (you can process line by line),
  - Easy to parse,
  - A natural format for RL environments and data pipelines.

How to phrase to professor:

> I train π_2 to produce line-delimited JSON (NDJSON) conditioned on an explicit [SCHEMA] block. This exactly matches the evaluation setup and allows the same model to accept arbitrary schemas produced by π_1.

3.2 SFT2 hyperparameters and tweaks

Most of π_2's trainer mirrors π_1's:

- Same three context lengths: **2048 / 4096 / 8192**,
- Similar time budgets and LR schedule.

But there are a few differences.

Smaller eval & logging frequencies:

- `per_step_eval_samples = 200`
- `final_eval_samples = 1000`
- `log_freq = 50`
- `eval_freq = 200`

Why:

- Table prompts and outputs are **longer** than schema outputs.
- Evaluation on each step costs more.
- So you evaluate less frequently to keep time overhead manageable.

Dataloader settings:

- `num_workers = min(8, cpu_cnt)`
- `prefetch_factor = 2`
- `persistent_workers = False`

Why:

- On HPC clusters / containers (Apptainer/Singularity), too many workers or large prefetch can cause:
  - Memory thrashing,
  - Worker crashes.
- These conservative settings give more stable behavior.

Generation max_new_tokens = 512

- Schema outputs are short → 256 tokens is enough.
- Tables can have many rows → you allow up to **512 new tokens** so the model can output more rows before hitting the limit.

3.3 Quick sanity check for π_2

After training π_2, you run a `_quick_table_sanity` check that measures:

- JSON validity rate (how many lines parse as JSON),
- Column order correctness (matches schema-defined order),
- Simple type consistency checks.

Why:

- Cross-entropy loss (LM loss) doesn't guarantee:
  - Proper JSON syntax,
  - Correct field ordering,
  - Basic consistency.

The sanity check:

- Is cheap to run,
- Directly tells you whether the model respects the "schema contract".

How to phrase:

> Because table extraction is more constrained than generic text generation, I added a sanity check that measures JSON validity, column-order fidelity, and simple type consistency. These structural properties are not fully captured by language-model loss.

---

4. Infrastructure: VERL, FSDP, FlashAttention-2, LoRA

4.1 Why Llama 3.1-8B + LoRA?

Base model:

- Llama 3.1-8B (or similar 8B instruct model):
  - Good at language understanding,
  - Good zero-shot reasoning,
  - Strong base for schema and table tasks.

LoRA instead of full fine-tuning:

- Full fine-tuning:
    - Trains all parameters,
    - Much heavier in terms of GPU and storage.
- LoRA:
    - Trains only small low-rank adapters,
    - Reduces trainable parameter count,
    - Lowers memory usage,
    - Lets you keep the base model fixed and just swap adapters.

This is a standard approach for modern LLM training, especially in research settings with limited compute.

4.2 Why VERL + FSDP + FlashAttention-2?

VERL:

- Provides:
    - SFT trainer with FSDP built in,
    - RL trainers (like GRPO/PPO),
    - Good integration with vLLM for sampling.
- Advantage: you don't need to build RL infrastructure from scratch.

FSDP (Fully Sharded Data Parallel):

- Shards:
    - Model parameters,
    - Optimizer states,
    - Gradients across GPUs.
- Allows:
    - Training 8B models with long context (up to 8k),
    - Larger batch sizes,
    - Fit into limited GPU memory.

FlashAttention-2:

- Optimized attention implementation:
    - Faster than standard attention,
    - More memory-efficient (streaming key/value).
- This makes long-context training feasible.

How to phrase:

I use VERL as the training framework, combined with FSDP and FlashAttention-2. VERL provides consistent SFT and RL support, while FSDP and FlashAttention-2 make it feasible to fine-tune an 8B-parameter model with 8k context on the available GPUs.

---

5. How to talk about specific values in a defense

Here's a compact summary you can literally read off:

- **Two models (π_1, π_2)**
  → Separate schema induction and table extraction for interpretability, modularity, and cleaner RL.
- **SFT → local RL → pipeline RL**
  → Start with supervised learning on gold labels, then optimize non-differentiable metrics per stage, then train the pipeline jointly.
- **Three SFT stages: 2k / 4k / 8k tokens**
  → Curriculum over sequence length to reduce compute early and gradually adapt to long documents.
- **Time-based training: 4h / 5h / 5h; est_step_seconds = 22.0**
  → Plan training based on wall-clock compute, with `max_steps` for reproducibility.
- **LoRA: rank 128, alpha 256, dropout 0.05; target Q/K/V/O and MLP (gate/up/down)**
  → Mid-range capacity vs. memory; 2×rank alpha heuristic; small dropout; standard attention + MLP targeting.
- **Learning rates and warmup ratios:**
  - Stage 1: 1.5e-4, warmup 0.08
  - Stage 2: 8.0e-5, warmup 0.05
  - Stage 3: 5.0e-5, warmup 0.03
    → More aggressive updates when learning basic format; more conservative updates for long-context refinement.
- **Bucketing: chars_per_token = 4.0, bucket_margin = 0.25**
  → Approximate token lengths for routing examples to appropriate context stages, with a smooth boundary.
- **Prompt structure:** `[POLICY] [METADATA] [DOCUMENT] [TASK]`, with tags like `<|policy|>`
  → Explicitly defined structure that's easy for humans and reward models to parse and manipulate.
- **Table output as NDJSON**
  → One JSON object per line, easy to parse, evaluate, and use for RL.
- **Quick sanity metrics for π_2:**
  → `json_valid_rate`, `key_order_pass_rate`, `type_check_pass_rate` to assess structural correctness on top of LM loss.

If you'd like, I can now turn this into a nicely formatted "Methodology & Hyperparameter Justification" section in formal academic style that you can slot straight into your report.