A Three-Stage Pipeline for Fine-Tuning Dedicated NPC Language Models Using Synthetic Persona Data

S.G. Tidwell – East Texas A&M

Graduate Student – Master's Computer Science - Artificial Intelligence

11/20/2025

ABSTRACTION

This work presents a practical three-stage pipeline for constructing dedicated non-player characters (NPC) language models using synthetic persona-aligned data. Rather than relying on a single general-purpose model with prompt-based persona conditioning, the system trains a separate, fully fine-tuned 3B parameter model per NPC. The pipeline consists of:

- ***Seed generation,*** where a local teacher model expands a structured persona brief into hundreds of constrained situation "seeds".
- ***Response generation,*** where the same teacher model produces multi-paragraph, in-character responses for each seed.
- ***Full fine-tuning,*** where the synthetic corpus is used to specialize a base Llama-3.2-3B-Instruct checkpoint into a dedicated NPC model.

Inference is served from a Python based API running on dedicated GPUs, with one fp16 NPC model resident in memory at a time and lazy loading for model switching. The approach yields stable, persona-consistent NPCs with minimal manual authoring effort, and provides a blueprint for reproducible, persona-centric fine-tuning workflows.

INTRODUCTION

Large language models (LLMs) are increasingly used to drive conversational NPCs in games and interactive fiction. The most common patter is to reuse a single general-purpose model and attempt to "lock" different characters via prompts, system messages, or retrieval-augmented generation (RAG). While convenient, this strategy often suffers from:

- **Persona drift** – characters gradually lose their distinctive voice.
- **Lore bleeding -** facts, relationships, or mannerisms leak between NPCs.
- **Prompt bloat** – long control prompts consume context that could be used for player interaction.

To address these issues, this work adopts a different stance: for major NPCs, train a dedicated model per character. Each important NPC is backed by its own fully fine-tuned LLM with weights specialized to that persona.

This paper describes a three-stage pipeline that automates the creation of such NPC specific models:

1. **Seed generation** – a local teacher model creates structured, short "seed situations" annotated with tone, setting, and lore.

2. **Response generation** – the teacher model expands each seed into multi-paragraph, in-character responses.
3. **Full fine-tuning** – the synthetic corpus is used to fine-tune a Llama-3.2-3B-Instruct model into a dedicated NPC model.

The pipeline is implemented in Python using Hugging Face Transformers, runs efficiently on a single RTX 5090 class GPU, and is deployed with a Python inference server that lazy-loads NPC models on demand.

SYSTEM OVERVIEW

At a high level, the system uses one strong local "teacher" model (Qwen/Qwen2.5-7B-Instruct) to generate synthetic persona data, and a smaller "student" model (meta-llama/Llama-3.2-3B-Instruct) as the deployable NPC brain.

The workflow is:

1. **Persona brief:** the designer defines a structured persona for an NPC (e.g. Mindella Stormbringer, a witty, flirty cleric with specific lore constraints).
2. **Seed generator:**
    a. Prompts the teacher model as a "content planner"
    b. Produces compact situation seeds structured as JSON.
3. **Response generator:**
    a. Reads the seeds.
    b. Prompts the teacher as the NPC, generating 150-300 word in-character responses for each seed.
    c. Writes all data to JSONL formatted file
4. **Fine-tuning:**
    a. Each NPC has its own model directory with weights.
    b. A Python API server loads/unloads models on demand.
    c. Only one fp16 NPC model is resident at a time to respect VRAM limits.

The following sections detail each stage.

STAGE I – Seed Generation

Persona-Aware Seed Design

The seed generation treats the teacher model as a content planner rather than as the final NPC. The custom-built Python script defines:

- A persona brief describing canonical lore and constraints for the NPC (weapon, enemies, clothing, interests, tone)

- Valid tones, settings and categories, such as:
    - **Tones**: [“solemn”, “playful”, “teasing”, “stern”, “compassionate”, …]
    - **Settings**: “tavern, “market”, “shrine”, “abbey courtyard”, etc.
    - **Categories**: “prayer/faith”, “after-battle triage”, “ethics/morality”, etc.

The system prompt instructs the model to output JSON Lines (JSONL) with a fixed schema per line:

- “seed” – a short (<18 words) concrete situation.
- “tags” – 2 – 4 short labels.
- “tone” – one of the allowed tone settings.
- “setting” – one of the allowed settings.
- “lore_targets” – 1-2 specific world details to weave in later.

A user-level template then requests:

> “Generate exactly *N* unique topic seeds for the category *X*. Return JSONL: One JSON
>
> Object per line, no extra text.”

Robust JSON Parsing

Because LLMs sometimes deviate from strict formatting, the script implements a robust parser:

- Attempts to parse each line as JSON.
- Falls back to *ast.literal_eval* for pseudo-JSON with single quotes.
- If JSONL fails, attempts to recover from:
    - A top-level array of objects.
    - Arbitrary text containing balanced {….} blocks.

Helper functions like fixing trailing commas, extracting top arrays, extract JSON objects, help clean any malformed output.

Each parsed object is normalized into a row: *seed, target_str, tone, setting, lore_target_str*

Validation includes:

- Enforcing max seed length (~20 words).
- Normalizing tones and settings against allowed lists.
- Stripping noise and whitespace.

Category Balancing and Deduplication

The custom-built function, *generate_seeds*, orchestrates generation:

- Accepts a desired total seed count (e.g., 250).
- Distributes counts across categories using balancing counts.
- For each category:
    - Calls custom Python code to request the teacher to output *n* seeds.
    - Cleans and deduplicates seeds by forcing all seeds to lower by category.

If the total count is still below the target after one pass, the script performs a top-up phase, sampling additional seeds from categories until the quota is met. Finally, each seed is assigned a unique *id* and written to a TSV file. The TSV becomes the canonical seed catalog for the NPC.

STAGE II – Response Generation

Persona-Locked Teacher Prompt

The second stage script treats the teacher model as the NPC itself, not a planner. A system persona string defines the NPC's voice and constraints:

- Witty, flirty, non-nonsense human cleric.
- Canon details (weapon, enemies, shop, weapon, armor, etc.).
- "Speak only in-world."
- Vary rhythm (mix short punchy lines and longer sentences).

The user template is structed as:

- Category, seed, tags, tone, settings, lore_targets.
- Explicit constraints:
    - Mention lore targets naturally where possible.
    - Keep canon consistent.
    - No titles, preambles, or out-of-character explanations.
    - Output only the NPC's speech/text.

This encourages the model to generate content that is rich, but tightly bound to the persona and world.

Generation Loop and Validation

The function defined as *forge* drives the process of generation and validation:

1. Load seeds from the TSV file using a Python routine.
2. For each seed:
    a. Generate a target number of variants (e.g. *per_seed=3*).
    b. Use *gen_one(...)* to call the teacher model with the system persona and user brief.
3. Validate each candidate response:
    a. Minimum word count (*min_words*, eg.110).
    b. Reject outputs mentioning AI-related leak phrases (e.g. “as an AI”, “language model”, “Qwen”).
    c. Normalize whitespace and text.
    d. Deduplicate using a SHA-1 hash (*response_hash)* of the normalized text across the entire run.

If a variant fails validation, the script retries up to a configurable number of times. If still invalid, the variant is skipped an logged.

JSONL Output and Resumability

Each accepted response is appended to a JSON object to an output JSONL file.

The script is resumable:

- *load_existing_jsonl(...)* scans the output file to find how many variants already exists per seed.
- On rerun, it skips completed seeds and continues generating more response until the configured *per_seed* count is reached.

This design allows incremental dataset growth without manual bookkeeping. The resulting JSONL file forms the core synthetic training corpus for the NPC.

STAGE III – Full Fine-Tuning on Llama-3.2-3B-Instruct

The Python script performs a full fine-tune on a base model (meta-llama/Llama-3.2-3B-Instruct) using only the NPC’s synthetic responses.

Data Loading and Wrapping

The script supports two input formats:

- Plain text (.txt) one sample per line.
- JSONL (.jsonl) expects a “response” field.

For JSONL file types:

- Filters out short responses using a minimum word threshold (e.g. *min_words=60)*.
- Wraps each response with explicit boundary tokens

These markers help the model distinguish sample boundaries in the packed training sequences. If <BOS> / </BOS> tokens are not preset in the tokenizer vocabulary, they are added and the model's embeddings resized accordingly.

Tokenization and Sequence Packing

The dataset is tokenized with:

- Truncation to a fixed *seq_len (e.g. 512)*.
- No additional special tokens beyond what is already in the text.

To improve training efficiency, the script packs multiple responses into contiguous sequences of fixed length:

- Concatenates all *input_ids*, inserting *eos_token_id* between samples.
- Computes the largest multiple of *seq_len* that fits.
- Slices the concatenated sequence into *seq_len* chunks and constructs matching *attention_mask* arrays.

This "packing" reduces padding and makes better use of GPU compute.

Training Configuration

Key training arguments:

- Model: *meta-llama/Llama-3.2-3B-Instruct*
- Precision: *bf16* or *fp16* on GPU; *fp32* on CPU
- Gradient checkpointing enabled to reduce VRAM usage
- Attention implementation forced to "eager" for stability.
- Batch size: small per device (e.g. *batch=1*) with higher *gradient_accum* (e.g. 24) to simulate larger effective batch sizes.
- Optimizer:
    - Default: Adafactor (memory-efficient, via Adafactor + AdafactorSchedule).
    - Optional: *adamw_torch* or 8-bit AdamW via bitsandbytes if requested.
- Learning rate: ~6e-6
- Epochs: typically ~1.0-1.5 passes over the packed dataset.
- Weight decay and warmup ratio configured modestly.

The script uses Hugging Face *Trainer* with a custom optimizer when Adafactor is selected. Evaluation is optional; for small, tightly targeted datasets, a tiny held-out split (e.g. 1% of sequence) is used when the dataset is large enough. At the end of training, the script saves the fine-tuned model weights (in safetensors format) and the updated tokenizer including BOS if added. The output directory becomes the deployable NPC model folder.

INFERENCE AND DEPLOYMENT

Although the deployment script is not shown in this paper, the architecture is straightforward:

- The fine tuned NPC models are stored in separate directories.
- A Python inference server (e.g. FastAPI or Flask) exposes endpoints such as:
    - GET /list – List available NPCs.
    - POST /chat/{npc} – Send a message and receive the NPC's reply.
    - POST /preload/{npc} – Proactively load an NPC model into memory.
- The server maintains an in-process cache:
    - At most one fp16 Llama-3.2-3B NPC model resident at a time (to fit withing VRAM).
    - When a new NPC is requested:
        - If no model is loaded, load the requested one.
        - If another NPC is loaded, delete it and free CUDA memory, then load the new model.
- Requests are serialized or lightly queued to avoid overlapping model loads.

This "one-NPC-at-a-time" constraint keeps the system simple and GPU friendly while still allowing a large roster of NPCs behind a single API surface. Clients (e.g. C# frontends, webUIs, or a BBS system) call the Python API and treat each NPC as a stable conversational endpoint.

EVALUATION

Formal quantitative evaluation of NPC persona quality is challenging, but several practical criteria emerge:

- Persona consistency
    - Does the NPC maintain a coherent voice, tone, and worldview across long conversations?
    - Does it respect canonical facts (weapons, location, enemies, relationships)?

- Style stability across seeds
    - Responses derived from different categories and seeds (e.g. “after-battle triage” vs. “festivals/holy days”) should still sound like the same character.
- Absence of OOC behavior
    - The model should avoid breaking character, mentioning AI systems, or exposing training process details.
- Diversity within constraints
    - Even with a strong persona lock, the NPC should not become repetitive or template-like.

Empirically, this pipeline yields NPCs that exhibit:

- Strong alignment with their persona beliefs.
- Low incidence of AI self-reference due to validator filters.
- High continuity in how they discuss recurring lore (e.g. specific towns, enemies, or relationships).

Future work could introduce automatic metrics, such as embedding-based similarity to seed descriptors, classifier-based persona adherence scores, or human evaluation protocols.

LIMITATIONS AND FUTURE WORK

While effective, the system has several limitations:

- **Data source homogeneity** – all training data is synthetic, generated from a single teacher model; style diversity is constrained by the teacher’s own biases.
- **Per NPC cost** – each major NPC requires its own fine-tuning run and model storage, which, while feasible for 3B models, may become expensive for very large casts.
- **No explicit memory module** – NPC “memory” is encoded implicitly in weights rather than via an external memory or RAG mechanism; this simplifies design but limits dynamic, long-term learning.

Future directions include:

- Combining this persona-locked fine-tune with lightweight RAG for dynamic world state.
- Exploring smaller base models or quantization for lower-cost deployment.
- Introducing shared base layers with light per-NPC adapters to reduce storage while preserving identity.
- Developing automated persona adherence benchmarks for systematic evaluation.

CONCLUSION

This paper has presented a practical, end-to-end pipeline for building dedicated NPC language models using synthetic persona-aligned data. By decomposing the process into seed generation, response expansion, and full fine-tuning, the system transforms a structured character brief into a deployable 3B-parameter NPC model with stable voice and lore.

The use of a strong local teacher model (Qwen2.5-7B-Instruct) for data generation and a smaller student model (Llama-3.2-3B-Instruct) for deployment strikes a pragmatic balance between quality and cost. A Python-based inference service with lazy-loaded fp16 models, enables multiple NPCs to share a single GPU while maintaining strong persona isolation.

In contrast to prompt-only persona conditioning, this approach treats each major NPC as a first-class model, with its own corpus, weights, and API entry point. For narrative-drive systems, games, and interactive worlds, such dedicated NPC brains offer a robust foundation for long-term, coherent storytelling.