stylgen_v0 – Team Guide

This guide explains how stylgen_v0 works, how to run it locally with Ollama, what each subsystem does (LLM provider, embeddings, retrieval, pipeline, streaming), and how to extend it safely.

Contents

Audience & Goals

- Audience: Engineers familiar with Python/REST, new to this codebase.
- Goals: Run locally, understand internals, and extend with confidence.

Quickstart

Prereqs

- Python 3.10+
- uv (https://docs.astral.sh/uv/)
- Ollama installed and serving, with `llama3:8b` pulled

Install & Run (two-terminal flow)

- Terminal A:

```
# optional logging
export STYLGEN_LOG_LEVEL=DEBUG
export STYLGEN_DEBUG=1
# LLM config
export OLLAMA_BASE=http://127.0.0.1:11434
export OLLAMA_MODEL=llama3:8b
# optional CORS for browser testing
export STYLGEN_DEV_CORS=1
```

```
uv sync
make run
```

- Terminal B:

```
curl -s http://127.0.0.1:8000/health
make e2e BASE=http://127.0.0.1:8000
```

One-shot E2E (single terminal)

```
make e2e-local
```

This starts the server, waits for health, runs the E2E script, and shuts down.

Streaming demo

```
curl -N -s -X POST http://127.0.0.1:8000/generate/stream \
  -H 'content-type: application/json' -d '{
    "user_id": "u1",
    "brief": {"keywords": ["onboarding"], "goal": "educate"},
    "llm_options": {"temperature": 0.7, "num_predict": 256}
  }'
```

Architecture

High-level flow

1. Persona creation: user samples + preferences → Persona Card (centroid embedding + exemplar IDs)
2. Retrieval: find style exemplars for grounding
3. Prompting: build system + user prompts from Persona + Brief
4. LLM drafting: generate 1–N variants (Ollama or Dummy fallback)
5. Critique & scoring: ban phrases, soft length; score style similarity & novelty
6. Selection & feedback: return variants, accept feedback for learning later

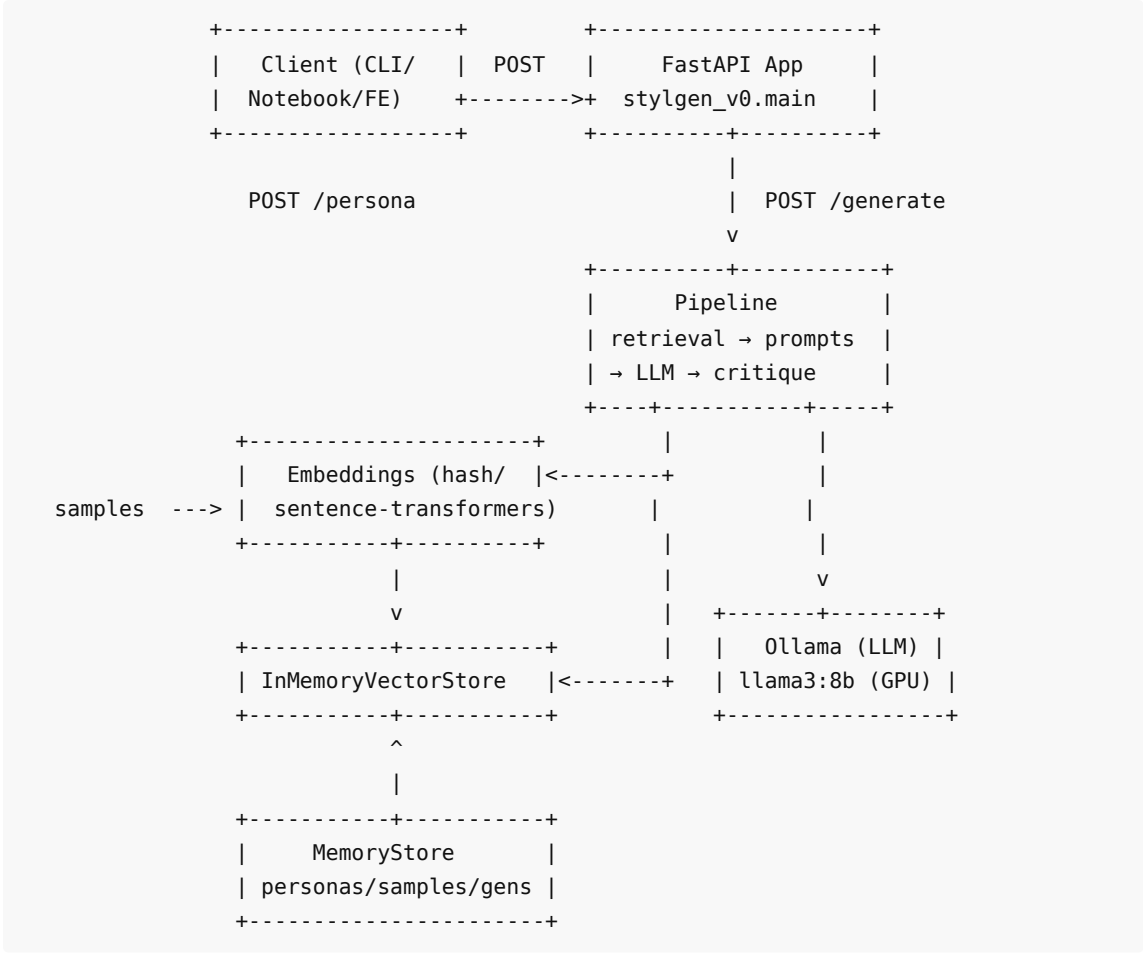
Key modules

- `stylgen_v0/main.py` : FastAPI app & routes; wiring and logging
- `stylgen_v0/models/schemas.py` : Pydantic request/response models
- `stylgen_v0/core/embeddings.py` : Hashing embedder, cosine, optional ST embedder
- `stylgen_v0/core/vector_store.py` : In-memory per-user vector index
- `stylgen_v0/core/persona.py` : Persona builder (centroid from sample embeddings)
- `stylgen_v0/core/llm.py` : LLM provider interface; Ollama & Dummy providers; streaming
- `stylgen_v0/core/pipeline.py` : Orchestrates retrieval → prompts → LLM → critique → score
- `stylgen_v0/storage/memory.py` : In-memory store for personas, samples, generations

Lifecycle notes

- All state is in-memory (single process). Restarting clears personas/samples/generations.
- LLM calls are stateless and depend only on the request + Persona Card.

Architecture diagram



Configuration

Core environment variables

- `OLLAMA_BASE` (default `http://127.0.0.1:11434`) — Ollama server URL
- `OLLAMA_MODEL` (default `llama3:8b`) — model name/tag for generation
- `STYLGEN_DEV_CORS=1` — enable permissive CORS in dev
- `STYLGEN_LOG_LEVEL` — `DEBUG` / `INFO` / `WARNING`; default `INFO`
- `STYLGEN_DEBUG=1` — log system/prompt previews and exemplar snippets
- `HOST` / `PORT` — used by Makefile/scripts

Embeddings config (optional)

- `STYLGEN_EMBEDDER=st` — switch to sentence-transformers embedder
- `STYLGEN_ST_MODEL` — sentence-transformers model name (default `intfloat/e5-large-v2`)
  - Requires: `uv sync --extra hf-embeddings`

Configuration diagram

```
Env Vars                App Wiring                              Behavior
--------                ----------                              --------
OLLAMA_BASE/MODEL  ---> OllamaProvider <--------------------> /generate[.stream]
STYLGEN_EMBEDDER   ---> HashingEmbedder | STEmbedder (auto fallback)
STYLGEN_ST_MODEL   ---> STEmbedder(model_name)
```

```
    STYLGEN_LOG_LEVEL  ---> logging level (INFO/DEBUG)
    STYLGEN_DEBUG      ---> prompt/system/exemplar previews in logs
    STYLGEN_DEV_CORS   ---> permissive CORS in dev
```

Data Models

PersonaCard (stored per user)

- `user_id` : user key
- `preferences` : tone descriptors, taboo phrases, formality (1–5), emoji/hashtags, structure
- `exemplar_ids` : the sample IDs we'll prefer as few-shot exemplars
- `centroid` : List[float] — centroid embedding of the user's samples

Requests

- `PersonaCreateRequest` : `{ user_id, samples: string[], preferences }`
- `GenerationBrief` : keywords, goal, audience, cta, length_hint, emoji, link
- `GenerationRequest` : `{ user_id, brief, num_variants=2, llm_options? }`
- `FeedbackRequest` : `{ user_id, generation_id, rating, tags?, edit_diff? }`

Responses

- `PersonaCreateResponse` : echo user, count, and PersonaCard
- `GenerationResponse` : `{ user_id, generation_id, chosen, variants }`
  - `GenerationVariant` : `{ text, score { style_similarity, novelty, structure_ok, length_ok } }`

Data model diagram

```
          +-------------------------+                +-------------------------+
          |         User            |                |      PersonaCard        |
          |  (identified by user_id)| 1 ---------- 1 |  user_id (FK to User)  |
          +-------------------------+                |  preferences            |
                                                     |  exemplar_ids [id]      |
                                                     |  centroid [float]       |
                                                     +-------------------------+

          +-------------------------+                +-------------------------+
          |     WritingSample       |  1..*      1   | InMemoryVectorStore     |
          |  id, text               | <------------  |  VecItem{id, text, vec} |
          |  (per user in Memory)   |   references   |  (per user namespace)   |
          +-------------------------+     ids        +-------------------------+

          +-------------------------+
          |    GenerationRecord     |
          |  id (generation_id)     |  * (per user)
          |  user_id  (FK to User)  | <--------------
          |  variants [GenerationVariant]
          |  chosen_index           |
          +-------------------------+

Feedback: `/feedback` references GenerationRecord.id (generation_id) and user_id for
ownership check.
```

LLM Stack (Ollama)

Provider abstraction (`LLMProvider`)

- `generate(prompt, system, temperature, options)` → str
- `stream_generate(...)` → Async generator yielding chunks

Ollama provider (`OllamaProvider`)

- Non-stream: `POST /api/generate` with `stream=false`
- Stream: `POST /api/generate` with `stream=true` (SSE lines)
- Per-request `options` (forwarded to Ollama):
  - `temperature` , `top_p` , `num_predict` , `repeat_penalty` , etc.
- Throughput logging: if response includes `eval_count` and `eval_duration` , we log tokens/sec at DEBUG.

LLM sampling primer (what the options mean)

- Temperature: flattens/sharpens probabilities. Lower (0.2–0.5) → safer/more deterministic; higher (0.8–1.0) → more creative/variable.
- Top-p (nucleus sampling): sample from the smallest set whose cumulative probability ≥ p (e.g., 0.9 prunes long tails).
- Top-k: limit to the top-k tokens by probability (often combined with top-p). Smaller k = more focused.
- Repeat penalty: discourages recent token repeats (values 1.05–1.2 help avoid loops).
- Num predict: max new tokens to generate. Use larger values (256–1024) for richer drafts and more stable throughput measurements.
- Seed: fix randomness for reproducibility.

Prompt format

- System prompt encodes persona constraints (tone, structure, taboo list, first-person, concrete detail, CTA).
- User prompt encodes the brief (goal, audience, keywords, constraints) + exemplar snippets as few-shot style guidance.
- We use `/api/generate` (not `/api/chat`) to keep explicit control over the prompt format and avoid implicit chat turns.

Tokens, throughput, latency

- Tokens are subword units (e.g., BPE). A word is not a token; common English words can be <1 token on average, rare ones 3+.
- Ollama exposes `eval_count` and `eval_duration` (ns). Tokens/sec = `eval_count / (eval_duration / 1e9)` .
- First-token latency is higher than steady-state decode; batch length/history affects KV cache size and speed.

GPU, quantization, offload (conceptual)

- Quantization (GGUF) reduces memory/VRAM with modest quality trade-offs (e.g., Q4/Q5 variants).
- Offload (`gpu_layers`) moves model layers to GPU. More layers on GPU → faster, but higher VRAM. Tune per GPU.
- KV cache stores past attention keys/values to accelerate decoding; longer outputs increase memory use.

Request flow (non-stream)

```
Client
  │  POST /generate {brief, llm_options}
  ▼
FastAPI (main.py)
  │  persona = MemoryStore.get(user)
  │  variants = Pipeline.generate(persona, brief, options)
  ▼
Pipeline
  │  ex_texts = VectorStore.top_k(centroid)
  │  system = build_system(persona)
  │  prompt = build_prompt(brief, ex_texts)
  │  text = Ollama.generate(prompt, system, options)
  │  text' = critique(text)
  │  score = style_sim(text', centroid) + novelty(text', samples)
  │  sort & choose
  ▼
FastAPI
  │  MemoryStore.add_generation(record)
  └─► Response {generation_id, chosen, variants}
```

Request flow (stream)

```
Client
  │  POST /generate/stream {brief, options}
  ▼
FastAPI
  │  ex_texts, system = Pipeline.prepare_generation_context(...)
  │  prompt = build_prompt(brief, ex_texts)
  │  stream = Ollama.stream_generate(prompt, system, options)
  └─► SSE: meta → data (chunks...) → done
```

Dummy provider (`DummyProvider`)

- Deterministic fallback text (ensures pipeline works even if Ollama is unavailable).
- Used only on LLM failure or in tests.

GPU notes

- GPU use is handled by Ollama. Confirm with `nvidia-smi` during generation.
- To tune offload, create a custom model with a `Modelfile` and `PARAMETER gpu_layers <N>` and run that model.

Embeddings & Retrieval

Hashing embedder (default)

- Tokenizes to lowercased whitespace tokens; hashes into a fixed-size vector (default 384 dims)
- Normalized TF-like vector; cosine similarity
- Pros: small, fast, zero heavy deps; Cons: weak semantics

Sentence-Transformers embedder (optional)

- Enable: `uv sync --extra hf-embeddings` and `export STYLGEN_EMBEDDER=st`
- Model: `STYLGEN_ST_MODEL` (default `intfloat/e5-large-v2`)

- Recommended on 3090:
  - `intfloat/e5-large-v2` — high-quality English semantic embeddings
  - `BAAI/bge-m3` — multilingual, strong retrieval
- Tips:
  - Keep `normalize_embeddings=True` (we pass normalized vectors to cosine)
  - Batch larger text sets for speed; monitor memory
  - If import fails, app logs a warning and falls back to hashing

Embedding primer

- Transformer encoders (E5/BGE) map text to dense vectors that capture semantic similarity; cosine distance correlates with topical/stylistic closeness.
- Normalization: cosine = dot product on unit vectors, so we normalize to make comparisons length-invariant and more stable.
- Pooling: most ST models use mean pooling; some use CLS. E5 often expects task prefixes (e.g., “query:”/“passage:”). We omit them here to bias toward style over instruction cues; add them if you prioritize topical retrieval.

Vector similarity & ANN

- We use exact cosine within a small per-user corpus. At larger scales, use approximate nearest neighbor (ANN) indices (FAISS/HNSW/Qdrant) for sub-linear retrieval.
- Partition by user (namespaces) to protect privacy and avoid cross-user style blending.

Vector store

- `InMemoryVectorStore` : per-user list of (id, text, embedding)
- `top_k(user_id, query_vec, k)` : cosine similarity
- `/persona` uses `replace()` to avoid unbounded duplicates on updates

Embeddings & retrieval diagram

```
User samples ──► Embedder (hash/ST) ──► sample embeddings
                                   │            │
                                   │            └─► Persona centroid (mean, normalized)
                                   └─► VectorStore.add(user, items)

On generate:
Persona centroid ──► VectorStore.top_k(user, centroid, k) ──► exemplar texts
```

Generation Pipeline

System prompt

- Derives from Persona preferences: tone, formality, emoji, hashtags, structure, banned phrases (global + per-user)
- First-person, concise sentences, include concrete detail + clear CTA

User prompt

- From `GenerationBrief` : goal, audience, keywords, constraints (length ~N chars, emoji policy, hashtags), CTA, optional link
- Injects exemplar snippets as guidance

Drafting

- Generate N variants (default 2)
- Temperature policy: if `llm_options.temperature` present, use it; else alternate 0.6/0.8
- Per-variant timing logged; on error, fall back to Dummy

Critique

- Remove banned phrases with case-insensitive, word-boundary regex
- Normalize spaces/tabs while preserving line breaks
- Soft length check: within ~60–140% of `length_hint`

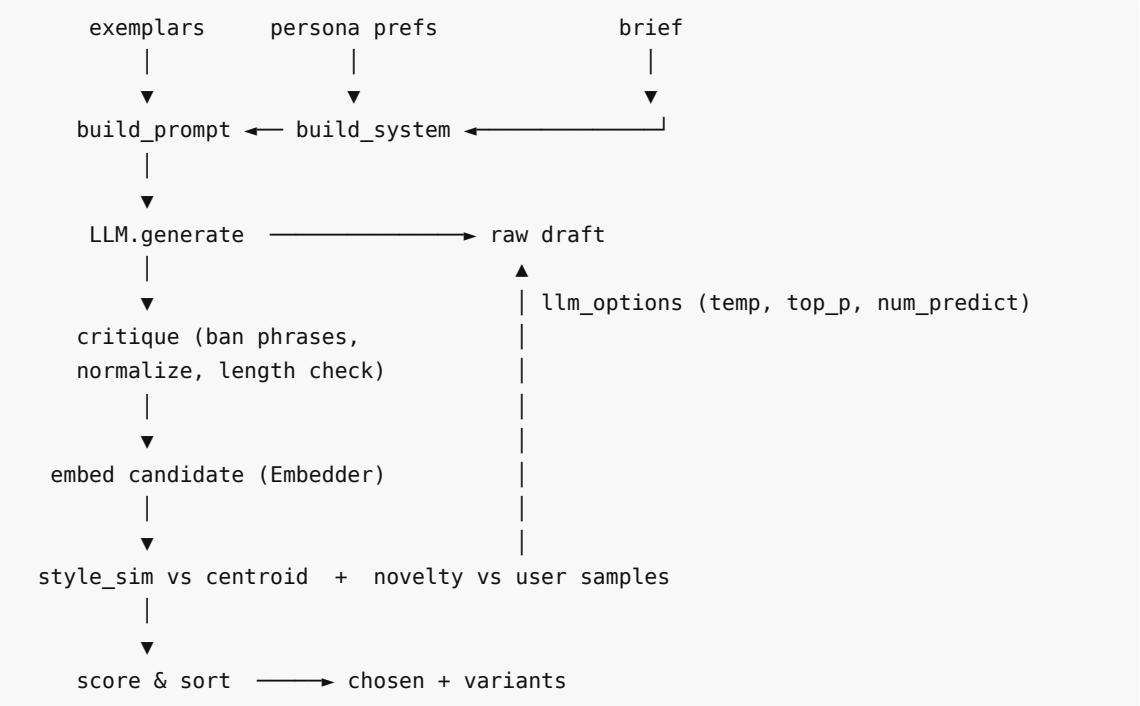
Beyond the basics (ideas to extend)

- Structure checks: verify a hook/lesson/CTA skeleton with simple heuristics (line counts, imperative verbs, CTA markers).
- Readability: track average sentence/paragraph length to match persona cadence; flag walls of text.
- Hashtags: de-duplicate, enforce niche vs generic ratio based on preferences.
- Novelty: add n-gram overlap penalties vs user corpus; maintain a recent-hook registry to encourage variety across posts.

Scoring & selection

- Style similarity: cosine(candidate, persona centroid)
- Novelty: `1 − max_sim(candidate, all_user_samples)`
- Composite: prioritize style + novelty, tie-break by `length_ok`
- Sort and choose top as `chosen`

Pipeline diagram



Endpoints & Contracts

Health

- `GET /health` → `{ "status": "ok" }`

Persona

- `POST /persona` → replaces user's samples and vectors; returns PersonaCard

```
{
  "user_id": "u1",
  "samples": ["...", "..."],
  "preferences": {"tone_descriptors": ["forthright"], "emoji_ok": true}
}
```

Generate (non-stream)

- `POST /generate` → returns `generation_id`, `chosen`, `variants`
- Supports `llm_options`:

```
{
  "user_id": "u1",
  "brief": {"keywords": ["onboarding"], "goal": "educate", "length_hint":
900, "emoji": true},
  "num_variants": 2,
  "llm_options": {"temperature": 0.7, "top_p": 0.9, "num_predict": 512}
}
```

Generate (stream)

- `POST /generate/stream` → `text/event-stream`
- Events: one `event: meta`, many `data:` chunks (raw text), final `event: done`
- Note: Critique/scoring do not run during streaming; use non-stream for final scored variants

Feedback

- `POST /feedback` → `{ "status": "received" }` if `generation_id` exists and belongs to `user_id`

Streaming (SSE)

Server-Sent Events format

- Each event is separated by a blank line (`\n\n`)
- We send:
  - `event: meta\ndata: { ... }\n\n` (exemplar previews, goal, keywords)
  - `data: <text chunk>\n\n` repeated while tokens stream in
  - `event: done\ndata: end\n\n` on completion

Client patterns

- curl: `curl -N -s -X POST ...`
- JS: EventSource only supports GET. For our POST endpoint, use `fetch()` and stream the response body (ReadableStream). See the sketch below.
- Python: `httpx.AsyncClient().stream("POST", ...)` and iterate `.aiter_text()`

SSE sequence (server → client)

```
POST /generate/stream
  server: event: meta\n
          data: {exemplars:[...],goal:'...',keywords:[...]}\n\n
          data: <chunk1>\n\n
          data: <chunk2>\n\n
          ...
          event: done\n
          data: end\n\n
```

Logging & Observability

Server logs (set `STYLGEN_LOG_LEVEL`)

- Persona: `persona.create` (start), `persona.created` (result)
- Generate: `generate.request` (input summary), per-variant DEBUG metrics, final `variants.sorted` + `generate.done`
- LLM throughput: tokens/sec logged at DEBUG when available (non-stream and stream completion)
- Debug previews (set `STYLGEN_DEBUG=1`): system/prompt/exemplar snippet previews

Suggested additions

- Request IDs; structured JSON logs per request; explicit stage timers
- Metrics: export counters/histograms (request latency, tokens/sec, error rates) via Prometheus.
- Tracing: OpenTelemetry spans for retrieval, prompting, LLM, critique, scoring.

Logging flow diagram

```
Client → /generate
  ↳ main: generate.request(user, goal, keywords)
  ↳ pipeline: system/prompt previews (DEBUG when STYLGEN_DEBUG=1)
  ↳ pipeline: variant[i] temp=… llm_ms=… sim=… nov=… length_ok=…
  ↳ pipeline: variants.sorted count=… top_sim=… top_nov=…
  ↳ main: generate.done(user, generation_id, chosen_sim, chosen_nov)
```

Testing & E2E

ASGI tests

- Location: `tests/test_api.py`
- Covers: health, persona→generate→feedback flow, streaming smoke test
- Run: `uv run pytest -q`

E2E scripts

- `scripts/e2e.py`: non-stream end-to-end; prints per-variant scores
- `make e2e BASE=http://127.0.0.1:8000` to run against a live server
- `make e2e-local`: start server → wait → run E2E → stop

Testing/E2E diagram

```
pytest (ASGI) ⟶ httpx.ASGITransport ⟶ FastAPI app (in-process)

e2e_local.sh
```

```
  ├─ uvicorn stylgen_v0.main (bg)
  ├─ wait for GET /health
  └─ run scripts/e2e.py against BASE
```

Notebook

- `notebooks/e2e_demo.ipynb` : step-by-step run with intermediate outputs

Performance & GPU Notes

Measuring

- Non-stream: read `eval_count` / `eval_duration` from Ollama JSON; tokens/sec = `eval_count / (eval_duration / 1e9)`
- App logs include `llm_ms` per variant for wall-clock feel

GPU confirmation

- `nvidia-smi` should show VRAM + utilization during generation
- Longer prompts and `num_predict` (e.g., 512+) give more stable measurements

GPU tuning (Ollama)

- Use a custom model with a `Modelfile` to set `gpu_layers`
- Fit offload to your VRAM; too high → OOM or fallback; too low → slower gen

Extensibility & Roadmap

Short-term

- Provider: add vLLM/OpenAI-compatible provider (respect `LLMProvider` contract)
- Vector store: Qdrant/FAISS adapter; per-user namespace; persistence
- Persistence: Postgres (personas/samples/gen), Redis (queues/cache)
- Critique/rerank: regex expansion, readability metrics, n-gram novelty, rerankers

Longer-term

- Style adapters: LoRA/QLoRA archetypes + optional user micro-adapters
- Preference optimization: ORPO/DPO using selections vs rejects
- Small classifier for “genericness” as a reranker feature

Security & Deployment

- CORS: dev-only `STYLGEN_DEV_CORS=1` ; restrict origins in prod
- Auth: API keys or OAuth in front of endpoints for multi-user scenarios
- Rate limiting: protect LLM budget; consider queues/backpressure
- Multi-process: in-memory stores are single-process; for gunicorn/multiple workers use external stores (DB/Redis/Qdrant)
- Containerization: pass env vars; GPU runtime configuration if using Docker
- PII & content safety: be explicit about what user samples/prompts are stored; add retention/deletion policies; avoid logging full prompts and personally identifiable content.

Troubleshooting & FAQ

- Connection refused: start server (`make run`), check `OLLAMA_BASE` / `OLLAMA_MODEL`
- Persona not found: run `/persona` before `/generate`

- Dummy output (template-like): Ollama unreachable; check service and env
- Streaming is choppy: try smaller `num_predict`; check network proxies buffering SSE
- `sentence-transformers` not found: `uv sync --extra hf-embeddings` and set `STYLGEN_EMBEDDER=st`
- No tokens/sec logs: set `STYLGEN_LOG_LEVEL=DEBUG`; non-stream or final stream event carries metrics
- Streaming timeouts/buffering: some proxies buffer or cut idle SSE connections. Disable proxy buffering for this route, send periodic keep-alives if needed, and increase idle timeouts.

Glossary

- Persona Card: structured style profile per user
- Centroid: average embedding of user's sample texts
- Exemplars: user texts used as few-shot guidance
- Novelty: 1 – max cosine similarity vs user samples
- SSE: Server-Sent Events; unidirectional event stream over HTTP
- Tokens/sec: generation throughput (higher is better)
- Offload: number of transformer layers placed on GPU

Appendix: Common Commands & Example Payloads

Install & run

```
uv sync
export OLLAMA_BASE=http://127.0.0.1:11434
export OLLAMA_MODEL=llama3:8b
make run
```

Persona

```
curl -s -X POST http://127.0.0.1:8000/persona \
  -H 'content-type: application/json' -d '{
    "user_id": "u1",
    "samples": ["...", "..."],
    "preferences": {"tone_descriptors": ["forthright"], "emoji_ok": true}
  }'
```

Generate (non-stream)

```
curl -s -X POST http://127.0.0.1:8000/generate \
  -H 'content-type: application/json' -d '{
    "user_id": "u1",
    "brief": {"keywords": ["onboarding"], "goal": "educate", "length_hint": 900,
"emoji": true},
    "num_variants": 2,
    "llm_options": {"temperature": 0.7, "top_p": 0.9, "num_predict": 512}
  }'
```

Generate (stream)

```
curl -N -s -X POST http://127.0.0.1:8000/generate/stream \
  -H 'content-type: application/json' -d '{
    "user_id": "u1",
    "brief": {"keywords": ["onboarding"], "goal": "educate"},
    "llm_options": {"temperature": 0.7, "num_predict": 256}
  }'
```

Feedback

```
curl -s -X POST http://127.0.0.1:8000/feedback \
  -H 'content-type: application/json' -d '{
    "user_id": "u1",
    "generation_id": "<from /generate>",
    "rating": 4,
    "tags": ["good tone"]
  }'
```

Sentence-Transformers embedder

```
uv sync --extra hf-embeddings
export STYLGEN_EMBEDDER=st
# optional override model
export STYLGEN_ST_MODEL=intfloat/e5-large-v2
make run
```

Streaming in JS (sketch using fetch + ReadableStream)

```
async function streamDraft(body) {
  const resp = await fetch('/generate/stream', {
    method: 'POST',
    headers: { 'content-type': 'application/json' },
    body: JSON.stringify(body),
  });
  const reader = resp.body.getReader();
  const decoder = new TextDecoder();
  let buffer = '';
  while (true) {
    const { value, done } = await reader.read();
    if (done) break;
    buffer += decoder.decode(value, { stream: true });
    // Process SSE lines: events separated by blank line
    let idx;
    while ((idx = buffer.indexOf('\n\n')) >= 0) {
      const raw = buffer.slice(0, idx);
      buffer = buffer.slice(idx + 2);
      const lines = raw.split('\n');
      let event = 'message';
      let data = '';
      for (const line of lines) {
```

```
        if (line.startsWith('event:')) event = line.slice(6).trim();
        if (line.startsWith('data:')) data += line.slice(5).trim();
      }
      if (event === 'meta') console.log('meta', data);
      else if (event === 'done') console.log('done');
      else console.log('chunk', data);
    }
  }
}

// Usage
streamDraft({ user_id: 'u1', brief: { keywords: ['onboarding'], goal: 'educate' }
});
```