■ Streaming A to Z – Mazedaar Style Mein!

■■■ *By: Hammad Bhai x GPT — Code Kings* ■

■ Chapter 1: "Streaming" Ka MatlaB Kia Hai?

Definition:
Streaming ka matlab hai **data ko ek saath bhejne ke bajaye**, **thoda-thoda (chunks/tokens)** karke bhejna.
Socho tum pizza order karte ho, aur wo ek slice bhejta hai, phir doosra slice, phir teesra — **you start eating while the rest is still being delivered**! ■
■ Yehi LLMs (like GPT) mein hota hai jab hum stream=True likhte hain.

■ Chapter 2: Tokens Ka Magic

■ *Token = Word ka chhota hissa*
Sentence vs **Tokens**:
"I love you!" → ["I", " love", " you", "!"]"Streaming is awesome." → ["Streaming", " is", " awesome", "."]* **Har token** ek mini–slice hai.
* Model in tokens ko generate karta hai, ek ek karke.

■ Chapter 3: Normal vs Streaming

FeatureNormal ModeStreaming ModeFirst Response LatencyPoora jawab generate hone tak intezaarPehla token turant, baaki cascadeUser ExperienceSlow typing feelLive "typing..." animation ✍■Network EfficiencySingle large payloadContinuous small payloadsFault ToleranceAgar fail, poora call wastePartial data milta — resilience!

■ Chapter 4: Real-Life Analogy – YouTube vs Download

Download: Poora video file download hone ka intezaar karo (32 GB!).
Streaming: Thoda buffer (5 sec) ke baad **turant playback**, phir background me download hota.
GPT streaming me: buffer word■by■word (ya sentence■by■sentence), phir display.

■■ Chapter 5: Streaming Ka Code (Pure Python)

```
import openai import time openai.api_key = "YOUR_API_KEY" def stream_chat(prompt: str): print(f"\n■
Prompt: {prompt}\n") start = time.time() response = openai.ChatCompletion.create(
model="gpt-3.5-turbo", messages=[{"role": "user", "content": prompt}], stream=True # ■ magic ) for
chunk in response: token = chunk["choices"][0]["delta"].get("content") if token: print(token, end="",
flush=True) print(f"\n■ Stream ended in {time.time()-start:.2f}s.") if __name__ == "__main__":
stream_chat("Explain merge sort like I'm 10 years old.")
```

■ Chapter 6: Use Cases (Jahan Streaming Chamketa Hai)

1. **Chat Interfaces** – ChatGPT typing effect ■
2. **Live Translation** – Speech to text continuously ■
3. **Code Assistants** – IDE suggestions as you type ■■■
4. **Voice Bots** – Real-time speech generation ■■
5. **Game Narration** – Dynamic story unfolding ■

■■ Chapter 7: Intermediate – FastAPI + SSE (Web Integration)

# fastapi_stream.py from fastapi import FastAPI from fastapi.responses import StreamingResponse import openai app = FastAPI() openai.api_key = "YOUR_API_KEY" def event_generator(prompt: str): response = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=[{"role":"user","content":prompt}], stream=True) for chunk in response: token = chunk["choices"][0]["delta"].get("content") if token: yield f"data: {token}\n\n" @app.get("/stream") def stream_chat(prompt: str): return StreamingResponse(event_generator(prompt), media_type="text/event-stream")**Frontend (HTML + JS):**
Live Streaming Chat const prompt = encodeURIComponent("What is AI?"); const evt = new EventSource(`/stream?prompt=${prompt}`); evt.onmessage = e => { document.getElementById("output").innerText += e.data; };

■ Chapter 8: Advanced – Chainlit Integration

# streaming_agent.py from chainlit import llm, Message from chainlit.runner import Runner runner = Runner(api_key="YOUR_API_KEY", model="gpt-3.5-turbo") @llm.stream async def stream_response(prompt: str): async for token in runner.run_streamed(prompt): await Message(content=token).stream() * `@llm.stream` activates streaming in Chainlit.
* `runner.run_streamed` se **async** tokens milte.

■ Chapter 9: Pro Tips & Patterns

1. **Batch Buffering**: collect n tokens then process;
2. **Concurrency**: asyncio.gather for parallel streams;
3. **Backpressure**: throttle frontend updates;
4. **Logging & Metrics**: track token count, latencies.buffer = "" count = 0 async for token in runner.run_streamed(prompt): buffer += token count += 1 if count >= 50: process_batch(buffer) buffer, count = "", 0 await Message(content=token).stream()

■■ Chapter 10: Error Handling & Resume Logic

last_token_id = None async def safe_stream(prompt): global last_token_id try: async for chunk in runner.run_streamed(prompt, start_token=last_token_id): last_token_id = chunk.id await Message(content=chunk.text).stream() except Exception as e: await Message(content=f"[Error: {e}. Resuming...]").send() await safe_stream(prompt) * Store last token ID.
* On failure, reconnect from last_token_id.

■ Chapter 11: Guardrails & Safety

buffer = "" async for token in runner.run_streamed(prompt): buffer += token await Message(content=token).stream() if len(buffer) >= 1000: if violates_policy(buffer): await

Message(content="[Blocked by guardrail]").send() break buffer = "" * violates_policy checks for profanity, PII, hallucinations.
* Stop streaming at violation.

■ Chapter 12: Performance & Production

- Scale SSE/WebSocket with load-balancer.
- Cache common prompts.
- Monitor: token rate, errors, latencies.
- Secure API keys and endpoints.

■ Chapter 13: Summary & Next Steps

1. Beginners: stream=True + loop.
2. Intermediate: FastAPI + SSE.
3. Advanced: Chainlit runner.run_streamed().
4. Pro: buffers, guardrails, error handling, monitoring.

Agla Topic: Context ya Guardrails? Bas bolo:
`*Hammad Bhai, next chapter chahiye!*` ■■