# Live Vision → LLM (Ollama)

**One‑line summary** Feed live camera frames (24 fps capture) into a vision LLM running on Ollama (e.g. llama3.2-vision), and let a user ask questions (“What do you see?”, “What dish can I make with these ingredients?”) which trigger the model to reason about the most recent frame(s). This page provides architecture, flow diagrams, recommended modes, and runnable Python code (on‑demand query mode + REST fallback).

## Assumptions

* Ollama is installed and running locally (default HTTP API at http://localhost:11434).
* A vision model (example: llama3.2-vision) has been pulled with ollama pull llama3.2-vision.
* Host machine has a camera (or RTSP stream) accessible by OpenCV.
* Python 3.9+ environment available.

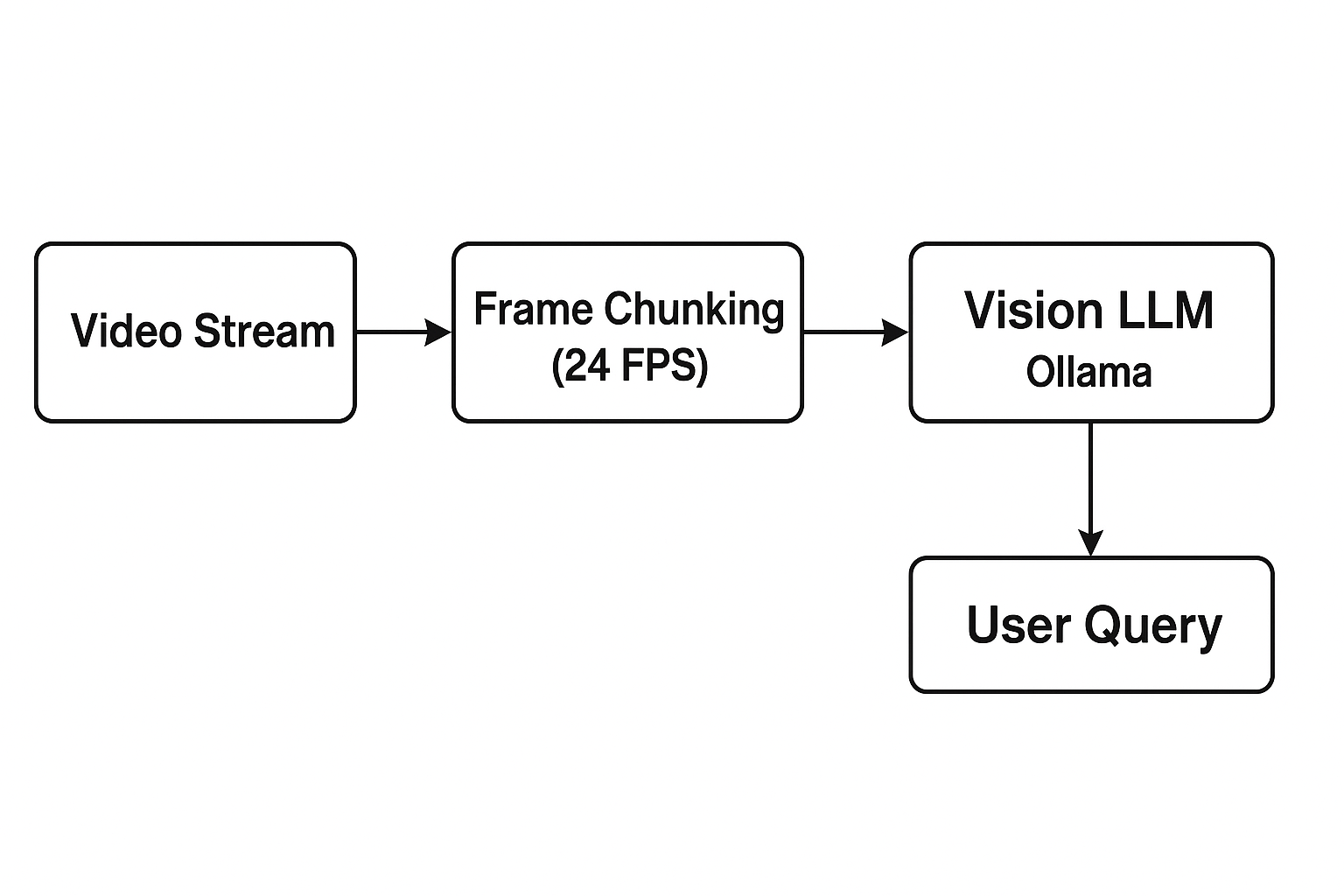
## Goals & recommended mode

**Recommended:** *On‑demand query mode* — capture at 24 fps into a rolling buffer but only call the vision LLM when the user asks a question. This minimizes expensive per‑frame inference while maintaining live capture fidelity.

**Alternate:** *Continuous push (24 fps calls)* — technically possible but extremely resource intensive (high GPU/CPU / latency). Use only with heavy optimization and small/light models.

## System components (short)

1. Camera / Video Source (webcam or RTSP)
2. FrameGrabber thread — captures frames at 24 FPS and keeps a rolling latest frame (and optional short history buffer)
3. FrameQueue (or simple latest frame store) — thread‑safe access point
4. QueryHandler (UI / CLI / API) — receives user questions and triggers a model query
5. ModelClient — wrapper to call Ollama either via ollama Python package or via REST (/api/chat) and pass images (base64)
6. ResponseHandler — formats and returns model text answers to UI



## Flow diagram (component view)

flowchart LR  
 A[Camera / RTSP] --> B(FrameGrabber @ 24 FPS)  
 B --> C{Latest frame store}  
 D[User (CLI / UI / Voice)] --> E[QueryHandler]  
 E -->|"What do you see?"| F{Mode selector}  
 F -->|on‑demand| G[Grab latest frame(s) from C]  
 G --> H[ModelClient (ollama chat/generate)]  
 H --> I[ResponseHandler]  
 I --> D  
 F -->|continuous| J[Stream frames to ModelClient at 24fps]  
 J --> H

## Sequence (on‑demand) — short

1. FrameGrabber fills latest frame every 1/24s.
2. User types question to QueryHandler.
3. QueryHandler asks ModelClient for inference, sending a base64 JPEG of the latest frame along with the text prompt.
4. Model returns text answer; ResponseHandler sends it back to UI.

## Practical recommendations (performance & quality)

* **Compress / resize** frames before sending: 800–1024 px width usually sufficient. Use JPEG quality 70–85 to reduce payload.
* **Send only 1–3 key frames** per query (not the full 24 frames) unless you explicitly need temporal reasoning.
* **Use motion detection or object detection** to avoid queries on blank frames (saves compute).
* **Prefer on‑demand** for general interactive Q&A; reserve continuous streaming only for specialized low‑latency pipelines with dedicated GPU hardware.

## Requirements (quick)

pip install ollama opencv-python-headless requests numpy  
# or if you want GUI preview: pip install opencv-python

Make sure Ollama is running and the model is pulled: ollama pull llama3.2-vision (or whichever vision model you prefer).

## Runnable Python: single file (on‑demand query mode + REST fallback)

"""live\_vision\_ollama.py  
Run: python live\_vision\_ollama.py  
Description: Capture camera at 24 FPS and allow interactive text queries. When user types a question,  
we send the latest camera frame (base64 JPG) to Ollama's vision model.  
"""  
  
import threading  
import time  
import cv2  
import base64  
import json  
import sys  
  
# Try to import ollama; if not available, we'll use REST via requests  
try:  
 import ollama  
 \_HAS\_OLLAMA = True  
except Exception:  
 \_HAS\_OLLAMA = False  
 import requests  
  
class FrameGrabber(threading.Thread):  
 def \_\_init\_\_(self, src=0, target\_fps=24, max\_width=1024):  
 super().\_\_init\_\_(daemon=True)  
 self.cap = cv2.VideoCapture(src)  
 self.running = True  
 self.lock = threading.Lock()  
 self.latest\_frame = None  
 self.target\_fps = target\_fps  
 self.max\_width = max\_width  
  
 def run(self):  
 interval = 1.0 / float(self.target\_fps)  
 while self.running:  
 t0 = time.time()  
 ret, frame = self.cap.read()  
 if not ret:  
 time.sleep(0.1)  
 continue  
 with self.lock:  
 self.latest\_frame = frame.copy()  
 elapsed = time.time() - t0  
 to\_sleep = max(0, interval - elapsed)  
 time.sleep(to\_sleep)  
  
 def stop(self):  
 self.running = False  
 try:  
 self.cap.release()  
 except Exception:  
 pass  
  
 def get\_latest\_frame\_b64(self, jpeg\_quality=80):  
 with self.lock:  
 f = None if self.latest\_frame is None else self.latest\_frame.copy()  
 if f is None:  
 return None  
 h, w = f.shape[:2]  
 if self.max\_width and w > self.max\_width:  
 scale = self.max\_width / float(w)  
 f = cv2.resize(f, (int(w\*scale), int(h\*scale)))  
 ok, buf = cv2.imencode('.jpg', f, [int(cv2.IMWRITE\_JPEG\_QUALITY), jpeg\_quality])  
 if not ok:  
 return None  
 b64 = base64.b64encode(buf.tobytes()).decode('ascii')  
 return b64  
  
  
class ModelClient:  
 def \_\_init\_\_(self, model='llama3.2-vision', host='http://localhost:11434'):  
 self.model = model  
 self.host = host  
 self.use\_ollama\_pkg = \_HAS\_OLLAMA  
  
 def query(self, prompt: str, image\_b64\_list=None, timeout=60):  
 messages = []  
 if image\_b64\_list:  
 messages.append({'role': 'user', 'content': prompt, 'images': image\_b64\_list})  
 else:  
 messages.append({'role': 'user', 'content': prompt})  
  
 if self.use\_ollama\_pkg:  
 # Using the ollama python package  
 resp = ollama.chat(model=self.model, messages=messages, stream=False)  
 try:  
 return resp.message.content  
 except Exception:  
 try:  
 return resp['message']['content']  
 except Exception:  
 return str(resp)  
 else:  
 payload = {'model': self.model, 'messages': messages}  
 r = requests.post(f'{self.host}/api/chat', json=payload, timeout=timeout)  
 r.raise\_for\_status()  
 j = r.json()  
 if isinstance(j, dict) and 'message' in j and isinstance(j['message'], dict) and 'content' in j['message']:  
 return j['message']['content']  
 return json.dumps(j)  
  
  
def interactive\_loop(frame\_grabber: FrameGrabber, client: ModelClient):  
 print("Interactive mode. Type a question and press Enter. Type 'exit' to quit.")  
 while True:  
 try:  
 prompt = input("USER> ").strip()  
 except (EOFError, KeyboardInterrupt):  
 print('\nShutting down...')  
 break  
 if not prompt:  
 continue  
 if prompt.lower() in ('exit', 'quit'):  
 break  
  
 # Grab one (or a few) latest frames. Adjust number if you need temporal reasoning.  
 b64 = frame\_grabber.get\_latest\_frame\_b64(jpeg\_quality=75)  
 if b64 is None:  
 print("No frame available yet. Try again in a moment.")  
 continue  
  
 print("Sending image to model... (this can be slow depending on model/hardware)")  
 try:  
 answer = client.query(prompt, image\_b64\_list=[b64])  
 print("MODEL>", answer)  
 except Exception as e:  
 print("Error calling model:", e)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 src = 0 # change to RTSP URL or video file as needed  
 fg = FrameGrabber(src=src, target\_fps=24, max\_width=1024)  
 fg.start()  
 client = ModelClient(model='llama3.2-vision')  
 try:  
 interactive\_loop(fg, client)  
 finally:  
 fg.stop()  
 time.sleep(0.2)  
 print('Stopped')

## How this satisfies the original idea

* The camera is captured at 24 FPS in the background so “live” data is available.
* The user can ask questions at any time; only then is the model invoked with the most recent frame, saving compute.
* The code includes both the ollama python client path (preferred) and a REST fallback if the package isn’t installed.

## Notes & next steps

* If you truly need frame‑by‑frame processing at 24 FPS into the model:
  + Use a lightweight on‑device vision model (object detector) to prefilter frames and only send frames with detected objects to the LLM.
  + Use a smaller vision LLM or quantized model.
  + Consider batching (send N frames as a list) if you need short temporal context.
* For multi‑camera or remote deployments: create a small relay/API service that accepts frames from cameras, runs local prefiltering, and forwards only selected frames to Ollama.

## Troubleshooting checklist

* If you get long delays or memory errors: try a smaller model or reduce frame resolution/quality.
* If the Python ollama client errors: ensure ollama is running (ollama serve) and the model is pulled.
* If no response: check http://localhost:11434/api/models (Ollama endpoint) to verify available models.

## Appendix: quick prompts to try

* “What are the main objects in this image?”
* “List the ingredients you can see in this photo.”
* “What dish could I make with these ingredients? Give 3 recipes with short steps.”

*End of single‑page design & code.*