Project VisionAssist – Development Journey

Abstract

VisionAssist is a self-contained wearable system designed to assist visually impaired users by combining real-time spatial awareness with visual scene understanding — entirely offline. The project explores the integration of edge-based computer vision, local AI inference, and sensor fusion on constrained hardware such as the Raspberry Pi 5 and ESP32 microcontrollers.

Our work involved evaluating modern vision-language models (VLMs), optimizing them for ARM architectures, and building supporting hardware for environmental awareness. After extensive experimentation with Moondream 2 and Florence 2 — including quantization, ONNX conversion, and cross-model bridging — we achieved full local inference using the BLIP model at real-time practical speeds. Through iterative design, Linux-based development, and custom sensor simulation tools, VisionAssist evolved into a modular proof-of-concept that demonstrates how assistive AI can run autonomously, without cloud dependence.

1. Conception of the Idea

The idea for VisionAssist originated in early April 2025, during discussions about assistive robotics. We had already been exploring navigation algorithms, sensor integration, and AI-based perception for robotics when we asked: “If a robot can navigate and interpret its environment, why can’t a human use similar technology to ‘see’ through AI?”

That question led to the concept of a wearable vision assistant — a small, self-contained system capable of describing the environment and warning of nearby obstacles, entirely offline.

2. Research and Literature Review

Before building anything, we conducted a literature review to determine how similar systems approached the problem. We found that most existing solutions relied on cloud-based AI and required constant internet connectivity. Only a few experimental projects achieved partial on-device inference, and none combined both vision

understanding and real-time spatial sensing.

From this review, we defined our non-negotiable design principles:

1. Portability: The system must be lightweight and wearable.

2. Full Local Operation: It must function without any external servers or online dependencies.

All processing — from image capture to speech output — had to happen on the device itself.

3. Sensor and Microcontroller Exploration

We began by testing available hardware components: ESP32 microcontrollers (FireBeetle 2 ESP32-S3 and ESP32-WROOM), OV2640 camera, GY-US42 ultrasonic rangefinder, Adafruit ICM-20649 IMU, and later replaced the ultrasonic sensor with a VL53L5CX Time-of-Flight (ToF) sensor for 3D depth mapping.

We learned to wire and program the ESP32s, connecting sensors via I^2C and GPIO, testing each component independently. Camera capture, sensor data streaming, and serial communication were validated early — providing a stable hardware foundation for integration.

4. Vision-Language Model Investigation

Next, we investigated suitable Vision-Language Models (VLMs). We began with Moondream 2 (1.8B) due to its small footprint and efficiency. It performed impressively on desktop PCs, generating coherent captions in about 9–10 seconds, but that was too heavy for embedded hardware.

We then tested Florence 2, Microsoft’s transformer-based VLM, which offered better architecture separation (encoder/decoder) but higher compute demands. We concluded that neither model would run efficiently on the Raspberry Pi 5 without significant optimization or compression.

5. Hardware Trade-offs and Platform Decision

We evaluated alternatives such as laptops, NUCs, and microcontrollers, but ruled them out due to power draw, thermal load, and portability constraints. The Raspberry Pi 5 emerged as the best balance of performance and efficiency, sharing the ARM architecture of modern smartphones — aligning with our long-term goal of mobile deployment. This choice forced us into the realm of quantization and runtime

optimization.

5.1 Working Environment and Linux Skills

Before model compression could begin, we set up our development environment on the Raspberry Pi — an essential learning step. We installed Raspberry Pi OS (64-bit) and configured it for headless operation via SSH. Using the Remote Development extension in VS Code, we created a smooth workflow: the Pi handled computation while we developed and debugged remotely from our Windows PCs.

During this phase, we learned to manage packages and virtual environments (apt, pip, and venv), enable hardware acceleration, configure VNC, recover from driver misconfigurations, edit system files (config.txt), manage services, monitor resources, and re-establish network connections after headless reboots. These Linux administration skills proved crucial later when we recompiled frameworks and optimized inference for ARM.

6. The October 9–10 Challenge: Quantization, Model Surgery, and the Bridge Experiments

Our quantization phase began on Thursday, 9 October 2025. The goal was simple on paper: compress Moondream 2 or Florence 2 enough to run on the Raspberry Pi 5. In practice, it became a two-day deep dive into model internals, runtime compilation, and data-type mismatches.

6.1 Moondream: The Quantization Wall

We pursued multiple optimization paths: GGUF conversion using llama.cpp for on-device inference — the model loaded partially, then failed due to RAM exhaustion. ONNX export produced enormous graphs full of unsupported ops; pruning and patching yielded incoherent output. Torch dynamic INT8 quantization worked on x86 but was unsupported on ARM.

We even rebuilt ONNX Runtime from source on the Pi with OpenMP threading and ARM64 flags, but new issues emerged at each turn: operator gaps, memory fragmentation, and dtype mismatches between float16, bfloat16, and int8.

6.2 Florence 2: The Frankenstein Bridge

Undeterred, we pivoted to Florence 2, leveraging its clean split between vision encoder

and text decoder. We constructed a custom bridge to connect the two halves manually: Exported the ViT encoder to ONNX FP16, yielding a 2 048-dimensional embedding. Loaded the GPT-style decoder in PyTorch. Inserted a projection MLP (2 048 → 2 304) to align dimensions. Flattened the model's nested ModuleDict into a linear Sequential chain for external calls. Injected projected embeddings directly into the decoder's EncodedImage dataclass.

It ran briefly — even generating tokens — before collapsing with FrozenInstanceError (immutable dataclasses) and precision mismatches between FP16 and bfloat16 tensors. After patching those, captions degenerated into meaningless but grammatically valid text. We'd built a functional yet semantically broken hybrid — the "Frankenstein Bridge."

6.3 Final Attempts and Realization

By Friday midday, we had implemented multiple quantization pipelines, built a cross-model embedding bridge, recompiled ONNX Runtime, and manually patched precision errors inside Microsoft's model structure. Still, the Pi could not sustain useful inference. At that point, we recognized that our battle was with physics, not software: limited memory, slow storage, and constrained thermal headroom.

By Friday afternoon, we made a strategic decision: "Stop patching the Titanic — build a new boat." This paved the way for our most successful phase — the BLIP implementation.

7. The BLIP Breakthrough

We restarted cleanly with BLIP (Bootstrapped Language-Image Pre-training), a lightweight yet powerful VLM. Within hours, we achieved full inference on the Raspberry Pi 5 using PyTorch CPU execution. After optimizing model loading, trimming runtime overhead, and forcing low-precision math, we achieved 3–5 seconds per caption — fast enough for static scene snapshots.

Results were striking: Inference speed: 3–5 s per caption (on a 4 GB Pi 5). Accuracy: Descriptive, coherent outputs comparable to Moondream. Stability: Long continuous runs without crashes or overheating.

We quickly integrated offline text-to-speech (TTS), achieving our first complete Vision → Language → Speech pipeline — entirely offline.

8. Returning to the Hardware: Sensors and Integration

With the vision module stable, we shifted back to the hardware. We wired the ESP32 with both ToF and proximity sensors, streaming data over serial to the Pi — the assistant’s spatial perception layer.

To keep software development moving, we built a sensor data generator that simulated realistic obstacle distances and motion patterns. This allowed us to develop and test smoothing, filtering, and obstacle detection logic, integrate sensor and vision modules before final hardware assembly, and validate system responses without relying on live hardware. This simulation-first approach enabled continuous progress even during hardware delays.

9. Current Status and Next Steps

By the end of that week, VisionAssist had evolved into a modular edge-AI prototype consisting of: Vision module: BLIP-based image captioning. Speech module: Offline TTS feedback. Sensor module: ESP32 with ToF and ultrasonic sensors. Integration layer: Event-driven sensor processing, smoothing, and audio feedback orchestration. The next milestone is a unified orchestrator to trigger image capture based on sensor events — giving the system contextual awareness and adaptive feedback.

10. Lessons Learned

ARM quantization remains fragmented; documentation and tooling lag behind x86. The Raspberry Pi 5 can handle moderate-sized VLMs if carefully optimized. Architectural separation between sensing, inference, and feedback simplifies scaling. Simpler architectures (BLIP) can outperform more advanced models (Moondream, Florence 2) under real-world constraints. Simulation tools accelerate software readiness before hardware maturity. Remote development through VS Code + SSH transforms the Pi into a practical embedded workstation. Strong Linux literacy and low-level debugging skills proved indispensable.

Conclusion

The VisionAssist project demonstrates that meaningful assistive AI can be achieved without cloud dependence by combining embedded vision, sensor fusion, and efficient model design. Despite hitting physical and software barriers with state-of-the-art VLMs, we built a functioning prototype that perceives, interprets, and communicates

environmental information — all on-device.

The journey reinforced that innovation in constrained environments demands flexibility: when large models fail, smaller, purpose-built ones often succeed. Through this process, we not only delivered a working system but also developed deep proficiency in embedded Linux, AI optimization, and hardware-software integration.

Future work will focus on real-time orchestration, improved spatial reasoning through multi-sensor fusion, and potential migration to smartphone-class ARM SoCs for production-grade wearable deployment.