HandsFree

Voice-Controlled, Location-Aware Personal Agent

Implementation & Technical Roadmap

Hackathon: Cactus x Google DeepMind — FunctionGemma Hackathon

Rubric 2: End-to-end product that uses function calls to solve a real problem

Rubric 3: Low-latency voice-to-action product using cactus_transcribe

One-liner pitch: *"You're driving, hands on the wheel. You speak. Your phone acts instantly — sends your live GPS location, sets alarms, checks weather — all without touching the screen, all processed on-device in under 500ms."*

Why this wins: It solves a real problem (you cannot type while driving, cycling, or cooking), showcases voice-to-action with on-device Whisper transcription (Rubric 3), executes meaningful function calls with real GPS data (Rubric 2), and the hybrid routing shines because simple commands stay blazing fast on-device while complex multi-action requests gracefully fall back to Gemini in the cloud.

Table of Contents

1. Product Overview & Architecture

HandsFree is a voice-first personal agent built on top of the Cactus inference engine and Google's FunctionGemma model. The user speaks a command into their Mac, and the system transcribes the audio on-device, detects if the command involves their current location, grabs real GPS coordinates locally, intelligently routes between on-device and cloud inference, and executes the resulting function calls — all in a single seamless pipeline.

How It Works End-to-End

- **Step 1 — Voice Capture:** User clicks the mic button in the Streamlit app and speaks. Audio is captured via the audio-recorder-streamlit component and saved as a temporary WAV file.
- **Step 2 — On-Device Transcription:** The WAV file is fed into cactus_transcribe (Whisper running locally on Mac via Cactus). No internet needed. Converts speech to text in ~100-200ms.
- **Step 3 — Location Intent Detection:** A lightweight keyword scanner checks if the user mentioned their location — phrases like "send my location", "I've reached", "I'm here". Pure string matching, zero cost.
- **Step 4 — GPS Injection:** If location intent is detected, the app grabs real GPS coordinates using pyobjc and Apple's CoreLocation framework. It reverse-geocodes to a human-readable address. Then rewrites the command: "Send my location to Mom" becomes "Send a message to Mom saying I'm at College Park, Maryland — https://maps.google.com/?q=38.98,-76.93". The Maps link is just a string — no API call.
- **Step 5 — Smart Routing:** A complexity estimator analyzes the query before running inference: How many action words ("and", "also", "then")? How many tools seem relevant? Simple single-tool → FunctionGemma on-device. Complex multi-tool → Gemini cloud. This avoids wasting time on local inference for tasks the small model will fail.
- **Step 6 — Function Calling:** Either FunctionGemma (on-device) or Gemini Flash (cloud) returns structured function calls.
- **Step 7 — Execution & Display:** Function calls are executed (simulated for demo). Results appear on screen with timing breakdowns, routing decisions, and pipeline visualization.

Architecture Diagram

```
User Speaks
|
v
cactus_transcribe (Whisper on-device) ..............  ~120ms
|
v
Location Intent Detection (keyword matching) .......  ~0ms
|
v [if location detected]
CoreLocation GPS Grab + Reverse Geocode ............  ~50ms
|
v
Query Rewriting (inject address + Maps link)
|
v
Complexity Estimator (simple / medium / complex)
```

```
|
+--[simple/medium]--> FunctionGemma (on-device) . ~80ms
|
+--[complex]--------> Gemini Flash (cloud) ...... ~800ms
|
v
Action Executor (execute function calls)
|
v
Display Results + Pipeline Visualization
```

2. Example Scenarios

Scenario 1: Location Sharing

You say: "Send my location to Mom"

```
Transcribe (Whisper, on-device) : "Send my location to Mom" [120ms]
Location detected : GPS: 38.9897, -76.9378 [50ms]
Reverse geocode : "College Park, Maryland" [30ms]
Query rewritten : "Send message to Mom saying
I'm at College Park, Maryland
https://maps.google.com/..."
Complexity: simple (1 tool) : Route: ON-DEVICE
FunctionGemma : send_message(Mom, "I'm at...") [80ms]
Executed : Message sent to Mom
------------------------------------------------------------
Total: 280ms | Source: 100% on-device | No internet used
```

Scenario 2: Context Chain — "I've reached the office"

You say: "I've reached the office"

The user has pre-configured an "Office Arrival" routine. One voice command triggers an entire chain of actions:

```
Action 1: send_message("Mom", "Reached office - [GPS link]")
Action 2: get_weather("College Park")
Action 3: set_alarm(hour=12, minute=30) -- lunch reminder
Action 4: play_music("focus playlist")
------------------------------------------------------------
Total: 350ms | 4 actions from 1 sentence | Fully on-device
```

Scenario 3: Simple Voice Command

You say: "What's the weather in Tokyo?"

```
Transcribe : "What's the weather in Tokyo?" [110ms]
No location intent
Complexity: simple
FunctionGemma : get_weather(location="Tokyo") [75ms]
Executed : Weather: 68F, Clear
------------------------------------------------------------
Total: 185ms | On-device
```

Scenario 4: Complex Multi-Action (Cloud Fallback)

You say: "Text Sarah I'm running late, check weather in NYC, and set a 10 min timer"

```
Transcribe : "Text Sarah I'm running late..." [130ms]
No location intent
Complexity: COMPLEX (3 tools, 2 conjunctions)
Route: CLOUD (Gemini Flash) [800ms]
Action 1: send_message("Sarah", "I'm running late")
Action 2: get_weather("NYC")
Action 3: set_timer(minutes=10)
```

```
------------------------------------------------------------
Total: 930ms | Cloud (correct routing)
```

Scenario 5: Offline Mode

Toggle "Airplane Mode" in sidebar. Wi-Fi off.

You say: "Set an alarm for 6 AM and play morning playlist"

```
Transcribe (still works, on-device) [120ms]
Complexity: medium
FunctionGemma (on-device, no internet) [90ms]
Action 1: set_alarm(hour=6, minute=0)
Action 2: play_music("morning playlist")
------------------------------------------------------------
Total: 210ms | Works without internet
```

Scenario 6: Speed Comparison Mode

Click "Compare Mode". Speak any command. Three panels show the same query processed by FunctionGemma only, Gemini only, and your Hybrid router — side by side with timing bars. Judges instantly see that on-device is 10x faster for simple tasks and cloud is more accurate for complex ones.

3. File Structure

```
functiongemma-hackathon/
|-- main.py # Improved generate_hybrid (keeps benchmark interface)
|-- benchmark.py # Untouched - scoring system
|-- submit.py # Untouched - leaderboard submission
|-- app.py # Streamlit UI - the full demo
|-- voice_pipeline.py # Orchestration: transcribe > detect > inject > route > execute
|-- location_service.py # Mac GPS via CoreLocation (pyobjc)
|-- context_chains.py # Pre-configured action routines
|-- requirements.txt # All dependencies
|-- cactus/ # Cloned cactus repo
|-- python/src/ # Python bindings
|-- weights/ # Model weights
|-- functiongemma-270m-it/
|-- whisper-small/
```

File	Purpose	Modify?
main.py	Hybrid routing logic (generate_hybrid). Benchmark interface must stay intact.	YES
app.py	Streamlit web UI — mic input, location sidebar, pipeline viz, compare mode.	CREATE
voice_pipeline.py	Orchestration: transcribe, detect intent, inject GPS, route, execute.	CREATE
location_service.py	CoreLocation GPS via pyobjc. Reverse geocoding. 30s cache. IP fallback.	CREATE
context_chains.py	Pre-configured routines (office arrival, heading home, morning).	CREATE
benchmark.py	Scoring system. 30 test cases. F1 + speed + on-device ratio.	NO
submit.py	Uploads main.py to leaderboard server.	NO

4. Phase 1: Foundation Setup

Estimated time: Already done (build fixed). Remaining: ~15 minutes for verification.

Step 1.1 — Verify Cactus build

Since you've already fixed the build, verify everything is working:

```
python main.py
```

You should see results from FunctionGemma (on-device), Gemini (cloud), and Hybrid.

Step 1.2 — Download Whisper model

You need the Whisper model for voice transcription:

```
cactus download whisper-small
```

Step 1.3 — Verify API keys

```
cactus auth # enter your Cactus token
export GEMINI_API_KEY="your-key-here" # from Google AI Studio
```

Step 1.4 — Run baseline benchmark

```
python benchmark.py
```

Note your baseline score. This is what you'll improve in Phase 5.

Step 1.5 — Install all dependencies

```
pip install pyobjc-framework-CoreLocation
pip install streamlit
pip install audio-recorder-streamlit
pip install google-genai
```

5. Phase 2: Location Service

Estimated time: 30 minutes

Step 2.1 — Create location_service.py

This file has one class: **LocationService**. It provides three capabilities:

- **_get_corelocation(timeout=10):** Uses pyobjc to access Apple's CoreLocation framework. Creates a CLLocationManager, sets a delegate that listens for location updates, calls startUpdatingLocation(), waits up to 10 seconds for a callback, then extracts latitude, longitude, and horizontal accuracy. This is the same API that native Mac apps use — fully on-device.
- **_reverse_geocode_apple(lat, lng):** Takes coordinates and uses Apple's CLGeocoder to convert them into a human-readable address like "College Park, Maryland, USA". Apple's geocoder uses cached data with minimal network, so it's essentially on-device.
- **LocationService class:** Wraps both functions with a 30-second cache. If get_location() is called twice within 30 seconds, the second call returns instantly from cache. If CoreLocation fails (permissions denied, etc.), it falls back to IP-based geolocation using ipinfo.io/json.

Public Methods

Method	Returns	Description
`get_location()`	dict	Full location: lat, lng, address, maps_link, source
`get_address()`	str	Human-readable address string
`get_coordinates()`	tuple	(latitude, longitude) tuple
`get_maps_link()`	str	Google Maps URL for current location

Step 2.2 — Enable Location Services

Go to **System Settings > Privacy & Security > Location Services**. Make sure it's enabled and that your terminal app (Terminal or iTerm) is listed and allowed.

Step 2.3 — Test standalone

```
python location_service.py
```

You should see your real coordinates and address printed. If it falls back to IP-based, check Location Services permissions.

6. Phase 3: Voice Pipeline

Estimated time: 1 hour

Step 3.1 — Tool Definitions

Define all 7 standard tools (get_weather, set_alarm, send_message, create_reminder, set_timer, play_music, search_contacts) plus one custom tool: **share_location** for explicitly sharing GPS coordinates with a contact.

Step 3.2 — Location Intent Detector

Create a list of keyword phrases that indicate location intent:

```
LOCATION_KEYWORDS = [
"my location", "where i am", "i'm at", "send location",
"share location", "i've reached", "i'm here",
"arrived at", "just arrived", "got to", ...
]

def detect_location_intent(text):
text_lower = text.lower()
return any(kw in text_lower for kw in LOCATION_KEYWORDS)
```

Step 3.3 — Location Injector (Query Rewriting)

This is the key function. If location intent is detected, it grabs GPS, reverse-geocodes, and rewrites the user's command using regex pattern matching:

- "Send my location to Mom" → "Send a message to Mom saying I'm at College Park, Maryland https://maps.google.com/?q=38.98,-76.93"
- "Tell Dad I've reached" → "Send a message to Dad saying I'm at {address} {maps_link}"
- For anything it cannot pattern-match, append the location string to the original text.

Step 3.4 — Complexity Estimator

Analyzes the query **before** it hits the LLM to decide routing:

```
def estimate_complexity(text, tools):
# Count multi-action keywords: "and", "also", "then", "plus"
# Count tool-related keywords matched to distinct tools
# If 3+ tools or 2+ conjunctions -> "complex"
# If 2 tools or 1 conjunction -> "medium"
# Otherwise -> "simple"
```

Step 3.5 — Transcription Wrapper

Wraps cactus_transcribe with error handling:

```
def transcribe_audio(audio_path):
model = cactus_init("cactus/weights/whisper-small")
prompt = "<|startoftranscript|><|en|><|transcribe|><|notimestamps|>"
response = cactus_transcribe(model, audio_path, prompt=prompt)
cactus_destroy(model)
return {"text": result["response"], "time_ms": elapsed}
```

Step 3.6 — Action Executor

Takes function calls and simulates execution. For send_message, returns "Message sent to Mom: I'm at College Park, MD". For get_weather, returns mock weather. This makes the demo feel complete.

Step 3.7 — HandsFreePipeline Class

The main orchestration class. Has two entry points: **process_voice(audio_path)** adds transcription first, then calls process_text. **process_text(text)** runs the full flow: detect location intent, inject GPS, estimate complexity, call generate_hybrid, execute actions. Both return a structured result with every step's output and timing.

Step 3.8 — Test with text input

```
python voice_pipeline.py
```

This runs hardcoded test commands through the pipeline to verify intent detection, location injection, and the full flow before adding voice.

7. Phase 4: Context Chains

Estimated time: 30 minutes

Context chains are pre-configured action routines triggered by a single voice command. When a chain matches, it **skips LLM inference entirely** and executes preset actions immediately — making "I've reached the office" trigger 4 actions in under 100ms.

Step 4.1 — Define Chain Structure

Each chain has: trigger phrases, a list of actions with tool name and arguments, and a description.

```
CONTEXT_CHAINS = {
"office_arrival": {
"triggers": ["reached the office", "arrived at office", ...],
"actions": [
{"tool": "send_message",
"args": {"recipient": "Mom",
"message": "Reached office - {location}"}},
{"tool": "get_weather",
"args": {"location": "{current_city}"}},
{"tool": "set_alarm",
"args": {"hour": 12, "minute": 30}},
{"tool": "play_music",
"args": {"song": "focus playlist"}},
],
},
"heading_home": { ... },
"morning_routine": { ... },
}
```

Step 4.2 — Chain Matching Function

```
def match_context_chain(text):
text_lower = text.lower()
for name, chain in CONTEXT_CHAINS.items():
if any(trigger in text_lower for trigger in chain["triggers"]):
return name, chain
return None, None
```

Step 4.3 — Placeholder Replacement

The {location} and {current_city} placeholders in chain actions get replaced with real GPS data at runtime. When the pipeline detects a matching chain, it calls LocationService, replaces placeholders, and returns the pre-built function calls.

Step 4.4 — Integration with Pipeline

In voice_pipeline.py, add a chain check **before** the LLM inference step. If match_context_chain() returns a match, build the function calls from the chain definition and skip generate_hybrid entirely. This is the fastest path.

Pre-Configured Chains

Chain Name	Trigger Phrases	Actions
Office Arrival	"reached the office", "arrived at office", "got to work"	send_message, get_weather, set_alarm(12:30), play_music(focus)
Heading Home	"heading home", "leaving office", "going home"	send_message, get_weather, play_music(driving)
Morning Routine	"good morning", "start my day"	get_weather, set_alarm(12:00), play_music(morning), create_remi

8. Phase 5: Improve generate_hybrid

Estimated time: 1 hour. This is for Rubric 1 — benchmark scoring.

> **Important:** Modify generate_hybrid in main.py but keep its interface identical so benchmark.py and submit.py still work. The function signature must remain: generate_hybrid(messages, tools, confidence_threshold=0.99)

Step 5.1 — Pre-Routing Complexity Check

Before calling generate_cactus, analyze the query text. If it contains 2+ conjunctions ("and", "also", "then") and seems to need 3+ different tools, skip local inference entirely and go straight to generate_cloud. This avoids wasting ~80ms on local inference that will produce wrong results for complex queries.

Step 5.2 — Tool Count Heuristic

If only 1 tool is provided, FunctionGemma basically cannot pick the wrong one. Set a very low confidence threshold (like 0.3) for single-tool cases. If 5+ tools are provided, be more aggressive about routing to cloud.

Step 5.3 — Dynamic Confidence Thresholds

Instead of one fixed threshold, use different thresholds based on estimated complexity:

```
complexity = estimate_complexity(text, tools)
thresholds = {
"simple": 0.5, # trust local more
"medium": 0.7, # moderate trust
"complex": 0.95, # almost always use cloud
}
```

Step 5.4 — Pattern Caching

Keep a dictionary mapping query patterns to known-good results. After a successful local inference where the result seems correct, hash the query template and cache it. Next time a similar pattern appears, skip confidence checks entirely. Example: "What's the weather in {X}" is a known-good pattern for FunctionGemma.

Step 5.5 — Run Benchmark and Iterate

```
python benchmark.py
```

Your goal: easy cases all on-device (fast), medium cases mostly on-device, hard cases correctly routed to cloud. The scoring formula:

- **F1 accuracy (60%)** — Are the function calls correct?
- **Speed (15%)** — Anything under 500ms gets full marks
- **On-device ratio (25%)** — More local = better
- **Difficulty weighting:** easy 20%, medium 30%, hard 50%

9. Phase 6: Streamlit Web UI

Estimated time: 1.5 hours

Step 6.1 — Page Layout

Three sections: left sidebar for settings/GPS/stats, center for input (mic + text), and results area with pipeline visualization. Page title: "HandsFree" with green color scheme.

Step 6.2 — Sidebar: Location Display

On page load, call LocationService().get_location(). Display the address, coordinates, a "source: on-device (CoreLocation)" badge, and a clickable "Open in Maps" link. Add a refresh button to re-fetch GPS.

Step 6.3 — Sidebar: Context Chains Manager

Show the pre-configured chains (Office Arrival, Heading Home, Morning Routine) with toggle switches to enable/disable each. Optionally let users add custom triggers.

Step 6.4 — Sidebar: Airplane Mode Toggle

A checkbox that disables cloud fallback. When enabled, generate_hybrid only uses generate_cactus and never calls Gemini. Show a plane icon when active. This is a powerful demo moment — turn off Wi-Fi and show it still works.

Step 6.5 — Sidebar: Session Statistics

Track and display: total commands, on-device count, cloud count, on-device %, average latency.

Step 6.6 — Voice Input (Microphone)

Use the audio-recorder-streamlit package:

```
from audio_recorder_streamlit import audio_recorder

audio_bytes = audio_recorder(
text="Click to speak",
pause_threshold=2.0,
sample_rate=16000,
)

if audio_bytes:
# Save to temp WAV file
with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as f:
f.write(audio_bytes)
audio_path = f.name
# Feed into pipeline
result = pipeline.process_voice(audio_path)
```

Step 6.7 — Text Input Fallback

A text box below the mic. Both paths feed into the same pipeline.

Step 6.8 — Pipeline Visualization

After processing a command, show a vertical step-by-step breakdown. Each step is an expandable card (st.expander) showing: step name with emoji, what happened, timing in ms, source badge (green = on-device,

blue = cloud). At the bottom, show total time and summary of actions executed.

Step 6.9 — Compare Mode

A toggle that switches to split-view with three columns. The same command runs through FunctionGemma-only, Gemini-only, and Hybrid. Show timing bars side by side. This is the most visually impressive part of the demo.

Step 6.10 — Run the App

```
streamlit run app.py
```

10. Phase 7: Testing & Submission

Estimated time: 30 minutes

Step 7.1 — Test all demo scenarios

Run through each of the 6 scenarios listed in Section 2. Verify:

- Voice recording captures clean audio
- Whisper transcription is accurate
- Location is grabbed with real coordinates (not IP fallback)
- Context chains trigger correctly
- Simple commands stay on-device
- Complex commands route to cloud
- Compare mode shows all three paths

Step 7.2 — Run final benchmark

```
python benchmark.py
```

Your score should be significantly better than baseline because easy/medium cases stay on-device.

Step 7.3 — Submit to leaderboard

```
python submit.py --team "YourTeamName" --location "YourCity"
```

You can only submit once per hour. Your best score is preserved.

Step 7.4 — Prepare the pitch

See Section 12 for the complete demo script.

11. Time Estimate Summary

Phase	Task	Time
1	Foundation: verify build, download Whisper, install deps	15 min
2	Location Service: CoreLocation + pyobjc + reverse geocode	30 min
3	Voice Pipeline: orchestration, intent detection, GPS injection	1 hr
4	Context Chains: pre-configured routines, chain matching	30 min
5	Improve generate_hybrid: complexity routing, caching	1 hr
6	Streamlit UI: mic, location sidebar, pipeline viz, compare mode	1.5 hrs
7	Testing, benchmarking, submission, pitch prep	30 min
	TOTAL	**~5 hours**

Build order tip: Complete Phases 2-3 first so you have a working pipeline you can test from the terminal. Then build the Streamlit UI around it (Phase 6). Phase 5 (benchmark optimization) can be done in parallel by a teammate.

12. Demo Script & Pitch Guide

3-minute demo script for judges

Opening (15 seconds)

"HandsFree is a voice-first agent for when you can't type — you're driving, cooking, cycling. You speak, and your device acts instantly. Everything runs on-device: voice transcription, GPS, and AI inference. Your data never leaves your machine."

Demo 1: Location Sharing (30 seconds)

Say: "Send my location to Mom". Point to the pipeline visualization showing real GPS coordinates grabbed from CoreLocation, the human-readable address, the Google Maps link embedded in the message. Highlight: "280 milliseconds, 100% on-device, zero internet."

Demo 2: Context Chain (30 seconds)

Say: "I've reached the office". Show 4 actions executing from a single sentence — message to Mom with GPS, weather check, lunch alarm, focus playlist. Highlight: "One sentence, four actions, 350ms. The routine is pre-configured and skips the LLM entirely."

Demo 3: Offline Mode (30 seconds)

Toggle Airplane Mode in the sidebar. Optionally turn off Wi-Fi on the Mac. Say: "Set an alarm for 6 AM and play morning playlist". Show it works. Highlight: "This works in a dead zone, on a mountain, in an airplane. 80% of commands don't need the internet."

Demo 4: Compare Mode (30 seconds)

Toggle Compare Mode. Say or type a command. Show three panels: FunctionGemma (85ms), Gemini (920ms), Hybrid (85ms). Highlight: "Same accuracy, 10x faster. Our router knows when to trust the local model."

Demo 5: Complex Fallback (15 seconds)

Say: "Text Sarah I'm running late, check weather in NYC, set a 10 minute timer". Show it routing to cloud. Highlight: "When the task is too complex for the 270M model, we route to Gemini Flash. The router detected 3 tools and 2 conjunctions."

Closing (15 seconds)

"HandsFree combines on-device transcription, real GPS, smart routing, and context chains into a single voice agent. 80% of commands run entirely on-device in under 300ms. Your location data never leaves your machine. Thank you."
