Project Cartesian: Master Context & Architecture Log

Last Updated: 2025-12-16 **Purpose:** This document is the single source of truth for architectural decisions, technical constraints, and "War Stories" (lessons learned). It is designed to instantly restore full developer context.

1. Core Architecture (The "Split Brain" System)

Q: What is the Language Stack?

Decision: Rust Core + Python Sidecar.

- **The Body (Rust):** `cartesian-core` handles the OS-level heavy lifting.
  - **GUI:** `iced` (Rust Native). Strictly no React/Webviews.
  - **Audio:** `pipewire` + `wireplumber` bindings.
  - **Process Control:** `lobotomy.rs` (Resource Governor).
- **The Mind (Python):** `cartesian-mind` handles the AI Intelligence.
  - **Inference:** PyTorch / `llama-cpp-python` (GGUF).
  - **Vision:** Moondream / OpenCV.
  - **Memory:** ChromaDB (managed via Python).
- **Communication:** IPC (Local Sockets or gRPC) connecting Body and Mind.

Q: How do we handle Updates (Stable vs. Rolling)?

Decision: Dual-Track Distro.

- **Stable (Default):** "Quarterly" releases. Users receive a tested snapshot of the `cartesian-core` binary and Python environment.
- **Bleeding Edge (Opt-in):** Users can enable a "Rolling" channel. This tracks the standard Arch repos and our latest builds, offering new AI features immediately but with higher breakage risk.

Q: What is the Hardware Resource Strategy?

Decision: Tiered Fallback (Prioritize Gaming).

- **Tier 1 (Intel):** Pin "Sidekick" AI models to **E-Cores** (Efficiency Cores).
- **Tier 2 (AMD):** Pin to **Compact Cores** (Zen 4c/5c) if detected.
- **Tier 3 (Fallback):** If no efficiency cores exist, use **Careful Bottlenecking** on P-Cores.
  - *Implementation:* Use `nice -n 19` and cgroup resource clamping to ensure the game *always* preempts the AI.

2. Feature Implementation Specifics

Q: How does Audio Routing work (Latency vs. Smarts)?

Decision: Hybrid (AI Classification -> Cached Lookup).

- **The Problem:** Running LLM inference on every process launch is too slow.
- **The Solution:**

1. **Pre-Population:** The OS ships with a `known_apps.json` registry of common IDs (Steam, Discord, Spotify).
2. **First Launch:** If an app is unknown, the AI analyzes the metadata *once* to categorize it (Game/Voice/Music).
3. **Cache:** This decision is written to the registry.
4. **Runtime:** Rust performs a near-instant lookup against the registry for all future launches.

Q: How do we develop the "Witness" (Vision) in a VM?

Decision: Software Fallback First.

- **Constraint:** VirtualBox/Docker cannot support NVENC (Hardware Encode) or DMA-BUF (Zero-Copy).
- **Protocol:** We must implement the **CPU Fallback** (XShm / PipeWire CPU Copy) first.
- **Future:** We will only implement the hardware-accelerated "Path A" and "Path B" once we validate on bare metal hardware.

Q: Which version of Iced are we using?

Decision: Pin Stable (0.13.x).

- **Strategy:** We stick to stable releases to prevent CI/CD breakage.
- **Watcher:** We are monitoring `iced_layershell`. If the "Holo-Badge" overlay requires unstable Wayland protocols, we may eventually need to track a specific `master` commit, but not yet.

Q: What is the Model Format Strategy?

Decision: Unified GGUF.

- **Standard:** All models (Text and Vision) will use `.gguf` format.
- **Vision:** We utilize the Multi-Modal Projector (MMProj) support in modern `llama.cpp` / `llama-cpp-python` to avoid needing a separate ONNX runtime for Moondream.

3. Build System (The "Factory")

Q: How do we build the ISO without a remote repository?

Decision: Local Vendoring with Dynamic HTTP Injection. We use a specific "Factory Pattern" to build the ISO entirely offline/locally:

1. **Compile:** `pkg/` compiles the Rust core into a `.pkg.tar.zst` artifact.
2. **Generate:** `build.sh` creates a temporary Arch repository in `iso/local_repo/` and runs `repo-add`.
3. **Serve:** The script spins up a temporary **Python HTTP Server** (`127.0.0.1:8050`) that hosts this repo *during the build process*.
4. **Inject:** We dynamically modify `pacman.conf` in the build profile to trust `http://127.0.0.1:8050/`. This bypasses strict file permission/symlink issues associated with `file://` protocols in `mkarchiso`.

4. "War Stories" & Critical Lessons

These protocols exist to prevent recurring failures discovered during the Windows Migration.

The "Invisible Enemy" (CRLF Line Endings)

- **The Issue:** Developing on Windows introduces Carriage Return (`\r`) characters into `build.sh`, `PKGBUILD`, and `profiledef.sh`. This causes Linux tools (bash, makepkg, mkarchiso) to crash silently or throw obscure "command not found" errors.
- **The Protocol:** The `iso/build.sh` script **MUST** recursively run `dos2unix` on `pkg/` and `iso/archiso_profile/` before doing *anything else*.
- **Rule:** Never remove the sanitization lines from `build.sh`.

The Docker Factory (Windows Workaround)

- **The Constraint:** `mkarchiso` cannot run on Windows or standard WSL2 (requires loop device mounting).
- **The Solution:** We use a **privileged** Docker container (`cartesian-builder`).
- **Dependencies:** This container *must* include:
  - `base-devel` (for compiling Rust)
  - `archiso` (for generating the ISO)
  - `dos2unix` (for sanitization)
  - `grub` + `efibootmgr` (Critical: required for building UEFI bootloaders).

The "Dirty Config" Loop

- **The Issue:** `mkarchiso` aggressively caches configuration in the `work/` directory. Changing `pacman.conf` or `profiledef.sh` often fails to propagate if the work directory exists.
- **The Protocol:** `build.sh` must `rm -rf work/` on *every single run* to ensure a clean build state.

Boot Mode Deprecation

- **Status:** The old ArchISO boot modes (specifically `uefi-x64.grub.esp`) are deprecated.
- **Action:** We must use the simplified `uefi.grub` syntax in `profiledef.sh` to ensure compatibility with modern `archiso` versions.