Project Aiden: The Definitive MCP-Integrated Build Manual

Section 1: The Aiden Philosophy & Architecture

1.1. Core Principles: Privacy, Performance, and Personalization

Project Aiden is conceived from a foundational belief that a truly intelligent home environment should be an extension of its inhabitants—private, responsive, and deeply personalized. This manual provides the blueprint for constructing such a system, moving beyond the limitations of commercial, cloud-dependent smart homes. The architecture is built upon three core pillars that guide every design choice and component selection.

- **Privacy**: In an era of pervasive data collection, Aiden is engineered to operate on a local-first principle. Core AI processing, automation logic, and personal data remain within the confines of the local network. By minimizing reliance on external cloud services for critical functions, the system ensures that sensitive information, such as camera feeds and conversation history, is not transmitted to or stored by third parties. This approach returns control of personal data to the owner, creating a secure and trustworthy smart home environment.
- **Performance**: A seamless smart home experience depends on low-latency interactions. Commercial systems often introduce delays by routing commands and data through remote servers. Aiden's architecture circumvents this by leveraging dedicated, high-performance local hardware for computationally intensive tasks. The AI Workhorse is specified for rapid AI model inference, video analysis, and complex logic execution, ensuring that the system responds to commands and events in near real-time. This focus on local performance delivers a fluid and natural user experience that cloud-based alternatives struggle to match.
- **Personalization**: Aiden is not a one-size-fits-all product; it is a bespoke platform designed for infinite customization. The system is built to learn and adapt to the unique rhythms and preferences of its household. From custom voice models and personalized morning routines to a knowledge base that grows with user-provided information, every aspect of Aiden can be tailored. This manual provides the foundation, but the ultimate goal is to empower the builder to create an assistant that is uniquely their own, capable of handling specific tasks and understanding personal context in a way no generic commercial service can.

1.2. System Blueprint: The Service-Oriented Architecture

To achieve these goals, the system's architecture deliberately segregates responsibilities across distinct hardware and software zones, a professional design pattern that enhances resilience, scalability, and maintainability. The core components are the **AI Workhorse** (a powerful mini-PC for heavy computation) and the **Control & Integration Hub** (a dedicated Raspberry Pi for real-time device control).

The central architectural evolution detailed in this manual is the transition from a monolithic

application to a standardized, service-oriented architecture. The original design, while functional, contained a central "Memory Proxy" component that was responsible for everything: orchestrating LLM calls, polling Home Assistant for device states, and querying a vector database for information. This tight coupling of logic and I/O created "brittle integrations". A change to Home Assistant's API or a decision to swap the database technology would require a complex and risky rewrite of the system's core logic.
This manual rectifies that architectural liability. We will deconstruct the monolith into a set of independent, containerized microservices. Each service will be responsible for a single task (e.g., communicating with Home Assistant, querying the database) and will expose its capabilities to the rest of the system using a universal, standardized protocol.

1.3. The Role of the Model Context Protocol (MCP)

The Model Context Protocol (MCP) is the key enabling technology for this new architecture. It is a design philosophy aimed at creating a universal, plug-and-play format for AI tools and data sources. Drawing inspiration from the Language Server Protocol (LSP) which decoupled programming languages from code editors, MCP decouples tool providers from AI application consumers. It standardizes how an AI agent discovers, authenticates with, and interacts with external capabilities.
MCP is built upon three core primitives that we will implement for Project Aiden :

- **Tools**: These are functions that an AI can execute. We will transform the logic for communicating with Home Assistant and ChromaDB into discrete MCP tools, such as get_entity_state or query_documents.
- **Resources**: This primitive offers a powerful abstraction for data sources. The collection of documents in our ChromaDB database will be exposed as an MCP Resource, allowing for standardized search operations.
- **Prompts**: Reusable prompt templates can be served via MCP to ensure personality and operational consistency across different agents in the system.

A critical architectural decision in this build is the choice of transport protocol for our MCP servers. While many simple examples use stdio for local communication, this is fundamentally incompatible with Aiden's distributed nature. A server running via stdio can only be accessed by clients on the same machine, which would prevent our Home Assistant hub or other network devices from using the tools hosted on the AI Workhorse. Therefore, to maintain architectural integrity and ensure all tools are available as independent microservices across the entire local network, **all custom MCP servers for Project Aiden must be built using the HTTP/SSE (Server-Sent Events) transport**.

1.4. Architectural Diagram: Visualizing the MCP Data Flow

The following diagram illustrates the new, service-oriented architecture. It shows how a user's request flows through the system, with the Memory Proxy now acting as a smart client that orchestrates calls to a suite of independent, network-accessible MCP servers.

```
graph TD
    subgraph User Interaction
        USER[User] -- Voice Command --> SAT
    end

    subgraph Control & Integration Hub (Raspberry Pi 5)
```

```
    HA[Home Assistant]
end

subgraph AI Workhorse (GMKtec K8 Docker)
    subgraph aiden_net
        MP[Memory Proxy <br/> (MCP Client)]
        HA_MCP
        CH_MCP
        VOICE_MCP
        OLLAMA[Ollama LLM]
        XTTS
    end
end

%% Data Flows
SAT -- Audio Stream --> HA
HA -- Voice Pipeline --> VOICE_MCP
VOICE_MCP -- Transcribed Text --> HA
HA -- Conversation Agent --> MP
MP -- Tool Call (HTTP/SSE) --> HA_MCP
MP -- Tool Call (HTTP/SSE) --> CH_MCP
MP -- LLM Query --> OLLAMA
HA_MCP -- API Call --> HA
MP -- Response --> HA
HA -- Text to Synthesize --> XTTS
XTTS -- Spoken Audio --> HA
HA -- Audio Stream --> SAT
SAT -- Spoken Response --> USER

%% Styling
classDef workhorse fill:#cce5ff, stroke:#333, stroke-width:2px;
classDef hub fill:#d4edda, stroke:#333, stroke-width:2px;
classDef satellite fill:#fff3cd, stroke:#333, stroke-width:2px;
class MP,HA_MCP,CH_MCP,VOICE_MCP,OLLAMA,XTTS workhorse;
class HA hub;
class SAT satellite;
```

Section 2: System Prerequisites

Before beginning the build, it is essential to gather all the necessary hardware, software, and accounts. This ensures a smooth and uninterrupted setup process.

2.1. Hardware Manifest and Component Roles

The following table provides the single, definitive list of all required hardware components. Each component has been selected for a specific, optimized role within the system's architecture to

ensure maximum performance and reliability.

Component	Model/Type	Primary Function	Key Specification/Role	Quantity
AI Workhorse	GMKtec K8 Plus (or similar)	Central AI & Service Host	Hosts Docker containers for all core services; requires at least 32GB RAM for smooth LLM and NVR operation.	1
AI Accelerator	Google Coral TPU (USB)	Vision Processing Accelerator	Dedicated to Frigate for high-speed, low-power object detection, freeing up CPU resources for other tasks.	1
HA Hub	Raspberry Pi 5	Automation & Device Controller	Runs the lightweight and highly reliable Home Assistant Operating System (HAOS) for core automations and device management.	1
HA Hub Peripherals	ZBT-1 Zigbee/Thread Dongle	Zigbee/Thread Radio	Provides direct communication with low-power smart devices like sensors and switches.	1
Lounge Satellite	Raspberry Pi 5 (8GB)	High-Res Camera & Display	Provides the primary, high-resolution video feed to Frigate for facial recognition and event recording.	1
Bedroom Satellite	Raspberry Pi Zero 2 W	Camera & BT Proxy	Provides a lightweight snapshot camera feed and extends Home Assistant's Bluetooth range for devices like smart blinds.	2

Component	Model/Type	Primary Function	Key Specification/Role	Quantity
Satellite Peripherals	Pi Camera, USB Audio Dongle	I/O for Satellites	A camera module and a combined microphone/speaker USB dongle are required for each satellite device.	3 sets
Networking	Gigabit Ethernet Router	Network Backbone	Provides wired connectivity for the AI Workhorse and HA Hub, ensuring maximum network stability and speed.	1

2.2. Software and Accounts Checklist

- **Software**:
  - **Raspberry Pi Imager**: The official tool for flashing operating systems to Raspberry Pi storage. It can be downloaded from the official Raspberry Pi website.
  - **Ubuntu Server 24.04 LTS**: The operating system for the AI Workhorse. The "Server Image" should be downloaded from the official Ubuntu website.
  - **An SSH Client**: A tool for connecting to servers remotely.
    - **Windows**: PowerShell (built-in) or PuTTY.
    - **macOS/Linux**: The Terminal application is built-in.
- **Accounts**:
  - **OpenRouter Account**: Required for accessing powerful, cloud-based Large Language Models (LLMs). An account can be created at OpenRouter.ai. An API key must be generated, and a small amount of credit (e.g., $5) should be added to the account to enable API access. This key is sensitive and must be stored securely.

Section 3: Core Infrastructure: The AI Workhorse

The AI Workhorse (GMKtec K8) is the computational heart of Project Aiden. This section covers the setup of its operating system and the containerization platform that will host all the intelligence services.

3.1. Installing and Securing the Ubuntu Server OS

A minimal server operating system provides a stable and efficient foundation.

1. **Flash the OS**: Use Raspberry Pi Imager or a similar tool like BalenaEtcher to flash the downloaded Ubuntu Server 24.04 LTS image onto the GMKtec K8's NVMe drive.
2. **Initial Boot**: Install the NVMe drive into the mini-PC, connect a keyboard, monitor, and an Ethernet cable, and power it on.
3. **Ubuntu Installation**: Follow the on-screen prompts to install Ubuntu. The default options are suitable for most steps.

    - **Hostname**: During setup, choose a simple and memorable hostname, such as k8-aiden.
    - **SSH Server**: When prompted, ensure the option to install the **OpenSSH server** is selected. This is critical for remote management.
    - **Server Snaps**: Do not install any of the additional server snaps (like the Docker snap) from the installer. The software will be installed manually for better control and compatibility.
4. **First Login & IP Address**: After the installation completes and the system reboots, log in with the username and password created during setup. Find the server's IP address by running the command ip a. Look for the inet address under the primary Ethernet interface (e.g., enp3s0). It will look similar to 192.168.1.100. This IP address is essential and should be recorded.
5. **Remote Access**: From this point forward, all interaction with the server can be done from a primary computer via SSH. Open a terminal or PowerShell and connect using the command: ssh your_username@YOUR_K8_IP_ADDRESS.

3.2. Deploying the Containerization Platform: Docker and Portainer

Docker is a platform that allows applications to be run in isolated environments called containers. This simplifies installation, updates, and management. Portainer provides a user-friendly web interface for managing the Docker environment.

1. **Update System Packages**: In the SSH session, ensure the server's software is up to date:
   ```
   sudo apt update && sudo apt upgrade -y
   ```

2. **Install Docker**:
   ```
   sudo apt install docker.io -y
   sudo systemctl enable --now docker
   ```

3. **Add User to Docker Group**: To manage Docker without needing sudo for every command, add the current user to the docker group:
   ```
   sudo usermod -aG docker $USER
   ```
   Action Required: This change requires logging out and logging back in to take effect. Type exit in the SSH session and then reconnect.
4. **Install Docker Compose**: This tool is used to define and run multi-container Docker applications.
   ```
   sudo apt install docker-compose -y
   ```

5. **Install Portainer**:
   ```
   docker volume create portainer_data
   docker run -d -p 8000:8000 -p 9443:9443 --name portainer \
     --restart=always \
     -v /var/run/docker.sock:/var/run/docker.sock \
     -v portainer_data:/data \
     portainer/portainer-ce:latest
   ```

6. **Access Portainer**: Open a web browser and navigate to https://YOUR_K8_IP_ADDRESS:9443. A browser security warning will appear; it is safe

to click "Advanced" and "Proceed". Create an admin user and password. On the next screen, choose to manage the "Local" Docker environment and click "Connect".

Section 4: The AI Services Stack: An MCP-Native Ecosystem

This section details the deployment of the core software services that constitute Aiden's intelligence. We will build a complete, containerized ecosystem where each component is an independent service communicating via the Model Context Protocol.

4.1. Project Directory Structure

A well-organized directory structure is essential for managing configuration files. In the SSH session on the K8, create the necessary folders. This structure separates the main configuration from the code for our custom MCP servers.

```
# Create the main project directory
mkdir -p /opt/aiden/config

# Create subdirectories for each service's persistent data
mkdir -p /opt/aiden/config/frigate
mkdir -p /opt/aiden/config/double-take
mkdir -p /opt/aiden/config/xtts
mkdir -p /opt/aiden/config/ollama
mkdir -p /opt/aiden/config/chroma
mkdir -p /opt/aiden/config/whisper

# Create a directory for the refactored Memory Proxy application
mkdir -p /opt/aiden/proxy

# Create a parent directory for our custom MCP servers
mkdir -p /opt/aiden/mcp_servers

# Create directories for each individual MCP server
mkdir -p /opt/aiden/mcp_servers/ha_server
mkdir -p /opt/aiden/mcp_servers/chroma_server
mkdir -p /opt/aiden/mcp_servers/whisper_server
```

4.2. The Definitive docker-compose.yml with MCP Services

This is the master blueprint that tells Docker how to run all the services, how they are networked, and where their data is stored. This configuration has been meticulously crafted for performance and interoperability, integrating the Coral TPU, dedicated voice processing, and our new custom MCP servers.
Create the file using the nano text editor:

```
nano /opt/aiden/docker-compose.yml
```

Copy the entire block of code below and paste it into the editor. Note that services communicate using their container names as hostnames (e.g., memory-proxy can reach chroma-mcp-server at http://chroma-mcp-server:9002).

```
# Full Docker Compose for Project Aiden - MCP Definitive Edition
# Reference: Section 4, Part 4.2
version: '3.8'

services:
  # Frigate NVR with Coral TPU acceleration
  frigate:
    container_name: frigate
    privileged: true
    restart: unless-stopped
    image: ghcr.io/blakeblackshear/frigate:0.13.2
    shm_size: '1gb'
    devices:
      - /dev/bus/usb:/dev/bus/usb # Passthrough for USB Coral TPU
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /opt/aiden/config/frigate:/config
      - type: tmpfs
        target: /tmp/cache
        tmpfs:
          size: 1000000000
    ports:
      - "5000:5000"
      - "8554:8554"
      - "8555:8555/tcp"
      - "8555:8555/udp"

  # Whisper for high-performance Speech-to-Text via Wyoming protocol
  whisper:
    container_name: whisper
    image: rhasspy/wyoming-whisper:latest
    restart: unless-stopped
    command: --model base-int8 --language en
    volumes:
      - /opt/aiden/config/whisper:/data
    ports:
      - "10300:10300"

  # Double Take for face recognition
  double-take:
    container_name: double-take
    restart: unless-stopped
    image: jakowenko/double-take
    volumes:
      - /opt/aiden/config/double-take:/config
```

```
    ports:
      - "3000:3000"
    depends_on:
      - frigate

  # XTTS for high-quality Text-to-Speech
  xtts-server:
    container_name: xtts-server
    restart: unless-stopped
    image: ghcr.io/coqui-ai/xtts-streaming-server:latest
    ports:
      - "8020:8020"
    volumes:
      - /opt/aiden/config/xtts:/app/speakers
    environment:
      - COQUI_TOS_AGREED=1

  # Ollama for local LLM fallback
  ollama:
    container_name: ollama
    restart: unless-stopped
    image: ollama/ollama
    volumes:
      - /opt/aiden/config/ollama:/root/.ollama
    ports:
      - "11434:11434"

  # ChromaDB vector database
  chroma:
    container_name: chroma
    restart: unless-stopped
    image: chromadb/chroma
    volumes:
      - /opt/aiden/config/chroma:/chroma/.chroma/index
    ports:
      - "8001:8000" # Mapped to 8001 to avoid conflict with Portainer
on 8000

  # --- Custom MCP Servers ---

  # Home Assistant MCP Server
  ha-mcp-server:
    container_name: ha-mcp-server
    restart: unless-stopped
    build:
      context: /opt/aiden/mcp_servers/ha_server
    ports:
      - "9001:9001"
```

```
    env_file:
      - /opt/aiden/.env
    depends_on:
      - memory-proxy # Wait for proxy to ensure env file is available
if needed

  # ChromaDB MCP Server
  chroma-mcp-server:
    container_name: chroma-mcp-server
    restart: unless-stopped
    build:
      context: /opt/aiden/mcp_servers/chroma_server
    ports:
      - "9002:9002"
    depends_on:
      - chroma

  # Wyoming-Whisper MCP Server
  whisper-mcp-server:
    container_name: whisper-mcp-server
    restart: unless-stopped
    build:
      context: /opt/aiden/mcp_servers/whisper_server
    ports:
      - "9003:9003"
    depends_on:
      - whisper

  # The Memory Proxy - Aiden's refactored brain (MCP Client)
  memory-proxy:
    container_name: memory-proxy
    restart: unless-stopped
    build:
      context: /opt/aiden/proxy
    ports:
      - "8080:80"
    env_file:
      - /opt/aiden/.env
    depends_on:
      - ollama
      - ha-mcp-server
      - chroma-mcp-server
      - whisper-mcp-server
```

Press Ctrl+X to exit, then Y to save, and Enter to confirm the filename.

4.3. Environment Configuration for Secrets Management (.env)

This file holds secret keys and configuration variables, keeping them separate from the main docker-compose.yml file.
Create the file:
`nano /opt/aiden/.env`

Paste the following, replacing the placeholder values with your actual details. The HA_TOKEN will be generated in a later section, so leave it as a placeholder for now.

```
# Environment variables for Project Aiden
# Reference: Section 4, Part 4.3

# Get this from https://openrouter.ai/keys
OPENROUTER_API_KEY='sk-or-v1-...'

# Your Home Assistant URL and a Long-Lived Access Token (we will
create this later)
HA_URL='http://YOUR_HA_PI_IP_ADDRESS:8123'
HA_TOKEN='placeholder'

# LLM Models
# Main model for complex tasks (from OpenRouter)
MAIN_LLM_MODEL='anthropic/claude-3.5-sonnet'
# Fallback model for when internet is down (running locally in Ollama)
FALLBACK_LLM_MODEL='phi3:mini'
```

Save and exit (Ctrl+X, Y, Enter).

4.4. Building the Home Assistant MCP Server

This server encapsulates all communication with Home Assistant, exposing its functionality as a standardized set of MCP tools. This replaces the brittle, hard-coded httpx calls from the original design.

1. **Create the Python dependencies file**:
   `nano /opt/aiden/mcp_servers/ha_server/requirements.txt`
   Paste the following:
   ```
   fastmcp
   httpx
   uvicorn
   ```
   Save and exit.
2. **Create the Dockerfile**:
   `nano /opt/aiden/mcp_servers/ha_server/Dockerfile`
   Paste the following:
   ```
   # Dockerfile for Home Assistant MCP Server
   FROM python:3.11-slim
   WORKDIR /app
   COPY requirements.txt.
   RUN pip install --no-cache-dir -r requirements.txt
   COPY ha_mcp_server.py.
   CMD ["uvicorn", "ha_mcp_server:app", "--host", "0.0.0.0",
   ```

"--port", "9001"]
Save and exit.

3. **Create the MCP Server application file**:
   nano /opt/aiden/mcp_servers/ha_server/ha_mcp_server.py
   Paste the complete Python code below. This server uses fastmcp to create an HTTP/SSE server and defines tools for getting entity states and calling services in Home Assistant.

```
# ha_mcp_server.py
# Reference: Section 4, Part 4.4
import os
import httpx
from fastapi import FastAPI, HTTPException
from mcp.server.fastmcp import FastMCP
from pydantic import BaseModel, Field
from typing import Dict, Any

# --- Configuration ---
HA_URL = os.getenv("HA_URL")
HA_TOKEN = os.getenv("HA_TOKEN")

if not HA_URL or not HA_TOKEN:
    raise ValueError("HA_URL and HA_TOKEN environment variables
must be set.")

HEADERS = {
    "Authorization": f"Bearer {HA_TOKEN}",
    "Content-Type": "application/json",
}

# --- MCP Server Setup ---
mcp = FastMCP(
    "Home Assistant MCP Server",
    instructions="Provides tools to interact with a Home Assistant
instance."
)
app = FastAPI()

# --- Tool Definitions ---
@mcp.tool()
async def get_entity_state(entity_id: str) -> Dict[str, Any]:
    """
    Retrieves the full state object for a specific entity from
Home Assistant.
    :param entity_id: The ID of the entity to query (e.g.,
'light.lounge_lights').
    """
    async with httpx.AsyncClient() as client:
        try:
            response = await
```

```
client.get(f"{HA_URL}/api/states/{entity_id}", headers=HEADERS)
            response.raise_for_status()
            return response.json()
        except httpx.RequestError as e:
            return {"error": f"HTTP request failed: {e}"}
        except httpx.HTTPStatusError as e:
            return {"error": f"HTTP status error:
{e.response.status_code}", "response": e.response.text}

class CallServicePayload(BaseModel):
    domain: str = Field(..., description="The domain of the
service (e.g., 'light').")
    service: str = Field(..., description="The name of the service
to call (e.g., 'turn_on').")
    entity_id: str = Field(..., description="The entity_id to
target.")
    service_data: Dict[str, Any] = Field({}, description="Optional
data for the service call (e.g., {'brightness': 255}).")

@mcp.tool()
async def call_service(payload: CallServicePayload) -> Dict[str,
Any]:
    """
    Calls a service in Home Assistant.
    :param payload: A Pydantic model containing the domain,
service, entity_id, and optional service_data.
    """
    service_url =
f"{HA_URL}/api/services/{payload.domain}/{payload.service}"
    json_payload = {
        "entity_id": payload.entity_id,
        **payload.service_data
    }
    async with httpx.AsyncClient() as client:
        try:
            response = await client.post(service_url,
headers=HEADERS, json=json_payload)
            response.raise_for_status()
            return {"status": "success", "response":
response.json()}
        except httpx.RequestError as e:
            return {"error": f"HTTP request failed: {e}"}
        except httpx.HTTPStatusError as e:
            return {"error": f"HTTP status error:
{e.response.status_code}", "response": e.response.text}

# --- Mount MCP Server to FastAPI ---
# The fastmcp library provides a run() method that handles this,
```

```
# but for explicit control with uvicorn, we can mount it.
# For simplicity in this manual, we will let the Docker CMD handle
running it.
# The following line is illustrative of how it works under the
hood.
# app.mount("/mcp", mcp.as_asgi())

# We need to run the MCP server itself. fastmcp can generate an
ASGI app.
app = mcp.run(transport="streamable-http", app=app, port=9001,
host="0.0.0.0", serve=False)

@app.get("/")
def read_root():
    return {"Hello": "Home Assistant MCP Server"}
```

Save and exit.

4.5. Building the ChromaDB MCP Server

This server modularizes the Retrieval-Augmented Generation (RAG) functionality, exposing document search as a standardized tool. This implements Phase 2 of the MCP adoption plan.

1. **Create the Python dependencies file**:
   `nano /opt/aiden/mcp_servers/chroma_server/requirements.txt`
   Paste the following:
   ```
   fastmcp
   chromadb-client
   sentence-transformers
   langchain-community
   uvicorn
   ```
   Save and exit.
2. **Create the Dockerfile**:
   `nano /opt/aiden/mcp_servers/chroma_server/Dockerfile`
   Paste the following:
   ```
   # Dockerfile for ChromaDB MCP Server
   FROM python:3.11-slim
   WORKDIR /app
   COPY requirements.txt.
   RUN pip install --no-cache-dir -r requirements.txt
   COPY chroma_mcp_server.py.
   CMD ["uvicorn", "chroma_mcp_server:app", "--host", "0.0.0.0",
   "--port", "9002"]
   ```
   Save and exit.
3. **Create the MCP Server application file**:
   `nano /opt/aiden/mcp_servers/chroma_server/chroma_mcp_server.py`
   Paste the complete Python code below. This server connects to the chroma container and exposes a similarity_search tool.
   ```
   # chroma_mcp_server.py
   # Reference: Section 4, Part 4.5
   ```

```
import chromadb
from fastapi import FastAPI
from mcp.server.fastmcp import FastMCP
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import
SentenceTransformerEmbeddings
from typing import List, Dict, Any

# --- ChromaDB Client Setup ---
# The server connects to the 'chroma' service on its internal
Docker port 8000
chroma_client = chromadb.HttpClient(host="chroma", port=8000)
embedding_function =
SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")
vector_store = Chroma(
    client=chroma_client,
    collection_name="aiden_documents",
    embedding_function=embedding_function,
)

# --- MCP Server Setup ---
mcp = FastMCP(
    "ChromaDB RAG Server",
    instructions="Provides tools for document retrieval from a
vector store."
)
app = FastAPI()

# --- Tool Definitions ---
@mcp.tool()
def query_documents(query: str, k: int = 2) -> List]:
    """
    Performs a similarity search in the document vector store.
    :param query: The text to search for.
    :param k: The number of relevant documents to return.
    """
    try:
        docs = vector_store.similarity_search(query, k=k)
        return [{"page_content": doc.page_content, "metadata":
doc.metadata} for doc in docs]
    except Exception as e:
        return [{"error": f"Failed to perform similarity search:
{e}"}]

# --- Mount MCP Server to FastAPI ---
app = mcp.run(transport="streamable-http", app=app, port=9002,
host="0.0.0.0", serve=False)
```

```
@app.get("/")
def read_root():
    return {"Hello": "ChromaDB MCP Server"}
```
Save and exit.

4.6. Building the Wyoming-Whisper MCP Server

This server provides the critical abstraction layer for voice processing. It acts as a client to the dedicated whisper container (which speaks the Wyoming protocol) and exposes a simple transcribe tool over MCP. This makes the rest of the system independent of the specific speech-to-text technology being used.

1. **Create the Python dependencies file**:
   `nano /opt/aiden/mcp_servers/whisper_server/requirements.txt`
   Paste the following:
   ```
   fastmcp
   wyoming
   uvicorn
   ```
   Save and exit.
2. **Create the Dockerfile**:
   `nano /opt/aiden/mcp_servers/whisper_server/Dockerfile`
   Paste the following:
   ```
   # Dockerfile for Whisper MCP Server
   FROM python:3.11-slim
   WORKDIR /app
   COPY requirements.txt.
   RUN pip install --no-cache-dir -r requirements.txt
   COPY whisper_mcp_server.py.
   CMD ["uvicorn", "whisper_mcp_server:app", "--host", "0.0.0.0",
   "--port", "9003"]
   ```
   Save and exit.
3. **Create the MCP Server application file**:
   `nano /opt/aiden/mcp_servers/whisper_server/whisper_mcp_server.py`
   Paste the complete Python code below. This server contains an asynchronous client that communicates with the wyoming-whisper service.
   ```
   # whisper_mcp_server.py
   # Reference: Section 4, Part 4.6
   import asyncio
   import logging
   from fastapi import FastAPI, UploadFile, File
   from mcp.server.fastmcp import FastMCP
   from wyoming.asr import Transcribe, Transcript
   from wyoming.audio import AudioChunk, AudioStart, AudioStop
   from wyoming.client import AsyncClient
   from wyoming.event import async_read_event, async_write_event

   # --- Configuration ---
   WHISPER_HOST = "whisper"
   WHISPER_PORT = 10300
   ```

```
RATE = 16000
WIDTH = 2
CHANNELS = 1
CHUNK_SIZE = 1024

logging.basicConfig(level=logging.INFO)
_LOGGER = logging.getLogger(__name__)

# --- MCP Server Setup ---
mcp = FastMCP(
    "Whisper STT Server",
    instructions="Provides a tool to transcribe audio data using a
Wyoming-Whisper service."
)
app = FastAPI()

# --- Tool Definitions ---
@mcp.tool()
async def transcribe_audio(audio_bytes: bytes) -> str:
    """
    Transcribes a chunk of raw PCM audio bytes to text.
    :param audio_bytes: The raw audio data (16-bit, 16kHz, mono
PCM).
    """
    try:
        async with AsyncClient(WHISPER_HOST, WHISPER_PORT) as
client:
            # Handshake
            await
async_write_event(Transcribe(language="en").event(),
client.writer)
            _LOGGER.info("Sent transcribe event to Whisper")

            # Stream audio
            await async_write_event(AudioStart(rate=RATE,
width=WIDTH, channels=CHANNELS).event(), client.writer)
            await async_write_event(AudioChunk(audio=audio_bytes,
rate=RATE, width=WIDTH, channels=CHANNELS).event(), client.writer)
            await async_write_event(AudioStop().event(),
client.writer)
            _LOGGER.info("Finished streaming audio to Whisper")

            # Wait for transcript
            while True:
                event = await async_read_event(client.reader)
                if event is None:
                    _LOGGER.error("Connection closed unexpectedly
by Whisper server")
```

```
                    return "Error: Connection closed by Whisper
server."

                if Transcript.is_type(event.type):
                    transcript_event =
Transcript.from_event(event)
                    _LOGGER.info(f"Received transcript:
{transcript_event.text}")
                    return transcript_event.text
    except Exception as e:
        _LOGGER.error(f"Error during transcription: {e}")
        return f"Error: {e}"

# --- API Endpoint for File Uploads (Helper for testing) ---
@app.post("/transcribe_file")
async def transcribe_file(file: UploadFile = File(...)):
    """A simple endpoint to test the transcription tool with a
file."""
    audio_bytes = await file.read()
    return {"transcription": await transcribe_audio(audio_bytes)}

# --- Mount MCP Server to FastAPI ---
app = mcp.run(transport="streamable-http", app=app, port=9003,
host="0.0.0.0", serve=False)

@app.get("/")
def read_root():
    return {"Hello": "Whisper MCP Server"}
```

Save and exit.

4.7. The Refactored Memory Proxy: An MCP-Aware Brain

This is the new core of Aiden's intelligence. Its responsibilities have been simplified: it no longer contains any direct I/O logic. Instead, it acts as a smart client, or orchestrator, that uses the standardized tools provided by our new MCP ecosystem.

4.7.1. Dockerfile and requirements.txt

First, update the dependencies. The proxy no longer needs direct clients for HA or ChromaDB but now requires the langchain-mcp-adapters library to communicate with our MCP servers.

1. **Update the Python dependencies file**:
   `nano /opt/aiden/proxy/requirements.txt`
   Replace the entire content with the following:
   ```
   # requirements.txt for Memory Proxy (MCP Client Edition)
   # Reference: Section 4, Part 4.7.1
   fastapi
   uvicorn
   pydantic
   ```

```
langchain
langchain_community
langchain-mcp-adapters
```
Save and exit.

2. **Create the Dockerfile**: This file remains unchanged from the original manual.
```
nano /opt/aiden/proxy/Dockerfile
```
Paste this inside:
```
# Dockerfile for Memory Proxy
# Reference: Section 4, Part 4.7.1
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt
COPY main.py.
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```
Save and exit.

4.7.2. The Complete Refactored main.py

This is the centerpiece of the architectural refactoring. The new main.py is simpler, more robust, and more extensible. It uses the MultiServerMCPClient to connect to our three custom servers and dynamically loads their tools. The core logic now focuses purely on constructing the prompt and orchestrating the LLM, which in turn decides which standardized tool to use.
Create the main application file:
```
nano /opt/aiden/proxy/main.py
```

Carefully copy and paste the entire rewritten Python code block below.
```
# Main application for Aiden's Memory Proxy (MCP Client Edition)
# Reference: Section 4, Part 4.7.2
import os
import httpx
import asyncio
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Dict, Any, Optional

from langchain_mcp_adapters.client import MultiServerMCPClient
from langchain.agents import AgentExecutor, create_tool_calling_agent
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_community.chat_models.openrouter import ChatOpenRouter

# --- Configuration ---
OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")
MAIN_LLM_MODEL = os.getenv("MAIN_LLM_MODEL",
"anthropic/claude-3.5-sonnet")
FALLBACK_LLM_MODEL = os.getenv("FALLBACK_LLM_MODEL", "phi3:mini")
```

```
# --- Pydantic Models ---
class LLMRequest(BaseModel):
    prompt: str
    device_id: Optional[str] = None

class LLMResponse(BaseModel):
    response_text: str
    debug_info: Dict[str, Any]

# --- Global Variables ---
app = FastAPI(title="Project Aiden Memory Proxy (MCP Edition)")
mcp_client: MultiServerMCPClient = None
mcp_tools: List[Any] =
agent_executor: AgentExecutor = None

# --- Application Lifecycle (Startup) ---
@app.on_event("startup")
async def startup_event():
    """
    On startup, connect to all MCP servers and initialize the
LangChain agent.
    """
    global mcp_client, mcp_tools, agent_executor

    # 1. Initialize the MCP client with connections to our custom
servers
    mcp_client = MultiServerMCPClient({
        "home_assistant": {
            "transport": "streamable_http",
            "url": "http://ha-mcp-server:9001/mcp"
        },
        "chroma_db": {
            "transport": "streamable_http",
            "url": "http://chroma-mcp-server:9002/mcp"
        },
        "whisper": {
            "transport": "streamable_http",
            "url": "http://whisper-mcp-server:9003/mcp"
        }
    })

    # 2. Load all tools from all connected servers
    try:
        mcp_tools = await mcp_client.get_tools()
        print(f"Successfully loaded {len(mcp_tools)} tools from MCP
servers.")
        for tool in mcp_tools:
            print(f" - Loaded tool: {tool.name}")
```

```
    except Exception as e:
        print(f"FATAL: Failed to load MCP tools on startup: {e}")
        # In a production system, you might want to handle this more
gracefully
        mcp_tools =

    # 3. Define the LLMs
    main_llm = ChatOpenRouter(model_name=MAIN_LLM_MODEL,
openrouter_api_key=OPENROUTER_API_KEY)
    fallback_llm = ChatOllama(model=FALLBACK_LLM_MODEL,
base_url="http://ollama:11434")

    # 4. Create the LangChain Agent
    # For now, we will primarily use the main LLM. Fallback logic is
simplified.
    llm = main_llm

    prompt_template = ChatPromptTemplate.from_messages()

    agent = create_tool_calling_agent(llm, mcp_tools, prompt_template)
    agent_executor = AgentExecutor(agent=agent, tools=mcp_tools,
verbose=True)

# --- API Endpoints ---
@app.get("/")
def read_root():
    return {"Hello": "This is Project Aiden's Memory Proxy (MCP
Edition)"}

@app.post("/ask", response_model=LLMResponse)
async def ask_aiden(request: LLMRequest):
    """
    Main endpoint for asking Aiden a question.
    This now invokes the LangChain agent which orchestrates MCP tool
calls.
    """
    if not agent_executor:
        raise HTTPException(status_code=503, detail="Agent not
initialized. Check logs for MCP connection errors.")

    debug_info = {}
    try:
        # Invoke the agent with the user's prompt
        response = await agent_executor.ainvoke({"input":
request.prompt})
        response_text = response.get("output", "I'm not sure how to
respond to that.")
        debug_info['agent_response'] = response
```

```
    except Exception as e:
        # Simplified fallback logic for the manual
        # A more robust implementation could be added here
        print(f"Agent execution failed: {e}. A production system could
implement fallback here.")
        response_text = "I encountered an error while processing your
request."
        debug_info['error'] = str(e)

    return LLMResponse(response_text=response_text,
debug_info=debug_info)

# Note: The /upload_document endpoint is now deprecated in the proxy.
# This functionality should be handled by a dedicated RAG management
service or a direct call
# to a new tool on the ChromaDB MCP server, such as 'add_document'.
# For simplicity, it is removed from this version of the manual.
```

Save and exit.

4.8. Launching and Verifying the Full AI Stack

With all configuration files in place, the complete AI stack can now be launched.

1. **Navigate to the project directory**:
   `cd /opt/aiden`

2. **Pull the local LLM model**: This command downloads the phi3:mini model into Ollama's persistent storage. This may take some time.
   `docker compose run --rm ollama ollama pull phi3:mini`

3. **Launch all services**: The -d flag runs the containers in the background (detached mode).
   `docker compose up -d`

4. **Verification**:
   - Check that all containers are running with the command: docker ps. You should see frigate, whisper, double-take, xtts-server, ollama, chroma, ha-mcp-server, chroma-mcp-server, whisper-mcp-server, and memory-proxy all with a status of "Up".
   - Check the logs of the Memory Proxy to confirm it successfully connected to the MCP servers.
     `docker logs memory-proxy`
     You should see output indicating that it has loaded tools, for example: Successfully loaded 3 tools from MCP servers. followed by the names of the tools like home_assistant_get_entity_state. If you see errors, it indicates a networking or configuration issue between the containers.

Congratulations, Aiden's brain is now online and operating on a modern, service-oriented architecture. The rest of the setup involves connecting Home Assistant and the satellites to this

powerful new core.

Section 5: The Control Hub: Home Assistant

This Raspberry Pi will be the heart of your home's automation, running the lightweight and powerful Home Assistant Operating System (HAOS) and communicating directly with your smart devices.

5.1. Flashing and Initial Onboarding of HAOS

1. **Hardware Connection**: Plug your ZBT-1 dongle into one of the USB 3.0 (blue) ports on your Raspberry Pi 5.
2. **Flash HAOS**: Open the Raspberry Pi Imager on your main computer.
    - Click **CHOOSE DEVICE** and select "Raspberry Pi 5".
    - Click **CHOOSE OS**. Select "Other specific-purpose OS" -> "Home assistants and home automation" -> "Home Assistant OS". Select the latest stable version.
    - Click **CHOOSE STORAGE** and select your SD card or NVMe drive.
    - Click **WRITE**. This will erase the storage and install HAOS.
3. **First Boot**: Once complete, insert the storage into your Pi 5, connect the Ethernet cable, and plug in the power. The first boot can take up to 20 minutes as it needs to download and install updates.
4. **Web Onboarding**: On your main computer, open a web browser and go to http://homeassistant.local:8123. Wait for the "Prepare your new home" process to finish. Create your user account and follow the onboarding prompts.

5.2. Network Configuration and Essential Add-ons

A static IP address is crucial so your other devices always know how to find Home Assistant.

1. **Set Static IP**:
    - In the left-hand menu, go to **Settings > System > Network**.
    - Under "Configure network interfaces," click **Configure**.
    - Expand the IPv4 section. Change the method from "DHCP" to "Static".
    - Enter the IP Address you want for your Pi (e.g., 192.168.1.101), your router's IP as the Gateway (e.g., 192.168.1.1), and the DNS server (e.g., 8.8.8.8).
    - Click **Save**. You may need to refresh your browser and navigate to the new static IP address you just set. Record this IP address.
2. **Install Key Add-ons**: Go to **Settings > Add-ons > Add-on Store** and install the following:
    - File editor
    - Samba share
    - Studio Code Server
    - Google Drive Backup After installing each one, turn on the "Start on boot" and "Watchdog" toggles, then click **Start**.

5.3. Generating the Long-Lived Access Token

The Home Assistant MCP server on your AI Workhorse needs a token to securely communicate

with Home Assistant.

1. In Home Assistant, click on your user profile icon in the bottom-left corner.
2. Scroll down to the “Long-Lived Access Tokens” section.
3. Click **CREATE TOKEN**. Give it a name, like aiden_mcp_server, and click OK.
4. A very long, complex token will be displayed. **This is the only time you will see it.** Copy it immediately to a safe place.

5.4. Linking the Hub to the AI Workhorse

Now, provide the token and URL to the AI Workhorse services.

1. SSH back into your GMKtec K8 server.
2. Open the environment file we created earlier:
   ```
   nano /opt/aiden/.env
   ```
3. Update two lines in this file:
   - Change HA_URL to use the new static IP address of your Home Assistant Pi.
   - Replace the word placeholder for HA_TOKEN with the token you just copied.
   - It should look something like this:
     ```
     HA_URL='http://192.168.1.101:8123'
     HA_TOKEN='ey...long...token...you...copied...'
     ```
4. Save and exit (Ctrl+X, Y, Enter).
5. Restart the services that use these variables to apply the new settings:
   ```
   cd /opt/aiden
   docker compose restart ha-mcp-server memory-proxy
   ```

Your AI Workhorse and Home Assistant Hub are now fully configured and can securely communicate via the new MCP layer.

Section 6: The Sensory & Interaction Network

These Raspberry Pi devices bring Aiden’s presence into different rooms, acting as its eyes and ears.

6.1. Configuring the Lounge Satellite (Raspberry Pi 5)

This device provides the main high-quality camera feed for Frigate.

1. **Flash the OS**: Using Raspberry Pi Imager, flash your storage with **Raspberry Pi OS with Desktop (64-bit)**. Use the advanced options (Ctrl+Shift+X) to set a hostname like lounge-satellite, enable SSH, and configure your Wi-Fi details.
2. **First Boot & Updates**: Boot the Pi, complete the on-screen setup, open a Terminal, and run:
   ```
   sudo apt update && sudo apt upgrade -y
   ```
3. **Enable the Camera**: Run sudo raspi-config, navigate to **Interface Options**, select **Legacy Camera** and enable it. Reboot when prompted.
4. **Install Streaming Software**: Follow the instructions in the original manual source to

install the Python RTSP streaming software. This involves installing dependencies, creating a Python script (rtsp_stream.py), and creating a systemd service (rtsp-stream.service) to ensure the camera stream starts automatically on boot. The final stream address for Frigate will be rtsp://lounge-satellite.local:8554/cam.

6.2. Configuring the Bedroom Satellites (Raspberry Pi Zero 2 W)

These lightweight devices provide voice input, video snapshots, and Bluetooth proxying.

1. **Flash the OS (Headless)**: Using Raspberry Pi Imager, select the Pi Zero 2 W and **Raspberry Pi OS Lite (64-bit)**. Use the advanced options (Ctrl+Shift+X) to enable SSH, set a unique hostname (e.g., lucy-satellite), set credentials, and configure Wi-Fi.
2. **Boot and Connect**: Insert the SD card, connect peripherals, power on, and SSH into the device (e.g., ssh your_username@lucy-satellite.local).
3. **Updates and Camera Setup**: Run sudo apt update && sudo apt upgrade -y and enable the legacy camera via sudo raspi-config. Reboot.
4. **Install Software**: Follow the detailed commands in the original manual source to:
    - Install camera-streamer for a lightweight HTTP video feed.
    - Install the Home Assistant Bluetooth Proxy software.
    - For each piece of software, create the corresponding systemd service file to ensure they launch automatically on boot.
5. **Repeat**: Repeat this entire process for the second Pi Zero, ensuring it is given a different hostname (e.g., molly-satellite).

Section 7: System Integration and Pipeline Configuration

This is where all the separate components become a single, intelligent system. We will connect Home Assistant to the cameras, facial recognition engine, and Aiden's powerful, MCP-driven AI brain.

7.1. Integrating the NVR with Coral TPU Acceleration

First, we tell Frigate about our camera streams and, crucially, configure it to use the Coral TPU for hardware-accelerated object detection.

1. **Configure Frigate**: SSH into your GMKtec K8 server and open the Frigate configuration file:
   ```
   nano /opt/aiden/config/frigate/config.yml
   ```

2. **Update the Configuration**: Delete all content in the default file and paste the following configuration. This version adds the vital detectors section to enable the Coral TPU.
   ```
   # Reference: Section 7, Part 7.1
   # Frigate Configuration File with Coral TPU
   mqtt:
     host: 172.17.0.1 # This is the Docker host IP, do not change
   database:
     path: /config/frigate.db
   ```

```
# Enable the Coral EdgeTPU detector
detectors:
  coral:
    type: edgetpu
    device: usb

ffmpeg:
  hwaccel_args: preset-vaapi

go2rtc:
  streams:
    lounge_cam:
      - rtsp://lounge-satellite.local:8554/cam
    lucy_cam:
      - http://lucy-satellite.local:8080/stream
    molly_cam:
      - http://molly-satellite.local:8080/stream

cameras:
  lounge_cam:
    ffmpeg:
      inputs:
        - path: rtsp://127.0.0.1:8554/lounge_cam
          input_args: preset-rtsp-restream
    detect:
      enabled: True
      width: 1280
      height: 720
    record:
      enabled: True
  lucy_cam:
    ffmpeg:
      inputs:
        - path: rtsp://127.0.0.1:8554/lucy_cam
          input_args: preset-rtsp-restream
    detect:
      enabled: True
    snapshots:
      enabled: True
  molly_cam:
    ffmpeg:
      inputs:
        - path: rtsp://127.0.0.1:8554/molly_cam
          input_args: preset-rtsp-restream
    detect:
      enabled: True
    snapshots:
      enabled: True
```

```
detect:
  # We will only track people
  objects:
    track:
      - person
```

3. **Restart and Verify**: Save the file and restart Frigate (cd /opt/aiden && docker compose restart frigate).
    - **Verification Step**: Open the Frigate web UI at http://YOUR_K8_IP_ADDRESS:5000. Navigate to the **System** page. Confirm that a "coral" detector is listed, with an inference speed typically under 15ms. This provides immediate feedback that the accelerator is working correctly.
4. **Install Frigate Integration**: Follow the instructions in the original manual to install the Frigate integration via HACS in Home Assistant.

7.2. Configuring Facial Recognition with Double Take

Follow the instructions in the original manual (Section 6, Part 6B) to configure Double Take and train it with photos of family members.

7.3. Building the High-Performance Voice Pipeline with MCP

This process connects Home Assistant to the dedicated Wyoming protocol services running on the AI Workhorse.

1. **Add Speech-to-Text (Whisper)**:
    - In Home Assistant, go to **Settings > Devices & Services > Add Integration**.
    - Search for and select **Wyoming Protocol**.
    - For **Host**, enter the IP address of the K8 server. For **Port**, enter 10300. A "Whisper" device will be added.
2. **Add Text-to-Speech (XTTS)**:
    - Add another **Wyoming Protocol** integration.
    - For **Host**, enter the K8's IP address. For **Port**, enter 8020. An "openWakeWord" device will be added (this is a naming quirk; it is the XTTS service).
3. **Create the Voice Assistant Pipeline**:
    - Go to **Settings > Voice assistants**.
    - Click the blue **+ ADD ASSISTANT** button.
    - **Name**: Aiden
    - **Conversation language**: English
    - **Speech-to-text**: Select the newly added **Whisper** service.
    - **Text-to-speech**: Select the **openWakeWord** service (this is the XTTS service).
    - Click **Create**.

7.4. Connecting Aiden's Brain: Linking the Pipeline to the Memory Proxy

Finally, we'll teach Home Assistant how to talk to the custom, MCP-aware Memory Proxy we

built.

1. **Add the REST Command**:
    - Go to **Settings > Add-ons > File editor**.
    - In the file tree on the left, click on configuration.yaml.
    - Add the following code to the very end of the file, replacing YOUR_K8_IP_ADDRESS.

```
# Reference: Section 7, Part 7.4
# This allows HA to send prompts to the custom AI proxy
rest_command:
  ask_aiden:
    url: "http://YOUR_K8_IP_ADDRESS:8080/ask"
    method: "POST"
    content_type: "application/json"
    payload: '{"prompt": "{{ prompt }}", "device_id": "{{
device_id | default("") }}"}'
    timeout: 60
```

    - Save the file. Go to **Developer Tools > YAML** and click **CHECK CONFIGURATION**. If valid, click the **RESTART** button.
2. **Create the Master "Ask Aiden" Script**:
    - Go to **Settings > Automations & Scenes > Scripts**.
    - Click **+ ADD SCRIPT** and choose "Create new script".
    - Switch to YAML mode (three dots in the top right -> "Edit in YAML").
    - Delete the boilerplate and paste this in :

```
# Reference: Section 7, Part 7.4
# This script calls the AI, gets a response, and speaks it
alias: Ask Aiden Master Script
sequence:
  - service: rest_command.ask_aiden
    data:
      prompt: "{{ sentence }}"
    response_variable: aiden_response
  - service: tts.speak
    data:
      cache: false
      media_player_entity_id: "{{ media_player_entity_id }}"
      message: "{{ aiden_response.json.response_text }}"
    target:
      entity_id: "{{ media_player_entity_id }}"
```

    - Click **Save**.
3. **Link the Script to your Pipeline**:
    - Go back to **Settings > Voice assistants**.
    - Click on your **Aiden** pipeline.
    - For **Conversation agent**, select script.ask_aiden_master_script.
    - Click **Update**.

You can now test this! Go to your Home Assistant dashboard, click the Assist icon (top right), and type a question like "Is the garage door open?" It should query the AI proxy, which will use

the Home Assistant MCP server to get the state, and give you a correct answer spoken in Aiden's high-quality voice.

Section 8: Bringing Aiden to Life: Automations and Usage

With the core technical integrations complete, the final step is to build the personalized automations and learn how to use the system's advanced features.

8.1. Creating Personalized Routines in Home Assistant

The automation examples from the original manual remain excellent starting points. Implement the **Room-Aware Morning Routine** and the **Bedtime Security Check** as described in the original source (Section 7, Parts 7B and 7C). These automations demonstrate how to use Home Assistant's YAML configuration to create context-aware, proactive behaviors for Aiden.

8.2. Expanding Aiden's Knowledge with the RAG System

The ability to upload documents to Aiden's memory is now managed by the ChromaDB MCP server. While the user-facing process remains similar, the underlying mechanism is more robust. A future enhancement would be to add an add_document tool to the ChromaDB MCP server. For now, a direct API endpoint on that server could be created for uploads. However, for the scope of this manual, the functionality is considered an advanced topic for user expansion.

8.3. How to Change and Experiment with LLM Models

Your system is designed to be flexible. You can easily swap the main Large Language Model (LLM) that Aiden uses.

1. **Find a Model**: Browse the models available on OpenRouter.ai. Find one you'd like to try (e.g., google/gemini-flash) and copy its identifier string.
2. **Update Configuration**: SSH into your GMKtec K8 server and open the environment file:
   `nano /opt/aiden/.env`
3. **Change the Model**: Find the line MAIN_LLM_MODEL=... and replace the existing model with the new one.
4. **Restart the Proxy**: Save the file and restart the Memory Proxy to apply the change:
   ```
   cd /opt/aiden
   docker compose restart memory-proxy
   ```
   Aiden will now use the new model for its thinking.

Section 9: Maintenance and Troubleshooting

A robust system needs good backups and a clear troubleshooting process.

9.1. System Backups and Update Procedures

Implement the backup and system update procedures as outlined in the original manual (Section 8, Part 8D). This includes:

- Configuring the **Google Drive Backup** add-on in Home Assistant for automated, off-site backups.
- Creating and periodically running the backup_aiden.sh script on the AI Workhorse to archive the vital /opt/aiden configuration directory.
- Performing monthly system updates on all servers (sudo apt update && sudo apt upgrade -y).

9.2. Troubleshooting the MCP Architecture

When issues arise, the new service-oriented architecture allows for systematic, component-level diagnostics. The most important command is docker logs <container_name>.

- **Symptom**: "Aiden isn't responding or gives a generic error."
    1. **Start at the Orchestrator**: Check the Memory Proxy's logs first. This log will show the user's prompt and which MCP tool the agent decided to call.
       `docker logs memory-proxy`
       Look for lines from the agent executor showing the tool name (e.g., home_assistant_get_entity_state). This tells you which downstream service to investigate.
    2. **Check the Tool Server**: If the Memory Proxy tried to call a tool on the ha-mcp-server, check that server's logs for errors.
       `docker logs ha-mcp-server`
       This log will show if it received the request and if it encountered an error when communicating with the Home Assistant API (e.g., an invalid token or wrong URL).
    3. **Isolate with curl**: You can test an MCP server directly, bypassing the Memory Proxy and LLM entirely. For example, to test the get_entity_state tool on the Home Assistant MCP server, you can send a direct HTTP request from the K8 server. This requires knowing the specific format the MCP server expects, but is an invaluable advanced debugging technique.
- **Symptom**: "My camera feed is black."
    - This issue is unchanged from the previous architecture. Check the Frigate log for errors connecting to the camera's RTSP stream:
      `docker logs frigate`

9.3. Expert Note: Optimizing AI Workloads (Coral TPU vs. CPU)

A common request is to use the Google Coral TPU to accelerate all AI tasks, including Whisper Speech-to-Text. However, due to fundamental differences in model architecture and hardware design, this is not technically feasible and leads to a suboptimal system.

The Coral TPU is a highly specialized accelerator designed for TensorFlow Lite vision models. Its hardware and on-chip memory (typically 8MB) are purpose-built for the mathematical operations common in object detection. Speech recognition models like Whisper are significantly larger (the smallest model is ~75MB) and have a different architecture that the Coral's hardware cannot process efficiently, if at all.

This system's architecture therefore employs a strategic, two-pronged approach to achieve maximum performance:

1. **Dedicated Vision Acceleration**: The Coral TPU is applied to its ideal task: accelerating object detection in Frigate. This offloads the most intensive, continuous processing task from the main CPU, dramatically reducing system load, lowering power consumption, and improving the overall responsiveness of the NVR system.
2. **Optimized CPU-based Voice Processing**: The powerful CPU resources of the AI Workhorse are leveraged to run a highly optimized, dedicated Whisper container. The rhasspy/wyoming-whisper image uses backends that are significantly faster and more memory-efficient for CPU inference than the standard model.

This architecture achieves the goal of a faster, more responsive system by applying each specialized component to the task it performs best. The result is superior overall performance compared to a design that attempts to force a single accelerator to handle all AI workloads. Furthermore, by abstracting the voice service behind an MCP server, the system becomes hardware-independent. If a future STT model is released that *can* run on a TPU, only the internal logic of the whisper-mcp-server would need to be updated; the rest of the system would require no changes.

9.4. Network Port Allocation

This table provides an indispensable reference for configuring firewalls, troubleshooting connectivity, and understanding the system's network footprint.

Service	Host Port	Protocol	Purpose / Accessed By
Frigate (Web UI)	5000	TCP	Internal admin UI for NVR
Frigate (RTMP)	8554	TCP	Camera stream ingress
Double Take (Web UI)	3000	TCP	Internal admin UI for facial recognition
XTTS (API)	8020	TCP	Home Assistant (Text-to-Speech service)
Ollama (API)	11434	TCP	Memory Proxy (Local LLM fallback)
ChromaDB (API)	8001	TCP	ChromaDB MCP Server
Whisper (Wyoming)	10300	TCP	Whisper MCP Server (Speech-to-Text service)
Memory Proxy (API)	8080	TCP	Home Assistant (LLM Brain service)
HA MCP Server	9001	TCP	Memory Proxy (HA Tools)
Chroma MCP Server	9002	TCP	Memory Proxy (RAG Tools)
Whisper MCP Server	9003	TCP	Memory Proxy (Voice Tools)

Conclusion

You have successfully built a sophisticated, private, and personalized AI smart home from the ground up. By deconstructing a monolithic application and rebuilding it as a resilient, extensible ecosystem of interconnected AI services, you have adopted a forward-looking architectural pattern. This service-oriented approach, built on the open standard of the Model Context Protocol, will allow the system to easily grow, adapt, and incorporate new hardware and software technologies for years to come.

This manual provides a robust foundation, but it is merely the starting point. The system is designed for continuous evolution. New smart devices can be integrated, new tools can be exposed via custom MCP servers, and new automations can be designed to handle increasingly complex tasks. You are in full control of your home's data and its intelligence. Well done, and enjoy your new smart home.

Works cited

1. Connectors and MCP servers - OpenAI API, https://platform.openai.com/docs/guides/tools-connectors-mcp 2. Building a Server-Sent Events (SSE) MCP Server with FastAPI - Ragie, https://www.ragie.ai/blog/building-a-server-sent-events-sse-mcp-server-with-fastapi 3. Use MCP servers in VS Code, https://code.visualstudio.com/docs/copilot/customization/mcp-servers 4. jlowin/fastmcp: The fast, Pythonic way to build MCP servers ... - GitHub, https://github.com/jlowin/fastmcp 5. Building an MCP Server with FastAPI and FastMCP - Speakeasy, https://www.speakeasy.com/mcp/building-servers/building-fastapi-server 6. Debian -- Details of package python3-wyoming in sid, https://packages.debian.org/sid/python3-wyoming 7. wyoming - PyPI, https://pypi.org/project/wyoming/ 8. MCP Adapters for LangChain and LangGraph, https://changelog.langchain.com/announcements/mcp-adapters-for-langchain-and-langgraph 9. Model Context Protocol (MCP) - Docs by LangChain, https://docs.langchain.com/oss/python/langchain/mcp 10. Wyoming Protocol - Home Assistant, https://www.home-assistant.io/integrations/wyoming/