Relatório - Documentação Técnica

Alunos:

- Roldão de Oliveira Neto - 2022002954
- Rodrigo William da Silva - 2022002936
- Ryan Felipe Ferreira Ribeiro - 2021020958

Introdução e Motivação

As arquiteturas *cloud-centric* dos assistentes virtuais comerciais atuais impõem riscos críticos de privacidade, exigindo *streaming* contínuo de áudio e processamento remoto de dados sensíveis. O problema central é a dependência de servidores de terceiros para a interpretação de linguagem natural (NLU), o que expõe o usuário à mineração de dados e exfiltração de telemetria. Falta no mercado uma solução *open source* baseada em *edge computing* que garanta a soberania dos dados, em que o processamento é inteiramente local (*on-device*) e a interface de rede (WAN) é ativada apenas para requisições de busca explícitas (*web search*), eliminando vetores de vazamento passivo de informações.

O projeto utiliza, como *hardware* de referência, uma Raspberry Pi 5 (8 GB de RAM) para hospedar o *pipeline* de inferência do Agente SLM (*Small Language Model*). A arquitetura de *software* é projetada para ser agnóstica e modular, permitindo o *deploy* em *hardware* legado x86/x64 (desktops e laptops antigos) com Linux Debian. Essa abordagem valida a viabilidade técnica de rodar modelos de linguagem quantizados em infraestrutura de consumo de baixo custo, transformando equipamentos obsoletos em nós de processamento locais seguros, reduzindo e-waste e descentralizando a inteligência computacional fora dos *datacenters* proprietários.

O objetivo é implementar um assistente de voz autônomo capaz de orquestrar a automação residencial via *"function calling"* local.

- **Critério de Sucesso Primário (Alta Prioridade):** Precisão semântica e confiabilidade do SLM na classificação de intenções. O sistema deve demonstrar alta acurácia na distinção entre comandos de controle local (IoT) e necessidades de consulta externa, garantindo a execução correta dos *tools/funções*.
- **Critério de Sucesso Secundário (Baixa Prioridade):** Latência de inferência. Nesta versão *alpha*, a otimização de performance é preterida em favor da corretude lógica da execução das tarefas.

Revisão Bibliográfica

O desenvolvimento de assistentes virtuais baseados em Grandes Modelos de Linguagem (LLMs) operando na borda (*edge*) envolve a interseção de três áreas principais de pesquisa: a computação de borda voltada para privacidade, a otimização de modelos de linguagem (SLMs e Quantização) e a arquitetura de agentes cognitivos (RAG e Function Calling). Esta seção fundamenta as decisões de projeto com base no estado da arte dessas tecnologias.

Computação de Borda e Privacidade

Tradicionalmente, assistentes de voz comerciais dependem de arquiteturas centralizadas em nuvem para o Processamento de Linguagem Natural (NLP). Embora escalável, essa abordagem introduz latência de rede e vetores críticos de vulnerabilidade de privacidade. Estudos recentes sobre *Edge AI* enfatizam a necessidade de processamento local (*on-device*) para mitigar a exfiltração de dados sensíveis. Conforme apontado por pesquisas sobre riscos em assistentes de voz [1], a execução local elimina a dependência de conectividade constante e garante que dados biométricos (voz) não deixem o perímetro físico do usuário, alinhando-se aos princípios de *Privacy by Design*.

2.2. Small Language Models (SLMs) e Quantização

A viabilidade de executar modelos generativos em hardware com recursos limitados, como a Raspberry Pi 5, é possibilitada pelo avanço dos *Small Language Models* (SLMs) e técnicas de compressão. A quantização reduz a precisão dos pesos do modelo (ex: de 16-bit para 4-bit) para diminuir o consumo de memória. O trabalho seminal de Dettmers et al. (2023) sobre QLoRA [2] demonstrou que é possível manter a performance de modelos de linguagem quantizados em 4 bits com degradação mínima de qualidade, viabilizando a inferência em dispositivos de consumo. O modelo escolhido para este projeto, Llama 3.2, incorpora essas otimizações nativamente, sendo desenhado especificamente para eficiência em cenários de *edge* [3].

2.3. Agentes Cognitivos e Function Calling

A evolução dos LLMs permitiu a transição de geradores de texto passivos para agentes autônomos. A técnica de *Function Calling* (ou *Tool Use*) implementa o paradigma *ReAct* (*Reasoning + Acting*), proposto por Yao et al. (2023) [4]. Essa abordagem permite que o modelo gere raciocínios intercalados com ações, estruturando saídas para invocar funções determinísticas (como acender uma luz) em vez de apenas gerar texto livre. Isso

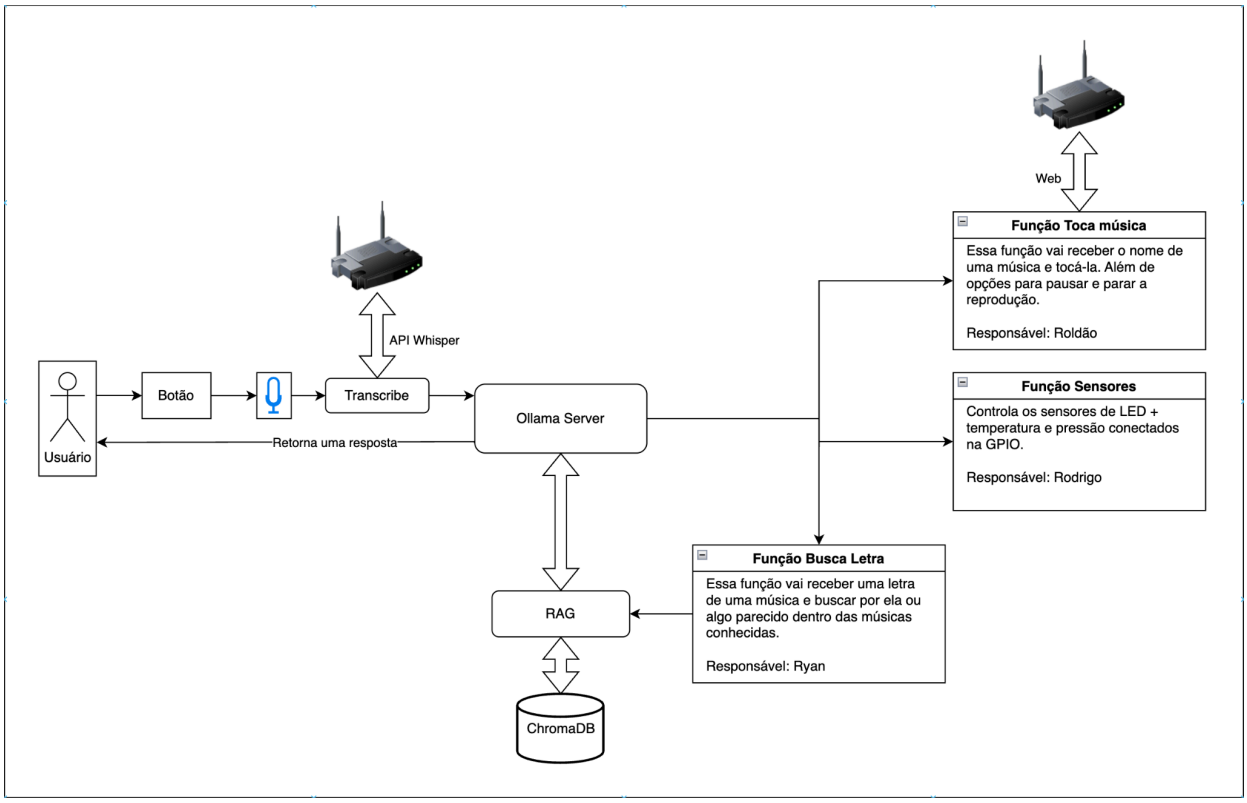
é crítico para sistemas IoT, onde a precisão semântica na ativação de atuadores é prioritária sobre a criatividade da resposta.

2.4. Retrieval-Augmented Generation (RAG)

Para tarefas que exigem conhecimento factual específico não contido nos pesos do modelo — como a identificação de letras de músicas no projeto — utiliza-se a técnica de *Retrieval-Augmented Generation* (RAG). Originalmente proposta por Lewis et al. (2020) [5], o RAG combina a capacidade generativa do modelo com a recuperação de informação baseada em similaridade vetorial. Ao injetar contexto relevante (trechos de letras recuperados via *embeddings*) no prompt do modelo, o sistema reduz "alucinações" e aumenta a precisão da resposta, permitindo uma busca semântica robusta mesmo diante de entradas imprecisas do usuário.

Projeto do Sistema

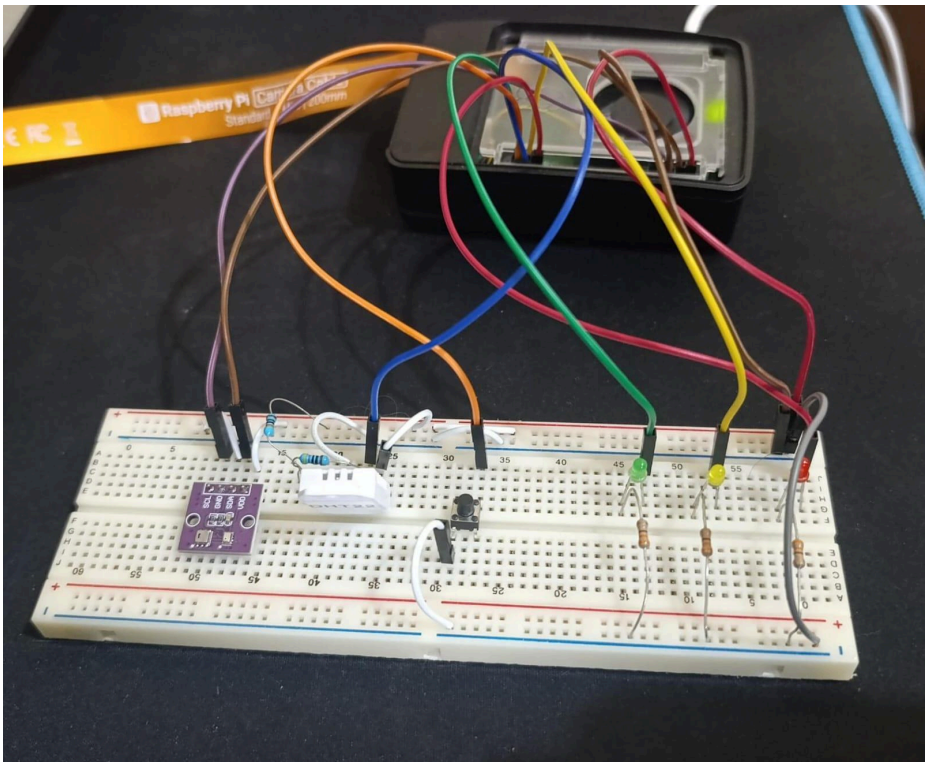
Diagrama de Arquitetura

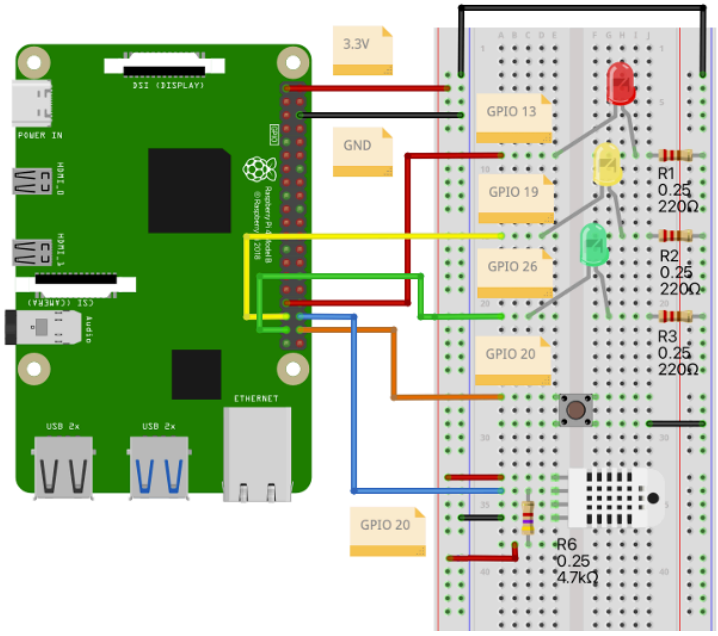


Seleção e Justificativa do Modelo SLM

Selecionou-se o modelo Llama 3.2 3B; a escolha fundamenta-se no equilíbrio ótimo entre capacidade de raciocínio (reasoning) e eficiência computacional para edge devices. Com 3 bilhões de parâmetros, o modelo viabiliza a execução local na Raspberry Pi 5 com *overhead* de memória reduzido, preservando recursos críticos e cumprindo os requisitos de <5B de parâmetros do projeto. O Llama 3.2 apresenta otimizações nativas para *tool use* e tarefas de resumo, garantindo a robustez necessária para o *function calling* estruturado.

Configuração de Hardware e Diagramas de Circuito





Detalhes da Implementação de Software

Função Main + Inferência:

A arquitetura de software do sistema foi dividida em dois módulos principais para garantir a separação de responsabilidades (separation of concerns): o main.py, focado no ciclo de vida da aplicação e I/O, e o inference.py, dedicado à lógica do agente cognitivo.

Main Loop O loop principal da aplicação (`main.py`) opera em um modelo de bloqueio síncrono. Para a captura de áudio, optou-se por uma abordagem "Push-to-Talk" utilizando a biblioteca `gpiozero` para monitorar interrupções no GPIO 20. Esta decisão técnica elimina a necessidade de um motor de *Wake Word* (como Porcupine ou Snowboy) rodando em background, liberando ciclos de CPU da Raspberry Pi 5 para a inferência do modelo de linguagem. O áudio é capturado apenas mediante a ação física do usuário, garantindo privacidade por design (privacy-by-design) e eficiência energética.

Implementação do Function Calling (Inference Engine) O módulo `inference.py` implementa a técnica de "Function Calling" nativa do Llama 3.2. Diferente de uma

abordagem simples de *prompt engineering*, o sistema injeta definições de ferramentas via esquema JSON (`tools_schema.py`) no contexto do modelo. O fluxo de execução segue uma lógica de dois passos:

1. **Inferência de Intenção:** O modelo recebe o input do usuário e decide se responde com texto ou gera um token especial de `tool_calls`.
2. **Despacho Dinâmico:** Se uma ferramenta é solicitada, o script intercepta a chamada, extrai os argumentos (ex: `{"status": "on"}`), valida-os contra o esquema definido e executa a função Python correspondente mapeada no dicionário `AVAILABLE_FUNCTIONS`.
3. **Injeção de Resultado:** O retorno da função (ex: "Luz ligada com sucesso") é anexado ao histórico da conversa como uma mensagem de papel `tool`, permitindo que o modelo gere uma resposta final natural ciente do que acabou de acontecer no mundo físico.

Guardrails e System Prompt Para mitigar alucinações, um problema comum em modelos quantizados de 3B parâmetros, implementou-se um `SYSTEM_PROMPT` rigoroso no início da stack de contexto. Este prompt define regras de exclusão, instruindo explicitamente o modelo a não responder perguntas sobre o estado dos dispositivos baseando-se em seu conhecimento pré-treinado, mas sim obrigando-o a invocar as ferramentas `get_environment_metrics` ou `control_light`. Além disso, esquemas de validação com tipos enumerados (`enum: ["on", "off"]`) no `tools_schema.py` impedem que o modelo envie parâmetros inválidos para o hardware.

Observabilidade e Tratamento de Erros O sistema incorpora um módulo de `logging` detalhado que registra o fluxo de decisão (User Input → Tool Selection → Execution → Final Response) e possíveis falhas na pipeline. Blocos de tratamento de exceção (`try-except`) envolvem tanto a captura de áudio quanto a inferência, garantindo que falhas transitórias em sensores ou erros de decodificação no Ollama não causem o encerramento abrupto (crash) do processo principal, mantendo o assistente disponível para novos comandos.

Função Tocar Música:

O componente de áudio foi implementado através da classe `MusicPlayer`, desenvolvida para gerenciar a busca e execução de *streams* de áudio do YouTube diretamente pelo terminal. A arquitetura da solução baseia-se na integração de dois utilitários externos controlados via Python:

1. **Extração de Mídia (`yt_dlp`):** A biblioteca é utilizada para realizar buscas (`ytsearch1:`) e extrair a URL direta do fluxo de áudio com a melhor qualidade disponível, sem a necessidade de baixar o arquivo para o disco.
2. **Controle de Reprodução (`mpv` + Sockets):** A execução do áudio é delegada ao `mpv`, rodando em um processo filho (`subprocess`). Para permitir o controle dinâmico (pausar ou parar) enquanto a música toca, o `mpv` é inicializado com um servidor IPC (`--input-ipc-server`). A comunicação é feita via Sockets Unix, enviando comandos em formato JSON para o processo em execução, garantindo baixa latência e controle assíncrono.

Para o correto funcionamento do módulo, é necessário instalar o player multimídia e a biblioteca de extração no ambiente Linux:

- Instalação do Player: `sudo apt install mpv`
- Instalação da Biblioteca Python: `pip install yt_dlp`

Função Buscar Música:

Visão Geral

A função detect_music implementa uma arquitetura RAG (Retrieval-Augmented Generation) para identificar músicas a partir de trechos de letras fornecidos pelo usuário. Esta abordagem combina busca semântica com raciocínio de modelos de linguagem, permitindo identificação precisa mesmo quando os usuários fornecem letras incompletas ou informais.

Banco de Dados Vetorial e Embeddings

O sistema utiliza ChromaDB como banco de dados vetorial persistente para armazenar letras de músicas. Cada música é convertida em embeddings vetoriais de alta dimensionalidade usando o modelo nomic-embed-text da Ollama, que executa localmente. Esses embeddings capturam o significado semântico das letras ao invés de apenas palavras-chave, permitindo que o sistema encontre correspondências entre frases como "is not my love" e "Billie Jean is not my love" apesar das diferenças na formulação. O banco de dados armazena tanto as representações vetoriais quanto metadados incluindo nomes das músicas e arquivos de origem.

Processo de Recuperação

Quando um usuário fornece trechos de letra, o texto de entrada é transformado em um vetor usando o mesmo modelo de embedding. Este vetor é então comparado com todas

as músicas armazenadas usando métricas de similaridade. O sistema recupera os 3 trechos de letras mais similares, fornecendo múltiplos candidatos para o processo de identificação. Esta abordagem de busca semântica lida efetivamente com variações na formulação, erros de digitação e versos incompletos.

Integração com Modelo de Linguagem

Os candidatos de músicas recuperados são passados para um modelo de linguagem (llama3.2) junto com um prompt especializado. O LLM atua como um especialista musical, analisando o contexto e a entrada do usuário para determinar qual música melhor corresponde às letras fornecidas. Operando com baixa temperatura (0.1) garante respostas consistentes e determinísticas. O LangChain orquestra todo o pipeline, conectando embeddings, recuperação, prompts e inferência em um fluxo de trabalho integrado.

Resposta e Performance

A função retorna uma resposta JSON estruturada contendo o nome da música identificada, fontes de confiança (os 3 principais candidatos da recuperação) e tempo de processamento. Isso permite que sistemas chamadores verifiquem resultados e avaliem a confiança. O tratamento de erros garante falhas graciosas com mensagens informativas.

Função Controla GPIO (LEDs + Temperatura e Pressão):

O módulo `hardware.py` atua como a camada de abstração de hardware (HAL), isolando a lógica de controle físico do restante da aplicação. A interação com os periféricos conectados à GPIO da Raspberry Pi 5 foi implementada utilizando as bibliotecas `gpiozero` para componentes digitais e `adafruit_dht` para a leitura de sensores complexos.

Monitoramento Ambiental (Temperatura e Umidade) Diferente do escopo inicial que previa medição de pressão, o sistema final concentrou-se no monitoramento termo-higrométrico utilizando o sensor DHT22. Este sensor foi conectado à GPIO 16. Para garantir a integridade do sinal digital e evitar leituras flutuantes (floating inputs) comuns em protocolos *one-wire*, foi implementado um resistor de *pull-up* de 4.7kΩ entre o pino de dados e o VCC (3.3V). No software, a função `get_environment_metrics` encapsula a leitura do sensor em um bloco `try-except`, mitigando falhas de *timing* comuns ao driver do DHT em sistemas Linux não-tempo-real, garantindo que o assistente reporte erros de leitura graciosamente em vez de travar o sistema.

Controle de Atuadores e Feedback Visual O sistema de atuação foi dividido em duas categorias funcionais:

1. **Atuação de Comando (Simulação de Carga):** Um LED verde conectado à GPIO 26 simula o controle de uma lâmpada residencial. A função `control_light` gerencia o estado deste pino e atualiza uma variável de estado em memória (`_system_state`), permitindo que o sistema tenha persistência volátil do status (Ligado/Desligado).
2. **Feedback de Processamento:** Para melhorar a experiência do usuário (UX) diante da latência de inferência, um LED amarelo foi adicionado à GPIO 19. Este atuador opera de forma assíncrona ao loop principal, piscando durante os períodos de processamento do SLM (Small Language Model) para indicar atividade computacional.

Mapeamento de Hardware (Pinout)

- **Entrada de Voz (Botão):** GPIO 20 (Pull-up interno ativado).
- **Sensor DHT22:** GPIO 16 (Pull-up externo de 4k7Ω).
- **Atuador Lâmpada (LED Verde):** GPIO 26.
- **Indicador de Status (LED Amarelo):** GPIO 19.

Abordagem de Integração do SLM

A integração do Small Language Model (SLM) ao ecossistema do projeto foi realizada através do framework **Ollama**, escolhido por oferecer uma camada de abstração eficiente para a execução de modelos quantizados em arquiteturas ARM64 (aarch64).

Infraestrutura de Execução (Runtime Environment) O servidor de inferência roda localmente na Raspberry Pi 5 como um serviço de segundo plano (*systemd service*), instalado via script de instalação nativo (`curl`). Esta arquitetura desacopla o motor de inferência da lógica da aplicação: o script Python `inference.py` atua como cliente, comunicando-se com o servidor Ollama via API HTTP local (através da biblioteca wrapper oficial `ollama-python`).

Configuração do Modelo O modelo selecionado foi o **Llama 3.2 3B**, utilizando a quantização padrão de 4-bits (Q4_K_M). Esta escolha estratégica, alinhada às recomendações da disciplina, permitiu a execução do modelo inteiramente na memória RAM (8GB) da Raspberry Pi 5 sem a necessidade de *offloading* agressivo para a swap ou ajustes manuais de alocação de memória de vídeo (`gpu_mem`), mantendo a estabilidade do sistema operacional Raspberry Pi OS Bookworm.

Estratégia de Chamada de Ferramentas (Native Tool Use) Diferente de abordagens tradicionais que exigem o parsing complexo de strings de saída, a integração utilizou a funcionalidade nativa de *Tool Use* do Llama 3.2. No módulo `inference.py`, a lista de dicionários definidos em `tools_schema.py` é passada diretamente para o parâmetro `tools` da função `ollama.chat`. Isso delega ao próprio modelo a responsabilidade de estruturar a saída. O sistema não precisa adivinhar quando chamar uma função; o servidor Ollama retorna explicitamente um objeto `tool_calls` quando a intenção do usuário exige interação física, simplificando drasticamente o código de controle e eliminando a necessidade de *few-shot prompting* complexo para formatação de JSON.

Parâmetros de Amostragem Para este MVP (Minimum Viable Product), optou-se por manter os hiperparâmetros de geração (como *temperature* e *top_k*) nos valores padrão do framework. Os testes práticos demonstraram que o comportamento *default* do modelo oferece um equilíbrio satisfatório entre criatividade na conversação e aderência às instruções do `SYSTEM_PROMPT` para a execução de comandos, sem a necessidade de *fine-tuning* adicional dos parâmetros de amostragem.

Estratégia de Engenharia de Prompts

Agent Principal:

Utilizamos estratégias de orientação “agêntica” para instruir o modelo de quando ou não utilizar determinada função. Como tínhamos ferramentas semanticamente semelhantes como tocar música e detectar música, era necessário instruções para o Agent ser capaz de desambiguar.

Agent RAG:

Para o processo de detecção da música cantada pelo usuário usamos uma estratégia de RAG (Retrieval Augmented Generation), no qual, a partir dos trechos retornados pela busca semântica instruímos o llm a responder baseado nos trechos retornados e na entrada do usuário qual música faz mais sentido ser a cantada pelo usuário.

Resultados e Discussão

Principais Conquistas

Sistemas de agentes de IA generativa estão em alta no mercado, porém em sua maioria temos aplicações que rodam em grandes clusters com poderio de computação e

escalabilidade. Dito isso, uma das grandes conquistas a se celebrar é justamente o fato de termos conseguido trazer uma aplicação que se encaixa em um cenário real de uso dentro de um ambiente de execução limitado.

Além disso, uma outra conquista foi justamente o fato de termos conseguido integrar uma interface speech-to-text o que permitiu uma experiência mais próxima de uma “Alexa” que era o intuito no início do projeto. Como também vale ressaltar a diversificação das funções do agente que vão desde interação com hardware para acender e apagar leds até operações mais alto nível como tocar música e implementação de busca fundamentada com RAG.

Desafios Enfrentados e Soluções

Dentre os desafios enfrentados podemos destacar inicialmente a precisão da “function calling” no qual a medida que fomos agregando novas funções o llama3.2 começou a apresentar maior variabilidade e imprecisão ao decidir entre qual função acionar em cada interação. A solução foi deixar o system prompt mais robusto com instruções claras de quando ele deveria chamar cada uma das funções. A primeira vista pode se pensar que apenas as descrições nos próprios “schemas” de cada função pode ser suficiente, porém à medida que a quantidade de possibilidades aumenta é necessário “reforçar” esses contratos via instrução sistêmica.

Um detalhe externo que também pode se considerar um desafio foi justamente o fato de não termos uma placa de desenvolvimento por integrante, isso exigiu que as funções fossem feitas e testadas de maneira apartada e em seguida juntamos cada componente o que exigiu um processo de integração extra.

Por fim a questão do hardware limitado implicou em uma latência de inferência que já era esperada, porém como envolvemos em uma das funções uma inferência extra para a detecção de música o processo ficou lento. Não houve em si uma solução para isso, nós reconhecemos que para quesito experimental conseguimos testar o que queríamos e para melhor experiência seria necessário um upgrade ou computação distribuída.

Trabalho Futuro e Conclusões

Melhorias Potenciais e Considerações de Escalabilidade

- Desenvolvimento de uma interface gráfica básica para melhorar a intuição e o controle do sistema por um usuário leigo.

- Desenvolvimento de novas funções para controlar dispositivos externos via Bluetooth.

- Desenvolvimento de feature de múltiplas opções de “cérebros” (SLM agentes) com diferentes quantidades de parâmetros e arquiteturas para abordar hardwares mais potentes e mais fracos.

- Melhoria da latência e precisão da aplicação.

Conclusões

A implementação deste MVP validou com êxito a viabilidade arquitetural de um assistente doméstico *privacy-first* operando em *edge*. A infraestrutura baseada na Raspberry Pi 5 demonstrou capacidade de processamento suficiente para a inferência do SLM, atendendo plenamente aos critérios de sucesso estabelecidos para a precisão do *function calling*. O agente foi capaz de interpretar comandos de linguagem natural e executar funções de controle de mídia (música) localmente, sem dependência de APIs em nuvem para o processamento semântico. Este resultado confirma a tese de que é possível dissociar a conveniência da automação residencial da exfiltração de dados, estabelecendo uma base sólida para a integração de *hardware* legado e a expansão futura do ecossistema *open source*.

Referências Bibliográficas

[1] K. Cheng et al., "Security and Privacy Problems in Voice Assistant Applications: A Survey," *arXiv preprint arXiv:2304.09486*, 2023.

[2] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[3] AI@Meta, "Llama 3 Model Card," *Meta AI*, 2024. Disponível em: https://github.com/meta-llama/llama3.

[4] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," *International Conference on Learning Representations (ICLR)*, 2023.

[5] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.