Análise do uso de Modelos de Linguagem de Grande Escala na Geração de Códigos para Automação Residencial

Antonio Cruz
Universidade Federal da Bahia
Departamento de Ciência da Computação
Salvador, Brasil
antonioscn@ufba.br

Paulo H. L. Rettore
Universidade Federal de Minas Gerais
Fraunhofer FKIE
Departamento de Ciência da Computação
Communication Systems Department
Belo Horizonte, Brasil
rettore@dcc.ufmg.br

Bruno P. Santos
Universidade Federal da Bahia
Departamento de Ciência da Computação
Salvador, Brasil
bruno.ps@ufba.br

ABSTRACT

This paper presents a comparative experimental evaluation of Large Language Models (LLMs) for generating YAML code in home automation using the Home Assistant (HA) platform. Four models were tested: GPT-4o, GPT-4.5, Gemini 2.5 Flash, and Gemini 2.5 Pro. Tests covered three complexity levels and two prompt types. Performance was analyzed with a 2^k factorial design and validated with ANOVA. Task complexity was the main factor affecting success, while prompt specificity reduced errors. All models succeeded in basic tasks, but only Gemini 2.5 Pro maintained functionality in complex scenarios. Error analysis showed that logic errors were most frequent, followed by syntax and entity errors. A local fine-tuned model (DeepSeek-r1:14B) was also evaluated. It worked for simpler tasks but required manual adjustments. The results highlight the importance of model robustness, prompt quality, and deployment context in LLM-based code generation for home automation.

KEYWORDS

Modelos de Linguagem de Grande Escala, Automação Residencial, Geração de Código, Home Assistant

1 INTRODUÇÃO

A automação residencial está cada vez mais presente no cotidiano das pessoas [4, 9]. *Softwares* especializados permitem controlar e integrar diversos dispositivos inteligentes a partir de uma única plataforma, funcionando como um *hub* central para gerenciar e automatizar funções. Entretanto, usuários não especialistas muitas vezes se veem sobrecarregados com os detalhes técnicos envolvidos na configuração e manutenção do comportamento desejado de seus ambientes inteligentes. Muitos não têm experiência em programação e têm medo de "quebrar" o sistema [5].

Para mitigar esse problema, têm sido desenvolvidos métodos intuitivos que permitem aos usuários interagir com dispositivos inteligentes e automatizar casas sem conhecimento técnico ou experiência em programação [18, 19]. Nesse contexto, os modelos de Linguagem de Grande Escala (LGE) se destacam por traduzir comandos em linguagem natural em diversas formas de saída, incluindo textos, histórias, resumos e trechos de código [19]. Estudos recentes indicam que LGEs podem gerar código ou *scripts* funcionais diretamente a partir de descrições textuais [18], ampliando o potencial de personalização e acessibilidade da automação residencial.

Embora os LGEs tenham encontrado aplicações em diversos domínios, sua integração efetiva em sistemas de automação residencial ainda é pouco explorada [10]. Nesse contexto, surge a oportunidade de explorar o uso dessas tecnologias na geração automatizada de rotinas, especialmente por meio da conversão de linguagem natural em código *YAML Ain't Markup Language (YAML)*, que é amplamente empregado em plataformas de automação. Além disso, essa abordagem poderia reduzir a necessidade de conhecimentos técnicos e facilitar o processo de configuração por usuários comuns.

O Home Assistant (HA) é uma plataforma de automação residencial de código aberto que prioriza o controle local e a privacidade do usuário [1]. Atualmente, é uma das soluções mais populares para a criação de rotinas e o gerenciamento de dispositivos inteligentes. Um de seus diferenciais é a possibilidade de configurar automações por meio de *scripts* em formato YAML Ain't Markup Language (YAML), que podem ser gerados por LGEs. Contudo, a integração entre os Modelos e o HA ainda apresenta desafios que devem ser analisados, como a verificação funcional das rotinas geradas, a interpretação correta de instruções e a comparação de desempenho entre diferentes LGEs aplicados a esse contexto.

Diante desse contexto, este trabalho propõe uma avaliação experimental da capacidade de diferentes LGEs na geração de rotinas automatizadas para o HA, por meio de códigos em YAML. O estudo analisa quatro modelos amplamente utilizados da OpenAI [16] e do Google [7]. Em um segundo momento, o modelo com melhor desempenho será usado na geração de dados sintéticos/códigos de exemplo para o treinamento de um modelo local, com o objetivo de avaliar sua eficácia e compará-lo ao modelo online de referência. O artigo adota *prompts* com diferentes níveis de detalhamento e cenários de automação com distintos níveis de complexidade, visando investigar a precisão, coerência e aplicabilidade dos códigos gerados na plataforma HA. As principais contribuições deste trabalho são:

- Análise do desempenho de diferentes modelos amplamente utilizados na geração de códigos YAML;
- Treinamento de um modelo leve, local e específico para a geração de códigos YAML para automação no HA;
- Avaliação comparativa entre o modelo treinado localmente e um modelo online, considerando sua aplicabilidade prática.

In: Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia'2025). Rio de Janeiro, Brazil. Porto Alegre: Brazilian Computer Society, 2025.

ISSN 2966-2753

O artigo está organizado da seguinte forma: Seção 2 apresenta trabalhos relacionados; Seção 3, a metodologia; Seção 4, a discussão dos resultados; e Seção 5, conclusões e perspectivas futuras.

2 TRABALHOS RELACIONADOS

A crescente demanda por soluções que reduzam a necessidade de conhecimento técnico na configuração de sistemas inteligentes tem impulsionado o avanço da Inteligência Artificial (IA), em especial dos modelos de Linguagem de Grande Escala [3]. Nesse contexto, diversos estudos vêm explorando o uso da IA para a geração automática de código, com destaque para sua aplicação na automação residencial [5]. A capacidade desses modelos de converter comandos em linguagem natural em código funcional representa um potencial significativo para simplificar a interação com dispositivos inteligentes. Este cenário justifica a revisão sistemática realizada para esta seção, que buscou trabalhos através do Google Acadêmico, nas bases *ACM Digital Library*, *IEEE Xplore* e *arXiv*, utilizando expressões como "*Large Language Models*", "*LLMs*", "*code generation*", "*automation*". Os estudos foram selecionados com base na relevância para o tema de geração de código e automação, priorizando pesquisas com abordagem experimental.

No campo específico da automação residencial com comandos em linguagem natural, diversas abordagens têm sido propostas visando reduzir a curva de aprendizado necessária e possibilitar uma maior personalização de dispositivos e ambientes, sem a necessidade de conhecimentos técnicos avançados. Roffarello et al. [11], por exemplo, propuseram um agente baseado em voz para a criação de automações simples no ecossistema do Google, permitindo ao usuário tanto definir quanto modificar gatilhos e ações a partir de comandos falados. Uma abordagem semelhante foi adotada por Barricelli et al. [2], que desenvolveram um agente multimodal combinando voz e toque no ecossistema Alexa, ampliando as possibilidades ao permitir múltiplas ações para um único comando.

Giudici et al. [6] expandiram essa abordagem ao propor o GreenIFTTT, um agente conversacional baseado no modelo GPT-4, que permite que usuários criem automações no HA com foco em sustentabilidade. A proposta oferece uma interface baseada no próprio ChatGPT que permite aos usuários criarem automações no HA sem necessidade de codificação, o que reduz a barreira de utilização da tecnologia e estimula a adoção de soluções inteligentes com foco em eficiência energética. Além disso, o modelo incorpora estratégias para identificar padrões de uso e gerar recomendações de economia.

De maneira similar, Gallo et al. [5] desenvolveram o RuleBot++, um *chatbot* projetado para interpretar descrições de automações residenciais mais complexas, interagindo com o usuário durante o processo de criação. O sistema proposto utiliza validação semântica contínua para aprimorar a exatidão e a coerência das regras geradas, proporcionando uma experiência mais confiável e personalizada. O grande foco é a interação contínua com o usuário, que permite ajustar regras com base em *feedbacks* em tempo real, resultando em rotinas mais precisas e alinhadas com as expectativas do usuário que está conversando com o *chat*.

Ambos os estudos mostram potencial de se ter LGEs como camada valiosa entre o usuário e os sistemas de automação, oferecendo interfaces convidativas e eficazes para a criação de configurações personalizadas em casas inteligentes. Além disso, o Harmony proposto por Yin et al. [22], construído com o LLaMA3-8B, opera localmente, sem conexão com a nuvem, priorizando a privacidade e a capacidade de personalização do ambiente através da análise do contexto e sem necessidade de comandos explícitos, antecipando ações com base em padrões de uso e estado do ambiente.

Contudo, apesar dos avanços, ainda são escassos estudos que realizam comparações sistemáticas entre diferentes LGEs quanto à geração de código funcional e coerente. Moraes et al. [13] investigaram o uso de LGEs em um domínio de *domain-specific languages* (DSLs) para mídia e observaram frequentes erros sintáticos e semânticos, além de alucinações quando o problema exige sincronismo e estruturas específicas. Wang e Zhu [20], por exemplo, propuseram um método de validação semântica utilizando testes metamórficos com o objetivo de verificar se pequenas variações no prompt resultam em respostas consistentes. Essa abordagem tem-se mostrado promissora para avaliar a robustez de sistemas baseados em linguagem natural. Apesar de exemplos embutidos no *prompt* melhorarem a saída, os estudos destacam a necessidade de restringir o espaço de geração por meio de padrões de código, além de empregar validações e testes automáticos. Essas evidências, ainda que em um domínio distinto do HA, reforçam a necessidade da adoção de *prompts* estruturados para aumentar a confiabilidade dos códigos e reduzir o retrabalho nas automações geradas.

Dessa forma, o presente trabalho visa preencher uma lacuna importante ao propor uma avaliação experimental estruturada que considera não apenas a correção funcional das rotinas geradas, mas também a necessidade de ajustes e a complexidade dos códigos produzidos, fornecendo apoio para uma adoção segura e eficaz dos LGEs na automação residencial.

3 METODOLOGIA

A metodologia consistiu em uma avaliação experimental dividida em três fases, conforme a Figura 1. O objetivo foi comparar a capacidade de diferentes Modelos de Linguagem de Grande Escala (LGE) na geração de rotinas de automação residencial para a plataforma HA, a partir de *prompts* em linguagem natural.

A **Fase 1 (Instrução)** representa a etapa inicial de geração de dados. Neste estágio, os LGEs online foram preparados com uma instrução de sistema geral, que os contextualizou como especialistas em automação residencial para o HA. Em seguida, foram submetidos a uma série de *prompts* com diferentes níveis de complexidade para gerar códigos de automação em formato YAML. O resultado desta fase foi um banco de dados (*Códigos BD*) gerados por cada modelo avaliado, formando a base para a etapa de validação.

Na **Fase 2 (Validação)**, foi realizada a verificação funcional de cada código YAML gerado. Cada automação foi implementada e executada em uma instância do HA para testar sua correção e eficácia operacional. Os códigos que apresentaram erros e não funcionaram foram catalogados em um banco de dados de erros (*Erros BD*) e descartados. Por outro lado, os códigos executados com sucesso, sem a necessidade de ajustes, foram armazenados em um banco de dados de treinamento (*Treinamento BD*), junto com outros exemplos. Esta fase foi essencial para quantificar o desempenho de cada LGE e identificar o modelo com a maior taxa de sucesso (*Melhor Modelo*), que serviria de referência para a fase seguinte.

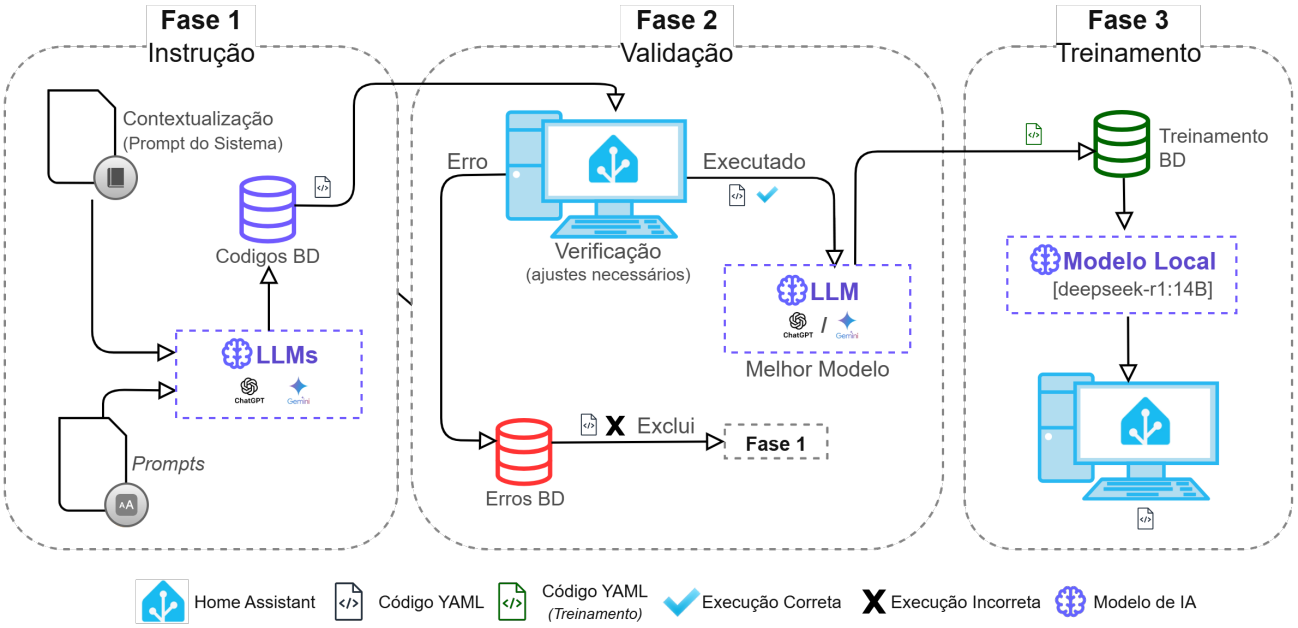

Figura 1: Fluxograma metodológico.

A **Fase 3 (Treinamento)** focou no desenvolvimento de uma alternativa local. Com o conjunto de códigos validados e funcionais (Treinamento BD), gerados pelo modelo de melhor desempenho na fase anterior, realizou-se o *fine-tuning* de um modelo local de código aberto (*DeepSeek-r1:14B*). O objetivo foi especializar o modelo na geração de códigos de automação. Ao final, o modelo local foi submetido aos mesmos *prompts* e cenários para avaliar sua capacidade e viabilidade de aplicação.

3.1 Seleção dos Modelos

Para este experimento foram selecionados quatro LGEs amplamente utilizados no estado da arte: ***GPT-4o, GPT-4.5, Gemini 2.5 Flash e Gemini 2.5 Pro***. Os modelos foram acessados através de suas interfaces oficiais, *OpenAI* [16] e *Google* [7], e executados a partir das mesmas instruções gerais para a geração de código YAML para rotinas de automação específicas para o Home Assistant (HA).

3.2 Contextualização dos Modelos

A fim de garantir consistência nos resultados e orientar os modelos na direção esperada, foi utilizada uma instrução geral (*prompt* de sistema) para contextualizar os LGEs e restringir sua atuação a um universo específico de aplicação: a geração de rotinas para automação residencial utilizando o HA. A instrução, fornecida como *prompt* inicial antes da apresentação dos comandos específicos, busca simular um cenário em que o modelo assume o papel de um especialista na área, sendo ela:

> "Você é um especialista em automação residencial e na criação de rotinas personalizadas para o HA. Sua principal função é analisar cuidadosamente as solicitações dos usuários e gerar automações completas em YAML, garantindo total compatibilidade com a plataforma.
> Todas as rotinas e automações que você criar devem ser funcionais e robustas (prontas para uso, sem erros de sintaxe ou lógica), bem estruturadas (organizadas, com comentários explicativos quando necessário, para facilitar a compreensão e futuras modificações), eficientes (projetadas para consumir recursos de forma otimizada) e integradas (utilizando apenas as entidades e *helpers* previamente definidas para simular os dispositivos e sensores reais).
> As automações devem se basear exclusivamente nas seguintes entidades e seus respectivos nomes declarados: Para Controle de Iluminação (Booleanos), utilize a Luz da Sala: luz_sala e a Luz do Corredor: luz_corredor. Para o Estado da Iluminação (Booleanos), use o Estado da Luz da Sala: luz_sala_estado e o Estado da Luz do Corredor: luz_corredor_estado. Para Sensor de Presença (Booleano), use o Sensor de Movimento do Corredor: sensor_mov_corredor. Para Sensores de Ambiente (Numéricos/Radiais), use o Sensor de Luminosidade do Corredor (0 lx a 1000 lx): sensor_luminos_corredor, o Sensor de Temperatura da Sala (15 °C a 35 °C): sensor_temp_sala e o Sensor de Temperatura da Cozinha (15 °C a 35 °C): sensor_temp_cozinha. Finalmente, para Controle de Climatização (Numérico/Radial), utilize o Simulador de Temperatura do

ar-condicionado da Sala (16 °C a 30 °C): temperatura _ar_sala".

Essa contextualização foi aplicada em todos os modelos testados, com o intuito de induzir comportamentos semelhantes na interpretação dos *prompts* passados, minimizando variações atribuídas a instruções genéricas ou interpretações divergentes.

3.3 Estrutura dos *Prompts*

Após a fase de contextualização dos modelos, a geração de rotinas foi orientada utilizando *prompts* cuidadosamente elaborados, seguindo uma estrutura de complexidade em três níveis: básico, intermediário e avançado. Cada um desses níveis continha duas variações distintas: uma versão simples e uma versão mais específica.

A ideia por trás do *prompt* simples é ser generalista, sem a necessidade de especificar detalhadamente os nomes das entidades envolvidas, o que permite uma interpretação mais ampla do modelo. Por outro lado, o *prompt* específico é mais detalhado, descrevendo não apenas o que a automação deve fazer, mas também fornecendo informações cruciais como os nomes exatos das entidades e, em alguns casos, valores ou condições específicas que seriam úteis na construção das automações. No total, essa metodologia resultou na definição de seis cenários de teste (*prompts*) distintos que foram aplicados em cada modelo, sendo eles:

(1) **Básico**
- **Simples:** *Faça uma automação para Ligar a luz da sala (Estado da Luz da Sala) ao pressionar um botão da Luz da Sala, e desligar a luz quando desligar o botão.*
- **Específico:** *Crie uma automação para Ligar a luz da sala (luz_sala _estado) quando o botão da Luz da Sala (luz_sala), muda o estado para 'on', e desligar a luz da sala (luz_sala _estado) quando desligar o botão (luz_sala).*

(2) **Intermediário**
- **Simples:** *Faça uma automação para que, quando estiver escuro, a luz do corredor acenda se existir algum movimento no corredor e depois de alguns segundos apague automaticamente.*
- **Específico:** *Crie uma automação para que, quando a luminosidade do sensor de luminosidade do corredor (sensor_luminos_corredor) estiver baixa e quando o sensor de movimento do corredor (sensor_mov_corredor) estiver ativo, a luz do corredor (luz_corredor) seja ligada. A ação deve ligar a luz, esperar por 10 segundos, e então desligar.*

(3) **Avançado**
- **Simples:** *Faça uma automação para ligar o ar- condicionado da sala usando a média de temperatura dos sensores de temperatura da sala e da cozinha. Se a média entre os sensores for maior que* 25 °C *ajuste o ar-condicionado para* 22 °C*, se for menor que* 21 °C *ajuste para* 24 °C*. A cada 10 minutos calcule a média da temperatura dos sensores, e repita o processo.*
- **Específico:** *Crie uma automação para ligar o ar- condicionado da sala (temperatura_ar_sala) e ajustar sua temperatura usando a média de temperatura entre o sensor de temperatura da sala (sensor_temp_sala) e o sensor de temperatura da cozinha (sensor_temp_cozinha). Se a média for maior que* 25 °C *ajuste a temperatura do ar-condicionado da sala para* 22 °C*, se for menor que* 21 °C *ajuste para* 24 °C*.*

3.4 Execução e Validação Prática no Home Assistant

Após a contextualização e preparos, as automações foram geradas pelos modelos selecionados. Em seguida, realizou-se a validação funcional para verificar se os códigos YAML produzidos eram executáveis. Os arquivos foram testados em uma instância do Home Assistant (HA) (versão 2025.6.3), executada localmente no mesmo notebook usado para gerar os códigos, em máquina virtual no Oracle VirtualBox [17], com Oracle Linux (64-bit), 2048 MB de RAM e dois núcleos de processamento. A controladora de vídeo foi VMSVGA, com 16 MB de memória de vídeo, 20 GB de armazenamento e imagem do Home Assistant (HA) com 32 GB adicionais. A rede usou adaptador Intel(R) PRO/1000 MT Desktop em modo bridge, garantindo conectividade para os serviços da plataforma.

3.4.1 Configuração do Home Assistant. Na plataforma, foi necessário configurar algumas entidades [8], etapa fundamental para a validação prática das rotinas geradas. Como apontado por Giudici et al.[6], a simulação desses dispositivos permite testar o comportamento dinâmico das rotinas e verificar a aderência entre os elementos descritos nos *prompts* e o ambiente operacional de destino, contribuindo para uma avaliação mais rigorosa da funcionalidade, robustez e aplicabilidade dos códigos gerados.

Nesse sentido, para garantir a funcionalidade das rotinas produzidas, foram criadas entidades simuladas, representando todos os sensores, atuadores e demais dispositivos mencionados nos *prompts*. Elas foram definidas manualmente também por meio de códigos YAML, utilizando a própria interface de desenvolvimento do HA, possibilitando a avaliação do comportamento das automações em diferentes condições de interação entre dispositivos. Ao final, obtivemos um *dashboard* central de gerenciamento e análise do comportamento dessas entidades, conforme ilustrado na Figura 2.

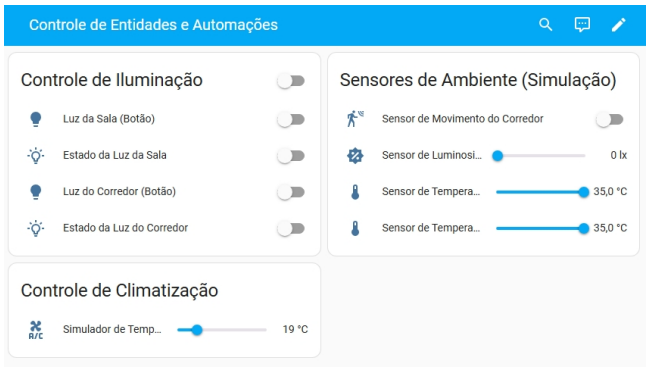

Figura 2: Painel de gestão do Home Assistant exibindo as entidades desenvolvidas para simulação e controle de automações residenciais.

Após a configuração das entidades no ambiente virtual, iniciaram-se os testes práticos, consistindo na execução, no HA, dos arquivos YAML gerados pelos LGEs. Essa etapa foi essencial para verificar

se os modelos avaliados não apenas produziam códigos semanticamente coerentes, mas também automações funcionais e compatíveis com os requisitos da plataforma, permitindo observar o comportamento das entidades conforme cenários de automação.

4 AVALIAÇÃO

A avaliação do estudo consistiu na verificação funcional dos códigos YAML gerados pelos modelos analisados. Para isso, estabelecemos um ambiente controlado utilizando a plataforma HA, onde as rotinas automatizadas foram testadas. Além disso, realizamos uma análise da estrutura dos códigos gerados, observando aspectos de clareza, eficiência e complexidade, e análises estatísticas para avaliar quantitativamente o desempenho dos diferentes modelos, verificando quantas automações foram executadas com sucesso, realizando comparações entre os resultados obtidos para cada cenário e nível de complexidade estabelecido na metodologia.

4.1 Análise Funcional e Fatorial

A análise funcional consistiu na execução direta de cada automação gerada, observando seu comportamento no ambiente com entidades simuladas que replicavam as condições descritas nos *prompts*. Esse procedimento não apenas confirmava a validade sintática do código, mas também permitia uma avaliação quantitativa de sua eficácia operacional. A performance dos modelos na geração de códigos foi avaliada segundo dois critérios objetivos e quantificáveis, essenciais para uma análise robusta dos resultados:

(1) **Correção Funcional**: Este critério binário (Sim/Não) verificou se o código YAML gerado pelo LGE funcionou corretamente logo na primeira execução no ambiente simulado do HA. Um resultado "Sim"indica a capacidade do modelo de produzir automações prontas para uso, minimizando a necessidade de intervenção humana inicial;
(2) **Necessidade de Ajuste**: Já este critério, também binário (Sim/Não), mede o esforço de refinamento após a geração e a proximidade do código com uma solução final implementável. A análise desse esforço incluiu não apenas a identificação de linhas para correção, seja no código ou dentro da plataforma do HA, mas também a avaliação de interações extras com os modelos online (como novos *prompts* solicitados) necessárias para obter um resultado funcional e otimizado.

4.1.1 ***Análise Fatorial.*** Para aprofundar a compreensão de como esses critérios foram influenciados pelas condições do teste, foi realizada uma análise sistemática baseada em um planejamento fatorial 2^k, uma ferramenta poderosa para investigar o efeito de múltiplos fatores em um experimento [12]. Especificamente, utilizamos um planejamento fatorial 2^3, o que significa que foram investigados três fatores principais, cada um em dois níveis.

Esta abordagem focou em três fatores críticos: a Complexidade do Cenário, o Tipo de *Prompt* e a Necessidade de um Ajuste Simples. A análise foi conduzida para cada um dos quatro modelos (*GPT-4o, GPT-4.5, Gemini 2.5 Flash e Gemini 2.5 Pro*), considerando os seguintes fatores e níveis:

(1) Fator A – Complexidade do Cenário:
   - Nível Baixo (−1): Intermediário
   - Nível Alto (+1): Avançado
(2) Fator B – Tipo de *Prompt*:
   - Nível Baixo (−1): Simples
   - Nível Alto (+1): Específico
(3) Fator C – Necessidade de Ajuste:
   - Nível Baixo (−1): Não
   - Nível Alto (+1): Sim

A principal variável de resposta analisada foi o sucesso da automação (Funcionou = s, Falhou = n).

4.1.2 Análise dos Efeitos Principais. O efeito principal da Complexidade (Fator A) demonstrou ser o mais impactante, mostrando uma forte correlação negativa com o sucesso. A transição do cenário intermediário (uma vez que todos alcançaram êxito no cenário básico) para o avançado, aumentou drasticamente a probabilidade de falha para a maioria dos modelos.

O efeito do Tipo de *prompt* (Fator B) foi, em geral, positivo, indicando que a mudança de um *prompt* simples para um específico tende a aumentar a taxa de sucesso. O efeito da Necessidade de Ajuste (Fator C) também se mostrou relevante; cenários que exigiram ajuste (nível +1) estiveram mais associados a falhas, sugerindo que a necessidade de intervenção é um forte indicador de que o modelo não gerou um código satisfatório.

Tabela 1: Tabela de Resultados para o Modelo GPT-4o.

Cenário	Prompt	Código Funcional	Com Ajuste
Básico	Simples	s	n
Básico	Específico	s	n
Intermediário	Simples	s	s
Intermediário	Específico	s	n
Avançado	Simples	n	s
Avançado	Específico	n	s

Tabela 2: Tabela de Resultados para o Modelo GPT-4.5.

Cenário	Prompt	Código Funcional	Com Ajuste
Básico	Simples	s	n
Básico	Específico	s	n
Intermediário	Simples	n	s
Intermediário	Específico	s	n
Avançado	Simples	n	s
Avançado	Específico	n	s

4.1.3 Interações. A análise das interações entre os fatores revelou alguns *insights* do desempenho de cada modelo:

- **Interação AB (Complexidade-*Prompt*)**: É a interação mais significativa. Para modelos como *GPT-4.5* e *Gemini 2.5 Flash*, um *prompt* específico (Fator B) gera melhor resultado em cenários mais complexos (Fator A). No *Gemini Flash*, por exemplo, a especificidade do *prompt* garantiu sucesso no cenário avançado, mostrando que descrições precisas podem compensar o aumento da dificuldade.

Tabela 3: Tabela de Resultados para o Modelo Gemini 2.5 Flash.

Cenário	Prompt	Código Funcional	Com Ajuste
Básico	Simples	s	n
Básico	Específico	s	n
Intermediário	Simples	s	s
Intermediário	Específico	s	n
Avançado	Simples	n	s
Avançado	Específico	s	s

Tabela 4: Tabela de Resultados para o Modelo Gemini 2.5 Pro.

Cenário	Prompt	Código Funcional	Com Ajuste
Básico	Simples	s	n
Básico	Específico	s	n
Intermediário	Simples	s	n
Intermediário	Específico	s	n
Avançado	Simples	s	n
Avançado	Específico	s	n

- **Interações com o Ajuste (AC e BC):** A necessidade de ajuste (Fator C) interage fortemente com os outros dois fatores. Um *prompt* específico (B=+1) consistentemente reduziu a necessidade de ajuste (C=-1) em cenários de nível intermediário, mostrando a interação BC. Da mesma forma, o aumento da complexidade (A=+1) eventualmente elevou a necessidade de ajuste (C=+1), evidenciando a interação AC. Isto indica que a necessidade de ajuste não é um fator isolado, mas sim uma consequência da combinação dos outros desafios impostos ao modelo.
- **Interações neutras:** *O Gemini 2.5 Pro* representa um caso à parte, pois neutraliza os efeitos dos fatores. Como o Gemini 2.5 Pro obteve sucesso sem necessidade de ajuste em todas as combinações testadas, o Fator C (Ajuste) permaneceu sempre no seu nível baixo (-1). Consequentemente, as interações tornam-se irrelevantes, não por uma falha na análise, mas porque a sua performance não apresenta variações de dificuldade e qualidade do *prompt*, operando consistentemente em qualquer estado.

Com isso, o delineamento fatorial 2^3 demonstra que o sucesso da automação é multifatorial. Embora a complexidade seja o fator dominante, sua influência depende da qualidade do *prompt* e da capacidade do modelo de gerar código sem ajustes. A análise das interações revelou que um *prompt* específico pode mitigar ajustes, aumentando a probabilidade de sucesso.

4.1.4 Tipo de Prompt. Conforme vimos anteriormente, a especificidade é um fator crucial para o sucesso da automação. Independentemente do modelo em questão, *prompts* mais detalhados e bem-estruturados apresentam menor probabilidade de erro e, de forma geral, funcionam melhor na primeira tentativa.

Esta relação direta entre detalhe e eficácia sugere que investir tempo na elaboração de um bom *prompt* é uma prática fundamental para otimizar os resultados em qualquer sistema de IA.

No entanto, a qualidade do *prompt* por si só não é o único fator determinante. Embora uma maior descrição possa corrigir a rota de um modelo em cenários de média complexidade, como foi o caso do *GPT-4.5*, ela não é suficiente para superar as limitações fundamentais em cenários avançados onde os modelos da *OpenAI* falharam independentemente da qualidade do *input*.

Neste contexto, os modelos da família *Gemini* demonstraram um desempenho superior. O *Gemini 2.5 Pro*, em particular, provou ser excepcionalmente mais robusto, alcançando sucesso em todas as combinações de complexidade e tipo de *prompt*. A sua capacidade de funcionar corretamente mesmo com descrições mais simples em cenários avançados, onde outros modelos falharam, destaca a sua arquitetura e treinamento mais avançados e confiáveis. Assim, embora *prompts* mais específicos melhorem o desempenho de forma geral, a escolha de um modelo mais capaz, como o *Gemini*, é essencial para garantir o sucesso em aplicações críticas e complexas.

4.1.5 Complexidade dos cenários. A análise comparativa dos modelos em relação à complexidade dos cenários, evidencia que a força de um modelo não pode ser medida apenas em situações simples. Em tarefas de nível básico, todos os modelos alcançaram 100% de sucesso. Esse resultado indica que, para tarefas mais fáceis, eles funcionam de forma parecida e podem ser trocados um pelo outro.

No entanto, quando passamos para os cenários de complexidade intermediária, as primeiras diferenças importantes de desempenho começam a aparecer. O modelo *GPT-4.5* apresenta redução de desempenho, e sua taxa de sucesso reduz para 50%. Esse ponto de virada sugere que um aumento moderado na dificuldade já é suficiente para mostrar as limitações de algumas arquiteturas.

A diferença fica ainda mais nítida em cenários de alta complexidade. Nesse caso, o *Gemini 2.5 Pro* manteve uma taxa de sucesso de 100%, o que demonstra maior consistência e confiabilidade em cenários de maior complexidade. O *Gemini 2.5 Flash*, por sua vez, mostra um sucesso parcial, com 50%, posicionando-se como uma solução intermediária. Por outro lado, modelos da família GPT não se saíram bem nas tarefas avançadas, destacando uma limitação crítica na capacidade deles de lidar com problemas mais gerais e complexos.

4.2 Avaliação por ANOVA

Com o objetivo de validar estatisticamente as observações e aprofundar a análise dos resultados, foi empregada a Análise de Variância (ANOVA) de dois fatores. A ANOVA é uma técnica estatística que compara a variância entre as médias de diferentes grupos com a variância dentro desses mesmos grupos. Se a variância entre os grupos for significativamente maior do que a variância interna, conclui-se que existe uma diferença estatisticamente significativa entre as médias dos grupos [12].

No presente estudo, a ANOVA avaliou o impacto de dois fatores independentes sobre a taxa de sucesso (variável dependente) na geração de códigos YAML:

(1) Modelo: O modelo de LGEs utilizado (*GPT-4, GPT-4.5, Gemini 2.5 Flash, Gemini 2.5 Pro*);

(2) Cenário: O nível de complexidade da automação solicitada (Básico, Intermediário, Avançado).

O objetivo foi testar os efeitos principais de cada fator (Modelo e Cenário), bem como o efeito de sua interação, a fim de determinar se a escolha do LGE e a complexidade da tarefa têm um impacto significativo na taxa de sucesso das automações. Para tanto, o modelo de ANOVA foi aplicado diretamente aos dados brutos de cada ensaio, onde a variável dependente assumiu valores binários (1 para sucesso, 0 para falha). Dessa forma, a análise avaliou a probabilidade de sucesso sem a necessidade de uma agregação prévia dos resultados, estimando o impacto dos fatores a partir das observações individuais.

Tabela 5: ANOVA - Taxa de Sucesso por Modelo e Cenário

Fonte de Variação	Soma dos Quadrados (sum_sq)	Graus de Liberdade (df)	Estatística F	Valor-p (PR(>F))
C(Modelo)	0.8333	3	3.3333	0.0562
C(Cenário)	1.7500	2	10.5000	**0.0023**
Residual	1.0000	12	-	-

A análise dos valores apresentados na Tabela 5 permite extrair algumas conclusões importantes. O fator Cenário obteve um valor-p de 0.0023, que é inferior ao nível de significância padrão ($\alpha = 0.05$), indicando um efeito significativo. Com isso, podemos afirmar que a complexidade da automação solicitada tem um impacto direto e determinante na probabilidade de sucesso da geração de código.

Além disso, o fator Modelo apresentou um valor-p de 0.0562, o que sugere que o desempenho do código gerado tende a variar um pouco entre os diferentes modelos de LGE, com a diferença observada nas taxas de sucesso estando no limite de ser considerada uma variação importante, e não apenas aleatória.

Os 12 graus de liberdade residuais (df = 12) na tabela de ANOVA indicam que houve repetições para as combinações de tratamento no experimento. Considerando as 12 combinações possíveis entre os 4 modelos e os 3 cenários, os graus de liberdade residuais confirmam que cada combinação foi testada mais de uma vez, permitindo uma análise robusta da variação dentro de cada grupo.

Com isso, podemos concluir que a análise estatística via ANOVA confirma as observações iniciais, mostrando que a complexidade da tarefa é um dos fatores que mais determina o sucesso do código, enquanto a escolha do modelo se mostra um fator relevante com forte tendência a influenciar no resultado final.

4.3 Análise Qualitativa dos Erros

Além da análise quantitativa de sucesso, foi realizada uma análise qualitativa para classificar os tipos de erros encontrados nos códigos que falharam ou que necessitaram de algum ajuste para funcionar. O objetivo foi identificar padrões de falha específicos de cada modelo, a fim de compreender melhor suas limitações. Os erros foram categorizados da seguinte forma:

(1) **Erro de Sintaxe**: O código gerado continha erros que violavam as regras da linguagem YAML, como indentação incorreta, uso indevido de hifens, dois-pontos ou aspas;
(2) **Erro de Lógica**: automação sintaticamente válida que não cumpre o comportamento descrito no *prompt*, por *condition* mal formulada ou *action* que não produz o efeito esperado;
(3) **Erro de Entidade**: O modelo produziu automações nomes de entidades que não existiam ou que eram inconsistentes com as definidas no *prompt* de sistema.

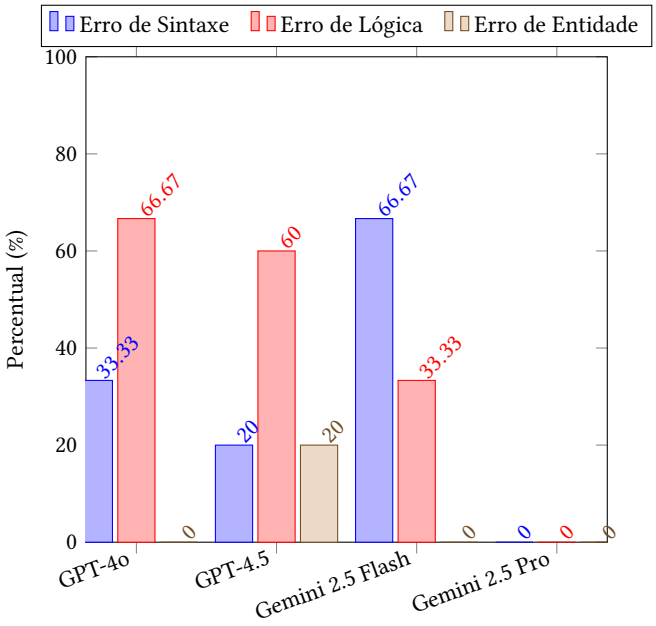

Figura 3: Distribuição dos tipos de erro por modelo.

A Figura 3 mostra que o Erro de Lógica foi o tipo de falha mais frequente, seguido pelo Erro de Sintaxe. Isso sugere que, embora a interpretação correta da intenção do usuário seja o maior desafio, a geração de código sintaticamente válido ainda não é garantida por todos os modelos. Além disso, um código pode conter mais de um tipo de erro. O modelo *GPT-4.5*, por exemplo, apresentou tanto um erro de sintaxe (falta de hífen) quanto um erro de entidade (geração de um nome de entidade inexistente).

Os modelos da família GPT (*GPT-4o e GPT-4.5*) foram os que mais cometeram erros. Já os erros do *Gemini 2.5 Flash* foram predominantemente de Sintaxe, e o único modelo a não cometer nenhum erro foi o *Gemini 2.5 Pro*, reforçando sua confiabilidade, conforme observamos nas análises anteriores.

4.4 Análise do Código

Ao realizar a análise dos resultados e dos códigos gerados, um aspecto qualitativo observado durante os testes foi a diferença no estilo do código gerado pelos diferentes modelos de LGE. Observamos que, principalmente nos modelos da família *Gemini*, houve a inclusão de comentários nos códigos gerados. Esses comentários geralmente explicavam a função de cada seção da automação, como o gatilho (*trigger*), as condições (*condition*) e as ações (*action*).

Este comportamento apresenta uma dualidade interessante que merece ser destacada:

(1) **Ponto de Vista da Eficiência**: Em um ambiente de produção, onde a concisão e a otimização são frequentemente priorizadas, a inclusão de comentários pode ser vista como um aumento desnecessário do número de linhas. O código

se torna mais longo, o que, em arquivos de configuração muito extensos, poderia acabar impactando a legibilidade e a manutenção para um desenvolvedor experiente que já compreende a estrutura do YAML do HA, por exemplo.

(2) **Ponto de Vista Educacional e de Depuração**: Para usuários iniciantes ou intermediários, com pouca ou nenhuma proficiência técnica (público-alvo deste estudo), ou mesmo especialistas analisando automações complexas, esses comentários são um recurso valioso. Funcionam como documentação embutida, explicando a lógica do código, acelerando o aprendizado e facilitando a identificação rápida de trechos que precisam de correção.

Sendo assim, os modelos da família *Gemini* tenderam a adotar uma abordagem didática, gerando um código que não apenas funciona, mas que também ensina. A inclusão de comentários parece ser uma característica intrínseca do seu processo de geração de códigos, priorizando a clareza e a compreensão do usuário, tornando-se um diferencial competitivo significativo em contextos educacionais, de suporte e para usuários que estão desenvolvendo suas habilidades em automação residencial. Em contrapartida, os modelos da família *GPT*, em geral, produziram um código mais direto, com o foco estritamente na entrega da solução funcional, sem a adição de muitos comentários, se alinhando mais com o que um desenvolvedor experiente escreveria manualmente.

5 DESENVOLVIMENTO DO MODELO LOCAL

Com base na análise e seleção do modelo online de melhor desempenho, o *Gemini 2.5 Pro*, o projeto avançou para a fase de desenvolvimento de um modelo de Linguagem de Pequena Escala Local (LPE). O objetivo é criar uma alternativa que possa gerar rotinas e códigos de automação de forma similar, com potencial para maior controle sobre privacidade e latência, e customização para ambientes domésticos inteligentes.

Para isso, selecionamos o modelo DeepSeek-r1, que se destaca por sua arquitetura otimizada para geração de código e se assemelha em desempenho a modelos de maior porte, como o Gemini Pro, em tarefas de raciocínio lógico e criativo [14]. O modelo foi obtido e inicializado localmente através do *Ollama*, uma plataforma de código aberto que facilita a execução de modelos de LPE diretamente no computador, via terminal [15].

5.1 Treinamento e validação do modelo local

Após a seleção do modelo, foi realizado um treinamento utilizando a abordagem de *fine-tuning*, processo em que um modelo que já foi extensivamente treinado em um grande volume de dados gerais é ajustado para uma tarefa mais específica ou um conjunto de dados particular [21]. Esse treinamento consiste em utilizar um pequeno grupo de 20 exemplos específicos de automação residencial gerados pelo modelo *Gemini 2.5 Pro*, previamente definido como referência devido ao seu alto desempenho nos testes realizados. Os exemplos utilizados durante o processo foram cuidadosamente escolhidos, contemplando as mesmas entidades pré-definidas para garantir consistência e coerência no aprendizado do modelo local. Além disso, foi adotado um *prompt* de sistema detalhado para orientar o processo de treinamento, visando melhorar a capacidade do modelo local em interpretar os *prompts*, assim como foi feito nos modelos online, e transformá-los em rotinas de automação funcionais.

Após realizar o treinamento, o modelo *DeepSeek-r1:14B* foi submetido aos mesmos testes de *prompt* e complexidade de cenários utilizados na avaliação dos modelos online. Os resultados indicam que o modelo local consegue gerar códigos de automações até o nível intermediário, mesmo realizando o processo de *fine-tuning*. Adicionalmente, os códigos gerados ainda necessitaram de ajustes manuais significativos dentro da plataforma do HA. Finalmente, podemos notar um longo período de tempo para que o modelo proveja a resposta.

Tabela 6: Tabela de Resultados para o Modelo Local.

Cenário	Prompt	Código Funcional	Com Ajuste
Básico	Simples	s	s
Básico	Específico	s	s
Intermediário	Simples	n	s
Intermediário	Específico	s	s
Avançado	Simples	n	s
Avançado	Específico	n	n

6 CONCLUSÕES E TRABALHOS FUTUROS

Este artigo apresentou uma avaliação experimental da capacidade de diferentes Linguagem de Grande Escala na geração de códigos YAML para automação residencial no Home Assistant. Os resultados indicam que a complexidade dos cenários e a especificidade dos *prompts* influenciam diretamente o sucesso, sendo a complexidade o fator principal. A análise fatorial 2^3 mostrou que *prompts* mais específicos reduzem ajustes e aumentam a taxa de sucesso.

Nos experimentos com modelos online (*GPT-4, GPT-4.5, Gemini 2.5 Flash, Gemini 2.5 Pro*), todos tiveram bom desempenho em tarefas simples, destacando-se o *Gemini 2.5 Pro* por manter alta taxa de sucesso mesmo com *prompts* básicos. A ANOVA confirmou a influência da complexidade e da escolha do modelo, enquanto a análise de erros apontou falhas de lógica como as mais recorrentes. Já o modelo local (*DeepSeek-r1:14B*), ajustado com dados do *Gemini 2.5 Pro*, mostrou-se funcional, porém demandou ajustes manuais e maior tempo de resposta, indicando necessidade de mais dados e otimização para competir com modelos online.

Como trabalhos futuros, propõe-se avaliar outros modelos locais de código aberto (*7B, 13B, 34B*) sem *fine-tuning* inicial; treinar o *DeepSeek-r1:14B* (ou variantes mais leves) com maior e mais diversa base de automações; e expandir o estudo para domínios correlatos, como geração de cenários para simulação de redes, exigindo não apenas correção sintática, mas coerência lógica, ampliando a validação em contextos técnicos complexos.

AGRADECIMENTOS

Gostaríamos de agradecer ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), ao Comitê Gestor da Internet no Brasil (CGI.br), à Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) e aos auxílios 2018/23097-3, 2020/05182-3, TIC 002/2015.

REFERÊNCIAS

[1] Home Assistant. 2024. Documentação - home-assistant.io. https://www.home-assistant.io/docs/. Acessado em: jun. 2025.

[2] Barbara Rita Barricelli, Daniela Fogli, Letizia Iemmolo, and Angela Locoro. 2022. A Multi-Modal Approach to Creating Routines for Smart Speakers. In *Proceedings of the 2022 International Conference on Advanced Visual Interfaces* (Frascati, Rome, Italy) *(AVI '22)*. Association for Computing Machinery, New York, NY, USA, Article 37, 5 pages. https://doi.org/10.1145/3531073.3531168

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Shabnam FakhrHosseini, Chaiwoo Lee, Sheng-Hung Lee, and Joseph Coughlin. 2025. A taxonomy of home automation: expert perspectives on the future of smarter homes. *Information Systems Frontiers* 27, 2 (2025), 449–466.

[5] Simone Gallo, Fabio Paternò, and Alessio Malizia. 2024. A conversational agent for creating automations exploiting large language models. *Personal and Ubiquitous Computing* 28, 6 (2024), 931–946.

[6] Mathyas Giudici, Luca Padalino, Giovanni Paolino, Ilaria Paratici, Alexandru Ionut Pascu, and Franca Garzotto. 2024. Designing home automation routines using an LLM-based chatbot. *Designs* 8, 3 (2024), 43.

[7] Google DeepMind. 2025. Gemini. https://deepmind.google/technologies/gemini/. Acessado em: jun. 2025.

[8] Home Assistant. 2025. Conceitos e terminologia. https://www.home-assistant.io/getting-started/concepts-terminology/. Acessado em: jun. 2025.

[9] Mrs M KALPANA. 2025. Survey and analysis of home automation system encompassing embedded systems, the Internet of Things (IoT) and AI algorithms. *Vidhyayana-An International Multidisciplinary Peer-Reviewed E-Journal-ISSN 2454-8596* 10, si4 (2025), 449–466.

[10] Evan King, Haoxiang Yu, Sangsu Lee, and Christine Julien. 2024. Sasha: creative goal-oriented reasoning in smart homes with large language models. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 1 (2024), 1–38.

[11] Alberto Monge Roffarello and Luigi De Russis. 2023. Defining Trigger-Action Rules via Voice: A Novel Approach for End-User Development in the IoT. In *End-User Development*, Lucio Davide Spano, Albrecht Schmidt, Carmen Santoro, and Simone Stumpf (Eds.). Springer Nature Switzerland, Cham, 65–83.

[12] Douglas C Montgomery. 2017. *Design and analysis of experiments*. John wiley & sons, Hoboken, NJ.

[13] Daniel Moraes, Polyana da Costa, Antonio Busson, José Boaro, Carlos Neto, and Sergio Colcher. 2023. On the Challenges of Using Large Language Models for NCL Code Generation. In *Anais Estendidos do XXIX Simpósio Brasileiro de Sistemas Multimídia e Web* (Ribeirão Preto/SP). SBC, Porto Alegre, RS, Brasil, 151–156. https://doi.org/10.5753/webmedia_estendido.2023.236175

[14] Ollama. 2025. DeepSeek-r1:14B. https://ollama.com/library/deepseek-r1:14b. Acessado em: jun. 2025.

[15] Ollama. 2025. Ollama. https://ollama.com/. Acessado em: jun. 2025.

[16] OpenAI. 2025. ChatGPT. https://chat.openai.com. Acessado em: jun. 2025.

[17] Oracle. 2024. Oracle VM VirtualBox. https://www.virtualbox.org/. Acessado em: jun. 2025.

[18] Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matthew Jones, Alessandro Morari, and Ruchir Puri. 2023. Automated Code generation for Information Technology Tasks in YAML through Large Language Models. arXiv:2305.02783 [cs.SE] https://arxiv.org/abs/2305.02783

[19] Mohaimenul Azam Khan Raiaan, Md Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. 2024. A review on large language models: Architectures, applications, taxonomies, open issues and challenges. *IEEE Access* 12 (2024), 26839–26874.

[20] Xiaoyin Wang and Dakai Zhu. 2024. Validating LLM-Generated Programs with Metamorphic Prompt Testing. arXiv:2406.06864 [cs.SE] https://arxiv.org/abs/2406.06864

[21] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned Language Models Are Zero-Shot Learners. arXiv:2109.01652 [cs.CL] https://arxiv.org/abs/2109.01652

[22] Ziqi Yin, Mingxin Zhang, and Daisuke Kawahara. 2024. Harmony: A Home Agent for Responsive Management and Action Optimization with a Locally Deployed Large Language Model. arXiv:2410.14252 [cs.HC] https://arxiv.org/abs/2410.14252