INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Geração Inteligente de Interfaces Gráficas para a Web

MIGUEL LEÇA DE OLIVEIRA ALMEIDA
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Geração Inteligente de Interfaces Gráficas para a *Web*

Miguel Leça de Oliveira Almeida

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de *Software*

Orientador: Dulce Mota

Supervisor Externo: João Martins

Porto, junho 2025

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim. As exceções estão explicitamente reconhecidas na secção "Considerações éticas" do primeiro capítulo. Esta secção também declara como as ferramentas de IA foram utilizadas e para que finalidade.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P. PORTO.

ISEP, Porto, 29 de junho de 2025

Resumo

A geração automática de interfaces gráficas para aplicações *web* tem vindo a afirmar-se como uma abordagem inovadora para aumentar a eficiência, a uniformidade e a qualidade no desenvolvimento de *software*. Este trabalho apresenta o desenvolvimento, implementação e validação de um sistema inteligente capaz de gerar código de *frontend* de forma automatizada a partir de mockups visuais, integrando modelos de *machine learning* e *pipelines* multimodais.

O processo de investigação partiu de uma revisão sistemática da literatura, com recurso à metodologia *PRISMA*, e da análise comparativa de abordagens existentes. A solução desenvolvida assenta num pipeline multimodal que combina um modelo de visão computacional, para interpretar os *mockups*, com um modelo de linguagem de grande escala *Large Language Model (LLM)* adaptado através de *fine-tuning* para gerar o código final.

A avaliação recorreu a um pipeline de testes automatizado e a métricas objetivas de qualidade, desempenho e produtividade, permitindo comprovar a viabilidade da automação inteligente na geração de interfaces. Os resultados obtidos evidenciam o potencial para reduzir o esforço manual, aumentar a consistência visual e acelerar o ciclo de desenvolvimento. Assim, a investigação realizada estabelece uma base sólida para futuras evoluções e para a industrialização desta abordagem em contextos organizacionais.

Palavras-chave: Geração Automática de Interfaces, *Machine Learning*, *Angular*, *Frontend*, Automação, *Mockups*.

Abstract

The automatic generation of graphical interfaces for web applications has emerged as an innovative approach to increasing efficiency, uniformity, and quality in software development. This work presents the development, implementation, and validation of an intelligent system capable of automatically generating frontend code from visual mockups, integrating machine learning models and multimodal pipelines.

The research process began with a systematic literature review, following the PRISMA methodology, and a comparative analysis of existing approaches. The developed solution is based on a multimodal pipeline that combines a computer vision model to interpret mockups with a LLM adapted via fine-tuning to generate the final code.

The evaluation employed an automated testing pipeline and objective metrics for quality, performance, and productivity, demonstrating the feasibility of intelligent automation in interface generation. The results highlight the potential to reduce manual effort, increase visual consistency, and accelerate the development lifecycle. Thus, this research establishes a solid foundation for future advancements and the industrialization of this approach in organizational contexts.

Keywords: Automatic Interface Generation, *Machine Learning*, *Angular*, *Frontend*, Automation, *Mockups*.

Agradecimentos

Gostaria de expressar os meus sinceros agradecimentos a todas as pessoas que contribuíram para o sucesso deste relatório. A realização deste projeto foi uma jornada desafiadora, mas o apoio e a colaboração de todos foram cruciais para alcançar os resultados obtidos.

Em primeiro lugar, gostaria de agradecer à minha família e à minha namorada pelo seu apoio incondicional ao longo de todo o processo. As palavras de encorajamento, o incentivo e a compreensão demonstrados foram fundamentais para manter a minha motivação e determinação durante os momentos mais exigentes.

Gostaria de expressar a minha profunda gratidão ao meu supervisor na empresa, João Martins, pela sua notável dedicação e empenho no auxílio prestado ao longo deste projeto. A sua abordagem proativa e disponibilidade constante para esclarecer dúvidas foram de extrema importância para a conclusão bem-sucedida deste relatório.

Também gostaria de agradecer à minha orientadora do ISEP, Dulce Mota, pelo seu suporte e orientação valiosa. As suas sugestões e conselhos especializados contribuíram significativamente para a qualidade do trabalho realizado.

Por fim, quero expressar a minha gratidão à empresa Sysnovare pela oportunidade de desenvolver este projeto e pelo contínuo apoio e simpatia desde o seu início. Agradeço à equipa pela disponibilidade para discussões e pela colaboração, que foram fundamentais para a realização deste relatório.

Não poderia, porém, deixar de prestar uma sentida homenagem ao meu avô, que infelizmente partiu durante a elaboração deste trabalho. Em todas as minhas visitas ao hospital para o ver, fazia questão de acompanhar a evolução da minha dissertação, perguntando sempre como estava a correr e transmitindo um enorme orgulho em cada conquista. Tenho a certeza de que, onde quer que esteja, continuará a torcer por mim e constituiu uma enorme força para concluir este trabalho.

A todos os mencionados e a todos aqueles que, de alguma forma, contribuíram para este trabalho, o meu mais sincero agradecimento. O vosso apoio foi imprescindível para o sucesso deste projeto.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Definições

Adobe XD	*Software* da Adobe para design e prototipagem de interfaces de utilizador.
Angular	Framework para desenvolvimento de aplicações *web*, mantido pela Google, que utiliza *TypeScript*.
Angular Material	Biblioteca de componentes *UI* para aplicações *Angular* que segue as diretrizes do Material Design.
API	Application Programming Interface – Conjunto de rotinas e padrões estabelecidos para acesso a funcionalidades ou dados de um serviço ou aplicação.
AST	Abstract Syntax Trees – Representação em árvore da estrutura sintática de código fonte usada para análise e processamento.
AutoML	Automated Machine Learning – Tecnologia que automatiza o processo de criação, treino e otimização de modelos de *machine learning*.
AWS	Amazon *Web* Services – Plataforma de serviços de computação em nuvem oferecida pela Amazon.
Backend	Parte de uma aplicação que é executada no servidor, responsável pela lógica de negócio, armazenamento e processamento de dados.
Balsamiq	Ferramenta para criação rápida de wireframes e protótipos de interfaces.
Benchmark	Conjunto de testes padronizados utilizados para avaliar o desempenho de modelos ou sistemas.
binário	Ficheiro executável que contém código máquina diretamente interpretado pelo sistema operativo.
BLEU	Bilingual Evaluation Understudy – Métrica para avaliação automática da qualidade de texto gerado comparando com referências humanas.
bounding boxes	Caixas delimitadoras que definem a área de interesse em imagens, geralmente usadas para identificar objetos ou texto.
browser	Programa utilizado para aceder e visualizar páginas e aplicações *web*.

Caffe	Framework open source para *deep learning*, focado em velocidade e modularidade.
caixas negras	Modelos de *IA* cujos processos internos são opacos e difíceis de interpretar ou explicar.
CCPA	California Consumer Privacy Act – Lei de proteção de dados pessoais da Califórnia, EUA.
ChartMimic	*Benchmark* utilizado para avaliar a capacidade de modelos em gerar código visualmente fundamentado a partir de gráficos e instruções textuais.
ChrF	Character n-gram F-score – Métrica que avalia a qualidade de texto gerado com base em n-gramas de caracteres.
Chrome Developer Tools	Conjunto de ferramentas integradas no navegador Google Chrome para desenvolvimento e depuração *web*.
Claude	Modelo de linguagem avançado desenvolvido para tarefas de processamento de linguagem natural.
CLIP	Contrastive Language-Image Pre-training – Modelo que associa imagens e texto para tarefas de reconhecimento e geração multimodal.
CLIP score	Métrica que avalia a correspondência semântica entre texto e imagem usando modelos de *IA*.
cloud	Infraestrutura de computação remota que permite armazenar e processar dados e aplicações através da internet.
CNN	Convolutional Neural *Network* – Rede neuronal convolucional utilizada para processamento e análise de imagens.
complexidade ciclomática	Métrica que quantifica a complexidade de um programa com base no número de caminhos lineares independentes no código.
CPU	Central Processing Unit – Unidade central de processamento, o componente principal de um computador responsável pela execução de instruções.
CSS	Cascading Style Sheets – Folhas de estilo em cascata que definem a apresentação e *layout* de páginas *web*.
CSWR	Client-Side *Web* Remodeling – Técnica que permite modificar e validar protótipos diretamente no navegador sem alterar o código original.

Datasets	Coleções organizadas de dados utilizadas para treino e avaliação de modelos de aprendizagem automática.
Deep learning	Subcampo do *machine learning* que utiliza redes neuronais profundas para aprendizagem automática.
Desenvolvimento ágil	Metodologia de desenvolvimento de *software* que promove entregas rápidas e iterativas com foco na colaboração.
Design	Processo de planeamento e criação de interfaces, produtos ou sistemas.
Design systems	Conjuntos padronizados de componentes, estilos e diretrizes que garantem consistência no design e desenvolvimento de interfaces.
Design-to-code	Processo automatizado que converte designs visuais em código funcional.
DigitalOcean	Plataforma de infraestrutura *cloud* que fornece servidores virtuais e serviços para desenvolvimento e alojamento de aplicações.
DINOv2-base	Modelo de visão computacional usado para gerar *embeddings* visuais que capturam características semânticas de imagens.
DRAFT	Modelo híbrido que integra geração aumentada por recuperação com *fine-tuning* para melhorar decisões de design.
drag-and-drop	Método de interação onde o utilizador arrasta e solta elementos numa interface gráfica para construir ou organizar conteúdos.
DreamBooth	Método de treino para personalizar modelos de Inteligência Artificial com imagens específicas, utilizando poucos dados.
dropdown	Campo de seleção numa interface gráfica que permite escolher uma opção a partir de uma lista suspensa.
DSL	Domain-Specific Language – Linguagem de programação especializada para um domínio ou tarefa específica.
Electron	Framework que permite criar aplicações desktop multiplataforma usando tecnologias *web* como *HTML*, *CSS* e *JavaScript*.
Embeddings	Representações numéricas de texto que capturam o significado contextual para facilitar comparações e buscas semânticas.
Endpoint	Ponto final de comunicação numa *API* onde são recebidos ou enviados dados.

epochs	Ciclos completos de treino de um modelo sobre o conjunto de dados.
Express	Framework para *Node.js* que facilita a criação de aplicações *web* e *API*.
FAISS	Facebook AI Similarity Search – Biblioteca desenvolvida pelo Facebook para indexação e pesquisa eficiente de vetores de alta dimensão.
Feedback	Retroalimentação ou retorno de informação dos utilizadores para melhorar sistemas ou processos.
Figma	Ferramenta de design colaborativo baseada na *web* para criação de interfaces e protótipos.
Fine-tuning	Processo de ajuste fino de um modelo de Inteligência Artificial para melhorar o desempenho em tarefas específicas.
Flexbox	Módulo *CSS* que facilita a criação de *layout*s flexíveis e responsivos em interfaces *web*.
Framework	Estrutura de suporte reutilizável que facilita o desenvolvimento de *software*.
Frontend	Parte de uma aplicação ou *web*site que interage diretamente com o utilizador.
GCP	Google Cloud Platform – Plataforma de serviços de computação em nuvem oferecida pela Google.
GDPR	General Data Protection Regulation – Regulamento europeu que define regras para proteção de dados pessoais.
GitHub Copilot	Assistente de programação baseado em *IA* que sugere código automaticamente durante o desenvolvimento.
Google Vertex AI	Plataforma da Google para criação, treino e implementação de modelos de Inteligência Artificial.
Google's Teachable Machine	Ferramenta da Google que permite criar modelos de *machine learning* de forma simples e acessível.
GPT	Generative Pre-trained Transformer – Modelo de linguagem baseado em transformadores, pré-treinado para gerar texto coerente e contextualizado.
GPU	Graphics Processing Unit – Unidade de processamento gráfico usada para acelerar cálculos, especialmente em *IA*.

HTML	HyperText Markup Language – Linguagem de marcação usada para estruturar conteúdo na *web*.
HTTP	HyperText Transfer Protocol – Protocolo de comunicação utilizado para transferência de informação na *web*.
Hugging Face	Plataforma e comunidade que disponibiliza ferramentas e modelos de Inteligência Artificial para processamento de linguagem natural.
IA	Inteligência Artificial – Área da informática que desenvolve sistemas capazes de realizar tarefas que normalmente requerem inteligência humana.
Input tokens	Unidades básicas de texto processadas por modelos de linguagem, usadas para calcular custos e limites.
Interface gráfica	Conjunto de elementos visuais que permitem a interação do utilizador com um sistema ou aplicação.
ISO/IEC 40500	Norma internacional que especifica os requisitos para acessibilidade de conteúdos *web*, equivalente às *WCAG* 2.0.
JavaScript	Linguagem de programação utilizada principalmente no desenvolvimento *web* para criar conteúdos dinâmicos em páginas *web*.
JSON	*JavaScript* Object Notation – Formato leve de troca de dados, fácil de ler e escrever para humanos e máquinas.
JSONL	*JSON* Lines – Formato de ficheiro onde cada linha é um objeto *JSON* separado, utilizado para armazenar grandes volumes de dados estruturados.
Jupyter Notebooks	Ambiente interativo para criação e partilha de documentos que contêm código, texto e visualizações.
Langchain	Biblioteca para construção de aplicações que combinam modelos de linguagem com outras fontes de dados e lógica.
Layout	Estrutura e organização visual dos elementos numa interface ou página *web*.
Lighthouse	Ferramenta automatizada do Google para auditoria de desempenho, acessibilidade e boas práticas em páginas *web*.

LLaVA	Large Language and Vision Assistant – Modelo multimodal que combina capacidades de linguagem e visão para assistentes inteligentes.
LLM	Large Language Model – Modelo de Linguagem de Grande Escala utilizado para processamento e geração de texto.
Local storage	Armazenamento local do navegador que permite guardar dados no dispositivo do utilizador para persistência entre sessões.
Low-code	Abordagem de desenvolvimento que permite criar aplicações com pouca necessidade de código manual, facilitando a democratização do desenvolvimento.
LoRA	*Low-Rank Adaptation* - Técnica que reduz o número de parâmetros treináveis num modelo, facilitando adaptações rápidas e eficientes.
LSTM	Long Short-Term Memory – Tipo de rede neural recorrente capaz de aprender dependências de longo prazo em sequências.
machine learning	Aprendizagem automática – técnica de Inteligência Artificial que permite aos sistemas aprender e melhorar a partir de dados sem serem explicitamente programados.
macOS	Sistema operativo desenvolvido pela Apple para computadores Macintosh.
Markdown	Linguagem de marcação simples usada para formatar texto com elementos como títulos, listas e código.
MSE	*Mean Squared Error* - Métrica que calcula a média dos quadrados das diferenças entre valores previstos e reais, usada para avaliar erros.
Mimetype	Tipo de conteúdo que indica o formato de um ficheiro ou dado transmitido na internet.
MLLM	Multimodal Large Language Models – Modelos de linguagem de grande escala que processam múltiplos tipos de dados, como texto e imagem.
MLOps	Machine Learning Operations – Conjunto de práticas para gerir e operacionalizar modelos de aprendizagem automática em produção.
mockup	Protótipo visual de uma interface utilizado em design e desenvolvimento para representar a aparência e estrutura antes da implementação.

Multimodal	Modelos ou sistemas que processam e combinam múltiplos tipos de dados, como texto e imagem.
Nebius	Serviço ou plataforma utilizada para processamento de visão computacional.
No-code	Abordagem de desenvolvimento de *software* que permite criar aplicações sem necessidade de escrever código manualmente.
Node.js	Ambiente de execução de *JavaScript* no servidor para desenvolvimento *backend*.
NVIDIA	Empresa fabricante de hardware, especialmente conhecida pelas suas *GPU* para computação gráfica e Inteligência Artificial.
Nx	Ferramenta para gestão de monorepositórios, popular em projetos *JavaScript*/*TypeScript*.
OCR	Optical Character Recognition – Tecnologia que converte imagens de texto em texto editável.
open-source	*Software* cujo código-fonte é aberto e disponível para uso e modificação.
OpenAI	Organização que desenvolve modelos avançados de Inteligência Artificial, incluindo *GPT*.
OpenCV	Biblioteca de visão computacional para processamento e análise de imagens e vídeos.
Output	Resultado produzido por um sistema ou processo.
Output tokens	Unidades básicas de texto geradas por modelos de linguagem, usadas para calcular custos e limites.
PBD	Programming by Demonstration - Técnica onde o utilizador demonstra ações para que o sistema aprenda e automatize tarefas.
PBE	Programming by Example - Técnica onde o utilizador fornece exemplos para que o sistema inferira regras ou programas.
PSNR	*Peak Signal-to-Noise Ratio* - Métrica que mede a qualidade de uma imagem comparando o sinal original com o ruído introduzido.
peer-reviewed	Processo de avaliação por pares, onde especialistas analisam a qualidade e validade de um trabalho científico antes da publicação.

Performance	Desempenho ou eficiência com que um sistema ou aplicação executa tarefas.
PIL	*Python* Imaging Library – Biblioteca para manipulação e processamento de imagens em *Python*.
Pipeline	Conjunto sequencial de processos automatizados que transformam dados de entrada em resultados finais.
plug-and-play	Sistema ou componente que pode ser utilizado imediatamente após a instalação, sem necessidade de configuração complexa.
plugins	Extensões de *software* que adicionam funcionalidades específicas a uma aplicação principal.
portable	Aplicação que pode ser executada sem necessidade de instalação no sistema operativo.
POST	Método *HTTP* utilizado para enviar dados a um servidor para processamento.
PRD	Product Requirements Document – Documento que descreve os requisitos e funcionalidades de um produto ou sistema.
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses – Conjunto de diretrizes para a elaboração e reporte de revisões sistemáticas e meta-análises.
Prompt	Instrução ou conjunto de dados fornecidos a um modelo de Inteligência Artificial para gerar uma resposta ou ação.
PTG	Page Transition Graphs – Representação gráfica dos fluxos de navegação entre páginas numa aplicação.
Python	Linguagem de programação de alto nível, frequentemente usada em ciência de dados e visão computacional.
PyTorch	Framework de *machine learning* de código aberto para desenvolvimento e treino de modelos de *IA*.
Query	Pedido ou consulta feita a um sistema para obter informação ou resultados específicos.
RAG	*Retrieval-Augmented Generation* – Técnica que combina recuperação de informação com geração de texto por modelos de *IA* para melhorar a precisão das respostas.

React	Biblioteca *JavaScript* para construção de interfaces de utilizador, desenvolvida pelo Facebook.
Regex	*Express*ão regular – Sequência de caracteres que define um padrão de pesquisa em texto.
REST	Representational State Transfer – Estilo arquitetural para desenvolvimento de serviços *web* que utilizam métodos *HTTP*.
RGPD	Regulamento Geral sobre a Proteção de Dados – Regulamento europeu que define normas para a proteção de dados pessoais.
RNF	Requisito Não Funcional – Especificação das características de qualidade que um sistema deve ter, como desempenho, segurança e usabilidade.
RNN	Recurrent Neural Networks – Redes neuronais recorrentes, utilizadas para processamento sequencial de dados.
ROUGE	Recall-Oriented Understudy for Gisting Evaluation – Métrica que avalia a qualidade de resumos e textos gerados com base na correspondência de sequências.
SaaS	*Software* as a Service – Modelo de distribuição de *software* onde a aplicação é disponibilizada online como serviço.
safety guardrails	Conjunto de mecanismos ou regras implementadas para garantir que sistemas de *IA* operem de forma segura e ética.
scaffolding	Técnica de desenvolvimento que gera automaticamente a estrutura básica de um projeto ou aplicação para acelerar o processo inicial.
script	Programa ou conjunto de comandos escritos para automatizar tarefas num ambiente de programação.
self-hosted	Solução onde a infraestrutura e *software* são instalados e geridos localmente pelo utilizador ou organização, em vez de na *cloud*.
Snippets	Fragmentos pequenos e reutilizáveis de código que facilitam a programação.
software	Conjunto de programas e outros dados operacionais utilizados por um computador.
Stable Diffusion	Modelo de geração de imagens por *IA* que cria imagens realistas a partir de descrições textuais.

Stack tecnológica	Conjunto de tecnologias, linguagens, *frameworks* e ferramentas utilizadas para desenvolver uma aplicação.
Stakeholders	Partes interessadas ou intervenientes num projeto ou sistema, como desenvolvedores, fornecedores e utilizadores.
standalone	Aplicação ou componente que funciona de forma independente, sem necessidade de componentes externos.
text-to-image	Modelos de *IA* que geram imagens a partir de descrições em texto.
text-to-text	Modelo de *IA* que transforma texto de entrada em texto de saída, usado para tarefas como tradução ou geração de código.
token	Unidade básica de texto processada por modelos de linguagem, podendo ser uma palavra, parte de palavra ou símbolo.
Transformer	Arquitetura de rede neuronal que utiliza mecanismos de atenção para processar sequências de dados, muito usada em processamento de linguagem natural.
TypeScript	Linguagem de programação que é um superconjunto do *JavaScript*, adicionando tipagem estática opcional e outras funcionalidades.
UI	User Interface – Interface de Utilizador, o ponto de interação entre o utilizador e um sistema ou aplicação.
VRAM	Video Random Access Memory – Memória dedicada das *GPU* para processamento gráfico e treino de modelos.
VSCode	*Visual Studio Code* – Editor de código-fonte desenvolvido pela Microsoft, utilizado para programação e desenvolvimento de *software*.
Vue.js	Framework progressivo para construção de interfaces de utilizador, focado na simplicidade e flexibilidade.
WCAG	*Web* Content Accessibility *Guidelines* – Conjunto de normas internacionais para garantir acessibilidade em conteúdos *web*.
WYSIWYG	What You See Is What You Get – Interface que permite editar conteúdos visualizando diretamente o resultado final.

1 Introdução

Neste capítulo é apresentado o enquadramento do tema, a motivação subjacente e o problema identificado. São definidos os objetivos do projeto, a metodologia do trabalho, a abordagem adotada para os atingir e os contributos esperados.

Descreve-se ainda a relevância do estudo tanto a nível académico como prático, bem como as considerações éticas associadas à condução da investigação. Por fim, é apresentada a estrutura do presente documento.

1.1 Enquadramento

O desenvolvimento de interfaces gráficas de utilizador representa uma das atividades mais críticas no ciclo de vida de aplicações *web* modernas. Num contexto marcado pela crescente complexidade dos sistemas e pela necessidade constante de proporcionar experiências de utilizador intuitivas, responsivas e esteticamente apelativas, a eficiência na criação e manutenção de interfaces tornou-se uma prioridade incontornável para equipas de desenvolvimento. As organizações procuram cada vez mais formas de acelerar os seus processos de entrega sem comprometer a qualidade, o que tem conduzido à exploração de tecnologias emergentes, com foco especial na Inteligência Artificial (*IA*) (Ng et al., 2024).

Apesar da evolução das ferramentas de design, como o *Figma* (Bilousova et al., 2021) ou o *Adobe XD* (MacHado & Campos, 2021), que oferecem ambientes ricos para a conceção visual de interfaces, o processo subsequente de transformação dessas representações visuais em código executável continua, em grande medida, manual. Esta transição, frequentemente morosa e propensa a erros, levanta desafios significativos relacionados com a consistência visual, a conformidade com padrões internos, a escalabilidade das soluções e a produtividade global das equipas (MacHado & Campos, 2021).

Neste cenário, têm emergido abordagens baseadas em *IA*, nomeadamente em modelos de linguagem de grande escala (*LLM*) e modelos de visão computacional, como potenciais

aceleradores do desenvolvimento *frontend* (Harrison Oke Ekpobimi et al., 2024). Estas abordagens visam automatizar, total ou parcialmente, a conversão de *mockups* em código funcional, abrindo portas a um novo paradigma na engenharia de interfaces (Aşiroğlu et al., 2019). Contudo, a sua adoção prática exige um estudo rigoroso sobre a forma como estas tecnologias podem ser treinadas, integradas em contextos empresariais reais e avaliadas de forma objetiva (Pacheco et al., 2021).

1.2 Descrição do Problema

Apesar do entusiasmo crescente em torno das tecnologias de automação para a geração de interfaces — conhecidas como soluções de *design-to-code* — a sua adoção em ambientes empresariais tem sido limitada por três lacunas estruturais que comprometem a sua viabilidade prática.

Primeiramente, verifica-se uma fraca **fidelidade ao contexto organizacional** (Ajimati et al., 2025; Kass et al., 2023a, 2023b). As ferramentas atualmente disponíveis tendem a gerar código genérico, desprovido de integração com bibliotecas internas, padrões arquitetónicos específicos ou *guidelines* de acessibilidade, como os definidos pelas *Web* Content Accessibility *Guidelines* (*WCAG*) (*Techniques for WCAG 2.2*, n.d.). O resultado é uma necessidade sistemática de reescrita e adaptação manual por parte das equipas de desenvolvimento, o que anula parte substancial do ganho esperado em eficiência.

Em segundo lugar, observa-se uma **fragmentação acentuada dos processos**. As soluções existentes operam em silos: *plugins* de ferramentas de design que exportam estruturas *JSON* (Kolthoff et al., 2025); modelos baseados em visão computacional geram *HTML* estático descontextualizado (Akter et al., 2024); modelos de linguagem produzem fragmentos de código (*snippets*) de forma desconexa e muitas vezes redundante (Tsai et al., 2025). A ausência de uma integração coerente entre estas etapas inviabiliza a construção de *pipelines* contínuos e reutilizáveis no ciclo de desenvolvimento de *software*.

Por fim, prevalece uma **avaliação excessivamente subjetiva dos resultados**. As abordagens tradicionais concentram-se sobretudo na aparência visual da interface gerada, negligenciando dimensões críticas como a manutenibilidade do código, a sua performance em contextos reais e a conformidade com princípios de engenharia de *software* como os princípios *SOLID* (Evtikhiev et al., 2023a). A inexistência de métricas robustas e sistemáticas dificulta a validação comparativa entre abordagens e inibe o processo de melhoria contínua das ferramentas automatizadas (Saad et al., 2025).

Estas limitações não constituem apenas barreiras de ordem técnica, mas também induzem fricções organizacionais significativas, enfraquecendo a confiança das equipas de desenvolvimento nos processos de automação e contribuindo para a perpetuação de ciclos manuais. A superação destes entraves requer uma abordagem holística que contemple, de forma integrada:

- A adaptação nativa a sistemas de design con*solid*ados, como o *Angular Material*;
- Suporte a requisitos não funcionais essenciais, nomeadamente a responsividade, a internacionalização (*i18n*) e a acessibilidade;
- A capacidade de incorporação de *feedback* humano em ciclos iterativos de validação e correção, garantindo um alinhamento contínuo com os objetivos estratégicos e os padrões técnicos da organização.

1.3 Objetivos

O principal objetivo deste trabalho consiste no desenvolvimento de uma solução com suporte em algoritmos de Inteligência Artificial que permita gerar automaticamente componentes de interface gráfica a partir de *mockups* visuais, de forma compatível com os requisitos técnicos, estéticos e funcionais de um ambiente empresarial real.

Para alcançar este propósito geral, definem-se os seguintes objetivos específicos:

- **Gerar Código Alinhado com Padrões Empresariais:** Conceber uma solução capaz de transformar *mockups* visuais em componentes de interface implementáveis, em conformidade com os *design systems*, *frameworks* e boas práticas adotadas no contexto empresarial.

- **Assegurar Coerência entre Design e Implementação:** Promover a consistência entre as intenções visuais expressas nos *mockups* e o código final gerado, minimizando a necessidade de ajustes manuais e garantindo aderência às diretrizes de acessibilidade, responsividade e reutilização.

- **Estabelecer um Processo Automatizado e Reprodutível:** Estruturar um processo técnico que permita, de forma repetível e escalável, a conversão de interfaces visuais em soluções de código eficazes e integráveis em sistemas reais de *frontend*.

- **Avaliar a Qualidade Técnica do Output:** Definir e aplicar métricas objetivas que permitam aferir a qualidade do código gerado em termos de fidelidade visual, manutenibilidade, desempenho e alinhamento com princípios sólidos de engenharia de *software*.

- **Explorar Estratégias de Otimização com Inteligência Artificial:** Estudar o impacto de diferentes estratégias baseadas em *IA* sobre a eficácia do processo de geração de código, analisando os pontos fortes e fracos entre abordagens mais simples e técnicas avançadas de adaptação contextual.

- **Reduzir o Esforço de Desenvolvimento Manual:** Quantificar os ganhos em produtividade, eficiência e consistência obtidos com a adoção da solução proposta, face às abordagens convencionais de implementação manual de interfaces.

1.4 Metodologia

A condução deste trabalho foi orientada por uma metodologia de investigação estruturada, desenhada para assegurar o rigor, a reprodutibilidade e o alinhamento com os objetivos definidos. O percurso metodológico foi dividido em duas fases macro, sequenciais e interdependentes: (i) **Conceção e Desenvolvimento da Solução** e (ii) **Implementação e Avaliação Experimental**.

A primeira fase foi dedicada à preparação teórica e ao desenho da arquitetura do sistema. Esta etapa inicial incluiu uma revisão sistemática da literatura, recorrendo à metodologia ***PRISMA***, para analisar criticamente as abordagens existentes na geração automática de *interfaces*. Com base neste levantamento, foram definidos os requisitos técnicos e funcionais da solução, o que culminou na conceção de um *pipeline* multimodal e na fundamentação das principais decisões de design. Esta fase foi crucial para estabelecer um alicerce teórico e técnico sólido, garantindo que a solução proposta seria não só inovadora, mas também viável e relevante.

A segunda fase consistiu na concretização técnica e na validação prática da solução proposta. Procedeu-se à implementação dos módulos principais do sistema, tanto do *backend* como do *frontend*, seguida pela realização de um conjunto de testes sistemáticos. Estes testes, baseados em *mockups* reais, visaram simular um ambiente empresarial realista, permitindo a recolha de métricas objetivas de desempenho, qualidade e produtividade. O objetivo desta fase foi aferir o impacto e a eficácia da solução no ciclo de desenvolvimento de interfaces e avaliar o seu potencial de integração em fluxos de trabalho existentes.

Para garantir a execução disciplinada e a gestão eficaz da complexidade do projeto, foi mantido um cronograma detalhado, em formato de diagrama de ***Gantt*** (disponível no Anexo A), que permitiu monitorizar o progresso e assegurar uma distribuição equilibrada de esforços. Esta abordagem metodológica rigorosa, que articula a fundamentação teórica com a validação empírica, assegurou que todas as etapas do projeto fossem tratadas com a profundidade e a seriedade exigidas por uma dissertação de natureza científica e aplicada.

1.5 Abordagem

Para resolver o problema da transformação de *mockups* em código, foi concebida uma solução técnica que assenta num **pipeline multimodal híbrido**. Esta abordagem combina a capacidade de interpretação visual de imagens com a geração de código contextualizado por modelos de linguagem avançados.

O processo inicia-se com a análise do *mockup* visual por um **modelo de visão computacional**, que o converte numa descrição textual e estruturada (*prompt*). Este *prompt* serve de entrada para ***LLM*** da *OpenAI*. Crucialmente, este *LLM* não é utilizado de forma genérica; ele é previamente ajustado ao contexto da empresa através de ***fine-tuning***.

Este treino específico garante que o código gerado adere ao *framework* ***Angular*** e utiliza componentes da biblioteca ***Angular Material***, respeitando as convenções e padrões internos.

Para materializar esta abordagem, a solução foi desenvolvida como uma aplicação *standalone* e portátil utilizando ***Electron***. A arquitetura é gerida num monorepositório com ***Turborepo***, separando o *frontend*, implementado em ***Angular***, do *backend* em ***Node.js*** com ***Express***, que orquestra a comunicação com as *API* externas de *IA* e gere a lógica da aplicação.

1.6 Contributos

A realização deste trabalho resultou no desenvolvimento e avaliação de uma solução inovadora de geração automática de interfaces, aplicada a contextos empresariais com *design systems* con*solid*ados. A investigação contribui de forma significativa para o campo da engenharia de *software* assistida por Inteligência Artificial, ao explorar como modelos multimodais podem ser utilizados para reduzir o esforço manual associado à implementação de componentes *UI* a partir de *mockups* visuais.

Um dos principais contributos desta dissertação é a proposta de um processo estruturado para transformar representações visuais em código funcional, respeitando normas internas, *frameworks* específicas e requisitos não funcionais como acessibilidade e responsividade. A solução desenvolvida evidencia como a integração de Inteligência Artificial no fluxo de trabalho de equipas de *frontend* pode contribuir para uma maior consistência visual, redução de erros humanos e aceleração do tempo de entrega.

Adicionalmente, o estudo estabelece um conjunto de critérios objetivos para avaliar a qualidade do código gerado, indo além da análise estética para incluir métricas relacionadas com manutenibilidade, alinhamento com padrões técnicos e desempenho. Estes critérios representam um referencial metodológico útil para futuras investigações ou iniciativas empresariais interessadas em adotar ferramentas de *design-to-code* baseadas em *IA*.

Finalmente, este trabalho reforça a relevância da interação entre ferramentas automatizadas e *feedback* humano num ciclo iterativo de melhoria contínua, demonstrando como a colaboração entre agentes inteligentes e programadores pode resultar em soluções mais eficazes, adaptáveis e integradas na realidade operacional das empresas. Os resultados obtidos contribuem para a maturação do debate sobre automação inteligente no desenvolvimento de interfaces e abrem caminho para investigações futuras centradas na escalabilidade e industrialização deste tipo de soluções.

1.7 Relevância do estudo

A problemática central abordada nesta investigação incide sobre a viabilidade e eficácia da aplicação de tecnologias de Inteligência Artificial na geração automatizada de interfaces

gráficas, a partir de *mockups* visuais, em ambientes empresariais com requisitos específicos e elevados padrões de qualidade.

A complexidade deste desafio ultrapassa a mera seleção técnica de modelos de *IA* apropriados. Envolve uma multiplicidade de fatores, desde a conformidade do código gerado com *frameworks* e *design systems* existentes, até à capacidade de integração em ciclos de desenvolvimento reais, onde aspetos como acessibilidade, responsividade e manutenibilidade são inegociáveis.

Um elemento crítico desta avaliação é a medição do desempenho da *IA* com base em critérios objetivos, nomeadamente a fidelidade visual do output, a eficiência no processo de geração, a clareza estrutural do código e a sua aderência a boas práticas de engenharia de *software*. Esta análise tem em consideração os interesses de diversos *stakeholders* — programadores, gestores de produto e utilizadores finais — assegurando que a solução tecnológica proposta responde às necessidades concretas de todos os intervenientes.

Do ponto de vista ético e técnico, esta investigação também reconhece os riscos associados à automação inteligente, como a possibilidade de perpetuar vieses ou gerar código ineficiente e difícil de manter. Assim, o estudo defende uma abordagem informada, crítica e iterativa, sustentada por evidência empírica e por metodologias transparentes, garantindo a robustez das conclusões obtidas e a sua aplicabilidade em contextos reais.

Num setor cada vez mais pressionado pela necessidade de entregar *software* de forma mais rápida, eficiente e escalável, a incapacidade de integrar eficazmente soluções de *IA* pode representar uma perda significativa de competitividade. Pelo contrário, uma integração bem-sucedida destas tecnologias tem o potencial de transformar profundamente os processos de desenvolvimento *frontend*, promovendo inovação, qualidade e sustentabilidade no ciclo de vida das interfaces digitais.

Em suma, a relevância deste estudo reside na sua abordagem multidimensional e orientada para a prática, procurando não apenas compreender, mas também demonstrar, de forma rigorosa e aplicada, como a Inteligência Artificial pode ser uma aliada estratégica na criação de interfaces modernas e eficientes.

1.8 Considerações Éticas

A presente investigação foi conduzida em estrita conformidade com os princípios de integridade científica e com o código de conduta ética instituído pelo Instituto Politécnico do Porto, cumprindo integralmente as disposições do Código de Boas Práticas e Conduta (República, 2020), bem como a Declaração de Integridade incluída neste documento. Importa destacar que todas as fontes externas utilizadas foram devidamente reconhecidas, respeitando os direitos de autor e assegurando a atribuição explícita dos contributos alheios, de acordo com as normas de citação académica.

No que concerne à redação do presente documento, foram empregues ferramentas de Inteligência Artificial exclusivamente como suporte à elaboração textual, nomeadamente para efeitos de reformulação linguística, enriquecimento lexical e auxílio estilístico na construção de frases. Estas ferramentas não foram utilizadas para gerar conteúdo técnico, introduzir informação factual, nem para substituir o processo analítico ou crítico inerente à investigação. O seu uso foi sempre supervisionado pelo autor, garantindo-se que nenhuma informação falsa, imprecisa ou infundada foi incorporada no corpo do texto.

A utilização de tais ferramentas foi pontual, controlada e declarada neste capítulo, alinhando-se com os princípios de transparência e responsabilidade ética exigidos no âmbito do ensino superior e da investigação científica. A autoria intelectual e científica do trabalho é integralmente assumida pelo autor, cabendo-lhe a total responsabilidade pela veracidade, originalidade e validade dos conteúdos apresentados.

1.9 Estrutura do documento

A presente dissertação encontra-se organizada de forma lógica e sequencial, acompanhando o percurso completo da investigação — desde a definição do problema até à avaliação dos resultados obtidos. Esta estrutura visa assegurar uma leitura coerente, promovendo a articulação entre os fundamentos teóricos, o desenvolvimento técnico e a reflexão crítica da solução proposta. O primeiro capítulo introduz o tema e estabelece o enquadramento geral do estudo, apresentando a motivação subjacente, o problema identificado, os objetivos definidos, a abordagem metodológica adotada, os contributos esperados, o planeamento das atividades e a relevância do trabalho.

A segunda parte do documento é dedicada à revisão do conhecimento existente na área em estudo. Com base numa metodologia de revisão sistemática, são analisadas as abordagens mais relevantes relacionadas com a geração automática de código a partir de *mockups*, incluindo modelos multimodais, estratégias de personalização de *IA* e métricas de avaliação do código gerado. Na terceira secção, procede-se à análise e conceção da solução proposta. São identificados os requisitos funcionais e não funcionais, fundamentadas as decisões de design, e apresentada a arquitetura técnica que sustenta o sistema desenvolvido, tendo sempre como referência o contexto empresarial que motivou o projeto.

A implementação da solução, descrita na quarta secção, inclui o desenvolvimento dos componentes principais, a seleção das tecnologias utilizadas e a superação dos desafios técnicos encontrados durante o processo de construção. Num quinto momento, são apresentados os resultados obtidos através de um conjunto de experimentações com *mockups* reais. A análise é conduzida com base em métricas previamente definidas, permitindo avaliar a qualidade e eficácia da solução em cenários representativos do seu contexto de aplicação.

A sexta secção encerra o trabalho com a apresentação das principais conclusões. É feita uma reflexão sobre os objetivos alcançados, os contributos obtidos e as limitações

identificadas, sendo também propostas linhas de investigação futura que possam dar continuidade e aprofundar os resultados deste estudo.

Em suma, a organização seguida permite uma exploração gradual e rigorosa da problemática abordada, sustentando o percurso investigativo numa lógica académica exigente, mas orientada para a aplicação prática em contextos reais.

2 Estado da Arte

Este capítulo procura estabelecer a base teórica da investigação, através da exploração crítica do conhecimento existente sobre a geração automática de interfaces gráficas com recurso a Inteligência Artificial. O foco incide particularmente na transformação de *mockups* em código *UI*, uma área em crescente evolução no âmbito da engenharia de *software* assistida por modelos de *machine learning* e *pipelines* multimodais.

Para orientar a análise, este capítulo inicia-se com a apresentação detalhada da metodologia de revisão sistemática da literatura, baseada no protocolo *PRISMA*, que estrutura a seleção e a síntese do conhecimento existente. A reflexão subsequente explora os principais modelos e abordagens para a conversão de mockups em código, com particular ênfase em *pipelines* multimodais e arquiteturas *vision-to-code*. De seguida, a análise aprofunda as estratégias de adaptação de modelos de *IA*, comparando criticamente técnicas como *fine-tuning* e *Retrieval-Augmented Generation (RAG)*, e examina as plataformas que facilitam a sua implementação. O capítulo debruça-se ainda sobre as métricas cruciais para a avaliação da qualidade do código gerado. Finalmente, são discutidas as implicações éticas, de segurança e de governança associadas à adoção destas tecnologias, oferecendo uma perspetiva abrangente e crítica que sustenta o desenvolvimento da solução proposta nos capítulos seguintes.

2.1 Metodologia

A metodologia de investigação seguida nesta dissertação fundamenta-se num modelo de revisão sistemática da literatura, orientado pelo método *PRISMA*. Esta metodologia permite uma abordagem estruturada e rigorosa, frequentemente utilizada em estudos académicos e científicos, garantindo que as fontes analisadas são relevantes, fiáveis e contextualizadas com os objetivos da investigação.

A adoção do modelo *PRISMA* permitiu não só a recolha de dados de forma exaustiva, como também a triagem criteriosa de literatura científica, técnica e prática relacionada com modelos de *IA* aplicados ao desenvolvimento *frontend*, incluindo a conversão de *mockups*, modelos *vision-to-code*, estratégias de *fine-tuning*, *RAG (Retrieval-Augmented Generation)*, e plataformas de treino e inferência. Os procedimentos adotados asseguraram a coerência entre as questões de investigação, os critérios de seleção, e a análise dos dados extraídos, facilitando a construção de um corpo teórico sólido que sustenta as conclusões deste trabalho.

2.1.1 Questões de pesquisa

A presente investigação procura compreender de que forma os avanços em Inteligência Artificial podem ser aplicados eficazmente ao processo de desenvolvimento de interfaces

gráficas, automatizando a transição de protótipos visuais para código funcional e adaptado ao contexto empresarial. Para orientar a análise, foram formuladas as seguintes questões de pesquisa:

- **Q1.** Quais são os principais modelos, técnicas e *pipelines* existentes para a conversão automática de *mockups* em código *UI*?
- **Q2.** De que forma se comparam as abordagens de *fine-tuning* com métodos baseados em recuperação contextual, como o *RAG*, na adaptação dos modelos de *IA* a contextos específicos?
- **Q3.** Que metodologias têm sido propostas para facilitar a personalização de modelos de *IA* por utilizadores finais sem conhecimento técnico?
- **Q4.** Que abordagens podem ser concebidas de geração de código frontend que se integrem diretamente em contextos empresariais, com o mínimo de adaptação aos processos existentes?

Estas questões constituem o fio condutor da revisão da literatura e da subsequente análise comparativa, permitindo avaliar não só o estado da arte, mas também o potencial de inovação prática no domínio da geração automatizada de interfaces gráficas com recurso a *IA*.

2.1.2 Fontes e conjuntos de dados

A recolha de dados para esta investigação baseou-se numa seleção criteriosa de coleções eletrónicas e repositórios científicos reconhecidos internacionalmente, de modo a garantir a relevância, fiabilidade e atualidade dos estudos analisados. Foram privilegiadas fontes que disponibilizam artigos revistos por pares, estudos técnicos e publicações especializadas no domínio da Inteligência Artificial aplicada ao desenvolvimento de *software* e design de interfaces.

As coleções selecionadas encontram-se abaixo representadas na Tabela 1:

Tabela 1 - Coleções utilizadas

Identificador	Base de Dados	*URL*
F1	ScienceDirect	https://www.sciencedirect.com
F2	IEEE Xplore	https://ieeexplore.ieee.org
F3	ACM Digital Library	https://dl.acm.org
F4	ArXiv	https://arxiv.org
F5	SpringerLink	https://link.springer.com
F6	Google Scholar	https://scholar.google.com

Estes repositórios foram selecionados por cobrirem amplamente as áreas de *machine learning*, engenharia de *software*, *HCI* (*Human-Computer Interaction*), e práticas de desenvolvimento assistido por *IA*. Além das coleções de referências bibliográficas, foram

também analisados documentos técnicos, dissertações de mestrado e documentação oficial de plataformas como *OpenAI*, *Nebius*, *Hugging Face* e outras, com o objetivo de compreender as suas capacidades tecnológicas no que respeita a treino, inferência e personalização de modelos.

Complementarmente, alguns estudos de caso e experiências práticas com ferramentas e plataformas comerciais foram incluídos para enriquecer a dimensão aplicada da investigação, oferecendo uma perspetiva mais realista sobre os desafios e oportunidades de integração em contexto empresarial.

2.1.3 Termos de pesquisa e estratégia *PRISMA*

A estratégia de pesquisa adotada nesta investigação baseou-se no protocolo *PRISMA* com o objetivo de garantir uma recolha sistemática, rigorosa e reprodutível da literatura relevante sobre a aplicação de Inteligência Artificial na automatização do desenvolvimento de interfaces gráficas. A utilização deste protocolo permite assegurar a transparência e robustez metodológica dos processos de identificação, triagem, elegibilidade e inclusão dos estudos analisados.

Para a definição dos termos de pesquisa, foram identificadas palavras-chave representativas das principais dimensões temáticas da investigação. Os conceitos centrais incluíram: *Inteligência Artificial*, *Machine Learning*, *Geração de Código*, *Interfaces Gráficas*, *Transformação de Mockups*, *Modelos Vision-to-Code*, *Fine-Tuning*, *RAG*, *Automação de Frontend* e *Plataformas de Treino de Modelos*. A formulação das *queries* foi feita utilizando operadores booleanos (AND, OR) para combinar termos relevantes e abranger diferentes variantes terminológicas.

As quatro principais *queries* definidas foram:

- ("*frontend*" OR "*UI*" OR "*user* interface") AND ("artificial intelligence" OR "*machine learning*") AND ("code generation" OR "*design-to-code*" OR "*mockup* conversion").
- ("*fine-tuning*" OR "transfer learning") AND ("retrieval-augmented generation" OR "*RAG*") AND ("comparison" OR "evaluation") AND ("code generation" OR "model adaptation").
- ("no code" OR "low code") AND ("*fine-tuning*" OR "Machine Learning" OR "Google's Teachable Machine").
- ("AI integration" OR "AI adoption" OR "AI introduction" OR "AI implementation" OR "AI onboarding") AND ("minimal effort" OR "plug-and-play") AND ("enterprise" OR "business").

Estas *queries* foram aplicadas em todas as coleções descritas na Tabela 1, sendo adaptadas conforme a sintaxe exigida por cada motor de busca. Foram também utilizados filtros para restringir os resultados a publicações entre 2015 e 2025, escritas em inglês ou português,

e que apresentassem estudos empíricos, *frameworks* técnicos, ou análises sistemáticas sobre as tecnologias em análise.

A aplicação do protocolo *PRISMA* será complementada nas próximas secções com a descrição detalhada dos critérios de elegibilidade, triagem e extração de dados, bem como com a apresentação do fluxograma do processo de seleção de estudos.

2.1.4 Critérios de elegibilidade e seleção

Para garantir a relevância e qualidade científica das fontes incluídas na revisão sistemática, foram definidos critérios de inclusão e exclusão rigorosos, alinhados com os objetivos da investigação.

Critérios de Inclusão (CI):

- **CI1.** A fonte aborda a aplicação de modelos de Inteligência Artificial no domínio da geração automática de interfaces gráficas ou no desenvolvimento *frontend* assistido.
- **CI2.** O artigo é revisto por pares (*peer-reviewed*) ou provém de fontes reconhecidas em repositórios científicos de acesso aberto (como arXiv ou HAL).
- **CI3.** O estudo apresenta uma abordagem prática, experimental ou metodológica sobre ferramentas, *pipelines*, ou estratégias de personalização de modelos de *IA* (ex. *fine-tuning*, *RAG*, *vision-to-code*).
- **CI4.** O conteúdo foi publicado entre 2015 e 2025, em inglês ou português.

Critérios de Exclusão (CE):

- **CE1.** O artigo não contém exemplos práticos, estudos de caso ou aplicações tangíveis no contexto da geração de código *UI* com *IA*.
- **CE2.** A publicação é puramente teórica ou opinativa, sem suporte empírico ou técnico relevante para a investigação.
- **CE3.** O conteúdo é redundante ou sobreposto a estudos mais completos encontrados em fases anteriores do processo de triagem.

Estes critérios foram aplicados durante as etapas de triagem descritas no procedimento *PRISMA*, assegurando uma seleção rigorosa e coerente com os objetivos da dissertação.

2.1.5 Procedimento de recolha e síntese de dados

O processo de recolha e síntese dos dados foi conduzido de acordo com a metodologia PRISMA, de forma a assegurar uma abordagem estruturada, transparente e reprodutível. O Anexo B apresenta o fluxograma que sintetiza as diferentes etapas envolvidas na seleção dos estudos para esta revisão.

Inicialmente, foram identificados **307** registos através da pesquisa nas coleções e registos selecionados. Antes de iniciar o rastreio, **11** destes registos foram removidos por serem duplicados, resultando num total de **296** registos únicos.

Na fase de rastreio, os **296** registos foram analisados com base nos seus títulos e resumos. Desta análise, **19** registos foram excluídos por não cumprirem os critérios de inclusão preliminares. Consequentemente, **277** relatórios foram considerados para a fase de recuperação.

Dos **277** relatórios procurados para recuperação, **39** não puderam ser obtidos (e.g., acesso restrito, artigos não localizados). Os **238** relatórios restantes foram então avaliados na íntegra quanto à sua elegibilidade. Nesta fase de avaliação detalhada, **41** relatórios foram excluídos pelos seguintes motivos:

- Falta de estudo prático (n = 12)
- Relevância insuficiente (n = 29)

Este processo culminou na inclusão de **197** estudos na revisão sistemática da literatura, os quais formaram a base para a análise do estado da arte apresentada neste capítulo.

2.2 Contextualização

O avanço das tecnologias de Inteligência Artificial tem impulsionado uma transformação substancial no campo da engenharia de *software*, particularmente no desenvolvimento automatizado de interfaces gráficas de utilizador (*UI*). Esta secção propõe-se a realizar uma análise crítica do estado da arte na utilização de modelos de *IA* para a conversão de *mockups* em código funcional, explorando os fundamentos técnicos, arquiteturas emergentes e abordagens multimodais que integram inputs visuais, textuais e semânticos (Duan et al., 2020; O'Donovan et al., 2015; Stige et al., 2023). São examinadas as principais estratégias de adaptação, nomeadamente o *fine-tuning* e o *RAG* (Ghodratnama & Zakershahrak, 2023), bem como os desafios na construção de *pipelines* robustos e personalizados (Siirtola & Röning, 2019). Igualmente relevante é a investigação de métodos que permitem que utilizadores finais colaborem no processo de afinação dos modelos de forma acessível e não técnica (Torka & Albayrak, 2024).

Complementarmente, são exploradas as frameworks de treino e implementação de *IA* atualmente disponíveis, desde ambientes altamente configuráveis como *TensorFlow*, *Keras*, *Caffe* (Thon et al., 2021), até soluções mais acessíveis como *OpenAI API* ou ferramentas *no-code* (Villegas-Ch. et al., 2021). A secção 2.9 aprofunda ainda as métricas utilizadas para avaliar a qualidade do código *UI* gerado, considerando aspetos de sintaxe, fidelidade visual, acessibilidade e usabilidade. Por fim, discute-se o enquadramento ético, regulamentar e de governança associado à aplicação destas tecnologias, destacando a importância da segurança de dados, da explicabilidade algorítmica e da conformidade com normas internacionais como o

RGPD. Esta análise integrada estabelece uma base sólida para o desenvolvimento responsável e eficaz de soluções inteligentes no domínio do *frontend* automatizado.

2.3 Transformação de *mockups* em código *UI*

A conversão automática de *mockups* em código *frontend* tem vindo a afirmar-se como uma área estratégica na interligação entre o design de interfaces e o desenvolvimento *web*. Os *mockups* — normalmente desenvolvidos em ferramentas como *Figma*, *Adobe XD* ou Sketch — são representações visuais estáticas que necessitam de ser traduzidas para código funcional em *HTML*, *CSS* e *JavaScript*. Este processo, quando realizado manualmente, exige conhecimentos técnicos especializados e um esforço significativo, implicando uma interpretação rigorosa dos elementos visuais, como *layout*, tipografia, cores e espaçamento, de modo a preservar a fidelidade visual e comportamental da interface concebida.

Com a crescente complexidade das interfaces modernas, bem como a exigência de responsividade, acessibilidade e desempenho, surgiram diversas abordagens automatizadas baseadas em Inteligência Artificial, com destaque para o uso de visão computacional e modelos de linguagem multimodais. O sistema *pix2code*, proposto por Beltramelli (2017), foi um dos primeiros a explorar redes neuronais profundas para prever código a partir de *screenshots* de interfaces (Beltramelli, 2017). Utilizando uma combinação de *CNN* para processamento de imagem e *RNN* para geração sequencial de *tokens*, o modelo alcançou uma precisão considerável, tendo posteriormente sido melhorado com *LSTM* bidirecionais (Baul'e et al., 2021; Y. Liu et al., 2018). As Figura 1 e Figura 2 ilustram a arquitetura de funcionamento do sistema *pix2code*, detalhando o fluxo desde a entrada da interface gráfica até a geração do código final:

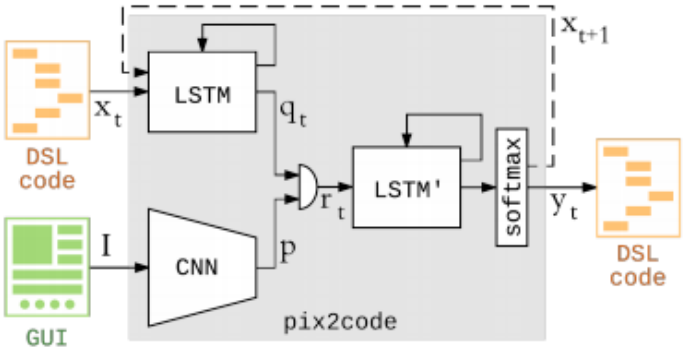


Figura 1 - Arquitetura da rede de treino do sistema pix2code.
(Szymkowiak, 2025)

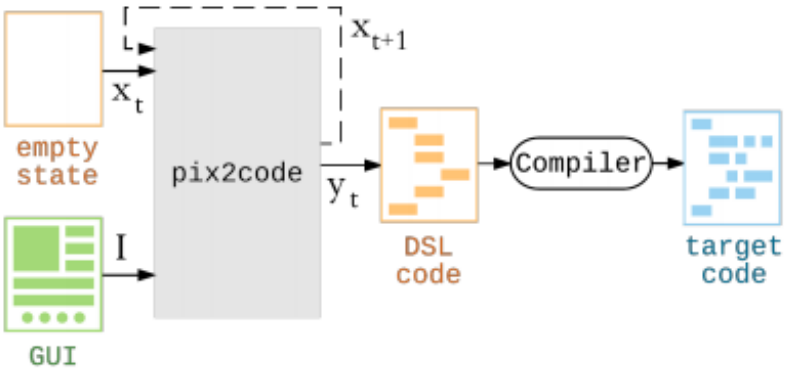


Figura 2 - Fluxo da rede de inferência do sistema pix2code.
(Szymkowiak, 2025)

Seguiram-se propostas como o *Sketch2Code*, que recorre a redes neurais profundas para deteção e classificação de elementos visuais a partir de esboços manuais, gerando representações de *UI* que podem ser convertidas em código (Baul'e et al., 2021; Jain et al., 2019). Por seu lado, o sistema *REMAUI* integra visão computacional e *OCR* para identificar componentes como texto, botões ou imagens em *mockups* visuais, automatizando a criação da interface (Nguyen & Csallner, 2015).

Com os avanços recentes nos modelos de linguagem multimodais, surgiram soluções como o *DeclarUI*, que combina visão computacional com *MLLM* e otimização orientada por compilador, gerando código declarativo de elevada qualidade a partir de *mockups* (T. Zhou et al., 2024). A abordagem *Prototype2Code* introduz um processo *end-to-end* que abrange desde a verificação do protótipo até à geração do *layout* e do estilo final, incluindo o reconhecimento de componentes de *UI* através de redes neuronais de grafos (Xiao et al., 2024).

Estudos recentes alargaram a capacidade de entrada, passando a incluir não apenas esboços manuais (Asiroglu et al., 2019; Suleri et al., 2019), mas também *mockups* criados em ferramentas como *Balsamiq* (Bouças & Esteves, 2020). A utilização de modelos como *GPT-4o* ou *Claude* revelou-se promissora na geração de código com estrutura sólida, gestão eficiente de estado e maior adaptabilidade (Ge et al., 2025).

Estas abordagens seguem geralmente um fluxo de trabalho dividido em três etapas: identificação e segmentação dos elementos visuais, criação de uma representação estrutural da interface e, por fim, geração do código *HTML*, *CSS* e, em alguns casos, *JavaScript* (Barua et al., 2022; Vitkare et al., 2023). Têm-se revelado eficazes na redução do tempo de desenvolvimento, aumentando a produtividade e promovendo maior consistência entre design e implementação.

Outras ferramentas recentes, como o *Frontend Diffusion*, tiram partido de *LLM* para transformar esboços e instruções textuais em *websites* completos (Ding et al., 2025; Q. Zhang et al., 2024), enquanto o *Design2Code* e o *screen-to-code* exploram a conversão direta de

imagens de design em código funcional, melhorando a integração entre a entrada visual e os requisitos de implementação (Ge et al., 2025).

A necessidade de conjuntos de dados robustos tem sido uma constante nesta área. O *WebCode2M*, com mais de 2,5 milhões de pares de imagens de *UI* e código correspondente, representa um recurso fundamental para o treino e avaliação de modelos, permitindo avanços significativos no desempenho e generalização das soluções (Gui et al., 2024).

Por fim, métodos como o *CSWR (Client-Side Web Remodeling)* contribuem para a experimentação e validação de protótipos diretamente no navegador, sem a necessidade de modificar o código original. A ferramenta *UX-Painter*, que implementa esta técnica, permite alterações *WYSIWYG* que podem ser persistidas e partilhadas entre navegadores, sendo particularmente útil para testes de usabilidade remotos (Gardey et al., 2020).

Em suma, a transformação de *mockups* em código *UI* tem vindo a evoluir rapidamente, alavancada por modelos de *deep learning* e linguagem multimodal, bem como por conjuntos de dados cada vez mais abrangentes. Estas soluções não só encurtam o ciclo de desenvolvimento, como também aumentam a qualidade das interfaces produzidas, contribuindo para uma maior convergência entre o design e a engenharia de *software*.

2.4 Modelos *Vision-to-Code* e *pipelines* multimodais

A conversão automática de designs visuais em código funcional tornou-se um campo emergente no desenvolvimento de interfaces, com impacto direto na produtividade e acessibilidade do desenvolvimento *frontend*. Este processo, tradicionalmente dependente de conhecimentos técnicos especializados, limita a participação de pessoas sem formação em programação, mesmo quando estas têm uma visão clara do que pretendem criar (Si et al., 2024). Adicionalmente, a necessidade de colaboração entre designers e programadores pode introduzir ineficiências e desalinhamentos entre o design e a sua implementação. Neste contexto, os modelos *Vision-to-Code* têm assumido um papel crucial ao permitir a automatização da geração de código a partir de imagens de interfaces ou protótipos.

A principal abordagem para esta transformação baseia-se na utilização de *Vision-Language Models – VLM*, que combinam entradas visuais e textuais para inferir a estrutura da interface e gerar o código correspondente. Estes modelos têm sido aplicados a cenários como a geração de código *HTML* a partir de *screenshots* ou esboços de *UI*, com resultados promissores (Laurençon et al., 2024). A capacidade dos *VLM* para compreender representações visuais e traduzi-las em estruturas semânticas tem impulsionado o surgimento de soluções *no-code* ou *low-code*, promovendo uma democratização efectiva do desenvolvimento *web* (Si et al., 2024).

A evolução dos sistemas *Vision-to-Code* pode ser entendida a partir de dois paradigmas principais: os *pipelines* tradicionais multi-etapas e os modelos multimodais *end-to-end*. No primeiro caso, destacam-se sistemas como o *Sketch2Code*, que utiliza visão computacional e serviços cognitivos para reconhecer elementos visuais desenhados à mão, extrair texto

manuscrito e gerar o *layout HTML* correspondente (Huang & Yang, 2022). Contudo, esta abordagem está a ser progressivamente ultrapassada por modelos de larga escala capazes de realizar todo o processo de forma integrada.

Modelos multimodais como o *CLIP*, *Flamingo*, *BLIP-2* e *LLaVA* têm demonstrado forte desempenho em tarefas que envolvem interpretação conjunta de texto e imagem, constituindo uma base sólida para a geração de código a partir de *mockups* ou *screenshots* (Alayrac et al., 2022; Le et al., 2024; H. Liu, Li, Li, et al., 2023; H. Liu, Li, Wu, et al., 2023; Radford et al., 2021). Outros modelos, como o *LayoutLMv3* e o *LayoutPrompter*, foram concebidos especificamente para a compreensão de *layout*s estruturados e a sua conversão em código utilizando técnicas de pré-treino auto-supervisionado e geração aumentada por recuperação (He et al., 2024; J. Lin et al., 2023).

Paralelamente, têm sido propostas abordagens especializadas para transformar diagramas e fluxogramas em código executável. O modelo *FloCo-T5*, por exemplo, converte imagens de fluxogramas em código *Python* através de um processo em duas etapas (Bechard et al., 2025; Shukla et al., 2025). Neste domínio, o *benchmark ChartMimic* tem sido utilizado para avaliar a capacidade de modelos multimodais em gerar código visualmente fundamentado com base em gráficos e instruções textuais (Shi et al., 2024).

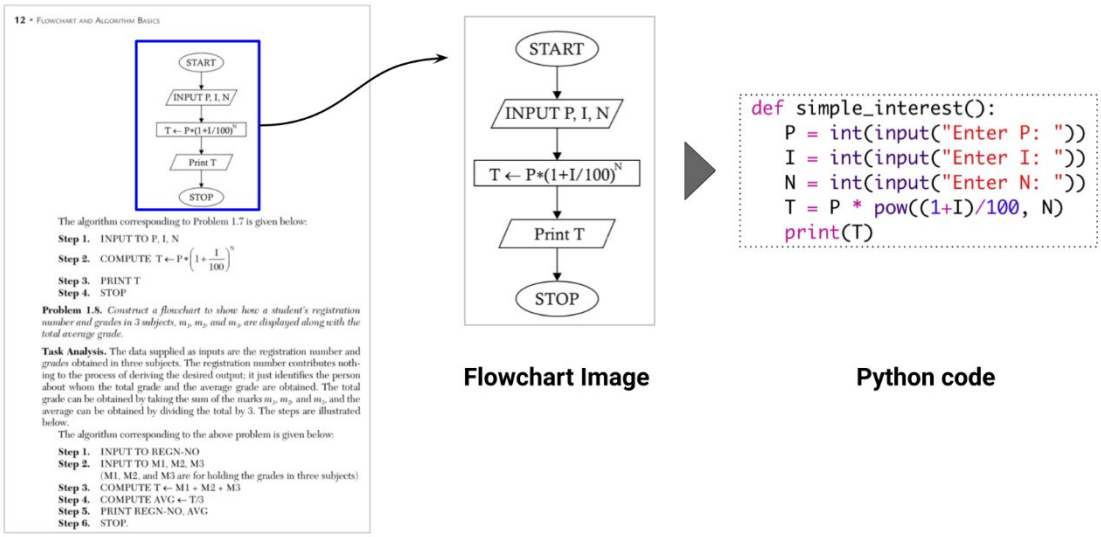


Figura 3 – Ilustração do funcionamento do modelo *FloCo-T5 (J, 2023)*

Na prática, várias plataformas comerciais incorporaram estas técnicas em *pipelines* multimodais aplicáveis ao desenvolvimento de *UI*. Ferramentas como o *Imgcook* e o *CodeFun* utilizam visão computacional profunda para identificar componentes e estruturas de *layout*, convertendo os resultados para linguagens de domínio específico (*DSL*), que são depois transformadas em código *HTML*, *React* ou *Vue* (Xiao et al., 2024). De forma semelhante, o *Anima* aproveita modelos de linguagem de grande escala para gerar código com base em requisitos expressos em linguagem natural, permitindo uma maior flexibilidade e adaptabilidade (Xiao et al., 2024).

No âmbito do design interativo, plataformas como o *React UI Workflow* adoptam fluxos declarativos baseados em *TypeSpec*, permitindo que desenvolvedores e clientes colaborem iterativamente na construção de interfaces com base em dados fictícios, ajustando o design e a lógica em tempo real (Marron, 2024). Já o sistema *Frontend Diffusion* emprega um fluxo de trabalho dividido em duas fases — geração de documentos de requisitos (*PRD*) a partir de esboços e posterior geração de código —, utilizando agentes especializados para garantir a consistência e robustez da interface final (Ding et al., 2025). A figura abaixo, representa o funcionamento base deste sistema:

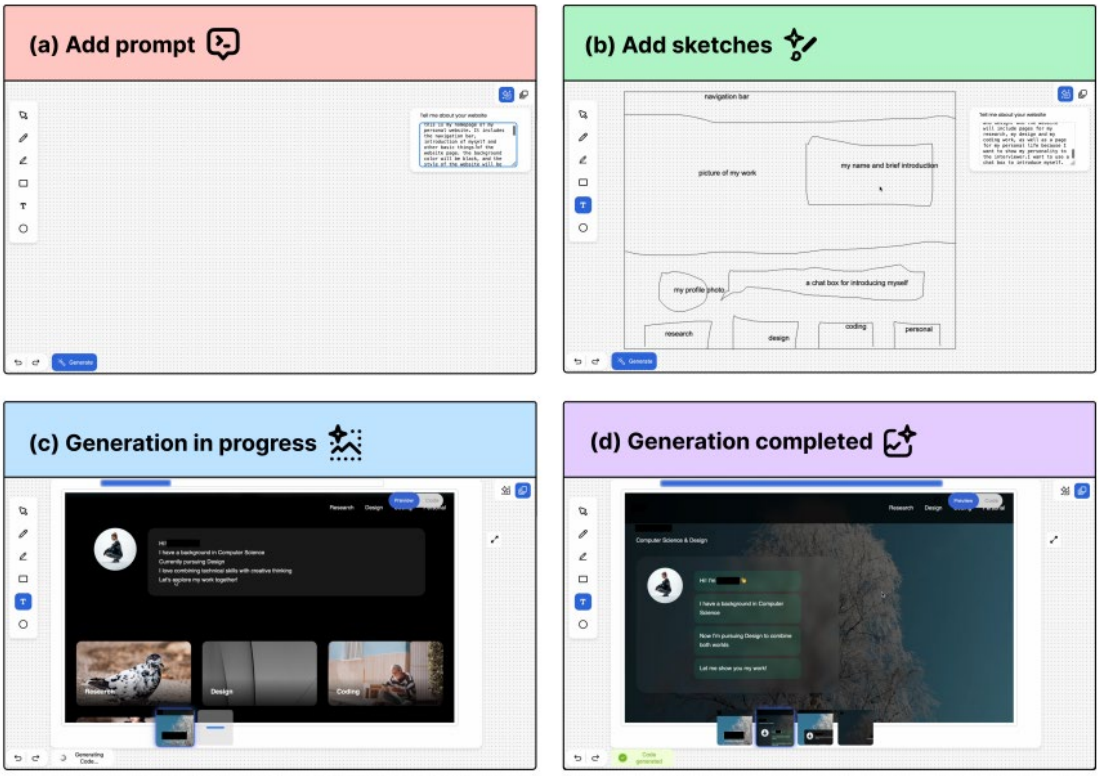


Figura 4 – Etapas de funcionamento do sistema *Frontend Diffusion*
(Ding et al., 2025)

Para sustentar estes avanços, têm sido desenvolvidos *datasets* específicos e ferramentas de suporte. O *LOWCODER*, por exemplo, é uma plataforma multimodal que permite o desenvolvimento de *pipelines* de *IA* através de interfaces visuais e comandos em linguagem natural, apoiando a construção de fluxos complexos com operadores personalizados (Y. Liu et al., 2024). Estas soluções ampliam a acessibilidade ao desenvolvimento de *software* e reforçam o papel da *IA* na automatização da construção de interfaces visuais.

Outro campo relevante é a geração de visualizações de dados, onde modelos como o Lida têm sido utilizados para criar código de gráficos a partir de *datasets* simples. Com a maturidade dos *LLM*, tem-se verificado uma transição do código genérico para soluções personalizadas que adaptam a visualização ao conteúdo dos dados (Bühler et al., 2024; Dibia, 2023). Esta geração de código de visualização, frequentemente em *Python* com bibliotecas

como *matplotlib*, é complementada por mecanismos de interpretação e correção automáticos, permitindo iteração até à obtenção de um gráfico funcional (S.-S. Chen et al., 2025).

No seu conjunto, os modelos *Vision-to-Code* e os *pipelines* multimodais constituem uma revolução nos processos de design e desenvolvimento *frontend*. Ao integrar múltiplas fontes de informação e capacidades de inferência contextual, estas abordagens não só automatizam tarefas antes manuais, como também aumentam a precisão, escalabilidade e acessibilidade da produção de interfaces. A sua aplicação prática estende-se desde a prototipagem rápida até à criação de aplicações funcionais completas, demonstrando um elevado potencial de transformação digital em ambientes académicos, empresariais e de inovação.

2.5 LLM: Fundamentos e Evolução

Os *Large Language Models* (*LLM*) são sistemas avançados de inteligência artificial desenvolvidos para compreender, processar e gerar texto com características semelhantes às da linguagem humana. Baseiam-se em técnicas de *deep learning*, com destaque para a arquitetura *transformer*, que revolucionou o campo do processamento de linguagem natural (Ahmed Ali Linkon et al., 2024; Gholami, 2024; van Lier & Muñoz-Gil, 2024). O seu objetivo principal consiste em modelar e prever distribuições probabilísticas de sequências linguísticas, permitindo-lhes produzir texto coerente e contextualmente adequado (Castillo-Eslava et al., 2023; Shanahan & Singler, 2024). Estes modelos são treinados em grandes volumes de dados provenientes de fontes diversificadas, como livros, artigos e páginas *web*, o que lhes permite aprender padrões, estruturas e subtilezas da linguagem natural (Pakalapati et al., 2023; T. Zhou et al., 2024).

Com base em milhões, ou mesmo milhares de milhões de parâmetros, os *LLM* demonstram desempenho notável em tarefas linguísticas como geração de texto, tradução, sumarização e análise de sentimento (Chang, 2023; Mukherjee & Chang, 2023). Através de métodos de aprendizagem *auto-supervisionada* e *semi-supervisionada*, estes modelos internalizam estruturas linguísticas e são capazes de prever a próxima palavra de uma sequência, base essencial para a sua capacidade de geração textual (Barbon et al., 2024; W. Zhang et al., 2023). Esta competência permite-lhes produzir respostas relevantes e, por vezes, indistinguíveis das redigidas por humanos (Pangtey et al., 2025), representando um avanço tecnológico significativo com múltiplas aplicações em diversos setores (Khatiwada et al., 2025; Raffel et al., 2020).

Estes avanços conduziram à emergência de modelos de larga escala com capacidades inéditas de generalização. Contudo, a crescente complexidade destes sistemas também trouxe desafios importantes, nomeadamente relacionados com a sua opacidade, potencial para gerar desinformação e dificuldades de adaptação a domínios específicos. Em resposta a estas limitações, têm vindo a ser desenvolvidas abordagens inovadoras que procuram aumentar a transparência, a eficiência e a capacidade de explicação dos *LLM*.

Os *LLM* destacam-se pelo seu desempenho em múltiplas tarefas, sustentado por arquiteturas profundas e grandes volumes de dados. Técnicas como *few-shot learning*, *in-context learning* e *instruction following* potenciam a sua capacidade de generalização a partir de exemplos mínimos (Alayrac et al., 2022; Zheng et al., 2024). A introdução da arquitetura *transformer* por Vaswani et al. (2017) foi um marco fundamental, permitindo capturar dependências linguísticas de longo alcance e treinar modelos de forma mais eficiente (Alassan et al., 2024; Vaswani et al., 2017; White et al., 2023). Este avanço viabilizou a criação de modelos como BERT (S. Chen & Lin, 2024), GPT-3 (Brown et al., 2020) e variantes como ChatGPT, PaLM2 ou Vicuna (Jin et al., 2024; Zheng et al., 2024).

No cerne dos sistemas modernos de inteligência artificial, e em particular dos LLM, encontram-se os ***embeddings***, que são representações vetoriais de baixa dimensão que codificam a informação essencial de objetos — como palavras, documentos ou imagens — num formato numérico processável por máquinas (Haghir Chehreghani, 2024). A sua principal característica é a capacidade de preservar as similaridades e as relações semânticas e estruturais dos dados originais, posicionando objetos semelhantes em pontos próximos no espaço vetorial (Tyshchuk et al., 2023; Vlasic & Aguinaga, 2024). Esta transformação de dados não processados em representações numéricas compactas é fundamental para tarefas como classificação, recomendação e, crucialmente, para a compreensão da linguagem natural (Vlasic & Aguinaga, 2024)

Existem diferentes tipos de *embeddings*, especializados para diferentes tipos de dados. Os ***word embeddings***, como os popularizados pelos modelos **Word2Vec**, **GloVe** e **fastText**, transformam palavras discretas em vetores contínuos, capturando relações semânticas (Bojanowski et al., 2017). Esta técnica evoluiu de *embeddings* estáticos (um vetor por palavra) para *embeddings* contextuais, como os gerados pelo **BERT**, que produzem vetores diferentes para a mesma palavra dependendo do seu contexto (Abdullahi et al., 2024; Devlin et al., 2019). O conceito foi estendido para ***document embeddings*** (e.g., **Doc2Vec**) e para outras modalidades, como ***image embeddings***, gerados por modelos como **CLIP** para capturar conteúdo visual, e ***graph embeddings*** (e.g., **node2vec**) que codificam a estrutura de redes e grafos de conhecimento (A. Cai et al., 2023; Grover & Leskovec, 2016).

Apesar do seu potencial, os *LLM* enfrentam limitações significativas, como alucinações (respostas incorretas), desatualização do conhecimento, falta de interpretabilidade e dificuldades em lidar com domínios técnicos específicos (Edwards, 2024; Gan et al., 2025; Guo et al., 2023). A sua natureza de *caixa negra* compromete a confiança nos seus resultados (Pei et al., 2024; Zhao et al., 2024). Neste contexto, o paradigma *RAG* tem ganho destaque como solução promissora, ao combinar modelos com fontes externas de conhecimento para gerar respostas mais factuais, atuais e auditáveis (Y. Gao et al., 2023; Lewis et al., 2020)).

A arquitetura *RAG* integra três componentes principais: uma base de conhecimento externa, um sistema de recuperação semântica e o modelo gerador. Esta abordagem permite incorporar informação contextual relevante no momento da inferência, superando as limitações de modelos puramente paramétricos (Campo et al., 2025; Clop & Teglia, 2024).

Variantes como o *Modular RAG* e abordagens reflexivas têm vindo a otimizar o processo de recuperação e geração de respostas (Asai et al., 2024; Du et al., 2025).

Complementarmente, várias estratégias têm sido propostas para tornar os *LLM* mais transparentes e interpretáveis. Entre estas incluem-se o *Chain-of-Thought reasoning*, *In-Context Learning*, *Tool Learning*, *Reinforcement Learning with Human Feedback*, e técnicas de compressão como *quantization* e *pruning* (Dettmers et al., 2023; Pei et al., 2024; Vakali & Dimitriadis, 2025). A integração com *Knowledge Graphs* e arquiteturas multiagente tem reforçado a capacidade dos modelos em lidar com tarefas complexas e domínios especializados. A evolução dos *LLM*, associada a abordagens como o *RAG* e técnicas de alinhamento e especialização, tem conduzido a modelos mais transparentes, fiáveis e adaptáveis, com maior potencial de integração em contextos reais e exigentes (H. Chen et al., 2025; Y. Gao et al., 2025).

2.6 Fine-tuning vs. *Retrieval-Augmented Generation*

A adaptação dos *LLM* a contextos específicos pode ser realizada essencialmente através de duas abordagens distintas: o *fine-tuning* e a Geração Aumentada por Recuperação (*Retrieval-Augmented Generation – RAG*). O *fine-tuning* consiste na continuação do treino de um modelo já existente com dados especializados de um determinado domínio, com o objetivo de refinar o seu desempenho em tarefas específicas. Esta abordagem, embora eficaz na adaptação do estilo, tom e consistência das respostas, implica a gestão de uma nova versão do modelo, exigindo consideráveis recursos computacionais, tempo e dados de elevada qualidade (Andreev et al., 2024; Barnett et al., 2024).

Por outro lado, os sistemas *RAG* integram mecanismos de recuperação de informação com modelos de linguagem, permitindo que o modelo aceda dinamicamente a conteúdos externos relevantes durante o processo de geração. Em vez de alterar os parâmetros do modelo base, o *RAG* complementa as suas capacidades com fontes de conhecimento atualizadas, assegurando maior precisão factual e flexibilidade (Banerjee et al., 2024; M. Gao et al., 2024). Este modelo é particularmente útil em contextos onde o conhecimento se encontra em constante evolução, como em aplicações empresariais ou técnicas, onde os dados e documentos de referência são frequentemente atualizados.

O *fine-tuning* continua a ser valorizado em cenários onde a consistência do estilo e da linguagem é crítica, como na produção de conteúdo técnico com vocabulário muito específico ou para manter coerência numa determinada linha editorial (Harbola & Purwar, 2025). Para tarefas como geração de código em linguagens de programação visuais ou *DSL*, o *fine-tuning* tem demonstrado bom desempenho, mesmo utilizando modelos de menor dimensão (Kang et al., 2025). Contudo, a sua aplicação generalizada é limitada devido às exigências computacionais e ao custo associado à reconfiguração e manutenção dos modelos (Andreev et al., 2024; D. Li et al., 2023).

Em contraste, os sistemas *RAG* destacam-se na incorporação de novos conhecimentos sem necessidade de treino adicional. Isto permite uma maior escalabilidade e adaptabilidade, além de facilitar a atualização contínua dos conteúdos de suporte. O desempenho de um sistema *RAG* depende fortemente da qualidade do seu mecanismo de recuperação, sendo essencial a curadoria de uma base de dados bem estruturada e semanticamente rica (Soudani et al., 2024). Em domínios com menos recursos disponíveis, combinar modelos mais pequenos com *RAG* tem mostrado resultados competitivos ou mesmo superiores aos obtidos por modelos maiores exclusivamente *fine-tuned* (Soudani et al., 2024).

Para além das abordagens tradicionais, têm sido propostos modelos híbridos que procuram combinar o melhor de ambos os mundos. O *DRAFT*, por exemplo, integra geração aumentada por recuperação com *fine-tuning*, conseguindo melhorias notáveis na geração de decisões de design (Dhar et al., 2025). Também em tarefas como geração de código com *API* não vistas anteriormente, os modelos *RAG* têm-se mostrado tão eficazes quanto o *fine-tuning*, especialmente no controlo de alucinações e na melhoria da robustez das respostas (Fan et al., 2024; Zitouni et al., 2024).

A nível de implementação, os requisitos técnicos das duas abordagens diferem significativamente. O *fine-tuning* exige o acesso ao modelo base, bem como capacidade computacional para realizar o treino com os novos dados. Métodos como o *Textual Inversion* e o *DreamBooth*, por exemplo, requerem milhares de passos de treino, mesmo em hardware avançado como as *GPU A100* (D. Li et al., 2023). Como resposta a estas exigências, surgiram técnicas mais eficientes como o *Low-Rank Adaptation* (*LoRA*), que reduz o número de parâmetros treináveis, facilitando adaptações leves e rápidas sem comprometer significativamente o desempenho (Harbola & Purwar, 2025).

Já os sistemas *RAG* operam de forma modular, integrando uma etapa de recuperação com o modelo base. O fluxo habitual inclui a formulação da consulta, a pesquisa por conteúdos relevantes numa base de dados indexada, a combinação dos trechos encontrados com o *prompt* do utilizador, e a geração da resposta final pelo *LLM* (M. Gao et al., 2024; Yang et al., 2024). Esta arquitetura modular permite que modelos já treinados sejam utilizados sem necessidade de alterações, o que reduz drasticamente os custos de implementação e manutenção.

Contudo, a eficácia do *RAG* está condicionada pela capacidade de o modelo integrar adequadamente a informação recuperada. Quando o modelo base não foi treinado com dados estruturados para incorporar documentos externos, os outputs podem ser menos fundamentados. Para mitigar este problema, têm surgido abordagens como o *Retrieval-Augmented In-context Tuning* (*RAIT*) e *frameworks* como o *DeepThink*, que reforçam a ligação entre o conteúdo recuperado e a geração da resposta (Y. Li et al., 2025; Srinivas et al., 2024).

No contexto da geração de código, especialmente em domínios como o desenvolvimento *frontend* ou engenharia de *software*, a utilização de *RAG* tem demonstrado particular eficácia. Modelos como o *RepoCoder*, por exemplo, utilizam recuperação baseada em similaridade para aceder a componentes de repositórios e melhorar a geração de código em contexto (Bi et al., 2024; F. Zhang et al., 2023). Esta abordagem é particularmente útil na

geração de código a partir de *mockups*, pois permite que o sistema recupere padrões de interface e convenções específicas da empresa.

A vantagem do *RAG* nestes casos reside na sua capacidade de alinhar o output do modelo com a base de conhecimento já existente na organização, respeitando os componentes e estilos definidos no repositório de código. Isto é especialmente importante quando se pretende gerar código com coerência estrutural e sem alucinações (Tinnes et al., 2024). Embora o *fine-tuning* também possa alcançar esse nível de integração, requer um *dataset* representativo e completo, além de custos significativos de treino.

Estudos recentes indicam que, para aplicações de geração de código com necessidades altamente contextuais, como a utilização de *API* internas ou componentes reutilizáveis, o *RAG* revela-se uma solução mais eficiente e escalável. A sua modularidade permite adaptações rápidas e específicas sem comprometer a generalização do modelo. Por outro lado, o *fine-tuning* permanece relevante para casos em que é necessário um controlo rigoroso sobre o estilo, vocabulário e consistência do modelo.

Para sintetizar e visualizar as diferenças fundamentais entre estas duas abordagens, a tabela seguinte apresenta uma análise comparativa dos seus principais atributos:

Tabela 2 – Comparação entre *Fine-Tuning* e *RAG*

Critério de Comparação	***Fine-Tuning***	***RAG***
Mecanismo Principal	Altera os pesos internos do modelo através de treino adicional com um *dataset* específico.	Aumenta o *prompt* do utilizador com informação relevante recuperada de uma fonte externa no momento da inferência.
Objetivo Primário	Adaptar o estilo, tom, comportamento e vocabulário do modelo a um domínio específico.	Aumentar a precisão factual, combater "alucinações" e fornecer acesso a conhecimento atualizado.
Gestão do Conhecimento	Internaliza o conhecimento no modelo (memória paramétrica). O conhecimento é estático após o treino.	Utiliza uma base de conhecimento externa e dinâmica (memória não-paramétrica) que pode ser atualizada facilmente.
Custo e Recursos	Elevado custo computacional e de tempo para treino. Exige a gestão de novas versões do modelo.	Custo de inferência ligeiramente superior, mas sem os custos elevados de treino. Mais eficiente em termos de recursos.
Manutenção e Escalabilidade	Complexo de manter e atualizar. Cada atualização de conhecimento requer um novo ciclo de fine-tuning.	Elevada escalabilidade e facilidade de manutenção. A base de conhecimento pode ser atualizada de forma independente.
Requisitos de Dados	Requer um *dataset* de elevada qualidade e dimensão considerável para ser eficaz.	Requer uma base de dados externa bem estruturada e indexada para o sistema de recuperação (*retriever*).
Principal Vantagem	Elevada consistência no estilo e comportamento. Excelente para tarefas de adaptação de personalidade ou formato.	Elevada precisão factual e transparência. Capacidade de citar fontes e usar dados em tempo real.
Principal Desafio	Risco de "esquecimento catastrófico" (perda de conhecimento geral) e elevados custos operacionais.	A qualidade da resposta depende criticamente da eficácia e relevância do sistema de recuperação.

Em suma, tanto o *fine-tuning* como o *RAG* apresentam vantagens e limitações que devem ser avaliadas consoante os objetivos da aplicação, os recursos disponíveis e o grau de

atualização e especialização da base de conhecimento. Em muitos cenários, uma combinação estratégica de ambas as abordagens pode oferecer resultados superiores, ao conjugar a adaptabilidade do *fine-tuning* com a atualidade e precisão factual proporcionadas pela recuperação aumentada.

2.7 Personalização facilitada de modelos de *IA*

O avanço das tecnologias de Inteligência Artificial aplicadas ao desenvolvimento de *software* tem vindo a promover a democratização do acesso à criação de interfaces digitais. Em particular, a capacidade de gerar código *frontend* a partir de *mockups* com recurso a modelos de *IA* está a tornar-se cada vez mais acessível a utilizadores sem conhecimentos técnicos especializados, através da integração de mecanismos de personalização adaptados a perfis não programadores. Esta evolução visa colmatar a tradicional distância entre o design visual e a implementação funcional, permitindo que indivíduos com pouca ou nenhuma experiência em codificação consigam transformar ideias em soluções interativas, recorrendo a modelos de *IA* ajustáveis às suas necessidades específicas.

Contudo, a adaptação destes modelos para públicos não técnicos continua a enfrentar obstáculos relacionados com a complexidade subjacente das arquiteturas envolvidas e com a falta de acessibilidade das interfaces de controlo. Para ultrapassar estes desafios, têm surgido diferentes estratégias que combinam interfaces visuais, técnicas de programação por demonstração, personalização baseada em linguagem natural e ensino interativo de máquinas. Ferramentas que integram paradigmas visuais, como a programação por blocos ou fluxos de dados, facilitam significativamente o envolvimento de utilizadores iniciantes, permitindo-lhes manipular componentes e estruturas lógicas sem escrever código de forma explícita (D. C.-E. Lin & Martelaro, 2023). Estas abordagens visuais, quando conjugadas com interfaces em linguagem natural e *feedback* imediato, contribuem para sistemas mais vivos, intuitivos e acessíveis (Rao et al., 2023).

Plataformas de desenvolvimento *low-code* e *no-code* têm sido particularmente eficazes neste domínio, ao permitirem a criação de aplicações baseadas em modelos de *IA* através de componentes gráficos de arrastar e largar, dispensando o utilizador da necessidade de escrever código. Ferramentas deste tipo, incluindo sistemas *AutoML* com suporte gráfico, têm sido amplamente utilizadas em contextos empresariais e educacionais, promovendo a adoção alargada da *IA* por utilizadores com diferentes níveis de literacia técnica (Esposito et al., 2023, 2025).

Para além das interfaces visuais, técnicas de programação por exemplo (*Programming By Example* (*PBE*)) e por demonstração (*Programming by Demonstration* (*PBD*)) têm vindo a revelar-se eficazes na capacitação de utilizadores para configurarem ou ensinarem o comportamento esperado a sistemas de *IA*. Na *PBE*, o utilizador fornece exemplos de entrada e saída desejados, permitindo ao sistema inferir regras ou programas que satisfaçam esses padrões (Sarkar, 2023). Já na *PBD*, os utilizadores demonstram ações em tempo real, e o sistema

aprende essas sequências como tarefas automatizáveis, com potencial para generalização (T. J.-J. Li et al., 2017, 2018). Estas abordagens são particularmente úteis na construção de interfaces ou fluxos de interação personalizados sem intervenção direta sobre o código-fonte.

Uma vertente complementar reside no ensino interativo de máquinas, onde o utilizador assume o papel de "professor", guiando o modelo na aprendizagem de uma tarefa específica com base em exemplos e *feedback* iterativo. Ferramentas como o *Teachable Machine* demonstram a viabilidade desta abordagem ao permitirem a criação rápida de modelos de classificação sem necessidade de codificação, através de demonstrações simples e interfaces *web* acessíveis (Carney et al., 2020; Monteiro et al., 2023). Estas tecnologias têm sido aplicadas com sucesso em contextos como educação, design de brinquedos interativos, visualização de dados sensoriais e prototipagem de interfaces baseadas em gestos e movimentos (Jordan et al., 2021; Tseng et al., 2021).

O controlo sobre os modelos de *IA* também tem evoluído para incorporar mecanismos mais transparentes e adaptativos, permitindo que os utilizadores ajustem parâmetros como originalidade, estrutura e tipo de saída gerada. Estes mecanismos podem ser disponibilizados através de *sliders*, desenhos, comandos de voz ou *sketching* visual, promovendo um maior controlo sobre a personalização do modelo e a geração iterativa de resultados (Chung et al., 2022; Chung & Adar, 2023; Rajaram et al., 2024). Exemplos como o *TaleBrush* demonstram como esboços visuais podem ser utilizados para guiar a geração textual, ideia que pode ser facilmente transposta para o domínio da geração de código *frontend* (Chung et al., 2022).

No domínio da prototipagem de interfaces, soluções como o *Misty* permitem a integração de exemplos visuais como *screenshots* ou esboços diretamente nas interfaces em construção, com suporte a manipulação direta e visualização semântica das alterações propostas (Y. Lu, Leung, et al., 2024). A personalização de modelos de geração de código torna-se assim mais acessível, à medida que as ferramentas passam a incluir modos de configuração baseados em preferências do utilizador, *feedback* interativo e sugestões iterativas baseadas em histórico de interações.

Por fim, o crescimento das plataformas *no-code* com foco na geração de código a partir de notas visuais e *mockups* está a reduzir substancialmente as barreiras à adoção da *IA* em contextos não técnicos (Jeong, 2025). Estes sistemas permitem não apenas a geração direta de interfaces funcionais, mas também a sua adaptação contínua em ciclos iterativos com a colaboração de utilizadores finais, designers e especialistas de domínio.

Para sintetizar as diferentes abordagens que facilitam a personalização de modelos de IA por parte de utilizadores com perfis não técnicos, a tabela seguinte apresenta uma comparação dos seus mecanismos, vantagens e casos de uso típicos:

Tabela 3 - Comparação de Abordagens para Personalização Facilitada de Modelos de IA

Abordagem	Mecanismo Principal	Nível de Abstração	Vantagens Principais	Casos de Uso Típicos
Plataformas *Low-code/No-code*	Utilização de interfaces gráficas, com componentes de arrastar e largar (*drag-and-drop*) e fluxos visuais.	Muito Alto	Elimina a necessidade de codificação, democratiza o acesso e acelera a prototipagem.	Criação de aplicações simples, automatização de fluxos de trabalho, *dashboards* de dados.
PBE	O utilizador fornece pares de entrada/saída, e o sistema infere as regras ou transformações lógicas.	Alto	Intuitivo, não requer conhecimento de algoritmos, focado no resultado desejado.	Transformação de dados em folhas de cálculo, extração de informação de texto, formatação de ficheiros.
PBD	O sistema observa e regista as ações do utilizador na interface gráfica para automatizar tarefas repetitivas.	Alto	Aprendizagem natural através da observação, ideal para automatizar fluxos de interação na UI.	Criação de *macros*, automação de testes de *UI*, personalização de fluxos de trabalho em *software*.
Ensino Interativo de Máquinas	O utilizador atua como "professor", fornecendo exemplos e feedback iterativo em tempo real para treinar um modelo de classificação ou regressão.	Médio	Controlo direto e granular sobre o processo de treino, feedback imediato, facilita a compreensão do comportamento do modelo.	Classificação de imagens ou sons, prototipagem de interfaces baseadas em gestos, sistemas de recomendação simples.
Personalização por Linguagem Natural	O utilizador descreve os requisitos ou ajustes desejados através de comandos em linguagem natural (texto ou voz).	Médio	Elevada flexibilidade e expressividade, interação conversacional e natural.	Geração de código a partir de descrições, ajuste de parâmetros de modelos de *IA* generativa, criação de *bots*.

Em síntese, a personalização facilitada de modelos de *IA* para geração de código *frontend* encontra-se numa fase de rápida maturação, suportada por um ecossistema de ferramentas que combinam programação visual, exemplos, demonstração e *feedback* guiado. Estas abordagens não apenas permitem expandir o leque de utilizadores que podem tirar partido de tecnologias avançadas, como também contribuem para acelerar o processo de design e desenvolvimento, respeitando as particularidades de cada utilizador ou organização.

2.8 Plataformas de Treino e Implementação de Modelos de *IA*

O desenvolvimento e aplicação de modelos de Inteligência Artificial na geração automática de interfaces gráficas exige uma infraestrutura técnica adequada para treino, afinação (*fine-tuning*) e implementação. Neste contexto, diversas plataformas foram concebidas para apoiar tanto utilizadores com conhecimento técnico avançado como perfis menos especializados, oferecendo níveis variáveis de controlo, flexibilidade e usabilidade.

Uma das abordagens mais completas e personalizáveis consiste na utilização de plataformas que permitem o aluguer de máquinas virtuais com especificações detalhadas, incluindo o número de *CPU*, *GPU* (como *NVIDIA A100*, *V100* ou *L4*), memória *RAM* e armazenamento. Exemplos incluem serviços como *Amazon SageMaker* (Karakus et al., 2021), *Google Vertex AI* (Schuller et al., 2022), *Microsoft Azure Machine Learning* (Kanhirakadavath & Chandran, 2022), *Paperspace* (PaperSpace, 2025) e *RunPod* (RunPod, 2025). Estas plataformas permitem acesso direto a ambientes como *Jupyter Notebooks*, onde o utilizador pode executar código *Python*, treinar modelos, monitorizar métricas e personalizar o *pipeline* de *IA* em detalhe. Esta abordagem é ideal para projetos de larga escala, mas exige conhecimento técnico

avançado em *frameworks* como *PyTorch*, *TensorFlow, Keras* ou *Caffe*, além de representar um investimento financeiro elevado, dado que os custos são calculados com base no tempo de uso da infraestrutura computacional (Bovenzi et al., 2025; Khoramnejad & Hossain, 2024).

Esses *frameworks* fundamentais, nomeadamente *TensorFlow* e *PyTorch*, representam os dois ecossistemas dominantes que sustentam grande parte do desenvolvimento em IA. O ***TensorFlow***, desenvolvido pela *Google Brain*, é amplamente adotado em ambientes empresariais devido ao seu robusto suporte para treino distribuído, escalabilidade em produção e compatibilidade com múltiplos dispositivos, desde servidores a plataformas móveis (Brendel et al., 2018; X. Chen et al., 2017). Utiliza um estilo de programação declarativo com otimização baseada em grafos, o que oferece grande flexibilidade para implementações em larga escala (Magadza & Viriri, 2021; Xu et al., 2021). Em contraste, o ***PyTorch***, criado pela equipa de investigação de IA do Facebook (*FAIR*), goza de elevada popularidade na comunidade académica e de investigação (Younesi et al., 2024; M. Zhou et al., 2024). A sua principal distinção é o uso de um grafo computacional dinâmico com execução "ansiosa" (*eager execution*), o que proporciona um desenvolvimento mais intuitivo, maior flexibilidade para trabalhar com arquiteturas complexas e uma depuração mais simples, tornando-o particularmente adequado para a experimentação (Mungoli, 2023; Shoaib et al., 2023; Xu et al., 2021).

Para além destes *frameworks* de baixo nível, existem API de alto nível e ferramentas especializadas que simplificam o processo de desenvolvimento. O ***Keras***, por exemplo, funciona como uma *API* de alto nível, agora profundamente integrada no ecossistema *TensorFlow*, que se foca em minimizar as ações do utilizador e permitir uma rápida experimentação, sendo ideal para iniciantes (Asghar et al., 2019; H. Li et al., 2021; Magadza & Viriri, 2021). Outros *frameworks*, como o ***Caffe***, foram desenvolvidos com um foco mais específico, destacando-se pela sua velocidade e eficiência em tarefas de processamento de imagem, embora o seu suporte para modelagem de linguagem seja menos robusto (H. Li et al., 2021). Embora estas ferramentas dominem o panorama, outras opções como **Microsoft's Cognitive Toolkit** (***CNTK***) e **Apache MXNet** oferecem vantagens para requisitos específicos, sendo a escolha final dependente do contexto do projeto (Kumar et al., 2021; Mungoli, 2023; Thon et al., 2021).

Para utilizadores com menor domínio técnico, têm surgido soluções *no-code* ou *low-code* que simplificam substancialmente o processo de treino e afinação de modelos. Estas plataformas geralmente oferecem interfaces gráficas baseadas em *drag-and-drop*, ou fluxos guiados onde o utilizador apenas necessita de carregar os dados (em formatos como *.jsonl* ou *.csv*), escolher o modelo-base e iniciar o processo de *fine-tuning*. Um exemplo proeminente nesta categoria é a *OpenAI API*, que fornece um *playground* intuitivo onde é possível testar modelos da série *GPT* (como o *GPT*-4), realizar *fine-tuning* de modelos *text-to-text* e integrar *embeddings* em aplicações via chamadas *API* (Gürcan, 2024). A limitação desta plataforma reside no seu foco exclusivo nos modelos da *OpenAI*, o que restringe a experimentação com arquiteturas alternativas (Bovenzi et al., 2025).

Uma alternativa mais flexível e recente é o *Nebius AI Studio*, que oferece um ecossistema completo para treino e inferência com suporte a *LLM*, tais como *Qwen*, *LLaMA*, *DeepSeek*, *Phi*, *Gemma*, *Hermes* e *Mistral*. Esta plataforma permite o treino de modelos do tipo *text-to-text* com *fine-tuning*, bem como a implementação de *RAG*. Além disso, disponibiliza modelos para tarefas de *embeddings*, text-to-image, vision, e safety guardrails. Tal como a *OpenAI*, também oferece um *playground* e uma *API REST* acessível para programadores em várias linguagens, permitindo a rápida integração dos modelos em aplicações reais, sem necessidade de gerir infraestrutura (W. Cai et al., 2025).

Além das duas plataformas mencionadas (*OpenAI API* e *Nebius AI Studio*), existem outras soluções relevantes no mercado, como:

- ***Hugging Face* Hub**: plataforma comunitária com milhares de modelos prontos a usar, oferece *AutoTrain* (interface *low-code* para *fine-tuning*) e integração com *Transformers*, *Gradio* e *Datasets*. Permite treinar localmente ou na *cloud* (incluindo com serviços como *AWS* e *GCP*) (Thakur, 2024).

- **Replicate**: fornece uma forma simples de correr modelos *open-source* como *Stable Diffusion*, *LLaMA*, *Whisper*, etc., através de *API* acessíveis. É orientado a *deploy*, mas também permite afinar modelos com *datasets* próprios (Replicate, 2025).

- **Weights & Biases**: plataforma de *Machine Learning Operations* (*MLOps*) usada para rastreamento de experiências, visualização de métricas, e organização de *pipelines* de treino e *deploy*, frequentemente usada em combinação com outras plataformas de computação (Patel et al., 2024).

- ***Anthropic*'s *Claude API***: permite integração direta com modelos competitivos como o *GPT*, embora o acesso a *fine-tuning* seja mais limitado (Amazon, 2024).

- **OpenRouter.ai**: atua como camada de abstração que permite alternar entre múltiplos provedores de *LLM* (como *Anthropic*, *Cohere*, *Mistral*, *Groq*, etc.), com uma *API* unificada (OpenRouter, 2025).

Estas plataformas permitem às equipas de desenvolvimento selecionar entre treinos customizados com elevado grau de controlo e soluções mais acessíveis e rápidas para integrar *IA* em aplicações reais. A escolha da plataforma ideal depende do nível de conhecimento técnico disponível, dos requisitos de personalização do projeto, da sensibilidade dos dados (que podem exigir ambientes isolados ou *self-hosted*) e do orçamento disponível.

Tal como podemos observar na Tabela 1 abaixo, as soluções variam substancialmente em termos de complexidade técnica, flexibilidade e variedade de modelos suportados. Desde ambientes especializados com acesso direto à infraestrutura computacional até plataformas *no-code* com interfaces intuitivas para *fine-tuning* e/ou *RAG*, a diversidade de abordagens disponíveis permite que organizações com diferentes perfis técnicos encontrem opções adequadas para os seus objetivos de integração de *IA* na geração automática de interfaces.

Tabela 4 - Comparação entre plataformas de treino e implementação de modelos de *IA*

Ferramenta	Conhecimento Técnico	*Fine-Tuning*	Suporte a *RAG*	Variedade de Modelos	Integração via *API*	Custo por Uso	Custo por Tempo de Atividade
Amazon SageMaker	Sim	Sim	Sim	Sim	Sim	Não	Sim
Google Vertex AI	Sim	Sim	Sim	Sim	Sim	Não	Sim
Microsoft Azure ML	Sim	Sim	Sim	Sim	Sim	Não	Sim
Paperspace	Sim	Sim	Sim	Sim	Sim	Não	Sim
RunPod	Sim	Sim	Sim	Sim	Sim	Não	Sim
OpenAI API	Não	Sim	Sim	Não	Sim	Sim	Não
Nebius AI Studio	Não	Sim	Sim	Sim	Sim	Sim	Não
***Hugging Face* Hub**	Sim	Sim	Sim	Sim	Sim	Sim	Não
Replicate	Não	Sim	Sim	Sim	Sim	Sim	Não
Weights & Biases	Sim	Sim	Sim	Sim	Sim	Sim	Não
Anthropic Claude API	Não	Não	Sim	Não	Sim	Sim	Não
OpenRouter.ai	Não	Não	Sim	Sim	Sim	Sim	Não

Em suma, a crescente variedade de plataformas para treino e implementação de modelos de *IA* democratizou o acesso a tecnologias avançadas, permitindo que tanto especialistas quanto utilizadores menos técnicos explorem soluções inteligentes para a geração de interfaces gráficas, adaptadas aos seus contextos específicos.

2.9 Métricas de avaliação de código *UI*

A avaliação do código *frontend* gerado automaticamente a partir de *mockups* representa uma componente crítica no desenvolvimento de sistemas baseados em Inteligência Artificial para a criação de interfaces gráficas. Este processo exige uma abordagem abrangente que contemple múltiplas dimensões, desde a fidelidade visual até à qualidade estrutural do código, passando pelo desempenho, acessibilidade e reutilização de componentes. Tais avaliações são fundamentais para aferir a eficácia das soluções propostas, identificar deficiências nos sistemas de geração automática e orientar o seu aprimoramento contínuo.

A fidelidade visual constitui uma das métricas mais utilizadas para verificar o grau de correspondência entre a interface gerada e o *mockup* original. Este aspeto é geralmente medido através de métricas objetivas como o ***Structural Similarity Index Measure*** (*SSIM*), o ***Peak Signal-to-Noise Ratio*** (*PSNR*) e o ***Mean Squared Error*** (*MSE*), que comparam imagens renderizadas com as de referência, captando diferenças estruturais, de contraste e luminância (Xiao et al., 2024). No entanto, a avaliação baseada apenas em pixels pode ser limitada, pelo que têm sido introduzidas métricas de similaridade semântica, como o *CLIP score*, uma métrica derivada do modelo ***CLIP*** *(Contrastive Language–Image Pretraining)* da *OpenAI*, que combina capacidades de visão computacional com compreensão de linguagem natural. Este modelo atua como um classificador de imagens baseado em texto, permitindo quantificar a correspondência entre uma representação visual e a sua descrição textual (Hessel et al., 2021a). De forma complementar, têm também sido explorados *embeddings* visuais obtidos por modelos como o ***DINOv2-base***, que possibilitam comparar interfaces com base na sua representação semântica, aproximando-se da perceção humana (Ge et al., 2025; T. Zhou et al., 2024). Adicionalmente, a

deteção automatizada de erros visuais tem demonstrado utilidade na identificação de inconsistências subtis, contribuindo para prever a satisfação dos utilizadores com base em fatores percetivos (Y. Lu, Yang, et al., 2024). Para sistematizar estas abordagens, a tabela seguinte resume as principais métricas, destacando o seu foco, mecanismo e os seus respetivos pontos fortes e limitações:

Tabela 5 - Métricas para Avaliação da Fidelidade Visual

Métrica / Técnica	Foco Principal	Como Funciona	Vantagens	Limitações
MSE *(Mean Squared Error)*	Erro ao nível do píxel (Anwar & Li, 2020)	Calcula a média dos quadrados das diferenças entre os píxeis correspondentes de duas imagens (Anwar & Li, 2020).	Simples de calcular e implementar (Anwar & Li, 2020).	Baixa correlação com a perceção visual humana; penaliza muito pequenas variações (Anwar & Li, 2020).
PSNR *(Peak Signal-to-Noise Ratio)*	Qualidade da imagem (ruído) (Huang et al., 2024)	Mede a razão entre a potência máxima de uma imagem e o ruído que a afeta, com base no *MSE* (Huang et al., 2024).	Métrica padrão em compressão de imagem, fácil de calcular (Huang et al., 2024).	Tal como o *MSE*, não reflete bem a qualidade visual percebida (Huang et al., 2024).
SSIM *(Structural Similarity Index)*	Similaridade Estrutural (Chatzitofis et al., 2020)	Compara luminância, contraste e estrutura entre duas imagens, imitando a perceção visual humana (Chatzitofis et al., 2020).	Melhor correlação com o julgamento humano do que o *PSNR/MSE (Chatzitofis et al., 2020)*.	Ainda é sensível a translações e rotações; não compreende o conteúdo (Chatzitofis et al., 2020).
CLIP Score	Correspondência Semântica Texto-Imagem (Hessel et al., 2021b)	Usa o modelo *CLIP* para medir quão bem uma imagem corresponde a uma descrição textual (Hessel et al., 2021b).	Avalia o conteúdo e a semântica, transcendendo a mera comparação de píxeis (Hessel et al., 2021b).	Depende da qualidade e especificidade do *prompt* textual para uma avaliação precisa (Hessel et al., 2021b).
Embeddings Visuais *(DINOv2)*	Similaridade Semântica Imagem-Imagem (Marjit et al., 2025)	Converte cada interface num vetor (*embedding*) que captura o seu significado visual para comparação direta (Marjit et al., 2025).	Permite comparações diretas sem necessidade de texto; captura o "significado" visual da interface (Marjit et al., 2025).	A interpretação dos *embeddings* pode ser abstrata; a qualidade depende do modelo de base (Marjit et al., 2025).
Deteção Automática de Erros Visuais	Identificação de Defeitos de *UI/UX* (Ozcinar & Rana, 2021)	Utiliza algoritmos de visão computacional para detetar inconsistências (e.g., desalinhamentos, sobreposições) (Ozcinar & Rana, 2021).	Automatiza a deteção de *bugs* visuais, ajudando a prever a satisfação do utilizador (Ozcinar & Rana, 2021).	Pode gerar falsos positivos e requer configuração para diferentes tipos de interfaces (Ozcinar & Rana, 2021).

No que respeita à qualidade estrutural do código gerado, as métricas tradicionais de avaliação textual, como ***BLEU***, ***ROUGE***, ***METEOR*** e ***ChrF***, revelam-se insuficientes por não captarem propriedades semânticas e sintáticas específicas do código. Para colmatar essa lacuna, métricas como o ***CodeBLEU*** têm sido desenvolvidas, incorporando árvores de sintaxe abstrata (*AST*) na análise, permitindo avaliar não apenas a similitude textual, mas também a conformidade estrutural e lógica do código (Chemnitz et al., 2023; J. Lu et al., 2024). Paralelamente, a **análise estática de código** oferece uma via eficaz para examinar o código sem necessidade de execução, avaliando correção sintática, legibilidade, redundância, estilo de nomenclatura, **complexidade ciclomática** e grau de acoplamento entre componentes (Di et al.,

2023). Estas métricas são cruciais em contextos industriais, onde a legibilidade e a manutenibilidade do código impactam diretamente na produtividade e sustentabilidade do projeto. A tabela seguinte resume as principais métricas e técnicas utilizadas para avaliar a qualidade estrutural do código gerado por sistemas de IA:

Tabela 6 - Métricas para Avaliação da Qualidade Estrutural do Código

Métrica / Técnica	Foco Principal	Como Funciona	Vantagens	Limitações
***BLEU** (Bilingual Evaluation Understudy)*	Similaridade Textual (n-gramas) (Evtikhiev et al., 2023b)	Compara n-gramas de palavras do código gerado com referências (Evtikhiev et al., 2023b).	Simples e rápido de calcular (Evtikhiev et al., 2023b).	Ignora a sintaxe e penaliza variações válidas de código (Evtikhiev et al., 2023b).
***ROUGE** (Recall-Oriented Understudy for Gisting Evaluation)*	Similaridade Textual (*recall*) (Zhuo, 2024)	Mede n-gramas focando-se na inclusão de elementos da referência (Zhuo, 2024).	Útil para avaliar a completude do código (Zhuo, 2024).	Não compreende a semântica do código, tal como o *BLEU (Zhuo, 2024)*.
***METEOR** (Metric for Evaluation of Translation with Explicit ORdering)*	Similaridade Textual e Semântica (Zhuo, 2024)	Alinha palavras com base em sinónimos e radicais, penalizando fragmentação (Zhuo, 2024).	Mais flexível que o *BLEU*; melhor correlação com o julgamento humano (Zhuo, 2024).	Ainda é baseada em texto, sem analisar a estrutura do código (Zhuo, 2024).
***ChrF** (Character n-gram F-score)*	Similaridade a Nível de Caracteres (Evtikhiev et al., 2023b)	Calcula a *F-score* da sobreposição de n-gramas de caracteres (Evtikhiev et al., 2023b).	Robusto a erros morfológicos e eficaz para palavras raras (Evtikhiev et al., 2023b).	Ignora a estrutura de palavras e a semântica (Evtikhiev et al., 2023b).
CodeBLEU	Qualidade Estrutural e Semântica (Evtikhiev et al., 2023b)	Combina *BLEU* com a análise da Árvore de Sintaxe Abstrata (*AST*) (Evtikhiev et al., 2023b).	Avalia a estrutura sintática e lógica, não apenas o texto (Evtikhiev et al., 2023b).	Mais complexo de implementar e computacionalmente exigente que o *BLEU (Evtikhiev et al., 2023b)*.
Análise Estática de Código	Qualidade e Manutenibilidade (Alikhashashneh et al., 2018)	Analisa o código sem o executar para detetar erros e violações de estilo (Alikhashashneh et al., 2018).	Identifica problemas de legibilidade, complexidade e manutenção (Alikhashashneh et al., 2018).	Não avalia a funcionalidade ou a correção lógica em tempo de execução (Alikhashashneh et al., 2018).
Complexidade Ciclomática	Complexidade Lógica (Heitz et al., 2024)	Mede o número de caminhos lógicos independentes no código (Heitz et al., 2024).	Métrica quantitativa da complexidade que ajuda a identificar código a corrigir (Heitz et al., 2024).	Não mede outros aspetos da qualidade, como a clareza ou eficiência (Heitz et al., 2024).

A **avaliação do desempenho** da interface gerada é igualmente essencial, uma vez que o tempo de carregamento, a eficiência de renderização e a responsividade a diferentes tamanhos de ecrã influenciam de forma decisiva a experiência do utilizador. Neste âmbito, ferramentas automáticas como o ***Lighthouse*** podem ser integradas em *pipelines* de avaliação para fornecer métricas quantitativas e comparáveis entre soluções (Agbozo, 2023). A **acessibilidade**, por sua vez, é avaliada segundo normas internacionais como as ***WCAG*** **e** ***ISO/IEC 40500*** (Ara et al., 2024), verificando se o código cumpre requisitos como contraste adequado, uso de texto alternativo, navegação por teclado e compatibilidade com tecnologias de apoio. A inclusão destas métricas não apenas assegura o cumprimento de requisitos legais e éticos,

como também promove uma experiência mais inclusiva para todos os utilizadores (Agbozo, 2023).

Outra métrica fundamental é a **Taxa de Sucesso de Compilação** (*CSR*), que avalia se o código gerado é tecnicamente viável, isto é, se compila com sucesso, carrega corretamente e corresponde visualmente à interface esperada. Esta métrica é particularmente relevante em cenários mais complexos, como a geração de aplicações móveis completas, onde a mera aparência visual não é suficiente para validar a funcionalidade da aplicação (T. Zhou et al., 2024). Complementarmente, a **análise de cobertura funcional**, como a verificação da lógica de navegação e interatividade com base em grafos de transição de páginas (*PTG*), permite aferir o alinhamento entre o comportamento da interface gerada e os fluxos previstos no design.

Por fim, a **avaliação da reutilização de componentes** assume particular importância na análise da qualidade interna do código. Um bom sistema de geração de *UI* deve ser capaz de identificar padrões visuais recorrentes e estruturar o código de forma modular, reutilizando componentes e evitando redundância. Esta capacidade pode ser quantificada pela taxa de reutilização de componentes e pela análise da profundidade e coesão da árvore de componentes criada. A consistência na aplicação de padrões de design e a implementação de boas práticas de responsividade, como o uso de *Flexbox* ou *Grid*, reforçam a qualidade e a escalabilidade do código gerado (Papamichail et al., 2019).

Em suma, a avaliação do código *UI* gerado automaticamente por sistemas de *IA* exige um conjunto diversificado de métricas, que abranjam a fidelidade visual, a qualidade estrutural do código, o desempenho, a acessibilidade, a compilação e a reutilização de componentes. A integração destas métricas num *pipeline* automatizado permite uma avaliação objetiva e sistemática, fornecendo indicadores cruciais para a melhoria dos sistemas de geração automática e garantindo que as interfaces produzidas não só respeitam os *mockups* de origem, como são funcionais, acessíveis, eficientes e escaláveis.

2.10 Riscos éticos, segurança e governança

A geração automática de código *frontend* a partir de *mockups* com *IA* levanta preocupações relevantes em torno da ética, segurança e governança. Estas questões tornam-se centrais à medida que a tecnologia se integra nos processos de desenvolvimento, exigindo uma abordagem crítica e responsável. Ferramentas como o *GPT*-4, *GitHub Copilot* ou sistemas especializados de *AI-to-code* já demonstram impactos relevantes nos fluxos de trabalho de equipas de desenvolvimento (*AI's Impact on Traditional Software Development*, 2025). No entanto, a adoção alargada destas soluções levanta preocupações substanciais ao nível da ética, segurança e governança, exigindo uma reflexão crítica sobre as suas implicações a curto e longo prazo.

Entre os principais riscos éticos, destaca-se a indefinição quanto à propriedade intelectual do código gerado. A utilização de modelos treinados com repositórios de código aberto coloca questões sobre os direitos de autor, nomeadamente quando o output se

assemelha fortemente a exemplos presentes nos dados de treino (Haque, 2024). Acresce o risco de perpetuação de preconceitos sociais por parte dos modelos, que podem reproduzir ou amplificar enviesamentos presentes nos dados, comprometendo a equidade e levantando dilemas éticos complexos (Floridi et al., 2020; Mirishli, 2025). A opacidade dos modelos utilizados, frequentemente encarados como "caixas negras", dificulta a explicação e justificação das decisões automatizadas, o que compromete a transparência e a confiança nos sistemas (Iorliam & Ingio, 2024).

A automação da produção de interfaces também poderá contribuir para o deslocamento de funções tradicionais no mercado de trabalho, especialmente entre desenvolvedores *frontend*, e levantar dúvidas sobre a autenticidade do trabalho produzido e a preservação da identidade profissional no processo criativo (Iorliam et al., 2024). Paralelamente, o código gerado pode incorporar vulnerabilidades ou falhas de segurança não detetadas, o que representa um risco acrescido em sistemas críticos, atribuindo à questão da responsabilidade legal um papel central na discussão (Haque, 2024).

As questões de segurança estendem-se igualmente ao uso de dados sensíveis. Tanto os *mockups* fornecidos como os dados envolvidos no treino dos modelos podem conter informação pessoal ou confidencial, exigindo o cumprimento rigoroso de regulamentos como o *GDPR* ou a *CCPA*. Neste sentido, a proteção de dados deve ser assegurada através de práticas como o consentimento explícito, anonimização e controlo de acesso (Baqar, 2024; Farahani & Ghasemi, 2024).

Face à crescente sofisticação e impacto destas tecnologias, a adoção de mecanismos robustos de governança torna-se imperativa. Estruturas como o padrão *Zero-Trust AI* propõem uma abordagem de segurança holística, integrando políticas, processos e controlo técnico ao longo de todo o ciclo de vida dos sistemas de *IA* (Harbi et al., 2023). A obrigatoriedade de avaliações de risco e análises de impacto ético e social poderá mitigar os potenciais danos colaterais da implementação destas tecnologias (Birkstedt et al., 2023; Farahani & Ghasemi, 2024).

A governança de sistemas de *IA* generativa requer ainda a articulação de múltiplos *stakeholders*, incluindo desenvolvedores, fornecedores, utilizadores, auditores, reguladores e especialistas em ética. A supervisão independente, aliada à criação de diretrizes baseadas em princípios como transparência, justiça, responsabilidade e supervisão humana, tal como defendido nas orientações da *OCDE* e da União Europeia, constitui a base para uma governação responsável (Labazanova et al., 2024; Sarkar, 2023).

Para garantir um desenvolvimento ético e seguro, é essencial incorporar princípios de *design ético* desde a conceção dos sistemas. Isto inclui a implementação de mecanismos para a mitigação de enviesamentos, validação de saídas e recolha de *feedback* dos utilizadores. A transparência no funcionamento dos modelos, bem como a possibilidade de personalização das respostas em alinhamento com os valores dos utilizadores, são também fatores determinantes (Mukherjee & Chang, 2023; Rajaram et al., 2024).

Adicionalmente, o papel da educação e da literacia digital é crucial. Desenvolvedores, designers e utilizadores devem estar conscientes dos riscos associados à *IA* generativa e preparados para atuar de forma crítica, ética e responsável. A sensibilização para limitações técnicas e implicações sociais das ferramentas utilizadas permite promover uma cultura de uso judicioso e informado destas tecnologias (Iorliam & Ingio, 2024; Labazanova et al., 2024).

Finalmente, a investigação contínua em ética da *IA*, mitigação de enviesamentos e desenvolvimento de mecanismos de controlo do output gerado constitui uma área prioritária (Tabassi, 2023). A complementaridade entre abordagens técnicas, regulamentares e educacionais revela-se fundamental para assegurar a confiabilidade, segurança e justiça dos sistemas de geração automática de código *frontend*, contribuindo para uma integração responsável destas soluções na prática industrial (Tabassi, 2023).

3 Análise e Design da Solução

Este capítulo dedica-se à análise crítica e à definição da arquitetura técnica da solução proposta, tendo como base os requisitos funcionais e não funcionais identificados. A construção da solução foi orientada por um conjunto de critérios que visam garantir não apenas a viabilidade técnica da implementação, mas também a sua adequação ao contexto organizacional onde será aplicada.

A análise desenvolvida tem como objetivo fundamentar as decisões de design adotadas ao longo do projeto, assegurando a coerência entre os objetivos delineados e as opções técnicas tomadas. Neste sentido, são discutidas as necessidades específicas da solução, bem como os principais desafios associados à geração automática de interfaces gráficas a partir de *mockups*. Serão ainda apresentadas diferentes alternativas de arquitetura, avaliadas em função da sua complexidade, escalabilidade e grau de alinhamento com os sistemas já utilizados na organização.

A seleção da abordagem final é sustentada por uma análise comparativa entre diferentes configurações possíveis, considerando aspetos como a modularidade da solução, o nível de integração entre os componentes de visão computacional e linguagem natural, e os requisitos de execução local da aplicação. Esta reflexão foi realizada de forma iterativa, em articulação com o ambiente empresarial, e servirá de base à implementação prática descrita nos capítulos seguintes.

3.1 Requisitos da Solução

A definição dos requisitos desta solução foi orientada pelos objetivos do projeto e pelas necessidades práticas identificadas no contexto empresarial onde a aplicação será utilizada. Para garantir uma abordagem eficaz, foi realizada uma distinção entre requisitos funcionais e não funcionais, permitindo uma análise estruturada das capacidades esperadas do sistema.

3.1.1 Requisitos Funcionais

Os requisitos funcionais referem-se às funcionalidades essenciais que a aplicação deve disponibilizar ao utilizador. Estes requisitos garantem que o sistema cumpre os seus objetivos principais, desde a submissão de *mockups* até à geração e disponibilização de código, assegurando uma utilização intuitiva e alinhada com as necessidades reais de desenvolvimento. Abaixo seguem-se os principais requisitos funcionais identificados:

- **RF1:** Como Programador, quero submeter uma imagem de um *mockup* para que possa dar início ao processo de geração de código de forma visual e direta.

- **RF2:** Como Programador, quero adicionar instruções textuais (*prompts*) ao *mockup* submetido para que possa refinar os requisitos, clarificar detalhes não visíveis na imagem e garantir que o código gerado cumpre as especificações técnicas.
- **RF3:** Como Programador, quero treinar um modelo de IA fornecendo um conjunto de exemplos (pares de *mockup* e código) para que o sistema aprenda e replique os padrões e as convenções específicas da minha equipa ou empresa.
- **RF4:** Como Programador, quero visualizar e gerir os exemplos de treino submetidos para que possa auditar a qualidade dos dados e reutilizá-los em futuras sessões de personalização do modelo.
- **RF5:** Como Programador, quero aceder e exportar facilmente o código gerado, com opções para copiar, descarregar ficheiros ou abrir diretamente no meu ambiente de desenvolvimento, para que possa integrá-lo no meu projeto de forma rápida e eficiente.

3.1.2 Requisitos Não Funcionais

Complementarmente às funcionalidades do sistema, os requisitos não funcionais descrevem características de qualidade que a solução deve apresentar, assegurando que esta seja viável, eficiente, acessível e facilmente integrável em diferentes contextos. Estes requisitos assumem especial importância na adoção em ambientes empresariais, onde a robustez, os custos e a portabilidade são fatores determinantes. Para uma análise mais clara e sistemática, estes requisitos foram agrupados em categorias, inspiradas em modelos de qualidade de software como o **FURPS+**, um acrónimo desenvolvido na Hewlett-Packard que categoriza os requisitos em Funcionalidade, Usabilidade, Fiabilidade, Desempenho e Suportabilidade (acrescido do "+" para outras restrições) (Atoum & Bong, 2015). Esta abordagem permite uma cobertura abrangente das características de qualidade do sistema (Atoum & Bong, 2015). Abaixo seguem-se os principais requisitos não funcionais considerados:

Usabilidade

- **RNF1: Intuitividade da Interface** – A interface deve ser concebida de forma a guiar o utilizador intuitivamente ao longo de todo o processo, desde a submissão até à revisão do resultado, minimizando a curva de aprendizagem.

- **RNF2: Acessibilidade** – A solução deve ser utilizável por qualquer membro da equipa de desenvolvimento, mesmo por aqueles sem formação técnica específica em Inteligência Artificial.

Fiabilidade

- **RNF3: Robustez na Geração** – O sistema deve apresentar uma elevada robustez, sendo capaz de processar imagens de diferentes formatos, resoluções e qualidades sem comprometer o processo de geração ou falhar de forma inesperada.

Desempenho

- **RNF4: Tempo de Resposta** – O tempo de execução do *pipeline* de geração de código (análise do *mockup* e geração textual) deve ser adequado para cenários de desenvolvimento rápido, proporcionando tempos de resposta que não interrompam o fluxo de trabalho do programador.

Suportabilidade e Manutenibilidade

- **RNF5: Escalabilidade Arquitetural** – A arquitetura da aplicação deve ser modular e escalável, permitindo a evolução futura com a integração de novos módulos, funcionalidades ou modelos de IA, sem comprometer a estabilidade do sistema existente.

Requisitos "+" (Design, Operacionais e Segurança)

- **RNF6: Conformidade do Código Gerado** – O código *frontend* gerado deve estar alinhado com os padrões, convenções e boas práticas definidos pela empresa, incluindo a utilização de *frameworks* e bibliotecas de componentes específicas.
- **RNF7: Portabilidade** – O sistema deve ser portátil, concebido para ser executado localmente como uma aplicação *standalone*, sem dependência de servidores dedicados ou de uma infraestrutura de *cloud* complexa.
- **RNF8: Eficiência de Custos** – A solução deve ser desenhada para minimizar os custos operacionais, privilegiando, sempre que possível, o uso de serviços e modelos de IA com estruturas de preço acessíveis.
- **RNF9: Proteção de Dados** – O sistema deve garantir que os dados submetidos pelo utilizador (como *mockups* e exemplos de código) são tratados com confidencialidade, não sendo expostos publicamente nem utilizados para outros fins sem o seu consentimento explícito.

3.2 Design de Abordagens de Integração de *IA*

No seguimento da identificação dos requisitos da solução e da análise do estado da arte, torna-se pertinente explorar diferentes abordagens de implementação/fluxo, de forma a compreender quais os caminhos tecnicamente viáveis para responder aos objetivos do projeto. Cada alternativa reflete uma abordagem distinta ao problema da geração automática de código *UI* a partir de *mockups*, com diferentes implicações ao nível da complexidade técnica, flexibilidade, custo e alinhamento com os padrões da empresa.

Uma das primeiras possibilidades consideradas corresponde à **geração direta com modelos *vision-to-code***, onde a imagem do *mockup* é processada por uma rede neuronal, geralmente do tipo *CNN* ou *transformer*, que produz automaticamente o código associado à interface. Esta abordagem destaca-se pela sua **simplicidade e rapidez**, permitindo automatizar por completo o processo de geração sem necessidade de *prompts* ou passos intermédios. Contudo, padece

de **limitações significativas em termos de personalização**, já que o modelo tende a seguir padrões genéricos, não necessariamente compatíveis com os sistemas de design utilizados internamente. Além disso, o treino destes modelos é sensível à variabilidade visual dos *mockups*, o que compromete a robustez da solução em contextos reais. Na Figura 5, é possível observar o diagrama de sequência da abordagem baseada em modelos *vision-to-code*, onde o utilizador submete um *mockup* através da interface da aplicação *web*. A imagem é enviada diretamente para um modelo de visão computacional (*CNN* ou *transformer*), que a processa e gera uma estrutura representativa da interface. Esta estrutura é então transformada automaticamente em código *UI* (*HTML*/*CSS*/JS) por um módulo de exportação, podendo opcionalmente ser guardada em sistema de ficheiros. Por fim, o código gerado é apresentado ao utilizador, encerrando o fluxo de interação.

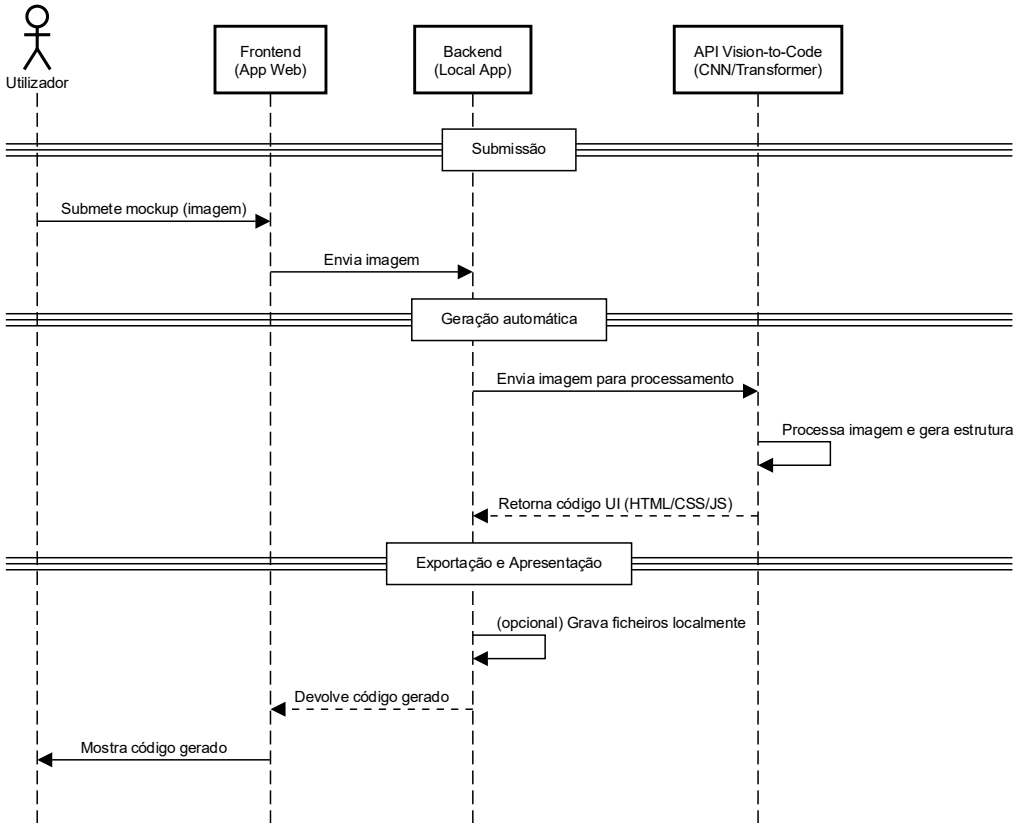


Figura 5 - Diagrama de Sequência da Alternativa 1

Uma segunda alternativa passa por um ***pipeline* multimodal que combina visão computacional e modelos de linguagem**. Neste caso, a imagem do *mockup* é primeiro analisada para extrair uma descrição textual ou estrutural, frequentemente em formato *JSON* ou linguagem natural controlada, que é depois utilizada como input para um *LLM*. Esta solução oferece uma **elevada flexibilidade**, permitindo adaptar a geração de código com base em instruções contextuais fornecidas pelo utilizador. Ao mesmo tempo, possibilita o alinhamento com as práticas da empresa através da **engenharia de *prompts***. No entanto, esta abordagem requer uma **extração precisa e consistente da representação textual** e depende da capacidade interpretativa do *LLM* para compreender e executar corretamente as instruções recebidas. Na

Figura 6, apresenta-se o diagrama de sequência da abordagem multimodal, que combina visão computacional com modelos de linguagem para gerar código *UI*. O fluxo inicia-se com a submissão de um *mockup* e instruções por parte do utilizador. A imagem é processada por um módulo de visão computacional que extrai uma descrição textual ou estrutural da interface. Esta descrição é, em seguida, combinada com as instruções do utilizador, originando um *prompt* estruturado que é enviado para um *LLM*. O código resultante é exportado e apresentado ao utilizador como saída final.

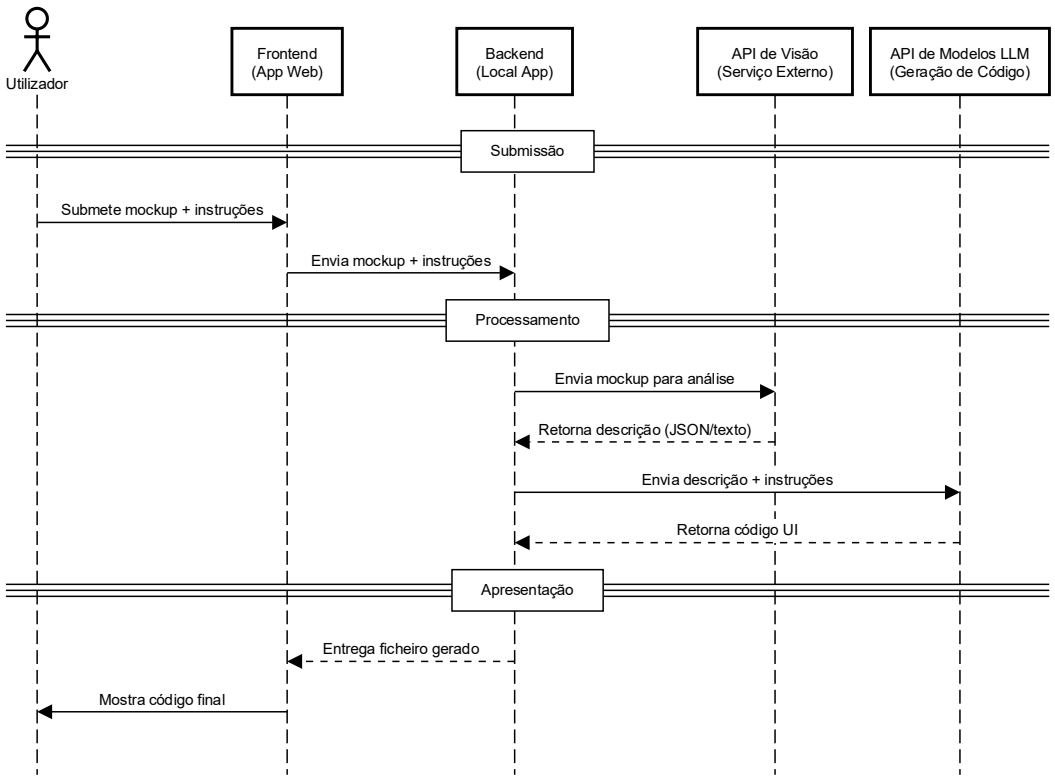


Figura 6 - Diagrama de Sequência da Alternativa 2

Outra via analisada assenta no ***fine-tuning* de um modelo de linguagem com dados específicos da empresa**, consistindo no treino supervisionado com pares de *mockup* e código real recolhidos do repositório interno. Esta abordagem permite obter **uma geração altamente personalizada e fiel aos padrões existentes**, dispensando a construção de *prompts* complexos em tempo de execução. No entanto, envolve **custos elevados associados ao processo de treino** e à preparação dos dados, além de ser menos adaptável a novos requisitos sem necessidade de retraining. Na Figura 7, é representado o diagrama de sequência da abordagem centrada no *fine-tuning* de um modelo de linguagem com dados empresariais. O processo é dividido em duas fases. Numa primeira fase, que ocorre de forma isolada, o modelo é treinado com exemplos reais de *mockups* e código extraídos dos repositórios internos da empresa, permitindo-lhe aprender os padrões específicos de implementação. Após o treino, o fluxo de utilização regular inicia-se com a submissão do *mockup* por parte do utilizador. O modelo, agora adaptado, gera diretamente o código correspondente, o qual é devolvido ao *frontend* e apresentado ao utilizador.

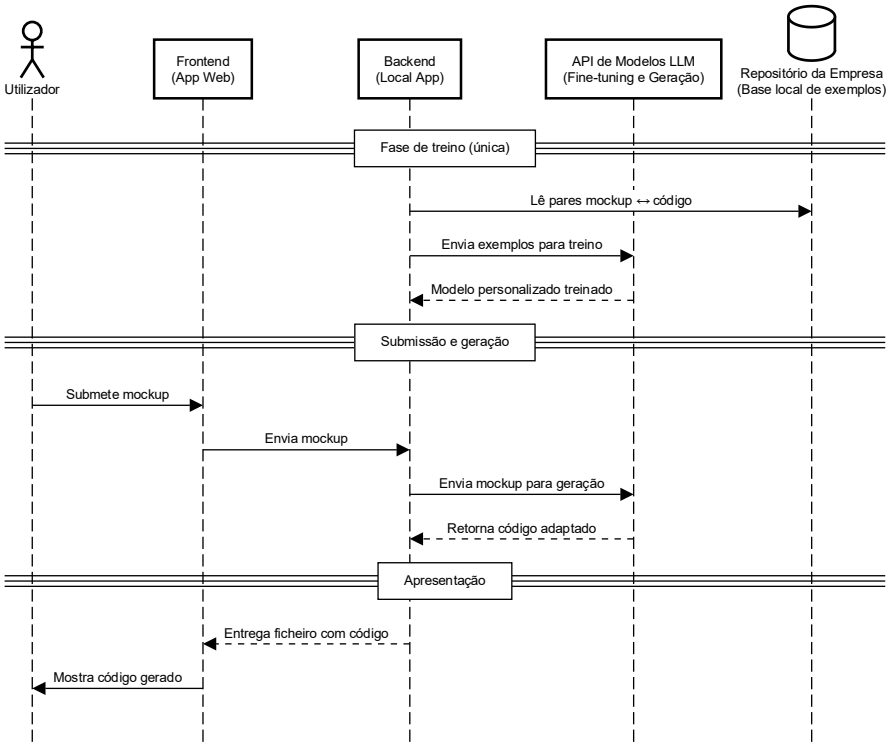


Figura 7 - Diagrama de Sequência da Alternativa 3

Por fim, foi considerada a alternativa baseada em ***RAG***, em que a imagem do *mockup* é complementada com instruções e, em paralelo, o sistema recupera exemplos reais do código da empresa que se assemelham ao caso atual. Esses exemplos são então utilizados como contexto adicional no momento da geração do código pelo *LLM*. Esta solução destaca-se pela **capacidade de contextualização sem necessidade de treino**, garantindo um elevado grau de alinhamento com os padrões internos através da reutilização de exemplos reais. Além disso, apresenta-se como uma alternativa **mais económica e facilmente escalável**, dado que permite atualizar ou expandir a base de exemplos sem intervenção técnica complexa. A principal limitação reside na **qualidade do mecanismo de recuperação**, que deve garantir uma correspondência relevante entre os exemplos e o *mockup* submetido, sob pena de comprometer a utilidade do código gerado. Na Figura 8, é possível observar o diagrama de sequência da abordagem baseada em *RAG*. Após a submissão do *mockup* e das instruções por parte do utilizador, o sistema ativa um motor de recuperação que consulta uma base de dados de exemplos reais da empresa. Os exemplos mais relevantes são selecionados e combinados com os dados submetidos, sendo posteriormente integrados num *prompt* enviado a um modelo de linguagem. Esta contextualização reforça a adequação do código gerado aos padrões internos, permitindo uma personalização eficaz sem necessidade de treino adicional. O resultado é entregue ao utilizador através da interface *web*.

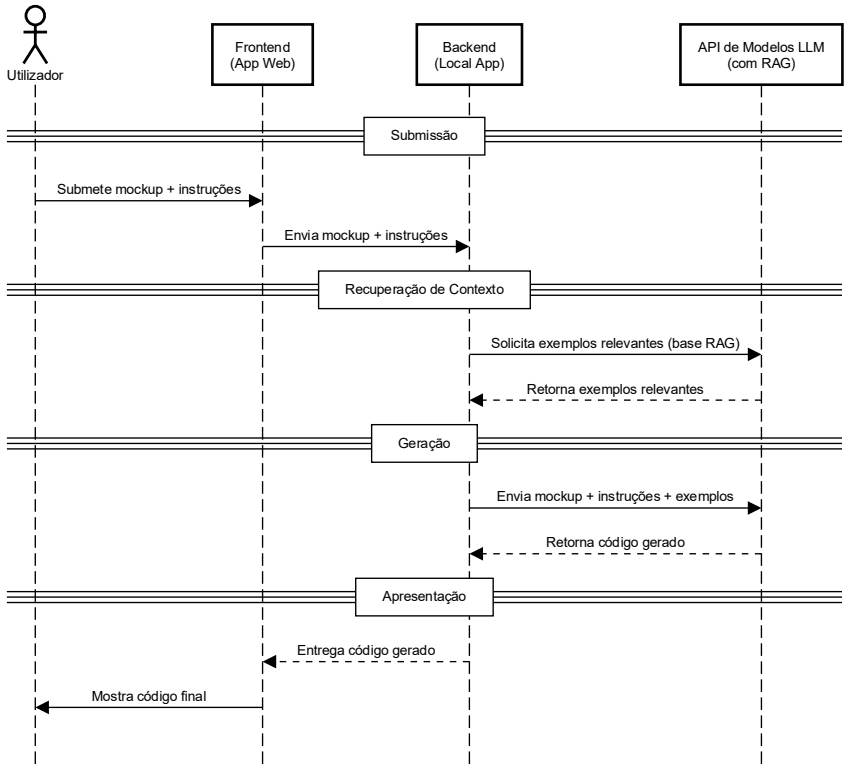


Figura 8 - Diagrama de Sequência da Alternativa 4

Estas quatro alternativas apresentam abordagens distintas, com diferentes graus de complexidade, adaptabilidade e custos operacionais. A análise comparativa destas opções fundamenta a decisão sobre o caminho seguido no desenvolvimento da solução, conforme será discutido na próxima secção.

3.3 Decisão Final de Design da Solução

Após a análise comparativa das várias alternativas de arquitetura apresentadas, optou-se por uma abordagem híbrida que combina os pontos fortes de cada uma das soluções anteriormente exploradas. A decisão foi orientada por critérios de eficácia prática, adaptabilidade à realidade empresarial e alinhamento com as tendências atuais da literatura no domínio da geração automática de interfaces.

A Figura 9 apresenta o modelo adotado para a solução final, representada através de um diagrama de sequência UML. Este diagrama descreve o ciclo de vida completo do sistema, desde a fase de personalização inicial do modelo até ao processo de geração de código e exportação final.

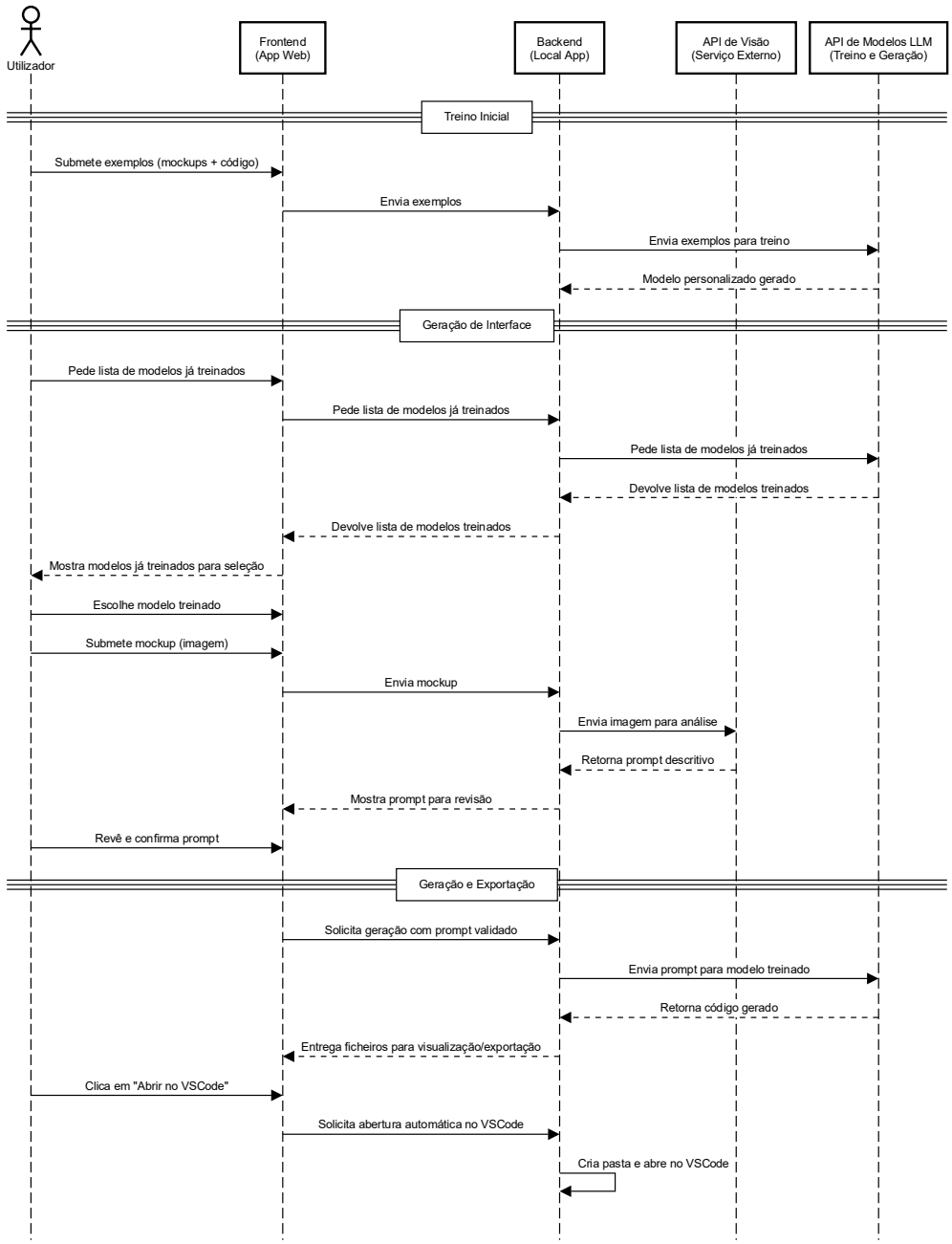


Figura 9 – Diagrama de Sequência Final

Esta abordagem está dividida em duas fases distintas: **treino inicial personalizado** e **utilização prática para geração de código**. A seguir, detalham-se as suas componentes e respetiva fundamentação.

Fase de Treino Inicial

O sistema permite que o utilizador submeta um conjunto de exemplos compostos por um pedido de interface gráfica e respetivo código associado. Estes pares alimentam um processo de *fine-tuning* automatizado que gera um modelo *LLM* personalizado, alinhado com os padrões técnicos e estéticos da organização. Esta abordagem aproveita a capacidade do *fine-*

tuning para adaptar o estilo, estrutura e vocabulário técnico do modelo às necessidades específicas da empresa, conforme discutido na secção 2.5.

Geração de Código com Base em Mockup

Na fase de utilização prática, o utilizador seleciona um dos modelos previamente treinados e submete uma imagem de *mockup*. Esta imagem é processada por uma *API* externa especializada em visão computacional, que extrai uma descrição textual estruturada da interface representada.

Este *prompt* descritivo é, posteriormente, apresentado ao utilizador para revisão — passo que permite incorporar *feedback* humano e aumentar a confiança no sistema, mitigando erros automáticos e reforçando o alinhamento com a intenção do design original.

Uma vez validado o *prompt*, o mesmo é enviado ao modelo de linguagem treinado, que gera automaticamente o código *UI* correspondente. Os ficheiros gerados podem ser visualizados no *frontend* da aplicação ou, opcionalmente, exportados diretamente para o sistema de ficheiros local, com possibilidade de abertura imediata no *VSCode*.

4 Experimentação e Implementação

Este capítulo marca a transição da fase de análise teórica e desenho concetual, detalhada nos capítulos anteriores, para a materialização técnica da solução. A sua principal finalidade é descrever o percurso prático de desenvolvimento, desde os testes em protótipos iniciais até à construção e distribuição da aplicação final. A estrutura do capítulo divide-se em duas vertentes complementares: a **fase de experimentação**, onde foram avaliadas diferentes abordagens de Inteligência Artificial para fundamentar as decisões tecnológicas, e a **fase de implementação**, que detalha a arquitetura, as tecnologias selecionadas e o desenvolvimento modular do *frontend* e *backend*. Este percurso culmina com a descrição do processo de empacotamento da solução numa aplicação *desktop* portátil e auto-suficiente.

4.1 Fase de Experimentação

A transição do desenho conceptual para a implementação exigiu uma fase preliminar de experimentação, cujo objetivo foi validar, em termos práticos, a viabilidade das principais abordagens de IA identificadas na revisão da literatura. Esta etapa foi crucial para avaliar os desafios, custos e a eficácia de cada estratégia no contexto específico da geração de código para a *web*. Nesta secção, são descritos os testes realizados com a técnica de *RAG* e, subsequentemente, com o *fine-tuning* de modelos de linguagem, detalhando as plataformas utilizadas, os resultados obtidos e as limitações encontradas que contribuíram para a decisão tecnológica final.

4.1.1 Testes com a técnica de *RAG*

Inicialmente, recorreu-se à utilização de **plataformas de computação em nuvem** que disponibilizam ambientes de desenvolvimento com acesso a recursos de alto desempenho, como **unidades de processamento gráfico (*GPU*)** e **unidades de processamento central (*CPU*)**, sob um modelo de **pagamento por utilização**. Este tipo de abordagem permite aceder temporariamente a máquinas com elevado poder computacional, ideal para a experimentação de soluções baseadas em modelos de Inteligência Artificial, sem necessidade de investimento em infraestrutura local.

Entre as plataformas exploradas, destacaram-se a ***DigitalOcean*** e a ***Paperspace***. A *DigitalOcean* é um fornecedor de infraestrutura *cloud* bastante popular entre equipas técnicas e *startups*, permitindo o lançamento de instâncias virtuais com imagens otimizadas para ciência de dados. Já a *Paperspace* especializa-se em fornecer ambientes de computação para treino e inferência de modelos de aprendizagem automática, através de soluções como o ***Gradient***, que oferece uma experiência integrada com ***JupyterLab*** e suporte imediato a *GPU*. Esta última plataforma revelou-se particularmente útil para testar a abordagem ***RAG***, uma vez que

disponibiliza ambientes prontos a usar com aceleração por *GPU*, essenciais para a execução eficiente de modelos de linguagem de grande dimensão.

O objetivo da experiência era avaliar a viabilidade de um sistema *RAG* aplicado ao **suporte técnico interno da empresa**. Para tal, preparou-se um **documento técnico estruturado em formato *Markdown* (*.md*)**, assumindo a forma de um **guia prático destinado a novos colaboradores da equipa de *frontend***. Este guia foi concebido com o intuito de compilar boas práticas, padrões internos e instruções detalhadas sobre o funcionamento e manutenção da base de código da empresa.

O documento encontrava-se dividido em múltiplas secções com títulos descritivos, como por exemplo:

```
## Como criar uma tabela

Para criar uma tabela necessitamos inicialmente de ...

## O que fazer quando os estilos não estão a ser compilados?

Quando os estilos não compilam, deve-se a ...
```

Cada secção abordava um tópico específico relacionado com o desenvolvimento *frontend*, cobrindo desde a criação de componentes até à resolução de problemas comuns. Esta estrutura segmentada não só facilitava a leitura e consulta, como também permitia a posterior **indexação automática** e recuperação de conteúdos relevantes de forma granular.

Com o documento preparado, procedeu-se à sua **segmentação automática com base na hierarquia dos títulos**. Cada secção foi tratada como uma unidade independente de conhecimento e processada com o modelo *all-MiniLM-L6-v2* da biblioteca ***SentenceTransformers***, gerando ***embeddings* semânticos** — vetores numéricos que capturam o significado contextual do texto. Os *embeddings* resultantes foram organizados num índice vetorial utilizando a biblioteca ***FAISS (Facebook AI Similarity Search)***, conhecida pela sua eficiência em tarefas de recuperação por similaridade.

Quando o utilizador colocava uma ***query* em linguagem natural**, esta era convertida num *embedding* com o mesmo modelo, e o índice era consultado para identificar as secções mais semanticamente próximas. Estas secções eram então **concatenadas num *prompt* estruturado**, posteriormente fornecido a um **modelo de linguagem da plataforma *Hugging Face***, mais especificamente o *mistralai/Mistral-7B-Instruct-v0.1*. Este modelo, treinado para seguir instruções e responder com base num contexto, devolvia uma resposta gerada com base exclusivamente no conteúdo recuperado.

Abaixo encontra-se uma versão simplificada do código utilizado no ambiente *JupyterLab* para implementar este *pipeline* de forma modular:

```
# Instalação de dependências (executar apenas uma vez)
!pip install transformers faiss-cpu sentence-transformers langchain markdown

# 1. Carregamento e segmentação do ficheiro Markdown
from pathlib import Path
import re

file_path = "documentacao_tecnica.md"
```

```
9.  markdown_text = Path(file_path).read_text(encoding='utf-8')
10.
11. def split_markdown_sections(md):
12.     sections = re.split(r'(#+ .+)', md)
13.     result = []
14.     for i in range(1, len(sections), 2):
15.         title = sections[i].strip()
16.         content = sections[i+1].strip()
17.         result.append((title, content))
18.     return result
19.
20. sections = split_markdown_sections(markdown_text)
21. documents = [f"{title}\n\n{content}" for title, content in sections]
22.
23. # 2. Geração dos embeddings com SentenceTransformers
24. from sentence_transformers import SentenceTransformer
25. embedding_model = SentenceTransformer("all-MiniLM-L6-v2")
26. embeddings = embedding_model.encode(documents, convert_to_numpy=True)
27.
28. # 3. Criação do índice FAISS
29. import faiss
30. import numpy as np
31.
32. index = faiss.IndexFlatL2(embeddings.shape[1])
33. index.add(embeddings)
34.
35. # 4. Pipeline de geração com modelo Hugging Face
36. from transformers import pipeline
37.
38. generator = pipeline("text-generation", model="mistralai/Mistral-7B-Instruct-v0.1",
    device=0)
39.
40. def rag_query(query, k=3):
41.     query_vec = embedding_model.encode([query])
42.     distances, indices = index.search(np.array(query_vec), k)
43.     context = "\n\n".join([documents[i] for i in indices[0]])
44.
45.     prompt = f"""Baseia-te no seguinte contexto para responder:
46.
47. {context}
48.
49. Pergunta: {query}
50. Resposta:"""
51.
52.     result = generator(prompt, max_new_tokens=300, do_sample=False)
53.     return result[0]['generated_text'].split("Resposta:")[-1].strip()
54.
55. # 5. Exemplo de utilização
56. print(rag_query("Como criar uma tabela com colunas personalizadas?"))
```

Este *pipeline*, inteiramente desenvolvido em ***Python*** e executado em ambiente ***JupyterLab***, permitiu testar de forma iterativa diferentes combinações de parâmetros — como o número de secções recuperadas, o número de *tokens* gerados, e a **temperatura de geração**. Este último parâmetro, a **temperatura**, controla o grau de aleatoriedade nas respostas do modelo: **valores baixos (ex.: 0.1)** resultam em respostas previsíveis e conservadoras, enquanto **valores mais elevados (ex.: 0.8)** favorecem maior criatividade, mas com maior risco de **alucinações** — ou seja, respostas incorretas ou inventadas.

Apesar da correta implementação técnica, **surgiram limitações importantes na fase de execução**. Inicialmente, o **tempo de resposta do modelo era excessivamente elevado**, ultrapassando frequentemente os 30 segundos. Para resolver este problema, foram atribuídos recursos computacionais mais robustos, incluindo **instâncias com *GPU* mais poderosas e mais memória *RAM***, o que permitiu reduzir o tempo de resposta para valores aceitáveis (≤ 10 segundos). No entanto, esta melhoria implicou **um aumento considerável dos custos de operação**, uma vez que plataformas como a *Paperspace* cobram proporcionalmente ao tipo de recurso utilizado.

Ainda assim, **a qualidade das respostas manteve-se inconsistente**. Com temperatura baixa, o modelo só respondia corretamente a perguntas praticamente idênticas às existentes no

documento. Ao aumentar ligeiramente a temperatura, as respostas tornavam-se imprevisíveis e, por vezes, descontextualizadas. Um caso ilustrativo foi o pedido para gerar uma nova tabela com colunas diferentes da ensinada (id, nome_imovel, proprietario, status), que resultou numa resposta completamente incorreta, com código fora da *framework Angular* e sem qualquer alinhamento com os padrões da empresa.

Este comportamento levantou dúvidas sobre duas possíveis causas principais:

- **Limitações do modelo utilizado**, com baixa capacidade de generalização técnica;
- **Insuficiência de exemplos e diversidade no documento base**.

Para validar a segunda hipótese, o documento foi **reforçado com novos exemplos**, variações de código e instruções alternativas, de forma a oferecer um espectro mais alargado de contextos. Contudo, os ganhos obtidos foram marginais.

Optou-se então por testar **modelos alternativos** da *Hugging Face*, incluindo:

- *tiiuae/falcon-7b-instruct*, que respondeu de forma rápida mas excessivamente sucinta e genérica;
- *mosaicml/mpt-7b-instruct*, com melhor estrutura de resposta, mas dificuldade em adaptar-se a domínios técnicos específicos;
- *NousResearch/Nous-Hermes-2-Mistral-7B-DPO*, que demonstrou ligeiras melhorias no seguimento de instruções, mas continuou a falhar em manter a coerência com o estilo de desenvolvimento esperado.

Mesmo com pequenas melhorias em fases finais, tornou-se evidente que o **volume acumulado de custos operacionais já era elevado o suficiente para comprometer a viabilidade económica da solução**. A infraestrutura necessária para suportar respostas rápidas e coerentes, aliada à necessidade de mais dados e possíveis ajustamentos por *fine-tuning*, indicava que **a abordagem *RAG* com modelos genéricos, sem adaptação profunda, não era sustentável para um cenário empresarial exigente**.

4.1.2 Testes com Fine-tuning de Modelos *Hugging Face*

Após a experimentação com a abordagem baseada em ***RAG***, explorou-se a possibilidade de melhorar a adequação das respostas através de ***fine-tuning* direto de modelos de linguagem**, utilizando como base o mesmo conjunto de dados técnico previamente estruturado. A motivação principal foi avaliar se um modelo adaptado à terminologia, estilo e padrões da empresa conseguiria superar as limitações observadas na etapa anterior, nomeadamente no que respeita à coerência das respostas com os exemplos práticos fornecidos.

Para essa experiência, foi utilizada novamente a plataforma ***Paperspace***, que já disponibilizava o ambiente de desenvolvimento necessário, com suporte a ***GPU*** e *notebooks*

JupyterLab. A opção por esta infraestrutura manteve-se pela sua flexibilidade na criação de ambientes temporários com acesso direto ao ecossistema *Hugging Face*.

Preparação dos dados

A primeira etapa consistiu na criação manual de um ficheiro *.jsonl* com a estrutura adequada para o processo de *fine-tuning*. Cada linha do ficheiro representava um par de entrada e saída, correspondente a uma pergunta e à resposta correta esperada. A estrutura seguiu o seguinte formato:

Exemplo de ficheiro data.*jsonl*:

```
{
    "instruction": "Como criar uma tabela em Angular com 3 colunas: nome, idade e estado?",
    "output": "<app-tabela><th>Nome</th><th>Idade</th><th>Estado</th></app-tabela>"
} {
    "instruction": "O que fazer quando os estilos não estão a ser compilados?",
    "output": "Verificar se os ficheiros .scss estão incluídos no angular.json e se não existem erros de sintaxe nos estilos personalizados."
} {
    "instruction": "Como criar um formulário reativo com validadores obrigatórios?",
    "output": "Utilizar FormBuilder e Validators.required. Exemplo:\nthis.form = this.fb.group({ nome: ['', Validators.required] });"
}
```

Este ficheiro foi construído manualmente com base nas instruções contidas no documento *Markdown* técnico, sendo adaptado para simular instruções reais que um colaborador poderia colocar ao sistema.

Processo de *fine-tuning*

O processo de treino foi feito utilizando a biblioteca transformers da *Hugging Face*, com o auxílio da classe *Trainer*. Testaram-se diferentes modelos de base, tais como:

- *tiiuae/falcon-rw-1b* (leve e rápido, mas com fraca coerência);
- *EleutherAI/gpt-neo-1.3B* (bom equilíbrio, mas exigente em memória);
- *google/flan-t5-base* (bom para instruções curtas, mas limitado para código);
- *mistralai/Mistral-7B-Instruct-v0.1* (poderoso, mas altamente dispendioso de treinar).

O código-base utilizado para iniciar o processo de *fine-tuning* foi o seguinte:

```
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForCausalLM, TrainingArguments, Trainer, DataCollatorForLanguageModeling

# Carregar dataset a partir de ficheiro jsonl
dataset = load_dataset("json", data_files="data.jsonl", split="train")

# Tokenizer e modelo base
model_name = "tiiuae/falcon-rw-1b"  # Pode ser alterado conforme o modelo a testar
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Tokenizar dados
def tokenize_function(example):
    prompt = f"Instrução: {example['instruction']}\nResposta:"
    full_text = prompt + " " + example['output']
```

```
    return tokenizer(full_text, truncation=True, padding="max_length",
max_length=512)

tokenized_dataset = dataset.map(tokenize_function)

# Argumentos de treino
training_args = TrainingArguments(
    output_dir="./results",
    per_device_train_batch_size=2,
    num_train_epochs=3,
    logging_steps=10,
    save_steps=50,
    save_total_limit=1,
    fp16=True,
    evaluation_strategy="no",
    report_to="none"
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset,
    data_collator=DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False),
)

# Iniciar treino
trainer.train()
```

Resultados e limitações

Os primeiros testes foram realizados com o modelo falcon-rw-1b, que completou três ciclos de treino (*epochs*) em cerca de **1 hora de treino numa instância *A100***, com um custo estimado de aproximadamente **2 a 3 dólares/hora**, dependendo da configuração da *GPU* e tempo de utilização da máquina. No entanto, a qualidade das respostas geradas após o treino foi insuficiente — as instruções mais simples eram compreendidas, mas os exemplos de código frequentemente perdiam formatação ou ignoravam elementos fundamentais, como nomes de propriedades ou componentes *Angular* específicos.

Outros modelos mais complexos, como o *mistralai/Mistral-7B-Instruct-v0.1*, exigiam **mais de 20 GB de VRAM**, o que implicava instâncias com custos superiores a **5 dólares por hora**, tornando o processo de treino financeiramente inviável para testes iterativos.

Mesmo nos casos onde foi possível treinar parcialmente, os ganhos obtidos foram marginais, sendo necessário um volume de dados de treino muito superior — provavelmente algumas centenas de exemplos variados e bem estruturados — para permitir ao modelo captar com eficácia os padrões desejados. Dado o **tempo de treino elevado e os custos associados**, esta hipótese foi **descartada logo à partida como estratégia viável para aplicação empresarial**, uma vez que comprometeria a escalabilidade e atratividade da solução para organizações com recursos limitados.

4.2 Decisão Tecnológica Fundamentada

Durante o desenvolvimento deste projeto, foram avaliadas diferentes estratégias para a implementação da geração automática de código *frontend* a partir de *mockups*. A primeira fase de experimentação centrou-se no uso de *notebooks Jupyter* em ambiente local, recorrendo a duas abordagens amplamente discutidas na literatura: o ***RAG*** e o ***fine-tuning*** de modelos de linguagem.

Contudo, as tentativas práticas demonstraram limitações significativas. A abordagem baseada em *RAG* revelou-se ineficaz no contexto específico da geração de código, dado que os modelos não conseguiam produzir estruturas sintáticas válidas e coerentes a partir da informação recuperada. Já o *fine-tuning* apresentou melhores resultados, com maior consistência e alinhamento com o domínio desejado, mas mostrou-se altamente dispendioso — tanto em termos de custo computacional como da necessidade de grandes volumes de dados anotados e estruturados.

Perante estas limitações, considerou-se a adoção de plataformas comerciais especializadas, nomeadamente a ***OpenAI API*** e o ***Nebius AI Studio***, as quais disponibilizam interfaces de fácil integração via *API* e suporte tanto a modelos *text-to-text* como a modelos multimodais (imagem + texto). Ambas as plataformas possibilitam ainda, de forma indireta, a simulação de um comportamento semelhante ao *RAG*, permitindo o envio de um *prompt* enriquecido com instruções e contexto adicional.

A **comparação técnica e económica** entre estas duas plataformas foi realizada com base nos modelos mais acessíveis e otimizados de cada fornecedor, contemplando os seguintes critérios: custo por milhão de *tokens* para entrada e saída de texto, custo de *fine-tuning*, e suporte a visão computacional. A Tabela 3 apresenta essa comparação de forma detalhada:

Tabela 7 - Comparação de preços entre *OpenAI API* e *Nebius AI Studio*

Plataforma	Tipo de Modelo	Nome do Modelo	Input	Output	*Fine-Tuning* Input	*Fine-Tuning* Output	Custo de Treino
OpenAI	*Text-to-Text*	*GPT-4.1 nano*	0.100	0.400	0.200	0.800	1.500
	Vision-to-Text	*GPT-4.1 nano*	0.100	0.400	N/D	N/D	N/D
Nebius	*Text-to-Text*	*gemma-2-2b-it*	0.100	0.300	0.100	0.300	0.400
	Vision-to-Text	*gemma-2-2b-it*	0.020	0.060	N/D	N/D	N/D

Nota: Todos os valores apresentados referem-se a preços em dólares americanos (USD) por milhão de tokens.

Como se observa, a **plataforma *Nebius* apresenta custos consideravelmente mais baixos**, especialmente no domínio da visão computacional, onde o modelo ***gemma-2-2b-it*** permite inferências a partir de imagens com preços altamente competitivos (apenas $0.02 por milhão de *tokens* de entrada e $0.06 por milhão de *tokens* de saída). Adicionalmente, oferece maior variedade de modelos *open-source*, com suporte a diferentes arquiteturas e possibilidades de *fine-tuning* leve.

Contudo, **a escolha da plataforma ideal não pode ser feita unicamente com base nos custos unitários**. É necessário considerar também a maturidade da *API*, a fiabilidade dos modelos, a documentação disponível, o suporte à integração com *pipelines* empresariais e o desempenho dos modelos em tarefas específicas como a geração de código.

Outro fator crítico prende-se com a **impossibilidade atual de realizar *fine-tuning* diretamente com *mockups* como entrada visual**. Nenhuma das plataformas permite o treino com pares do tipo <imagem do *mockup*, código gerado>. Os processos de *fine-tuning* continuam

a exigir pares exclusivamente textuais (<instrução, resposta>), o que inviabiliza uma adaptação direta do sistema com base em exemplos visuais reais.

Para contornar esta limitação, e com base nas estratégias analisadas no Estado da Arte, optou-se por **adotar um *pipeline* multimodal em duas fases**, onde:

1. Um **modelo de visão computacional** interpreta o *mockup* submetido (imagem) e converte-o num *prompt* descritivo em linguagem natural;

2. Esse *prompt* é, então, enviado para um **modelo *text-to-text* ajustado** ao sistema de design e à *framework* utilizada pela empresa, que gera o código correspondente.

Esta arquitetura modular permite conciliar **eficiência, custo controlado e adaptabilidade ao contexto empresarial**, ao mesmo tempo que alavanca o potencial dos *mockups* como ponto de partida para o processo de geração automatizada.

A implementação concreta da solução adotada será apresentada em detalhe no subcapítulo seguinte.

4.3 Implementação da Solução

A implementação da solução proposta foi organizada de forma modular e escalável, procurando refletir boas práticas de engenharia de *software* e garantir a sua integrabilidade futura com os padrões tecnológicos da empresa. Para tal, optou-se por adotar uma abordagem baseada em monorepositório, utilizando a ferramenta ***Turborepo*** da *Vercel*.

O ***Turborepo*** é uma solução moderna de gestão de monorepositórios, concebida para acelerar o desenvolvimento de aplicações compostas por múltiplos pacotes ou serviços interdependentes. Permite partilhar lógica comum entre aplicações, executar tarefas em paralelo, aplicar cache inteligente de *builds* e orquestrar processos de forma eficiente, o que se revelou particularmente útil num projeto que envolve tanto uma camada de *frontend* como um *backend* associado.

Antes de avançar com esta escolha, ponderou-se a utilização do ***Nx***, uma alternativa robusta e popular para a gestão de monorepositórios em projetos *JavaScript*/*TypeScript*. Contudo, a decisão recaiu sobre o *Turborepo* devido à sua maior leveza inicial, menor curva de configuração e integração mais fluída com projetos que necessitam de arranque rápido e elevada performance local — aspetos críticos durante o ciclo iterativo de uma prova de conceito.

4.3.1 Estrutura do Repositório e Tecnologias Selecionadas

Dado que a comunicação com os modelos de Inteligência Artificial se faz através de *API HTTP*, era necessário incluir um ***backend*** que servisse de camada intermediária entre a aplicação e os serviços externos. Neste contexto, embora existam várias linguagens adequadas

para este tipo de integração (*Python*, *Java*, *Node.js*, entre outras), a escolha recaiu sobre o ***Node.js*** com ***Express***, por três razões principais:

1. Familiaridade prévia com o ecossistema *JavaScript* e *TypeScript*;
2. Compatibilidade com a estrutura do *Turborepo*;
3. Rapidez de desenvolvimento e simplicidade de integração com *API REST*.

Do lado do ***frontend***, a *framework* escolhida foi o ***Angular***, justificando-se esta escolha não apenas pela experiência prévia do autor, mas também pela forte adoção da *framework* no contexto da empresa (Sysnovare), onde a solução poderá futuramente ser mantida. A utilização de ***Angular Material*** complementou a *stack* tecnológica, fornecendo um conjunto de componentes visuais consistentes e responsivos, alinhados com as boas práticas internas já estabelecidas na empresa.

O repositório principal foi denominado ***Blueprint-AI***, refletindo o propósito central do projeto: desenvolver um sistema inteligente capaz de gerar automaticamente o "plano base" (ou *blueprint*) de interfaces gráficas a partir de *mockups*. Dentro do repositório foram criadas duas aplicações principais:

- **blueprint-be**: aplicação *backend* em *Node.js*;
- **blueprint-fe**: aplicação *frontend* em *Angular*.

A vantagem do *Turborepo* manifestou-se desde o início do desenvolvimento, permitindo iniciar ambas as aplicações com um único comando — *npm run dev* — que ativa em simultâneo os servidores *backend* e *frontend* em portas distintas, apresentando-os num terminal dividido. Este mecanismo simplifica e acelera significativamente o ciclo de desenvolvimento, especialmente em ambientes de prototipagem rápida.

4.3.2 Práticas de Estruturação e Configuração

Apesar de se tratar de uma **prova de conceito**, foi seguida uma organização modular e sustentável do código. No *backend*, adotou-se uma estrutura baseada em *routes*, *controllers* e *services*, facilitando a legibilidade e manutenibilidade futura. Em ambos os lados da aplicação (*frontend* e *backend*), foi incluído um ficheiro *config.json*, responsável pela centralização das definições variáveis da aplicação, como *endpoints* externos ou parâmetros de configuração de ambiente.

Em termos de interface, foi concebido um **ecrã inicial** intuitivo e visualmente apelativo, pensado para facilitar a experiência do utilizador desde o primeiro contacto com a aplicação. A Figura 10 apresenta o *layout* da página inicial:

Figura 10 - Ecrã inicial da aplicação

Adicionalmente, foi elaborado um **diagrama de fluxo de interação** que resume as principais etapas da aplicação, desde a submissão de *mockups* até à visualização do código gerado. Este fluxo está representado na Figura 11:

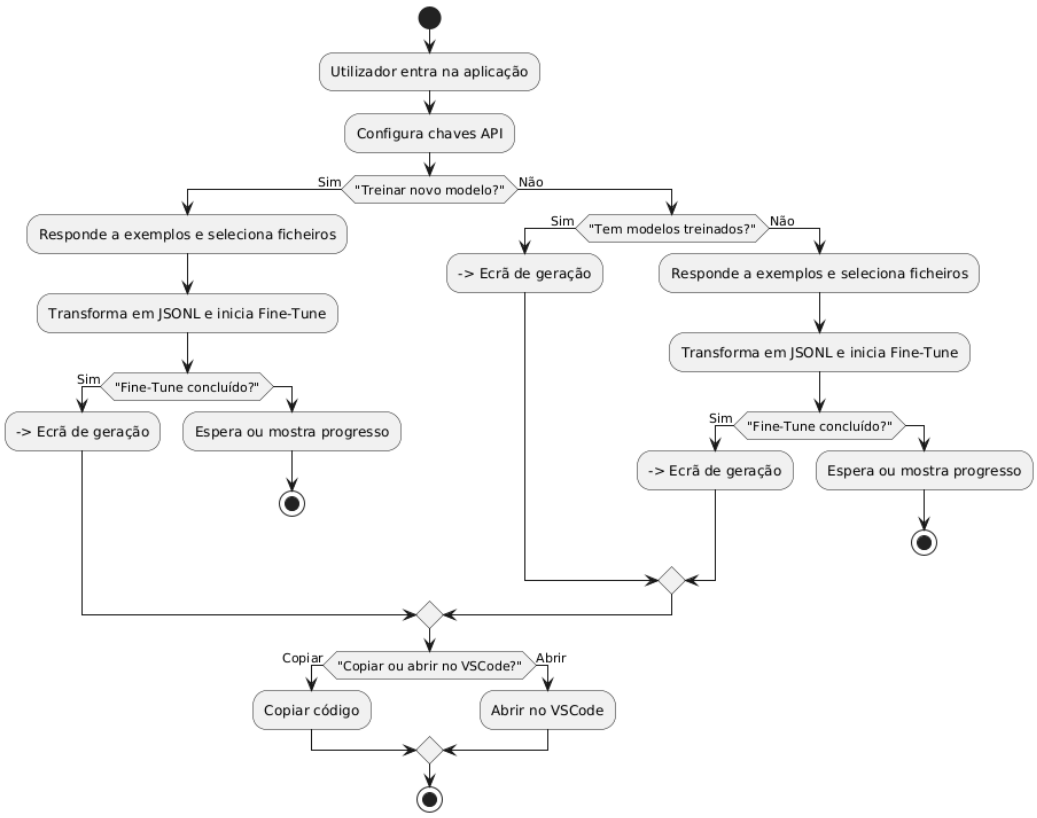


Figura 11 - Diagrama de fluxo da solução implementada

A implementação de cada componente do sistema — incluindo a lógica de comunicação com os modelos de *IA*, o tratamento dos *prompts*, e a integração entre visão computacional e geração de código — será descrita com maior detalhe nos subcapítulos seguintes.

4.3.3 Implementação do Backend

A implementação do *backend* foi um passo fundamental no desenvolvimento da solução, focando-se na criação de um conjunto de *endpoints* de API robustos para suportar as funcionalidades da aplicação. Esta camada de serviço é responsável por toda a lógica de negócio, desde o processamento de dados para treino até à geração final de código. Nesta secção, serão detalhadas as principais responsabilidades do *backend*, que incluem: **o processamento e agrupamento dos dados** para o *fine-tuning* dos modelos; a **orquestração do ciclo de vida do treino**, incluindo o seu início, cancelamento e monitorização; a **análise de imagens** (*mockups*) através da integração com a API de visão computacional; e, centralmente, a **geração do código-fonte** final a partir de um *prompt* e um modelo selecionado. Adicionalmente, serão apresentadas as rotas de apoio que foram implementadas para enriquecer a experiência de desenvolvimento, como a previsão assistida de exemplos e a integração direta com o ambiente de desenvolvimento local.

Processamento e Agrupamento de Dados para Treino

Inicialmente, foi desenvolvida uma rota para processar e formatar os dados de treino enviados pelo *frontend*. O objetivo desta rota, designada /groupData, é receber um conjunto de ficheiros e os seus respetivos conteúdos, e mapeá-los para um formato de resposta estruturado.

O *frontend* envia um *array* de objetos, onde cada objeto contém uma "pergunta" (o requisito, por exemplo, "criar uma tabela com três colunas") e um objeto files (com o nome do ficheiro como chave e o conteúdo como valor). O *backend* processa este *array* e converte-o para o formato *JSON* Lines (*JSONL*), que é o formato exigido para o *fine-tuning* de modelos de linguagem. Cada linha do ficheiro *JSONL* gerado contém um par de mensagens: uma do *user* (a pergunta) e uma do *assistant* (a estrutura de ficheiros e conteúdos em formato *JSON* que satisfaz o pedido). Este ficheiro é depois guardado temporariamente no servidor.

O código-fonte para o *handler* desta rota é apresentado abaixo.

```
1.  export const groupDataHandler = async (req: Request, res: Response) => {
2.    const data = req.body;
3.
4.    if (!Array.isArray(data)) {
5.      return res
6.        .status(400)
7.        .json({ error: "Formato inválido. Esperado um array." });
8.    }
9.
10.   try {
11.     const lines = data.map((item) => {
12.       validateItem(item);
13.
14.       const filesArray = Object.entries(item.files).map(
15.         ([filename, content]) => ({
16.           filename,
17.           content: content.trim(),
18.         })
19.       );
20.
```

```
        return JSON.stringify({
          messages: [
            { role: "user", content: item.question },
            {
              role: "assistant",
              content: JSON.stringify({ files: filesArray }, null, 2),
            },
          ],
        });
      });

      const filename = `grouped-${new Date().toISOString().replace(/[:.]/g, "-")}.jsonl`;
      const filePath = path.join(os.tmpdir(), filename);
      fs.writeFileSync(filePath, lines.join("\n"), "utf-8");

      res
        .status(201)
        .json({ message: "Ficheiro criado com sucesso", path: filePath });
    } catch (error: any) {
      res.status(400).json({ error: error.message });
    }
  };
```

Após a criação do ficheiro *JSONL*, o *endpoint* retorna o caminho para o ficheiro, que pode ser posteriormente descarregado através de uma rota de *download* dedicada.

Seleção da Plataforma e Implementação do *Fine-Tuning*

Para o processo de *fine-tuning*, foi realizada uma análise comparativa entre as plataformas *Nebius* AI e *OpenAI*. Utilizando o mesmo ficheiro *JSONL* de dados em ambos os *playgrounds*, os resultados divergiram significativamente. Os modelos da *Nebius* AI, incluindo o *deepseek* e o *gemma*, demonstraram uma fraca capacidade de generalização, produzindo resultados inesperados e "alucinações". Em contrapartida, os modelos da *OpenAI*, mesmo o *gpt-4.1*-nano (o modelo base à data), produziram resultados superiores. Testes subsequentes com os modelos mais avançados, como o *gpt-4.1-mini* e o *gpt-4.1*, mostraram melhorias incrementais na qualidade da geração de código.

Face a esta análise, optou-se por utilizar a *API* da *OpenAI* para o *fine-tuning*, dada a sua performance superior e a flexibilidade de poder selecionar o modelo a utilizar através de um ficheiro de configuração. Consequentemente, foram desenvolvidas as seguintes rotas para gerir todo o ciclo de vida do *fine-tuning*:

- **GET** /fineTune: Inicia o processo de *fine-tuning*.
- **POST** /fineTune/cancel: Cancela um trabalho de *fine-tuning* em curso.
- **GET** /fineTune/status: Verifica o estado dos trabalhos de *fine-tuning*.
- **GET** /fineTune/job: Obtém detalhes de um trabalho específico.
- **GET** /models: Lista os modelos disponíveis (incluindo os personalizados).
- **DELETE** /models: Remove um modelo personalizado.

Análise de Imagens (*Computer Vision*)

No que concerne à funcionalidade de visão computacional para interpretar *mockups*, a decisão foi diferente. Embora ambas as plataformas (*Nebius* AI e *OpenAI*) tenham apresentado

resultados de alta qualidade na interpretação de imagens, a estrutura de custos da *Nebius* AI revelou-se consideravelmente mais vantajosa:

- ***Nebius* AI:**
    - *Input tokens (1M)*: $0.05
    - *Output tokens (1M)*: $0.15
- ***OpenAI* (*gpt-4.1*-nano):**
    - *Input*: $0.10 por 1M de *tokens*
    - *Output*: $0.40 por 1M de *tokens*

Deste modo, para a extração do *prompt* a partir de uma imagem de *mockup*, optou-se pela utilização da *Nebius* AI. A rota /analyse foi implementada para este fim, recebendo um ficheiro de imagem e devolvendo o *prompt* textual correspondente.

```
router.post("/analyse", upload.single("file"), async (req, res) => {
  try {
    if (!req.file) {
      return res.status(400).json({ error: "Ficheiro não enviado." });
    }

    const base64Image = req.file.buffer.toString("base64");
    const fileType = req.file.mimetype;

    const response = await openai.chat.completions.create({
      model: "mistralai/Mistral-Small-3.1-24B-Instruct-2503",
      temperature: 0,
      messages: [
        {
          role: "system",
          content: config.mockupReadInstruction,
        },
        {
          role: "user",
          content: [
            { type: "text", text: "." },
            {
              type: "image_url",
              image_url: { url: `data:${fileType};base64,${base64Image}` },
            },
          ],
        },
      ],
    });

    const prompt = response.choices[0]?.message?.content;
    res.json({ prompt });
  } catch (error: any) {
    console.error("Erro ao analisar mockup:", error);
    res.status(500).json({ error: error.message });
  }
});
```

Geração de Código e Funcionalidades Auxiliares

A funcionalidade central de geração de código é orquestrada pela rota /generate. Este *endpoint* recebe um *prompt* (seja ele textual ou gerado pela análise de imagem), o identificador do modelo a ser utilizado (permitindo o uso de modelos base ou personalizados via *fine-tuning*) e uma instrução de sistema. O modelo processa esta informação e gera uma resposta em formato *JSON*, contendo uma estrutura de ficheiros e os seus respetivos conteúdos. O *backend* valida esta resposta e formata-a em *Markdown* para uma apresentação clara no *frontend*.

```
export const generateHandler = async (req: Request, res: Response) => {
```

```
    const prompt = req.body.prompt;
    const model = req.body.model;
    const instruction = req.body.instruction;

    try {
      const response = await openai.responses.create({
        model: model || config.openAIModel,
        input: prompt,
        temperature: 0,
        instructions: config.mockupGenerateInstruction || instruction,
      });

      const outputText = response.output_text;
      let parsed: any;
      try {
        parsed = JSON.parse(outputText);
      } catch (error) {
        // ... (error handling)
      }

      // ... (validation of parsed content)

      const markdown = parsed.files
        .map((file: { filename: string; content: string }) => {
          const lang = getLanguageFromExtension(file.filename);
          return `### Ficheiro:
${file.filename}\n\`\`\`${lang}\n${file.content.trim()}\n\`\`\``;
        })
        .join("\n\n");

      res.status(200).json({ files: parsed.files, markdown });
    } catch (err: any) {
      res.status(500).json({ error: err.message });
    }
};
```

Adicionalmente, foram implementadas rotas de apoio para melhorar a experiência de desenvolvimento:

- POST /open-in-vscode: Permite abrir a estrutura de ficheiros gerada diretamente no *Visual Studio Code*.
- POST /predict: Auxilia na criação de novos exemplos de treino. Utiliza os exemplos já existentes como contexto para prever a estrutura de código para um novo requisito, acelerando o processo de anotação de dados.

O *handler* da rota /predict constrói um *prompt* detalhado que inclui os exemplos fornecidos e o novo requisito, instruindo o modelo a gerar o código correspondente no formato *JSON* esperado, seguindo os padrões estabelecidos.

```
export const predictHandler = async (req: Request, res: Response) => {
  const data = req.body.examples;
  const predicted = req.body.predicted;

  if (!Array.isArray(data) || !predicted) {
    return res
      .status(400)
      .json({ error: "Formato inválido. Esperado um array." });
  }

  try {
    // ... (validation)

    const prompt = `Olha para estes dois exemplos... Quero sempre num json {question:
<question>, files: {"component.ts": "<content>"...}}`;

    const predictedRes = await openai.responses.create({
      model: config.openAIModel,
      input: prompt,
      // ... (configuration for JSON output)
    });

    const predicted = predictedRes.output_text;
    const parsed = JSON.parse(predicted);

    res.status(200).json({ predicted: parsed });
  } catch (error: any) {
    res.status(400).json({ error: error.message });
  }
};
```

4.3.4 Implementação do *Frontend*

A implementação do *frontend* foi orientada por um princípio fundamental: a criação de uma interface intuitiva e acessível. O objetivo principal foi abstrair a complexidade inerente aos processos de Inteligência Artificial, permitindo que programadores, mesmo sem conhecimentos especializados na área, pudessem utilizar a ferramenta de forma eficiente para gerar código a partir de *mockups* ou requisitos textuais. Para tal, recorreu-se ao *framework Angular*, em conjunto com a biblioteca de componentes *Angular Material*, que oferece um conjunto de elementos de *UI* robustos e esteticamente consistentes, cuja vasta documentação e utilização prévia na empresa justificaram a sua escolha.

Estrutura e Configuração Base

Para garantir a manutenibilidade e escalabilidade da aplicação, foi estabelecida uma arquitetura base sólida. Primeiramente, foram criados ficheiros de configuração centralizados. O ficheiro *config.json* isola variáveis de ambiente, como o *URL* do servidor, permitindo uma fácil alteração entre ambientes de desenvolvimento e produção.

```
1.  {
2.    "server": "http://localhost:3500/api/v1",
3.    "canAddExamples": false,
4.    "canAddInstructions": false
5.  }
```

Adicionalmente, um ficheiro labels.*json* foi implementado para gerir todos os textos da interface, o que não só simplifica a manutenção do código *HTML*, como também estabelece a base para um futuro suporte multilingue.

```
1.  {
2.    "config": {
3.      "pt": "Configuração",
4.      "en": "Configuration",
5.      "es": "Configuración"
6.    },
7.    "home": {
8.      "pt": "Página Inicial",
9.      "en": "Home",
10.     "es": "Inicio"
11.   },
12.   "title": {
13.     "pt": "Gerador Inteligente de Código",
14.     "en": "Intelligent Code Generator",
15.     "es": "Generador Inteligente de Código"
16.   },
17.   ...
18. }
```

Para promover a reutilização de código e centralizar lógicas comuns, foram criadas as classes abstratas *BaseComponent* e *BaseService*. Todos os componentes e serviços da aplicação herdam destas classes base.

O *BaseComponent* utiliza a injeção de dependências do *Angular* para fornecer acesso imediato a serviços essenciais, como *LabelService* e *ConfigService*, e a funcionalidades como *MatSnackBar* para notificações. Inclui também uma lógica para executar o método *afterLoad*() quando a navegação para um componente é concluída, garantindo que as inicializações ocorrem no momento certo.

```
1.  @Directive()
```

```
export abstract class BaseComponent implements OnInit, OnDestroy {
  labelService!: LabelService;
  configService!: ConfigService;
  snackbar!: MatSnackBar;
  // ...

  constructor(protected injector: Injector) {
    this.labelService = this.injector.get(LabelService);
    this.configService = this.injector.get(ConfigService);
    this.snackbar = this.injector.get(MatSnackBar);
    // ...
  }

  // ... (Lifecycle hooks and utility methods)

  getLabel(key: string): string {
    return this.labelService.getLabel(key);
  }

  afterLoad(): void {
    // Placeholder for actions after component load
  }
}
```

De forma análoga, o *BaseService* fornece uma instância do *HttpClient* e gere de forma reativa o *URL* da *API*, garantindo que todos os serviços descendentes comunicam com o *endpoint* correto, assim que este é carregado a partir da configuração.

```
export abstract class BaseService {
  labelService: LabelService;
  configService: ConfigService;
  http: HttpClient;
  private _apiUrlSubject = new BehaviorSubject<string | null>(null);

  constructor(protected injector: Injector) {
    this.labelService = this.injector.get(LabelService);
    this.configService = this.injector.get(ConfigService);
    this.http = this.injector.get(HttpClient);

    // Reactively set the API URL once configs are loaded
    combineLatest([
      this.labelService.loadedEmitter,
      this.configService.loadedEmitter,
    ]).subscribe(() => {
      const serverUrl = this.configService.getConfigValue('server');
      this._apiUrlSubject.next(serverUrl);
    });
  }

  get apiUrl(): string {
    return this._apiUrlSubject.value || this.configService.getConfigValue('server');
  }

  // ...
}
```

Interface de Treino e Anotação de Dados

A experiência do utilizador começa na página inicial (HomeComponent) apresentada na Figura 10, que apresenta as funcionalidades da solução de forma clara e visualmente atrativa, com um botão de ação rápida que guia o utilizador para a principal interface da aplicação: a de treino de modelos.

Nesta secção, o utilizador pode optar por treinar um novo modelo. Ao selecionar esta opção, é apresentado um componente stepper do *Angular Material* com 10 passos. Este número foi definido por ser o mínimo de exemplos exigido pela *API* da *OpenAI* para iniciar um processo de *fine-tuning*, garantindo ao mesmo tempo a recolha de um conjunto de exemplos suficientemente diversificado.

Cada passo do *stepper* corresponde a um exemplo de treino e é composto por uma pergunta (cujos textos são configuráveis no ficheiro labels.*json*) e uma área de trabalho onde o utilizador pode adicionar, renomear ou remover ficheiros, bem como inserir o código-fonte

correspondente. Para a edição de código, foi integrada a biblioteca *ngx-monaco-editor*, um *wrapper* do editor *Monaco* (o motor do *Visual Studio Code*) para *Angular*. Esta escolha proporcionou uma experiência de desenvolvimento familiar aos utilizadores, como é possível de ver na Figura 12, com funcionalidades avançadas como o reconhecimento automático da linguagem através da extensão do ficheiro e a respetiva coloração de sintaxe (*syntax highlighting*).

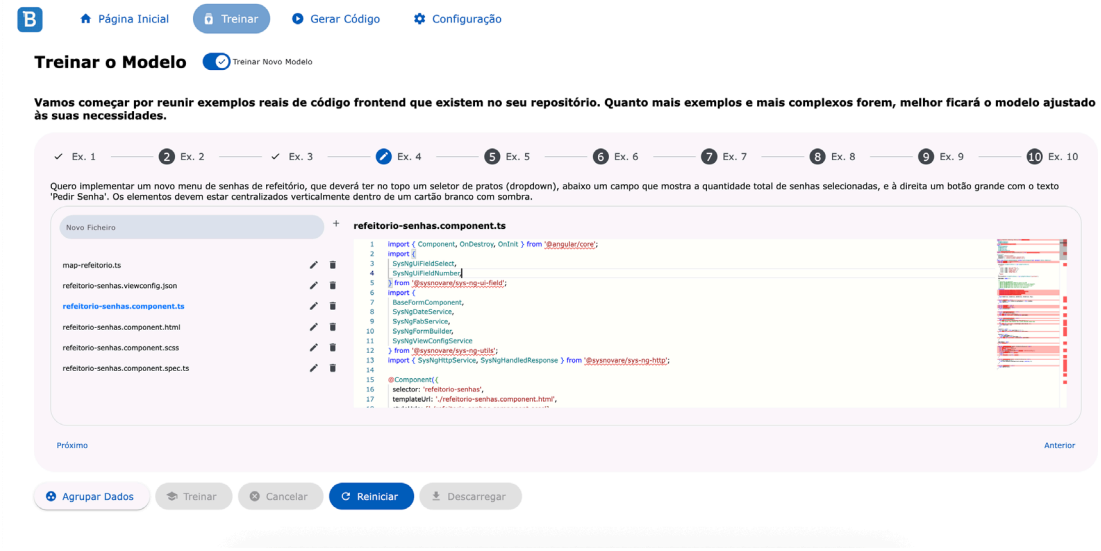


Figura 12 - Interface de Treino do Modelo

Para melhorar a usabilidade e robustez da interface, foram implementadas duas funcionalidades-chave:

1. **Persistência de Dados**: O progresso do utilizador na adição de exemplos é guardado automaticamente na memória local do navegador, como se pode ver nas ferramentas de desenvolvedor do *browser* na Figura 13. Isto previne a perda de trabalho em caso de um recarregamento acidental da página ou fecho do navegador.

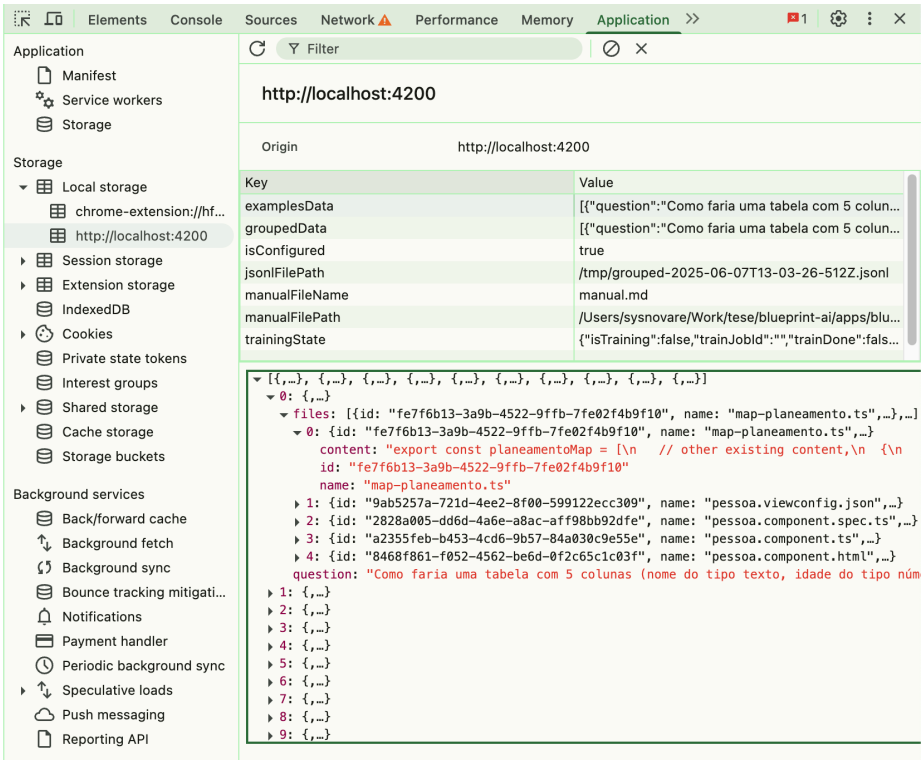


Figura 13 - Funcionalidade de Persistência de Dados

2. **Previsão Assistida**: A partir do quarto passo, é disponibilizado um botão de "Prever Código". Esta funcionalidade invoca a rota /predict do *backend*, enviando os três primeiros exemplos como contexto. O modelo de linguagem gera então uma sugestão de código para o requisito atual. Esta abordagem, evidenciada na Figura 14, demonstrou ser extremamente útil, pois acelera significativamente o processo de anotação, reduzindo o trabalho manual a meros ajustes.

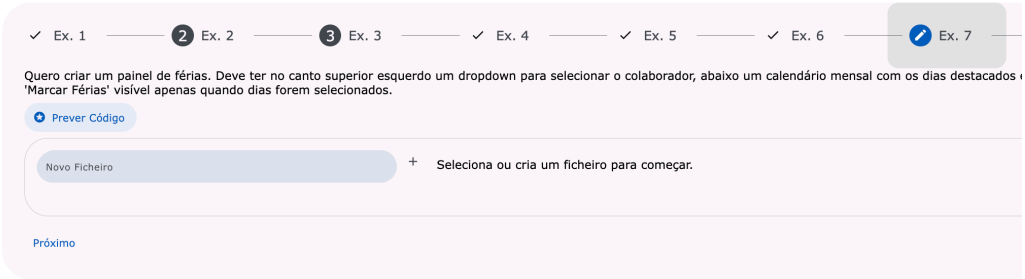


Figura 14 - Funcionalidade de previsão de código

Após o preenchimento dos 10 exemplos, o botão "Agrupar Dados" torna-se ativo. Ao ser clicado, invoca a rota /groupData do *backend* para con*solid*ar todos os exemplos num único ficheiro *JSONL*. Concluído este processo, os botões "Descarregar" (para obter o ficheiro *JSONL*) e "Treinar" ficam disponíveis.

Ao iniciar o treino, a interface fornece um *feedback* visual claro através de uma barra de progresso e da ativação do botão "Cancelar", que permite interromper o processo. Tal como na

fase de anotação, o estado do treino é persistido, permitindo que o utilizador feche a página e retome a monitorização mais tarde sem perder o contexto. Um botão "Reiniciar" está sempre presente para limpar todos os dados do *stepper* e iniciar o processo do zero. Na Figura 15 abaixo, conseguimos observar uma das fases iniciais do treino:

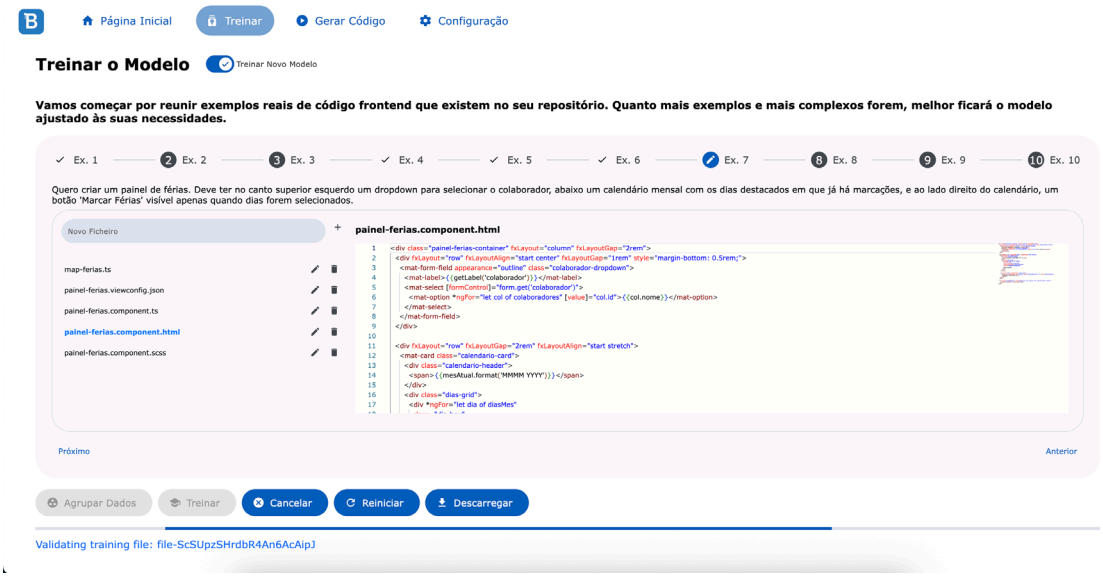


Figura 15 - Processo de treino do modelo de *IA*

Interface de Geração de Código

Uma vez concluído o processo de treino de modelos, a aplicação disponibiliza ao utilizador uma interface dedicada à geração de código a partir de *mockups* visuais. Esta interface encontra-se organizada em diferentes painéis interativos que orientam o utilizador ao longo do processo, desde a seleção do ficheiro de entrada até à exportação do código final.

A primeira secção da interface é dedicada à **importação do *mockup***, sendo apresentada sob a forma de um mat-card com o título *Mockup*. Nesta área, o utilizador pode carregar um ficheiro de imagem através do botão **Selecionar Ficheiro**, o qual está associado a um <input type="file"> escondido. Este primeiro passo está ilustrado na Figura 16:

Figura 16 - Seleção de ficheiro

Após a submissão da imagem, o sistema aciona um processo de análise automática, enviando o ficheiro para a rota */mockups/analyse* do *backend*. Durante esta etapa, a *interface* exibe um indicador de carregamento (*mat-spinner*), como se pode observar na Figura 17, para informar o utilizador de que a operação está em curso e a aguardar uma resposta do modelo de visão computacional.

Figura 17 - Carregamento de prompt

Durante todo este processo o utilizador pode pré-visualizar a imagem do *mockup* a qualquer momento com um tamanho aumentado, bastando para isso clicar na mesma, tal como evidenciado na Figura 18:

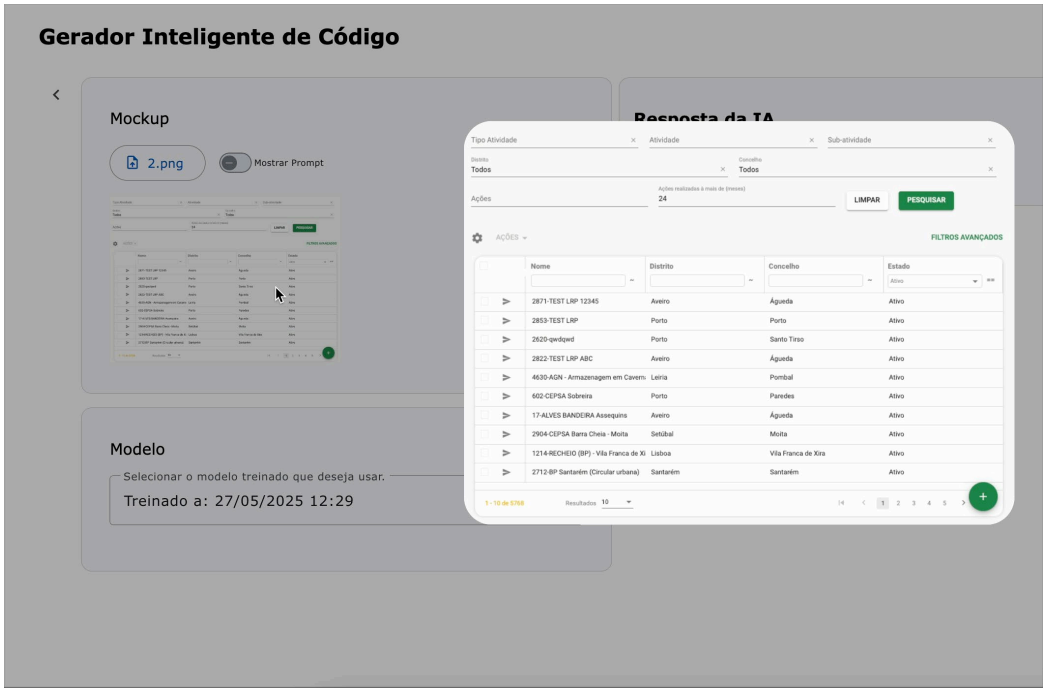


Figura 18 - Pré-visualização da imagem enviada

O resultado principal desta análise é a geração de um *prompt* textual, que é devolvido pelo *backend* e apresentado ao utilizador, conforme ilustra a Figura 19. Este texto, gerado pelo modelo de visão computacional, é automaticamente preenchido no campo de texto associado ao *formControlName="mockupPrompt*". Caso o utilizador pretenda visualizar ou editar este conteúdo, pode ativar o *switch* "Mostrar Prompt", que revela o campo de texto com o conteúdo gerado.

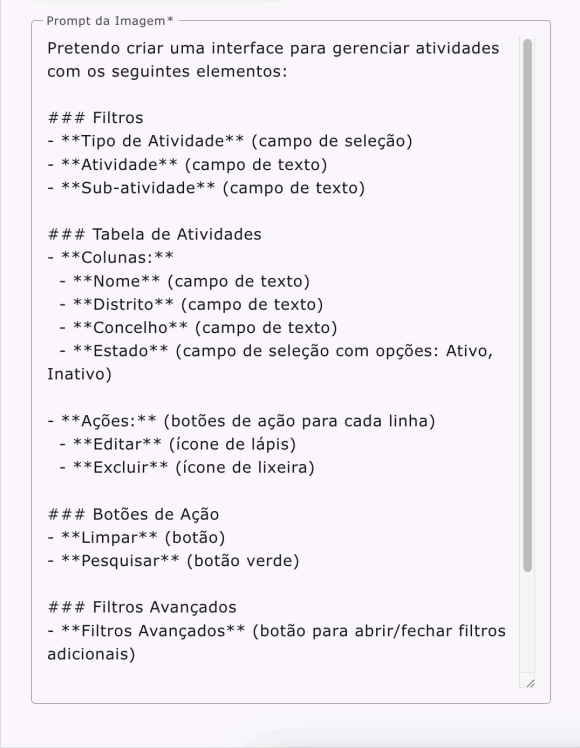


Figura 19 - Descrição devolvida pelo modelo de visão computacional

O resultado da análise consiste na geração de um *prompt textual* com base na imagem fornecida, sendo este automaticamente preenchido no campo de texto associado ao formControlName="*mockup*Prompt". Caso o utilizador pretenda visualizar ou editar este conteúdo, pode ativar o *switch* "**Mostrar Prompt"**, que revela o campo de texto com o conteúdo gerado.

Abaixo deste painel, encontra-se a secção **Modelo**, que disponibiliza um seletor (mat-select) contendo todos os modelos previamente treinados. Este seletor está ligado ao formulário reativo *Angular* através do campo formControlName="model" e permite ao utilizador escolher explicitamente o modelo a utilizar para a geração.

```
this.trainService.getExistingModels().subscribe((models) => {
  this.models = models;
  if (models.length > 0) {
    this.form.patchValue({ model: models[0] });
  }
});
```

Após selecionado o modelo e analisado o *mockup*, torna-se visível o botão **Gerar**, localizado no canto inferior direito da interface, tal como evidenciado na Figura 20. Este botão aciona o método submit(), que envia para o *backend* o *prompt* gerado, o identificador do modelo selecionado e, opcionalmente, uma instrução geral definida no topo do formulário. A resposta da rota */mockups*/generate inclui tanto o *markdown* com o conteúdo descritivo como a estrutura dos ficheiros sugeridos, os quais são apresentados no painel lateral direito da interface.

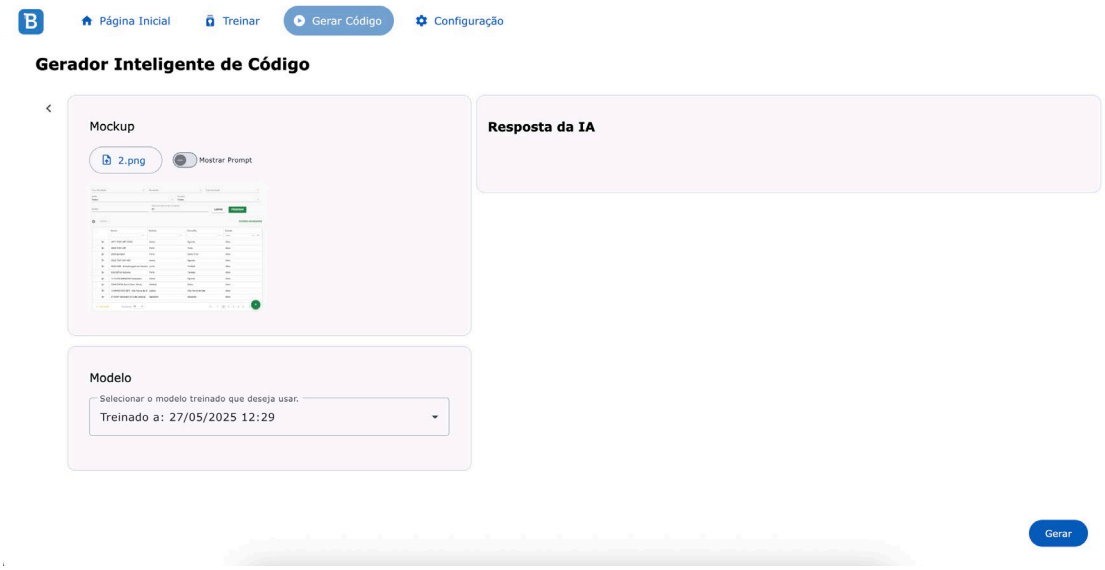


Figura 20 - Interface de Geração pronta a operar

O *markdown* gerado é renderizado de forma enriquecida, com blocos de código devidamente destacados, utilizando a biblioteca *ngx-markdown*, permitindo ao utilizador compreender facilmente a estrutura e o conteúdo dos ficheiros. Na Figura 21 abaixo, é possível visualizar a forma como se apresenta esta resposta.

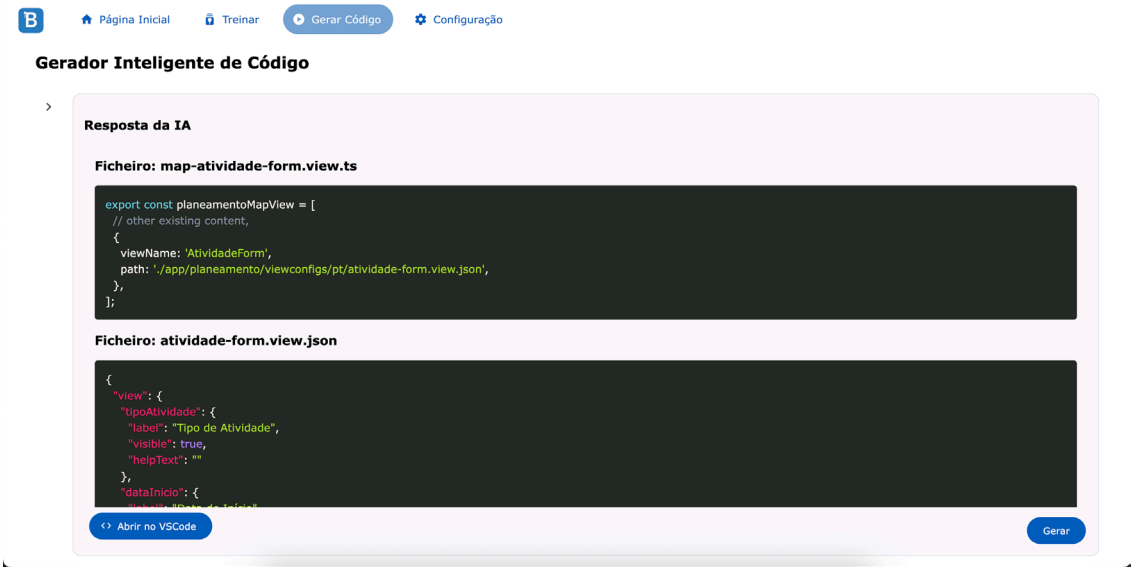


Figura 21 - Resposta da *IA* na aplicação

A funcionalidade é complementada pelo botão **Abrir no *VSCode***, que permite a criação automática dos ficheiros locais, dentro de uma pasta com identificador único baseado na data atual, e a sua abertura no *Visual Studio Code*, tal como é possível verificar na Figura 22. Esta operação é concretizada com recurso à rota /*mockups*/open-in-vscode, proporcionando uma experiência de integração direta entre a interface gráfica e o ambiente de desenvolvimento.

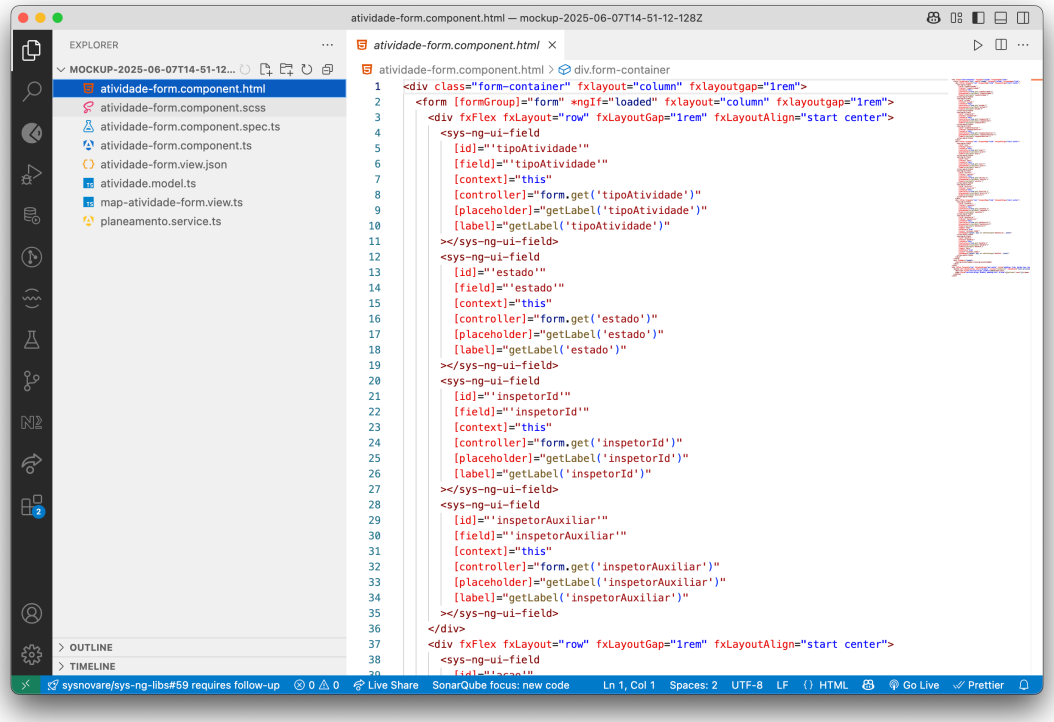


Figura 22 - Abertura do Código no *VSCode*

Desta forma, a interface de geração encapsula todo o fluxo, desde o input visual até à criação de código utilizável, mantendo a interação do utilizador num único ponto e promovendo uma experiência altamente intuitiva e eficiente.

Interface de Configurações

Complementarmente, a aplicação disponibiliza uma interface autónoma para a gestão de parâmetros técnicos essenciais, acessível através do separador **Configuração**. Esta secção foi concebida com o objetivo de **descentralizar a manutenção manual de ficheiros de configuração** e permitir ajustes dinâmicos sem intervenção direta no servidor.

A interface é composta por um conjunto de campos dinâmicos gerados com base nas chaves presentes no ficheiro *config.json* do *backend*. Cada linha da interface exibe uma chave de configuração, acompanhada por um campo de texto (<textarea>) onde é possível editar o respetivo valor, e um botão **Guardar**, que aciona a atualização do valor no servidor.

```
<mat-form-field>
  <mat-label>{{ key }}</mat-label>
  <textarea [(ngModel)]="editedConfigs[key]"></textarea>
</mat-form-field>
<button (click)="updateValue(key)">Guardar</button>
```

O método updateValue() envia a nova configuração para o *backend*, que atualiza o ficheiro em disco de forma persistente:

```
updateValue(key: string): void {
  const value = this.editedConfigs[key];
```

```
3.     this.configBeService.updateConfigKey(key, value).subscribe();
4.   }
```

Entre os parâmetros configuráveis encontram-se:

- **instructionTemplate** – texto pré-definido usado no treino e geração;
- **openAiApiKey** e **nebiusApiKey** – chaves privadas para integração com os serviços externos;
- **defaultModel** – nome do modelo a utilizar por omissão na geração;
- **canAddExamples** – flag que ativa ou desativa a adição de exemplos na interface.

Esta abordagem modular e reativa permite que alterações entrem em vigor sem necessidade de reiniciar o sistema, simplificando o processo de configuração para utilizadores técnicos. A interface descrita, encontra-se representada na Figura 23.

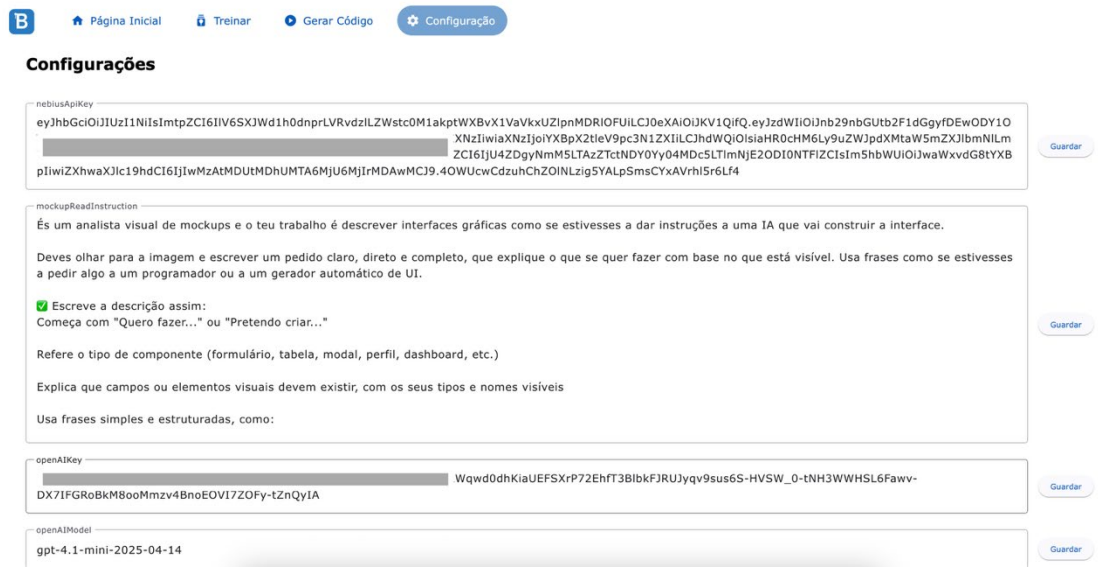


Figura 23 - Interface de Configurações

A inclusão desta funcionalidade reflete uma preocupação com a **manutenibilidade e extensibilidade** da aplicação, promovendo uma gestão mais transparente e autónoma da solução em ambientes empresariais.

4.3.5 Empacotamento da Aplicação

Para assegurar que a solução desenvolvida possa ser utilizada facilmente em diferentes sistemas operativos, mesmo por utilizadores que não possuam ambiente de desenvolvimento previamente configurado, foi adotada uma abordagem de empacotamento com recurso ao ***Electron***, uma *framework* que permite criar aplicações desktop multiplataforma baseadas em tecnologias *web*. Esta decisão visou maximizar a portabilidade e facilitar a distribuição da aplicação sem necessidade de instalação prévia de dependências como o *Node.js*.

Integração com o Monorepositório

A aplicação foi desenvolvida dentro de um **monorepositório gerido com *TurboRepo***, contendo duas aplicações principais:

- **blueprint-fe**: aplicação *Angular* responsável pela interface gráfica.
- **blueprint-be**: aplicação *Node.js* com *Express*, responsável pelas rotas de análise de *mockups*, geração e gestão de ficheiros.

Para integrar ambas num único binário desktop, foi criada uma aplicação denominada **blueprint-desktop**, cujo objetivo exclusivo é servir de **wrapper *Electron*** para empacotar as duas aplicações anteriores numa distribuição executável.

Configuração Inicial e Scripts de Build

A aplicação **blueprint-desktop** possui um *package.json* configurado para coordenar a construção dos projetos *frontend*, *backend* e do bundle final com o *electron-builder*. Abaixo apresenta-se o conteúdo base do ficheiro *package.json*:

```
{
  "name": "blueprint-desktop",
  "version": "1.0.0",
  "main": "main.js",
  "scripts": {
    "build": "npm run build:fe && npm run build:be && npm run package",
    "build:fe": "cd ../blueprint-fe && npm run build",
    "build:be": "cd ../blueprint-be && npm run build",
    "package": "electron-builder",
    "dev": "npm run build:fe && npm run build:be && electron .",
    "dist-mac": "electron-builder --mac",
    "dist-win": "npm run build && electron-builder --win"
  },
  "devDependencies": {
    "electron": "36.4.0",
    "electron-builder": "^26.0.12",
    "electron-packager": "^17.1.2"
  },
  "dependencies": {
    "cross-spawn": "^7.0.6",
    "fix-path": "^4.0.0",
    "shell-path": "^3.0.0",
    "wait-on": "^8.0.3"
  }
}
```

Este ficheiro permite, por exemplo, compilar a aplicação para ***macOS*** com npm run dist-mac ou para ***Windows*** com npm run dist-win.

Configuração do *Electron* Builder

A lógica de empacotamento foi detalhada num ficheiro ***electron-builder*.yml**, que define a estrutura de saída da aplicação, os ficheiros a incluir, os ícones para cada sistema operativo e os diretórios de output. Abaixo encontra-se um excerto representativo:

```
appId: com.sysnovare.blueprint
productName: BlueprintDesktop
directories:
  output: ../../dist/blueprint-desktop
files:
  - "../blueprint-fe/dist/blueprint-fe/**/*"
  - "../blueprint-be/dist/**/*"
  - "../../node_modules/**/*"
  - "main.js"
```

```
    - "!**/*.map"
mac:
  target:
    - dmg
    - zip
  icon: assets/icon.png
win:
  target:
    - nsis
    - portable
  icon: assets/icon.png
extraResources:
  - from: ../blueprint-fe/dist/blueprint-fe/browser
    to: dist/frontend
  - from: ../blueprint-be/dist
    to: dist/backend
  - from: resources/node/
    to: node
```

Esta configuração assegura que tanto os assets compilados do *frontend* como os ficheiros do *backend* e os binários do *Node.js* (disponibilizados na pasta resources/node/) sejam incluídos no bundle final.

Arquitetura do Ficheiro Main.js

O ficheiro main.js é o ponto de entrada da aplicação *Electron*. A sua função principal é **iniciar o *backend* de forma autónoma** e garantir que o *frontend* seja carregado apenas quando este estiver pronto. Este comportamento é particularmente importante para assegurar que o *frontend*, ao iniciar, consiga de imediato comunicar com o *backend* via *HTTP*.

```
function getBundledNodePath() {
  let nodeBinary = process.platform === "win32" ? "node.exe" : "node";
  return path.join(process.resourcesPath, "node", nodeBinary);
}
```

Esta função calcula o caminho para o binário do *Node.js* incluído na pasta resources, garantindo que o *backend* possa ser executado mesmo em sistemas sem *Node.js* previamente instalado. A função startBackend() utiliza este caminho para iniciar o servidor *backend*:

```
function startBackend() {
  const nodePath = getBundledNodePath();
  const backendPath = path.join(basePath, "dist/backend/src/index.js");

  backendProcess = spawn(nodePath, [backendPath], {
    cwd: path.join(basePath, "dist/backend"),
    stdio: "inherit",
  });
}
```

Após iniciar o *backend,* o processo principal do *Electron* aguarda a sua disponibilidade através do módulo wait-on, antes de carregar o *frontend*:

```
await waitOn({
  resources: ["http://localhost:3500/"],
  timeout: 3000,
});
```

Finalmente, é criada a janela da aplicação que carrega diretamente o index.*html* da aplicação *Angular*:

```
function createWindow() {
  const win = new BrowserWindow({
    width: 1200,
```

```
    height: 800,
    webPreferences: {
      contextIsolation: false,
      nodeIntegration: false,
    },
  });

  const pathToFrontend = path.join(basePath, "dist/frontend/index.html");
  win.loadFile(pathToFrontend);
}
```

Portabilidade, Lançamento e Distribuição Final

De forma a garantir que a aplicação possa ser executada em qualquer máquina, **sem necessidade de instalar o *Node.js* ou outras dependências**, foi adotada uma abordagem que inclui diretamente os **binários *standalone* oficiais do *Node.js*** para os sistemas operativos *Windows* e *macOS*. Estes binários foram obtidos a partir do *web*site oficial do *Node.js* e colocados na pasta resources/node/, sendo posteriormente empacotados juntamente com a aplicação. Esta estratégia garante que o ficheiro main.js possa invocar o *backend* com o binário apropriado (node.exe no *Windows* ou node no *macOS*), tornando a aplicação **totalmente portátil e auto-suficiente**.

A distribuição da aplicação é feita através do ***electron-builder***, utilizando configurações definidas no ficheiro *electron-builder*.yml. Este processo permite gerar, de forma automatizada, diferentes tipos de ficheiros executáveis ou instaladores, dependendo do sistema operativo alvo:

- **Para *Windows***:
    - Executável portátil (.exe);
    - Instalador com assistente (.nsis);
    - Arquivo .zip (versão não instalável).
- **Para *macOS***:
    - Imagem de disco .dmg (instalador gráfico);
    - Arquivo .zip (versão não instalável).

O processo completo de *build* e empacotamento é realizado a partir dos **scripts definidos no *package.json* da aplicação blueprint-desktop**, conforme ilustrado abaixo:

```
"scripts": {
  "build": "npm run build:fe && npm run build:be && npm run package",
  "build:fe": "cd ../blueprint-fe && npm run build",
  "build:be": "cd ../blueprint-be && npm run build",
  "package": "electron-builder",
  "dist-mac": "electron-builder --mac",
  "dist-win": "npm run build && electron-builder --win"
}
```

A sequência típica para gerar os artefactos de produção é a seguinte:

- **npm run *build***: compila o *frontend Angular* (blueprint-fe) e o *backend Node.js* (blueprint-be) com os comandos apropriados de cada projeto, colocando os ficheiros resultantes nas pastas dist/blueprint-fe e dist/blueprint-be, respetivamente.
- **npm run package**: após o *build* das apps, executa o *electron-builder* para empacotar os ficheiros gerados juntamente com o ficheiro main.js e os binários do *Node.js*, produzindo o artefacto para a plataforma local.
- **npm run dist-mac**: gera os ficheiros .dmg e .zip para *macOS*.
- **npm run dist-win**: compila tudo e gera os ficheiros .exe, .nsis e .zip para *Windows*.

Durante o processo de empacotamento, são incluídos:

- O *frontend* compilado, acessível através do caminho dist/*frontend*/index.*html*.
- O *backend* como ficheiro *Node.js* executável (index.js) invocado diretamente pelo *Electron*.
- O main.js que orquestra a execução e ligação entre ambos.
- Os binários de *Node.js* para cada sistema.
- Os ícones e metadados da aplicação (ex. icon.png, productName, appId).

Este processo assegura que o utilizador final **pode executar a aplicação com um duplo clique**, sem necessidade de configuração adicional, oferecendo uma experiência fluida, profissional e alinhada com as exigências de distribuição de *software* moderno.

5 Testes e Avaliação de Resultados

Após a descrição do processo de implementação no capítulo anterior, o presente capítulo dedica-se à fase crucial de validação: a análise e avaliação sistemática dos resultados obtidos. O objetivo primordial deste capítulo é aferir, de forma objetiva e multifacetada, a eficácia, a qualidade e a viabilidade da solução *Blueprint-AI* em cenários práticos e representativos, contrastando o desempenho da abordagem automatizada com os métodos de desenvolvimento tradicionais.

Para tal, a avaliação foi estruturada em torno de eixos complementares. Inicia-se com a análise qualitativa de um fluxo de utilização completo, desde a submissão do *mockup* até à integração do código gerado, para demonstrar a operacionalidade do sistema. Subsequentemente, é apresentado um *pipeline* de testes totalmente automatizado, concebido para avaliar quantitativamente a fidelidade visual e estrutural dos resultados. A análise aprofunda-se com um estudo detalhado dos custos de utilização, projetando o impacto económico da solução em diferentes escalas de adoção, e, por fim, é realizada uma comparação direta dos tempos operacionais. Em conjunto, estas análises fornecem uma validação empírica robusta do protótipo, quantificando os ganhos em eficiência e produtividade e fundamentando as conclusões sobre o potencial prático da geração inteligente de interfaces.

5.1 Teste do Fluxo de Utilização da Solução

Para aferir a eficácia e a operacionalidade da solução desenvolvida, procedeu-se à realização de um teste completo, envolvendo todas as etapas do processo, desde a seleção do *mockup* até à análise do resultado final. O objetivo central deste teste consistiu em validar a capacidade do sistema para interpretar corretamente um *mockup* de interface e gerar, de forma automática, o respetivo código de *frontend*, avaliando simultaneamente as limitações e potencialidades da abordagem adotada.

O processo iniciou-se com a seleção de um *mockup* representativo de uma interface de formulário, apresentado na Figura 24. Em seguida, recorreu-se à aplicação Blueprint AI, que já dispunha de um modelo de Inteligência Artificial previamente treinado com exemplos relevantes ao contexto empresarial em análise.

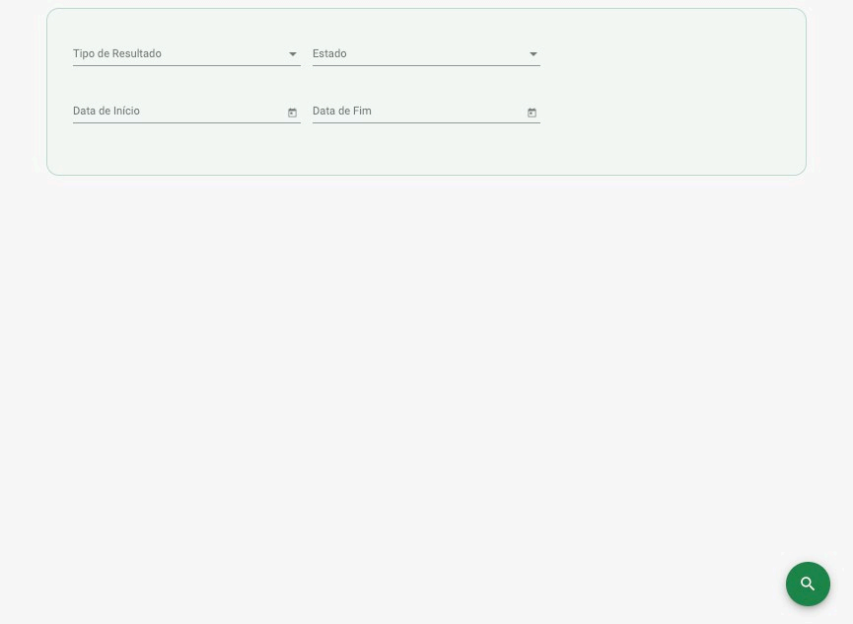


Figura 24 - Mockup utilizado para validação funcional

Após o acesso à plataforma, procedeu-se ao carregamento (upload) do *mockup* na área designada para o efeito. De imediato, foi gerado automaticamente um *prompt* descritivo a partir da análise do *mockup*, utilizando o módulo de visão computacional. O *prompt* obtido detalhava os principais campos identificados na interface, nomeadamente:

- **Tipo de Resultado** (campo de seleção do tipo *dropdown*)
- **Estado** (campo de seleção do tipo *dropdown*)
- **Data de Início** (campo de data com ícone de calendário)
- **Data de Fim** (campo de data com ícone de calendário)

Importa salientar que, embora o sistema tenha reconhecido corretamente a maioria dos elementos, foi detetada uma omissão relevante: o modelo não identificou o botão localizado no canto inferior esquerdo do *mockup*, assinalado por um ícone de lupa. Esta limitação potencialmente impacta a fidelidade do resultado gerado, uma vez que elementos de interação importantes poderão não ser refletidos no código produzido.

Concluída a etapa de análise do *mockup*, foi selecionado o modelo mais recentemente treinado para a tarefa em questão, prosseguindo-se então com a geração automática do código. Uma vez finalizado este processo, o código gerado tornou-se imediatamente acessível, sendo possível a sua visualização e abertura direta no editor *VSCode*, funcionalidade essa ilustrada nas Figura 25 e Figura 26:

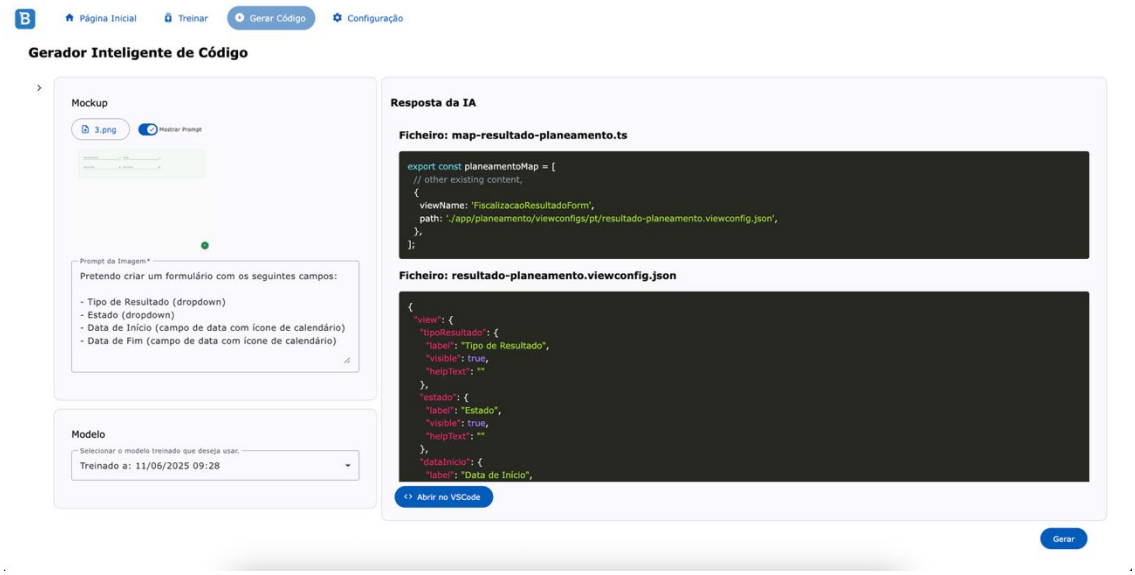


Figura 25 - Geração de código com sucesso

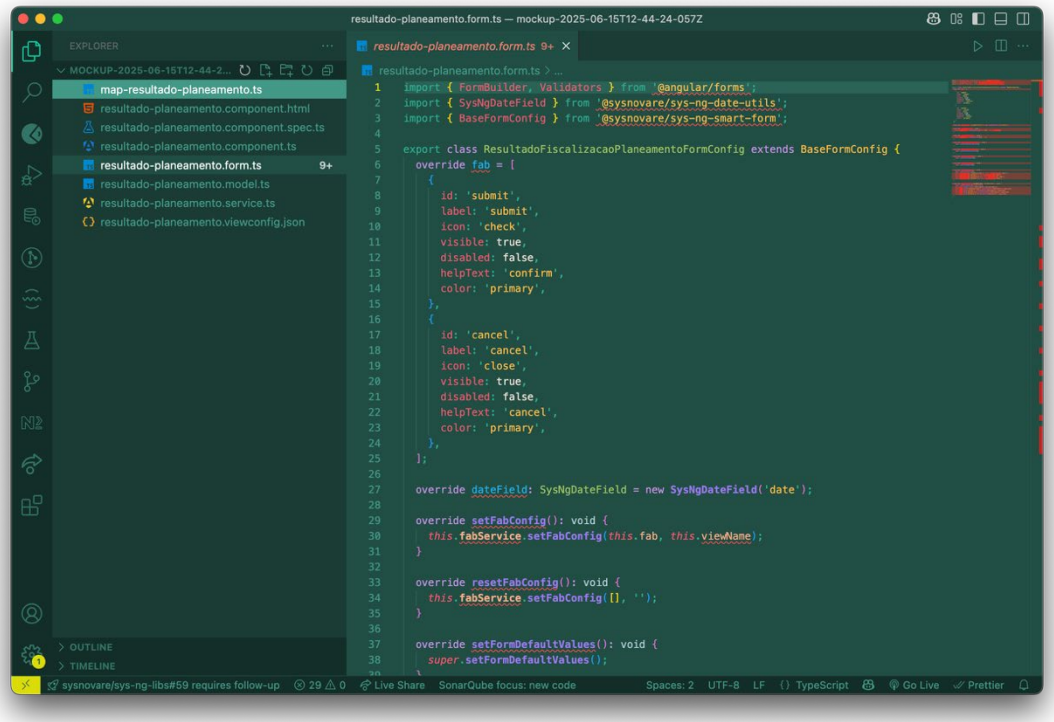


Figura 26 - Abertura do código no *VSCode* automaticamente

A fase seguinte corresponde à integração manual do código gerado na aplicação de base. De notar que, nesta etapa, ainda não existe uma automação total do processo de integração, cabendo ao utilizador a tarefa de identificar o local apropriado para inserção dos ficheiros e realizar os ajustes necessários. Durante este procedimento, foram detetadas algumas inconsistências, nomeadamente “imports” incorretos e definições (“overrides”) desnecessárias,

cuja correção, no entanto, se revelou célere, não ultrapassando os cinco minutos de intervenção manual.

Após estas correções, foi possível compilar e executar a aplicação, obtendo-se uma interface gráfica final (Figura 27) cuja fidelidade face ao *mockup* inicial foi analisada. Uma avaliação comparativa, lado a lado, evidencia que ambos apresentam os mesmos campos de texto e símbolos, demonstrando um grau elevado de precisão estrutural. Contudo, foram observadas algumas divergências: os botões "Cancelar" e "Submeter", presentes no resultado final, não existiam no *mockup* original, enquanto o botão com ícone de lupa, constante do *mockup*, não foi replicado na interface gerada.

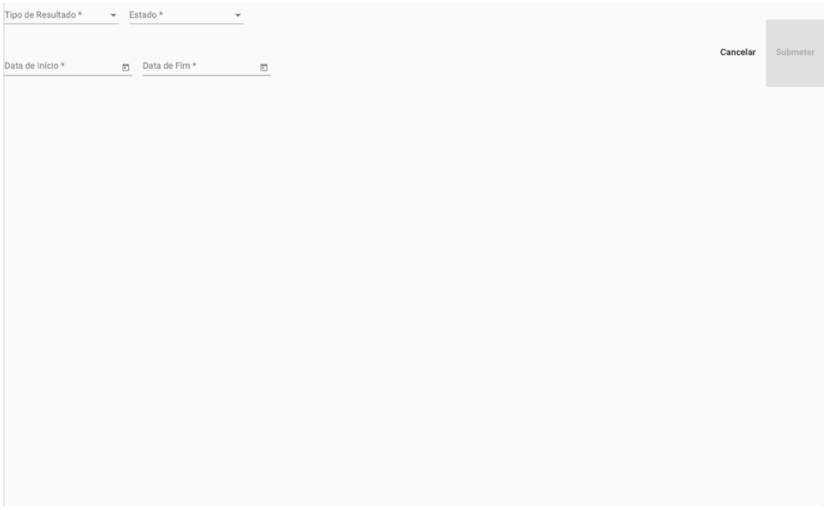


Figura 27 - Interface Gráfica gerada pelo modelo

Em suma, este teste permitiu demonstrar a facilidade de utilização e a eficácia da solução na automatização do processo de geração de interfaces, ao mesmo tempo que evidenciou algumas limitações ao nível da fidelidade visual e da cobertura dos elementos interativos, reforçando a importância da validação manual e da iteração entre utilizador e sistema.

5.2 Automatização da Integração e Avaliação dos Resultados

A validação sistemática da solução desenvolvida exigiu a implementação de um *pipeline* de testes totalmente automatizado, capaz de acelerar o processo de integração do código gerado e de avaliar de forma objetiva a fidelidade dos resultados face aos *mockups* originais. Este *pipeline* baseou-se em dois scripts complementares, concebidos para eliminar a dependência da intervenção manual e garantir uma avaliação técnica rigorosa e reprodutível.

O primeiro script, desenvolvido em *Node.js*, foi projetado para automatizar todas as operações envolvidas na preparação dos exemplos de teste. Este script percorre um conjunto de *mockups*, realiza o upload das imagens para a *API* da solução e recebe, em resposta, os ficheiros de código gerados pelo sistema de *IA*. Com base num sistema de mapeamento de padrões de ficheiros, estes são integrados automaticamente nas localizações adequadas do

projeto *Angular*, sejam componentes, serviços ou configurações. A escolha do *Node.js* justifica-se pela sua eficiência na manipulação de ficheiros e na integração com *API REST*, facilitando a orquestração de todo o fluxo. Após a escrita dos ficheiros, a aplicação pode ser imediatamente compilada e executada, permitindo ao utilizador realizar apenas pequenos ajustes residuais, como a correção de imports ou remoção de funções redundantes. Este processo foi aplicado a seis *mockups* distintos, cada um representando diferentes tipos de interface e complexidade, o que permitiu avaliar a robustez da abordagem. Abaixo encontra-se um excerto de código do script de integração e mapeamento automático de ficheiros:

```
const filesMap = {
  "*.component.html, *.component.ts, *.component.scss": "src/app/components/*",
  "*.service.ts": "src/app/services/*",
  // ... outros padrões
};
function getOutputPath(filename) {
  for (const [pattern, dest] of Object.entries(filesMap)) {
    // Regex para identificar destino
    const regex = patternToRegex(pattern);
    if (regex.test(filename)) {
      if (dest.endsWith("/*")) {
        const dir = path.join(appPath, dest.slice(0, -2));
        return path.join(dir, filename);
      } else {
        return path.join(appPath, dest);
      }
    }
  }
  return null;
}
```

Depois de integrados os ficheiros e compilada a aplicação, foi possível capturar o resultado apresentado no *browser* e proceder à comparação automática com o *mockup* original. Para este efeito, foi desenvolvido um segundo script em *Python*, recorrendo a bibliotecas de visão computacional e *OCR*, nomeadamente *EasyOCR*, *OpenCV* e *PIL*.

Este script extrai os textos e respetivas localizações (*bounding boxes*) de ambas as imagens, identifica correspondências entre campos com base numa métrica de similaridade (*SequenceMatcher*) e produz duas saídas: uma imagem comparativa lado a lado com destaques visuais, e um ficheiro *JSON* com o resumo estatístico dos resultados obtidos. Abaixo também é possível ver um pequeno excerto desse código:

```
def draw_side_by_side(mockup_img, output_img, matches, unmatched, filename):
    # Redimensiona imagens para a mesma altura
    h = max(mockup_img.height, output_img.height)
    new_m = mockup_img.resize((mockup_img.width, h))
    new_o = output_img.resize((output_img.width, h))
    canvas = Image.new("RGBA", (new_m.width + new_o.width, h + 40), (255,255,255,255))
    canvas.paste(new_m, (0, 40))
    canvas.paste(new_o, (new_m.width, 40))

    draw = ImageDraw.Draw(canvas)
    # Labels e caixas verdes/vermelhas
    for m, o in matches:
        x1, y1, x2, y2 = m['bbox']
        draw.rectangle([x1, y1+40, x2, y2+40], outline=(0,255,0), width=3)
        xo1, yo1, xo2, yo2 = o['bbox']
        draw.rectangle([xo1 + new_m.width, yo1+40, xo2 + new_m.width, yo2+40],
outline=(0,255,0), width=3)
    for m in unmatched:
        x1, y1, x2, y2 = m['bbox']
        draw.rectangle([x1, y1+40, x2, y2+40], outline=(255,0,0), width=3)
    canvas.save(os.path.join(result_dir, f'diff_{filename}'))
```

O resultado deste processo é facilmente visualizável, conforme ilustrado na Figura 28. Esta figura apresenta um exemplo real de comparação entre *mockup* e output, em que todos os campos principais do *mockup* “3.png” foram corretamente reconhecidos e destacados a verde

em ambas as imagens. Pelo contrário, noutras figuras, como no *mockup* “5.png”, observam-se múltiplos campos a vermelho, evidenciando limitações na replicação da estrutura e conteúdo originais, como se pode observar na Figura 29.

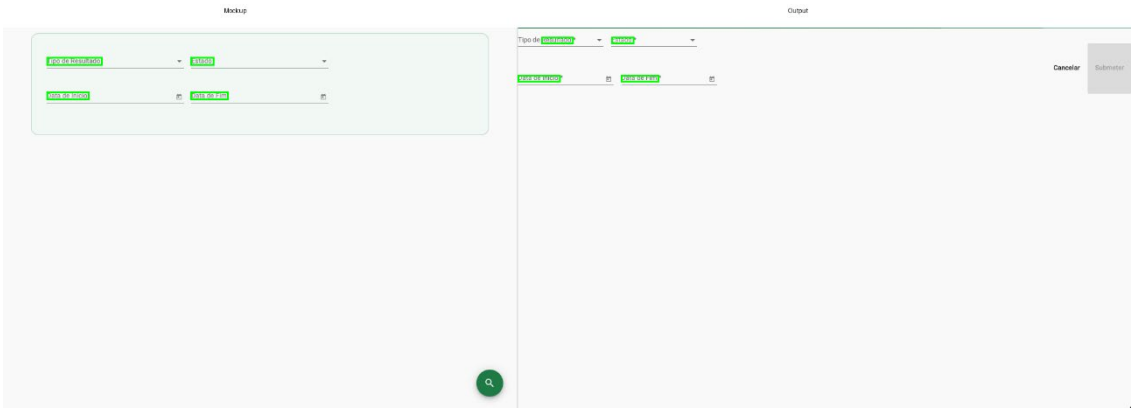


Figura 28 - Representação Visual Comparativa do Mockup 3

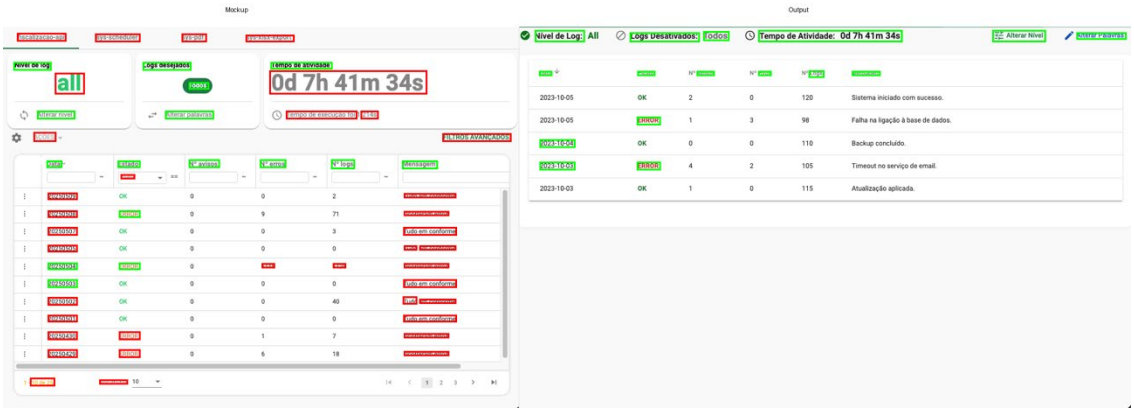


Figura 29 - Representação Visual Comparativa do Mockup 5

Os resultados quantitativos desta análise encontram-se sumarizados na Tabela 4, que indica para cada exemplo o número de matches, diferenças e a percentagem global de similaridade, bem como os tipos de ajustes manuais ainda necessários.

Tabela 8 - Resultados da análise automática dos *mockups*

Mockup	Matches	Diferenças	Percentagem Similaridade (%)	Ajustes Manuais	Observações
1	7	38	9,46	Correção de imports	Campos de empresa e morada não replicados
2	13	41	22,22	Remoção de funções redundantes	Perda de campos de detalhe e filtros
3	4	0	100,00	Nenhum	Correspondência perfeita para campos principais
4	9	2	81,82	Ajuste de estilos	Perda em campos de data, resto fiel
5	16	38	30,19	Correção de nomes de variáveis	Várias linhas e labels omitidas
7	15	8	65,22	Ajuste de formatação	Campos de instrução e botões replicados

Esta abordagem técnica e automatizada, para além de acelerar o ciclo de testes, permitiu detetar tendências importantes: a solução é altamente fiável para estruturas simples, mas apresenta desafios em interfaces com elevada densidade de informação ou elementos gráficos menos convencionais. A dependência de *OCR* introduz limitações em campos com tipografia pouco comum ou sobreposição de elementos, mas a análise visual lado a lado continua a ser uma ferramenta eficaz para validar o desempenho geral. Importa referir que, embora os ajustes manuais necessários após a geração sejam residuais, subsistem oportunidades de melhoria — nomeadamente ao nível da normalização de imports e da robustez do modelo para campos menos frequentes.

No conjunto, o *pipeline* desenvolvido assegura uma avaliação objetiva, quantitativa e facilmente auditável, promovendo a transparência na validação de soluções de geração automática de interfaces e demonstrando o potencial de integração destes métodos no ciclo de desenvolvimento de *software* empresarial.

5.3 Análise de Custos de Utilização

No contexto da validação da solução desenvolvida, foi realizada uma análise experimental dos custos associados a cada etapa do *pipeline* de geração automática de interfaces, com base em testes reais efetuados durante o desenvolvimento e avaliação da plataforma. Estes testes incidiram sobre dois eixos principais: (i) custos do processamento inicial de *mockups* via visão computacional (*Nebius*) e (ii) custos de geração de código com modelos de linguagem (*OpenAI*, *GPT*-4.1 e variantes). Os resultados permitem fundamentar a viabilidade económica da abordagem e prever o impacto financeiro em diferentes escalas de utilização.

Testes de Custos na Transformação de *Mockups* em Prompts (*Nebius* Vision)

Para automatizar a conversão de *mockups* em descrições textuais (*prompts*), foi testada a *API Nebius* com o modelo *mistralai/Mistral-Small-3.1-24B-Instruct-2503*, específico para tarefas de visão computacional. Durante os testes, registou-se que cada conversão típica envolveu cerca de 1.141 *tokens* de input (imagem/processamento) e 68 *tokens* de output (descrição gerada). Os preços praticados, à data, encontram-se na Figura 30.

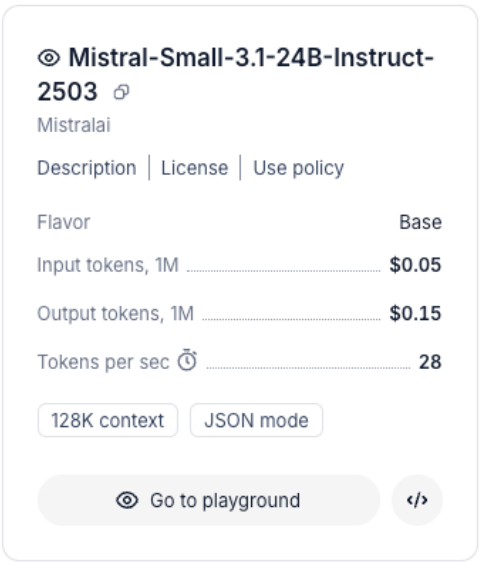


Figura 30 - Preços do modelo *Mistral* para Visão Computacional na Plataforma *Nebius*

Com base nos registos reais, o custo médio apurado por conversão foi:

$$Custo\ Nebius = \frac{1.141}{1.000.000} \times \$0{,}05 + \frac{68}{1.000.000} \times \$0{,}15 \approx \$0{,}000067$$

Este valor confirmou-se em múltiplos testes, demonstrando que o custo da etapa de visão computacional é praticamente irrelevante no contexto global do *pipeline*.

Testes de Custos de Geração de Código com Modelos de Linguagem

De seguida, procedeu-se à avaliação dos custos associados à geração automática de código, recorrendo a modelos de linguagem de última geração. Os testes centraram-se nos modelos da família *GPT*-4.1 (standard, mini e nano), cujos custos à data do documento se encontram representados na Figura 31, com a maioria dos ensaios a privilegiar o *GPT*-4.1 standard, dada a sua superioridade em qualidade e robustez nos cenários empresariais testados.

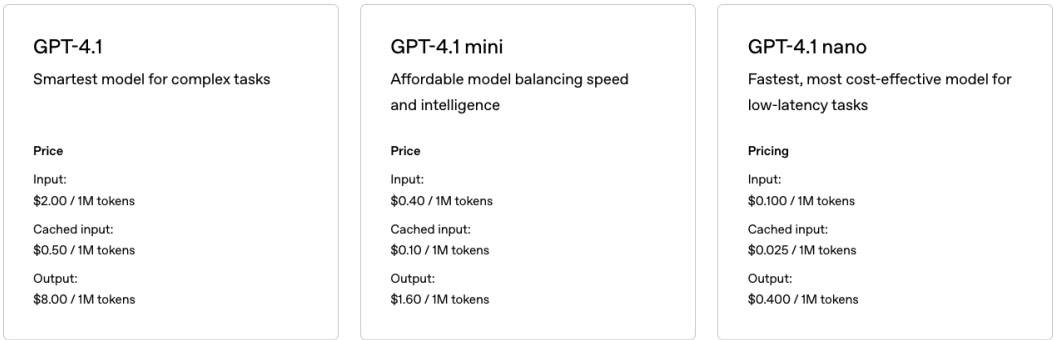


Figura 31 - Custos da Família *GPT*-4.1 para respostas

Durante os testes, foram recolhidos os seguintes dados de utilização e custos, para diferentes pedidos, observáveis abaixo na Tabela 5:

Tabela 9 - Testes de custos relativos ao uso do *GPT*-4.1

Pedido	Tokens Input	Tokens Output	Custo Input ($)	Custo Output ($)	Custo Total ($)
Teste 1	189	1.167	0,000378	0,009336	0,009714
Teste 2	410	1.430	0,000820	0,011440	0,012260
Teste 3	2.050	7.200	0,004100	0,057600	0,061700
Teste 4	750	430	0,001500	0,003440	0,004940

Os testes demonstraram que, mesmo para pedidos de maior dimensão (ex: Teste 3), o custo individual por geração de código manteve-se sempre inferior a sete cêntimos de dólar, evidenciando a sustentabilidade económica da solução.

Custo Global por Ciclo de Utilização

Combinando ambas as etapas do *pipeline* — (i) transformação do *mockup* em *prompt* (*Nebius*) e (ii) geração de código (*OpenAI*) — foi possível apurar o custo médio total de um ciclo completo de utilização, suportado por medições reais:

$$Custo\ Total\ por\ ciclo = Custo\ Nebius + Custo\ GPT4.1 = \$0{,}000067 + \$0{,}00971 \approx \$0{,}00978$$

Ou seja, cada iteração experimental (da imagem ao código final) teve um custo prático inferior a um cêntimo de dólar, sendo a parcela correspondente à visão computacional praticamente desprezável face ao processamento do modelo de linguagem.

Projeção de Custos para Cenários Alargados

Para avaliar o impacto económico da solução em diferentes contextos de utilização, foram definidos e simulados três cenários representativos de adoção. No **primeiro cenário**, correspondente a uma pequena equipa de desenvolvimento composta por dois utilizadores, cada elemento realizou, em média, vinte pedidos por dia ao longo de vinte e dois dias úteis mensais. Este padrão de utilização resulta num total de **880 ciclos mensais**. Considerando os custos apurados nos testes experimentais, o processamento das imagens através do serviço *Nebius* representou uma despesa mensal de apenas $0,06, enquanto a geração de código com o modelo *GPT*-4.1 totalizou $8,55. Assim, o custo global mensal deste cenário ascendeu a aproximadamente **$8,61**, um valor perfeitamente comportável face aos benefícios introduzidos pela automação.

No **segundo cenário**, simulando uma empresa de média dimensão, foi considerada uma equipa de dez utilizadores, cada um realizando cinquenta pedidos diários durante vinte e dois dias úteis por mês. O volume de utilização eleva-se, neste contexto, a **11.000 ciclos mensais**. A análise dos custos evidencia que a componente *Nebius* mantém-se praticamente residual, com

um total de $0,74, enquanto a utilização do modelo *GPT*-4.1 corresponde a $107,00, o que perfaz um custo total mensal de cerca de **$107,74**.

Por fim, o **terceiro cenário** reflete a adoção em larga escala numa plataforma SaaS, com mil utilizadores ativos a efetuar dez pedidos por dia durante vinte e dois dias por mês. Esta configuração resulta em **220.000 ciclos mensais**, sendo os custos de utilização do serviço *Nebius* de $14,74 e os custos associados ao modelo *GPT*-4.1 de $2.137,08, o que determina um custo total mensal aproximado de **$2.151,82**.

Estes resultados demonstram que, mesmo em cenários de elevada utilização, o custo adicional decorrente da etapa de visão computacional é praticamente insignificante face ao valor total, permanecendo a componente principal associada à geração de código com o modelo de linguagem. Em todos os cenários, os custos absolutos mantêm-se plenamente justificáveis perante os ganhos de produtividade e a escalabilidade proporcionada pela solução.

Reflexão e Conclusão Experimental

Os testes realizados permitem concluir que a solução apresenta custos de utilização extremamente baixos por ciclo completo, mesmo com modelos de topo. A experiência demonstrou que a componente de visão computacional tem um impacto económico praticamente nulo, sendo o principal custo associado à geração de código — ainda assim, em valores perfeitamente aceitáveis para a grande maioria das organizações.

Em suma, a validação experimental dos custos comprova a viabilidade técnica e financeira da abordagem proposta, quer em contextos de pequena escala, quer em cenários empresariais ou plataformas SaaS de grande dimensão.

5.4 Análise de Tempos Operacionais

A validação experimental da solução incluiu a monitorização rigorosa dos tempos de operação em todas as fases do *pipeline* automatizado, permitindo uma avaliação objetiva da eficiência do sistema comparativamente ao desenvolvimento manual tradicional. Para garantir precisão e reprodutibilidade, foram utilizadas ferramentas digitais adequadas a cada contexto e todos os resultados foram tratados estatisticamente.

Metodologia de Medição

O tempo de execução do processo de ***fine-tuning*** foi registado com o cronómetro de precisão do sistema operativo (aplicação "Relógio" do *macOS*), iniciado no momento exato do envio do pedido e interrompido assim que a plataforma *API* da *OpenAI* indicou a conclusão do treino. Para as **operações online da aplicação**, recorreu-se à funcionalidade ***Network > Timing*** das *Chrome Developer Tools*, permitindo medir ao milissegundo a duração de cada pedido *HTTP* e todos os tempos associados ao ciclo de requisição e resposta *cloud*.

No caso do **desenvolvimento manual**, foi utilizada a técnica de *stopwatch*, recorrendo à aplicação *Timer* de smartphone, sendo as medições feitas por um observador independente.

Descrição dos *Endpoints*

Para a fase de testes, foram medidos de forma sistemática os tempos de resposta dos dois principais *endpoints* (ou pontos finais de *API*) do sistema, ambos essenciais no *pipeline* de automação:

- **Endpoint /analyse:** Este *endpoint* é responsável por receber um ficheiro de imagem (*mockup* da interface) e proceder à análise da mesma através de um modelo de visão computacional. O resultado do *endpoint* é a transformação da imagem num *prompt* descritivo em linguagem natural, adequado para ser processado posteriormente por um modelo de linguagem generativo. Este serviço representa a primeira etapa do *pipeline* automático e corresponde, do ponto de vista computacional, à tradução visual-textual do *mockup*.
- **Endpoint /generate:** Após obter o *prompt* descritivo, este *endpoint* processa esse texto recorrendo a um modelo de linguagem (ex: *GPT*-4.1), gerando o código fonte correspondente à interface pretendida. O *endpoint* /generate representa, assim, a etapa central do *pipeline*, onde ocorre a conversão automática do requisito textual para código executável, sendo fundamental para a automatização da produção de *software frontend*.

Ambos os *endpoints* foram integrados numa interface *web* para testes, tendo o tráfego monitorizado e os tempos de execução registados para análise.

Resultados Experimentais: Fine-tuning

Três execuções distintas do processo de *fine-tuning* do modelo foram cronometradas, utilizando o cronómetro do sistema operativo, com os resultados sumarizados na Tabela 6 abaixo.

Tabela 10 - Tempos de Resposta do Fine-tuning

Execução	Tempo (min)
Execução #1	32,8
Execução #2	39,4
Execução #3	45,2
Média	39,1
Desvio Padrão	6,2

Resultados Experimentais: Tempos de Resposta dos *Endpoints*

A Tabela 7 apresenta os tempos registados para ambos os *endpoints*, recolhidos através do painel "*Network*" das *Chrome DevTools*, permitindo analisar separadamente o desempenho da análise visual e da geração de código.

Tabela 11 - Tempos de Resposta dos pedidos às *API*'s dos Modelos de AI

Teste	**/analyse** (s)	**/generate** (s)
Execução #1	4,03	38,51
Execução #2	4,21	34,60
Execução #3	3,89	42,75
Execução #4	4,37	36,88
Média	4,13	38,19
Desvio Padrão	0,19	3,14

Resultados Experimentais: Comparação com Desenvolvimento Manual

Para aferir a eficiência do sistema automatizado, comparativamente à implementação manual, foi pedido a um programador júnior que implementasse os mesmos *mockups* do zero, recorrendo a técnicas de *stopwatch* operadas por um observador independente. Os resultados são apresentados na Tabela 8.

Tabela 12 - Comparação de Tempo de Implementação Humano vs AI

Mockup	Tempo Humano (min)	Tempo AI (s)	Ajustes Finais (min)
1	71	42	4
2	67	39	5
3	69	43	6
Média	69	41,3	5
Desvio Padrão	2,0	2,0	1,0

Comparação de Custos: *IA* vs. Desenvolvimento Manual

Complementando a análise dos tempos operacionais, foi também efetuada uma comparação direta entre o custo económico do *pipeline* automatizado e o custo estimado do trabalho humano tradicional. Admitindo um valor de referência de **€18 por hora** para um programador júnior em Portugal, o custo associado ao desenvolvimento manual de um *mockup*, com um tempo médio de 69 minutos, é de aproximadamente **€20,70** por interface gerada:

Em contrapartida, o custo do *pipeline* automático, considerando o processamento completo (analyse + generate) e os pequenos ajustes finais, é dramaticamente inferior. Com base nos custos experimentais detalhados na secção anterior, o custo de processamento da *IA* situa-se, em média, **abaixo de €0,01 por *mockup*** (mesmo incluindo a componente *Nebius*),

enquanto o tempo de intervenção manual adicional é de apenas 5 minutos, traduzindo-se num custo humano marginal de **€1,50**. A Tabela 9 sintetiza esta comparação:

Tabela 13 - Comparação dos tempos e custos por *mockup*: *IA* vs. desenvolvimento manual

Processo	Tempo Total Médio	Custo Unitário (€)
Desenvolvimento manual	69 min	20,70
***IA* (*pipeline* automático)**	1 min (execução)	<0,01
Ajustes manuais pós-*IA*	5 min	1,50
Total *IA* + ajustes	6 min	1,51

A análise evidencia que, mesmo contabilizando o tempo de revisão e integração manual indispensável após a geração automática, o custo total da abordagem baseada em *IA* representa uma fração do custo do desenvolvimento tradicional — sem prejuízo da qualidade final, desde que se mantenha a supervisão técnica apropriada.

É fundamental referir que estes valores têm apenas finalidade ilustrativa e não constituem incentivo à substituição de profissionais humanos. O papel do programador permanece central, quer na validação dos outputs, quer na adaptação contínua dos sistemas automáticos aos requisitos específicos do projeto.

6 Conclusões

A presente dissertação propôs-se a investigar e desenvolver uma solução inovadora para a geração inteligente de interfaces gráficas para a *web*, com foco na sua aplicabilidade em contextos empresariais. Ao longo deste trabalho, foram exploradas diversas abordagens de Inteligência Artificial, culminando na implementação e avaliação de um protótipo funcional. Este capítulo final sintetiza os principais resultados obtidos, discute as limitações encontradas e aponta direções para trabalhos futuros.

6.1 Objetivos atingidos

A concretização deste projeto de investigação e desenvolvimento permitiu alcançar, de forma significativa e demonstrável, os objetivos primordiais e específicos delineados no início da dissertação. O desiderato central, que consistia no desenvolvimento de uma solução tecnologicamente avançada, alicerçada em *IA*, capaz de gerar automaticamente componentes de interface gráfica a partir de *mockups* visuais, e que fosse simultaneamente compatível com os rigorosos requisitos técnicos, estéticos e funcionais de um ambiente empresarial real, foi efetivamente materializado. A aplicação *Blueprint-AI* representa a concretização deste objetivo, servindo como uma prova de conceito robusta e um veículo para a exploração das potencialidades e desafios desta abordagem.

6.1.1 Alinhamento com Padrões Empresariais e Coerência entre Design e Implementação

Um dos pilares fundamentais do projeto residia na capacidade de **gerar código que não só fosse funcional, mas que também estivesse intrinsecamente alinhado com os padrões empresariais** prevalecentes. Este alinhamento transcende a mera produção de código sintaticamente correto, englobando a adesão a *frameworks* específicos, a utilização de bibliotecas de componentes padronizadas e a conformidade com as convenções de estilo e arquitetura da organização. No contexto deste trabalho, o foco recaiu sobre o *framework Angular* e a biblioteca *Angular Material*, escolhas representativas de tecnologias amplamente adotadas em ambientes de desenvolvimento corporativo. A solução *Blueprint-AI* demonstrou uma competência notável nesta área, primariamente através da implementação de um sofisticado *pipeline* multimodal. Este *pipeline* inicia-se com a intervenção de um modelo de visão computacional, o qual assume a responsabilidade crítica de analisar e interpretar semanticamente o *mockup* visual submetido pelo utilizador. Esta etapa inicial é de capital importância, pois traduz a representação puramente imagética da interface numa descrição estruturada e textual, passível de ser processada por módulos subsequentes de forma mais precisa.

A descrição textual gerada pelo modelo de visão serve, então, de *input* para *LLM*. Crucialmente, este *LLM* não é um modelo genérico, mas sim um que foi submetido a um processo de afinação (*fine-tuning*). Este processo de especialização envolveu o treino do modelo com um conjunto curado de exemplos de código e padrões de design específicos do domínio empresarial em análise, incluindo a utilização de componentes *Angular Material* e as convenções de nomenclatura e estruturação preferidas. O *fine-tuning* revelou-se um diferenciador crítico, capacitando o *LLM* a gerar *outputs* que refletem não apenas a estrutura lógica da interface delineada no *mockup*, mas também a sua estética, a correta instanciação de componentes *Angular Material* e a adesão às boas práticas de desenvolvimento preconizadas pela organização.

Adicionalmente, a **coerência entre a intenção original do design, expressa no *mockup*, e a implementação final do código** foi ativamente promovida e considerada um fator de sucesso. A arquitetura de duas fases adotada, onde a análise visual culmina na produção de um *prompt* textual intermédio, oferece um ponto de controlo e validação humana de valor inestimável. Antes da geração efetiva do código pelo *LLM* afinado, o utilizador tem a oportunidade de rever e, se necessário, refinar este *prompt* descritivo. Esta intervenção humana, embora introduza um elemento não totalmente automatizado no processo, provou ser fundamental para mitigar ambiguidades inerentes à interpretação visual, corrigir potenciais interpretações erradas por parte do modelo de visão e assegurar um alinhamento mais preciso com as nuances e subtilezas do design original. Os resultados da avaliação, detalhados extensivamente no Capítulo 5, confirmaram que esta abordagem híbrida minimiza significativamente a necessidade de ajustes manuais extensivos no código gerado, especialmente para componentes de interface com complexidade moderada. No entanto, a mesma avaliação também sublinhou que a supervisão humana continua a ser pertinente e recomendável para elementos de interação mais intrincados ou para interfaces com requisitos de design altamente específicos e não convencionais, onde a capacidade de generalização dos modelos atuais pode encontrar limitações.

6.1.2 Estabelecimento de um Processo Automatizado, Reprodutível e Escalável

A criação de um **processo que fosse simultaneamente automatizado, reprodutível e escalável** constituiu um objetivo específico essencial, visando garantir a eficiência operacional e a viabilidade da solução em cenários de desenvolvimento de *software* prático e contínuo. Este objetivo foi plenamente alcançado através da arquitetura e da implementação da aplicação *Blueprint-AI*. O sistema foi meticulosamente concebido para encapsular o fluxo de trabalho completo, desde o momento inicial da submissão do *mockup* visual pelo utilizador, progredindo para a análise inteligente e extração de características por parte do módulo de visão computacional, a subsequente geração do *prompt* descritivo, a criteriosa seleção, por parte do utilizador ou do sistema, do modelo de linguagem treinado mais adequado para a tarefa em mãos, e culminando na geração e exportação do código *frontend* correspondente. A natureza intrinsecamente automatizada deste fluxo reduz drasticamente a necessidade de

intervenção manual nas fases intermédias, o que se traduz em ganhos de tempo e na minimização de erros humanos.

A reprodutibilidade dos resultados é assegurada pela natureza determinística (ou controladamente estocástica, dependendo da configuração dos modelos de *IA*) do processo: para um dado *input* (*mockup* e configurações) e um modelo de *IA* específico, o sistema tenderá a produzir *outputs* consistentes. Esta consistência é vital para a confiança no sistema e para a sua integração em *pipelines* de desenvolvimento mais amplos. A componentização da solução, com uma clara e bem definida separação de responsabilidades entre o *frontend* (desenvolvido em *Angular*, responsável pela interação com o utilizador e apresentação dos resultados) e o *backend* (desenvolvido em *Node.js* com *Express*, responsável pela orquestração das chamadas às *API* dos modelos de *IA*, gestão de ficheiros e lógica de negócio), juntamente com a utilização de *API* para a comunicação com os serviços de Inteligência Artificial externos (neste caso, *OpenAI* e *Nebius* AI), garante não apenas a modularidade do sistema, mas também a sua **escalabilidade**. Esta arquitetura permite que cada componente possa evoluir de forma independente – por exemplo, a interface do utilizador pode ser redesenhada sem impactar a lógica de geração de código, ou novos modelos de *IA* podem ser integrados no *backend* sem necessidade de alterar o *frontend*. Esta flexibilidade é crucial para a sustentabilidade e a adaptabilidade da solução a longo prazo, permitindo que se mantenha relevante face à rápida evolução das tecnologias de *IA* e das necessidades dos utilizadores.

6.1.3 Avaliação Criteriosa da Qualidade Técnica do Output Gerado

A simples capacidade de gerar código automaticamente não seria, por si só, suficiente para validar a utilidade da solução. Uma **avaliação rigorosa e multifacetada da qualidade técnica do *output* gerado** era, portanto, imperativa. Para endereçar este objetivo, foi inicialmente realizado um levantamento da literatura (apresentado na Secção 2.8) para identificar métricas e abordagens relevantes para a avaliação de código gerado por *IA*. Com base neste estudo, foram definidas e subsequentemente aplicadas (no Capítulo 5) um conjunto de métricas objetivas. Estas métricas foram concebidas para abranger diversas dimensões da qualidade, incluindo, mas não se limitando a: fidelidade visual da interface gerada em comparação com o *mockup* original (avaliada através de técnicas de comparação de imagem e extração de características visuais), correção estrutural e semântica do código *HTML*, *CSS* e *TypeScript* produzido, e conformidade com as boas práticas de desenvolvimento *Angular*.

Para operacionalizar esta avaliação de forma eficiente e sistemática, foi desenvolvido um *pipeline* de testes automatizado, cuja implementação é detalhada na Secção 5.2. Este *pipeline* combinou a orquestração de tarefas utilizando *scripts* em *Node.js* (para submeter *mockups* à *Blueprint-AI*, receber o código gerado e integrá-lo num projeto *Angular* de teste) com *scripts* em *Python* equipados com bibliotecas especializadas em visão computacional (como *EasyOCR* e *OpenCV*) e processamento de imagem (*PIL*) para a análise comparativa entre o *mockup* e a interface renderizada. Os resultados quantitativos obtidos através desta avaliação automatizada revelaram uma elevada precisão e fidelidade na replicação de interfaces com

estruturas mais simples e que utilizavam *layouts* convencionais. No entanto, este processo também permitiu identificar, de forma objetiva, os desafios que a solução enfrenta ao lidar com *mockups* de maior complexidade, como aqueles com alta densidade de informação, elementos gráficos não standard, ou sobreposição de texto, onde as limitações inerentes a tecnologias como o Reconhecimento Ótico de Caracteres (*OCR*) podem influenciar negativamente a qualidade do *prompt* intermédio e, por conseguinte, a fidelidade do código final gerado. Apesar destas limitações identificadas, a análise visual qualitativa, lado a lado, facilitada pelas ferramentas de comparação de imagem desenvolvidas (como exemplificado nas Figuras 19 e 20 da tese), continuou a ser uma ferramenta poderosa e intuitiva para a validação do desempenho global e para a identificação precisa de áreas que requerem melhoria e otimização futuras.

6.1.4 Exploração de Estratégias de Otimização com Inteligência Artificial e Decisão Fundamentada

Uma componente fundamental da investigação realizada consistiu na **exploração e comparação crítica de diferentes estratégias baseadas em Inteligência Artificial** para a tarefa de geração de código, conforme detalhado no Capítulo 4. A fase de experimentação inicial foi dedicada a investigar a aplicabilidade de duas abordagens proeminentes na literatura: *RAG* e o *fine-tuning* direto de *LLM* com dados puramente textuais. A abordagem *RAG*, que conceptualmente visa enriquecer o contexto fornecido ao *LLM* através da recuperação de exemplos relevantes de uma base de conhecimento externa, demonstrou-se, no contexto específico da geração de código *frontend* a partir de descrições de *mockups*, menos eficaz do que inicialmente antecipado. Os modelos genéricos, mesmo quando suplementados com contexto relevante, apresentaram dificuldades em produzir estruturas sintáticas consistentemente válidas e em aderir estritamente às convenções de *frameworks* específicos como o *Angular*.

Por outro lado, o *fine-tuning* de *LLM* utilizando pares de instrução-código (onde a instrução seria uma descrição textual detalhada da interface e o código seria a sua implementação *Angular*) apresentou resultados mais promissores em termos de consistência estilística e alinhamento com o domínio desejado. Contudo, esta abordagem revelou-se altamente dispendiosa e proibitiva em termos práticos. Os custos computacionais associados ao treino de *LLM* de grande escala são significativos, exigindo acesso a *GPU* potentes e longos períodos de processamento. Adicionalmente, a preparação de grandes volumes de dados de treino de alta qualidade, devidamente anotados e estruturados, representa um esforço considerável, muitas vezes inviável para projetos de investigação com recursos limitados ou para aplicações empresariais de nicho que não dispõem de vastos repositórios de dados específicos para esta tarefa.

Perante estas constatações empíricas e após uma análise aprofundada das capacidades, flexibilidade e estruturas de custos das plataformas de *IA* comerciais especializadas (nomeadamente, *OpenAI API* e *Nebius AI Studio*), a decisão tecnológica fundamentada, apresentada e justificada na Secção 4.2, pendeu para a adoção de um ***pipeline* multimodal de**

natureza híbrida. Esta arquitetura estratégica, que constitui o cerne da *Blueprint-AI*, combina de forma inteligente um modelo de visão computacional (fornecido pela *Nebius* AI, escolhida pela sua estrutura de custos mais vantajosa para a análise de imagens em larga escala) para a interpretação inicial do *mockup* e a geração de um *prompt* descritivo detalhado, com um modelo *text-to-text* (neste caso, o *GPT*-4.1 da *OpenAI*, selecionado pela sua superior capacidade de geração de código complexo e pela maior flexibilidade oferecida nos seus processos de *fine-tuning*) que é subsequentemente afinado para gerar o código *frontend* final. Esta solução híbrida permitiu alavancar os pontos fortes de cada tipo de tecnologia – a eficiência e especialização da visão computacional para a interpretação visual e a sofisticação e capacidade de raciocínio dos *LLM* textuais para a geração de código – enquanto mitigava, de forma pragmática, as suas respetivas limitações em termos de custo, complexidade de adaptação e requisitos de dados. Esta abordagem revelou-se a mais equilibrada e otimizada para os objetivos e constrangimentos do presente projeto.

6.1.5 Redução Quantificável do Esforço de Desenvolvimento Manual

Finalmente, um dos objetivos mais pragmáticos e com impacto direto na prática de desenvolvimento de *software* era **quantificar, de forma objetiva, os ganhos em produtividade, eficiência e consistência** que poderiam ser obtidos com a adoção da solução *Blueprint-AI*, quando comparada com as abordagens convencionais de implementação manual de interfaces gráficas. Esta quantificação foi meticulosamente realizada através de uma análise comparativa detalhada dos tempos operacionais e dos custos de utilização associados a cada abordagem, conforme exposto e discutido nas Secções 5.3 e 5.4 da tese. Para este efeito, foi conduzido um estudo de caso comparativo, no qual um programador com perfil júnior foi incumbido de implementar manualmente um conjunto de *mockups* idênticos àqueles que foram processados pela *Blueprint-AI*. Os tempos de cada tarefa foram rigorosamente cronometrados e os custos associados calculados.

Os resultados desta análise comparativa foram reveladores e demonstraram uma redução substancial e inequívoca no tempo de desenvolvimento. Enquanto a implementação manual de um *mockup* típico consumiu, em média, 69 minutos de trabalho do programador, o *pipeline* de *IA* completo da *Blueprint-AI* (incluindo a fase de análise visual pelo modelo *Nebius* e a fase de geração de código pelo modelo *OpenAI*) executou a mesma tarefa em aproximadamente 1 minuto de tempo de processamento das *API*. A este tempo de processamento automatizado, somaram-se, em média, cerca de 5 minutos dedicados pelo programador para realizar os ajustes manuais residuais (como correções de *imports* ou pequenas refatorações) e para integrar o código gerado no projeto principal. Mesmo contabilizando esta intervenção humana final, a redução total de tempo foi da ordem dos 90%. Esta drástica redução de tempo traduziu-se, naturalmente, numa redução de custo igualmente expressiva: o custo estimado do desenvolvimento puramente manual por interface foi calculado em aproximadamente €20,70 (considerando um custo/hora de referência para um programador júnior), enquanto o custo total da abordagem assistida por *IA*, incluindo os custos irrisórios de utilização das *API* (inferiores a €0,01 por ciclo completo) e o tempo de ajuste

humano, foi de cerca de €1,51 por interface. Estes resultados demonstram inequivocamente que, mesmo considerando a necessidade de uma supervisão e intervenção manual complementar, a solução *Blueprint-AI* possui o potencial de acelerar significativamente o ciclo de desenvolvimento inicial de interfaces, permitindo que os programadores dediquem o seu tempo e esforço a tarefas de maior complexidade lógica, resolução de problemas de negócio e inovação, em vez de se concentrarem em tarefas repetitivas de tradução de *designs* em código.

Em conclusão, a conjugação destes resultados evidencia que os objetivos propostos para esta dissertação foram, na sua generalidade, atingidos com sucesso. O trabalho desenvolvido não apenas validou a viabilidade técnica da geração inteligente de interfaces gráficas para a *web* em contextos empresariais, mas também culminou num protótipo funcional (*Blueprint-AI*) que serve como uma plataforma para futuras investigações e como uma demonstração tangível dos benefícios desta abordagem. As aprendizagens recolhidas sobre a seleção de modelos de *IA*, as estratégias de integração, os desafios de *fine-tuning* e a importância da avaliação contínua são contributos valiosos para a área da engenharia de *software* assistida por Inteligência Artificial.

6.2 Limitações

Apesar dos resultados promissores e do alcance substancial dos objetivos, é crucial reconhecer, com transparência, as limitações inerentes ao presente estudo e à solução *Blueprint-AI*. Estas limitações contextualizam os resultados, identificam as fronteiras da tecnologia atual e orientam futuros desenvolvimentos. Uma das principais limitações reside na **fidelidade visual da replicação de *mockups* particularmente complexos ou com estilos não convencionais**. Embora o módulo de visão computacional demonstre boa capacidade, elementos gráficos atípicos, tipografias raras ou sobreposições densas podem levar a interpretações erradas, resultando em desvios entre o *design* original e o código gerado, como a omissão de ícones específicos observada nos testes.

Outro ponto de constrangimento relaciona-se com a **capacidade de generalização e "criatividade" dos *LLM*** quando confrontados com requisitos de *design* ou funcionalidades que divergem significativamente dos seus dados de treino. Mesmo com *fine-tuning*, a eficácia dos *LLM* diminui para cenários que exigem inovação radical, tendendo a produzir soluções mais genéricas ou menos otimizadas, o que reforça o papel da *Blueprint-AI* mais como aceleradora para padrões estabelecidos do que para a exploração de paradigmas de interface completamente novos. A **dependência de *API* externas** (*OpenAI*, *Nebius* AI), embora permita acesso a modelos de ponta, introduz riscos operacionais como custos variáveis, latência, disponibilidade de serviço e alterações nas políticas dos fornecedores, além de um processo de *fine-tuning* que opera como uma "caixa-preta", com controlo limitado sobre os seus mecanismos internos.

É fundamental também delimitar o **âmbito da geração de código, que se foca primariamente em componentes de *UI* estáticos ou com interatividade básica**. A *Blueprint-*

AI não abrange, na sua forma atual, a geração de lógica de negócio complexa, gestão de estado avançada, configuração de rotas complexas ou integrações profundas com *backends*, sendo mais uma ferramenta de *scaffolding* inicial do que uma geradora de aplicações completas. Consequentemente, a **integração e manutenção contínua do código gerado** permanecem como responsabilidades da equipa de desenvolvimento. O código produzido, embora estruturado, necessita de ser compreendido, integrado e mantido para evitar a acumulação de dívida técnica.

Por fim, as **limitações da avaliação experimental** devem ser consideradas. O conjunto de *mockups* de teste, embora diversificado, é finito e pode não representar todos os cenários de *design* possíveis, exigindo cautela na generalização dos resultados. As métricas objetivas podem não capturar todos os aspetos qualitativos de uma interface, como usabilidade ou acessibilidade profunda, e a comparação com o desenvolvimento manual, baseada num único programador júnior, serve como uma linha de base indicativa, mas não universal. O reconhecimento destas limitações é essencial para direcionar os esforços futuros no sentido de robustecer e expandir as capacidades da *Blueprint-AI*.

6.3 Trabalho Futuro

Os resultados encorajadores e as aprendizagens obtidas com a *Blueprint-AI* abrem um leque promissor de oportunidades para investigações e desenvolvimentos futuros, visando não só superar as limitações identificadas, mas também expandir significativamente as capacidades e o impacto da solução. Uma direção prioritária consiste no **aprimoramento contínuo da precisão e robustez da interpretação visual dos *mockups***. Isto poderá envolver a exploração de modelos de visão computacional mais avançados, com maior capacidade de generalização para *designs* complexos ou estilisticamente diversos, e a implementação de técnicas para uma segmentação mais granular e hierárquica dos elementos da interface, resultando em *prompts* textuais intermédios de maior fidelidade.

Outra vertente crucial para evolução futura é a **melhoria da capacidade de geração de código para além da estrutura e estilo, abrangendo interatividade básica e lógica de apresentação**. A investigação poderia focar-se em permitir que os utilizadores especifiquem comportamentos de forma intuitiva, talvez através de anotações nos *mockups* ou linguagem natural, e em treinar *LLM* para traduzir estas especificações em código *TypeScript* funcional e idiomático para *frameworks* como o *Angular*. Adicionalmente, para aumentar a relevância da *Blueprint-AI* em contextos empresariais específicos, seria de grande valor a **integração com sistemas de *design* e bibliotecas de componentes personalizados da organização**, permitindo que o código gerado utilize diretamente os ativos de desenvolvimento internos, o que poderia ser alcançado através de *fine-tuning* mais direcionado ou técnicas de *RAG*.

A implementação de **mecanismos de *feedback* explícito e aprendizagem contínua** é outra área promissora. Permitir que os programadores avaliem e corrijam o código gerado poderia alimentar um ciclo de refinamento iterativo dos modelos de *IA*, adaptando-os

progressivamente às preferências e padrões da equipa. A longo prazo, o **suporte a múltiplos *frameworks* e linguagens de *frontend***, como *React* ou *Vue*.js, aumentaria drasticamente a aplicabilidade da solução. A **automatização da integração do código gerado** em projetos existentes, possivelmente com sugestões de código assistidas por *IA*, também reduziria significativamente o esforço manual remanescente.

Uma direção de investigação particularmente transformadora seria a evolução da Blueprint-AI para um paradigma de ***Agentic AI***. Diferentemente da abordagem gerativa atual, que responde a um pedido específico, um sistema agente seria projetado para perseguir objetivos complexos com mínima intervenção humana, demonstrando capacidades avançadas de planeamento, raciocínio e utilização de ferramentas (Acharya et al., 2025; Lou et al., 2025). Isto representaria uma mudança fundamental do papel do programador, de um utilizador que guia a ferramenta para um supervisor que delega autonomia a um agente de *software* (Sapkota et al., 2025). Em vez de gerar o código para um único *mockup*, o programador poderia solicitar a implementação de uma funcionalidade completa, como "integrar uma nova *API* de pagamentos", e o agente assumiria a responsabilidade de planear e executar todas as etapas necessárias. A exploração de arquiteturas agentes, como as baseadas no paradigma *AIM* (*Agentic Iterative Monologue*) (Lupoiu et al., 2025), poderia transformar a ferramenta de um gerador de código para um verdadeiro engenheiro de software autónomo, capaz de lidar com a complexidade do desenvolvimento de funcionalidades completas (Jiang et al., 2025).

Finalmente, para validar de forma mais abrangente a solução, serão necessários **estudos de usabilidade e avaliação em contextos reais de desenvolvimento de larga escala**, envolvendo equipas diversas e investigando não só métricas técnicas, mas também o impacto na dinâmica da equipa e na satisfação dos programadores. Paralelamente, a **exploração contínua das implicações éticas e da governança da geração de código por *IA*** é fundamental, abordando questões de propriedade intelectual, segurança e o impacto no mercado de trabalho. O prosseguimento da investigação nestas direções tem o potencial de *solid*ificar a *Blueprint-AI* como uma ferramenta transformadora no desenvolvimento de *software*.

Referências

Abdullahi, T., Singh, R., & Eickhoff, C. (2024). *Retrieval Augmented Zero-Shot Text Classification*. https://doi.org/10.1145/3664190.3672514

Acharya, D. B., Kuppan, K., & Divya, B. (2025). Agentic AI: Autonomous Intelligence for Complex Goals - A Comprehensive Survey. *IEEE Access*. https://doi.org/10.1109/ACCESS.2025.3532853

Agbozo, E. (2023). A Hybrid Data-Driven Web-Based UI-UX Assessment Model. *ArXiv.Org*. https://doi.org/https://doi.org/10.48550/arXiv.2301.08992

Ahmed Ali Linkon, Mujiba Shaima, Md Shohail Uddin Sarker, Badruddowza, Norun Nabi, Md Nasir Uddin Rana, Sandip Kumar Ghosh, Hammed Esa, & Faiaz Rahat Chowdhury. (2024). Advancements and Applications of Generative Artificial Intelligence and Large Language Models on Business Management: A Comprehensive Review. *Journal of Computer Science and Technology Studies*, *6*, 225–232. https://doi.org/10.32996/jcsts.2024.6.1.26

AI's Impact on Traditional Software Development. (2025). https://doi.org/10.47363/JAICC/2024(3)E145

Ajimati, M. O., Carroll, N., & Maher, M. (2025). Adoption of low-code and no-code development: A systematic literature review and future research agenda. *Journal of Systems and Software*, *222*. https://doi.org/10.1016/j.jss.2024.112300

Akter, S. N., Madaan, A., Lee, S., Yang, Y., & Nyberg, E. (2024). *Self-Imagine: Effective Unimodal Reasoning with Multimodal Models using Self-Imagination*.

Alassan, M. S. Y., Espejel, J. L., Bouhandi, M., Dahhane, W., & Ettifouri, E. H. (2024). *Benchmarking Open-Source Language Models for Efficient Question Answering in Industrial Applications*.

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., Ring, R., Rutherford, E., Cabi, S., Han, T., Gong, Z., Samangooei, S., Monteiro, M., Menick, J., Borgeaud, S., … Simonyan, K. (2022). Flamingo: a Visual Language Model for Few-Shot Learning. *Neural Information Processing Systems*.

Alikhashashneh, E., Raje, R., & Hill, J. (2018). Using software engineering metrics to evaluate the quality of static code analysis tools. *Proceedings - 2018 1st International Conference on Data Intelligence and Security, ICDIS 2018*, 65–72. https://doi.org/10.1109/ICDIS.2018.00017

Amazon. (2024). *Fine-tune Anthropic's Claude 3 Haiku in Amazon Bedrock to boost model accuracy and quality | Amazon Web Services*. https://aws.amazon.com/pt/blogs/machine-learning/fine-tune-anthropics-claude-3-haiku-in-amazon-bedrock-to-boost-model-accuracy-and-quality

Andreev, A., Kotsenko, A., Varlamov, O., Kim, R., & Goryachkin, B. (2024). Text processing using LLM for automatic creation of agricultural crops knowledge bases. *BIO Web of Conferences*.

Anwar, S., & Li, C. (2020). Diving deeper into underwater image enhancement: A survey. *Signal Processing: Image Communication*, *89*, 115978. https://doi.org/10.1016/j.image.2020.115978

Ara, J., Sik-Lanyi, C., & Kelemen, A. (2024). Accessibility engineering in web evaluation process: a systematic literature review. In *Universal Access in the Information Society* (Vol. 23, pp. 653–686). Springer Science and Business Media Deutschland GmbH. https://doi.org/10.1007/s10209-023-00967-2

Asai, A., Wu, Z., Wang, Y., Sil, A., & Hajishirzi, H. (2024). SELF-RAG: LEARNING TO RETRIEVE, GENERATE, AND CRITIQUE THROUGH SELF-REFLECTION. *12th International Conference on Learning Representations, ICLR 2024*.

Asghar, M. Z., Abbas, M., Zeeshan, K., Kotilainen, P., & Hämäläinen, T. (2019). Assessment of Deep Learning Methodology for Self-Organizing 5G Networks. *Applied Sciences*, *9*(15), 2975. https://doi.org/10.3390/app9152975

Asiroglu, B., Mete, B. R., Yildiz, E., Nalcakan, Y., Sezen, A., Dağtekin, M., & Ensari, T. (2019). Automatic HTML Code Generation from Mock-Up Images Using Machine Learning Techniques. *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT)*.

Aşiroğlu, B., Mete, B. R., Yildiz, E., Nalçakan, Y., Sezen, A., Dağtekin, M., & Ensari, T. (2019, April). Automatic HTML code generation from mock-up images using machine learning techniques. *2019 Scientific Meeting on Electrical-Electronics and Biomedical Engineering and Computer Science, EBBT 2019*. https://doi.org/10.1109/EBBT.2019.8741736

Atoum, I., & Bong, C. H. (2015). *Measuring Software Quality in Use: State-of-the-Art and Research Challenges*.

Banerjee, S., Agarwal, A., & Singla, S. (2024). LLMs Will Always Hallucinate, and We Need to Live With This. *ArXiv.Org*.

Baqar, M. (2024). Balancing Innovation and Ethics in AI-Driven Software Development. *ArXiv.Org*.

Barbon, S., Ceravolo, P., Groppe, S., Jarrar, M., Maghool, S., Sèdes, F., Sahri, S., & Van Keulen, M. (2024, June). Are Large Language Models the New Interface for Data Pipelines? *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BIDEDE 2024, in Conjunction with the 2024 ACM SIGMOD/PODS Conference*. https://doi.org/10.1145/3663741.3664785

Barnett, S., Kurniawan, S., Thudumu, S., Brannelly, Z., & Abdelrazek, M. (2024). Seven Failure Points When Engineering a Retrieval Augmented Generation System. *2024 IEEE/ACM 3rd International Conference on AI Engineering – Software Engineering for AI (CAIN)*.

Barua, S. S., Zulkarnain, I. M., Roy, A., Alam, Md. G. R., & Uddin, Md. Z. (2022). Sketch2FullStack: Generating Skeleton Code of Full Stack Website and Application from Sketch using Deep Learning and Computer Vision. *ArXiv.Org*.

Baul'e, D., Wangenheim, C. V, Wangenheim, A. V, Hauck, J., & J'unior, E. C. V. (2021). Automatic code generation from sketches of mobile applications in end-user development using Deep Learning. *ArXiv.Org*.

Bechard, P., Wang, C., Abaskohi, A., Rodriguez, J. A., Pal, C., Vázquez, D., Gella, S., Rajeswar, S., & Taslakian, P. (2025). StarFlow: Generating Structured Workflow Outputs From Sketch Images. *ArXiv.Org*.

Beltramelli, T. (2017). pix2code: Generating Code from a Graphical User Interface Screenshot. *Engineering Interactive Computing System*.

Bi, Z., Wan, Y., Wang, Z., Zhang, H., Guan, B., Lu, F., Zhang, Z., Sui, Y., Shi, X., & Jin, H. (2024). Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback. *Annual Meeting of the Association for Computational Linguistics*.

Bilousova, L. I., Gryzun, L. E., & Zhytienova, N. V. (2021). Fundamentals of UI/UX design as a component of the pre-service specialist's curriculum. *SHS Web of Conferences*, *104*, 02015. https://doi.org/10.1051/shsconf/202110402015

Birkstedt, T., Minkkinen, M., Tandon, A., & Mäntymäki, M. (2023). AI governance: themes, knowledge gaps and future agendas. *Internet Research*.

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, *5*, 135–146. https://doi.org/10.1162/tacl_a_00051

Bouças, T., & Esteves, A. (2020). Converting Web Pages Mockups to HTML using Machine Learning. *International Conference on Web Information Systems and Technologies*.

Bovenzi, G., Cerasuolo, F., Ciuonzo, D., Di Monda, D., Guarino, I., Montieri, A., Persico, V., & Pescapè, A. (2025). *Mapping the Landscape of Generative AI in Network Monitoring and Management*. https://doi.org/10.1109/TNSM.2025.3543022

Brendel, W., Rauber, J., & Bethge, M. (2018). Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R.,

Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., … Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, *2020-December*.

Bühler, K., Höllt, T., Schulz, T., & V'azquez, P.-P. (2024). AI-in-the-loop: The future of biomedical visual analytics applications in the era of AI. *ArXiv.Org*.

Cai, A., Rick, S. R., Heyman, J. L., Zhang, Y., Filipowicz, A., Hong, M., Klenk, M., & Malone, T. (2023). DesignAID: Using Generative AI and Semantic Diversity for Design Inspiration. *Proceedings of The ACM Collective Intelligence Conference*, 1–11. https://doi.org/10.1145/3582269.3615596

Cai, W., Shi, T., Zhao, X., & Song, D. (2025). *Are You Getting What You Pay For? Auditing Model Substitution in LLM APIs*.

Campo, D. N., Conde, J., Alonso, Á., Huecas, G., Salvachúa, J., & Reviriego, P. (2025). *Real-time Spatial Retrieval Augmented Generation for Urban Environments*.

Carney, M., Webster, B., Alvarado, I., Phillips, K., Howell, N., Griffith, J., Jongejan, J., Pitaru, A., & Chen, A. (2020). Teachable Machine: Approachable Web-Based Tool for Exploring Machine Learning Classification. *CHI Extended Abstracts*.

Castillo-Eslava, F., Mougan, C., Romero-Reche, A., & Staab, S. (2023). *The Role of Large Language Models in the Recognition of Territorial Sovereignty: An Analysis of the Construction of Legitimacy*.

Chang, D. T. (2023). *Concept-Oriented Deep Learning with Large Language Models*.

Chatzitofis, A., Saroglou, L., Boutis, P., Drakoulis, P., Zioulis, N., Subramanyam, S., Kevelham, B., Charbonnier, C., Cesar, P., Zarpalas, D., Kollias, S., & Daras, P. (2020). HUMAN4D: A Human-Centric Multimodal Dataset for Motions and Immersive Media. *IEEE Access*, *8*, 176241–176262. https://doi.org/10.1109/ACCESS.2020.3026276

Chemnitz, L., Reichenbach, D., Aldebes, H., Naveed, M., Narasimhan, K., & Mezini, M. (2023). Towards Code Generation from BDD Test Case Specifications: A Vision. *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*.

Chen, H., Deng, W., Yang, S., Xu, J., Jiang, Z., Ngai, E. C. H., Liu, J., & Liu, X. (2025). Towards Edge General Intelligence via Large Language Models: Opportunities and Challenges. *IEEE Network*. https://doi.org/10.1109/MNET.2025.3539338

Chen, S., & Lin, G. (2024). *LLM Reasoning Engine: Specialized Training for Enhanced Mathematical Reasoning*.

Chen, S.-S., Lee, J., & Liang, P. P. (2025). Interactive Sketchpad: A Multimodal Tutoring System for Collaborative, Visual Problem-Solving. *CHI Extended Abstracts*.

Chen, X., Liu, C., Li, B., Lu, K., & Song, D. (2017). *Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning*.

Chung, J. J. Y., & Adar, E. (2023). Artinter: AI-powered Boundary Objects for Commissioning Visual Arts. *Conference on Designing Interactive Systems*.

Chung, J. J. Y., Kim, W., Yoo, K. M., Lee, H., Adar, E., & Chang, M. (2022). TaleBrush: Sketching Stories with Generative Pretrained Language Models. *International Conference on Human Factors in Computing Systems*.

Clop, C., & Teglia, Y. (2024). *Backdoored Retrievers for Prompt Injection Attacks on Retrieval Augmented Generation of Large Language Models*.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. *Neural Information Processing Systems*.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, *1*, 4171–4186.

Dhar, R., Kakran, A., Karan, A., Vaidhyanathan, K., & Varma, V. (2025). *DRAFT-ing Architectural Design Decisions using LLMs*.

Di, P., Li, J., Yu, H., Jiang, W., Cai, W., Cao, Y., Chen, C., Chen, D., Chen, H., Chen, L., Fan, G., Gong, J., Gong, Z., Hu, W., Guo, T., Lei, Z., Li, T., Li, Z., Liang, M., … Zhu, X. (2023). CodeFuse-13B: A Pretrained Multi-Lingual Code Large Language Model. *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

Dibia, V. C. (2023). LIDA: A Tool for Automatic Generation of Grammar-Agnostic Visualizations and Infographics using Large Language Models. *Annual Meeting of the Association for Computational Linguistics*.

Ding, Z., Zhang, Q., Chi, M., & Wang, Z. (2025). Frontend Diffusion: Empowering Self-Representation of Junior Researchers and Designers Through Agentic Workflows. *ArXiv.Org*.

Du, S., Zhao, J., Shi, J., Xie, Z., Jiang, X., Bai, Y., & He, L. (2025). *A Survey on the Optimization of Large Language Model-based Agents*.

Duan, P., Wierzynski, C., & Nachman, L. (2020). Optimizing User Interface Layouts via Gradient Descent. *International Conference on Human Factors in Computing Systems*.

Edwards, C. (2024). *Hybrid Context Retrieval Augmented Generation Pipeline: LLM-Augmented Knowledge Graphs and Vector Database for Accreditation Reporting Assistance*.

Esposito, A., Calvano, M., Curci, A., Desolda, G., Lanzilotti, R., Lorusso, C., & Piccinno, A. (2023). End-User Development for Artificial Intelligence: A Systematic Literature Review. *International Symposium on End-User Development*.

Esposito, A., Calvano, M., Curci, A., Greco, F., Lanzilotti, R., of Bari Aldo Moro, A. P. U., & of Pisa, U. (2025). *Explanation-Driven Interventions for Artificial Intelligence Model Customization: Empowering End-Users to Tailor Black-Box AI in Rhinocytology*.

Evtikhiev, M., Bogomolov, E., Sokolov, Y., & Bryksin, T. (2023a). Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software*, *203*, 111741. https://doi.org/https://doi.org/10.1016/j.jss.2023.111741

Evtikhiev, M., Bogomolov, E., Sokolov, Y., & Bryksin, T. (2023b). Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software*, *203*, 111741. https://doi.org/10.1016/j.jss.2023.111741

Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., & Li, Q. (2024). A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. *Knowledge Discovery and Data Mining*.

Farahani, M. S., & Ghasemi, G. (2024). Will artificial intelligence threaten humanity? *Sustainable Economies*.

Floridi, L., Cowls, J., King, T. C., & Taddeo, M. (2020). How to Design AI for Social Good: Seven Essential Factors. *Science and Engineering Ethics*.

Gan, A., Yu, H., Zhang, K., Liu, Q., Yan, W., Huang, Z., Tong, S., & Hu, G. (2025). *Retrieval Augmented Generation Evaluation in the Era of Large Language Models: A Comprehensive Survey*.

Gao, M., Zhao, J., Lin, Z., Ding, W., Hou, X., Feng, Y., Li, C., & Guo, M. (2024). AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs. *ICCD*.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Guo, Q., Wang, M., & Wang, H. (2023). Retrieval-Augmented Generation for Large Language Models: A Survey. *ArXiv.Org*.

Gao, Y., Xiong, Y., Zhong, Y., Bi, Y., Xue, M., & Wang, H. (2025). *Synergizing RAG and Reasoning: A Systematic Review*.

Gardey, J. C., Garrido, A., Firmenich, S., Grigera, J., & Rossi, G. (2020). UX-Painter: An Approach to Explore Interaction Fixes in the Browser. *Proc. ACM Hum. Comput. Interact.*

Ge, T., Liu, Y., Ye, J., Li, T., & Wang, C. (2025). Advancing vision-language models in front-end development via data synthesis. *ArXiv.Org*.

Ghodratnama, S., & Zakershahrak, M. (2023). Adapting LLMs for Efficient, Personalized Information Retrieval: Methods and Implications. *ICSOC Workshops*.

Gholami, S. (2024). Do generative large language models need billions of parameters? In *Redefining Security With Cyber AI* (pp. 37–55). IGI Global. https://doi.org/10.4018/979-8-3693-6517-5.ch003

Grover, A., & Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, *13-17-August-2016*, 855–864. https://doi.org/10.1145/2939672.2939754

Gui, Y., Li, Z., Wan, Y., Shi, Y., Zhang, H., Su, Y., Chen, B., Chen, D., Wu, S., Zhou, X., Jiang, W., Jin, H., & Zhang, X. (2024). WebCode2M: A Real-World Dataset for Code Generation from Webpage Designs. *The Web Conference*.

Guo, T., Guo, K., Nan, B., Liang, Z., Guo, Z., Chawla, N. V., Wiest, O., & Zhang, X. (2023). What can Large Language Models do in chemistry? A comprehensive benchmark on eight tasks. *Advances in Neural Information Processing Systems*, *36*.

Gürcan, Ö. (2024). LLM-Augmented Agent-Based Modelling for Social Simulations: Challenges and Opportunities. *Frontiers in Artificial Intelligence and Applications* , *386*, 134–144. https://doi.org/10.3233/FAIA240190

Haghir Chehreghani, M. (2024). The embeddings world and Artificial General Intelligence. In *Cognitive Systems Research* (Vol. 84). Elsevier B.V. https://doi.org/10.1016/j.cogsys.2023.101201

Haque, Md. A. (2024). LLMs: A Game-Changer for Software Engineers? *ArXiv.Org*.

Harbi, S. Al, Tidjon, L., & Khomh, F. (2023). Responsible Design Patterns for Machine Learning Pipelines. *ArXiv.Org*.

Harbola, C., & Purwar, A. (2025). *KnowsLM: A framework for evaluation of small language models for knowledge augmentation and humanised conversations*.

Harrison Oke Ekpobimi, Regina Coelis Kandekere, & Adebamigbe Alex Fasanmade. (2024). The future of software development: integrating AI and machine learning into front-end technologies. *Global Journal of Advanced Research and Reviews*, *2*, 069–077. https://doi.org/10.58175/gjarr.2024.2.1.0031

He, J.-Y., Wang, Y., Wang, L., Lu, H., He, J.-Y., Li, C., Chen, H., Lan, J., Luo, B., & Geng, Y. (2024). GLDesigner: Leveraging Multi-Modal LLMs as Designer for Enhanced Aesthetic Text Glyph Layouts. *ArXiv.Org*.

Heitz, L. B., Chamas, J., & Scherb, C. (2024). *Evaluation of the Programming Skills of Large Language Models*.

Hessel, J., Holtzman, A., Forbes, M., Le Bras, R., & Choi, Y. (2021a). CLIPScore: A Reference-free Evaluation Metric for Image Captioning. *EMNLP 2021 - 2021 Conference on Empirical Methods in Natural Language Processing, Proceedings*, 7514–7528. https://doi.org/10.18653/v1/2021.emnlp-main.595

Hessel, J., Holtzman, A., Forbes, M., Le Bras, R., & Choi, Y. (2021b). CLIPScore: A Reference-free Evaluation Metric for Image Captioning. *EMNLP 2021 - 2021 Conference on Empirical*

Methods in Natural Language Processing, Proceedings, 7514–7528. https://doi.org/10.18653/v1/2021.emnlp-main.595

Huang, W., & Yang, P. (2022). Application Analysis and Research of Artificial Intelligence Technology in the Creative Stage of Web Design. *BCP Social Sciences & Humanities*.

Huang, W., Zhao, X., Banks, A., Cox, V., & Huang, X. (2024). Hierarchical Distribution-aware Testing of Deep Learning. *ACM Transactions on Software Engineering and Methodology*, *33*(2), 1–35. https://doi.org/10.1145/3625290

Iorliam, A., & Ingio, J. A. (2024). A Comparative Analysis of Generative Artificial Intelligence Tools for Natural Language Processing. *Journal of Computing Theories and Applications*.

J, V. I. (2023). *Towards Making Flowchart Images Machine Interpretable*. Https://Vl2g.Github.Io/Projects/Floco/.

Jain, V., Agrawal, P., Banga, S., Kapoor, R., & Gulyani, S. (2019). Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network. *ArXiv.Org*.

Jeong, C. (2025). Beyond Text: Implementing Multimodal Large Language Model-Powered Multi-Agent Systems Using a No-Code Platform. *Journal of Intelligence and Information Systems*.

Jiang, S., Xie, M., Chen, F. Y., Ma, J., & Luo, J. (2025). *Intelligent Design 4.0: Paradigm Evolution Toward the Agentic AI Era*.

Jin, C., Zhang, Z., Jiang, X., Liu, F., Liu, X., Liu, X., & Jin, X. (2024). *RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation*.

Jordan, B., Devasia, N., Hong, J., Williams, R., & Breazeal, C. (2021). PoseBlocks: A Toolkit for Creating (and Dancing) with AI. *AAAI Conference on Artificial Intelligence*.

Kang, D., Cho, J., Jeon, Y., Jang, S., Lee, M., Cho, J., & Lee, G. G. (2025). Retrieval-Augmented Fine-Tuning With Preference Optimization For Visual Program Generation. *ArXiv.Org*.

Kanhirakadavath, M. R., & Chandran, M. S. M. (2022). Investigation of Eye-Tracking Scan Path as a Biomarker for Autism Screening Using Machine Learning Algorithms. *Diagnostics*, *12*. https://doi.org/10.3390/diagnostics12020518

Karakus, C., Huilgol, R., Wu, F., Subramanian, A., Daniel, C., Cavdar, D., Xu, T., Chen, H., Rahnama, A., & Quintela, L. (2021). *Amazon SageMaker Model Parallelism: A General and Flexible Framework for Large Model Training*.

Kass, S., Strahringer, S., & Westner, M. (2023a). A Multiple Mini Case Study on the Adoption of Low Code Development Platforms in Work Systems. *IEEE Access*, *11*, 118762–118786. https://doi.org/10.1109/ACCESS.2023.3325092

Kass, S., Strahringer, S., & Westner, M. (2023b). Practitioners' Perceptions on the Adoption of Low Code Development Platforms. *IEEE Access*, *11*, 29009–29034. https://doi.org/10.1109/ACCESS.2023.3258539

Khatiwada, K., Hopper, J., Cheatham, J., Joshi, A., & Baidya, S. (2025). *Large Language Models in the IoT Ecosystem -- A Survey on Security Challenges and Applications*.

Khoramnejad, F., & Hossain, E. (2024). *Generative AI for the Optimization of Next-Generation Wireless Networks: Basics, State-of-the-Art, and Open Challenges*. https://doi.org/10.1109/COMST.2025.3535554

Kolthoff, K., Kretzer, F., Bartelt, C., Maedche, A., & Ponzetto, S. P. (2025). *GUIDE: LLM-Driven GUI Generation Decomposition for Automated Prototyping*.

Kumar, A., Finley, B., Braud, T., Tarkoma, S., & Hui, P. (2021). Sketching an AI Marketplace: Tech, Economic, and Regulatory Aspects. *IEEE Access*, *9*, 13761–13774. https://doi.org/10.1109/ACCESS.2021.3050929

Labazanova, S., Aygumov, T. G., & Mursaliev, M. Kh. (2024). Issues with generative artificial intelligence tools. *ITM Web of Conferences*.

Laurençon, H., Tronchon, L., & Sanh, V. (2024). Unlocking the conversion of Web Screenshots into HTML Code with the WebSight Dataset. *ArXiv.Org*.

Le, C. C., Vinh, H. C. T., Phan, H. N., Le, D. D., Nguyen, T. N., & Bui, N. D. Q. (2024). VISUALCODER: Guiding Large Language Models in Code Execution with Fine-grained Multimodal Chain-of-Thought Reasoning. *ArXiv.Org*.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Neural Information Processing Systems*.

Li, D., Li, J., & Hoi, S. C. H. (2023). BLIP-Diffusion: Pre-trained Subject Representation for Controllable Text-to-Image Generation and Editing. *Neural Information Processing Systems*.

Li, H., Pan, Y., Zhao, J., & Zhang, L. (2021). Skin disease diagnosis with deep learning: A review. *Neurocomputing*, *464*, 364–393. https://doi.org/10.1016/j.neucom.2021.08.096

Li, T. J.-J., Azaria, A., & Myers, B. (2017). SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. *International Conference on Human Factors in Computing Systems*.

Li, T. J.-J., Labutov, I., Li, X. N., Zhang, X., Shi, W., Ding, W., Mitchell, T. M., & Myers, B. (2018). APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. *IEEE Symposium on Visual Languages / Human-Centric Computing Languages and Environments*.

Li, Y., Luo, M., Gong, Y., Lin, C., Jiao, J., Liu, Y., & Huang, K. (2025). DeepThink: Aligning Language Models with Domain-Specific User Intents. *ArXiv.Org*.

Lin, D. C.-E., & Martelaro, N. (2023). Jigsaw: Supporting Designers to Prototype Multimodal Applications by Chaining AI Foundation Models. *International Conference on Human Factors in Computing Systems*.

Lin, J., Guo, J., Sun, S., Yang, Z., Lou, J.-G., & Zhang, D. (2023). LayoutPrompter: Awaken the Design Ability of Large Language Models. *Neural Information Processing Systems*.

Liu, H., Li, C., Li, Y., & Lee, Y. J. (2023). Improved Baselines with Visual Instruction Tuning. *Computer Vision and Pattern Recognition*.

Liu, H., Li, C., Wu, Q., & Lee, Y. J. (2023). Visual Instruction Tuning. *Neural Information Processing Systems*.

Liu, Y., Chen, J., Bi, T., Grundy, J., Wang, Y., Chen, T., Tang, Y., & Zheng, Z. (2024). An Empirical Study on Low Code Programming using Traditional vs Large Language Model Support. *ArXiv.Org*.

Liu, Y., Hu, Q., & Shu, K. (2018). Improving pix2code based Bi-directional LSTM. *2018 IEEE International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*.

Lou, B., Lu, T., Raghu, T. S., & Zhang, Y. (2025). *Unraveling Human-AI Teaming: A Review and Outlook*.

Lu, J., Wang, H., Liu, Z., Liang, K., Bao, L., & Yang, X. (2024). Instructive Code Retriever: Learn from Large Language Model's Feedback for Code Intelligence Tasks. *International Conference on Automated Software Engineering*.

Lu, Y., Leung, A., Swearngin, A., Nichols, J., & Barik, T. (2024). Misty: UI Prototyping Through Interactive Conceptual Blending. *International Conference on Human Factors in Computing Systems*.

Lu, Y., Yang, Y., Zhao, Q., Zhang, C., & Li, T. J.-J. (2024). AI Assistance for UX: A Literature Review Through Human-Centered AI. *ArXiv.Org*.

Lupoiu, R., Shao, Y., Dai, T., Mao, C., Edee, K., & Fan, J. A. (2025). *A multi-agentic framework for real-time, autonomous freeform metasurface design*.

MacHado, C., & Campos, J. C. (2021). Towards the integration of user interface prototyping and model-based development. *ICGI 2021 - 2021 International Conference on Graphics and Interaction, Proceedings*. https://doi.org/10.1109/ICGI54032.2021.9655284

Magadza, T., & Viriri, S. (2021). Deep Learning for Brain Tumor Segmentation: A Survey of State-of-the-Art. *Journal of Imaging*, *7*(2), 19. https://doi.org/10.3390/jimaging7020019

Marjit, S., Singh, H., Mathur, N., Paul, S., Yu, C.-M., & Chen, P.-Y. (2025). DiffuseKronA: A Parameter Efficient Fine-tuning Method for Personalized Diffusion Models. *2025 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 3529–3538. https://doi.org/10.1109/WACV61041.2025.00348

Marron, M. (2024). A New Generation of Intelligent Development Environments. *Ide*.

Mirishli, S. (2025). The Role of Legal Frameworks in Shaping Ethical Artificial Intelligence Use in Corporate Governance. *ArXiv.Org*.

Monteiro, K., Vatsal, R., Chulpongsatorn, N., Parnami, A., & Suzuki, R. (2023). Teachable Reality: Prototyping Tangible Augmented Reality with Everyday Objects by Leveraging Interactive Machine Teaching. *International Conference on Human Factors in Computing Systems*.

Mukherjee, A., & Chang, H. H. (2023). The Creative Frontier of Generative AI: Managing the Novelty-Usefulness Tradeoff. *ArXiv.Org*.

Mungoli, N. (2023). Scalable, Distributed AI Frameworks: Leveraging Cloud Computing for Enhanced Deep Learning Performance and Efficiency. *ArXiv.Org*.

Ng, K. K., Fauzi, L., Leow, L., & Ng, J. (2024). *Harnessing the Potential of Gen-AI Coding Assistants in Public Sector Software Development*.

Nguyen, T., & Csallner, C. (2015). Reverse Engineering Mobile Application User Interfaces with REMAUI (T). *International Conference on Automated Software Engineering*.

O'Donovan, P., Agarwala, A., & Hertzmann, A. (2015). DesignScape: Design with Interactive Layout Suggestions. *International Conference on Human Factors in Computing Systems*.

OpenRouter. (2025). *OpenRouter documentation*. https://openrouter.ai/docs/quickstart

Ozcinar, C., & Rana, A. (2021). Quality Assessment of Super-Resolved Omnidirectional Image Quality Using Tangential Views. *Electronic Imaging*, *33*(9), 295-1-295–297. https://doi.org/10.2352/ISSN.2470-1173.2021.9.IQSP-295

Pacheco, J., Garbatov, S., & Goulao, M. (2021). Improving Collaboration Efficiency Between UX/UI Designers and Developers in a Low-Code Platform. *Companion Proceedings - 24th International Conference on Model-Driven Engineering Languages and Systems, MODELS-C 2021*, 138–147. https://doi.org/10.1109/MODELS-C53483.2021.00025

Pakalapati, N., Venkatasubbu, S., & Sistla, S. M. K. (2023). The Convergence of AI/ML and DevSecOps: Revolutionizing Software Development. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (Online)*, *2*, 189–212. https://doi.org/10.60087/jklst.vol2.n2.p212

Pangtey, L., Bhatnagar, A., Bansal, S., Dar, S. S., & Kumar, N. (2025). *Large Language Models Meet Stance Detection: A Survey of Tasks, Methods, Applications, Challenges and Future Directions*.

Papamichail, M. D., Diamantopoulos, T., & Symeonidis, A. L. (2019). Measuring the reusability of software components using static analysis metrics and reuse rate information. *Journal of Systems and Software*, *158*. https://doi.org/10.1016/j.jss.2019.110423

PaperSpace. (2025). *Fully Managed Notebooks for AI Development | Paperspace*. https://www.paperspace.com/notebooks

Patel, H., Ramanan, B. A., Khan, M. A., Williams, T., Friedman, B., & Drabeck, L. (2024). *Automating Code Adaptation for MLOps -- A Benchmarking Study on LLMs*.

Pei, Q., Wu, L., Gao, K., Zhu, J., Wang, Y., Wang, Z., Qin, T., & Yan, R. (2024). *Leveraging Biomolecule and Natural Language through Multi-Modal Learning: A Survey*.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). Learning Transferable Visual Models From Natural Language Supervision. *International Conference on Machine Learning*.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, *21*.

Rajaram, S., Numan, N., Kumaravel, B. T., Marquardt, N., & Wilson, A. D. (2024). BlendScape: Enabling End-User Customization of Video-Conferencing Environments through Generative AI. *ACM Symposium on User Interface Software and Technology*.

Rao, N., Tsay, J., Kate, K., Hellendoorn, V. J., & Hirzel, M. (2023). AI for Low-Code for AI. *International Conference on Intelligent User Interfaces*.

Replicate. (2025, May). *Replicate API Documentation*. Replicate.

República, D. da. (2020). *Despacho n.º 11171/2020*. Https://Diariodarepublica.Pt/Dr/Detalhe/Despacho/11171-2020-148327696.

RunPod. (2025). *RunPod Documentation - AI Cloud Platform | RunPod Documentation*. https://docs.runpod.io/?_gl=1

Saad, M., López, J. A. H., Chen, B., Ernst, N., Varró, D., & Sharma, T. (2025). *SENAI: Towards Software Engineering Native Generative Artificial Intelligence*.

Sapkota, R., Roumeliotis, K. I., & Karkee, M. (2025). *Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI*.

Sarkar, A. (2023). Will Code Remain a Relevant User Interface for End-User Programming with Generative AI Models? *SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

Schuller, P., Costeira, J. P., Crowley, J., Grosinger, J., Ingrand, F., Kockemann, U., Saffiotti, A., & Welss, M. (2022). Composing Complex and Hybrid AI Solutions. *ArXiv*, *abs/2202.12566*. https://api.semanticscholar.org/CorpusID:247154821

Shanahan, M., & Singler, B. (2024). *Existential Conversations with Large Language Models: Content, Community, and Culture*.

Shi, C., Yang, C., Liu, Y., Shui, B., Wang, J., Jing, M., Xu, L., Zhu, X., Li, S., Zhang, Y., Liu, G., Nie, X., Cai, D., & Yang, Y. (2024). ChartMimic: Evaluating LMM's Cross-Modal Reasoning Capability via Chart-to-Code Generation. *ArXiv.Org*.

Shoaib, M., Shah, B., El-Sappagh, S., Ali, A., Ullah, A., Alenezi, F., Gechev, T., Hussain, T., & Ali, F. (2023). An advanced deep learning models-based plant disease detection: A review of recent research. *Frontiers in Plant Science*, *14*. https://doi.org/10.3389/fpls.2023.1158933

Shukla, S., Gatti, P., Kumar, Y., Yadav, V., & Mishra, A. (2025). Towards Making Flowchart Images Machine Interpretable. *IEEE International Conference on Document Analysis and Recognition*.

Si, C., Zhang, Y., Li, R., Yang, Z., Liu, R., & Yang, D. (2024). *Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering*.

Siirtola, P., & Röning, J. (2019). Incremental Learning to Personalize Human Activity Recognition Models: The Importance of Human AI Collaboration †. *Italian National Conference on Sensors*.

Soudani, H., Kanoulas, E., & Hasibi, F. (2024). Fine Tuning vs. Retrieval Augmented Generation for Less Popular Knowledge. *SIGIR-AP*.

Srinivas, S. S., Sannidhi, G., & Runkana, V. (2024). Retrieval-Augmented Instruction Tuning for Automated Process Engineering Calculations : A Tool-Chaining Problem-Solving Framework with Attributable Reflection. *ArXiv.Org*.

Stige, Å., Zamani, E., Mikalef, P., & Zhu, Y. (2023). Artificial intelligence (AI) for user experience (UX) design: a systematic literature review and future research agenda. *Information Technology and People*.

Suleri, S., Pandian, V. P. S., Shishkovets, S., & Jarke, M. (2019). Eve: A Sketch-based Software Prototyping Workbench. *CHI Extended Abstracts*.

Szymkowiak, T. (2025, June). *Pix2Code: Automating Front-End development*. Https://Theosz.Medium.Com/Pix2code-Automating-Front-End-Development-B9e9087c38e6.

Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. https://doi.org/10.6028/NIST.AI.100-1

Techniques for WCAG 2.2. (n.d.). Https://Www.W3.Org/WAI/WCAG22/Techniques/.

Thakur, A. (2024). *AutoTrain: No-code training for state-of-the-art models*.

Thon, C., Finke, B., Kwade, A., & Schilde, C. (2021). Artificial Intelligence in Process Engineering. *Advanced Intelligent Systems*.

Tinnes, C., Welter, A., & Apel, S. (2024). *Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets*.

Torka, S., & Albayrak, S. (2024). Optimizing AI-Assisted Code Generation. *ArXiv.Org*.

Tsai, C.-C., Yu, C.-M., Lin, Y.-D., Wu, Y.-S., & Lee, W.-B. (2025). *Beyond Natural Language Perplexity: Detecting Dead Code Poisoning in Code Generation Datasets*.

Tseng, T., Murai, Y., Freed, N., Gelosi, D., Ta, T. D., & Kawahara, Y. (2021). PlushPal: Storytelling with Interactive Plush Toys and Machine Learning. *International Conference on Interaction Design and Children*.

Tyshchuk, K., Karpikova, P., Spiridonov, A., Prutianova, A., Razzhigaev, A., & Panchenko, A. (2023). On Isotropy of Multimodal Embeddings. *Information*, *14*(7), 392. https://doi.org/10.3390/info14070392

Vakali, A., & Dimitriadis, I. (2025). *FAIRTOPIA: Envisioning Multi-Agent Guardianship for Disrupting Unfair AI Pipelines*.

van Lier, M., & Muñoz-Gil, G. (2024). *Artificial Agency and Large Language Models*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *2017-December*, 5999–6009.

Villegas-Ch., W., Garcia-Ortiz, J., & Sánchez-Viteri, S. (2021). Identification of the Factors That Influence University Learning with Low-Code/No-Code Artificial Intelligence Techniques. *Electronics*.

Vitkare, G., Jejurkar, R., Kamble, S., Thakare, Y., & A.P.Lahare, Prof. (2023). AUTOMATED HTML CODE GENERATION FROM HAND DRAWN IMAGES USING MACHINE LEARNING METHODS. *International Journal of Progressive Research in Engineering Management and Science*.

Vlasic, A., & Aguinaga, S. (2024). *QuOp: A Quantum Operator Representation for Nodes*.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*.

Xiao, S., Chen, Y., Li, J., Chen, L., Sun, L., & Zhou, T. (2024). Prototype2Code: End-to-end Front-end Code Generation from UI Design Prototypes. *Conference on Computability in Europe*.

Xu, Y., Liu, X., Cao, X., Huang, C., Liu, E., Qian, S., Liu, X., Wu, Y., Dong, F., Qiu, C.-W., Qiu, J., Hua, K., Su, W., Wu, J., Xu, H., Han, Y., Fu, C., Yin, Z., Liu, M., … Zhang, J. (2021). Artificial intelligence: A powerful paradigm for scientific research. *The Innovation*, *2*(4), 100179. https://doi.org/10.1016/j.xinn.2021.100179

Yang, L., Xu, M., & Ke, W. (2024). Enhancing Question Answering Precision with Optimized Vector Retrieval and Instructions. *ArXiv.Org*.

Younesi, A., Ansari, M., Fazli, M., Ejlali, A., Shafique, M., & Henkel, J. (2024). A Comprehensive Survey of Convolutions in Deep Learning: Applications, Challenges, and Future Trends. *IEEE Access*, *12*, 41180–41218. https://doi.org/10.1109/ACCESS.2024.3376441

Zhang, F., Chen, B., Zhang, Y., Liu, J., Zan, D., Mao, Y., Lou, J.-G., & Chen, W. (2023). RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. *Conference on Empirical Methods in Natural Language Processing*.

Zhang, Q., Hendra, L. B., Chi, M., & Ding, Z. (2024). Frontend Diffusion: Exploring Intent-Based User Interfaces through Abstract-to-Detailed Task Transitions. *ArXiv.Org*.

Zhang, W., Guo, Y., Niu, L., Li, P., Zhang, C., Wan, Z., Yan, J., Farrukh, F. U. D., & Zhang, D. (2023). *LP-SLAM: Language-Perceptive RGB-D SLAM system based on Large Language Model*.

Zhao, H., Yang, F., Lakkaraju, H., & Du, M. (2024). *Opening the Black Box of Large Language Models: Two Views on Holistic Interpretability*.

Zheng, P., Zhao, Y., Gong, Z., Zhu, H., & Wu, S. (2024). *SimpleLLM4AD: An End-to-End Vision-Language Model with Graph Visual Question Answering for Autonomous Driving*.

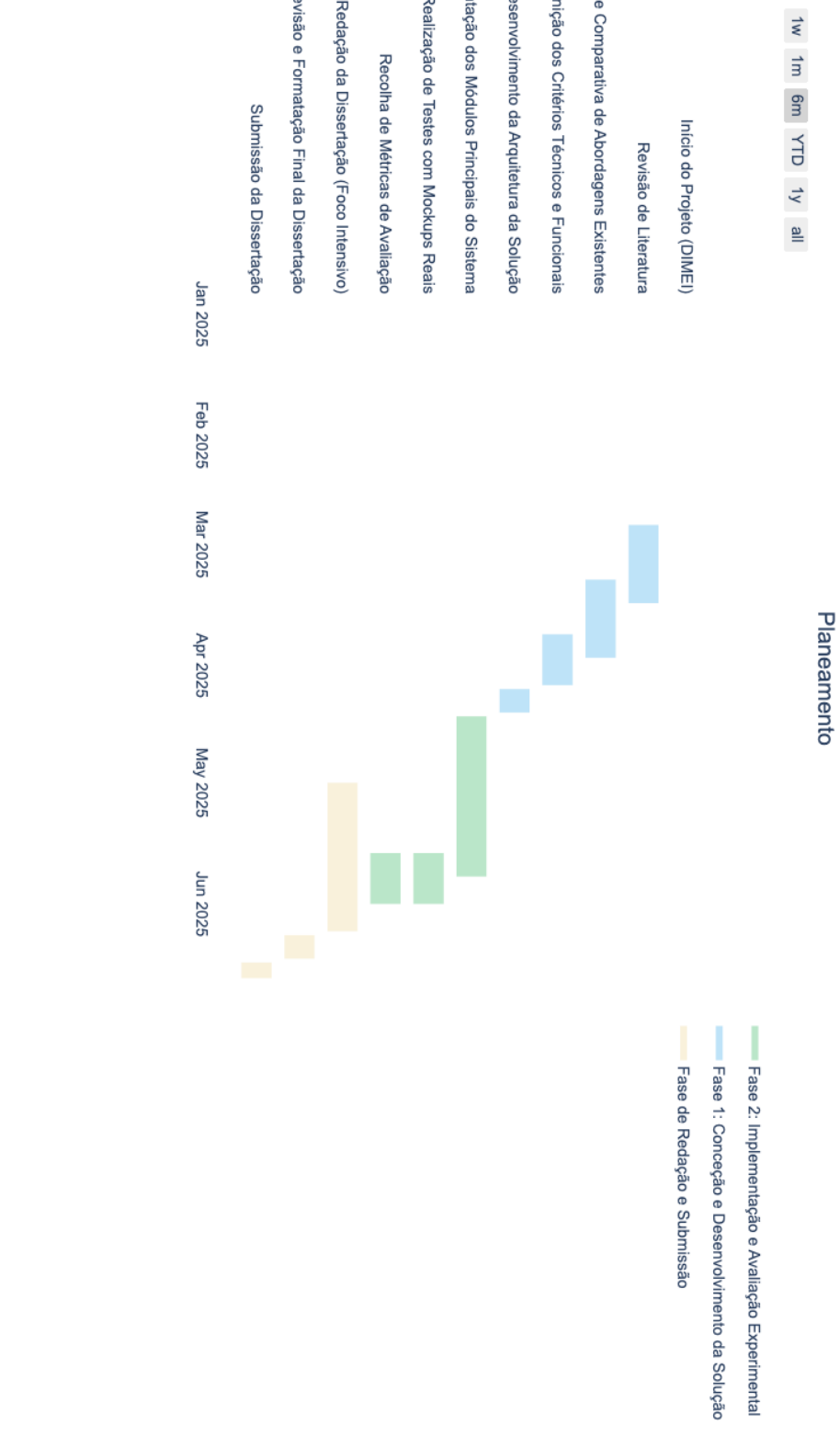
Zhou, M., Gao, X., Wu, J., Liu, K., Sun, H., & Li, L. (2024). Investigating White-Box Attacks for On-Device Models. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–12. https://doi.org/10.1145/3597503.3639144

Zhou, T., Zhao, Y., Hou, X., Sun, X., Chen, K., & Wang, H. (2024). Bridging Design and Development with Automated Declarative UI Code Generation. *ArXiv.Org*.

Zhuo, T. Y. (2024). ICE-Score: Instructing Large Language Models to Evaluate Code. *EACL 2024 - 18th Conference of the European Chapter of the Association for Computational Linguistics, Findings of EACL 2024*, 2232–2242.

Zitouni, M. N., Anda, A., Rajpal, S., Amyot, D., & Mylopoulos, J. (2024). Towards the LLM-Based Generation of Formal Specifications from Natural-Language Contracts: Early Experiments with Symboleo. *ArXiv.Org*.

Anexo A. Diagrama de GANTT



Anexo B. Diagrama de *PRISMA*