Programa Institucional de Bolsas de Iniciação em Desenvolvimento Tecnológico e Inovação (PIBITI)

Insper

Direito

Decisões Judiciais Automatizadas: Sistema Baseado em Programação, Templates e IA Generativa

Antonio Carlos Ferreira da Silva Filho

Orientador: Julio Adolfo Zucon Trecenti

São Paulo

2025

Programa Institucional de Bolsas de Iniciação em Desenvolvimento Tecnológico e Inovação (PIBITI)

Antonio Carlos Ferreira da Silva Filho

Decisões Judiciais Automatizadas: Sistema Baseado em Programação, Templates e IA Generativa

Relatório final apresentado ao Programa Institucional de Bolsas de Iniciação em Desenvolvimento Tecnológico e Inovação (PIBITI)

Orientador: Julio Adolfo Zucon Trecenti

São Paulo

2025

Resumo

O aumento contínuo do volume processual e a necessidade de padronização de textos gerados em decisões judiciais ou outros documentos no meio jurídico motivam até hoje diversas tentativas de criar ferramentas de automação de documentos. Modelos de linguagem de larga escala (LLMs) ampliam o potencial de automação, mas a geração direta de texto traz riscos de alucinação e dificuldades de verificação que podem ser tão grandes quanto os potenciais ganhos de produtividade. Este trabalho propõe uma abordagem intermediária, o tipo human in the loop (HITL) que transforma o problema de redação automática por um problema de geração de códigos, capaz de criar de documentos em Microsoft Word, usando exclusivamente conteúdos previamente elaborados por humanos. O trabalho fornece três contribuições principais: um esquema de marcação em diferentes estruturas baseado em tags; um prompt estruturado que, a partir de um documento anotado e um conjunto de metadados, produz um código na linguagem de programação Python capaz de gerar novos documentos, preservando formatação, marcações e aplicar regras de complexas, baseadas no contexto e variáveis definidas no template fornecido; uma aplicação interativa na web para configuração de templates parametrizados, geração de códigos usando LLMs e criação dos documentos a partir do preenchimento de um formulário criado automaticamente. Em avaliação exploratória com especialistas em decisões judiciais, a estratégia mostrou viabilidade prática, reduzindo alucinações ao restringir a saída ao espaço definido pelas anotações e padronizando trechos repetitivos de decisões. As limitações concentram-se na qualidade da anotação e na cobertura de variações em problemas de borda. A abordagem é replicável para outros tipos de documentos, como contratos, petições e despachos repetitivos, podendo impactar positivamente na produtividade de diversos atores do sistema de justiça.

Palavras-chave: automação de documentos jurídicos; modelos de linguagem; human in the loop; interface web; inteligência artificial.

Abstract

The continuous growth in case volume and the need to standardize texts produced in judicial decisions and other legal documents have long motivated attempts to build document automation tools. Large Language Models (LLMs) expand the automation frontier, but direct text generation increases hallucination risks and verification burdens that can offset productivity gains. This work proposes an intermediate, human in the loop (HITL) approach that reframes the automatic drafting problem as a code generation problem, aiming to create Microsoft Word documents using exclusively human authored content. It offers three main contributions: a tag based markup scheme for multiple structural layers; a structured prompt that, given an annotated document and metadata, produces Python code capable of generating new documents while preserving formatting, and applying complex, context dependent rules based on variables of the template; and a web based interactive application for configuring parameterized templates, generating code via LLMs, and instantiating documents from an automatically generated form. In an exploratory evaluation with specialists in judicial decisions, the strategy proved practically viable: it reduced hallucinations by constraining outputs to the annotated space and standardized repetitive sections of decisions while keeping humans in control of the final content. Limitations center on annotation quality and coverage of edge cases. The approach is replicable for other document types, such as contracts and routine orders, and can positively impact the productivity of various actors across the justice system.

Keyword: legal document assembly; large language models; artificial intelligence; human in the loop; web interface.

Sumário

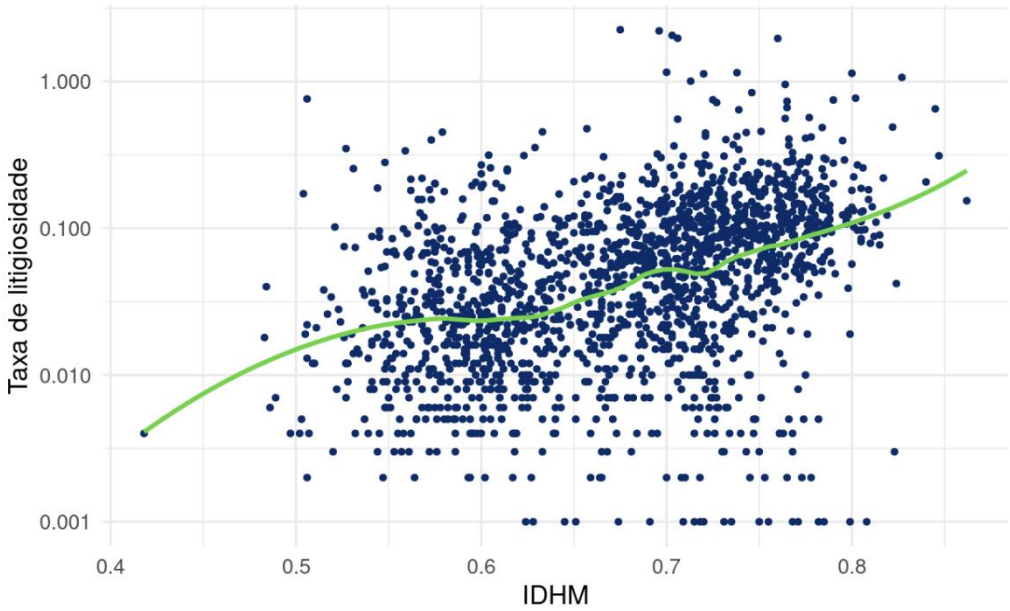
Sumário

1. INTRODUÇÃO 6
1.1 Objetivo 8
2. REVISÃO DA LITERATURA 10
3. METODOLOGIA 14
3.1 Ferramentas e Tecnologias Fundamentais 14
3.1.1 GPT (*Generative Pre-trained Transformer*) 14
3.1.2 GEMINI 14
3.1.3 Shiny for Python 15
3.2 *Template* Inicial e Prova Preliminar de Conceito 15
3.3 Aprimoramento do Prompt e Desenvolvimento da Aplicação 18
4. RESULTADOS 20
4.1 Análise Estrutural dos Documentos 20
4.2 Uso da Ferramenta: Utilização das Estruturas como *Tags* para Criação de *Template* Parametrizado 21
4.3 Uso da ferramenta: Upload de *Template* e Configurações Avançadas 22
4.4 Uso da ferramenta: Geração do Código 24
4.5 Uso da ferramenta: Geração do Documento 26
4.6 Resultados Práticos: Entrevistas com Profissionais da Área 27
5. CONSIDERAÇÕES FINAIS E LIMITAÇÕES 29
Referências 30
Apêndice A – Parametrização Inicial 32
Apêndice B- Código Desenvolvido Manualmente 33
Apêndice C – Aplicação Antiga 36
Apêndice D – Códigos do Shiny e da API 37
Apêndice E – Template Parametrizado com *tags* e *Markdown* 45
Apêndice F – Estrutura de Dados 46

1. INTRODUÇÃO

O aumento de casos novos no judiciário é um fenômeno previsível (TORQUATO et al., 2021). À medida que a sociedade evolui, as pessoas estabelecem mais vínculos, contratos, aumentando a quantidade de conflitos e, portanto, litígios. Além disso, espera-se que a evolução de uma sociedade tenha como consequência o aumento do acesso à justiça, o que também leva a mais litígios.

Figura 1 - Relação entre o Índice de Desenvolvimento Humano Municipal (IDHM) e a taxa de litigiosidade nas comarcas do Brasil.



Fonte: Associação Brasileira de Jurimetria (2017)

O volume crescente de processos no judiciário é um desafio para a administração da Justiça no Brasil. Existem duas formas principais de enfrentar o problema: i) criando incentivos à desjudicialização, evitando que casos desnecessários cheguem ao judiciário, e ii) desenvolvendo estratégias para aumentar a produtividade, encerrando os casos que entram mais rapidamente e evitando o aumento do estoque de processos.

Diversas tentativas de desjudicialização estão sendo implementadas, não só no Poder Judiciário, como no Executivo. Uma delas é o incentivo ao uso de plataformas que permitem composição extrajudicial, como a plataforma consumidor.gov.br, que evita a entrada de um volume ainda maior de processos relacionados ao direito do consumidor (CNJ, 2017). Outra são mecanismos oriundos de mudanças legislativas, como a mediação e a conciliação, que incentivam os acordos e encerram processos sem a necessidade de julgamento de mérito (Fonseca; Lima, 2021). Essas estratégias, apesar de importantíssimas para evitar a litigância frívola e o uso ineficiente do sistema judiciário, não são suficientes, porque ainda existe um estoque enorme de processos aguardando finalização.

Para monitorar e incentivar o aumento da produtividade do judiciário, o CNJ utiliza, desde 2009, um programa de metas de produtividade. As metas buscam, entre outras atividades, reduzir o acervo de processos pendentes e encerrar mais casos do que o número de casos novos em cada mês ou ano. As metas do CNJ funcionam como um mecanismo de incentivo para que os tribunais melhorem a eficiência das equipes, incorporem mais tecnologias e resolvam os casos repetitivos de forma estratégica, como, por exemplo, utilizando o Incidente de Resolução de Demandas Repetitivas (Brasil, 2015).

As metas do CNJ, no entanto, não garantem nem a melhoria dos trabalhos das equipes nas varas e câmaras, nem a qualidade das decisões. Frequentemente, os magistrados se veem em uma situação de excesso de trabalho para atingir as metas (Asensi et al., 2022), ao mesmo tempo em que não podem reduzir a qualidade das decisões apenas para aumentar a produtividade.

Uma estratégia antiga utilizada para aumentar a produtividade sem necessariamente perder a qualidade é o uso de *templates*. Um *template* de documento pode ser entendido como um modelo predefinido com estruturas e campos variáveis, que permite gerar documentos personalizados de forma mais rápida e consistente. Por exemplo, uma vara judicial pode ter modelos de decisão para situações recorrentes, já fazendo uma seleção da jurisprudência a ser utilizada, dependendo da situação. Os *templates* geralmente são construídos manualmente, a partir do uso de decisões passadas como base para criação de padrões de escrita.

Mais recentemente, em novembro de 2022, o lançamento de uma nova ferramenta mudou o mundo. Produto de avanços da capacidade computacional (Lu, 2018), de uma importante inovação na arquitetura de redes neurais (Vaswani et al., 2017), trouxe uma ferramenta que seria uma “resposta” para o problema da produtividade. Trata-se do ChatGPT, da OpenAI, seguido de suas atualizações e seus concorrentes.

O ChatGPT e concorrentes são alimentados por modelos de linguagem de larga escala (*Large Language Models*, ou LLMs). Tratam-se de modelos estatísticos treinados a partir de trilhões de textos, jurídicos ou não, capazes de imitar a escrita humana, produzindo textos completos a partir de uma solicitação do usuário. Com essas ferramentas, é possível gerar textos de diversas páginas apenas com um pedido simples, com resultados aceitáveis para diversas aplicações.

A popularidade do ChatGPT e de seus concorrentes, como o Gemini, da Google, o Claude da Anthropic e a chinesa DeepSeek, é um fenômeno sem precedentes. Tais ferramentas representam impactos profundos em diversas áreas, como codificação, produção artística e ensino (Eloundou et al., 2024).

E claro, o Direito não ficou de fora desse avanço tecnológico. A maior parte da comunicação entre atores do Direito se dá a partir de documentos, ou seja, de texto. Então não é surpreendente que as aplicações dos modelos de linguagem sejam imediatas. Basta pedir ao ChatGPT ou concorrente que produza uma petição inicial, contestação ou sentença judicial a partir de uma lista de regras, e a ferramenta criará um texto que atende a esse pedido.

No entanto, o aumento de produtividade exponencial não é, necessariamente, acompanhado da qualidade. LLMs frequentemente geram textos que não atendem exatamente ao que foi solicitado, ou mesmo textos que simplesmente não são reais. São as conhecidas “alucinações”. São inúmeros os casos encontrados no judiciário de geração de jurisprudência falsa, argumentos jurídicos inválidos e fatos distorcidos. O uso dessas ferramentas para elaborar decisões judiciais, portanto, é inadequado ou até perigoso para o sistema.

Nesse sentido, cresce a importância do conceito “*human in the loop*” (HITL). Trata-se de um princípio de desenho de soluções de tecnologia que requerem a interação com humanos ao elaborar uma solução (Mosqueira-Rey et al., 2023). Com isso, os textos eventualmente gerados por LLMs devem ser revisados e verificados pelos humanos. Espera-se, com isso, fazer um balanço entre aumento de produtividade e manutenção da qualidade.

Infelizmente, o trabalho de verificar o conteúdo gerado por um LLM pode ser tão ou mais trabalhoso que criar um novo documento. Dessa forma, o uso de LLMs acaba sendo ineficiente, gastando mais recursos do que em um cenário em que não está sendo utilizado.

O presente trabalho buscou seguir o princípio do HITL para automatização de documentos jurídicos em Microsoft Word (“Word”), fornecendo textos cujos insumos são totalmente gerados por humanos, sem a geração de textos em linguagem natural por LLMs. A abordagem transforma o problema de gerar textos em um problema de gerar um código, que por sua vez é capaz de elaborar documentos a partir de regras extraídas de variáveis e lógicas complexas inferidas a partir de um documento de origem. O código gerador de documentos é gerado a partir do uso de LLMs, que tem como insumo uma decisão real anotada, que pode ser produzida por humanos em poucos minutos. A solução, portanto, não escreve nenhum texto novo, reduzindo o risco de alucinações e colocando o humano no controle dos documentos gerados. Uma ferramenta online foi desenvolvida para permitir a interação de pessoas sem domínio de programação com a solução proposta.

O resultado esperado do trabalho é que a ferramenta criada possa ser utilizada por magistrados e outros colaboradores do sistema de justiça para aumentar sua produtividade, sem com isso reduzir a qualidade dos textos gerados. Pelo contrário: por conta da padronização na geração de documentos e nas tarefas repetitivas, é possível que a qualidade dos documentos seja maior do que o observado atualmente.

1.1 Objetivo

Este trabalho tem como objetivo desenvolver uma metodologia para criação de geradores de documentos Word, a partir de um documento anotado por humanos e do uso de LLMs. Especificamente, os objetivos do projeto são:

1. Elaborar uma metodologia de anotação de documentos, definindo entradas (*inputs*), variáveis (categorias) e lógicas (regras de inclusão e exclusão de

elementos textuais), a ser utilizada por um humano para anotar textos jurídicos;

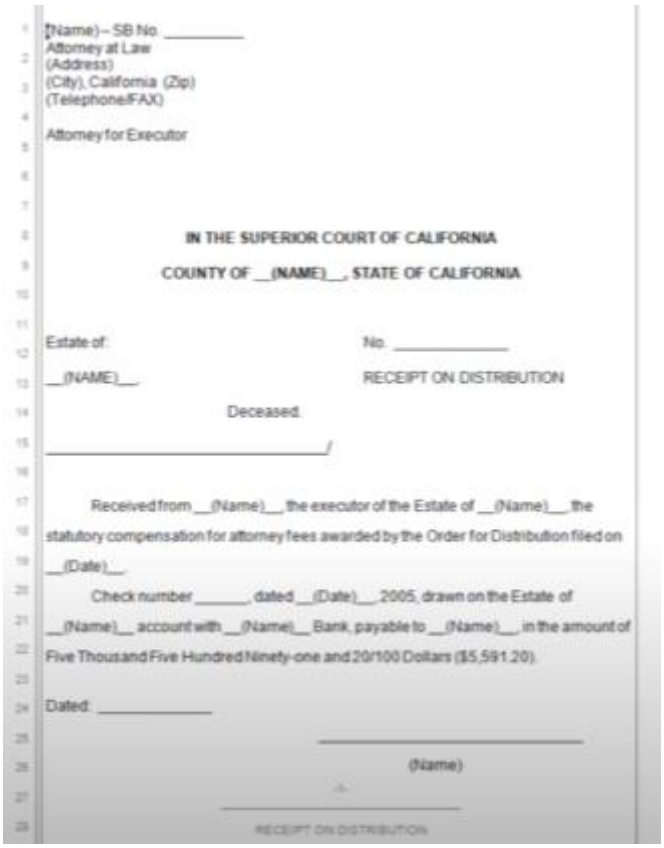
2. Construir, aplicar e aprimorar um *prompt* de LLM para gerar um código gerador de documentos jurídicos a partir do documento anotado;
3. Desenvolver uma ferramenta interativa que permite ao usuário i) inserir documentos anotados, ii) criar novos códigos geradores de documentos e iii) aplicar os códigos geradores a casos novos.

2. REVISÃO DA LITERATURA

Um *template* de documento Word pode ser entendido como um **modelo predefinido com campos variáveis, que permite gerar documentos personalizados a partir do preenchimento desses campos**. Tradicionalmente, esses *templates* são construídos manualmente por especialistas, muitas vezes a partir de documentos existentes nos quais os dados específicos de cada caso (nomes, datas, valores, jurisprudência, etc.) são destacados e substituídos por marcadores ou campos editáveis. É comum encontrar tais documentos em contratos e documentos de “mala direta”, a partir de recursos disponíveis na ferramenta.

Antes da ascensão da inteligência artificial generativa, a automatização de documentos jurídicos apoiava-se fortemente em *templates* fixos e métodos baseados em regras. Um exemplo ilustrativo vem dos anos 1990, quando advogados empregavam macros em editores de texto (como WordPerfect) para montar petições automaticamente a partir de dados fornecidos pelo usuário.

Figura 2 - Documento gerado no WordPerfect.


(Name) – SB No. _____
Attorney at Law
(Address)
(City), California (Zip)
(Telephone/FAX)

Attorney for Executor

IN THE SUPERIOR COURT OF CALIFORNIA

COUNTY OF __(NAME)__, STATE OF CALIFORNIA

Estate of: No. _____

__(NAME)__ RECEIPT ON DISTRIBUTION

Deceased.

_____/

Received from __(Name)__ the executor of the Estate of __(Name)__ the statutory compensation for attorney fees awarded by the Order for Distribution filed on __(Date)__.

Check number _____ dated __(Date)__, 2005, drawn on the Estate of __(Name)__ account with __(Name)__ Bank, payable to __(Name)__ in the amount of Five Thousand Five Hundred Ninety-one and 20/100 Dollars ($5,591.20).

Dated: _____

_____

(Name)

_____

RECEIPT ON DISTRIBUTION

Fonte: Fonte: Vídeo. (Techquickie, 2019)

Nessa abordagem, cada documento seguia um roteiro pré-definido: trechos invariáveis de texto (comum a todos os documentos do tipo, como fórmulas de saudação, cláusulas padrão, estrutura de seções) eram combinados com campos

vazios para informações variáveis (como nomes das partes ou datas). O preenchimento desses campos podia ser feito manualmente (pelo advogado revisando o documento) ou semi-automaticamente, por meio de planilhas que populavam o *template* com base nas respostas fornecidas. Ferramentas de automação consagradas, como o HotDocs, consolidaram-se nesse período e na década seguinte, permitindo que escritórios definissem questionários e regras lógicas para gerar documentos Word a partir de *templates* parametrizados (Lauritsen; Soudakoff, 1998).

Os *templates* tradicionais frequentemente utilizavam formatos estruturados como XML ou linguagens de marcação específicas. O formato DOCX, atualmente utilizado pelo Word, é internamente um conjunto de arquivos XML, o que possibilita a criação de *placeholders* (campos de preenchimento) dentro do documento e sua posterior substituição por dados via código ou ferramentas de mesclagem. Bibliotecas como *python-docx* (Mckinney; Wilson, 2024) permitem manipular programaticamente documentos Word, inserindo textos em marcadores delimitados ou realizando substituição de textos por meio de scripts.

Outra técnica comum envolve o uso de expressões regulares (*regex*) para localizar padrões textuais nos documentos (Jurafsky; Martin, 2023). Por exemplo, identificar sequências como "Nome: xxx" ou formatos de data, e então extrair ou substituir esses trechos conforme necessário. Embora o uso de *regex* para processar documentos extensos seja considerado frágil por conta de variações linguísticas e exceções, essa abordagem pode ser empregada em algumas aplicações específicas, como a extração de números de processo ou CPF em petições padronizadas.

Paralelamente, desenvolveu-se um ecossistema de ferramentas de automação de documentos jurídicos que encapsulam lógicas de *template* e regras, muitas vezes com interfaces amigáveis para pessoas que não sabem programar. Por exemplo, o Docassemble, criado por Jonathan Pyle, um programador e advogado, é um sistema de código aberto lançado em 2017 que permite inserir elementos textuais em *templates* pré definidos.

No ecossistema como um todo, o denominador comum é que um humano precisa, inicialmente, modelar o documento. É necessário definir quais partes são variáveis, quais são fixas e quais regras determinam a inclusão ou exclusão de certas cláusulas. Isso exige um trabalho manual significativo de análise de documentos e criação do *template*, embora traga ganhos de produtividade substanciais a longo prazo (Calloway, 2025).

Com o avanço do aprendizado de máquina na década de 2010, surgiram métodos para extrair automaticamente estruturas e campos de documentos jurídicos a partir de exemplos anotados, aliviando em parte a dependência de regras manuais rígidas (Lample et al., 2016). Nessas abordagens, modelos estatísticos de aprendizado supervisionado como *Named Entity Recognition* (NER) são treinados sobre documentos onde as informações de interesse foram previamente anotadas por humanos. Por exemplo, um conjunto de contratos poderia ser manualmente anotado indicando as frases que correspondem à cláusula de arbitragem, ao foro eleito, aos prazos, etc. Com base nessas anotações, os algoritmos podem aprender padrões linguísticos e contextuais que delimitam cada campo.

Ferramentas comerciais de análise contratual surgidas na segunda metade da década de 2010 incorporaram aprendizado de máquina para identificar cláusulas e obrigações em textos jurídicos de forma semi-automatizada. A Luminance, por exemplo, auxilia escritórios na revisão de contratos, utilizando desde técnicas tradicionais de aprendizado de máquina até ferramentas de IA generativa. A adoção dessas ferramentas reflete-se nos ganhos de eficiência de escritórios: tarefas de triagem e identificação de riscos contratuais tornaram-se significativamente mais rápidas e acuradas com o apoio de IA. As soluções de aprendizado de máquina, no entanto, focavam mais na extração de informações em documentos existentes do que na geração autônoma de novos documentos.

A partir de 2020, com o advento dos LLMs, o campo de automação de documentos jurídicos entrou em uma nova fase. Esses LLMs possuem a notável capacidade de aprender padrões textuais complexos sem treinamento específico para cada tarefa, através de aprendizado de poucos exemplos (*few-shot*) ou mesmo zero-shot, simplesmente ao serem instruídos por prompts (Wang et al., 2020). Assim, atividades como redigir um contrato, resumir uma petição ou extrair metadados relevantes tornaram-se viáveis com mínima ou nenhuma customização de modelo, algo impensável nos paradigmas anteriores, que requeriam conjuntos de dados anotados e modelos treinados do zero para cada caso de uso.

Uma das aplicações mais diretas dos LLMs no contexto jurídico é a geração de documentos a partir de descrições em linguagem natural. Por exemplo, já existem ferramentas que, com base em alguns parâmetros fornecidos pelo usuário (como nomes das partes, natureza do contrato e pontos negociais básicos), obtêm do LLM um primeiro rascunho de contrato ou petição. No Brasil, um exemplo notável é a Assert.IA, ferramenta que está sendo desenvolvida pelo Tribunal de Contas da União (TCU) para elaboração de instruções técnicas e outros documentos a partir de representações e denúncias (Pereira et al., 2025).

Entretanto, a aplicação de LLMs em contexto jurídico traz desafios e requer uma postura crítica quanto aos resultados gerados. Um risco bem documentado é o das “alucinações”: o modelo pode, com grande fluência, inventar fatos ou jurisprudência que não existem no documento, se instado a completar lacunas (Dahl et al., 2024). Em tarefas de geração de documentos, isso implica a necessidade de revisão humana cuidadosa: profissionais devem conferir se o documento produzido pela IA está juridicamente adequado, sem omissões problemáticas ou adições indevidas. Essa revisão pode ser tão ou mais demorada que a geração de um documento usando as metodologias tradicionais, o que pode tornar o uso de LLMs inviável para essa aplicação.

A hipótese da presente pesquisa é que, quando bem empregados, os LLMs podem sim resultar em um salto de eficiência. Ao invés de revisar textos gerados por IA, é possível considerar o problema de geração de documentos por IA em um problema de geração de códigos por IA. O código, por sua vez, considera apenas elementos textuais que já existem na base de conhecimento fornecida pelos usuários, mitigando o risco de alucinação. Como consequência, em vez de funcionar como um substituto dos sistemas baseados em *templates*, os LLMs podem atuar como elementos de apoio e aceleração na criação de *templates*, com soluções que podem

ser incorporadas em ferramentas que já existem no mercado.

3. METODOLOGIA

O desenvolvimento da presente pesquisa foi subdivido em duas diferentes etapas. A primeira delas consistiu no desenvolvimento do método de parametrização, culminando numa prova de conceito, cujo objetivo era obter uma validação preliminar do método proposto antes de se buscar maiores avanços. Enquanto a segunda, além do aprimoramento das técnicas já desenvolvidas, teve como enfoque o desenvolvimento e otimização da ferramenta interativa. A presente seção, portanto, irá abordar a metodologia e as tecnologias empregadas durante as referidas etapas.

3.1 Ferramentas e Tecnologias Fundamentais

A execução da pesquisa dependeu de um conjunto de tecnologias, notadamente os LLMs explicitados abaixo e o Shiny.

3.1.1 GPT (*Generative Pre-trained Transformer*)

Desenvolvido pela OpenAI, o GPT é uma família de modelos de linguagem que popularizou a abordagem de pré-treinamento em larga escala. O termo "*Generative Pre-trained Transformer*" descreve seu método de funcionamento: ele é "pré-treinado" em um vasto conjunto de dados para aprender a prever o próximo *token* (parte de uma palavra) em uma sequência (arquitetura auto-regressiva) e, em seguida, pode ser ajustado (*fine-tuned*) para tarefas específicas. A principal inovação demonstrada pela série GPT foi a emergência de capacidades de "aprendizado com poucos exemplos" (*few-shot learning*), onde o modelo pode executar tarefas para as quais não foi explicitamente treinado, apenas com base em alguns exemplos fornecidos no prompt (BROWN et al., 2020). Sua capacidade de seguir instruções complexas, manter o contexto em longas conversas e gerar conteúdo criativo o torna uma ferramenta poderosa para tarefas de geração de código, tradução e sumarização.

3.1.2 GEMINI

O Gemini representa a nova geração de modelos de IA desenvolvida pela Google, concebida desde o início como uma arquitetura nativamente multimodal. Diferentemente de modelos anteriores que combinavam componentes distintos para diferentes modalidades, o Gemini foi pré-treinado de forma integrada em texto, imagens, áudio, vídeo e código. Essa abordagem permite um raciocínio mais sofisticado e uma compreensão mais profunda de informações contextuais complexas (Gemini Team Google, 2025). A família Gemini é oferecida em diferentes versões, Pro e Flash, otimizados para diferentes aplicações, desde tarefas de alta complexidade até aplicações de baixa latência. Uma de suas características notáveis é a janela de

contexto expandida, tanto para entradas quanto para saídas, que permite processar e gerar volumes de informação em uma única requisição, uma propriedade muito conveniente para aplicações que demandam a manipulação de documentos ou códigos extensos.

3.1.3 Shiny for Python

Criada pela Posit (anteriormente RStudio), o Shiny for Python é uma ferramenta de desenvolvimento rápido de interfaces *web*, adaptada do popular framework Shiny, criado inicialmente para a linguagem R (R Core Team, 2024). O sistema permite a criação de aplicações web interativas diretamente a partir de código Python. Uma tecnologia central do Shiny é seu paradigma de programação reativa. Este modelo estabelece um vínculo dinâmico entre as entradas do usuário (*widgets* da interface, como *sliders*, caixas de texto e menus de seleção) e as saídas (como gráficos, tabelas e textos) que são processadas no servidor, similar ao que existiria em uma aplicação em na linguagem JavaScript, mas diretamente pelo Python. Quando um usuário interage com um widget, o Shiny detecta a mudança e executa automaticamente apenas as partes do código que dependem daquela entrada, atualizando as saídas de forma eficiente, sem a necessidade de programar explicitamente toda a cadeia de eventos.

A arquitetura de uma aplicação Shiny é tipicamente dividida em duas componentes principais: a interface do usuário, que define a aparência e o layout da aplicação de forma declarativa, e a lógica do servidor, que contém o código Python imperativo que executa as análises e gera os resultados. Isso permite liberdade e velocidade de desenvolvimento em dois sentidos: por um lado, a interface do usuário é apenas uma simplificação da linguagem *web* (com HTML, CSS e JavaScript), podendo ser substituída, no todo ou em partes, por elementos web puros. Por outro lado, o server possui todas as facilidades de programação do Python, permitindo o desenvolvimento rápido de aplicações, sem as preocupações associadas ao desenvolvimento de sistemas mais robustos.

3.2 *Template* Inicial e Prova Preliminar de Conceito

Inicialmente foi desenvolvido um *template* de word para um tipo de decisão judicial específica. Tal *template* deveria conter marcações claras e objetivas, para que possam ser utilizadas como base no desenvolvimento de um código de geração de arquivo que recebe uma lista de parâmetros e cria um documento word preenchido. A decisão escolhida para tal foi o Acordão de Segundo Grau no contexto de apelação, mais especificamente Acordão que versa sobre **ação regressiva de energia elétrica**.

O tipo de ação mencionado versa sobre a sub-rogação de uma seguradora nos direitos de um segurado que vivenciou algum tipo de sinistro devido à má prestação de serviços elétricos. Assim, a seguradora, após o pagamento da indenização, aciona

a companhia elétrica no judiciário buscando ressarcimento pelos valores pagos, a partir do direito de regresso.

A jurisprudência quanto a este caso é relativamente pacífica em segundo grau, existindo duas principais vertentes. Uma delas, por se tratar de responsabilidade objetiva, dá o ganho de causa para a seguradora sempre que presente um laudo técnico comprobatório do sinistro, uma vez que teria sido provado a ação ou omissão, o dano, bem como o nexo de causalidade. Não obstante, a outra vertente tende a ser mais exigente quanto ao método comprobatório, dando o ganho de causa somente para os casos em que ocorreu perícia Judicial.

Desta forma a ação configura causa repetitiva, gerando, assim, as condições ideias para que seja elaborado um *template* abrangente, a partir da compilação das possíveis variações de cada caso, que possa ser utilizado tanto para testes quanto para casos futuros. Uma vez que as decisões tendem a ser sempre muito similares, apresentando maiores variações quanto aos argumentos utilizados pelas partes, mesmo que o resultado permaneça inalterado.

A escolha justifica-se também pela elevada incidência de recursos referentes a esse tipo de ação. Uma rápida pesquisa jurisprudencial no E-SAJ de São Paulo, utilizando-se do termo “REGRESSIVA ENERGIA ELÉTRICA” no campo “Ementa”, revela a existência de mais de 43 mil julgados sobre o assunto. Ademais, esta ação configura causa repetitiva, gerando, assim, as condições ideias para que seja elaborado um *template* abrangente, a partir da compilação das possíveis variações de cada caso, que possa ser utilizado tanto para testes quanto para casos futuros.

Para a criação do *template* foi escolhida uma câmara do TJSP e um desembargador que tenha atuado frequentemente como relator neste tipo de ação. Mais especificamente, foi selecionada a 27ª Câmara de Direito Privado e um de seus desembargadores, dado que, após refinar os filtros de busca para este novo critério foi encontrada uma base suficiente de variações entre os Acórdãos analisados, desde preliminares rejeitadas à diferentes fundamentações utilizadas (provavelmente devido às diferentes alegações feitas pelas partes), possibilitando a elaboração do *template* nas especificações desejadas.

Quanto ao desenvolvimento do *template* em si, foi compilado em um único documento de word a estrutura, como o cabeçalho ou o relatório, e as diversas partes distintas dos Acórdãos analisados, por mais que elas fossem contraditórias (como a presença de Nega-se ou Dá-se provimento no mesmo documento). Após a criação desse “Acórdão Compilado” ou desta “Compilação de Acórdãos” realizou-se uma análise sobre sua composição, dividindo seus trechos em diferentes categorias.

Tendo sido feito o *template* seria necessário, antes da realização de qualquer tipo de teste, a criação de um código manual elaborado a partir das marcações desenvolvidas para ser utilizado como critério de validação. Assim, neste primeiro momento, foi utilizado como critério de parametrização o uso de diferentes cores para representar cada uma das categorias desenvolvidas. Muito embora esse sistema não possa ser utilizado diretamente por uma IA generativa para a obtenção do fim desejado, ele acaba sendo mais eficiente para a presente etapa de desenvolvimento,

uma vez que permite uma maior e mais rápida identificação pelo olho humano dos padrões necessários para a elaboração do pretendido código.

Cabe ressaltar, também, que neste momento foram realizadas algumas simplificações, como a remoção da parte descritiva do relatório, em face à finalidade desta etapa do projeto ser a criação de um *template* e código para serem utilizados como mecanismo de verificabilidade em testes futuros. Dado que não era proveitoso adentrar em situações mais complicadas ainda nesta ocasião. Uma *print* deste primeiro *template* encontra-se no Apêndice A.

Findada a criação do *template* iniciou-se o desenvolvimento do Código. Conforme já explicitado, o objetivo atual seria a elaboração de um script, seguindo uma lógica de geração de arquivo, que recebe uma lista de parâmetros e cria um documento word preenchido. Para tal foi utilizada a biblioteca Python “docx”, por permitir a padronização necessária para criação de documentos a partir de programação, assim é possível estruturar todo um Acórdão com os padrões adequados, como fonte, espaçamento etc., além de possibilitar o preenchimento das categorias supramencionadas com os parâmetros devidos para cada caso.

Quanto à estruturação do código, optou-se pela utilização da função “Input()” para cada parâmetro individual. Muito embora esta solução possa ser ineficiente em comparação à criação de uma grande função que já contenha todos os parâmetros necessários para a geração do documento, ela traz uma maior acessibilidade ao código, permitindo que um profissional de direito, leigo em programação, possa apenas rodar o script em seu terminal e, assim, gerar os documentos de seu interesse, sem ter que, necessariamente, interagir com o código em um Python Interpreter. Destaca-se que esse tipo de estrutura, mesmo que funcional neste momento, foi substituída após o desenvolvimento da interface interativa.

Para o desenvolvimento do código, foram adicionados ao script todos os parágrafos contidos no *template*, a partir do método “docx” “.add_paragraph()”, foram, também, definidas as propriedades de alinhamento e indentação adequadas para cada tipo de parágrafo. O conteúdo de cada um deles foi preenchido pelo método “.add_run”, tendo sido individualizadas as runs que continham características únicas, como itálico para os termos em latim, e agrupadas àquelas que tivessem propriedades em comum, como tipo e tamanho de fonte. Cabe ressaltar que o código foi divido em duas partes, a primeira delas contendo todos os *inputs*, bem como algumas condicionais, e a segunda contendo o documento “docx”. A primeira iteração deste script foi denominada “teste_avançado” e consta no Apêndice B.

Conforme já mencionado, o documento foi marcado, inicialmente, por cores, para fácil visualização de um humano. Como apenas o texto foi passado para os modelos de IA, eles precisaram ser adaptados para um sistema de marcação. O primeiro sistema considerou o uso de diferentes símbolos (como “@” e “#”), utilizados para representar cada estrutura desejada no texto, com uma explicação desses símbolos no prompt.

Com a conclusão da marcação do template, tornou-se viável a utilização de IA generativa para testar a efetividade da metodologia proposta. Para tal foram realizados testes utilizando a API (Application Programming Interface) de 3 LLMs

diferentes, o GPT-4o da OpenAI, o Gemini-2.0-Pro, da Google, e o Claude-Sonnet-3.5 da Anthropic. Os modelos utilizados eram aqueles disponíveis até 02/02/2025.

Quanto a estes, cabe ressaltar alguns resultados. Primeiramente, o Claude foi descartado por não conseguir gerar um número suficiente de output tokens para gerar o código em sua íntegra. Em segundo lugar, os modelos da OpenAI apresentaram um entendimento maior de alguns conceitos, performando melhor na tarefa de nomear os *inputs*, não obstante os modelos da Google produziram códigos mais eficazes. Ademais, as marcações utilizadas mostraram-se problemáticas, levando a sua substituição por tags na próxima etapa. Por fim, os LLMs foram capazes de entender e interpretar corretamente a metodologia proposta. Dessa forma, resta findada esta etapa de desenvolvimento.

Cabe, entretanto, notar que maior enfoque não foi dado a cada um desses testes e ao conteúdo gerado, dado o seu propósito preliminar e estritamente exploratório, conforme já mencionado. Afinal, foram utilizados prompts simples e não otimizados, visando unicamente validar a abordagem proposta e realizar uma filtragem inicial dos LLMs a serem empregados no desenvolvimento subsequente.

3.3 Aprimoramento do Prompt e Desenvolvimento da Aplicação

Com a prova preliminar de conceito bem-sucedida, mostrou-se necessário aprimorar o prompt para garantir a obtenção de resultados eficazes e consistentes. A metodologia utilizada para esta etapa, dada a natureza do projeto, foi principalmente "refinamento experimental". Portanto, em vez de detalhar cada um dos múltiplos testes realizados, esta seção focará nos avanços e resultados obtidos em cada estágio principal de desenvolvimento tanto do prompt quanto da interface interativa, uma vez que os avanços, por via de regra, aconteceram conjuntamente.

O refinamento inicial do prompt implementou a Engenharia de Contexto. A referida técnica aumenta dinamicamente o prompt com informações relevantes e externas no momento da inferência. O processo tipicamente envolve duas etapas: primeiro, a identificação de exemplos relevantes para informar o modelo, com informações pertinentes para que não cometa erros (como partes da estrutura de um código ou saída desejada), segundo a realização de testes iterativos colocando e removendo esses exemplos, buscando um balanço entre flexibilidade e precisão. Com isso, o prompt foi separado em duas partes, primeiramente era solicitado que o LLM extraísse do *template* as informações necessárias e as organizasse em uma estrutura de dados formato *json*. A partir desta estrutura era então solicitado que a IA gerasse o código desejado. Ao fazer isso, o LLM é "ancorado" (*grounded*) nos fatos específicos do problema, o que resulta em respostas factualmente mais precisas e reduz drasticamente o risco de alucinações.

Para garantir a padronização da estrutura de dados gerados pelas respostas das APIs foi necessário desenvolver *structured outputs*, a partir do Pydantic. O Pydantic é uma biblioteca Python para validação e *parsing* de dados, cuja principal função, no contexto de uma API, é garantir a geração de um *structured output* (saída

estruturada). Diferentemente de uma saída de texto livre e não formatada, uma saída estruturada é um objeto de dados com um esquema (*schema*) predefinido, onde cada campo possui um nome, um tipo (como *string*, inteiro ou booleano) e um formato esperados. Isso é alcançado pela definição de modelos de dados utilizando as anotações de tipo do Python. Quando a API gera uma resposta, o Pydantic a valida e a força a se conformar a esse modelo. O resultado é uma resposta de API que não é apenas previsível e confiável, mas também auto-documentada, estabelecendo um contrato de dados rigoroso entre o servidor e qualquer aplicação cliente que a consuma.

Restando refinado o prompt, foi possível começar a implementação da ferramenta interativa. Para tal foi utilizado o Shiny, por permitir um desenvolvimento rápido e, essencialmente, simples. Sendo, assim, ideal para produzir uma prova de conceito eficaz e funcional. Neste primeiro momento, o Shiny atuava somente como uma integração da API para uma interface *web*, ao mesmo tempo em que permitia a customização interativa do código gerado. No Apêndice C estão presentes *prints* do funcionamento da ferramenta durante este estágio de desenvolvimento.

Contudo, a abordagem inicial, embora funcional, apresentava limitações significativas que comprometiam a escalabilidade da solução. Identificaram-se dois desafios principais: primeiramente, a inexistência de um método eficiente para transmitir configurações de formatação complexas ao LLM; em segundo lugar, o consumo excessivo de tokens de saída, dado que o modelo precisava reproduzir todo o conteúdo do *template* para gerar a resposta. Assim, o foco desta segunda etapa foi a superação desses empecilhos, visando a otimização da ferramenta.

Primeiramente, cabe dizer que os modelos da OpenAI foram descartados na fase final do projeto devido ao seu limite de tokens de saída ser significativamente inferior ao do Gemini. Ainda nesta fase, foram implementadas estruturas regulares no código do Shiny, para já extrair as informações relevantes dos *templates*, não mais delegando esta função para o LLM, o que ocasiona numa diminuição do número de tokens gastos. Consequente, esta mudança, trouxe também alterações ao *structured output*. Ademais, foi adicionada a interface uma seção destinada à definição de configurações para a formatação. Por fim, foram introduzidas mudanças ao prompt, visando adaptá-lo a esta nova estrutura. As versões finais destes códigos estão disponíveis no apêndice D.

4. RESULTADOS

Ao longo do presente trabalho foi possível desenvolver um teste de conceito funcional e eficaz, mesmo apresentando ainda algumas limitações que serão abordadas posteriormente. Assim, esta seção explicitará o funcionamento da ferramenta e as maneiras quais o usuário pode interagir com ela, bem como os demais resultados que pré-datam o aplicativo, mas que ainda servem como seu alicerce.

4.1 Análise Estrutural dos Documentos

Neste primeiro momento cabe tratar do método de parametrização proposto. Conforme já exposto, visou-se destrinchar a estrutura de qualquer tipo de documento, mais especificamente documentos jurídicos, para melhor identificar seus componentes e entender como cada uma dessas estruturas reage a mudanças e variações de caso a caso. Resultando, assim, na definição de 4 diferentes estruturas que atuam de maneira análoga a peças durante o processo de “montagem” de um documento.

A mais primordial delas engloba todos os trechos fixos de um documento que se mantém estáveis independente do contexto em que vão ser inseridos, tal tipo de estrutura foi classificada como “elemento textual”. Em contraposição aos “elementos textuais” existem também os componentes que são suscetíveis a alterações a depender do contexto de cada caso, as estruturas que aqui me refiro foram subdivididas em diferentes categorias dada as suas especificidades. Uma delas compreende os segmentos passíveis de serem preenchidos por qualquer valor, que tende, ainda, a ser individualizado e único para cada caso, como por exemplo o número de uma apelação, as folhas de uma sentença ou até mesmo o relatório das contrarrazões. Dadas essas características, o presente grupo recebeu a denominação de “*inputs*”.

No mais, existem trechos que muito embora não sejam fixos suas variações ocorrem de maneira já predeterminada, em outras palavras os valores que tais trechos podem adotar são fixos, como no caso do resultado de uma sentença (procedente, improcedente ou parcial procedente) ou da parte apelante (autor(a) ou réu). É possível ainda argumentar que as referidas estruturas não passam de “*inputs*” que já tiveram seus valores pré-definidos por uma questão de conveniência, dada a sua tendência de assumir sempre um mesmo grupo de valores, o que acaba por evidenciar ainda mais a utilidade da criação da subcategoria, doravante, classificada como “Variável”.

Por fim, foram identificados componentes cuja variação não era definida pelo seu conteúdo, ou seja, não chegam a adotar valores diferentes, ao invés disso sua única alteração é caracterizada pela sua simples inserção, ou falta disso, no documento final. Um exemplo proeminente de segmento que compõe este subgrupo seria um parágrafo dedicado ao afastamento de uma preliminar. Dessa forma, estes segmentos receberam a identificação de “Lógicas”, por responderem à uma lógica de simples inclusão.

Uma maneira alternativa de olhar para estas estruturas, e com maiores implicações práticas, seria pensar nelas como “variáveis” que aceitam um valor nulo, de forma que o primeiro valor já pré-definido é o segmento em sua integra enquanto o segundo valor é simplesmente vazio, o que acaba por ocasionar em sua não inclusão. As vantagens desta abordagem serão melhor discutidas nos tópicos adiante.

4.2 Uso da Ferramenta: Utilização das Estruturas como *Tags* para Criação de *Template* Parametrizado

A mera separação de diferentes estruturas em subgrupos de nada valeria sem que um possível usuário fosse capaz de implementá-la em seus documentos de maneira simples e eficaz. Dessa forma, a seguinte seção abordará o primeiro passo necessário para a utilização da ferramenta desenvolvida, a criação de um *template* parametrizado a partir da marcação de um arquivo de texto.

Para tal, foram utilizadas *tags* HTML para representar cada um dos subgrupos supramencionados. Assim, para demarcar um *input*, variável ou lógica basta somente inserir o termo entre colchetes angulares (< >) antes do conteúdo que se queira demarcar e repetir a marcação, desta vez adicionando uma barra entre eles, ao final do mesmo conteúdo (</>). Não obstante, para fins de parametrização e posterior geração do código, cada um dos subgrupos apresenta suas peculiaridades quanto ao padrão de marcação que deve ser seguido para melhor funcionamento e eficácia da aplicação.

Quanto aos *inputs*, o conteúdo entre as *tags* é de relativa baixa importância, de modo que a simples adição das referidas *tags* em qualquer trecho do documento é o suficiente para garantir a existência de um *input* responsável por preencher aquele segmento no gerador de documentos. Portanto, o seguinte exemplo de marcação alcançaria os resultados mencionados: “<input> Input1 </input>”. Todavia, para fins de otimização e maior clareza no momento da atribuição de valores para os *inputs* é recomendado que o conteúdo entre as *tags* seja um exemplo de valor ou uma descrição curta do que deverá ser preenchido.

As variáveis, por sua vez, devem ter as opções fixas de valores sujeitos a serem adotados, separados por uma barra ou ponto e vírgula, como o conteúdo entre as *tags*. De maneira mais simples, o *template* criado deve conter as opções utilizadas para o preenchimento de um determinado segmento dentre as marcações de variáveis. Por exemplo: “<variável> Opção 1/Opção 2</variável>”.

No que tange às *tags* de lógica, elas devem vir acompanhadas de todo o trecho que se almeja inserir baseado na lógica de simples inclusão (Ex: <lógica> Segmento ou parágrafo sobre um determinado assunto </lógica>. Alternativamente, como já mencionado, é possível também inserir mais de um conteúdo, novamente separado por uma barra ou ponto e vírgula, dentre as *tags*, nesse caso a lógica agirá como uma variável cuja opção de não inclusão é implícita.

Ademais, a aplicação desenvolvida suporta *nested tags,* de forma que é possível inserir *inputs* e variáveis dentro de lógicas e vice-versa, bem como

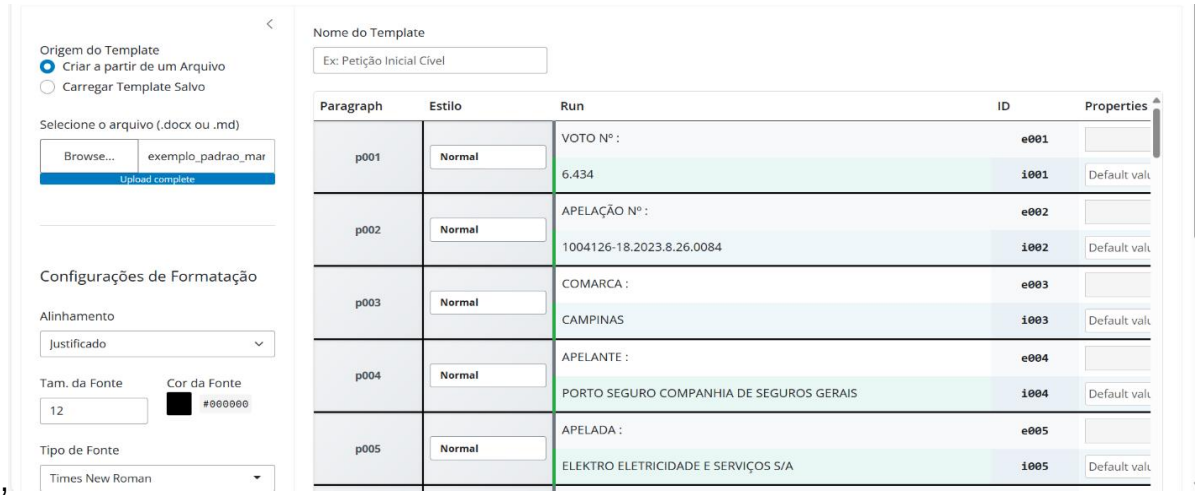
sincronização das marcações por meio de classes. A sintaxe, para essa finalidade, requer a inclusão do atributo *class* na *tag* de abertura, ao qual se deve atribuir um valor que o identifique (por exemplo, <lógica class="preliminar">). Mediante essa abordagem, o valor atribuído a um *input* é replicado para todos os demais de mesma classe, a mesma opção de uma variável valerá igualmente para outras, enquanto a inclusão de um segmento de lógica implicará na inclusão dos demais. É oportuno notar que variáveis de mesma classe devem conter, sempre, a mesma quantidade de opções. O referido recurso é essencial para lidar com casos em que uma lógica ou variável contenha mais de um parágrafo, uma vez que não há suporte para dois ou mais parágrafos dentro da mesma marcação. *Prints* de um *template* parametrizado encontram-se no Apêndice E.

Ainda nesta etapa, é facultado ao usuário definir segmentos que devem vir formatados com negrito, itálico etc. a partir de marcações clássicas de *Markdown.* Finalmente, é importante ressaltar que as marcações podem ser realizadas tanto em arquivos Word (*.docx*), para maior praticidade, ou *Markdown* (*.md*), para melhor visualização.

4.3 Uso da ferramenta: Upload de *Template* e Configurações Avançadas

Ainda que preparado um *template* conforme a técnica elaborada restam algumas definições que impediriam o uso prático efetivo da ferramenta, caso não pudessem ser configuradas, como por exemplo a definição de alinhamento, do tipo e tamanho de fonte etc. Por isso, com o upload do *template* na aplicação, antes mesmo da integração com a API, foi desenvolvida uma aba destinada à configuração do *template*.

Figura 3 – Tabela Shiny



Fonte: Elaborado pelo autor (2025)

Nesta etapa, no canto inferior esquerdo, é possível definir configurações padrão de formatação, que serão aplicadas a todos os parágrafos durante a geração de

documentos, sendo essas: alinhamento; tamanho da fonte; tipo de fonte, dentre uma lista pré-definida de opções que suporta adição de novas fontes pelo usuário; margens, superior, inferior, esquerda e direita; espaçamento, antes, depois e entre linhas, bem como recuo de primeira linha e esquerdo. Não foi adicionado suporte para recuo direito, devido à sua pouca utilização em documentos judiciais, da mesma maneira que não há suporte para instruções de cabeçalho e rodapé, uma vez que os órgãos judiciais possuem ferramentas internas, como o E-SAJ e E-PROC, responsáveis pela inclusão destes elementos.

No mais, com o *upload* do *template*, no painel direito, é renderizada uma tabela composta por 6 colunas, cujo número de linhas corresponde à quantidade de parágrafos do documento. A coluna inicial, *Paragraph*, identifica cada parágrafo com um ID sequencial (por exemplo, p001, p002, p003...). Enquanto a coluna seguinte, Estilo, oferece um menu de seleção para definir a natureza do parágrafo, com a opção padrão "Normal" — que mantém a formatação padrão do *template* — e "*Specific*", que pode ser selecionada para definir configurações específicas daquele parágrafo. Tendo sido utilizada a formatação especifica, uma nova opção, "*Inherit",* estará disponível na linha imediatamente a baixo, que replicará as configurações definidas para o parágrafo anterior.

A terceira coluna, *Runs,* filtra os diferentes elementos contidos em cada parágrafo, renderizando linhas aninhadas para cada um deles, diferenciando-os visualmente por meio de um esquema de cores: Elementos Textuais (cinza), *Inputs* (verde), Variáveis (amarelo) e Lógicas (vermelho). Esta coluna, bem como as seguintes, seguem a mesma referida estrutura de linhas aninhadas, desta forma as células correspondem agora a cada um dos elementos individualizados ao invés do parágrafo como um todo.

Na sequência, a coluna "Id" identifica cada elemento, novamente com um ID sequencial. Entretanto, nesta instância, dado os diferentes tipos de elementos, cada categoria segue sua própria sequência e é identificada pela primeira letra de sua denominação (por exemplo, para *inputs*: i001, i002...). Já a quinta coluna, *Properties*, reage de forma diferente a cada um dos elementos. Para *inputs*, a célula vem vazia por definição e é facultado ao usuário preenchê-la da maneira que julgar mais adequada, uma vez que não somente será preenchida automaticamente pelo LLM, quando vazia, mas também seu valor será posteriormente utilizado como *placeholder* para o *input* correspondente. Quanto às variáveis, as células vêm preenchidas com as opções definidas no *template* para a variável, sendo possível, nesta etapa, alterar os valores ou ainda adicionar novos. Já para as para as Lógicas, as células oferecem novamente um *dropdown*, desta vez com a opção padrão "Simples" – indicando uma lógica de simples inserção, atuando, conforme já mencionado, como variáveis que admitem valor nulo—, "-" — representando uma lógica cujo funcionamento deve ser inferido pelo LLM, essencialmente delegando essa função para o modelo — e "Custom", que automaticamente irá renderizar uma caixa de texto onde o usuário pode definir um funcionamento customizado para aquela lógica (como por exemplo sempre adicionar aquele segmento quando uma outra condição for preenchida).

Por fim, a coluna *Class* exibe o valor do atributo *class* extraído das *tags* que o possuam. O sistema também faculta ao usuário a modificação deste campo, possibilitando tanto a inserção de novas classes quanto a remoção daquelas

preexistentes. Completa a configuração é necessário inserir um nome para o *template* e salvá-lo.

Malgrado o maior enfoque dado às opções de configuração, a importância desta etapa estende-se a outra função igualmente crucial. Com o salvamento do *template* é gerada uma estrutura de dados, no formato "*json*", contendo todos os aspectos relevantes para a criação do código. Tal estrutura é uma lista, com dicionários individualizados para cada um dos parágrafos, sendo que cada dicionário possui um total de 6 itens. Id, preenchido por uma *string* referente ao id do parágrafo, "Estilo", preenchido ou pela *string* "Normal" ou por um outro dicionário com a configuração específica de formatação. Os itens subsequentes, *Inputs*, Lógicas, Variáveis e Elementos_textuais contêm listas de parâmetros específicos para cada categoria. Essas listas são vazias caso os elementos correspondentes não existam no parágrafo.

Os itens das listas possuem sempre uma identificação referente ao tipo de elemento ao qual correspondem (por exemplo, id_input, para *inputs*, variavel_label, para variáveis etc.). Todas as listas possuem, também, o item *Id*, sendo preenchido por uma *string* referente ao *id* do elemento na tabela. Para os *inputs*, variáveis e lógicas estão presentes os itens *label*, sempre uma *string* vazia; *Properties*, *string* preenchida conforme o disposto na tabela e *class,* sendo um *Null object*, quando ausente, ou uma *string* com o valor correspondente. Outrossim, comum as listas de elementos textuais e lógicas é o item "conteúdo", contendo o texto do segmento em sua integra. Por fim, somente as listas de lógicas possuem os itens *Options* e descrição_detalhada, sendo sempre *strings* vazias.

A formação da referida estrutura de dados justifica-se pela sua função de extrair as informações primordiais para a criação do código, possibilitando assim a separação, organização e identificação dos tipos de elementos e seus conteúdos, bem como a indexação das instruções de formatação a cada parágrafo. Tornando, assim, o projeto mais escalável, por não depender do LLM para extrair essas informações, o que implicaria em alto custo de tokens, bem como mais eficaz, uma vez que o conteúdo gerado pela IA tende a ser mais consistente, dado a estruturação previa dos conteúdos que devem estar presentes no documento a ser gerado pelo código. Está contidos no Apêndice F a estrutura de dados.

4.4 Uso da ferramenta: Geração do Código

Uma vez salvo o *template* com as configurações personalizadas, torna-se possível utilizá-lo para a geração do código na aba "Criar Código Gerador". Ela é responsável pela integração da API do Gemini à aplicação, realizada de tal forma qual o LLM recebe o *template* salvo no banco de dados e a partir do *structured output* gera o conteúdo solicitado.

O prompt, conforme já mencionado, foi dividido em duas partes. Assim, num primeiro momento o LLM preenche os itens que estavam vazios na estrutura de dados original e agrupa todos os dicionários das listas de *inputs*, variáveis, lógicas e elementos textuais, que estavam previamente distribuídos entre os parágrafos, em

listas únicas para cada categoria. Outrossim, o LLM gera uma nova lista, paragrafos_metadata, na qual são indexados a cada paragrafo sua configuração de formatação completa, não mais bastando-se em utilizar somente a *string* “Normal”, e uma sucinta lista, “elementos_ids”, contendo apenas a identificação dos elementos pertencentes àquele parágrafo. Dessa forma, o sistema de substituições necessário para o correto funcionamento do código dinâmico é otimizado, dado que as listas de elementos podem ser usadas como parâmetros de uma função, enquanto a quinta lista facilita a formatação correta de cada parágrafo, bem como evita “alucinações” quanto ao conteúdo de cada parágrafo.

Realizada a reestruturação dos dados é gerado também o código, que opera a partir de uma função chamada `gerar_documento()` que recebe os seguintes parâmetros: *inputs* (lista de *inputs*), variaveis (lista de variáveis), logica (lista de lógicas), elementos_textuais (lista de textos), paragrafos_metadata (metadados de formatação) e estrutura_paragrafos (estrutura original para referência). O processo iterativo é guiado pela lista paragrafos_metadata, onde cada item corresponde a um parágrafo a ser gerado no documento final. Para cada parágrafo, a função aplica primeiramente as configurações de nível de bloco, como alinhamento e espaçamento vertical, com base nos metadados de formatação. Em seguida, o conteúdo textual é processado para aplicar a formatação de nível de caractere. O código implementa estratégias para a interpretação de *Markdown* (como * para itálico e ** para negrito), segmentando o texto em múltiplos runs - que são segmentos contíguos de texto com formatação uniforme. Cada run recebe então os atributos de fonte específicos (família, tamanho e cor) extraídos da estrutura de dados. Sendo que a natureza dinâmica do documento é assegurada pela substituição de *placeholders* pelos valores contidos nas listas *inputs* e variáveis e lógica. Findada a geração do código gerador é necessário salvá-lo para, assim, utilizá-lo na aba seguinte onde são gerados os documentos dinâmicos “.docx”.

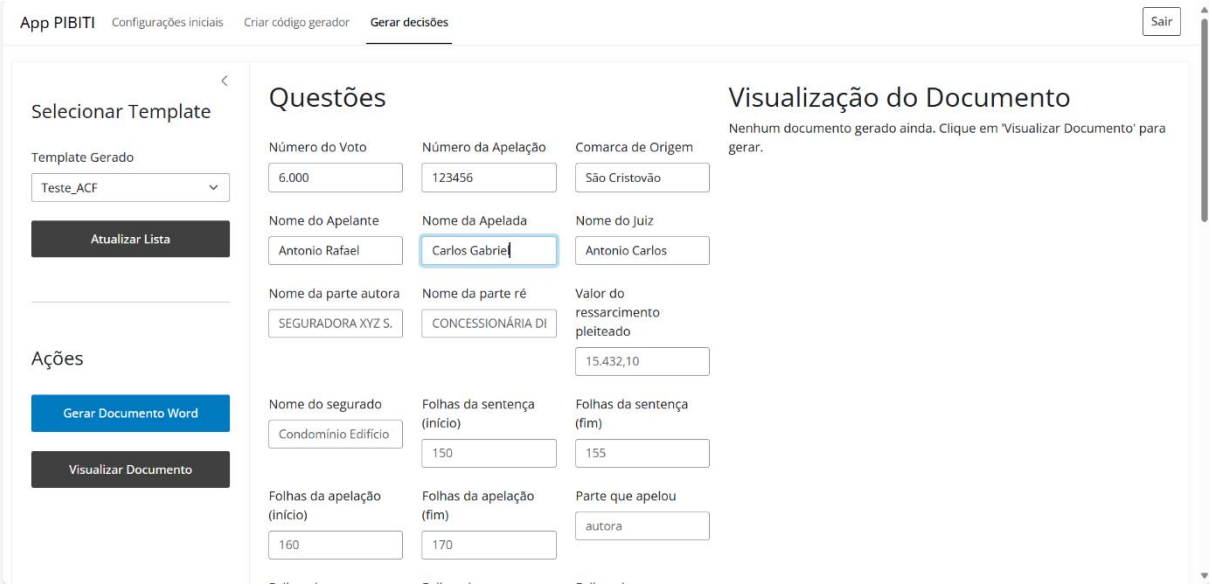
Ainda sobre o prompt é importante mencionar que a fim de garantir a maior eficácia do conteúdo gerado foi necessário encontrar um equilíbrio entre detalhamento e simplicidade das instruções passadas. Uma vez que um prompt mais elaborado e preciso é necessário para evitar “alucinações” que possivelmente pudessem levar ao não funcionamento do código. Contudo, um excesso de detalhamento limitaria a capacidade do modelo de desenvolver soluções inovadoras e robustas para os desafios inerentes à variabilidade dos projetos. Portanto, a estratégia adotada foi combinar especificações técnicas rigorosas com objetivos conceituais mais abstratos.

Ressalta-se, por fim, que é facultado ao usuário escolher entre os Modelos Gemini 2.5 Flash e Gemini 2.5 Pro (e suas respectivas versões experimentais). A decisão de limitar a aplicação a este ecossistema de LLMs surgiu uma análise comparativa aprofundada, a qual evidencia um dos principais resultados deste trabalho: até o momento, os modelos mais adequados para o desenvolvimento de soluções análogas são aqueles mencionados. Dada a sua capacidade de gerar até 64 mil *output tokens*, os referidos modelos tornam-se cruciais para garantir a escalabilidade do projeto, possibilitando que uma gama maior de *templates*, especialmente aqueles de maior complexidade e extensão, sejam utilizados.

4.5 Uso da ferramenta: Geração do Documento

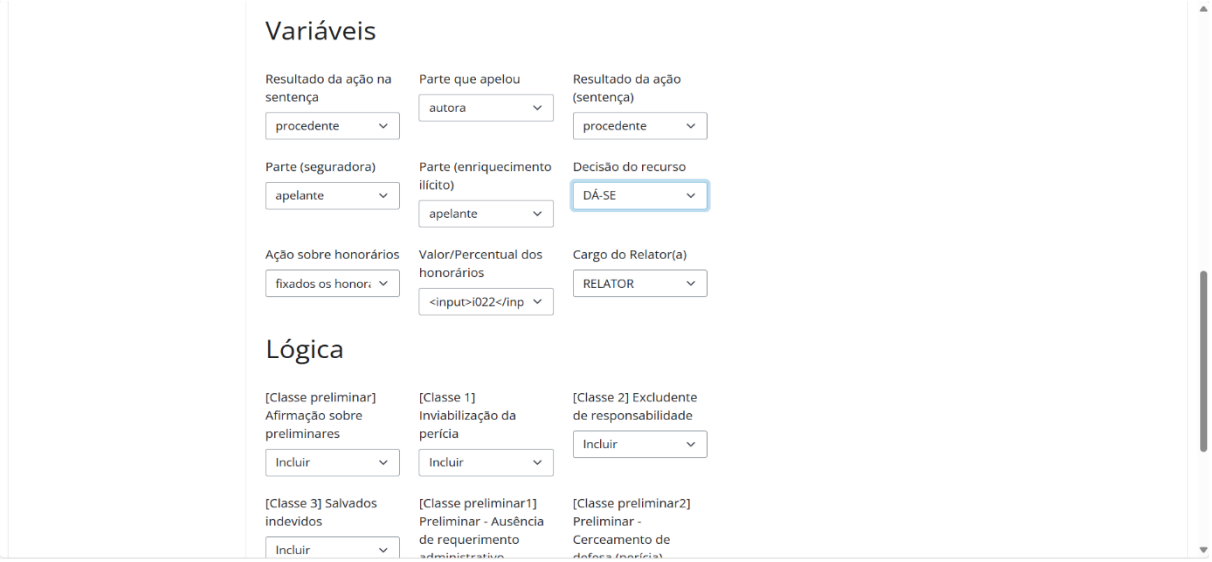
Finalmente, tendo sido gerado e salvo o código requerido, a aba “Gerar decisões” possibilita a geração dos documentos personalizados. Nela é realizada a escolha do código desejado, dentre aqueles salvos no banco de dados, ocasionando na renderização dos elementos necessários para a customização do documento. De modo que as *labels* previamente mencionadas tornam-se identificadores, servindo também como descrição, de cada elemento, as *Properties* são utilizadas como *placeholders* para os *inputs* e como opções em um *dropdown* para as variáveis. Enquanto *logica_options*, preenchida pelo LLM, torna-se as opções do *dropdown* para as lógicas.

Figura 4 – Aba destinada à geração dinâmica de documentos (*Inputs*)



Fonte: Elaborado pelo autor (2025)

Figura 5 - Aba destinada à geração dinâmica de documentos (Variáveis e Lógicas)



Fonte: Elaborado pelo autor (2025)

A partir disso, a interface do usuário, desenvolvida com o Shiny, serve como a camada de interação para a coleta de dados. A interatividade é viabilizada pela arquitetura reativa do Shiny, na qual os componentes reativos (reactives) são empregados para monitorar as entradas do usuário em tempo real, como a seleção de opções em menus e o preenchimento de campos de texto. Uma vez que os dados são coletados, a mesma reatividade aciona a execução da função gerar_documento(), iniciando o processo de instanciação do documento final. Assim, após a customização, basta apenas clicar no botão “Gerar Documento Word” que o arquivo “.docx” é baixado (...)

Figura 6 – Documento Gerado

VOTO Nº : 6.000

APELAÇÃO Nº : 123456

COMARCA : São Cristóvão

APELANTE : Antonio Rafael

APELADA : Carlos Gabriel

JUIZ : Antonio Carlos

Ação de regresso por sub-rogação. Prestação de serviços. Energia elétrica. Falha no fornecimento por sobrecarga de energia. Danificação de equipamentos. Cobertura do sinistro pela seguradora. SENTENÇA de improcedência. APELAÇÃO da seguradora autora, que pugna pela reforma total da sentença. EXAME: Preliminares afastadas. Sub-rogação da apelante seguradora nos direitos das seguradas. Inversão do ônus da prova operada “ope legis”, “ex vi” do artigo 14, §3º, do Código do Consumidor. Responsabilidade civil objetiva amparada no citado artigo 14 do CDC e no artigo 37, §6º, da Constituição Federal. Parecer que acompanha a inicial

Fonte: Elaborado pelo autor (2025)

Como é possível verificar na Figura 6, o arquivo Word gerado contém as especificações que foram determinadas na configuração inicial do *template*. Além disso, o conteúdo depende das informações que são passadas nos *inputs*, variáveis e lógicas inseridas na interface gráfica, comprovando que o algoritmo gerado pela IA foi capaz de representar as lógicas do documento anotado por humanos em um código automatizável.

4.6 Resultados Práticos: Entrevistas com Profissionais da Área

Para verificar empiricamente a eficácia da aplicação desenvolvida foram conduzidas duas entrevistas com profissionais da área. A primeira delas com uma juíza e a segunda com uma desembargadora.

A primeira entrevistada destacou o debate corrente no judiciário sobre a implementação de novas tecnologias baseadas em LLMs. A preocupação central, segundo ela, é o risco de que tais ferramentas resultem em uma delegação indevida da análise e da prerrogativa decisória do juiz para a inteligência artificial. Contudo, após interagir com a ferramenta, a magistrada concluiu que a abordagem proposta contorna esse problema, afirmando que “a solução [...] preserva o caráter humano da decisão [...] ao mesmo tempo em que permite um aumento na produtividade”. Adicionalmente, ela ressaltou o potencial da aplicação para padronizar decisões e reduzir erros humanos, qualidades que considerou essenciais para o fortalecimento da segurança jurídica.

Enquanto a segunda entrevistada, comentou que a ferramenta teria muita utilidade na elaboração de minutas para causas repetitivas, principalmente para aqueles responsáveis por elaborar as referidas minutas. Todavia, a entrevistada apontou também algumas limitações que a ferramenta possui, principalmente quando comparada aos sistemas já disponíveis no E-SAJ. Notadamente ela se referiu ao seu caráter individualizado de *templates*, de modo que não é possível compartilhar os geradores entre os membros de um mesmo gabinete, e a sua incapacidade da ferramenta de extrair automaticamente as informações do cabeçalho.

5. CONSIDERAÇÕES FINAIS E LIMITAÇÕES

A presente pesquisa partiu do desafio de desenvolver uma ferramenta para aumentar a produtividade sistema de justiça a partir de IA, sem com isso reduzir a qualidade dos textos gerados, propondo uma solução que integra a capacidade generativa de LLMs com uma interface interativa. O desenvolvimento culminou em uma ferramenta funcional que automatiza a geração de código para a instanciação de documentos personalizados, a partir de um método de parametrização robusto e de uma arquitetura de software que prioriza a eficiência e a escalabilidade.

A validação empírica, conduzida por meio de entrevistas com magistradas, corroborou a eficácia da abordagem. Demonstrou-se que a ferramenta não apenas acelera o fluxo de trabalho, mas o faz de uma maneira que preserva a prerrogativa decisória do jurista, mantendo assim o HITL. Adicionalmente, os resultados indicam um potencial significativo para a padronização de decisões e a redução de erros humanos, contribuindo para o fortalecimento da segurança jurídica.

Não obstante os resultados positivos, é fundamental reconhecer as limitações do presente trabalho. Sendo elas:

A falta de um sistema capaz de preencher campos simples (por exemplo o nome das partes) de modo automatizado, o que implica em custo de eficiência.

A ausência de suporte para o “aninhamento” de *tags* do mesmo tipo (por exemplo uma lógica dentro de uma lógica ou uma variável dentro de uma variável).

A ausência de uma função capaz de preencher diversas vezes os campos necessários para a geração do documento, que caso desenvolvido representaria um aumento ainda mais significativo na produtividade.

Todavia, cabe ressaltar que as limitações identificadas, passam longe de invalidarem a pesquisa. Especialmente em relação a ausência de suporte para o *nesting* das mesmas *tags*, uma vez que o usuário ainda é capaz de manipular estes recursos para obter resultados semelhantes, ou tão eficazes e funcionais quanto, mesmo que menos eficientes. Como por exemplo, usar uma variável de mais opções ou até mesmo adicionar mais de uma opção para a lógica, sendo uma delas o texto na integra e a outra o texto sem o trecho que seria a lógica aninhada.

Em suma, este trabalho demonstrou a viabilidade e o potencial da aplicação de LLMs como ferramentas de auxílio à produção jurídica, propondo uma arquitetura que equilibra automação e controle humano. A solução desenvolvida não apenas representa um avanço técnico na produção de documentos, mas também contribui para o debate sobre a integração responsável da inteligência artificial no campo do Direito.

Referências

ASENSI, Felipe Dutra; CHAVES, Rodrigo; PESSOA, Juliane da Silva. Pesquisa empírica sobre qualidade de vida no trabalho de magistrados da Justiça Trabalhista. **Revista de Estudos Empíricos em Direito** (Brazilian Journal of Empirical Legal Studies), v. 9, 2022. DOI: https://doi.org/10.19092/reed.v9.568.

BRASIL. Código de Processo Civil. Lei nº 13.105, de 16 de março de 2015. Institui o Código de Processo Civil. **Diário Oficial da União: seção 1**, Brasília, DF, 17 mar. 2015. Disponível em: https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2015/lei/l13105.htm. Acesso em: 16 ago. 2025.

BROWN, Tom B.; MANN, Benjamin; RYDER, Nick; SUBBIAH, Melanie; KAPLAN, Jared; DHARIWAL, Prafulla; et al. Language models are few-shot learners. In: **Advances in Neural Information Processing Systems**, v. 33, p. 1877-1901, 2020.

CALLOWAY, Jim. Thoughts on Legal Document Automation. **Oklahoma Bar Journal**, Law Practice Tips, janeiro de 2025. Disponível em: https://www.okbar.org/lpt_articles/thoughts-on-legal-document-automation/. Acesso em: 16 ago. 2025.

CONSELHO NACIONAL DE JUSTIÇA. ASSOCIAÇÃO BRASILEIRA DE JURIMETRIA. **Os maiores litigantes em ações consumeristas**: mapeamento e proposições. Brasília: CNJ, 2017. Disponível em: https://www.cnj.jus.br/wp-content/uploads/2021/09/just-pesquisa-maiores-litigantes2018.pdf. Acesso em: 16 ago. 2025.

CONSELHO NACIONAL DE JUSTIÇA. Justiça em Números 2024. Brasília: CNJ, 2024. 448 p. Disponível em: https://www.cnj.jus.br/wp-content/uploads/2025/04/justica-em-numeros-2024.pdf. Acesso em: 16 ago. 2025.

DAHL, Matthew; MAGESH, Varun; SUZGUN, Mirac; HO, Daniel E. Large legal fictions: profiling legal hallucinations in large language models. **Journal of Legal Analysis**, v. 16, n. 1, p. 64–93, 2024. DOI: https://doi.org/10.1093/jla/laae003.

ELOUNDOU, Tyna; MANNING, Sam; MISHKIN, Pamela; ROCK, Daniel. GPTs are GPTs: Labor market impact potential of LLMs. **Science**, Washington, DC, v. 384, n. 6702, p. 1306-1308, 21 jun. 2024. DOI: https://doi.org/10.1126/science.adj0998.

FONSECA, Felipe Camara; LIMA, Walber Cunha. A importância da mediação e conciliação no combate ao congestionamento do Poder Judiciário brasileiro. **Revista de Estudos Jurídicos do UNI-RN**, Natal, n. 5. 2021.

GEMINI TEAM GOOGLE. **Gemini**: A Family of Highly Capable Multimodal Models. arXiv, 19 dez. 2023. arXiv:2312.11805 [cs.CL]. Última versão em 9 maio 2025. Disponível em: https://arxiv.org/abs/2312.11805. Acesso em: 18 ago. 2025.

LAMPLE, Guillaume; BALLERINI, Miguel; BENGIO, Samy; et al. Neural Architectures for Named Entity Recognition. In: **Proceedings of NAACL-HLT**, 2016.

JURAFSKY, Daniel; MARTIN, James H. Speech and Language Processing: **An Introduction to Natural Language Processing, Computational Linguistics, and**

Speech Recognition. 3. ed. (draft online). Prentice Hall, 2023. Disponível em: https://web.stanford.edu/~jurafsky/slp3/.

LAURITSEN, Marc; SOUDAKOFF, Alan. **Power tools for document preparation**. 1998. Disponível em: http://www.capstonepractice.com/amlaw6.pdf. Acesso em: 16 ago. 2025.

LU, Chien-Ping. Industrial technology advances: AI, native supercomputing and the revival of Moore's Law. **National Science Review**, v. 5, n. 1, p. 24–26, 2018. DOI: https://doi.org/10.1093/nsr/nwx004.

MCKINNEY, S.; WILSON, M. **python-docx**: Create and update Microsoft Word (.docx) files. Versão 1.1.2. 2024. Disponível em: https://github.com/python-openxml/python-docx. Acesso em: 16 ago. 2025.

MOSQUEIRA-REY, Eduardo; HERNÁNDEZ-PEREIRA, Elena; ALONSO-RÍOS, David; BOBES-BASCARÁN, José; FERNÁNDEZ-LEAL, Ángel. Human-in-the-loop machine learning: a state of the art. **Artificial Intelligence Review**, [s. l.], v. 56, p. 3005–3054, 2023. DOI: https://doi.org/10.1007/s10462-022-10246-w.

PEREIRA, Jayr; ASSUMPCAO, Andre; TRECENTI, Julio; AIROSA, Luiz; LENTE, Caio; CLÉTO, Jhonatan; DOBINS, Guilherme; NOGUEIRA, Rodrigo; MITCHELL, Luis; LOTUFO, Roberto. INACIA: Integrating Large Language Models in Brazilian Audit Courts: Opportunities and Challenges. **Digital Government: Research and Practice**, v. 6, n. 1, p. 1-20, 2025. DOI: https://doi.org/10.1145/3652951.

R CORE TEAM. R: A language and environment for statistical computing. Vienna: **R Foundation for Statistical Computing**, 2024. Disponível em: https://www.R-project.org/. Acesso em: 18 ago. 2025.

TORQUATO, João; TRECENTI, Julio; WITKOSKI, Katerine; FELIZ, Ricardo. Associação Brasileira de Jurimetria: **Litigiosidade e desenvolvimento**. 2021. Disponível em: https://lab.abj.org.br/posts/2021-01-15-litigiosidade/. Acesso em: 16 ago. 2025.

VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Łukasz; POLOSUKHIN, Illia. Attention is all you need. In: **Proceedings of the 31st International Conference on Neural Information Processing Systems** (NIPS 2017). Long Beach: Curran Associates, 2017. p. 6000–6010. Disponível em: https://arxiv.org/abs/1706.03762. Acesso em: 16 ago. 2025.

WANG, Yaqing; YAO, Quanming; KWOK, James T.; NIU, Lionel M. Generalizing from a few examples: **A survey on few-shot learning. ACM Computing Surveys**, v. 53, n. 3, p. 1-34, 2020. DOI: https://doi.org/10.1145/3386252.

Apêndice A – Parametrização Inicial

Figura A.1 – Excerto de *Template* Inicial

VOTO N° : número do voto

APELAÇÃO N° : número de apelação

COMARCA : comarca

APELANTE : parte apelante

APELADA : parte apelada

JUIZ : juiz

Vistos.

Trata-se de "AÇÃO DE RESSARCIMENTO DE DANOS" ajuizada por parte autora contra parte ré, visando à condenação da ré ao pagamento de R$ valor, a título de regresso, ante a cobertura securitária paga ao segurado nome do segurado, pelos danos causados aos seus bens.

A r. sentença apelada (fls. 225/235) julgou (procedente/improcedente) a ação.

Apelou a (autora/ ré) a fls. 231/254.

Contrarrazões a fls. 263/273.

É o relatório.

Fonte: Elaborado pelo autor (2025)

Apêndice B- Código Desenvolvido Manualmente

Quadro B.1 - Código “teste_avançado”

```
import docx
from docx.shared import Pt, Inches

# Coleta de dados iniciais para o Voto
Quem_Apelou = input("A apelação é da parte autora?").lower()
n_voto = input("Número do Voto:")
n_apelacao = input("Número da Apelação:")
comarca = input("Comarca:")
apelante = input("Apelante:")
apelada = input("Apelada:")
juiz = input("Juiz:")
valor_da_acao = input("Valor pleiteado:")
N_De_Segurados = input("Existe mais de um segurado?")
Genero_Segurado = input("O segurado é do gênero masculino?")
segurado = input("Nome do(s) segurado/a(s):")
nc_sentenca = input("Fls em que começa a sentença:")
nf_sentenca = input("Fls em que termina a sentença:")
# ... (restante da coleta de dados)

# Lógica condicional para definir variáveis com base na parte apelante
if Quem_Apelou in ['s', 'sim', '1', 'true']:
    d_sentenca = "improcedente"
    parte_apelante = "autora"
    concessionaria = "apelada"
    seguradora = "apelante"
    jurisprudencia_1 = "APELAÇÃO. SEGURO RESIDENCIAL..."
    jurisprudencia_2 = "Consumidor e processual. Fornecimento de energia
elétrica..."
    r_recurso = "dá-se"
    majorando = "e honorários advocatícios, fixando-os em"
    autora = apelante
    re = apelada
    CPC = ""
else:
    d_sentenca = "procedente"
    parte_apelante = "ré"
    concessionaria = "apelante"
    seguradora = "apelada"
    jurisprudencia_1 = "ENERGIA ELÉTRICA – Ação regressiva de
ressarcimento de danos..."
    jurisprudencia_2 = "AÇÃO DE REGRESSO POR SUB-ROGAÇÃO. Prestação de
serviços..."
    r_recurso = "nega-se"
    majorando = "majorados os honorários advocatícios para"
    autora = apelada
    re = apelante
    CPC = " e 11 °"

# Lógica para definir sufixos de gênero e número
if Genero_Segurado in ['s', 'sim', '1', 'true'] and N_De_Segurados not in
['s', 'sim', '1', 'true']:
    sufixo = "o"
elif Genero_Segurado in ['s', 'sim', '1', 'true'] and N_De_Segurados in
['s', 'sim', '1', 'true']:
    sufixo = "os"
```

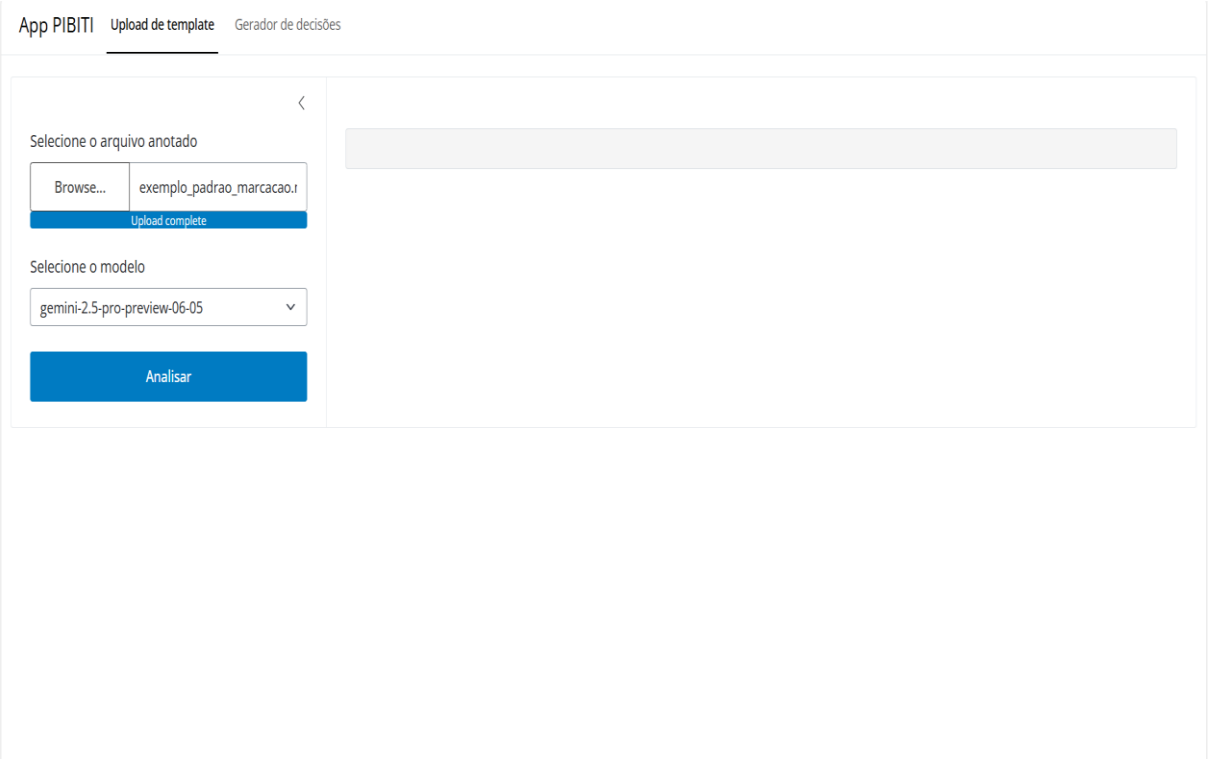
```
elif Genero_Segurado not in ['s', 'sim', '1', 'true'] and N_De_Segurados
not in ['s', 'sim', '1', 'true']:
    sufixo = "a"
elif Genero_Segurado not in ['s', 'sim', '1', 'true'] and N_De_Segurados
in ['s', 'sim', '1', 'true']:
    sufixo = "as"

# Coleta de dados sobre preliminares e argumentos de mérito
PRELIMINAR_INTERESSE_DE_AGIR = input("Foi alegado falta de intersse de
agir?").lower()
PRELIMINAR_CERCEAMENTO_DE_DEFESA = input("Foi alegado cerceamento de
defesa?").lower()
# ... (restante da coleta de dados)

# Lógica condicional para construir seções de preliminares
if PRELIMINAR_INTERESSE_DE_AGIR  in ['s', 'sim', '1', 'true']:
    preliminar_1 = "Afasta-se de proêmio a preliminar de ausência de
prévio requerimento..."
    jurisprudencia_relacionada_1 = "APELAÇÃO CÍVEL. Ação regressiva de
ressarcimento de danos..."
# ... (restante da lógica de preliminares)

# Lógica condicional para construir seções de mérito
if ORIGEM_DO_DANO  in ['s', 'sim', '1', 'true']:
    x = "Malgrado tenha a concessionária invocado a tese de ausência de
prova..."
# ... (restante da lógica de mérito)

# Lógica para definir honorários
if EQUIDADE  in ['s', 'sim', '1', 'true']:
    valor_honorarios_1 = input("Valor dos honorarios:")
    honorarios = f"R$ {valor_honorarios_1}, por equidade..."
else:
    valor_honorarios_2 = input("Valor dos honorarios:")
    honorarios = f"{valor_honorarios_2}% do valor da condenação..."

# Geração do Documento Word (.docx)
VOTO = docx.Document()

# Adiciona o cabeçalho do Voto
VOTO_N = VOTO.add_paragraph()
VOTO_N.add_run(f"VOTO N°\t\t: {n_voto}".upper())
# ... (formatação do cabeçalho)

# Adiciona o primeiro parágrafo (Relatório)
PARAGRAFO_1 = VOTO.add_paragraph()
PARAGRAFO_1.add_run("Trata-se de “ação de ressarcimento de danos”...")
# ... (restante da construção do relatório)

# Adiciona parágrafos sobre o andamento processual
PARAGRAFO_2 = VOTO.add_paragraph()
PARAGRAFO_2.add_run(f"A r. sentença apelada (fls.
{nc_sentenca}/{nf_sentenca})...")
# ... (restante dos parágrafos)

# Adiciona as seções de preliminares, se aplicável
if PRELIMINAR_INTERESSE_DE_AGIR in ['s', 'sim', '1', 'true']:
    PRELIMINAR_1 = VOTO.add_paragraph()
    PRELIMINAR_1.add_run(f"{preliminar_1}")
    # ... (restante da construção das preliminares)
```

```
# Adiciona os parágrafos de mérito
PARAGRAFO_11 = VOTO.add_paragraph()
PARAGRAFO_11.add_run(f"No mérito, a relação contratual...")
# ... (restante da construção do mérito)

# Adiciona a conclusão e dispositivo
PARAGRAFO_19 = VOTO.add_paragraph()
PARAGRAFO_19.add_run(f"Ante o exposto, {r_recurso} PROVIMENTO AO
RECURSO...")
# ... (restante da construção da conclusão)

# Salva o documento final
VOTO.save(f"{n_voto} - {n_apelacao}.docx")
```

Fonte: Elaborado pelo autor (2024)

Apêndice C – Aplicação Antiga

Figura C.1 – Upload de *Template* na Aplicação Antiga



Fonte: Elaborado pelo autor (2025)

Figura C.2 – Aba destinada à geração de Documentos na aplicação antiga

Fonte: Elaborado pelo autor (2025)

Apêndice D – Códigos do Shiny e da API

Quadro D.1 - Código da aplicação Shiny

```
import json
import logging
import os
import re
import shutil
import subprocess
import tempfile

import pandas as pd
from dotenv import load_dotenv
from docx import Document
from pathlib import Path
from requests import session
from shiny import App, reactive, render, ui
from shiny.types import FileInfo

# Importação de módulos locais para interação com o banco de dados e
lógica de negócio
from src.database import (
    delete_template,
    get_template_names,
    load_config_json,
    save_config,
    get_config_names,
    save_generated_template,
    load_generated_template
)
from src.wordgen.analisar_template_unificado import
analisar_template_unificado

# Configuração do sistema de logging para monitoramento da aplicação
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
logger = logging.getLogger(__name__)
load_dotenv()

# Lista de fontes comuns para preencher as opções da interface
common_fonts_list = sorted(list(set([
    "Arial", "Times New Roman", "Calibri", "Courier New", "Verdana",
"Georgia",
    "Helvetica", "Roboto", "Open Sans", "Lato", "Montserrat", "Garamond"
    # ... (lista abreviada para concisão)
])))

# --- Definição da Interface do Usuário (UI) ---
app_ui = ui.page_navbar(
    # ... (código completo da UI aqui, incluindo as três abas e seus
componentes)
    title="App PIBITI",
    id="page",
)

# --- Lógica do Servidor (Backend) ---
def server(input, output, session):
```

```
    logger.info(f"Nova sessão de usuário iniciada: {session.id}")

    # Valores reativos para armazenar o estado da aplicação
    processed_data = reactive.Value()
    generated_code_data = reactive.Value(None)
    table_changes = reactive.Value({})

    # --- Lógica da Aba 1: Configurações Iniciais ---

    @reactive.effect
    @reactive.event(input.template_file)
    def _process_uploaded_file():
        """Processa o arquivo .docx ou .md enviado pelo usuário."""
        # ... (código de processamento de arquivo aqui)
        json_data = _extract_data_from_paragraphs(paragraphs)
        processed_data.set(json_data)

    @render.ui
    def tabela_word():
        """Renderiza a tabela interativa para edição do template."""
        # ... (código HTML e JavaScript para a tabela aqui)
        return ui.HTML(table_html + js_code)

    @reactive.effect
    @reactive.event(input.save_button)
    def _save_to_db():
        """Salva a configuração do template no banco de dados."""
        # ... (código para aplicar mudanças e salvar no DB aqui)
        save_config(template_name=name, config_data=config_data,
user_id=user_id)
        update_template_list()

    # --- Lógica da Aba 2: Criar Código Gerador ---

    @reactive.effect
    @reactive.event(input.generate_code)
    def _generate_code():
        """Envia a configuração para o LLM e gera o código Python."""
        config_name = input.config_to_load()
        selected_model = input.model_selection()
        config_data = load_config_json(config_name)

        # Invoca a função que interage com a API do LLM
        result_json = analisar_template_unificado(config_data,
modelo=selected_model)
        output_modelo = json.loads(result_json)
        generated_code_data.set(output_modelo)

    @render.code
    def codigo_gerado():
        """Exibe o código Python gerado pela IA."""
        analise = generated_code_data.get()
        if not analise:
            return "# Clique em 'Gerar Código' para ver o código Python
aqui"
        return analise.get("codigo", "# Nenhum código foi gerado")

    # --- Lógica da Aba 3: Gerar Decisões ---

    @render.ui
    def questoes():
```

```
        """Renderiza dinamicamente os campos de input para o usuário."""
        # ... (código para gerar inputs aqui)
        return ui.layout_columns(*ui_elements)

    @render.ui
    def variaveis():
        """Renderiza dinamicamente os menus de seleção para as
variáveis."""
        # ... (código para gerar selects aqui)
        return ui.layout_columns(*ui_elements)

    @render.ui
    def logica():
        """Renderiza dinamicamente os menus de seleção para a lógica
condicional."""
        # ... (código para gerar selects de lógica aqui)
        return ui.layout_columns(*ui_elements)

    @render.download(filename="decisao.docx")
    def gerar_word():
        """
        Handler principal para a geração do documento final.
        Este é o trecho que ilustra a execução do código gerado pela IA.
        """
        template_name = input.template_to_use()
        if not template_name:
            ui.notification_show("Selecione um template primeiro!",
type="error")
            return

        template_data = load_generated_template(template_name)
        codigo = template_data.get('codigo', '')
        if not codigo:
            ui.notification_show("Template não possui código gerado!",
type="error")
            return

        try:
            # 1. Executa o código gerado pela IA para definir a função
`gerar_documento`
            exec(codigo, globals())

            # 2. Obtém a estrutura de dados completa do template
            template_completo = template_data.get('template_completo',
{})
            inputs = template_completo.get('inputs', [])
            variaveis = template_completo.get('variaveis', [])
            # ... (obtenção dos outros parâmetros)

            # 3. Coleta as decisões do usuário a partir da interface
reativa
            questoes = lista_questoes()
            variaveis_usuario = lista_variaveis()
            logica_usuario = lista_logica()

            # 4. Atualiza as estruturas de dados com as escolhas do
usuário
            for questao in questoes:
                for inp in inputs:
                    if inp['id_input'] == questao['id']:
                        inp['user_choice'] = questao['input_properties']
```

```
            # ... (lógica similar para variáveis e lógicas)

            # 5. Invoca a função gerada pela IA com os dados atualizados
            resultado = gerar_documento(
                inputs, variaveis, logica, elementos_textuais,
                paragrafos_metadata, estrutura_paragrafos
            )

            doc = resultado[0] if isinstance(resultado, tuple) else
resultado

            # 6. Salva o documento em um arquivo temporário para download
            with tempfile.TemporaryDirectory() as tmpdir:
                final_path = os.path.join(tmpdir, "decisao.docx")
                doc.save(final_path)
                return final_path

        except Exception as e:
            ui.notification_show(f"Erro ao gerar documento: {str(e)}",
type="error")

# Inicialização da aplicação Shiny
app = App(app_ui, server,
static_assets=os.path.join(os.path.dirname(__file__), "static"))
```

Fonte: Elaborado pelo autor (2025)

Quadro D.2 – Código Structured Output Gemini

```
import json
import os
from datetime import datetime
from typing import List, Optional

from dotenv import load_dotenv
from google import genai
from google.genai import types
from pydantic import BaseModel, Field

# Carrega variáveis de ambiente e inicializa o cliente da API do Gemini
assert load_dotenv()
client = genai.Client(api_key=os.getenv("GENAI_API_KEY"))

# --- Definição dos Modelos Pydantic para Structured Output ---
# Estes modelos definem um esquema rigoroso para a resposta da API do
LLM,
# garantindo que a saída seja previsível, validada e fácil de processar.

class InputFlat(BaseModel):
    """
    Define a estrutura para um elemento de input.
    """
    id_input: str = Field(..., description="ID do input (ex: i001,
i002)")
    label_input: str = Field(default="", description="Label descritivo do
input - deve ser preenchido sempre")
    input_properties: str = Field(default="",
description="Propriedades/placeholder do input - deve preencher se
vazio")
    input_class: Optional[str] = Field(None, description="Classe CSS do
input - pode ser None se não definida")
```

```
class VariavelFlat(BaseModel):
    """
    Define a estrutura para um elemento de variável.
    """
    id_var: str = Field(..., description="ID da variável (ex: v001,
v002)")
    label_var: str = Field(default="", description="Label descritivo da
variável - deve ser sempre preenchido")
    variavel_properties: str = Field(..., description="Opções da variável
separadas por ponto e vírgula (ex: 'opcao1;opcao2;opcao3')")
    variavel_class: Optional[str] = Field(None, description="Classe CSS
da variável - pode ser None se não definida")

class LogicaFlat(BaseModel):
    """
    Define a estrutura para um elemento de lógica condicional.
    """
    id_logic: str = Field(..., description="ID da lógica (ex: l001,
l002)")
    paragraph_id: str = Field(..., description="ID do parágrafo que
contém esta lógica - referência para cross-referência")
    label_logic: str = Field(default="", description="Label descritivo da
lógica - deve ser sempre preenchido")
    logica_properties: str = Field(..., description="Tipo da lógica (ex:
'Simples', 'Custom', aparecendo customizada, '-' para ser inferida')")
    logica_conteudo: str = Field(..., description="Conteúdo textual
dentro da tag <lógica>")
    logica_descricao_detalhada: str = Field(..., description="Descrição
detalhada da lógica condicional para implementação no código - deve ser
sempre preenchido")
    logica_options: str = Field(default="", description="Opções
específicas para a lógica separadas por ponto e vírgula - deve ser
preenchido sempre que aplicável")
    logica_class: Optional[str] = Field(None, description="Classe CSS da
lógica - pode ser None se não definida")

class ElementoTextualFlat(BaseModel):
    """
    Define a estrutura para um segmento de texto estático.
    """
    id_elemento: str = Field(..., description="ID do elemento (ex: e001,
e002)")
    conteudo_elemento: str = Field(..., description="Texto do elemento")

class FormattingDict(BaseModel):
    """Define o esquema para o dicionário de formatação de parágrafos."""
    font_size: Optional[int] = Field(default=12)
    font_family: Optional[str] = Field(default="Times New Roman")
    font_color: Optional[str] = Field(default="#000000")
    alignment: Optional[int] = Field(default=3)
    margin_top: Optional[float] = Field(default=2.5)
    margin_bottom: Optional[float] = Field(default=2.5)
    margin_left: Optional[float] = Field(default=3.0)
    margin_right: Optional[float] = Field(default=3.0)
    paragraph_spacing_before: Optional[int] = Field(default=6)
    paragraph_spacing_after: Optional[int] = Field(default=6)
    line_spacing: Optional[float] = Field(default=1.5)
    first_line_indent_cm: Optional[float] = Field(default=1.25)
    left_indent: Optional[float] = Field(default=0)

class ParagrafoMetadata(BaseModel):
```

```
    """Define a estrutura para os metadados de cada parágrafo."""
    id: str = Field(..., description="ID do parágrafo (ex: p001, p002)")
    formatting: FormattingDict = Field(default_factory=FormattingDict,
description="Dicionário de formatação")
    elementos_ids: List[str] = Field(default=[], description="Lista
ordenada de IDs dos elementos neste parágrafo")

class OutputTemplate(BaseModel):
    """
    Modelo Pydantic principal que define a estrutura completa da resposta
da API.
    Este é o "contrato de dados" que o LLM deve seguir.
    """
    inputs: List[InputFlat] = Field(default=[], description="Lista plana
de todos os inputs")
    variaveis: List[VariavelFlat] = Field(default=[], description="Lista
plana de todas as variáveis")
    logica: List[LogicaFlat] = Field(default=[], description="Lista plana
de todos os elementos de lógica")
    elementos_textuais: List[ElementoTextualFlat] = Field(default=[],
description="Lista plana de todos os elementos de texto")
    paragrafos_metadata: List[ParagrafoMetadata] = Field(default=[],
description="Lista de metadados dos parágrafos")
    codigo: str = Field(..., description="Código Python gerado para criar
o documento Word")

# --- Função Principal de Interação com a API ---

def analisar_template_gemini(config_data, ex_codigo_gerado='Nenhum
exemplo fornecido', modelo='gemini-2.5-pro-preview-06-05'):
    """
    Analisa um template, enviando-o para a API do Gemini e esperando uma
resposta estruturada.

    Args:
        config_data: Dicionário com a estrutura do template e
configurações de formatação.
        ex_codigo_gerado: String de um exemplo de código para guiar o
modelo (opcional).
        modelo: Identificador do modelo Gemini a ser utilizado.

    Returns:
        Um dicionário Python com a resposta da API, já validada pelo
Pydantic.
    """
    with open('prompts/estruturar_dados_e_codigo.md', "r", encoding="utf-
8") as f:
        prompt = f.read()
    with open('prompts/template_msg.md', "r", encoding="utf-8") as f:
        msg_template = f.read()

    template_content = config_data.get('config_content', {})
    config_json_str = json.dumps(template_content, indent=2,
ensure_ascii=False)

    formatting_info = f"""
    CONFIGURAÇÕES DE FORMATAÇÃO OBRIGATÓRIAS:
    - Alinhamento: {config_data.get('alignment', 'Justificado')}
    - Tamanho da fonte: {config_data.get('font_size', 12)}pt
    - Família da fonte: {config_data.get('font_family', 'Times New
Roman')}
```

```
    - Cor da fonte: {config_data.get('font_color', '#000000')}
    - etc...
    """

    msg = msg_template.format(
        exemplo_padrao_marcacao=config_json_str,
        exemplo_codigo_gerado=ex_codigo_gerado,
        regras_formatacao=formatting_info
    )

    full_prompt = prompt + "\n\n" + msg

    # Realiza a chamada à API do Gemini, especificando o esquema de saída
(Pydantic)
    response = client.models.generate_content(
        model=modelo,
        contents=full_prompt,
        config = types.GenerateContentConfig(
            temperature=0.0,
            response_mime_type='application/json',
            response_schema=OutputTemplate,
        )
    )

    output_text = response.text
    if output_text is None:
        raise ValueError("A resposta da API foi nula, possivelmente o
limite de tokens foi atingido.")

    result = json.loads(output_text)
    return result

# --- Funções Auxiliares para Manipulação de Dados ---

def completar_output_com_dados_originais(output_modelo, config_data):
    """Adiciona metadados ao output do modelo antes de salvar."""
    output_completo = {
        **output_modelo,
        'metadados': {
            'modelo_usado': 'gemini-2.5-pro-preview-06-05',
            'versao_analise': '3.0',
            'timestamp': datetime.now().isoformat()
        }
    }
    return output_completo

def salvar_template_completo(template_name, output_completo, user_id):
    """Salva o template processado e o código gerado no banco de
dados."""
    from ..database import get_db_connection
    import psycopg2

    conn = None
    try:
        conn = get_db_connection()
        with conn.cursor() as cur:
            # Lógica de inserção/atualização no banco de dados
            conn.commit()
    except psycopg2.Error as error:
        if conn:
            conn.rollback()
```

```
        print(f"Erro ao salvar template completo: {error}")
        raise error
    finally:
        if conn:
            conn.close()
```

Fonte: Elaborado pelo autor (2025)

Apêndice E – Template Parametrizado com *tags* e *Markdown*

Figura E.1 – Excertos de *Template* parametrizado com *tags* e *markdown* (Parte 1)

```
VOTO Nº     : <input>5.001</input>

APELAÇÃO Nº : <input>12345</input>

COMARCA     : <input>São Paulo</input>

APELANTE    : <input>Parte Apelante</input>

APELADA     : <input>Parte ré</input>

JUIZ        : <input>juiz</input>

**Ação de regresso por sub-rogação. Prestação de serviços. Energia elétrica. Falha no fornecimento por sobrecarga de energia. Danificação de
equipamentos. Cobertura do sinistro pela seguradora. SENTENÇA de improcedência. APELAÇÃO da seguradora autora, que pugna pela reforma total da
sentença. EXAME:<lógica> Preliminares afastadas.</lógica> Sub-rogação da apelante seguradora nos direitos das seguradas. Inversão do ônus da prova
operada "ope legis", "ex vi" do artigo 14, §3º, do Código do Consumidor.  Responsabilidade civil objetiva amparada no citado artigo 14 do CDC e no
artigo 37, §6º, da Constituição Federal. Parecer que acompanha a inicial elaborado por empresa independente e presumivelmente idônea. Prova dos
autos que favorece a seguradora, uma vez constatada a comprovação da ocorrência do sinistro, do dano e do nexo de causalidade entre um e outro, e
ainda do desembolso em razão da cobertura dos prejuízos às seguradas. Falha da concessionária na prestação do serviço às seguradas da autora.
<lógica class=1> Inviabilização da perícia técnica pelo decurso do tempo.</lógica><lógica class=2> Excludente de responsabilidade civil por força
maior. Inocorrência. Fortuito interno.</lógica><lógica class=3> Salvados indevidos diante da ausência de comprovação de aproveitamento do bem
danificado pela apelante.</lógica> Pretensão de reembolso, pela sub-rogação, que deve ser acolhida com a reforma da sentença prolatada. Correção
monetária incidente a contar do desembolso, "ex vi" da Súmula 43 do C. Superior Tribunal de Justiça, e juros de mora a contar da citação, "ex vi"
do artigo 405 do Código Civil. RECURSO PROVIDO.**

Vistos.

Trata-se de "**AÇÃO DE RESSARCIMENTO DE DANOS**," ajuizada por <input>autor</input> contra <input>réu</input>, visando à condenação da ré ao
pagamento de R$ <input>valor da causa</input>, a título de regresso, ante a cobertura securitária paga ao segurado <input>MITRA DIOCESANA DE
CARAGUATATUBA</input>, pelos danos causados aos seus bens.

A r. sentença apelada (fls. <input>225</input>/<input>228</input>) julgou <variável>procedente/improcedente</variável> a ação.

Apelou a <variável>(autora/ ré/)</variável> a fls. <input>231</input>/<input>244</input>. <input>Preliminarmente alega, em síntese, que se
caracterizou cerceamento de defesa. No mérito, alega que foi produzida prova capaz de demonstrar cabalmente o nexo causal que enseja o
ressarcimento. Os documentos juntados contêm força probatória, por serem elaborados por empresa especializada e idônea. O caso deve ser analisado
sob a luz do Código de Defesa do Consumidor, culminando assim na responsabilidade objetiva da ré e na inversão do ônus da prova.</input>
```

Fonte: Elaborado pelo autor (2025)

Figura E.1 – Excertos de *Template* parametrizado com *tags* e *markdown* (Parte 2)

```
É o relatório.

Recurso tempestivo e preparado, ante fls. <input>245</input>/<input>246</input>.
A apelação comporta conhecimento, uma vez observados os requisitos de admissibilidade inseridos nos artigos 1.009 e seguintes do Código de
Processo Civil.

Cuida-se de apelo que se insurge contra a r.sentença que julgou <variável>(procedente/improcedente)</variável> ação regressiva movida pela
seguradora em face de sub-rogação gerada pelo pagamento de indenização securitária efetuado em consequência de dano decorrente de oscilação de
energia elétrica nos bens do segurado.

<lógica class=preliminar1>Afasta-se de proêmio a preliminar de ausência de prévio requerimento administrativo. O que se privilegia, neste cenário,
é a ativação do comando constitucional do direito de ação, insculpido no artigo 5º, XXXV da Constituição da República. Assim, prevalece o
entendimento desta Câmara:</lógica>

<lógica class=preliminar1>***APELAÇÃO CÍVEL. Ação regressiva de ressarcimento de danos. Oscilação na rede de distribuição de energia elétrica.
Sentença de procedência. Inconformismo da ré. Não acolhimento. Desnecessidade de prévio requerimento administrativo que não tem o condão de se
sobrepor ao direito de ação, garantido pelo artigo 5º, inciso XXXV, da Constituição Federal. Queima de equipamentos eletroeletrônicos. Prova
documental produzida pela autora hábil a comprovar o nexo de causalidade entre a variação de energia fornecida pela ré e os danos nos equipamentos
dos Condominios segurados. Laudos técnicos apresentados pela autora emitidos por empresas terceirizadas, idôneas e especializadas, sem interesse
na causa. Relação de consumo. Incidência do CDC ao caso concreto  sub-rogação. Responsabilidade objetiva da concessionária ré (art. 14, do CDC c.
c. art. 37, §6º, da CF/88). Ré que não comprovou a exclusão de sua responsabilidade (art. 373, II, do CPC c.c art. 14, §3º do CDC). Sentença
mantida. RECURSO DESPROVIDO.(TJSP; Apelação  Cível  1044577-71.2022.8.26.0100;  Relator (a): Sergio Alfieri; Órgão Julgador: 27ª Câmara de
Direito Privado; Foro Regional II - Santo Amaro - 4ª Vara Cível; Data do Julgamento: 31/03/2023; Data de Registro: 31/03/2023)***.</lógica>

<lógica class=preliminar2>Não vinga a preliminar de cerceamento de defesa devido à falta de prova pericial simplificada. A perícia simplificada
não seria capaz de comprovar o que deseja a companhia ré. Devido ao transcurso do tempo, ao local em que ocorreu o sinistro e ao estado das
instalações nele contidas, dificilmente estariam inalterados os equipamentos, para que fosse possível conclusão segura a partir da prova
pretendida. Outrossim, como cediço o juiz é o destinatário da prova, daí porque a ele cabe indeferir aquelas que se mostrarem desnecessárias ao
seu convencimento, como preconizado pelo art. 370 do Código de Processo Civil. Nesse Sentido:</lógica>
```

Fonte: Elaborado pelo autor (2025)

Apêndice F – Estrutura de Dados

O presente apêndice detalha a estrutura de dados hierárquica, em formato JSON, que é posteriormente enviada ao LLM. O Quadro F.1 apresenta dois exemplos de objetos de parágrafo para ilustrar como a estrutura acomoda tanto parágrafos simples quanto aqueles com múltiplos componentes dinâmicos.

Quadro F.1 - Excerto da estrutura de dados hierárquica em formato JSON


```
[
  {
    "id": "p001",
    "estilo": {
      "type": "Specific",
      "formatting": {
        "alignment": "Esquerda",
        "font_size": 15,
        "font_color": "#000000",
        "margin_top": 2.5,
        "font_family": "Times New Roman",
        "left_indent": 0,
        "margin_left": 3,
        "line_spacing": 1.5,
        "margin_right": 3,
        "margin_bottom": 2.5,
        "first_line_indent_cm": 0,
        "paragraph_spacing_after": 6,
        "paragraph_spacing_before": 6
      }
    },
    "inputs": [
      {
        "id_input": "i001",
        "input_class": null,
        "label_input": "Número do Voto",
        "input_properties": ""
      }
    ],
    "variaveis": [],
    "logicas": [],
    "elementos": [
      {
        "id_elemento": "e001",
        "conteudo_elemento": "VOTO N°\\t\\t: "
      }
    ]
  },
  {
    "id": "p002",
    "estilo": {
      "type": "Specific",
      "formatting": {
        "alignment": "Justificado",
        "font_size": 14,
        "font_color": "#000000",
        "margin_top": 2.5,
        "font_family": "Times New Roman",
        "left_indent": 0,
        "margin_left": 3,
        "line_spacing": 1.5,
```

```
        "margin_right": 3,
        "margin_bottom": 2.5,
        "first_line_indent_cm": 1.25,
        "paragraph_spacing_after": 6,
        "paragraph_spacing_before": 6
      }
    },
    "inputs": [],
    "variaveis": [
      {
        "id_var": "v001",
        "variavel_class": null,
        "label_var": "Resultado do Julgamento",
        "variavel_properties": "procedente;improcedente"
      }
    ],
    "logicas": [
      {
        "id_logic": "l001",
        "logica_class": null,
        "paragraph_id": "p002",
        "label_logic": "Justificativa de Honorários",
        "logica_options": "Incluir;Não incluir",
        "logica_conteudo": "Os honorários advocatícios são majorados nos
termos do art. 85, § 11, do CPC.",
        "logica_properties": "Simples",
        "logica_descricao_detalhada": "Incluir o parágrafo sobre a
majoração de honorários se a apelação for desprovida."
      }
    ],
    "elementos": [
      {
        "id_elemento": "e002",
        "conteudo_elemento": "A r. sentença apelada julgou a ação "
      },
      {
        "id_elemento": "e003",
        "conteudo_elemento": ". "
      }
    ]
  }
]
```


Fonte: Elaborado pelo autor (2025)