SmartResume Generator — Model Picker

1. Overview

SmartResume Generator — Model Picker is a web-based Streamlit application that leverages Google's Generative AI models to automatically generate professional resumes. Users can provide personal, educational, and career details, select an AI model for generation, and download the output in multiple formats (TXT, DOCX, PDF, LaTeX PDF).

Key Features:

- AI-powered resume generation using Google Generative AI.
- Model picker to choose preferred AI model.
- Multiple resume formats: plain text, DOCX, PDF, and premium LaTeX PDF.
- Customizable resume style and industry focus.
- Automatic formatting of headings, bullets, and spacing.
- Fallback mechanisms for API key retrieval and model listing.

2. Architecture

2.1 Components

Component	**Description**
app.py	Main Streamlit interface for user input, model selection, and resume generation.
resume_generator.py	Backend logic for configuring API, listing models, picking default model, generating resume text, and prompt building.
premium_pdf.py	Optional premium PDF generator using ReportLab for justified text, bold headings, and clean formatting.
resume_latex.py	Optional LaTeX-based PDF generator using pylatex for high-quality resumes with Times font and proper bullet formatting.

2.2 Data Flow

1. User enters personal information, job title, skills, experience, projects, and education.
2. User selects a resume style (professional, ats, creative) and industry.
3. App builds a detailed AI prompt using resume_generator.build_resume_prompt.
4. Selected AI model generates resume text via resume_generator.generate_with_model.
5. Resume text is cleaned and formatted.

6. Users can preview the resume and download it in TXT, DOCX, PDF, or LaTeX PDF formats.

3. Setup Instructions

3.1 Requirements

- Python >= 3.10
- Streamlit
- google-generativeai
- docx (python-docx)
- fpdf or fpdf2
- reportlab (optional, for premium PDF)
- pylatex and LaTeX installation (optional, for LaTeX PDF)
- requests (for REST API fallback)

3.2 Environment Variables

The app requires a **GEMINI_API_KEY** for accessing Google Generative AI:
export GEMINI_API_KEY="your_api_key_here"

Or add it to .streamlit/secrets.toml:
[GEMINI_API_KEY]
value = "your_api_key_here"

4. User Guide

4.1 Input Fields

Section	Input	Description
Personal Information	Full Name, Email, Phone, LinkedIn	Basic contact information. LinkedIn optional.
Job Title	Text input	Desired job role.
Details	Summary	Brief professional overview.
Details	Skills	Comma-separated list of skills.
Details	Experience	Roles and achievements in bullets or paragraphs.
Details	Projects	Optional projects with description and outcomes.

Details	Education	Optional academic qualifications.
Resume Style	Dropdown	Options: professional, ats, creative.
Industry	Dropdown	Options: General, Software, AI/ML, Finance, Marketing, Design, Other.

4.2 Model Selection

- Sidebar shows all available models retrieved via SDK or REST.
- Users can select a model for text generation.
- Default model is auto-picked using resume_generator.pick_text_model.

4.3 Generating Resume

1. Fill in the required fields (Name, Job Title at minimum).
2. Click **Generate Resume**.
3. App will display a preview in the interface.

4.4 Download Options

- **TXT** — Plain text version.
- **DOCX** — Word document with bold headings and paragraphs.
- **PDF** — Standard PDF via FPDF with justified text.
- **Premium LaTeX PDF** — High-quality PDF using LaTeX with proper fonts and bullets (requires LaTeX installed).

5. Developer Guide

5.1 API Configuration

```
import resume_generator as rg
rg.configure_api("YOUR_API_KEY")
rg.set_model_name("model_name_here")
```

5.2 Resume Generation Programmatically

```
user = {
    "name": "Jane Doe",
    "job_title": "Machine Learning Engineer",
    "summary": "Experienced ML engineer...",
    "skills": "Python, TensorFlow, SQL",
    "experience": "Role at Company: Achievements",
    "projects": "Project X: description",
    "education": "MSc Computer Science",
    "email": "jane@example.com",
    "phone": "+1-555-555-5555",
    "linkedin": "https://linkedin.com/in/janedoe"
}

prompt = rg.build_resume_prompt(user, style="professional", industry="AI/ML")
resume_text = rg.generate_with_model(prompt)
resume_text = rg.clean_resume_text(resume_text)
```

5.3 Generating Premium PDF

```
from premium_pdf import generate_premium_pdf
pdf_bytes = generate_premium_pdf(user, resume_text)
with open("resume.pdf", "wb") as f:
    f.write(pdf_bytes)
```

5.4 Generating LaTeX PDF

```
from resume_latex import generate_latex_resume
pdf_bytes = generate_latex_resume(user, resume_text)
with open("resume_latex.pdf", "wb") as f:
    f.write(pdf_bytes)
```

5.5 Code

app.py

```
import streamlit as st
import os
import io
from datetime import datetime
from docx import Document
from fpdf import FPDF
import resume_generator as rg
```

```
from resume_latex import generate_latex_resume
# -----------------------
# Page config
# -----------------------
st.set_page_config(page_title="SmartResume Generator", layout="centered")
st.title("📄 SmartResume Generator — Model Picker")
# -----------------------
# Load API key
# -----------------------
API_KEY = None
try:
    API_KEY = st.secrets["GEMINI_API_KEY"]
except Exception:
    API_KEY = os.environ.get("GEMINI_API_KEY")
if not API_KEY:
    st.error("GEMINI_API_KEY missing. Add it to .streamlit/secrets.toml or set it
as environment variable.")
    st.stop()
# -----------------------
# Configure SDK
# -----------------------
try:
    rg.configure_api(API_KEY)
except Exception as e:
    st.error(f"API configuration failed: {e}")
    st.stop()
# -----------------------
# List available models
# -----------------------
st.sidebar.header("Model selection")
models = []
try:
    raw_models = rg.list_models_via_sdk()
    if isinstance(raw_models, dict) and "models" in raw_models:
        models_raw = raw_models["models"]
    else:
        models_raw = raw_models
    models = models_raw if isinstance(models_raw, list) else list(models_raw)
except Exception:
    st.sidebar.info("SDK list_models() not available — falling back to REST.")
    try:
        models = rg.list_models_via_rest(API_KEY)
    except Exception as ex:
        st.sidebar.error(f"Failed to list models via REST: {ex}")
        st.stop()
# Build readable short names
short_names = []
for m in models:
    try:
        if isinstance(m, dict):
            full = m.get("name") or m.get("model") or m.get("displayName") or
str(m)
        else:
            full = getattr(m, "name", None) or getattr(m, "model", None) or
str(m)
    except Exception:
        full = str(m)
    short = full.split("/")[-1] if "/" in full else full
    short_names.append(short)
# Pick default model
picked = rg.pick_text_model(models) if models else None
default_index = short_names.index(picked) if (picked and picked in short_names)
else 0
if short_names:
    model_choice = st.sidebar.selectbox("Choose model for generation",
short_names, index=default_index)
else:
    st.sidebar.warning("No models available for this key.")
    model_choice = None
# Map to full model ID
selected_full = None
for m in models:
    try:
        full = m.get("name") if isinstance(m, dict) else getattr(m, "name", None)
        if full and full.endswith(model_choice):
            selected_full = full
            break
    except Exception:
        continue
if not selected_full:
    selected_full = model_choice
rg.set_model_name(selected_full)
st.sidebar.write("Using model:", selected_full or "—")
# ---------------------------
# UI Inputs
# ---------------------------
st.subheader("Personal Information")
col1, col2 = st.columns(2)
with col1:
    name = st.text_input("Full Name", placeholder="Jane Doe")
    email = st.text_input("Email", placeholder="jane.doe@example.com")
with col2:
    phone = st.text_input("Phone", placeholder="+1-555-555-5555")
    linkedin = st.text_input("LinkedIn (optional)",
placeholder="https://linkedin.com/in/username")
job_title = st.text_input("Job Title", placeholder="Machine Learning Engineer")
st.subheader("Details")
summary = st.text_area("Professional Summary", height=120, placeholder="Brief
overview of your career goals or expertise.")
skills = st.text_area("Skills (comma-separated)", height=80, placeholder="Python,
Machine Learning, SQL, ...")
experience = st.text_area("Experience (give bullets or paragraphs)", height=200,
placeholder="Role at Company (Year - Year):\n- Achievement 1\n- Achievement 2")
projects = st.text_area("Projects (optional)", height=120, placeholder="Project
name: description, outcomes, tools used")
education = st.text_area("Education (optional)", height=80, placeholder="Degree,
University, Year")
style = st.selectbox("Resume style", ["professional", "ats", "creative"])
```

```
industry = st.selectbox("Industry", ["General", "Software", "AI/ML", "Finance",
"Marketing", "Design", "Other"])
# ---------------------------
# Helper formatting functions
# ---------------------------
def format_resume_text(text: str) -> str:
    """Normalize bullets, headings, and blank lines."""
    if not text:
        return ""
    lines = []
    for raw in text.splitlines():
        s = raw.strip()
        if not s:
            lines.append("")
            continue
        if s.startswith(("•", "-", "*")):
            content = s[1:].strip()
            lines.append("• " + content)
            continue
        if s.isupper() or s.endswith(":"):
            lines.append(s.upper())
            continue
        lines.append(s)
    return "\n".join(lines).strip()
# ---------------------------
# Generate / Download block
# ---------------------------
if st.button("Generate Resume"):
    if not name or not job_title:
        st.warning("Please enter at least Name and Job Title.")
        st.stop()
    user = {
        "name": name,
        "job_title": job_title,
        "summary": summary,
        "skills": skills,
        "experience": experience,
        "projects": projects,
        "education": education,
        "email": email,
        "phone": phone,
        "linkedin": linkedin
    }
    prompt = rg.build_resume_prompt(user, style=style, industry=industry)
    with st.spinner("Generating resume..."):
        try:
            raw = rg.generate_with_model(prompt)
            resume_text = rg.clean_resume_text(raw)
            resume_text = format_resume_text(resume_text)
        except Exception as e:
            st.error(f"Failed to generate resume: {e}")
            st.stop()
    st.success("Generated!")
    st.markdown("### Preview")
    st.code(resume_text, language="text")
    # ---------------------------
    # Download helpers
    # ---------------------------
    def get_txt_bytes(t: str) -> bytes:
        return t.encode("utf-8")
    def get_docx_bytes(t: str) -> bytes:
        doc = Document()
        for line in t.splitlines():
            if line.strip() == "":
                doc.add_paragraph()
            elif line.isupper():
                p = doc.add_paragraph()
                run = p.add_run(line)
                run.bold = True
                p.paragraph_format.alignment = 3  # justify
            else:
                p = doc.add_paragraph(line)
```

```
                p.paragraph_format.alignment = 3  # justify
        mem = io.BytesIO()
        doc.save(mem)
        mem.seek(0)
        return mem.read()
    def get_pdf_bytes(t: str) -> bytes:
        fixes = {"•": "-", "–": "-", "—": "-", "’": "'", "‘": "'", "“": '"', "”":
'"', "·": "-"}
        for k, v in fixes.items():
            t = t.replace(k, v)
        pdf = FPDF()
        pdf.set_auto_page_break(auto=True, margin=12)
        pdf.add_page()
        pdf.set_font("Times", size=12)
        for line in t.splitlines():
            if line.isupper():
                pdf.set_font("Times", style="B", size=12)
                pdf.multi_cell(0, 7, line)
                pdf.set_font("Times", size=12)
            else:
                pdf.multi_cell(0, 7, line)
        return pdf.output(dest="S").encode("latin-1")
    # Filenames
    now = datetime.now().strftime("%Y%m%d_%H%M%S")
    safe = name.replace(" ", "_")
    col1, col2, col3 = st.columns(3)
    with col1:
        st.download_button("📄 Download TXT", get_txt_bytes(resume_text),
f"{safe}_{now}.txt", mime="text/plain")
    with col2:
        st.download_button("📄 Download DOCX", get_docx_bytes(resume_text),
f"{safe}_{now}.docx",
                           mime="application/vnd.openxmlformats-
officedocument.wordprocessingml.document")
    with col3:
        st.download_button("📄 Download PDF", get_pdf_bytes(resume_text),
f"{safe}_{now}.pdf", mime="application/pdf")
    # Premium LaTeX PDF
    try:
        latex_pdf = generate_latex_resume(user, resume_text)
        st.download_button("⭐ Download Premium LaTeX PDF", latex_pdf,
f"{safe}_{now}_Premium.pdf", mime="application/pdf")
    except Exception as e:
        st.error(f"Premium LaTeX PDF generation failed: {e}\nEnsure LaTeX
(pdflatex) is installed and pylatex is available.")
```

resume_generator.py

```
import os
import json
import google.generativeai as genai
# -----------------------
# Global model name
# -----------------------
MODEL_NAME = None
# -----------------------
# Configure API
# -----------------------
def configure_api(api_key: str = None):
    """
    Configure the Google Generative AI SDK with your API key.
    """
    key = api_key or os.environ.get("GEMINI_API_KEY")
    if not key:
        raise ValueError("GEMINI_API_KEY missing. Set it in secrets or
environment.")
    genai.configure(api_key=key)
# -----------------------
# List models via SDK
# -----------------------
def list_models_via_sdk(page_size: int = 100):
    """
    List models via the installed Google GenAI SDK.
    Returns a list of model dicts or raises if unsupported.
    """
    if hasattr(genai, "list_models"):
        return genai.list_models(page_size=page_size)
    if hasattr(genai, "models") and hasattr(genai.models, "list"):
        return genai.models.list(page_size=page_size)
    raise RuntimeError("list_models not available in this GenAI SDK")
# -----------------------
# List models via REST
# -----------------------
def list_models_via_rest(api_key: str):
    """
    Fallback: call the REST endpoint to list accessible models.
    Returns parsed JSON models list.
    """
    import requests
    url =
f"https://generativelanguage.googleapis.com/v1beta/models?key={api_key}"
    resp = requests.get(url, timeout=30)
    resp.raise_for_status()
    data = resp.json()
    return data.get("models", [])
# -----------------------
# Pick default text model
# -----------------------
def pick_text_model(models):
    """
    Heuristic selection of a text generation model.
    Returns the model name string or None.
    """
    candidates = []
    seen = set()
    for m in models:
        mdict = None
        try:
            mdict = dict(m)
        except Exception:
            try:
                mdict = m.__dict__
            except Exception:
                mdict = m
        name = mdict.get("name") or mdict.get("model") or mdict.get("modelName")
or mdict.get("id")
        if not name:
            name = mdict.get("model") or mdict.get("displayName")
        if not name:
            continue
        short = name.split("/")[-1] if "/" in name else name
        # Check supported methods / capabilities
        capabilities = mdict.get("supported_methods") or
mdict.get("supportedMethods") or mdict.get("capabilities") or
mdict.get("supportedTasks")
        caps_text = json.dumps(capabilities) if capabilities is not None else ""
        if "generate" in caps_text.lower() or "generatecontent" in
caps_text.lower() or "text" in caps_text.lower() or "chat" in caps_text.lower():
            if short not in seen:
                seen.add(short)
                candidates.append(short)
    # Fallback: any gemini-* model
    if not candidates:
        for m in models:
            mdict = None
            try:
                mdict = dict(m)
            except Exception:
                try:
                    mdict = m.__dict__
                except Exception:
                    mdict = m
            name = mdict.get("name") or mdict.get("model") or
mdict.get("displayName")
            if not name:
                continue
            short = name.split("/")[-1] if "/" in name else name
            if isinstance(short, str) and ("gemini" in short.lower() or "text" in
short.lower()):
                if short not in seen:
                    seen.add(short)
                    candidates.append(short)
    return candidates[0] if candidates else None
# -----------------------
# Set global model
# -----------------------
def set_model_name(name: str):
    """
    Set the global MODEL_NAME. Must be called after selecting a model.
    """
    global MODEL_NAME
    MODEL_NAME = name
# -----------------------
# Generate resume content
# -----------------------
def generate_with_model(prompt: str) -> str:
    """
    Generate content using the current MODEL_NAME.
    Must call configure_api() first and set MODEL_NAME.
    """
    if MODEL_NAME is None:
        raise RuntimeError("MODEL_NAME is not set. Call set_model_name() with a
valid model id.")
    model = genai.GenerativeModel(MODEL_NAME)
    try:
        response = model.generate_content(prompt)
    except TypeError:
        response = model.generate_content(contents=prompt)
    except Exception as e:
        raise RuntimeError(f"Model generation error: {e}")
    # Defensive extraction
    if hasattr(response, "text") and isinstance(response.text, str):
        return response.text.strip()
    try:
        cand = getattr(response, "candidates", None)
        if cand and len(cand) > 0:
            first = cand[0]
            if hasattr(first, "text"):
                return first.text.strip()
            cont = getattr(first, "content", None)
            if cont and len(cont) > 0:
                part = cont[0]
                return getattr(part, "text", None) or (part.get("text") if
isinstance(part, dict) else None) or str(part)
    except Exception:
        pass
    return str(response).strip()
# -----------------------
# Clean placeholders
# -----------------------
def clean_resume_text(text: str) -> str:
    if not text:
        return text
    return (text.replace("[Add Email Address]", "[Your Email Address]")
                .replace("[Add Phone Number]", "[Your Phone Number]")
                .replace("[Add LinkedIn Profile URL (optional)]", "[Your LinkedIn
URL (optional)]")
                .strip())
# -----------------------
# Build resume prompt
# -----------------------
def build_resume_prompt(user, style="professional", industry="General"):
    """
    Build a detailed prompt for AI resume generation.
    """
    return f"""
You are an expert resume writer skilled in ATS-friendly formatting.
```

```
Write a polished, well-structured resume based on the details below.
--- Personal Details ---
Name: {user.get("name", "")}
Job Title: {user.get("job_title", "")}
Email: {user.get("email", "")}
Phone: {user.get("phone", "")}
LinkedIn: {user.get("linkedin", "")}
--- Professional Summary ---
{user.get("summary", "")}
--- Skills ---
{user.get("skills", "")}
--- Experience ---
{user.get("experience", "")}
--- Projects ---
{user.get("projects", "")}
--- Education ---
{user.get("education", "")}
--- Requirements ---
• The style should be: {style}
• Industry focus: {industry}
• Format in neat bullet-points.
• Include strong action verbs and measurable achievements when possible.
• Do NOT add placeholders like [Email Here] or [Phone Here].
• Do NOT include content unrelated to resumes.
• Start with the candidate's name as the heading.
• Final output should be ready-to-use resume text.
Generate only resume content. No explanations.
"""
```

6. Formatting Rules

The SmartResume Generator applies consistent formatting rules to ensure resumes are professional, ATS-friendly, and readable across multiple output formats (TXT, DOCX, PDF, LaTeX PDF). Below is a detailed guide:

6.1 Headings and Section Titles

- Lines that are **fully uppercase** (e.g., EXPERIENCE) or **ending with a colon** (e.g., Projects:) are treated as **section headings**.
- Headings are rendered as:
    - **Bold** in DOCX, PDF, and LaTeX PDF.
    - Larger font size in LaTeX PDF and premium PDFs.
    - Separated from content by a horizontal line or spacing in PDFs and LaTeX.
- In LaTeX PDF:
    - Headings include a horizontal rule (\noindent\rule) for visual separation.
    - Extra vertical spacing is added before and after the heading for readability.
- Headings are left-aligned in all formats.

6.2 Bullets

- Lines starting with •, -, or * are recognized as **bullet points**.
- Bullets are rendered as:
    - • in DOCX and LaTeX.
    - - in standard FPDF PDF (replacing unsupported characters for clean output).
- Indentation:
    - DOCX: bullets slightly indented.
    - LaTeX PDF: left margin controlled via enumitem (left=0.4cm by default).
- Multiple consecutive bullets are grouped into the same list (itemize) in LaTeX.
- Bullet spacing:
    - Line spacing between bullets is small to maintain compact readability.
    - Additional spacing is inserted between sections.

6.3 Normal Text / Paragraphs

- Non-heading and non-bullet lines are treated as **paragraphs**.
- Text alignment:
    - Standard PDFs and LaTeX PDFs are **justified**.
    - DOCX paragraphs are left-aligned with justified appearance.
- Line breaks:
    - TXT: preserves user input line breaks.
    - DOCX: blank lines create paragraph separation.
    - PDF / LaTeX PDF: blank lines create vertical spacing between paragraphs.

- Paragraph spacing is consistent to improve readability without wasting space.

6.4 Fonts and Styles

- **DOCX**:
  - Headings: bold (run.bold = True).
  - Body: standard Times or system default.
- **PDF (FPDF)**:
  - Body: Times font.
  - Headings: bold Times font.
- **Premium LaTeX PDF**:
  - Uses Times (\usepackage{times}), Helvetica (\usepackage{helvet}), and Courier (\usepackage{courier}) for optional typewriter sections.
  - Headings are bold and optionally larger (\Huge for the name, bold for sections).
  - Bullets and normal text respect spacing defined by enumitem and LaTeX paragraph rules.

6.5 Contact Information

- Displayed prominently at the top under the candidate name.
- Format:
  - <Email> | <Phone> | <LinkedIn URL>
- In LaTeX PDF, email and LinkedIn can be **clickable links** using \href{mailto:...} and \href{...}.
- In DOCX and standard PDFs, plain text is used to ensure compatibility.

6.6 Special Characters

- Certain special characters are normalized for compatibility:
  - Replacements in PDF: “” → ", ’ → ', • → -.
  - LaTeX escape: _, &, %, #, {, } are automatically escaped.
- Commas, colons, and dashes are preserved in resume text.

6.7 Placeholder Handling

- Default placeholders from AI output are automatically replaced:
  - [Add Email Address] → [Your Email Address]
  - [Add Phone Number] → [Your Phone Number]

    - [Add LinkedIn Profile URL (optional)] → [Your LinkedIn URL (optional)]
- No other placeholders or prompts are included in the final resume.

6.8 Page Layout (PDF / LaTeX)

- Margins:
    - Standard PDF: 12 units in FPDF (approx. 0.5 inch).
    - LaTeX PDF: 1-inch margin using geometry.
- Multi-page support:
    - Standard PDF: handles simple multi-page resumes.
    - LaTeX PDF: automatically breaks pages and maintains formatting.

6.9 Style Customization

- Users can choose style:
    - **Professional**: clean, standard bullet points, concise language.
    - **ATS**: optimized for Applicant Tracking Systems, minimal formatting.
    - **Creative**: more expressive language, optional headings or design elements.
- Industry selection may influence phrasing and emphasis (e.g., software, AI/ML, finance).

7. Error Handling

The SmartResume Generator implements multiple layers of **error handling** to ensure a smooth user experience and provide informative messages when issues occur.

7.1 Missing API Key

- **Cause:** The app requires a GEMINI_API_KEY to access Google Generative AI models.
- **Detection:** At startup, the app checks:
    - .streamlit/secrets.toml
    - Environment variables (GEMINI_API_KEY)
- **Behavior:** If the key is missing:
    - Streamlit displays a clear error message:
    - "GEMINI_API_KEY missing. Add it to .streamlit/secrets.toml or set it as environment variable."
    - App execution stops (st.stop()), preventing further errors.

7.2 Model Listing Failure

- **Cause:** SDK version may not support list_models() or network issues prevent REST access.
- **Behavior:**
    - The app attempts to list models via SDK.
    - If SDK fails, it falls back to **REST API**.
    - If both fail, Streamlit shows an error:
    - "Failed to list models via REST: <exception message>"
    - Users are prevented from generating resumes until a model is available.

7.3 Model Generation Failure

- **Cause:** Could be due to:
    - Invalid model selection
    - API request errors (network, timeout)
    - SDK inconsistencies
- **Behavior:**
    - The exception is caught and displayed:
    - "Failed to generate resume: <exception message>"
    - Resume generation is aborted gracefully without crashing the app.

7.4 LaTeX PDF Generation Failure

- **Cause:** Missing LaTeX installation (pdflatex) or pylatex not installed.
- **Behavior:**
    - The app captures exceptions during PDF compilation.
    - Displays error message with troubleshooting hints:
    - "Premium LaTeX PDF generation failed: <exception> Ensure LaTeX (pdflatex) is installed and pylatex is available."

7.5 Other Runtime Errors

- Input validation:
    - Minimum required fields: Name and Job Title.
    - If missing, the app warns:
    - "Please enter at least Name and Job Title."

- Download errors:
    - File generation failures are caught and logged.
    - Users are notified via Streamlit message.

8. Known Limitations

While the SmartResume Generator is robust, there are some **limitations to be aware of**:

8.1 PDF Output Limitations

- Standard FPDF PDF:
    - Multi-page resumes may not break elegantly if content exceeds one page.
    - Bullet wrapping can sometimes be inconsistent for long text.
- Premium LaTeX PDF:
    - Multi-page resumes handled automatically with proper section breaks.
    - Requires a working LaTeX installation; otherwise generation fails.

8.2 Character Rendering

- FPDF PDF:
    - Some unusual Unicode characters (emojis, special symbols, non-Latin scripts) may not render correctly.
    - Recommended to stick with standard ASCII / UTF-8 characters.
- LaTeX PDF:
    - Better Unicode support but may require additional LaTeX packages (inputenc, fontenc) for extended scripts.

8.3 Model Selection Heuristics

- The app uses heuristics (pick_text_model) to select the default text-generation model.
- In rare cases:
    - An outdated or incompatible model may be selected.
    - Users can manually select another model from the sidebar to resolve this.

8.4 Input Size and Complexity

- Very large inputs (long experience sections or numerous projects) may increase AI response time.

- Extremely verbose prompts could lead to truncated or incomplete resume output.

8.5 LaTeX Dependency

- LaTeX PDF generation is optional.
- Without pdflatex installed, only TXT, DOCX, or standard PDF outputs are available.

9. Future Enhancements

The SmartResume Generator is designed to evolve. Planned enhancements include:

9.1 Multi-Language Support

- Generate resumes in languages other than English.
- Include language-specific formatting rules and ATS considerations.
- Example: Spanish, French, German resumes with correct grammar and accents.

9.2 Section Ordering Control

- Allow users to customize the **order of sections**:
    - Skills first, then Experience
    - Education first for fresh graduates
    - Projects first for technical portfolios
- Improves flexibility for different industries and career stages.

9.3 Resume Tone Control

- Offer tone selection in AI prompt:
    - Formal / professional
    - Casual / creative
    - Executive / senior-level
- Adjusts language, action verbs, and phrasing accordingly.

9.4 Enhanced PDF Formatting

- Improve bullet indentation, spacing, and line wrapping in standard PDFs.
- Auto-detect long paragraphs and insert appropriate spacing.
- Support multiple fonts for headers, body, and skills sections.

9.5 Automatic Hyperlinks

- Add clickable links in DOCX and PDF for:
    - LinkedIn profile
    - Email address (mailto:)
    - Portfolio or personal website
- Improves interactivity for digital resumes.

9.6 AI Model Management

- Allow advanced users to:
    - Choose between multiple generations for comparison.
    - Save preferred models for different resume styles.
    - Fine-tune model prompts for domain-specific resumes (e.g., AI, Finance, Design).

9.7 Analytics & Feedback

- Collect user feedback on generated resumes.
- Improve prompt templates based on success metrics (ATS passes, recruiter feedback).
- Track formatting issues to improve PDF and LaTeX output quality.

10. Conclusion and Future Scope

10.1 Conclusion

The **SmartResume Generator — Model Picker** demonstrates a practical application of AI-powered content generation for resume creation. By integrating Google Generative AI with a user-friendly Streamlit interface, the system allows users to generate professional, well-formatted resumes tailored to different industries and styles. Key achievements of the system include:

- Dynamic **AI model selection** for text generation.
- **Multi-format resume outputs**, including TXT, DOCX, PDF, and LaTeX PDF.
- **Automated formatting**, including headings, bullets, spacing, and placeholder cleanup.
- Error handling mechanisms ensuring robustness during model selection, API communication, and file generation.

The tool effectively reduces the time and effort required for resume creation while maintaining quality and ATS compliance. It provides both novice users and professionals with a convenient platform for generating ready-to-use resumes.

10.2 Future Scope

The SmartResume Generator has several avenues for enhancement to increase utility and sophistication:

1. **Multi-language support:** Generate resumes in multiple languages with industry-appropriate phrasing.
2. **Customizable section ordering:** Allow users to rearrange sections (Skills, Experience, Education, Projects) according to career level and domain.
3. **Resume tone and style control:** Introduce options for formal, casual, creative, or executive tones.
4. **Enhanced PDF formatting:** Improved bullet spacing, multi-page handling, font customization, and clickable hyperlinks in DOCX/PDF.
5. **AI model fine-tuning:** Enable domain-specific prompts or custom model selection for specialized careers (AI/ML, Finance, Design).
6. **Analytics and feedback loop:** Collect user feedback to improve prompt templates and resume quality.
7. **Integration with job platforms:** Provide seamless resume submission options to LinkedIn, Indeed, or company portals.

These enhancements aim to make the SmartResume Generator more adaptive, intelligent, and interactive, bridging the gap between AI text generation and real-world career application tools.

11. References

The following references are cited in IEEE style:

[1] Google Cloud, “Generative AI Overview,” Google Cloud, 2024. [Online]. Available: https://cloud.google.com/generative-ai.

[2] Streamlit Inc., “Streamlit Documentation,” 2024. [Online]. Available: https://docs.streamlit.io/.

[3] Python Software Foundation, “Python 3.10 Documentation,” 2024. [Online]. Available:

https://docs.python.org/3/.

[4] M. Jelte, *PyLaTeX: A Python library for LaTeX document generation*, 2023. [Online]. Available: https://jeltef.github.io/PyLaTeX/current/.

[5] ReportLab, "ReportLab User Guide," 2024. [Online]. Available: https://www.reportlab.com/docs/.

[6] S. S. Singh and R. K. Verma, "Automated Resume Generation Using AI Techniques," *International Journal of Computer Applications*, vol. 182, no. 25, pp. 1–7, 2021.

[7] FPDF2 Project, "FPDF2 Documentation," 2024. [Online]. Available: https://pyfpdf.github.io/fpdf2/.

[8] python-docx Contributors, "python-docx Documentation," 2024. [Online]. Available: https://python-docx.readthedocs.io/.

[9] Google, "Gemini Models: Google Generative AI Models Reference," 2024. [Online]. Available: https://developers.generativeai.google/models.