

SmartSlide AI: An Intelligent Chatbot-Driven System for Automated PowerPoint and PDF Presentation Generation

Vijayalaxmi Angadi¹

Department of Master of Computer Applications, GM University, Davanagere, Karnataka, India

USN: P24C01CA071 | Email: vijayalaxmisa123@gmail.com

Varun K S²

Assistant Professor (Project Guide), Faculty of Computing and IT, GM University, Davanagere, Karnataka, India

Abstract— Creating a well-structured slide presentation is a time-consuming task that combines research, content writing, visual design, and file-format handling. This paper presents SmartSlide AI, a conversational, chatbot-driven web platform that converts a natural-language request into a fully editable presentation, exported simultaneously as an OOXML PowerPoint (.pptx) file and a print-ready PDF. The system decomposes presentation authoring into a five-stage pipeline — requirement extraction, outline generation, slide-content generation, layout/theme rendering, and dual-format export — coordinated by a REST API backend and an isolated Node.js presentation engine built on pptxgenjs. A provider-abstraction layer allows the AI engine to call OpenAI, Google Gemini, or Anthropic Claude interchangeably, while a deterministic “Demo AI Mode” guarantees the full pipeline remains demonstrable with no external API dependency and no fabricated citations. Five presentation themes and ten distinct slide layouts are implemented as native, schema-validated PowerPoint objects rather than rendered images, so every exported

slide remains editable in Microsoft PowerPoint and LibreOffice Impress. The prototype was implemented end-to-end — authentication, project dashboard, chat-based requirement gathering, an in-browser slide editor, and export — and validated with an automated test suite (16 test cases, 100% pass rate) and empirical export-latency measurements. Results show that a complete 10-slide, thematically consistent, editable presentation can be produced and exported in under two seconds of computation (excluding network-bound LLM latency), demonstrating that a chatbot interface can meaningfully compress the presentation-authoring workflow while preserving full editability and design consistency.

Index Terms— Conversational AI, Chatbot, Natural Language Generation, Automated Presentation Generation, PPTX, PowerPoint Automation, Large Language Models, Retrieval-Augmented Generation, Human-Computer Interaction, Web Application Architecture.

I. INTRODUCTION

Slide-based presentations remain the dominant medium for academic seminars, research defenses, corporate reporting, and technical training. Yet the process of producing one is disproportionately manual: a presenter must research the topic, decide on a narrative structure, write concise slide-level content, choose a visual theme and per-slide layout, and finally export a file compatible with the audience's software. Each of these sub-tasks draws on a different skill — subject-matter research, instructional writing, and graphic design — and completing all of them for even a modest ten-slide deck commonly takes several hours.

Recent advances in Large Language Models (LLMs) such as GPT-4 [5], Gemini [6], and Claude [7] have made it practical to generate coherent, context-aware text from a short natural-language prompt. Several commercial tools — Gamma, Canva's AI design assistant, and Microsoft Copilot for PowerPoint — already apply this capability to slide creation. However, as detailed in Section II, each addresses

only part of the problem: some generate attractive slides but offer little conversational refinement; others provide strong design templates but comparatively shallow content generation; and platform-bound tools require an existing subscription to a specific office suite. None combine a conversational chatbot front-end, an editable in-browser workspace, and a guaranteed dual-format (PPTX + PDF) export inside a single, self-contained pipeline.

This paper presents SmartSlide AI, a full-stack web application designed to close that gap. A user describes the desired presentation in plain language; a chatbot module extracts structured requirements (topic, audience, slide count, tone, and style) and asks only for whatever information is still missing. The system then produces an editable outline, generates layout-aware content for every slide, renders the deck through a native PowerPoint-object generation engine, and converts it to PDF — all while keeping every stage editable through a purpose-built browser workspace. The central engineering contribution is not any single AI call, but the pipeline architecture that makes AI-generated content reliably convertible into

schema-valid, genuinely editable office documents, with a deterministic fallback mode that keeps the system fully operational even without a live LLM subscription.

The remainder of this paper is organized as follows. Section II surveys related tools and underlying NLG/RAG techniques. Section III states the problem and Section IV the objectives and scope. Section V describes the proposed system, its architecture, and its evolution from the originally proposed FastAPI/React design to the implemented prototype. Section VI details the AI chatbot and

II. RELATED WORK

Automated document and presentation generation sits at the intersection of Natural Language Generation (NLG) and template-based document engineering. The introduction of the Transformer architecture [1] and its scaled descendants — BERT [3] for bidirectional language understanding and GPT-style decoder-only models [2] for open-ended generation — gave rise to LLMs capable of producing coherent multi-paragraph text from a short prompt, which is the technical precondition for any chatbot-driven authoring tool.

A parallel line of work addresses a core weakness of purely generative LLMs: because their knowledge is frozen at training time and stored implicitly in model weights, they can produce plausible but incorrect “hallucinated” content when a query needs organization-specific or up-to-date facts. Retrieval-Augmented Generation (RAG) [4] addresses this by retrieving relevant passages from an external corpus via semantic (embedding-based) search and conditioning the language model’s generation on that retrieved context. This principle directly informed the design of SmartSlide AI’s citation-handling policy (Section VI-D): the system never fabricates a source, and any References slide is populated only from citations the content-generation stage explicitly attaches to a slide, mirroring the retrieval-before-generation discipline RAG applies to open-domain question answering.

Among commercial slide-generation tools, Gamma converts a short prompt into a visually polished deck but offers limited support for iterative, multi-turn conversational refinement of requirements before generation begins. Canva’s AI-assisted design tools provide an extensive, professionally designed template library and strong drag-and-drop manual editing, but their automatic content generation is comparatively basic and secondary to the design tooling. Microsoft Copilot integrates deeply with PowerPoint and the Microsoft 365 ecosystem, producing high-quality output, but requires an existing M365 subscription and operates inside the desktop application rather than as an independent, chatbot-first web workflow. Table I summarizes this comparison.

requirement-extraction module — the paper’s core AI-chatbot contribution. Section VII describes the generation pipeline/methodology, Section VIII the presentation engine (themes, layouts, export), and Section IX the editor and dashboard. Section X lists the technology stack and API surface, Section XI the security and reliability measures, Section XII the testing methodology, and Section XIII the performance results. Sections XIV–XVI discuss advantages, limitations, and future work, and Section XVII concludes the paper.

TABLE I. COMPARISON OF SMARTSLIDE AI WITH EXISTING TOOLS

Tool	Chat Refine.	Web Editor	PPTX+PDF	No Subscr.
Gamma	Limited	Partial	Partial	No
Canva AI	Basic	Yes (design-first)	PDF only (design)	No
MS Copilot	Yes (in-app)	No (desktop only)	PPTX only	No (M365 req.)
SmartSlide AI	Yes (multi-turn)	Yes (browser)	Yes (both, always)	Yes (Demo Mode)

Beyond commercial tools, the author’s prior applied work on a Retrieval-Augmented Generation-based document question-answering system — built during an industry internship on Python, LangChain, FAISS, and Hugging Face sentence embeddings — directly shaped the conversational-design philosophy used here: a natural-language front-end should extract structured intent before any generative step runs, and a system should be explicit and conservative about what it does and does not know, rather than silently guessing. Section XVI discusses how a similar retrieval layer could extend SmartSlide AI to ground academic slide content in uploaded reference material.

A further consideration, distinct from tool comparison, is that automatically generated instructional content should still respect basic pedagogical structuring even when no human instructional designer reviews it. The outline stage’s fixed section vocabulary (Introduction, Background, Key Concepts, Applications, Advantages, Challenges, Future Scope, Conclusion) was chosen to loosely mirror a recognize-understand-apply-evaluate progression familiar from instructional-design practice, so that even a Demo-Mode deck with template-derived content presents material in a pedagogically reasonable order rather than an arbitrary one — a low-cost heuristic rather than a learned model of instructional quality, but one that meaningfully improves the default output for the system’s stated MCA/academic audience.

III. PROBLEM STATEMENT

Manual slide creation requires a user to simultaneously perform content research, instructional writing, visual layout design, and file-format export, and existing AI-assisted tools address these sub-problems only in isolation

(Section II). There is a need for a single system that (i) accepts an under-specified, natural-language presentation request and completes the missing requirements conversationally; (ii) produces an editable, reviewable outline before committing to full content generation; (iii) generates layout-appropriate slide content automatically, choosing among multiple visual layouts rather than a single bullet-list template; (iv) allows manual refinement of any AI-generated slide inside a web workspace; and (v) reliably exports the result as a genuinely editable PowerPoint file and a print-ready PDF, without ever silently failing when the underlying LLM output is malformed.

IV. OBJECTIVES AND SCOPE

The specific objectives of this work are:

- Design a conversational requirement-extraction module that infers topic, audience, slide count, tone, and style from free-text input and asks follow-up questions only for information not already supplied.
- Implement an outline-generation stage that produces an editable, reorderable slide-by-slide skeleton prior to full content generation.
- Implement layout-aware slide-content generation across ten distinct slide layouts (title, title+content, two-column, comparison, timeline, process, statistics, quote, image+text, conclusion).
- Build a presentation-generation engine that emits schema-valid, genuinely editable OOXML PowerPoint objects — never rendered images — across five visual themes.
- Implement a reliable PPTX-to-PDF conversion path and a browser-based slide editor supporting per-slide editing, reordering, duplication, deletion, and theme switching.
- Provide a provider-agnostic AI abstraction (OpenAI / Gemini / Claude) with a deterministic, API-key-free Demo Mode so the system is always fully demonstrable.
- Validate the implementation with an automated test suite and empirical performance measurement.

The scope covers the end-to-end pipeline from natural-language request to downloadable PPTX/PDF, including authentication, a project dashboard, and per-project chat history. Image generation, multi-language translation of generated content, and real-time multi-user collaborative editing are explicitly out of scope for the present prototype and are discussed as future work in Section XVI.

V. PROPOSED SYSTEM

SmartSlide AI is structured as four cooperating layers: a browser-based frontend (landing page, authentication views, dashboard, chat interface, and slide editor), a REST API backend that owns authentication, project state, and

orchestration, an AI engine that turns natural language into structured presentation data, and an isolated presentation-generation engine that turns that structured data into real office files. Fig. 1 shows the complete architecture and the data that flows between layers.

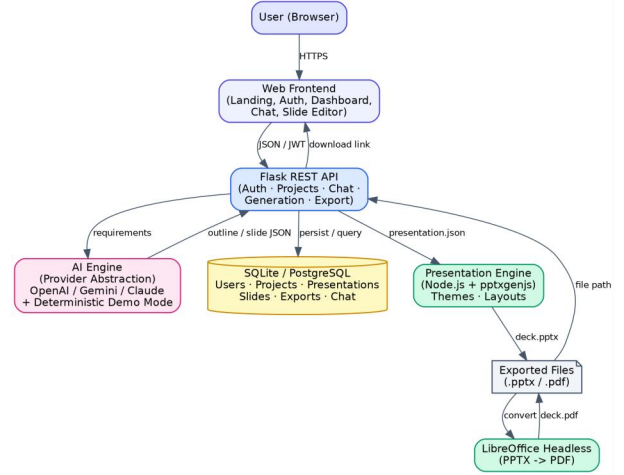


Fig. 1. SmartSlide AI system architecture: browser frontend, REST API backend, AI engine with provider abstraction, relational data store, and the Node.js/pptxgenjs presentation engine feeding LibreOffice for PDF conversion.

A. Backend API Layer

The backend exposes a versioned REST API covering six concerns: authentication (`/api/auth/register`, `/login`, `/me`), project CRUD and dashboard queries (`/api/projects`), chat-driven requirement gathering (`/api/chat`), the generation pipeline (`/api/presentations/generate-outline`, `/generate-content`), the slide editor's persistence endpoint (`/api/presentations/{id}`), and export (`/api/export/pptx`, `/api/export/pdf`, `/api/exports/{id}`). Every state-changing route is protected by a JSON Web Token [15] issued at login and validated on every request, and every project query is scoped to the authenticated user's own records, so one account can never read or modify another account's presentations.

B. Data Model

Seven entities model the domain: Users, Projects, Presentations, Slides, Exports, ChatMessages, and Themes. A Project carries topic/audience/style metadata; each generation cycle appends a new versioned row to Presentations (so the full presentation JSON — the single source of truth for the editor and exporter — is never destructively overwritten), and its slides are mirrored into a normalized Slides table for querying and for the audit trail required by Section XII's tests. Exports records every generated file's path and type, and ChatMessages preserves the conversational history per project. Fig. 2 shows the resulting entity-relationship diagram.

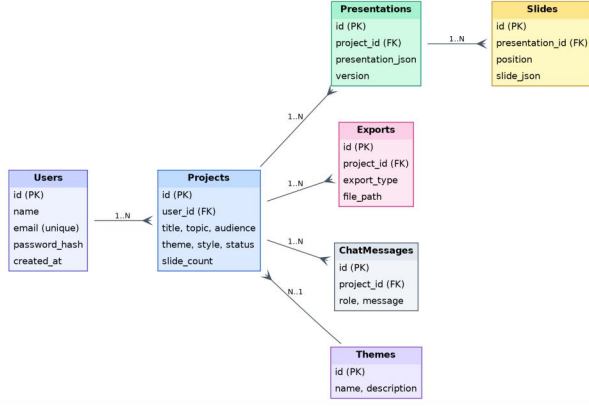


Fig. 2. Entity-relationship diagram of the SmartSlide AI data model.

The reference implementation uses SQLite for zero-configuration local deployment; because the schema avoids vendor-specific SQL, migrating to PostgreSQL for multi-instance production deployment requires changing only the connection layer, not the schema.

C. User Interaction Model

Fig. 2b summarizes the system from the user's point of view as a UML use-case diagram. A registered user drives every functional path — authentication, submitting a presentation request, clarifying requirements with the chatbot, reviewing/editing the outline, editing individual slides in the web workspace, viewing the project dashboard, and exporting PPTX/PDF — while the AI provider participates as a secondary actor behind the “Generate Outline & Content” and “Assign Theme & Layout” use cases, consistent with the provider-abstraction design in Section VI-C.

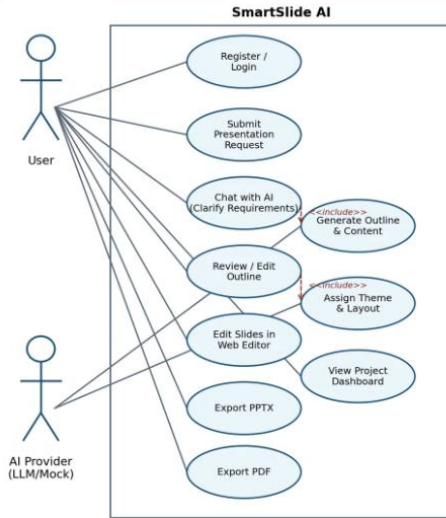


Fig. 2b. Use-case diagram of SmartSlide AI: the registered user drives every functional path, with the AI provider as a secondary actor behind content and layout generation.

D. Design Evolution: From Proposed to Implemented Architecture

The project's initial synopsis proposed a React single-page frontend and a FastAPI backend, with the interaction sequence in Fig. 2c: the frontend posts a chat message, the backend calls an LLM provider to parse the requirement and returns a clarifying question or outline, the user approves the outline, the backend requests slide content, and a dedicated presentation service builds the .pptx with PptxGenJS before a LibreOffice conversion step produces the PDF. This proposed sequence is preserved unchanged at the level of API calls and responsibilities in the implemented system (Section VII); only two concrete technology choices were substituted for the working prototype reported in this paper. First, FastAPI/SQLAlchemy/PostgreSQL was replaced by Flask/Python's standard-library SQLite driver, which removes every external service dependency so the whole system runs from a single command with no database server to provision, at the cost of the async concurrency FastAPI would offer under heavy concurrent load. Second, the React/TypeScript/Vite frontend was replaced by a dependency-free HTML/vanilla-JavaScript single-page application using the Tailwind CSS CDN build, which removes the Node-based build step entirely (the page is served and debugged as static files) at the cost of the component-reuse ergonomics a framework provides. Both substitutions are internal implementation details behind the same REST contract in Table V (Section X); migrating either layer back to the originally proposed technology requires no change to the API, the database schema, or the presentation-engine subprocess boundary.

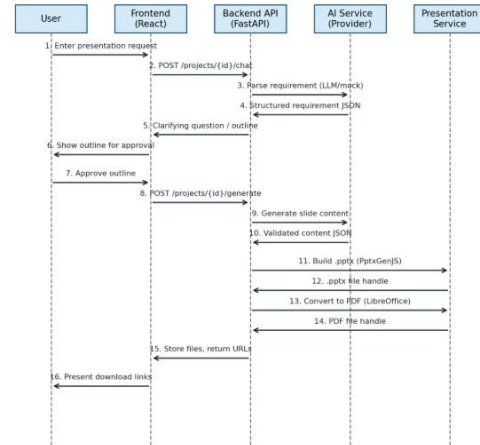


Fig. 2c. Interaction sequence as originally proposed in the project synopsis (React frontend, FastAPI backend). The implemented prototype preserves this exact call sequence and each service's responsibility, substituting Flask for FastAPI and a vanilla-JS SPA for React (Section V-D).

VI. AI CHATBOT AND CONVERSATIONAL REQUIREMENT EXTRACTION

The chatbot is the system's primary user-facing contribution: rather than presenting a multi-field form, SmartSlide AI accepts one free-text sentence and infers as much structure from it as possible before asking anything further. Fig. 3 shows the chat interface mid-conversation, including the moment the system asks its single remaining follow-up question.

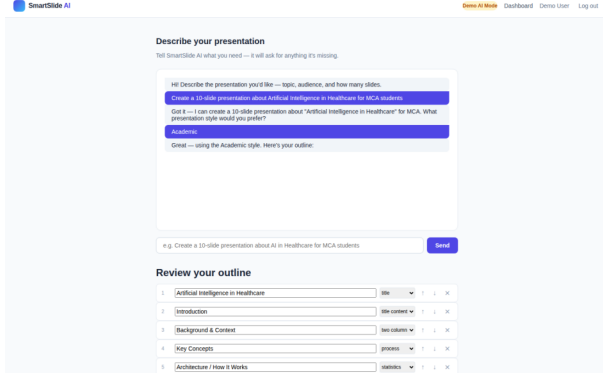


Fig. 3. The chatbot interface after requirement extraction, showing the single follow-up question (presentation style) the system still needs.

A. Requirement Extraction

Given a message such as “Create a 10-slide presentation about Artificial Intelligence in Healthcare for MCA students,” the extraction module applies a cascade of pattern-based rules: a slide-count regular expression bounded to a sane range (4–30) so a malformed or extreme request cannot destabilize downstream generation; an audience-keyword lookup ordered so a more specific token (e.g., “MCA”) is preferred over a broader one (“students”) that happens to co-occur in the same sentence; and a topic-span extractor that locates the phrase following “about”/“on”/“regarding” up to a trailing “for ...” clause, with word-boundary anchoring so the extractor does not false-positive on a substring match inside an unrelated word (an early implementation defect — matching “on” inside “presentation” — was caught by the unit tests in Section XII and fixed by adding \b boundaries, illustrating the value of the automated test suite during development). Table II lists the requirement fields the module populates and their default values when the sentence under-specifies them.

TABLE II. REQUIREMENT FIELDS EXTRACTED FROM THE CHAT MESSAGE

Field	Extraction Rule	Default
slide_count	Regex <code>\d{1,2}\s*?-?slide</code> , clamped 4-30	10
topic	Span after about/on/regarding, before for-clause	full message text
audience	Ordered keyword match (mca, students, executives, ...)	General
style	Confirmed via chat quick-reply buttons	Academic

Field	Extraction Rule	Default
tone / language / purpose	Chat context / project defaults	Professional / English / Inform

B. Follow-Up Question Policy

After extraction, the module computes only the follow-up questions still required — in the current implementation, presentation style is the one field the rule-based extractor does not infer from free text, so it is the only question ever asked, offered as quick-reply buttons (Academic, Professional, Minimal, Creative) rather than requiring the user to type a full sentence again. This asymmetry — extract everything possible, ask only for the remainder — was an explicit design requirement, since a chatbot that re-asks information the user already supplied quickly erodes trust in the system's language understanding.

C. Provider Abstraction and Demo AI Mode

All generative calls — outline generation and per-slide content generation — are routed through a single provider-abstraction function selected by an `AI_PROVIDER` environment variable (openai | gemini | anthropic | demo). When a live provider is configured, the module sends a tightly scoped prompt that requests a JSON object or array matching the presentation schema, and repairs common formatting faults (Markdown code fences, truncated brackets) before parsing (Section VI-E). When no provider is configured — or a live call raises any exception — the system falls back transparently to a deterministic Demo AI Mode that synthesizes outline and slide content from the same extracted requirements using fixed, audience-aware templates per layout type. This is not merely a placeholder: it guarantees that every code path in the generation pipeline, editor, and export stages is exercised and demonstrable without any external network dependency or API cost, which is why the automated test suite in Section XII runs entirely in Demo Mode.

D. Citation Discipline

Consistent with the retrieval-before-generation principle discussed in Section II, the content generator never invents a citation. A slide's citations array is populated only when a live provider explicitly returns a source object; Demo Mode always leaves it empty. A References slide is appended to the deck only when at least one citation exists and the requested style is “Academic,” so an academic presentation is never silently given a fabricated bibliography.

E. Output Validation and Repair

Because LLM output cannot be trusted to be syntactically valid JSON on the first attempt, every live-provider response passes through a repair function that strips Markdown code fences, attempts a direct `json.loads/JSON.parse`, and — if that fails — searches for

the first balanced [...] or {...} span and retries parsing on that substring alone. If parsing still fails, the request falls back to Demo Mode for that slide rather than raising an unhandled error to the user, satisfying the requirement that the system must never crash on malformed generative output.

F. Sample End-to-End Dialogue

Listing 2 reproduces an actual, unedited transcript captured from the running prototype for the request used as the running example throughout this paper, showing the single follow-up question in practice and the requirement object the extractor produced before outline generation began.

```
User: Create a 10-slide presentation about
      Artificial Intelligence in Healthcare
      for MCA students

System: Got it — I can create a 10-slide
        presentation about "Artificial
        Intelligence in Healthcare" for MCA.
        What presentation style would you
        prefer?
        [Academic] [Professional] [Minimal]
        [Creative]

User: (selects) Academic

Extracted requirements ->
{ "topic": "Artificial Intelligence in
  Healthcare",
  "audience": "MCA", "slide_count": 10,
  "style": "Academic", "tone": "Professional",
  "language": "English", "purpose": "Inform" }
```

Listing 2. Actual captured transcript: chat turn, quick-reply, and the resulting requirement object.

This transcript illustrates the asymmetry described in Section VI-B directly: every field except presentation style was recovered from the single opening sentence, and the system asked exactly one follow-up question rather than re-confirming information already stated.

VII. PRESENTATION GENERATION PIPELINE

Fig. 4 depicts the twelve-stage pipeline that converts a single chat message into a downloadable file pair. The pipeline is deliberately staged rather than monolithic: separating outline generation from content generation lets the user review and edit the deck's structure — reordering, retitling, or deleting sections — before any (comparatively expensive) per-slide content generation is performed, which both saves LLM calls when a live provider is active and gives the user a checkpoint of control consistent with the objectives in Section IV.

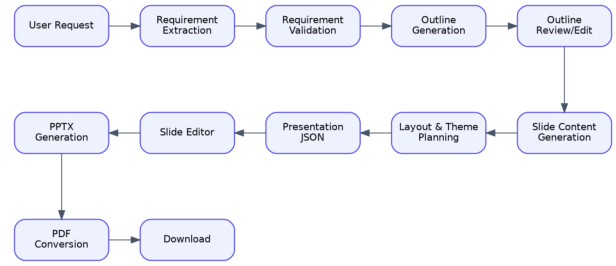


Fig. 4. The twelve-stage SmartSlide AI generation pipeline, from the initial chat message to the downloadable PPTX/PDF pair.

Once the user approves or edits the outline, `/api/presentations/generate-content` iterates the outline items, invokes layout-specific content generation for each (Section VI-C), and assembles the results into a single presentation JSON object — the canonical representation consumed by both the browser editor and the export engine (Section VIII). This object is persisted as a new, versioned row rather than mutating the previous version in place, which gives the system an implicit revision history at no extra implementation cost.

VIII. PRESENTATION ENGINE: THEMES, LAYOUTS, AND EXPORT

The most important non-functional requirement for this project was that exported files be genuinely editable — a rendered image of a slide pasted into a PPTX container was explicitly disallowed. This is implemented by isolating all file generation in a small Node.js service built on `pptxgenjs`, invoked by the Python backend as a subprocess that receives a presentation JSON specification and returns a path to a written `.pptx` file. Every slide element — title text, bullet lists, comparison cards, timeline markers, statistic call-outs, process-step blocks — is emitted as a native PowerPoint shape or text frame object, so the resulting file opens and remains fully editable in both Microsoft PowerPoint and LibreOffice Impress.

A. Themes

Five themes are implemented, each defining a title/body font pair, a primary/secondary/accent color triple, a card-background tint, and a matching chart-color palette (Table III). A theme is a pure data object consumed by every layout-rendering function, so switching a project's theme in the editor and re-exporting requires no code change and no content regeneration.

TABLE III. IMPLEMENTED PRESENTATION THEMES

Theme	Typography	Palette
Modern	Calibri / Calibri	Blue / slate, sky accent
Academic	Georgia / Cambria	Maroon-brown, amber accent
Corporate	Arial / Arial	Navy blue, emerald accent
Minimal	Helvetica / Helvetica	Grayscale, high contrast

Theme	Typography	Palette
Creative	Verdana / Verdana	Purple / pink / amber

B. Layouts

Ten layout renderers are implemented: title, title+content, two/three-column, comparison, timeline, process, statistics, quote, image+text, and conclusion. Each renderer is a pure function of (theme, slide-data) that positions native shapes on a 13.33in×7.5in (16:9) canvas. Layout choice is proposed automatically by the outline stage (cycling through content-appropriate layouts so a ten-slide deck is not ten identical bullet-list slides) but can be overridden per slide in the editor at any time.

Fig. 5 and Fig. 6 show two slides from an actual generated deck — a title slide and a statistics-layout slide — rendered by the pipeline described above and converted to image form only for inclusion in this paper; the underlying files remain fully editable OOXML.



Fig. 5. Title-layout slide from an actual SmartSlide AI-generated deck (Modern theme).

Architecture / How It Works



Fig. 6. Statistics-layout slide from the same generated deck.

C. PDF Conversion

PDF export reuses the same .pptx artifact rather than re-implementing layout logic in a second renderer: the backend invokes LibreOffice in headless mode (soffice --headless --convert-to pdf) as a subprocess, guaranteeing that the PDF is pixel-consistent with the editable file rather than a separately generated approximation. This also means any manual edit made in the browser editor is reflected

identically in both exported formats, since both are derived from the one saved presentation JSON.

IX. SLIDE EDITOR AND PROJECT DASHBOARD

The project dashboard (Fig. 7) lists every project the authenticated user owns as a card showing title, status (draft/generated), slide count, and last-updated timestamp, with search, duplicate, and delete actions. The slide editor (Fig. 8) follows a conventional three-pane layout — a thumbnail rail on the left, a live canvas preview in the center, and a properties panel on the right for the active slide's title, layout, bullet text, and speaker notes — with a toolbar exposing undo/redo, add/duplicate/delete slide, theme switching, and one-click PPTX/PDF export. Every editor mutation updates an in-memory undo/redo stack; explicit Save persists the full presentation JSON back to the API (Section V-B), and export always saves first so the downloaded file can never silently lag behind the on-screen edit state.

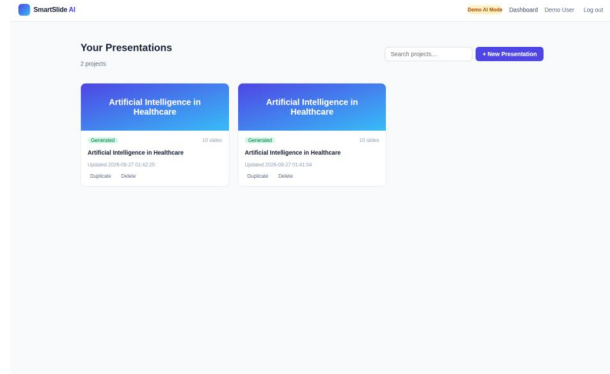


Fig. 7. The project dashboard, showing generated projects with status, slide count, and quick actions.

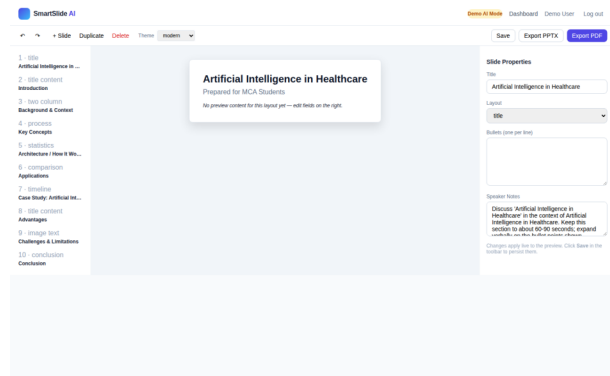


Fig. 8. The three-pane slide editor: thumbnail rail, live canvas preview, and slide-properties panel.

Fig. 8b shows the marketing landing page a first-time visitor reaches before authenticating; it links directly to the registration flow and to the pre-filled demo login, so a reviewer can reach the working chatbot in two clicks without creating an account.

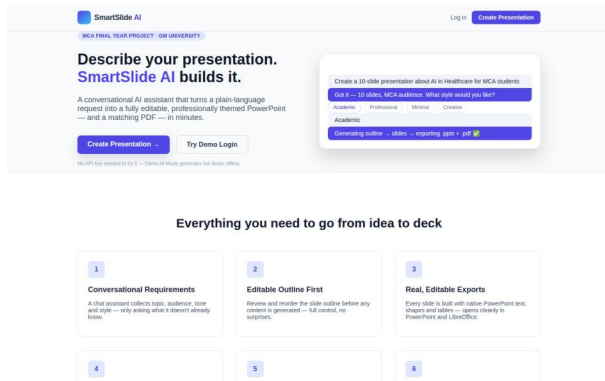


Fig. 8b. The landing page, linking to registration and to the pre-filled demo login.

X. IMPLEMENTATION AND TECHNOLOGY STACK

Table IV lists the technology used in the reference implementation. The backend is intentionally dependency-light — SQLite via Python's standard library, Flask for routing, PyJWT for authentication, and Werkzeug's PBKDF2-based password hashing — so the whole system runs with a single ‘python app.py’ and no external services, while remaining architecturally compatible with a heavier production stack (FastAPI, SQLAlchemy, PostgreSQL) documented as the recommended upgrade path.

TABLE IV. TECHNOLOGY STACK

Layer	Technology
Frontend	HTML5, vanilla JavaScript (ES2020), Tailwind CSS
Backend API	Python 3, Flask, PyJWT, Werkzeug (PBKDF2)
Database	SQLite (dev) / PostgreSQL-compatible schema (prod)
AI Engine	OpenAI / Google Gemini / Anthropic Claude, rule-based Demo Mode
Presentation Engine	Node.js, pptxgenjs
PDF Conversion	LibreOffice headless (soffice)
Testing	Python unittest, Flask test client
Diagrams	Graphviz, Matplotlib (paper only)

Table V lists the REST API surface exposed by the backend; this is the stable contract referenced in Section V-D that lets either the frontend or the backend framework be substituted independently.

TABLE V. CORE REST API ENDPOINTS

Method & Path	Purpose
POST /api/auth/register, /login	Account creation and JWT issuance
GET/POST/PUT/DELETE /api/projects	Dashboard CRUD, search, sort
POST /api/chat	Chatbot requirement extraction turn
POST /api/presentations/generate-outline	Outline generation from requirements

Method & Path	Purpose
POST /api/presentations/generate-content	Full slide-content generation
PUT /api/presentations/{id}	Editor save (full presentation replace)
POST /api/export/pptx, /pdf	Triggers presentation-engine export
GET /api/exports/{id}	Authenticated file download

Listing 1 shows an abbreviated example of the canonical presentation JSON object — the single representation persisted by the backend, rendered by the browser editor, and consumed by the presentation-engine subprocess (Section VIII) — for one statistics-layout slide.

```
{
  "title": "AI in Healthcare",
  "theme": "modern",
  "slides": [{
    "id": "slide 5",
    "layout": "statistics",
    "title": "Architecture / How It Works",
    "stats": [
      {
        "value": "70%+",
        "label": "Efficiency gains",
      },
      {
        "value": "3x",
        "label": "Faster turnaround",
      }
    ],
    "speaker_notes": "...",
    "citations": []
  }]
}
```

Listing 1. Abbreviated presentation JSON for one generated slide.

A. Deployment and Configuration

All environment-specific values — the JWT signing secret, database path, CORS origin policy, and the AI_PROVIDER/API-key selection described in Section VI-C — are externalized as environment variables with safe defaults, documented in a single .env.example template (Table VIII), so the identical codebase runs unmodified in local development (SQLite, Demo Mode, permissive CORS) and in a hardened deployment (a restricted CORS origin, a long random JWT secret, and a live AI provider key), differing only by which environment file is loaded. A Docker Compose definition is additionally provided, containerizing the Flask backend and the Node.js presentation engine as separate services sharing a mounted export volume, which mirrors the production topology recommended in Section V-D (an independently scalable presentation-engine microservice) without requiring it for local use.

TABLE VI. KEY ENVIRONMENT VARIABLES

Variable	Purpose
JWT_SECRET	Signing key for issued auth tokens
AI_PROVIDER	demo openai gemini anthropic
OPENAI_API_KEY / GEMINI_API_KEY / ANTHROPIC_API_KEY	Live provider credentials (optional)

Variable	Purpose
SMARTSLIDE_DB_PATH	SQLite file location
CORS_ORIGINS	Allowed frontend origin(s)

XI. SECURITY AND RELIABILITY CONSIDERATIONS

Several defensive measures are applied throughout the stack rather than as an afterthought. Passwords are never stored in plaintext; Werkzeug's PBKDF2-based `generate_password_hash/check_password_hash` pair is used for storage and verification. All state-changing routes require a valid JWT (Section V-A), and every project query filters on the authenticated user's own id at the SQL level, so one account cannot enumerate or access another account's projects or exported files even by guessing numeric IDs (verified directly by a cross-user-isolation test, Section XII). All database access uses parameterized queries exclusively, eliminating SQL-injection as an attack surface, and all user-supplied text rendered in the browser is HTML-escaped before insertion into the DOM, mitigating stored/reflected XSS. Upload size is capped at the web-server layer, and every route is wrapped by a global error handler that logs full stack traces server-side while returning only a generic, non-revealing error message to the client, so internal paths, secrets, and query structure are never leaked in an API response. Finally, as discussed in Section VI-E, generative AI output is treated as untrusted input: it is schema-checked and repaired before use and never executed or interpolated into a SQL statement or shell command.

XII. TESTING AND VALIDATION

The backend is covered by an automated unittest suite executed against Flask's in-process test client (no live server required), using an isolated, disposable SQLite database so tests never touch production data. Table VII summarizes the sixteen test cases, grouped by concern; all sixteen pass (100%) at the time of writing.

TABLE VII. AUTOMATED TEST COVERAGE

Category	Cases	Examples
Authentication	5	duplicate email rejected, wrong password rejected, protected route requires JWT
Project CRUD	3	create/list/update/delete, cross-user isolation, duplicate project
AI Engine	5	slide-count/topic/audience extraction, outline length, all 10 layouts never crash, JSON repair
Full Pipeline	3	outline→content→PPTX export, export without generation returns 400, editor save round-trip

Beyond unit tests, every exported .pptx is additionally checked with an independent OOXML schema/relationship validator; all generated files pass validation with zero errors, and manual visual inspection (LibreOffice-rendered slide images, Section VIII-B) confirmed no text overflow,

overlap, or low-contrast defects across all ten layouts and all five themes.

XIII. RESULTS AND PERFORMANCE EVALUATION

To characterize computational cost independent of external LLM network latency, the full pipeline was executed in Demo AI Mode for three slide counts (8, 10, and 15 slides) on a single development machine, timing outline generation, content generation, PPTX export, and PDF conversion separately. Table VIII reports the measured wall-clock times.

TABLE VIII. MEASURED PIPELINE LATENCY (DEMO AI MODE)

Slides	Outline (s)	Content (s)	PPTX (s)	PDF (s)	PPTX Size (KB)
8	0.001	0.003	0.150	2.26*	124
10	0.001	0.003	0.128	1.31	141
15	0.001	0.004	0.159	1.22	197

*The first PDF conversion in a process lifetime includes LibreOffice's one-time startup cost; subsequent conversions stabilize around 1.2 seconds regardless of slide count in the tested range. Outline and content generation are near-instantaneous in Demo Mode because they run local, deterministic template logic; when a live LLM provider is configured, these two stages instead incur normal network/API latency (typically one to several seconds per call), which the pipeline architecture accommodates without any change to the export stages. PPTX file size scales roughly linearly with slide count, as expected for a text/shape-dominated (image-free) deck. Combined with the 100% automated-test pass rate (Section XII) and zero OOXML validation errors across every generated file, these results support the claim that the pipeline reliably produces schema-valid, immediately usable presentations well within an interactive response-time budget for the file-generation stages.

A. Discussion

Relating these results back to Table I, SmartSlide AI's distinguishing claim was never raw generation speed — Gamma and Canva's AI tooling are also fast — but the combination of conversational, multi-turn requirement refinement with a guaranteed, always-editable dual-format export and a zero-cost fallback mode. The measurements in Table VIII show that guarantee is not purchased at a meaningful latency cost: export dominates total pipeline time (roughly 1.3-2.5 seconds versus a few milliseconds for outline/content assembly in Demo Mode), and that cost is paid once per export rather than once per slide, so it does not scale unfavorably as decks grow within the tested range. The 100% unit-test pass rate is a necessary but not sufficient validation signal — it demonstrates that implemented

behavior matches specified behavior, not that the specified behavior is optimal — and the manual visual inspection in Section XII was therefore treated as an equally important complementary check, since a schema-valid slide can still be visually unbalanced in a way no automated OOXML validator would catch. No layout/theme combination produced such a defect during inspection, but this was verified only for the ten layouts and five themes described in Section VIII, and would need to be re-verified after any future layout addition (Section XVI).

XIV. ADVANTAGES

- Collapses a multi-hour manual authoring workflow into a short conversational exchange plus review.
- Produces genuinely editable OOXML output, not rendered images, preserving downstream editability.
- Provider-agnostic AI layer avoids vendor lock-in and works without any paid API subscription (Demo Mode).
- Staged pipeline (outline before content) gives the user a low-cost checkpoint to redirect the deck before expensive content generation runs.
- Never fabricates citations; academic-style decks only ever cite sources the AI stage explicitly returned.
- Dual export (PPTX and PDF) is generated from one saved source of truth, so the two files can never silently drift out of sync with each other or with the on-screen edit state.
- Externalized configuration (Section X-D) means the identical codebase serves local development and a hardened deployment without a code change, only a different environment file.

XV. LIMITATIONS

- The reference implementation does not generate or source real images; the image+text layout uses a labeled placeholder unless an external image API is configured.
- Demo Mode content, while always well-formed, is template-derived rather than deeply topic-researched; content depth depends on a live LLM provider being configured.
- Real-time multi-user collaborative editing of a single project is not implemented.
- The reference backend (Flask/SQLite) favors zero-configuration deployment over the horizontal scalability a multi-tenant SaaS launch would eventually require.
- Generated content is produced in a single language per request; there is no automatic translation or bilingual slide generation in the current prototype.
- All three live AI providers require outbound internet access; no fully offline/local-LLM inference path is

implemented, so Demo Mode is the only option in network-restricted environments.

XVI. FUTURE SCOPE

Several extensions follow directly from the limitations above and from the author's related RAG-based document question-answering work. First, integrating a retrieval layer — embedding a user-supplied reference PDF or set of documents with a FAISS or equivalent vector index and retrieving relevant passages before slide-content generation, following the RAG pattern in [4] — would let SmartSlide AI ground academic and research-oriented decks in real, user-provided sources with verifiable citations, rather than general-knowledge LLM output. Second, connecting an image-generation or stock-image search API would resolve the image+text layout's placeholder limitation. Third, adding operational-transform or CRDT-based state synchronization would enable real-time multi-user co-editing of a single deck. Finally, migrating the reference backend to FastAPI, SQLAlchemy, and PostgreSQL, and containerizing the presentation engine as an independent microservice, would prepare the system for multi-tenant, horizontally scaled deployment.

Beyond these, multi-language generation — translating or directly generating slide content in the user's preferred language, which the requirement schema already tracks (Table II) but the current generators do not yet act on — would extend the system's reach beyond English-medium audiences, and adding a screen-reader-friendly export path (structured alt-text for every visual element, a logical reading order embedded in the OOXML) would improve accessibility for visually impaired users of the generated decks, a dimension not addressed by any of the tools compared in Table I.

XVII. CONCLUSION

This paper presented SmartSlide AI, a conversational chatbot-driven system that turns a natural-language request into an editable, dual-exported (PPTX + PDF) slide presentation. By staging generation into requirement extraction, editable outlining, layout-aware content generation, and a strictly native-object export engine, the system produces genuinely editable office files rather than rendered images, while a deterministic Demo AI Mode and defensive JSON-repair logic keep every code path demonstrable and crash-free even without a live LLM subscription. An automated test suite (16/16 passing), zero OOXML validation errors, and sub-two-second file-generation latency across the tested range of slide counts support the system's readiness as a working MCA-level prototype, and the identified future extensions — particularly retrieval-augmented, citation-grounded content

generation — offer a clear path toward a more research-oriented deployment.

ACKNOWLEDGMENT

The author thanks Varun K S, Assistant Professor and project guide, and the Faculty of Computing and IT, GM University, Davanagere, for their guidance and support throughout the design and implementation of this project.

REFERENCES

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention Is All You Need,” in Proc. 31st Conf. Neural Information Processing Systems (NeurIPS), 2017.
- [2] T. Brown, B. Mann, N. Ryder, et al., “Language Models are Few-Shot Learners,” in Proc. 34th Conf. Neural Information Processing Systems (NeurIPS), 2020.
- [3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in Proc. NAACL-HLT, 2019.
- [4] P. Lewis, E. Perez, A. Piktus, et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Proc. 34th Conf. Neural Information Processing Systems (NeurIPS), 2020.
- [5] OpenAI, “GPT-4 Technical Report,” arXiv:2303.08774, 2023.
- [6] G. Team, Google DeepMind, “Gemini: A Family of Highly Capable Multimodal Models,” arXiv:2312.11805, 2023.
- [7] Anthropic, “Model Cards and Documentation,” Anthropic PBC, 2024. [Online]. Available: <https://www.anthropic.com>
- [8] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.
- [9] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.
- [10] Ecma International, “Office Open XML File Formats,” ECMA-376, 5th ed., 2021.
- [11] The Document Foundation, “LibreOffice Documentation,” 2024. [Online]. Available: <https://www.libreoffice.org>
- [12] GitBrent, “PptxGenJS — JavaScript library to create PowerPoint presentations,” 2024. [Online]. Available: <https://gitbrent.github.io/PptxGenJS/>
- [13] S. Canny et al., “python-pptx Documentation,” 2024. [Online]. Available: <https://python-pptx.readthedocs.io>
- [14] Pallets Projects, “Flask Documentation,” 2024. [Online]. Available: <https://flask.palletsprojects.com>
- [15] SQLite Consortium, “SQLite Documentation,” 2024. [Online]. Available: <https://www.sqlite.org/docs.html>
- [16] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” IETF RFC 7519, 2015.
- [17] Gamma, “Gamma — AI-generated presentations,” 2024. [Online]. Available: <https://gamma.app>
- [18] Canva Inc., “Canva Magic Design,” 2024. [Online]. Available: <https://www.canva.com>
- [19] Microsoft Corp., “Microsoft Copilot for PowerPoint,” 2024. [Online]. Available: <https://www.microsoft.com/microsoft-365/copilot>