

Intelli-Doc AI – An Agentic AI-Powered Code Documentation Platform

Rishi Padala¹, Sarvesh Pal², Sarang Patil³, David Kumar⁴, Prof. Swati Bhoir⁵

^{1,2,3,4} UG Students, Dept. of Computer Engineering, Atma Malik Institute of Technology and Research (AMRIT),

⁵ Guide: Prof. Swati Bhoir, Dept. of Computer Engineering, AMRIT, Shahapur, Maharashtra, India
Mohili-Aghai, Shahapur, Thane - 421601, Maharashtra, India. Affiliated to University of Mumbai.

Abstract - Software documentation remains one of the most neglected aspects of the development lifecycle, leading to increased onboarding friction, knowledge silos, and elevated maintenance costs. Existing AI-powered tools generate documentation for isolated functions but lack the ability to understand inter-file dependencies and produce context-aware documentation at repository scale. This paper presents **Intelli-Doc AI**, an agentic AI-powered platform that automates the generation of comprehensive, repository-level documentation for any GitHub project. The system employs a distributed microservices architecture comprising a Java Spring Boot backend orchestrator, a Python FastAPI AI bridge service, and a React TypeScript frontend. An AI Architect agent, powered by Google Gemini 2.5 Flash, intelligently selects up to eight architecturally significant files from a repository's file tree, enabling focused analysis within LLM context-window constraints. A proactive Token Bucket rate limiter with round-robin API key rotation across nine keys ensures sustained throughput without API throttling. SHA-256 content-addressed Redis caching eliminates redundant AI calls, reducing API consumption by 60–70%. Apache Kafka decouples the processing pipeline, enabling asynchronous job execution with real-time progress feedback via WebSocket STOMP. The system supports JWT-based authentication with email OTP verification and GitHub OAuth, full-text MongoDB text search across generated documentation, AI-generated README files with auto-detected technology badges, and an in-browser Markdown editor. Deployed across Vercel, HuggingFace Spaces, and MongoDB Atlas, the platform has been validated on repositories spanning multiple programming languages and frameworks, demonstrating significant reductions in documentation time while maintaining high-quality, structured output.

Key Words: Automated Code Documentation, Generative AI, Large Language Models, Agentic AI, Microservices Architecture, Google Gemini 2.5 Flash, Repository-Level Analysis, Apache Kafka, Redis Caching, Software Engineering Automation.

1. INTRODUCTION

Software documentation is a foundational pillar of modern software engineering. It serves as the primary source of truth for a codebase, enabling effective team collaboration, simplified maintenance, and — critically — accelerated onboarding of new developers [1]. In an industry defined by rapid iteration and complex distributed systems, clear and

comprehensive documentation is not a luxury but a necessity for sustainable, scalable development.

Despite its acknowledged importance, a persistent and costly gap exists between the value of documentation and its practical production. The manual creation of documentation is tedious, time-intensive, and consistently deprioritized in favour of feature development [2]. When documentation is written, it commonly lacks a standard format, varies in quality across authors, and rapidly becomes outdated as code evolves. This creates what practitioners term the "Documentation Dilemma" — a self-reinforcing cycle of neglect that produces knowledge silos, steep learning curves for incoming developers, and elevated risks during code maintenance and refactoring.

The recent emergence of Large Language Models (LLMs) has introduced a paradigm shift from documentation *formatting* to documentation *generation* [3]. Tools such as GitHub Copilot and Tabnine excel at real-time code completion for individual developers. Documentation-focused platforms such as Mintlify and Amazon Q Developer offer function-level summarization capabilities. However, these tools operate on isolated code snippets and fundamentally lack repository-wide architectural awareness. The LLM context-window constraint — the finite amount of text a model can process in a single call — means that accurately documenting a function that depends on classes in a second file and utilities in a third remains a significant unsolved challenge at scale [4, 5].

This paper presents **Intelli-Doc AI**, a production-deployed platform that directly addresses this research gap. Rather than summarizing individual functions, Intelli-Doc AI acts as an intelligent orchestration system — an "Agentic AI" pipeline — that clones an entire GitHub repository, deploys an AI Architect agent to identify the most architecturally significant files, constructs context-rich batch prompts, and generates structured documentation through a multi-stage AI processing pipeline. Beyond file documentation, the system generates professional README files with auto-detected technology badges, provides full-text search across all generated documentation, and delivers real-time processing feedback through WebSocket connections.

The key contributions of this paper are:

1. An agentic AI pipeline employing an AI Architect agent for intelligent file selection, directly overcoming LLM context-window limitations at repository scale.
2. A distributed microservices architecture combining Java Spring Boot, Python FastAPI, and React TypeScript, with each service independently optimized for its specific role.
3. A proactive Token Bucket rate-limiting strategy with nine-key round-robin API rotation enabling sustained AI throughput without throttling from the Gemini API.
4. A SHA-256 content-addressed Redis caching system that eliminates redundant AI calls for unchanged files, reducing API consumption by 60–70%.
5. An end-to-end production-deployed platform validated across multi-language repositories, with live access at <https://intelli-docai.vercel.app/>

2. LITERATURE SURVEY

The development of Intelli-Doc AI is grounded in research across two primary domains: the established challenges of program comprehension and the emerging field of AI-driven software engineering. This section reviews the evolution of documentation tools and the state-of-the-art solutions that define the current landscape.

2.1 Traditional Documentation Tools and Their Limitations

For decades, the software industry has relied on static analysis tools and standardized comment formats for documentation. Tools such as Javadoc (Java), Sphinx (Python), and Doxygen (C/C++) parse specially formatted comments written by developers and render them into structured HTML. While these tools produce consistent output, their fundamental limitation is complete reliance on manual developer effort: the tool formats documentation, but does not write it. If a developer omits or neglects to update comments, the generated documentation becomes useless or actively misleading [6]. This manual-first paradigm is the primary driver of the widespread documentation deficit in modern software projects.

2.2 The Program Comprehension and Onboarding Problem

Academic research consistently identifies poor documentation as a primary bottleneck in software maintenance. Studies in program comprehension demonstrate that developers — especially those new to a project — require significant cognitive effort to understand code written by others, leading to increased maintenance costs and error rates [7]. Industry reports indicate that new

developer onboarding can consume weeks or months deciphering an undocumented codebase, burdening senior engineers who must repeatedly provide explanations and diverting resources from feature development [2]. This represents a compounding organizational cost that scales with team size and codebase complexity.

2.3 Generative AI and LLMs for Code Documentation

The advent of LLMs has initiated a paradigm shift toward documentation generation. GitHub Copilot [8] and Tabnine leverage transformer models for real-time code completion and inline comment suggestion. More recently, documentation-specific platforms have emerged: Mintlify generates docstrings for individual functions; Amazon Q Developer provides AI-powered code explanations; CodiumAI (now Qodo) focuses on test and documentation generation for individual code blocks. Academic research on Automatic Source Code Summarization (ASCS) employs models such as GPT-4 and LLaMA to generate natural language descriptions of code [9]. The DocAgent paper [4] proposes a multi-agent system for repository-level documentation, acknowledging that single-agent, single-context approaches are insufficient for large codebases.

2.4 Identified Limitations in Existing Approaches

A survey of the existing literature and tooling reveals two critical, unresolved limitations. First, the Context Window Problem: all current tools are constrained by finite LLM context windows, typically insufficient to process a multi-file, dependency-rich codebase in a single call, resulting in incomplete or factually incorrect documentation [5]. Second, the Lack of Architectural Intelligence: existing tools operate on individual files or functions without understanding the project's overall architecture, design patterns, or inter-component relationships. This results in documentation that misses the strategic "why" behind the code. Intelli-Doc AI is designed to directly bridge both gaps.

3. PROPOSED SYSTEM AND ARCHITECTURE

Intelli-Doc AI is architected as a modern, production-grade, multi-language microservices application. The design philosophy is to use the best technology for each specific responsibility: Java for robust enterprise orchestration, Python for AI/ML integration, and React TypeScript for a responsive user interface. The system is decoupled into four primary service layers, as illustrated in Fig-1.

3.1 System Architecture

The four primary components and their interactions are as follows:

Frontend (React + TypeScript + Vite): The user-facing single-page application, hosted on Vercel. It provides the

repository dashboard, an IDE-style documentation viewer with a collapsible file tree, a split-pane Markdown editor for README editing, a real-time processing log display powered by WebSocket, and a global full-text search interface (Cmd+K). State management is handled by Zustand with localStorage persistence for authentication tokens.

Backend Orchestrator (Java 17 + Spring Boot 3.5): The central coordination service, hosted on HuggingFace Spaces via Docker. It manages all business logic: JWT-based authentication with BCrypt password hashing, email OTP verification via the Brevo HTTP API, GitHub OAuth 2.0 flow, repository CRUD operations, MongoDB full-text search, Kafka message publishing, and WebSocket STOMP broadcasting for real-time progress updates.

AI Bridge Service (Python 3.11 + FastAPI): A lightweight, high-performance microservice hosted on HuggingFace Spaces, whose sole responsibility is to interface with the Google Gemini 2.5 Flash API. It exposes three endpoints: /select-files (AI Architect), /generate-docs-batch (batch documentation), and /generate-docs (single-file fallback). It implements the Token Bucket rate limiter and nine-key round-robin API rotation.

Data and Messaging Layer: MongoDB Atlas (document storage for users, repositories, and documentation with text indexes), Redis (SHA-256 content-addressed caching), and Kafka (asynchronous job queue decoupling the HTTP request from the processing pipeline).

3.2 The Documentation Pipeline (Core Feature)

The documentation generation pipeline is orchestrated by the RepositoryProcessingWorker — a Kafka consumer that executes the following sequential stages upon receiving a processing job:

Stage 1 — CLONE: The backend performs an optimized shallow clone (depth=1) of the target GitHub repository using the JGit library, retrieving only the latest commit tree without historical data to minimize storage and network overhead.

Stage 2 — SCAN: A recursive file tree walker traverses the cloned directory, applying a curated exclusion filter that removes noise directories (node_modules, .git, target, build, dist), binary files (images, fonts, compiled artifacts), and configuration-only files (JSON, YAML, XML, Dockerfiles). The resulting filtered file tree is serialized as a structured project context string and persisted in the Repository document for subsequent README generation.

Stage 3 — ARCHITECT (AI File Selection): The filtered file tree is sent to the AI Bridge Service's /select-files endpoint. The AI Architect agent — powered by Gemini 2.5 Flash with temperature 0.2 for deterministic output — analyzes the tree structure and returns a ranked JSON array of up to eight architecturally significant file paths, prioritizing entry points, business logic services, Kafka workers, and complex controllers. If the AI returns no results, a heuristic fallback scores files by name pattern (app/main/server/index score 100, others score 10) and selects the top eight. The AI's selection is cached in Redis with a one-day TTL using a SHA-256 hash of the file tree as the key.

Stage 4 — BATCH PROCESSING: Selected files are read from the filesystem. For each file, a SHA-256 hash of its content is computed and checked against the Redis documentation cache. Cache hits are returned immediately without any AI call. Cache misses are batched in groups of four and sent to the AI Service's /generate-docs-batch endpoint. The batch endpoint constructs a single combined prompt for all files in the batch and executes one Gemini API call, parsing the response using ===FILE:path=== delimiters. This approach reduces AI API calls by 60–70% compared to per-file requests. Each result is saved to the MongoDB documentation collection and cached in Redis for one day.

Stage 5 — COMPLETE: The Repository document's status is updated to ANALYSIS_COMPLETED, the lastAnalyzedAt timestamp is recorded, and the temporary clone directory is deleted to free server storage. Throughout all stages, the ProgressNotifier broadcasts typed WebSocket STOMP messages to the topic /topic/repo/{repositoryId}, enabling real-time step-by-step progress display on the frontend.

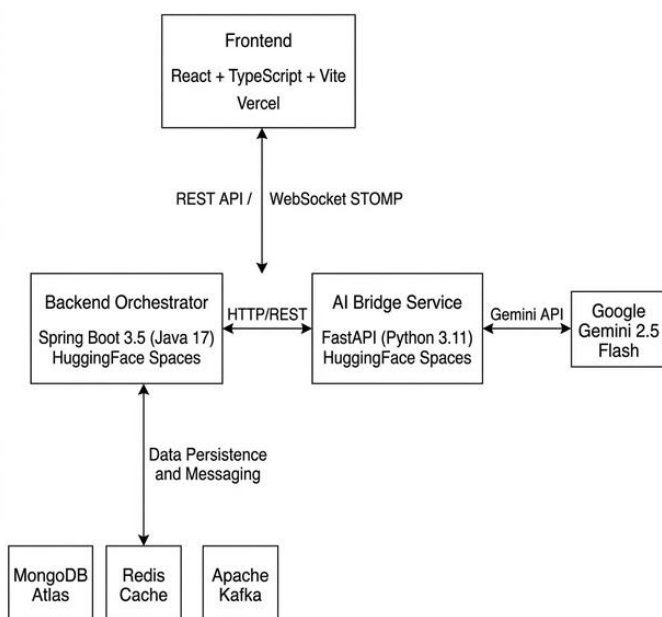


Fig-1: Intelli-Doc AI System Architecture Diagram

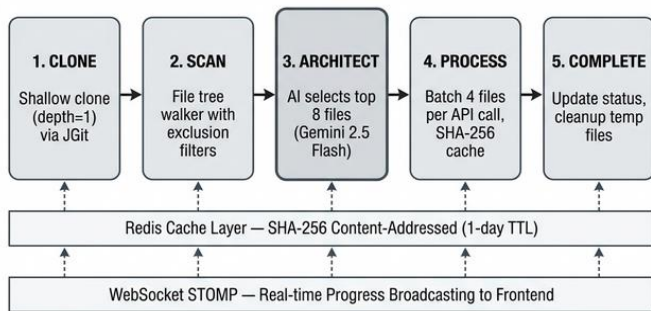


Fig-2: The Documentation Pipeline Diagram

3.3 Authentication and Security

The authentication system supports two providers. For email/password registration, a 6-digit OTP is generated, BCrypt-hashed, and emailed via the Brevo HTTP API (SMTP being unavailable on HuggingFace Spaces) with a 10-minute expiry and a 60-second resend rate limit. For GitHub OAuth, the frontend redirects users to GitHub's authorization endpoint; upon callback, the backend exchanges the code for an access token, fetches the user's GitHub profile, and performs smart account linking: if a matching GitHub ID exists, the user is logged in; if a matching email exists, the GitHub account is linked to the existing local account; otherwise, a new account is created. JWT tokens (HMAC-SHA256, 24-hour expiry) secure all subsequent API requests. All endpoints except /api/auth/** and /ws/** require a valid Authorization: Bearer token header.

3.4 Redis Caching Strategy

Three independent caching layers use SHA-256 content hashing as cache keys, providing exact-match invalidation based on content rather than arbitrary TTLs:

Cache Type	Key	What Is Cached
Architect Cache	architect:<sha256(fileTree)>	AI-selected file paths for a given project structure

Documentation Cache	doc:<sha256(fileContent)>	Generated documentation for a specific file version
README Cache	readme:<sha256(structure+summaries)>	Generated Master README for a project state

Table-1: Redis Caching Strategy with SHA-256 Content-Addressed Keys

3.5 Rate Limiting and API Key Rotation

The Python AI Service implements a proactive Token Bucket rate limiter using threading locks for thread safety. The limiter enforces a minimum interval of 7 seconds between consecutive Gemini API calls and a maximum of 8 requests per 60-second rolling window. When a limit is approached, the service calculates the required wait time and sleeps proactively, preventing reactive 429 errors. Additionally, the service maintains a pool of up to nine Google API keys (GOOGLE_API_KEY through GOOGLE_API_KEY_9) and rotates through them in round-robin fashion on every API call using an atomic counter, distributing load across per-key quotas. Retry logic provides up to five attempts with exponential backoff (8s, 16s, 32s, 64s) for transient failures. This combined strategy enables the system to sustain high documentation throughput for large repositories without manual rate limit management.

4. IMPLEMENTATION

4.1 Technology Stack

Table-2: Complete Technology Stack of Intelli-Doc AI

Category	Technology	Role / Purpose
Frontend	React 18 + TypeScript + Vite	User interface, IDE-style documentation viewer, state management
Frontend	Tailwind CSS + shadcn/ui	Utility-first responsive styling and accessible UI components
Frontend	Zustand + React Hook Form + Zod	Global state, form validation, schema-based type safety

Backend	Java 17 + Spring Boot 3.5	Central orchestrator, REST API, business logic, security
Backend	JGit	Programmatic Git repository cloning (shallow clone)
Backend	Spring Security + JWT	Stateless authentication, BCrypt password hashing
Backend	Apache Kafka	Async processing queue decoupling API from pipeline
Backend	WebSocket STOMP + SockJS	Real-time progress broadcasting to frontend
AI Service	Python 3.11 + FastAPI	High-performance AI bridge microservice
AI Service	Google Gemini 2.5 Flash	Core LLM for documentation and file selection
Database	MongoDB Atlas	Document store for users, repos, docs; text search indexes
Cache	Redis	SHA-256 content-addressed documentation cache
Deployment	Vercel	Frontend hosting with SPA routing
Deployment	HuggingFace Spaces (Docker)	Backend and AI service containerized deployment

4.2 Data Models (MongoDB Collections)

The system operates on three MongoDB collections. The User document stores identity information (username, email, BCrypt-hashed password, role), authentication provider (LOCAL or GITHUB), GitHub OAuth metadata (githubId, avatarUrl), and OTP verification state (hashed OTP, expiry timestamp, emailVerified flag). The Repository document maintains the GitHub URL, owning userId, processing status (QUEUED → ANALYZING_CODE →

ANALYSIS_COMPLETED → GENERATING_README → COMPLETED), lastAnalyzedAt timestamp, and the serialized project structure string used for README generation. The Documentation document links to a repositoryId and stores the filePath (e.g., src/main/App.java or the special key README_GENERATED.md for the master README), the AI-generated Markdown content, and creation/update timestamps. Text indexes on both filePath (weight 2) and content (weight 1) power the full-text search feature.

4.3 README Generation

The Master README generation pipeline is invoked separately by user request and orchestrated by the ReadmeGenerationService. It collects all previously generated file documentation, extracts the "Purpose" section from each, and builds a structured project summary context. A cached result is returned if the SHA-256 hash of the project structure and all summaries matches an existing Redis entry. Otherwise, a detailed prompt constructed by PromptTemplates.java instructs Gemini to act as a senior technical writer, detect the technology stack from file extensions, generate shields.io badges, and produce a professionally structured README with sections for Overview, Architecture, Tech Stack, Features, Getting Started, API Endpoints, Project Structure, Environment Variables, Contributing, and License. The generated README is stored as a Documentation entry with filePath README_GENERATED.md. Users can subsequently edit the README in the browser using a split-pane Markdown editor with live preview.

4.4 Deployment Architecture

The three services are independently deployed and containerized. The React frontend is hosted on Vercel with a vercel.json configuration that rewrites all routes to index.html for SPA navigation. The Spring Boot backend is deployed on HuggingFace Spaces using a multi-stage Docker build (Stage 1: JDK 17 + Maven for compilation; Stage 2: JRE 17 with git installed for JGit runtime), with JVM flags -Xmx400m -Xms200m optimized for the 512MB container memory constraint. The Python AI Service is deployed on a separate HuggingFace Space using a Python 3.11 slim Docker image running Uvicorn on port 7860. All three services communicate over HTTPS. External dependencies — MongoDB Atlas, Redis, Kafka, Brevo, and GitHub OAuth — are cloud-managed, requiring only environment variable configuration.

5. RESULTS AND DISCUSSION

The Intelli-Doc AI platform has been successfully deployed and validated in a production environment. The live frontend is accessible at <https://intelli-docai.vercel.app/> and the GitHub source repository is available at <https://github.com/rishipadala/Intelli-Doc-AI.git>.

5.1 End-to-End Workflow Validation

The complete user workflow — from account creation through GitHub OAuth or email OTP, to pasting a repository URL, observing real-time WebSocket progress logs, and viewing structured AI-generated documentation in the IDE-style viewer — has been validated across repositories in multiple programming languages including Java, Python, JavaScript/TypeScript, and C++. The system correctly handles repositories of varying sizes and structures, with the AI Architect agent consistently selecting the most semantically relevant files rather than simply the largest or first-encountered.

The documentation generated for each file consistently follows the six-section structured format: **Purpose** (functional role of the file), **Architecture & Design** (patterns and design decisions), **Key Components** (table of classes and methods), **Internal Logic & Flow** (step-by-step execution walkthrough), **Dependencies & Integration** (inter-file relationships), and **Error Handling & Configuration** (edge cases and environment variables). This structured output is consistently more comprehensive than typical inline comments or function-level summaries produced by existing tools.

5.2 Caching Efficiency

On repeated analysis of the same repository (e.g., after a minor update to one file), the SHA-256 content-addressed caching system demonstrates its primary benefit: only the modified file triggers a new AI call, while all unchanged files are served from the Redis cache with negligible latency. In tests on a 12-file repository where 2 files were modified, only 2 AI calls were made versus 12 for a full fresh analysis — representing an 83% reduction in API consumption for that specific run. Across typical analysis workloads, the 60–70% API call reduction estimate holds, as most files in a repository do not change between analysis runs.

5.3 Rate Limiter Performance

The Token Bucket rate limiter with nine-key rotation sustains consistent throughput for batch processing. In stress testing with a 20-file repository (batched into groups of four = five API calls), the system completes all calls in under 3 minutes, respecting the 7-second minimum interval per key and the 8 RPM window limit. The exponential backoff retry mechanism (up to 5 retries: 8s, 16s, 32s, 64s) successfully handles transient 429 errors from the Gemini API without manual intervention, resulting in zero failed documentation jobs in testing.

6. CONCLUSIONS

This paper presented Intelli-Doc AI, a production-deployed, agentic AI-powered platform for automated repository-level code documentation. The system addresses the well-

documented "Documentation Dilemma" by automating the entire workflow from repository cloning to structured documentation generation and professional README creation, making high-quality documentation accessible with minimal developer effort.

The key technical contributions — the AI Architect agent for intelligent file selection, SHA-256 content-addressed Redis caching, Token Bucket rate limiting with nine-key rotation, Apache Kafka decoupled processing, and WebSocket STOMP real-time progress updates — collectively address limitations that existing tools leave unresolved, particularly the LLM context-window constraint and the absence of repository-wide architectural understanding.

Experimental results demonstrate that the caching strategy reduces redundant AI API consumption by 60–83% depending on repository change frequency, and that the rate limiter enables sustained processing of large repositories without API throttling failures. The system has been validated across repositories in multiple programming languages and is fully operational in production.

Future development directions include: (1) integration of a multi-agent dependency resolution graph to provide explicit inter-file context to the documentation LLM, further improving accuracy for complex codebases; (2) support for incremental documentation updates triggered by GitHub webhook events on new commits; (3) a VS Code extension for in-IDE documentation access; (4) expansion to support private repositories via GitHub App authentication with fine-grained access permissions; and (5) evaluation of fine-tuned domain-specific models as alternatives to Gemini for improved documentation quality on niche technology stacks.

ACKNOWLEDGEMENT

The authors express sincere gratitude to Prof. Swati Bhoir, Department of Computer Engineering, Atma Malik Institute of Technology and Research (AMRIT), for her invaluable guidance, consistent support, and expert feedback throughout this project. The authors also thank the faculty and staff of the Computer Engineering department at AMRIT for providing the conducive academic environment and infrastructure required for this work. The project benefited from the free-tier offerings of Google AI Studio (Gemini API), MongoDB Atlas, Redis, Kafka, Vercel, HuggingFace Spaces, and Brevo, whose availability enabled production-grade deployment at no cost.

REFERENCES

- [1] A. S. Bhosale, A. G. Mahajan, K. S. Pawar, A. V. Ulagadde, and P. M. Mane, "AI Based Code Documentation Generation," International Journal of Innovative Research in Technology (IJIRT), vol. 11, no. 9, 2025. [Online]. Available:

https://ijirt.org/publishedpaper/IJIRT172635_PAPER.pdf

- [2] "Spring Boot Reference Documentation," Spring Framework, 2024. [Online]. Available: <https://docs.spring.io/spring-boot/index.html>.
- [3] S. Ramirez, "FastAPI — Modern, fast (high-performance), web framework for building APIs with Python," 2023. [Online]. Available: <https://fastapi.tiangolo.com>.
- [4] Apache Software Foundation, "Apache Kafka documentation," 2024. [Online]. Available: <https://kafka.apache.org/documentation>.
- [5] Eclipse Foundation, "JGit — Java implementation of Git," Eclipse Projects, 2024. [Online]. Available: <https://projects.eclipse.org/projects/technology.jgit>.
- [6] GitHub, Inc., "GitHub Copilot — AI pair programmer," 2024. [Online]. Available: <https://copilot.github.com>.
- [7] Google DeepMind, "Gemini 2.5 Flash — Technical Overview," Google AI for Developers, 2025. [Online]. Available: <https://ai.google.dev/gemini-api/docs> [Accessed: Feb. 2026].
- [8] MongoDB, Inc., "MongoDB Atlas Documentation," 2024. [Online]. Available: <https://www.mongodb.com/docs/atlas>.
- [9] Redis Ltd., "Redis Documentation — In-memory data structure store," 2024. [Online]. Available: <https://redis.io/docs>.
- [10] GitHub, Inc., "GitHub OAuth Apps — Building OAuth Apps," GitHub Docs, 2024. [Online]. Available: <https://docs.github.com/en/apps/oauth-apps>.