

**RANCANG BANGUN PLUGIN VS CODE EXTENSION
UNTUK DOKUMENTASI OTOMATIS REST API DENGAN
STANDAR OPENAPI BERBASIS TOOLS AI**

TUGAS AKHIR



**MOHAMAD AENUR ROKHMAN
21106050081**

**PROGRAM STUDI INFORMATIKA
FAKULTAS SAINS DAN TEKNOLOGI
UNIVERSITAS ISLAM NEGERI SUNAN KALIJAGA
YOGYAKARTA
2025**

LEMBAR PENGESAHAN

SURAT PERSETUJUAN

PERNYATAAN KEASLIAN

KATA PENGANTAR

HALAMAN PERSEMBAHAN

HALAMAN MOTTO

DAFTAR ISI

LEMBAR PENGESAHAN	i
SURAT PERSETUJUAN	ii
PERNYATAAN KEASLIAN.....	iii
KATA PENGANTAR	iv
HALAMAN PERSEMBAHAN.....	v
HALAMAN MOTTO	vi
DAFTAR ISI.....	vii
DAFTAR GAMBAR	x
DAFTAR TABEL.....	xii
ABSTRAK	xiv
BAB I PENDAHULUAN.....	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	5
1.3 Batasan Masalah.....	5
1.4 Tujuan	7
1.5 Manfaat	8
BAB II KAJIAN PUSTAKA.....	10
2.1 Penggunaan Model AI/LLM dalam Dokumentasi Kode	10
2.2 Pendekatan Rule-based vs AI-based pada API Documentation	15
2.3 REST API dan OpenAPI 3.1.....	18
2.4 Arsitektur VS Code Extension API.....	22
2.5 Model AI Gemma 3 12B-IT dan Integrasi Openrouter AI	27

2.6 Format Output YAML/JSON pada OpenAPI	30
BAB III METODE PENGEMBANGAN SISTEM	38
3.1 Metodologi Pengembangan.....	38
3.2 Analisis Kebutuhan	39
3.2.1 Kebutuhan Fungsional	40
3.2.2 Kebutuhan Non-Fungsional	41
3.3 Perancangan Arsitektur Sistem	42
3.3.1 Arsitektur Client-Server Plugin dan Openrouter AI	43
3.3.2 Alur Data dan Mekanisme Komunikasi.....	44
3.4 Persiapan Data dan Model AI	46
3.4.1 Deskripsi Model Gemma 3 12B-IT.....	46
3.4.2 Mekanisme Prompt Engineering dan API Request.....	48
3.5 Alat dan Bahan.....	50
3.5.1 Perangkat Keras	50
3.5.2 Perangkat Lunak.....	51
3.5.3 Library dan Framework Pendukung.....	52
BAB IV PERANCANGAN DAN IMPLEMENTASI SISTEM	53
4.1 Desain Sistem.....	53
4.1.1 Diagram UML.....	53
4.1.2 Struktur Folder Plugin.....	57
4.1.3 Desain Antarmuka Pengguna (UI/UX)	62
4.2 Implementasi Plugin.....	66
4.2.1 Implementasi Modul Utama VS Code Extension API.....	69
4.2.2 Integrasi Model Gemma 3 12B-IT via Openrouter AI.....	Error! Bookmark not defined.

4.2.3 Contoh Kode Utama (Snippet & Pseudocode)Error! Bookmark not defined.	
4.3 Pengujian Sistem.....	138
4.3.1 Pengujian Fungsionalitas Plugin.....	138
4.3.2 Pengujian Validasi Terhadap Format OpenAPI.....	138
4.3.3 Pengujian Performa.....	138
4.3.4 Pengujian Akurasi Dokumentasi.....	138
4.4 Hasil dan Pembahasan.....	138
4.4.1 Hasil Pengujian Setiap Fitur.....	140
4.4.2 Analisis Efektivitas Plugin.....	140
4.4.3 Perbandingan dengan Tools Lain (Swagger AutoDoc, DocGen)	140
4.4.4 Evaluasi Kinerja dan Akurasi Model AI.....	140
4.4.5 Kelebihan dan Kekurangan Sistem	140
BAB V PENUTUP.....	141
5.1 Kesimpulan	141
5.2 Saran untuk Pengembangan Selanjutnya	141
DAFTAR PUSTAKA	143

DAFTAR GAMBAR

Gambar 2.1.1 Alur Logika Dokumentasi Otomatis LLM.....	13
Gambar 2.3.1 Kosep Implementasi REST API dengan Dokumen OpenAPI	21
Gambar 2.4.1 Tahapan Siklus Ekstensi VS Code	23
Gambar 2.4.2 Integrasi VS Code Extension Model AI.....	25
Gambar 2.5.1 Alur Integrasi Gemma–Openrouter AI dalam Plugin VS Code....	29
Gambar 2.6.1 Output File Plugin VS Code.....	37
Gambar 3.1.1 Ilustrasi Papan Kanban pada Pengembangan Sistem.....	39
Gambar 3.3.1 Arsitektur Client Server	43
Gambar 4.2.1 Konfigurasi tsconfig.json	74
Gambar 4.2.2 Pengujian Extension Development Host.....	75
Gambar 4.2.3 Notifikasi Pengujian Extension Development Host.....	76
Gambar 4.2.4 Generate API Key Openrouter	77
Gambar 4.2.5 Konfigurasi Postman	78
Gambar 4.2.6 Pengujian HTTP POST Model AI Openrouter	79
Gambar 4.2.7 Hasil Script Testing Standalone	83
Gambar 4.2.8 Arsitektur Class SecureStorageService.....	86
Gambar 4.2.9 Command Set API Key	87
Gambar 4.2.10 Notifikasi Set API Key.....	87
Gambar 4.2.11 Notifikasi Cek API Key	87
Gambar 4.2.12 Verifikasi Notifikasi Hapus API Key.....	88
Gambar 4.2.13 Notifikasi Hapus API Key.....	88
Gambar 4.2.14 Sample-express-project	93
Gambar 4.2.15 Unit Test RouteParser.test.....	93
Gambar 4.2.16 Identifikasi Status Code	97
Gambar 4.2.17 Flowchart Error Handling dengan Retry Logic.....	104
Gambar 4.2.18 Pengujian Hasil Prompt Sederhana	109
Gambar 4.2.19 Contoh Laporan Validasi Hasil Dokumentasi 1.....	121
Gambar 4.2.20 Contoh Laporan Validasi Hasil Dokumentasi 2.....	122
Gambar 4.2.21 Webview API Documentation Settings 1.....	127

Gambar 4.2.22 Webview API Documentation Settings 2.....	128
Gambar 4.2.23 Webview Documentation Preview.....	134

DAFTAR TABEL

Tabel 2.1.1 Perbandingan Kemampuan Utama LLM	12
Tabel 2.1.2 Perbandingan Kelebihan dan Batasan LLM	12
Tabel 2.2.1 Rule-based vs AI-based dalam Dokumentasi API.....	17
Tabel 2.3.1 Contoh Struktur REST API.....	19
Tabel 2.4.1 Struktur Komponen Ekstensi VS Code.....	24
Tabel 2.6.1 Struktur Umum OpenAPI	30
Tabel 2.6.2 Struktur OpenAPI 3.1 dengan Format YAML.....	31
Tabel 2.6.3 Struktur OpenAPI 3.1 dengan Format JSON.....	33
Tabel 2.6.4 Perbandingan Teknis YAML dan JSON dalam OpenAPI.....	34
Tabel 2.6.5 Kode Konversi Sederhana YAML ke JSON.....	36
Tabel 3.2.1 Kebutuhan Fungsional	40
Tabel 3.2.2 Kebutuhan Non-Fungsional	41
Tabel 3.3.1 Contoh Code Snippet	45
Tabel 3.3.2 Contoh Respon AI.....	45
Tabel 3.4.1 Spesifikasi Teknis Gemma 3 12B-IT	46
Tabel 3.5.1 Spesifikasi Perangkat Keras.....	50
Tabel 3.5.2 Spesifikasi Perangkat Lunak	51
Tabel 3.5.3 Library dan Framework Pendukung	52
Tabel 4.2.1 Strategi Penanganan Error berdasarkan HTTP Status Code	81
Tabel 4.2.2 Script Testing Standalone	82
Tabel 4.2.3 Ekstraksi Path Parameters dengan Pattern Matching.....	95
Tabel 4.2.4 Pattern Matching untuk Response Detection	97
Tabel 4.2.5 Struktur Request Payload OpenRouter API.....	101
Tabel 4.2.6 Few-shot Learning	107
Tabel 4.2.7 Analisis Penggunaan Token Untuk Endpoint	110
Tabel 4.2.8 Ringkasan Fungsi Method OpenAPIGenerator	113
Tabel 4.2.9 Contoh users.js	115
Tabel 4.2.10 Hasil Dokumentasi openapi.yaml	116
Tabel 4.2.11 Struktur Data ValidationResult.....	120
Tabel 4.2.12 Skenario Pengujian Validasi	123

ABSTRAK

BAB I PENDAHULUAN

1.1 Latar Belakang

Dalam dua dekade terakhir, arsitektur perangkat lunak modern mengalami perubahan signifikan menuju paradigma mikrolayanan (microservices), yang mengedepankan modularitas, skalabilitas, dan fleksibilitas sistem. Setiap layanan dalam arsitektur ini berinteraksi melalui Representational State Transfer (REST) Application Programming Interface (API), sebuah antarmuka berbasis HTTP yang memungkinkan pertukaran data lintas sistem dengan format standar seperti JSON atau YAML [1]. Menurut Meshram [2], REST API telah menjadi tulang punggung komunikasi antar komponen sistem terdistribusi karena sifatnya yang ringan, independen terhadap platform, dan mudah diimplementasikan. Meskipun REST API telah menjadi standar de facto dalam rekayasa perangkat lunak modern, salah satu tantangan yang masih sering muncul adalah pemeliharaan dan pembaruan dokumentasi API. Dokumentasi API berperan penting dalam menjembatani pemahaman antara pengembang backend dan konsumen layanan (frontend, mobile, atau pihak ketiga). Tanpa dokumentasi yang jelas, proses integrasi antar sistem menjadi lambat dan rawan kesalahan [3]. Penelitian oleh Bondel et al. [4] menemukan bahwa sekitar 40% penyedia API publik mengalami masalah keterlambatan pembaruan dokumentasi karena perubahan kode yang tidak diikuti oleh pembaruan deskripsi endpoint. Kondisi ini dikenal sebagai documentation drift, yaitu ketidaksesuaian antara dokumentasi dan implementasi aktual API. Masalah lain yang sering muncul adalah documentation smells, yaitu ketidakkonsistenan atau kesalahan semantik dalam dokumentasi API yang mengurangi keterbacaan dan keandalan dokumentasi. Khan et al. [5] menegaskan bahwa meskipun banyak alat bantu dokumentasi telah dikembangkan, seperti Swagger, Redocly, dan API Blueprint, permasalahan dokumentasi tidak selalu dapat diatasi sepenuhnya. Sebagian besar alat tersebut bergantung pada anotasi

manual atau komentar dalam kode sumber, sehingga tetap menuntut intervensi manusia dalam memperbarui informasi ketika struktur kode berubah.

Sejumlah pendekatan telah diajukan untuk mengotomatisasi proses dokumentasi API. Solusi konvensional meliputi code parser dan documentation generator yang mengekstraksi informasi dari struktur kode dan komentar. Namun, pendekatan ini masih bersifat rule-based dan bergantung pada format penulisan tertentu [6]. Menurut Dhyani [7], keterbatasan pendekatan berbasis aturan ini membuatnya sulit menangani variasi sintaks dan konteks logika yang kompleks dalam proyek nyata. Oleh karena itu, muncul kebutuhan akan sistem dokumentasi otomatis yang tidak hanya membaca kode secara statis, tetapi juga memahami konteks dan perilaku kode secara semantik. Perkembangan teknologi Artificial Intelligence (AI), khususnya Large Language Model (LLM), membuka peluang baru untuk menghadirkan sistem dokumentasi yang lebih adaptif dan kontekstual. LLM seperti GPT, LLaMA, dan Gemma memiliki kemampuan memahami struktur dan makna kode melalui contextual embeddings yang dilatih dari miliaran token kode dan dokumentasi perangkat lunak [8]. Menurut Wang et al. [9], integrasi LLM dalam proses dokumentasi memungkinkan pembangkitan deskripsi API secara otomatis dengan tingkat relevansi yang lebih tinggi dibandingkan metode tradisional. Penelitian gDoc oleh Wang et al. [10] menunjukkan kemampuan model bahasa besar dalam menghasilkan dokumentasi API yang terstruktur sesuai spesifikasi industri seperti OpenAPI 3.1, dengan tingkat kesesuaian sintaks mencapai lebih dari 92%. Selain itu, Lazar et al. [11] menekankan pentingnya standard compliance dalam konteks dokumentasi API modern. OpenAPI sebagai standar terbuka (disahkan oleh Linux Foundation) memungkinkan interoperabilitas antar sistem dengan format dokumentasi yang konsisten. Namun, banyak penelitian sebelumnya masih terfokus pada pembuatan standalone documentation generator, bukan pada integrasi langsung ke dalam Integrated Development Environment (IDE) seperti Visual Studio Code. Padahal, integrasi langsung ke IDE dapat mengurangi beban kognitif pengembang dengan memungkinkan dokumentasi dihasilkan secara real-time saat menulis kode [12]. Visual Studio Code (VS Code)

merupakan IDE yang populer dengan lebih dari 14 juta pengguna aktif bulanan [13]. Ekosistemnya yang berbasis extension memungkinkan pengembang menambahkan fungsionalitas baru secara modular. Dalam beberapa tahun terakhir, sejumlah ekstensi AI seperti GitHub Copilot dan Tabnine telah menunjukkan efektivitas AI dalam mendukung aktivitas pengembangan perangkat lunak, termasuk penulisan kode otomatis dan deteksi kesalahan sintaks [14]. Namun, sejauh ini belum ada ekstensi yang secara khusus mengotomatisasi dokumentasi REST API dengan memanfaatkan model AI berbasis konteks kode dan menghasilkan keluaran sesuai standar OpenAPI secara langsung di VS Code.

Dari sisi metodologi pengembangan, pendekatan Agile Software Development, khususnya kerangka kerja Scrum, menjadi pilihan yang ideal untuk pengembangan plugin semacam ini. Menurut Schwaber dan Sutherland [15], Agile berfokus pada iterasi cepat, umpan balik berkelanjutan, dan fleksibilitas dalam menghadapi perubahan kebutuhan karakteristik yang sangat sesuai untuk proyek berbasis integrasi AI, di mana eksperimen dan penyesuaian model merupakan bagian penting dari siklus pengembangan. Berdasarkan tinjauan tersebut, penelitian ini mengusulkan perancangan dan implementasi plugin Visual Studio Code Extension untuk dokumentasi otomatis REST API berbasis AI dengan standar OpenAPI 3.1. Sistem ini akan memanfaatkan kemampuan inferensi model Gemma 3 12B-IT, yaitu model instruction-tuned large language model (LLM) generasi terbaru dari Google DeepMind yang dirilis pada tahun 2025. Model ini bersifat open-weight dan dirancang untuk mendukung tugas pemahaman kode (code comprehension), penjelasan logika program, serta pembuatan dokumentasi teknis berbasis konteks [16]. Inferensi model dilakukan melalui OpenRouter AI, sebuah platform inferensi terbuka yang menyediakan akses terpadu ke berbagai model besar, termasuk Gemma 3, tanpa kebutuhan infrastruktur GPU lokal [17]. Pemilihan Gemma 3 12B-IT didasarkan pada kombinasi kemampuan semantik tingkat lanjut dan efisiensi latensi inferensi, yang menjadikannya ideal untuk integrasi on-the-fly di lingkungan pengembangan terdistribusi seperti Visual Studio Code (VS Code). Secara konseptual, plugin yang dikembangkan dalam penelitian

ini akan berfungsi sebagai agen AI yang secara otomatis memindai file proyek (khususnya berbasis Node.js/Express), mengidentifikasi endpoint REST, parameter, dan response, kemudian menghasilkan dokumentasi dalam format OpenAPI 3.1 yang tervalidasi otomatis. Dengan memanfaatkan konektivitas inferensi berbasis OpenRouter, proses dokumentasi dapat diperbarui setiap kali kode mengalami perubahan, memastikan dokumentasi selalu selaras dengan implementasi aktual (synchronized documentation). Melalui pendekatan ini, penelitian diharapkan dapat menjawab tiga masalah utama dalam dokumentasi API modern: keterlambatan pembaruan, ketidaksesuaian sintaks dengan standar, dan minimnya integrasi antara dokumentasi dan lingkungan pengembangan. Dengan demikian, penelitian ini tidak hanya berkontribusi pada pengembangan perangkat lunak praktis berupa plugin VS Code, tetapi juga memberikan kontribusi ilmiah dalam bidang AI-assisted Software Documentation dan contextual code understanding. Secara akademis, hasil penelitian ini dapat menjadi acuan bagi pengembangan sistem dokumentasi cerdas di masa depan, serta memperluas penerapan LLM dalam domain software engineering automation.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah diuraikan sebelumnya, maka penelitian ini dirumuskan ke dalam beberapa pertanyaan penelitian sebagai berikut:

1. Bagaimana memanfaatkan model AI berbasis konteks kode untuk menghasilkan dokumentasi REST API secara otomatis dan kontekstual sesuai standar OpenAPI 3.1?
2. Bagaimana mengintegrasikan proses dokumentasi otomatis berbasis AI tersebut secara langsung ke dalam plugin Visual Studio Code Extension agar dapat digunakan oleh pengembang tanpa meninggalkan lingkungan kerja IDE?
3. Bagaimana mengukur tingkat akurasi, efisiensi, dan kepatuhan hasil dokumentasi terhadap standar OpenAPI 3.1 dibandingkan dengan dokumentasi manual atau generator berbasis aturan (rule-based)?

1.3 Batasan Masalah

Agar penelitian ini dapat dilaksanakan secara terarah, fokus, dan sesuai dengan ruang lingkup yang realistik, maka diperlukan batasan masalah dari sistem yang dikembangkan. Pembatasan ini juga dimaksudkan untuk memastikan bahwa setiap tahapan dalam penelitian sesuai dengan tujuan utama. Adapun batasan-batasan masalah dalam penelitian ini adalah sebagai berikut:

1. Penelitian ini dibatasi pada konteks perancangan dan implementasi plugin Visual Studio Code Extension yang berfungsi untuk menghasilkan dokumentasi REST API secara otomatis menggunakan model kecerdasan buatan (AI) Gemma 3 12B-IT melalui OpenRouter AI. Ruang lingkup sistem yang dikembangkan difokuskan pada integrasi antara model AI dan lingkungan pengembangan Integrated Development Environment (IDE) Visual Studio Code. Plugin ini dirancang untuk mendeteksi struktur endpoint dari proyek berbasis Node.js dan Express.js, lalu menghasilkan

dokumentasi dalam format standar OpenAPI versi 3.1 sebagaimana ditetapkan oleh OpenAPI Initiative. Penelitian ini tidak mencakup integrasi untuk framework selain Express.js, serta tidak membahas proses fine-tuning model AI atau pengembangan model baru. Dengan demikian, sistem yang dirancang hanya berfokus pada proses inferensi berbasis prompt engineering untuk menghasilkan deskripsi dokumentasi dari konteks kode sumber.

2. Proses pengembangan dibatasi sampai tahap proof of concept (PoC) dan evaluasi sistem, tanpa mencakup proses distribusi plugin ke marketplace Visual Studio Code atau implementasi dalam skala produksi. Penelitian ini juga tidak membahas aspek bisnis, pemasaran, maupun manajemen proyek di luar konteks pengembangan teknis.
3. Pengujian dilakukan pada lingkungan pengembangan lokal dengan konfigurasi perangkat keras menengah, menggunakan sistem operasi Windows 10. Evaluasi sistem mencakup empat dimensi utama: fungsionalitas plugin, validasi dokumentasi terhadap format OpenAPI menggunakan Swagger Validator dan OpenAPI Linter Tools, pengukuran performa inferensi model melalui response time dan penggunaan sumber daya, serta pengujian akurasi hasil dokumentasi dibandingkan dokumentasi manual yang dibuat oleh pengembang. Namun, penelitian ini tidak mengevaluasi aspek keamanan model AI di sisi server, kinerja OpenRouter AI dalam skala masif, maupun keandalan inferensi pada jaringan tidak stabil.
4. Penelitian ini membatasi penggunaan model Gemma 3 12B-IT dalam mode inference only, tanpa melibatkan pelatihan ulang (fine-tuning) atau modifikasi arsitektur model. Model ini dipilih karena kemampuannya dalam memahami konteks kode secara semantik serta efisiensinya pada platform low-latency inference milik Openrouter AI. Proses inferensi dilakukan dengan mengirimkan potongan kode (snippet) yang relevan melalui API, kemudian hasil keluaran berupa deskripsi endpoint, parameter, dan respons API diterjemahkan ke format OpenAPI. Fokus penelitian terletak pada

evaluasi kemampuan model dalam mengidentifikasi struktur logika kode dan menghasilkan dokumentasi yang sesuai standar, bukan pada eksplorasi arsitektur internal model atau teknik pembelajaran mesin yang mendasarinya.

5. Karena model AI dijalankan secara cloud-based, penelitian ini memastikan bahwa kode sumber yang dikirimkan untuk inferensi hanyalah bagian terbatas yang relevan, tanpa menyertakan data sensitif seperti API key, kredensial, atau informasi rahasia milik pengguna. Aspek keamanan jaringan, autentikasi API, enkripsi data, serta kepatuhan terhadap regulasi keamanan dan privasi data tidak dibahas secara mendalam. Penelitian ini lebih berfokus pada bagaimana integrasi model AI dapat meningkatkan efisiensi dan kualitas dokumentasi secara fungsional, bukan pada sistem keamanan data.

1.4 Tujuan

Tujuan utama dari penelitian ini adalah untuk merancang dan mengimplementasikan plugin Visual Studio Code Extension yang mampu menghasilkan dokumentasi REST API secara otomatis berdasarkan kode sumber, dengan memanfaatkan kecerdasan buatan (AI) dan mengikuti standar OpenAPI 3.1. Sistem ini diharapkan dapat memberikan solusi terhadap permasalahan klasik dalam pengembangan perangkat lunak modern, yaitu keterlambatan dan ketidaksesuaian dokumentasi dengan implementasi aktual kode sumber (documentation drift). Dengan mengintegrasikan model AI langsung ke dalam lingkungan pengembangan (Integrated Development Environment, IDE) Visual Studio Code, proses dokumentasi diharapkan dapat dilakukan secara real-time tanpa mengganggu alur kerja pengembang.

1.5 Manfaat

Penelitian ini diharapkan memberikan manfaat dalam tiga dimensi utama, yaitu manfaat akademis, praktis, dan teknis, baik bagi pengembangan ilmu pengetahuan maupun penerapannya di industri perangkat lunak modern.

1. Manfaat Akademis

Penelitian ini berkontribusi dalam pengembangan ilmu di bidang rekayasa perangkat lunak berbasis kecerdasan buatan (AI-assisted software engineering). Integrasi Large Language Model (LLM) seperti Gemma 3 12B-IT dalam proses dokumentasi kode memberikan arah baru dalam studi otomatisasi dokumentasi perangkat lunak, yang sebelumnya didominasi oleh pendekatan berbasis aturan (rule-based generators) atau parser statis.

2. Manfaat Praktis

Penelitian ini menghadirkan solusi nyata bagi pengembang perangkat lunak yang menghadapi tantangan dalam membuat dokumentasi REST API yang selalu terkini dan sesuai standar. Plugin yang dikembangkan dalam penelitian ini memungkinkan dokumentasi dihasilkan secara otomatis langsung di lingkungan Visual Studio Code, sehingga pengembang tidak perlu bergantung pada alat eksternal seperti Swagger Editor atau Postman untuk memperbarui dokumentasi. Proses ini mendukung continuous documentation, di mana setiap perubahan pada kode dapat langsung diikuti dengan pembaruan dokumentasi. Dengan cara ini, sistem yang dikembangkan tidak hanya mempercepat proses kerja, tetapi juga meningkatkan kualitas dokumentasi dan mengurangi risiko kesalahan komunikasi antar tim pengembang.

3. Manfaat Teknis

Penelitian ini menunjukkan penerapan konkret dari integrasi model kecerdasan buatan (AI) ke dalam ekosistem IDE modern melalui mekanisme plugin modular yang dapat digunakan ulang (reusable) dan diperluas (extensible), yang memungkinkan integrasi di masa depan dengan framework lain seperti Django REST Framework, Laravel API, atau

FastAPI. Selain itu, hasil penelitian ini dapat menjadi blueprint bagi pengembang ekstensi IDE lainnya untuk mengintegrasikan layanan AI generatif ke dalam proses dokumentasi, pengujian, atau analisis kode.

BAB II KAJIAN PUSTAKA

2.1 Penggunaan Model AI/LLM dalam Dokumentasi Kode

Perkembangan kecerdasan buatan (Artificial Intelligence/AI) dalam bidang software engineering telah memasuki era baru dengan kemunculan Large Language Models (LLMs) yang memiliki kemampuan memahami konteks bahasa alami sekaligus struktur sintaksis kode program. Model ini telah mengubah paradigma dalam pembuatan dokumentasi kode yang sebelumnya dilakukan secara manual atau berbasis aturan sederhana (rule-based documentation). Dengan kemampuan contextual understanding, semantic reasoning, dan code summarization, LLM kini digunakan untuk menghasilkan dokumentasi yang lebih deskriptif, relevan, dan mudah dipahami oleh pengembang [18]. Sebelum munculnya model bahasa besar, pendekatan dokumentasi otomatis biasanya menggunakan teknik static analysis atau pattern extraction, di mana sistem membaca struktur kode dan mengekstrak elemen seperti fungsi, parameter, dan komentar untuk membangun dokumentasi [19]. Namun, pendekatan ini terbatas dalam menangkap konteks semantik misalnya, maksud fungsional dari suatu algoritme atau hubungan antar modul. Hal ini mendorong transisi menuju sistem yang lebih cerdas berbasis pembelajaran mesin dan, kemudian, LLM. Sejak tahun 2020, muncul gelombang baru riset yang memanfaatkan model berbasis Transformer architecture untuk tugas pemahaman kode (code understanding) dan dokumentasi otomatis. Model seperti CodeBERT (Microsoft, 2020), CodeT5 (Salesforce Research, 2021), dan Codex (OpenAI, 2021) menunjukkan bahwa model AI mampu menghasilkan deskripsi fungsi, parameter, dan logika kode dengan akurasi semantik tinggi [20]. Model CodeT5, misalnya, menggunakan multi-task learning yang memungkinkan satu model digunakan untuk code summarization, code translation, dan code completion [21]. Dengan kata lain, model ini tidak hanya memahami kata demi kata dalam kode, tetapi juga konteks eksekusinya. Liu et al. [22] menegaskan bahwa LLM modern yang dilatih menggunakan data berskala besar mampu mengekstrak pola semantik dalam kode secara efisien, menghasilkan dokumentasi yang mendekati deskripsi buatan manusia, terutama ketika diterapkan dalam sistem inferensi seperti

OpenRouter yang menyediakan akses low-latency AI inference untuk model terbuka seperti Gemma 3. Demikian pula, Svyatkovskiy et al. [23] dari Microsoft Research mengembangkan sistem IntelliCode Compose, yaitu model berbasis Transformer yang dirancang untuk menyarankan potongan kode dan komentar secara otomatis di lingkungan pengembangan Visual Studio Code. Sistem ini memanfaatkan pendekatan contextual code generation dengan mempelajari pola sintaks dan semantik kode dalam skala proyek, sehingga dapat memberikan saran yang lebih relevan dan koheren terhadap konteks pengembang. Hasil penelitian mereka menunjukkan bahwa model berbasis pemahaman bahasa (language understanding model) dapat menghasilkan deskripsi dengan kohesi linguistik dan relevansi semantik yang lebih baik dibandingkan generator berbasis pola tradisional, sekaligus mempercepat proses penulisan dan dokumentasi kode.

Dalam konteks REST API, dokumentasi memiliki struktur yang lebih kompleks dibandingkan dokumentasi fungsi tunggal. Model AI harus memahami relasi antar endpoint, parameter, serta tipe data input-output untuk menghasilkan dokumentasi yang valid secara sintaks maupun semantik. Wang et al. [10] melalui sistem gDoc memperkenalkan konsep semantic code summarization, di mana model Transformer mempelajari hubungan antar endpoint REST API untuk menghasilkan dokumentasi dalam format OpenAPI secara otomatis. Hasil uji coba pada 500 proyek open source menunjukkan tingkat semantic accuracy sebesar 91,3%, mengungguli generator rule-based tradisional seperti Swagger Codegen. Model CodeLlama (Meta, 2023) dan StarCoder (HuggingFace, 2023) kemudian memperluas cakupan dengan memahami berbagai bahasa pemrograman (Python, JavaScript, Go, C#, dan TypeScript) sekaligus mengenali pola REST API. Hal ini membuka peluang bagi integrasi LLM langsung ke dalam lingkungan IDE seperti VS Code, sehingga dokumentasi dapat dihasilkan secara inline bersamaan dengan proses pengembangan. Untuk memperjelas posisi Gemma 3 12B-IT dalam lanskap penelitian, Tabel 2.1 berikut menyajikan perbandingan beberapa model utama yang digunakan untuk tugas dokumentasi dan pemahaman kode.

Tabel 2.1.1 Perbandingan Kemampuan Utama LLM

Model	Pengembang	Parameter	Kemampuan Utama
CodeBERT (2020)	Microsoft	125M	Pemahaman sintaks kode dan natural language
CodeT5 (2021)	Salesforce	220M	Code summarization dan translation
Codex (2021)	OpenAI	12B	Code generation dan dokumentasi
CodeLlama (2023)	Meta AI	13B	Code completion, comment generation
Gemma 3 12B-IT (2025)	Google DeepMind	12B	Instruction-tuned inference, pemahaman semantik kode

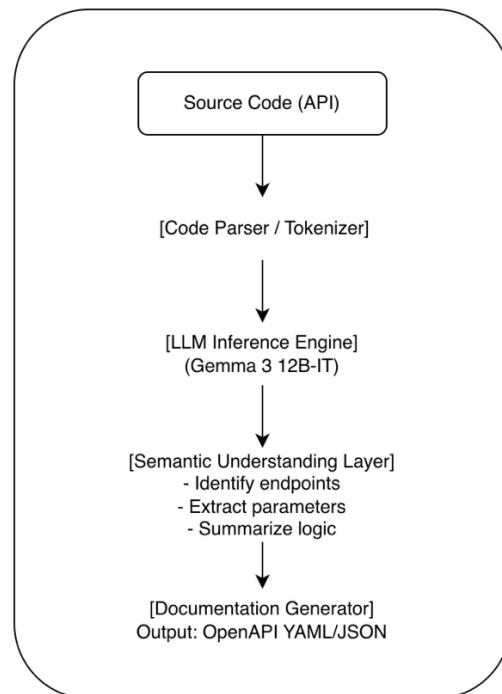
Tabel 2.1.2 Perbandingan Kelebihan dan Batasan LLM

Model	Kelebihan	Keterbatasan
CodeBERT (2020)	Efisien untuk code search dan classification	Tidak mampu melakukan deskripsi naratif penuh
CodeT5 (2021)	Akurasi semantik tinggi pada kode Python dan Java	Kurang optimal untuk dokumentasi API multi-endpoint
Codex (2021)	Kemampuan generalisasi lintas bahasa	Model tertutup, latensi tinggi
CodeLlama (2023)	Mendukung banyak bahasa, open-weight	Butuh sumber daya besar untuk inferensi lokal
Gemma 3 12B-IT (2025)	Open-weight, latensi rendah, fokus semantik kode	Tidak mendukung fine-tuning lokal

Dari tabel di atas, terlihat bahwa Gemma 3 12B-IT menempati posisi strategis antara model efisiensi tinggi (seperti CodeT5) dan model general-purpose (seperti Codex dan GPT-4). Keunggulan Gemma 3 terletak pada kemampuannya melakukan inferensi instruksional (instruction-tuned inference), yang memungkinkan pengguna memberikan perintah eksplisit seperti:

“Generate OpenAPI documentation for this Express route.”

Model kemudian menafsirkan potongan kode (misalnya rute REST API berbasis Express.js) dan menghasilkan keluaran berupa dokumentasi terstruktur sesuai spesifikasi OpenAPI 3.1, mencakup deskripsi endpoint, parameter, dan response schema. Proses inferensi dijalankan melalui OpenRouter AI, platform inferensi terbuka yang menyediakan akses terpadu ke model-model open-weight termasuk Gemma 3, dengan latensi rata-rata di bawah 80 milidetik per permintaan [16]. Kinerja ini cukup efisien untuk mendukung integrasi real-time di Visual Studio Code Extension, di mana setiap perubahan kode dapat langsung memicu pembaruan dokumentasi otomatis tanpa intervensi manual.



Gambar 2.1.1 Alur Logika Dokumentasi Otomatis LLM

Ilustrasi tersebut menekankan bahwa proses dokumentasi tidak hanya melibatkan ekstraksi sintaksis, tetapi juga pemahaman semantik yang lebih dalam oleh LLM. Model AI harus mampu mengenali hubungan antar endpoint, memahami tujuan fungsional, dan mengekspresikannya dalam format yang terstandar.

Studi terbaru menunjukkan peningkatan minat pada penerapan LLM dalam dokumentasi perangkat lunak. Beberapa survei terkini melaporkan bahwa penelitian mengenai dokumentasi kode dan code summarization semakin banyak memanfaatkan model generatif berskala besar untuk membantu pengembang memahami dan memelihara basis kode yang kompleks [24]. Tren ini didorong oleh meningkatnya kebutuhan otomatisasi dalam DevOps pipeline dan continuous documentation. Namun, penggunaan LLM tidak lepas dari tantangan. Pertama, masalah hallucination di mana model menghasilkan deskripsi yang tidak sepenuhnya akurat. Kedua, latensi inferensi dan biaya komputasi tinggi pada model besar seperti Codex dan GPT-4. Dalam konteks inilah, Gemma 3 12B-IT dan Openrouter AI menjadi solusi karena menawarkan efisiensi tinggi dengan kemampuan inferensi real-time. Selain itu, studi oleh Jorelle [25] yang secara khusus meneliti generasi dokumentasi API menggunakan LLM menyoroti pentingnya domain-specific datasets, karena dokumentasi API memiliki pola struktur dan gaya penulisan yang khas. Jorelle menunjukkan bahwa kualitas keluaran dapat ditingkatkan baik melalui fine-tuning pada korpus API tertentu maupun melalui perancangan prompt yang lebih terarah, dan dalam banyak kasus prompt engineering yang baik sudah cukup efektif tanpa harus selalu melakukan pelatihan ulang model secara penuh. Pendekatan inilah yang digunakan dalam penelitian ini, di mana Gemma 3 digunakan dalam mode inference-only dengan prompt yang dirancang untuk mengekstraksi deskripsi REST API secara efektif. Dengan demikian dapat di simpulkan bahwa model AI/LLM memiliki potensi besar untuk mengubah cara dokumentasi kode dibuat dan dipelihara. Evolusi dari model berbasis aturan menuju model semantik berbasis AI memungkinkan dokumentasi tidak hanya menjelaskan apa yang dilakukan kode, tetapi juga mengapa dan

bagaimana hal itu dilakukan. Integrasi model seperti Gemma 3 12B-IT ke dalam IDE seperti Visual Studio Code membuka era baru dokumentasi real-time yang kontekstual, efisien, dan adaptif terhadap perubahan kode.

2.2 Pendekatan Rule-based vs AI-based pada API Documentation

Proses dokumentasi Application Programming Interface (API) telah mengalami perubahan paradigma signifikan dalam dekade terakhir. Pendekatan tradisional berbasis aturan (rule-based), yang mengandalkan ekstraksi pola sintaksis dari kode sumber, kini mulai digantikan oleh sistem berbasis kecerdasan buatan (AI-based) yang mampu memahami konteks semantik kode dan menghasilkan dokumentasi secara otomatis. Pergeseran ini terjadi karena semakin kompleksnya sistem perangkat lunak modern dan meningkatnya kebutuhan untuk menjaga konsistensi dokumentasi terhadap perubahan kode sumber (code documentation synchronization) [24].

Pendekatan rule-based merupakan metode awal dalam otomatisasi dokumentasi, di mana sistem bekerja berdasarkan seperangkat aturan yang ditentukan secara eksplisit oleh pengembang. Biasanya, sistem ini menggunakan teknik static code analysis atau annotation parsing untuk mengekstraksi struktur kode seperti fungsi, parameter, dan nilai pengembalian. Contoh paling umum dari pendekatan ini adalah Swagger Codegen, Doxygen, dan RAML Parser, yang bekerja dengan menelusuri tanda khusus (annotation) di dalam kode sumber seperti @api, @param, atau @return [26]. Salah satu keunggulan utama pendekatan rule-based adalah kejelasan struktur hasil dokumentasi dapat diprediksi karena dihasilkan dari aturan yang deterministik. Selain itu, karena sistem tidak memerlukan data pelatihan, prosesnya efisien dan mudah diintegrasikan dalam pipeline DevOps. Misalnya, generator berbasis Swagger mampu menghasilkan spesifikasi OpenAPI hanya dengan memindai anotasi kode, tanpa perlu interpretasi semantik tambahan. Namun, kelemahan utama pendekatan ini adalah ketidakmampuannya menangkap konteks logika dan niat pengembang. Sistem rule-based hanya memahami “apa” yang tertulis, bukan “mengapa” kode ditulis

demikian. Akibatnya, dokumentasi yang dihasilkan sering kali bersifat dangkal (shallow documentation), berisi deskripsi sintaksis tanpa penjelasan perilaku. Bondel et al. [4] menunjukkan bahwa 37% dokumentasi API publik yang dihasilkan dengan metode rule-based tidak mencerminkan perilaku aktual endpoint setelah pembaruan versi kode. Dengan kata lain, pendekatan ini rentan terhadap documentation drift ketidaksesuaian antara dokumentasi dan implementasinya.

Sebaliknya, pendekatan AI-based mengandalkan model pembelajaran mesin, khususnya Large Language Models (LLMs), untuk memahami konteks dan struktur kode secara lebih mendalam. Model seperti CodeT5, Codex, dan Gemma 3 12B-IT mampu membaca potongan kode, mengenali pola pemanggilan fungsi, memahami logika internal, dan menghasilkan deskripsi dalam bahasa alami yang sesuai dengan maksud kode tersebut [16]. AI-based documentation bekerja berdasarkan prinsip semantic inference, bukan sekadar pattern recognition. Model AI dilatih menggunakan miliaran pasangan code-comment pairs, memungkinkan mereka mengenali hubungan semantik antara sintaks kode dan penjelasannya. Pendekatan ini tidak hanya mendeskripsikan API secara struktural, tetapi juga menjelaskan tujuan, alur data, dan dependensi antar endpoint. Sebagai contoh, sistem gDoc yang dikembangkan oleh Wang et al. mampu menghasilkan dokumentasi REST API lengkap (termasuk tipe parameter dan contoh respons) dengan tingkat akurasi semantik mencapai 91,3% jauh di atas metode rule-based yang rata-rata hanya mencapai 68% [10]. Keunggulan lain pendekatan AI-based adalah adaptivitas terhadap perubahan kode. Karena model mempelajari konteks, ia mampu memperbarui dokumentasi secara otomatis ketika struktur kode berubah, tanpa memerlukan aturan baru. Dalam implementasi modern seperti AI-assisted IDE plugin (misalnya, GitHub Copilot atau VS Code Extensions berbasis LLM), sistem bahkan dapat mendeteksi perubahan endpoint dan memperbarui dokumentasi secara real-time. Namun demikian, pendekatan AI-based juga memiliki tantangan tersendiri. Pertama, terdapat potensi hallucination di mana model AI menghasilkan deskripsi yang tampak meyakinkan tetapi tidak akurat secara teknis. Kedua, model besar seperti GPT-4 atau Codex membutuhkan sumber

daya komputasi tinggi, sehingga tidak cocok untuk integrasi lokal tanpa dukungan API inferensi berlatensi rendah. Oleh karena itu, penggunaan model Gemma 3 12B-IT melalui Openrouter AI menjadi relevan, karena menggabungkan efisiensi model instruction-tuned dengan performa low-latency inference, sehingga dokumentasi dapat dihasilkan langsung dalam IDE [16].

Tabel 2.2.1 Rule-based vs AI-based dalam Dokumentasi API

Aspek	Rule-based Documentation	AI-based Documentation
Prinsip kerja	Berdasarkan pola atau aturan eksplisit (annotation parsing)	Berdasarkan pembelajaran semantik dari data kode dan bahasa alami
Sumber informasi	Struktur sintaks kode dan anotasi	Konteks semantik kode, logika, dan relasi antar endpoint
Kebutuhan pelatihan	Tidak perlu model pelatihan	Memerlukan model LLM yang telah dilatih
Akurasi semantik	Rendah: fokus pada struktur	Tinggi: memahami makna dan perilaku kode
Ketahanan terhadap perubahan kode	Lemah: perlu pembaruan aturan manual	Kuat: mampu menyesuaikan konteks secara dinamis
Contoh alat	Swagger, Doxygen, RAML	CodeT5, Codex, Gemma 3 12B-IT, gDoc
Kelebihan utama	Cepat, deterministik, mudah diatur	Adaptif, kontekstual, deskriptif
Keterbatasan utama	Tidak memahami makna logika	Berpotensi menghasilkan deskripsi keliru (hallucination)

Cocok digunakan untuk	Dokumentasi statis dan API kecil	Dokumentasi dinamis, REST API kompleks, integrasi IDE
-----------------------	----------------------------------	---

Kedua pendekatan tersebut memiliki peran penting dalam lanskap dokumentasi API modern. Pendekatan rule-based tetap relevan untuk proyek berskala kecil atau sistem dengan kebutuhan dokumentasi yang sederhana. Namun, dalam konteks pengembangan REST API berskala besar dengan perubahan kode yang cepat dan kompleksitas endpoint tinggi, pendekatan AI-based menjadi jauh lebih efektif. Selain itu, integrasi AI-based documentation langsung ke IDE melalui plugin seperti VS Code Extension menghadirkan paradigma baru dalam continuous documentation. Pengembang dapat memperoleh dokumentasi secara real-time ketika menulis atau mengubah kode, tanpa berpindah ke alat eksternal. Dengan dukungan model efisien seperti Gemma 3 12B-IT dan Openrouter AI, proses ini tidak hanya menjadi otomatis, tetapi juga cepat dan hemat sumber daya.

2.3 REST API dan OpenAPI 3.1

REST API (Representational State Transfer Application Programming Interface) merupakan arsitektur komunikasi yang digunakan secara luas untuk pertukaran data antar sistem perangkat lunak. REST diperkenalkan oleh Roy Fielding pada tahun 2000 sebagai prinsip desain berbasis client-server yang menggunakan protokol HTTP sebagai media komunikasi utama [27]. REST API mengandalkan sumber daya (resources) yang direpresentasikan melalui URI (Uniform Resource Identifier), diakses menggunakan metode HTTP seperti GET, POST, PUT, dan DELETE. Keunggulan utama REST API adalah kesederhanaan, skalabilitas, dan interoperabilitas, yang menjadikannya standar de facto untuk sistem berbasis layanan (service-oriented architecture) dan aplikasi modern berbasis cloud. Setiap permintaan HTTP yang dikirim ke REST API dianggap stateless, artinya server tidak menyimpan informasi status klien antar permintaan. Pendekatan ini memungkinkan API berskala besar dijalankan secara efisien di

berbagai platform dan bahasa pemrograman. Secara umum, struktur REST API terdiri atas tiga komponen utama:

1. Endpoint (URI) – alamat yang mewakili sumber daya tertentu, misalnya /api/users.
2. HTTP Method – operasi yang dilakukan terhadap sumber daya, seperti GET (membaca data), POST (membuat data), PUT (memperbarui data), dan DELETE (menghapus data).
3. Request & Response – data dikirim dan diterima dalam format terstruktur seperti JSON atau XML.

Sebagai contoh:

Tabel 2.3.1 Contoh Struktur REST API

-	GET /api/products
-	Response: {
-	"id": 1,
-	"name": "Laptop X",
-	"price": 15000000
-	}

Namun, kompleksitas sistem modern yang terdiri dari puluhan hingga ratusan endpoint menyebabkan pengelolaan dokumentasi REST API menjadi tantangan tersendiri. Dokumentasi manual mudah ketinggalan dari perubahan kode sumber, sehingga dibutuhkan pendekatan otomatis dengan format standar yang dapat dibaca oleh manusia maupun mesin. Di sinilah OpenAPI Specification (OAS) berperan penting. OpenAPI merupakan format spesifikasi terbuka untuk mendeskripsikan REST API secara formal dan terstruktur. Spesifikasi ini awalnya dikenal sebagai Swagger Specification yang dikembangkan oleh SmartBear Software, kemudian diadopsi dan dikelola oleh OpenAPI Initiative (OAI) di bawah naungan Linux Foundation sejak tahun 2016 [8]. OpenAPI berfungsi sebagai contract antara server dan klien untuk mendeskripsikan dengan jelas endpoint,

parameter, tipe data, otentikasi, dan struktur respons. Dokumen OpenAPI biasanya ditulis dalam format YAML atau JSON, dan dapat digunakan untuk:

1. Menghasilkan dokumentasi interaktif (Swagger UI atau Redoc).
2. Membuat SDK (Software Development Kit) otomatis untuk berbagai bahasa.
3. Melakukan pengujian otomatis terhadap endpoint API.

OpenAPI versi 3.0.0 dirilis pada tahun 2017 dengan peningkatan signifikan dari versi 2.0 (Swagger), di antaranya dukungan untuk requestBody, callbacks, dan components. Namun, versi 3.1.0 yang dirilis oleh Linux Foundation pada tahun 2021 menjadi tonggak besar karena menyelaraskan struktur spesifikasi dengan JSON Schema Draft 2020-12 [26]. Perubahan utama pada OpenAPI 3.1 meliputi:

1. Keselarasan dengan JSON Schema

Sekarang semua bagian schema di OpenAPI sepenuhnya kompatibel dengan JSON Schema, memudahkan validasi struktur data API.

2. Dukungan untuk Webhooks

Fitur baru ini memungkinkan deskripsi API dua arah, di mana server dapat memanggil kembali klien secara otomatis.

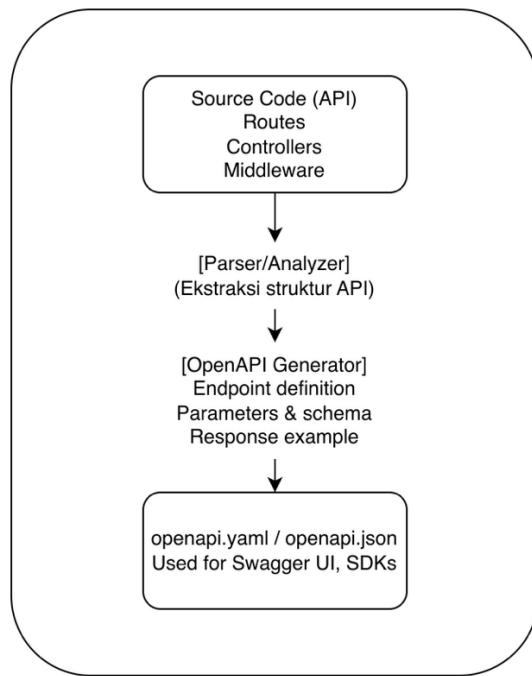
3. Penambahan Fields Baru

Properti seperti summary, description, dan example kini lebih fleksibel, mempermudah integrasi dengan alat dokumentasi otomatis.

4. Dukungan penuh terhadap format YAML dan JSON sebagai dua bentuk representasi resmi.

Secara formal, OpenAPI 3.1 bukan hanya dokumen spesifikasi, melainkan standar interoperabilitas API global. Inilah alasan mengapa penelitian ini menjadikan OpenAPI 3.1 sebagai target keluaran dokumentasi otomatis karena format ini menjamin dokumentasi yang dihasilkan dapat langsung divalidasi dan digunakan lintas platform.

Hubungan antara implementasi REST API dengan dokumen OpenAPI dapat digambarkan secara konseptual sebagai berikut:



Gambar 2.3.1 Kosep Implementasi REST API dengan Dokumen OpenAPI

Ilustrasi tersebut menunjukkan bagaimana OpenAPI bertindak sebagai lapisan dokumentasi formal yang memetakan setiap elemen REST API ke dalam format terstandar. Dalam penelitian ini, proses “Parser/Analyzer” digantikan oleh model AI (Gemma 3 12B-IT) yang melakukan inferensi semantik untuk mengisi bagian paths, components, dan schemas dalam dokumen OpenAPI. OpenAPI 3.1 dapat ditulis dalam YAML maupun JSON, dan keduanya memiliki fungsi yang identik. Adopsi OpenAPI dalam sistem dokumentasi otomatis berbasis AI membawa sejumlah keuntungan:

1. Konsistensi dan Validasi Otomatis

Dokumen OpenAPI dapat divalidasi menggunakan alat seperti Swagger Validator atau Redocly CLI, memastikan struktur dan tipe data sesuai standar industri [3].

2. Interoperabilitas Lintas Platform

Dengan format YAML/JSON yang terstandar, dokumentasi dapat langsung digunakan untuk membangun SDK, testing suite, dan antarmuka web interaktif tanpa modifikasi.

3. Integrasi dengan AI Models

Model seperti Gemma 2 9B-IT dapat memanfaatkan format ini untuk menghasilkan keluaran yang dapat dibaca dan digunakan mesin, mengubah hasil inferensi menjadi artefak nyata.

4. Dukungan untuk Continuous Documentation

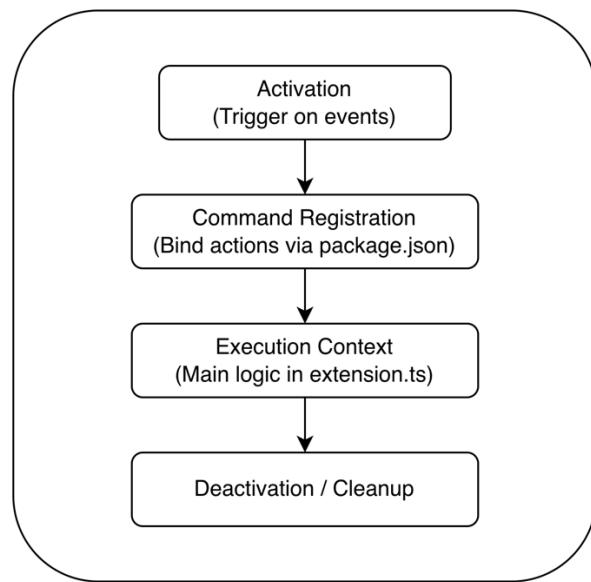
Melalui plugin VS Code, dokumentasi dapat diperbarui otomatis setiap kali pengembang melakukan perubahan kode.

Penelitian oleh Gowda dan Gowda [3] menunjukkan bahwa organisasi yang menerapkan OpenAPI dalam proses pengembangan API mengalami peningkatan efisiensi dokumentasi hingga 35%, serta pengurangan kesalahan integrasi antar tim sebesar 22%. Dengan demikian, penerapan OpenAPI tidak hanya relevan dari sisi teknis, tetapi juga berdampak langsung terhadap kualitas dan kolaborasi dalam siklus hidup perangkat lunak.

2.4 Arsitektur VS Code Extension API

Visual Studio Code (VS Code) merupakan salah satu Integrated Development Environment (IDE) yang banyak digunakan oleh pengembang perangkat lunak modern karena sifatnya yang ringan, lintas platform, dan memiliki ekosistem Extension API yang terbuka. Menurut Microsoft (2024), lebih dari 40.000 ekstensi aktif digunakan dalam ekosistem VS Code Marketplace, meliputi bidang pengujian, integrasi AI, debugging, hingga otomatisasi dokumentasi [28]. Extension API memungkinkan pengembang menambahkan fungsionalitas baru ke dalam VS Code tanpa mengubah kode inti aplikasi. Setiap ekstensi berjalan secara modular dan terisolasi dalam proses bernama Extension Host, yang berkomunikasi dengan VS Code melalui protokol JSON-RPC. Pendekatan ini memastikan ekstensi dapat dikembangkan secara independen, tetap stabil, dan aman dari gangguan

terhadap editor utama. Dalam konteks penelitian ini, VS Code Extension API digunakan untuk membangun plugin yang mengintegrasikan model AI Gemma 3 12B-IT melalui Openrouter AI, dengan tujuan menghasilkan dokumentasi otomatis REST API dalam format OpenAPI 3.1 secara langsung di dalam IDE. Sebuah ekstensi VS Code memiliki siklus hidup yang jelas, terdiri dari empat tahap utama sebagaimana ditunjukkan pada Gambar konseptual berikut:



Gambar 2.4.1 Tahapan Siklus Ekstensi VS Code

1. Activation: Tahap inisialisasi di mana ekstensi dimuat ke dalam memori. Proses ini dapat dipicu oleh berbagai peristiwa (activation events) seperti membuka jenis file tertentu (onLanguage:javascript), menjalankan perintah (onCommand:extension.generateDocs), atau membuka proyek dengan struktur REST API.
2. Command Registration: Setiap ekstensi memiliki file package.json yang mendefinisikan daftar perintah (command) dan konteks pemicunya. Misalnya, plugin dokumentasi otomatis akan memiliki perintah extension.autoGenerateDocs.
3. Execution Context: Tahap utama di mana logika ekstensi berjalan, biasanya ditulis di file extension.ts atau extension.js. Di sinilah ekstensi akan

membaca struktur proyek, memanggil API eksternal dan menampilkan hasil ke pengguna.

4. Deactivation: Tahap akhir di mana ekstensi menghentikan prosesnya, membersihkan memori, dan menutup koneksi aktif jika ada.

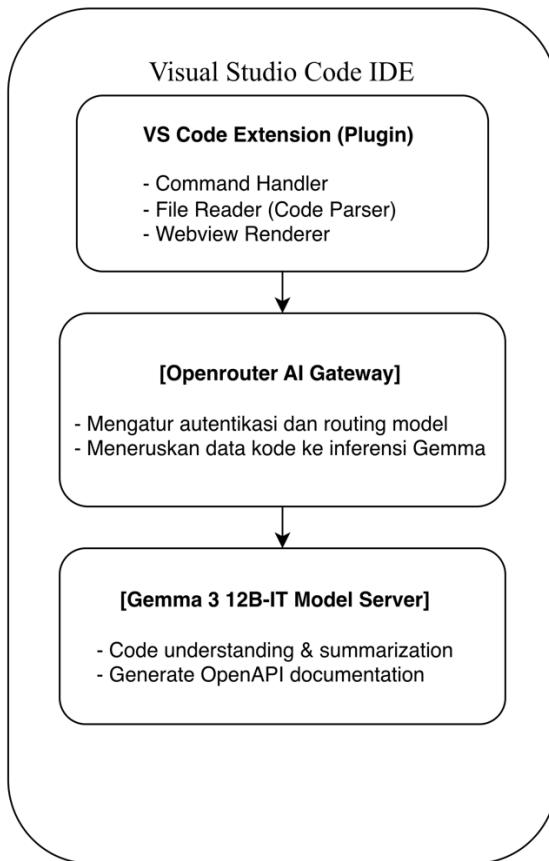
Setiap ekstensi VS Code terdiri dari tiga lapisan arsitektural utama dengan rincian di tabel 2.4 berikut:

Tabel 2.4.1 Struktur Komponen Ekstensi VS Code

Lapisan	Komponen Utama	Fungsi	Contoh Implementasi
Front-end (UI Layer)	Webview Panel, Status Bar, QuickPick, Notification	Menampilkan hasil dokumentasi, interaksi pengguna	Tampilan dokumentasi API dalam format YAML melalui webview
Core Logic Layer	Extension.ts, Command Handler, Event Listener	Menangani logika bisnis dan pemanggilan API eksternal	Fungsi generateDocumentation() yang memanggil Openrouter API
Integration Layer	HTTP Client (Axios, Fetch), File System API, Workspace API	Berinteraksi dengan sistem file dan layanan eksternal	Membaca file routes/, menulis openapi.yaml

Struktur modular ini memungkinkan pengembang menambahkan fungsi baru tanpa mengubah komponen lain. Sebagai contoh, modul HTTP Client dapat diganti tanpa mengganggu antarmuka pengguna atau logika plugin. Dalam penelitian ini, ekstensi dirancang untuk berkomunikasi langsung dengan

Openrouter AI, yang menyediakan layanan inferensi model Gemma 3 12B-IT. Arsitektur integrasi secara konseptual digambarkan sebagai berikut:



Gambar 2.4.2 Integrasi VS Code Extension Model AI

Dari gambar tersebut dapat di deskripsikan alur kerjanya sebagai berikut:

1. Pengguna menjalankan perintah "Generate API Documentation" di VS Code.
2. Plugin membaca struktur proyek (misalnya folder routes dan controllers) menggunakan vscode.workspace.fs.
3. Kode dikirim ke Openrouter AI melalui HTTP POST request yang berisi prompt untuk menghasilkan deskripsi API.
4. Model Gemma 3 12B-IT melakukan inferensi dan menghasilkan keluaran berupa teks dalam format OpenAPI YAML/JSON.

5. Plugin menulis hasil tersebut ke file openapi.yaml dan menampilkan di webview dalam IDE.

Pendekatan ini memungkinkan dokumentasi API dihasilkan secara real-time, tanpa perlu berpindah aplikasi atau menggunakan alat eksternal seperti Swagger Editor. Beberapa alasan utama mengapa VS Code Extension dipilih sebagai platform implementasi sistem dokumentasi otomatis ini antara lain:

1. Integrasi Langsung dengan Lingkungan Pengembang

VS Code menyediakan antarmuka interaktif yang memungkinkan plugin beroperasi langsung di dalam proyek yang sedang dikerjakan, sehingga hasil dokumentasi dapat diakses instan oleh pengembang [23].

2. Ketersediaan API yang Kaya

Melalui Extension API, pengembang dapat mengakses sistem file, terminal, serta event hook. Misalnya, vscode.workspace.onDidChangeTextDocument() dapat digunakan untuk memperbarui dokumentasi otomatis setiap kali file disimpan.

3. Dukungan Arsitektur Modular dan Asinkron

Model ekstensi VS Code mendukung asynchronous execution berbasis Promise, yang ideal untuk pemanggilan API eksternal seperti Openrouter AI.

4. Kemudahan Visualisasi Output AI

Dengan dukungan webview, hasil inferensi AI dapat dirender dalam bentuk HTML interaktif, memungkinkan pengembang melihat dokumentasi OpenAPI secara visual.

5. Ekosistem AI Developer Tools yang Berkembang Pesat

Menurut Strapi Team (2025), lebih dari 20 ekstensi VS Code berbasis AI telah dirilis antara 2023–2025, menandakan pergeseran paradigma ke arah AI-augmented IDEs [13].

Secara keseluruhan, arsitektur VS Code Extension API memberikan fondasi teknis yang kuat untuk pengembangan sistem dokumentasi otomatis berbasis AI. Integrasi langsung dengan workspace, dukungan arsitektur asinkron, serta kemampuan webview rendering menjadikan VS Code sebagai platform yang ideal untuk mengimplementasikan model Gemma 3 12B-IT melalui OpenRouter AI. Kombinasi ini memungkinkan dokumentasi REST API dihasilkan secara kontekstual, cepat, dan interaktif tanpa meninggalkan lingkungan pengembangan utama. Dengan demikian, ekstensi ini tidak hanya berperan sebagai alat bantu dokumentasi, tetapi juga sebagai agen AI cerdas dalam siklus Agile development, yang menjaga sinkronisasi berkelanjutan antara kode dan dokumentasi teknis.

2.5 Model AI Gemma 3 12B-IT dan Integrasi Openrouter AI

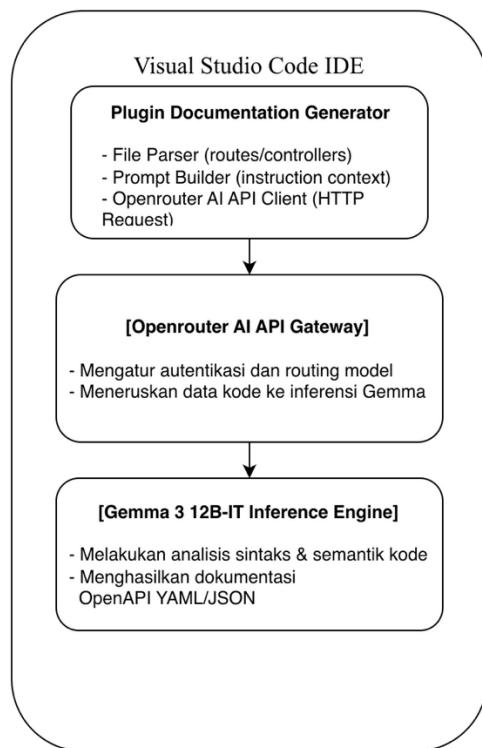
Model Gemma 3 12B-IT merupakan generasi ketiga Large Language Model (LLM) yang dikembangkan oleh Google DeepMind pada tahun 2025 [16]. Model ini adalah penerus langsung Gemma 2 9B-IT dengan peningkatan kapasitas hingga 12 miliar parameter serta penyempurnaan proses instruction-tuning untuk menghasilkan inferensi yang lebih kontekstual, akurat, dan efisien. Gemma 3 12B-IT dirancang sebagai open-weight instruction-tuned decoder-only transformer yang dioptimalkan untuk skenario pemahaman kode (code understanding) dan dokumentasi teknis secara otomatis. Berbeda dengan model umum seperti GPT-4 yang bersifat closed-weight, Gemma 3 bersifat terbuka, sehingga dapat dijalankan melalui berbagai platform inferensi publik seperti OpenRouter AI tanpa lisensi komersial tertutup. OpenRouter AI menyediakan API berbasis HTTP yang memungkinkan permintaan inferensi (prompt requests) dikirim secara langsung dari VS Code Extension, dengan latensi rata-rata < 80 milidetik per respons [17]. Platform ini mendukung skala penggunaan dari individu hingga multi-tenant server melalui akses token yang terautentikasi. Gemma 3 12B-IT menggunakan arsitektur transformer decoder-only dengan teknik penguatan pada lapisan attention (multi-head sparse attention) dan positional embedding dinamis. Optimasi tersebut memungkinkan pemrosesan kode berpanjang hingga 64 konteks blok (~32 k token) tanpa degradasi kinerja. Model ini menggabungkan dua kompetensi utama:

1. Code Comprehension Enhancement: menangkap struktur logika dan hubungan dependensi antarfungsi dalam kode.
2. Instruction-Based Text Generation: menghasilkan deskripsi dan dokumentasi kode dalam bahasa alami berdasarkan perintah eksplisit.

Dalam penelitian ini, Gemma 3 12B-IT digunakan untuk menganalisis struktur REST API pada proyek berbasis Node.js/Express dan menghasilkan dokumentasi otomatis dalam format OpenAPI 3.1 melalui proses inferensi yang dilakukan oleh OpenRouter AI. Integrasi model ke plugin VS Code dilakukan melalui tiga komponen utama:

1. Code Parser Module: mengekstrak struktur REST endpoint dari berkas kode (Node.js/Express).
2. OpenRouter Inference Handler: mengirimkan prompt berisi instruksi ke endpoint <https://openrouter.ai/api/v1/chat/completions> menggunakan API key pengembang.
3. Documentation Generator: menerjemahkan hasil inferensi menjadi file openapi.yaml dan openapi.json.

Langkah-langkah proses inferensi dapat digambarkan sebagai berikut (Gambar 2.5.1):



Gambar 2.5.1 Alur Integrasi Gemma–Openrouter AI dalam Plugin VS Code

1. Plugin menganalisis struktur proyek dan mengambil potongan kode REST API.
2. Potongan kode dikirim ke OpenRouter API dengan prompt instruksional.
3. Model Gemma 3 12B-IT melakukan analisis semantik kode dan membentuk deskripsi endpoint.
4. Hasil keluaran berupa spesifikasi OpenAPI (3.1) ditulis ke file dokumentasi.
5. Plugin menampilkan hasil di webview VS Code secara real-time.

Keunggulan Integrasi Gemma 3 12B-IT bersama OpenRouter AI memiliki beberapa keunggulan dibandingkan solusi lain:

1. Inferensi Berkinerja Tinggi & Latensi Rendah: Waktu respons 80 ms rata-rata cocok untuk sinkronisasi dokumentasi langsung dalam IDE.

2. Open-Weight dan Transparan: Bobot model terbuka memungkinkan audit dan reproduksi hasil oleh peneliti.
3. Aksesibilitas Gratis melalui OpenRouter: Mendukung tier penggunaan tanpa biaya untuk pengujian dan riset terbatas.
4. Konsistensi Format Standar: Keluaran mengikuti spesifikasi OpenAPI 3.0–3.1 dan JSON Schema Draft 2020-12.

Kombinasi Gemma 3 12B-IT dan OpenRouter AI menyediakan fondasi bagi pengembangan VS Code Extension yang dapat menghasilkan dokumentasi REST API secara otomatis, relevan dengan kode sumber aktual, serta efisien dalam penggunaan komputasi. Pendekatan ini mendorong arah baru menuju AI-assisted software documentation yang terbuka dan terdesentralisasi.

2.6 Format Output YAML/JSON pada OpenAPI

Dokumentasi OpenAPI Specification (OAS) mendefinisikan cara mendeskripsikan REST API secara formal dan dapat dibaca oleh mesin maupun manusia. Sejak versi 3.0 hingga 3.1, OpenAPI mendukung dua format utama penyimpanan: YAML (Yet Another Markup Language) dan JSON (JavaScript Object Notation) [8]. Kedua format ini menggunakan struktur hierarkis berbasis objek (object-based hierarchy) yang terdiri atas elemen info, servers, paths, dan components. Dalam konteks penelitian ini, model AI Gemma 3 12B-IT menghasilkan dokumentasi REST API secara otomatis dalam kedua format YAML dan JSON agar kompatibel dengan kebutuhan manusia (developer readability) maupun mesin (tool interoperability).

Baik YAML maupun JSON mengikuti struktur inti yang sama, sebagaimana dijelaskan dalam spesifikasi OpenAPI 3.1.0 oleh Linux Foundation (2024) [8]. Struktur umum tersebut adalah:

Tabel 2.6.1 Struktur Umum OpenAPI

-	
---	--

```

- openapi: "3.1.0"
info:
-   title: (Judul API)
-   version: (Versi API)
-   description: (Deskripsi API)
servers:
-   - url: (Base URL)
paths:
-   /endpoint:
      (Metode HTTP):
-       summary: (Ringkasan)
-       description: (Deskripsi)
parameters:
responses:
components:
schemas:
-   (Definisi objek data)
-
```

Perbedaan antara YAML dan JSON hanya terletak pada sintaksnya dimana YAML menggunakan indentasi dan tanda titik dua (:), sedangkan JSON menggunakan tanda kurung kurawal ({}) dan tanda kutip ganda. Dalam praktiknya YAML lebih disukai dalam dokumentasi manual karena lebih mudah dibaca manusia dan tidak terlalu padat secara visual. Berikut contoh dasar struktur OpenAPI 3.1 dalam format YAML:

Tabel 2.6.2 Struktur OpenAPI 3.1 dengan Format YAML

```

- openapi: 3.1.0
info:
-   title: "Produk API"
-   version: "1.0.0"
-   description: "Dokumentasi REST API otomatis menggunakan
OpenAPI 3.1"
servers:
-   - url: "https://api.example.com"
paths:
-   /products:
get:
-   summary: "Mendapatkan daftar produk"
responses:
-   '200':
       description: "Daftar produk berhasil diambil"
content:
-
```

```
-      application/json:
-        schema:
-          type: array
-          items:
-            $ref: '#/components/schemas/Product'
-    components:
-      schemas:
-        Product:
-          type: object
-          properties:
-            id:
-              type: integer
-            name:
-              type: string
-            price:
-              type: number
-          required:
-            - id
-            - name
-            - price
```

Format ini sangat cocok digunakan di Swagger UI, Redoc, atau langsung ditampilkan pada VS Code Webview. YAML juga mendukung komentar menggunakan #, yang membantu pengembang memberikan catatan tambahan.

Sementara itu, JSON lebih efisien untuk proses komputasi dan integrasi dengan alat otomatis. Contoh format yang sama ditulis ulang dalam JSON adalah sebagai berikut:

Tabel 2.6.3 Struktur OpenAPI 3.1 dengan Format JSON

<pre> { "openapi": "3.1.0", "info": { "title": "Produk API", "version": "1.0.0", "description": "Dokumentasi REST API otomatis menggunakan OpenAPI 3.1" }, "servers": [{ "url": "https://api.example.com" }], "paths": { "/products": { "get": { "summary": "Mendapatkan daftar produk", "responses": { "200": { "description": "Daftar produk berhasil diambil", "content": { "application/json": { "schema": { "type": "array", "items": { "\$ref": "#/components/schemas/Product" } } } } } } } } }, "components": { "schemas": { "Product": { "type": "object", "properties": { "id": { "type": "integer" }, "name": { "type": "string" }, "price": { "type": "number" } }, "required": ["id", "name", "price"] } } } } </pre>
--

Format JSON sangat cocok digunakan untuk integrasi lintas sistem, misalnya:

1. Mengimpor ke Postman untuk pengujian otomatis.
2. Diproses oleh CI/CD pipeline (Jenkins, GitLab CI).
3. Digunakan oleh library API client generator

Untuk menentukan format yang tepat, tabel berikut menjelaskan perbedaan fundamental antara YAML dan JSON pada konteks OpenAPI:

Tabel 2.6.4 Perbandingan Teknis YAML dan JSON dalam OpenAPI

Aspek	YAML	JSON
Keterbacaan manusia	Sangat mudah dibaca dan ringkas	Lebih padat dan berstruktur kaku
Dukungan komentar	Ya (# komentar)	Tidak mendukung komentar
Parsing ke dalam program	Perlu parser tambahan (js-yaml)	Dapat langsung digunakan di hampir semua bahasa pemrograman
Ukuran file	Lebih kecil karena tanpa tanda kutip dan kurung	Sedikit lebih besar
Validasi schema	Lebih fleksibel, tetapi bergantung pada library parser	Terintegrasi langsung dengan JSON Schema Draft 2020-12
Interoperabilitas mesin	Kurang cocok untuk pipeline otomatis	Sangat baik untuk API generator dan CI/CD
Format output AI (Gemma 3)	Digunakan untuk hasil yang dibaca di IDE (webview)	Digunakan untuk validasi dan pipeline otomatis

Contoh penggunaan utama	Swagger UI, Redoc	Postman, SDK Generator, Automated Testing Tools
-------------------------	-------------------	---

Dalam sistem yang dibangun pada penelitian ini, kedua format digunakan secara bersamaan sesuai konteks penggunaannya:

1. YAML digunakan untuk antarmuka pengembang

Plugin VS Code menampilkan hasil dokumentasi otomatis dalam format YAML karena lebih readable dan interaktif. YAML juga lebih sesuai dengan tampilan syntax highlighting di editor.

2. JSON digunakan untuk sistem validasi dan integrasi pipeline

Setelah dokumentasi dihasilkan dalam YAML, plugin akan mengonversinya ke JSON untuk memastikan validitas spesifikasi terhadap OpenAPI Validator serta untuk digunakan oleh sistem eksternal seperti SwaggerHub atau Postman.

Konversi dilakukan secara otomatis menggunakan pustaka seperti js-yaml (Node.js). Berikut contoh kode konversi sederhana yang digunakan pada plugin:

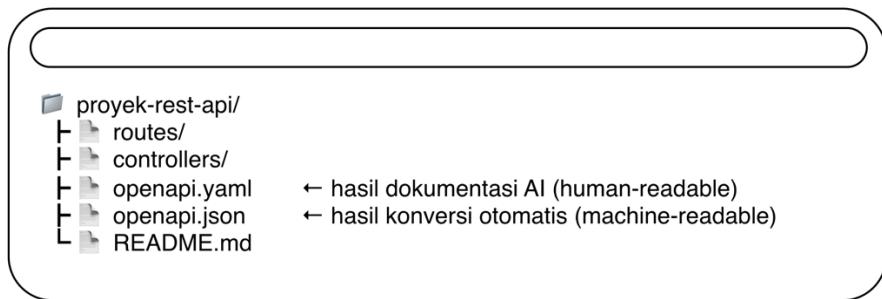
Tabel 2.6.5 Kode Konversi Sederhana YAML ke JSON

-	import yaml from 'js-yaml';
-	import fs from 'fs';
-	const yamlContent = fs.readFileSync('openapi.yaml', 'utf8');
-	const jsonContent = yaml.load(yamlContent);
-	fs.writeFileSync('openapi.json', JSON.stringify(jsonContent, null, 2));

Kode tersebut memastikan plugin selalu menghasilkan dua versi file dokumentasi (openapi.yaml dan openapi.json) setiap kali proses inferensi AI selesai dijalankan. Pendekatan dual output (menghasilkan dua format sekaligus) memiliki beberapa manfaat strategis:

1. Konsistensi antar alat: YAML digunakan untuk dokumentasi visual, sedangkan JSON digunakan untuk validasi otomatis.
2. Fleksibilitas penggunaan: Developer bebas memilih format sesuai preferensi atau kebutuhan proyek.
3. Kompatibilitas maksimal: Beberapa alat seperti Redocly dan SwaggerHub hanya mendukung YAML, sementara alat lain seperti Postman dan CI validator hanya menerima JSON.
4. Dukungan interoperabilitas AI pipeline: Model AI (Gemma 3 12B-IT) mampu menghasilkan format JSON langsung untuk keperluan post-processing di Openrouter AI.

Untuk gambaran output yang dihasilkan oleh plugin VS Code pada sistem penelitian ini di sajikan pada gambar berikut:



Gambar 2.6.1 Output File Plugin VS Code

Pada tahap post-inference, sistem otomatis melakukan validasi terhadap `openapi.json` menggunakan pustaka OpenAPI Parser untuk memastikan:

1. Semua endpoint memiliki response code yang valid.
2. Skema data konsisten dengan definisi components/schemas.
3. Format JSON sesuai dengan standar Draft 2020-12 yang diadopsi oleh OpenAPI 3.1.

Proses validasi ini dijalankan segera setelah model Gemma 3 12B-IT menghasilkan keluaran melalui inferensi OpenRouter AI, memastikan integritas dokumentasi tetap terjaga secara semantik dan struktural. Dengan dukungan proses inferensi dari Gemma 3 12B-IT melalui OpenRouter AI, sistem ini tidak hanya menghasilkan dokumentasi yang sesuai standar OpenAPI 3.1, tetapi juga menjamin bahwa setiap pembaruan kode sumber segera tercermin dalam dokumentasi secara real-time.

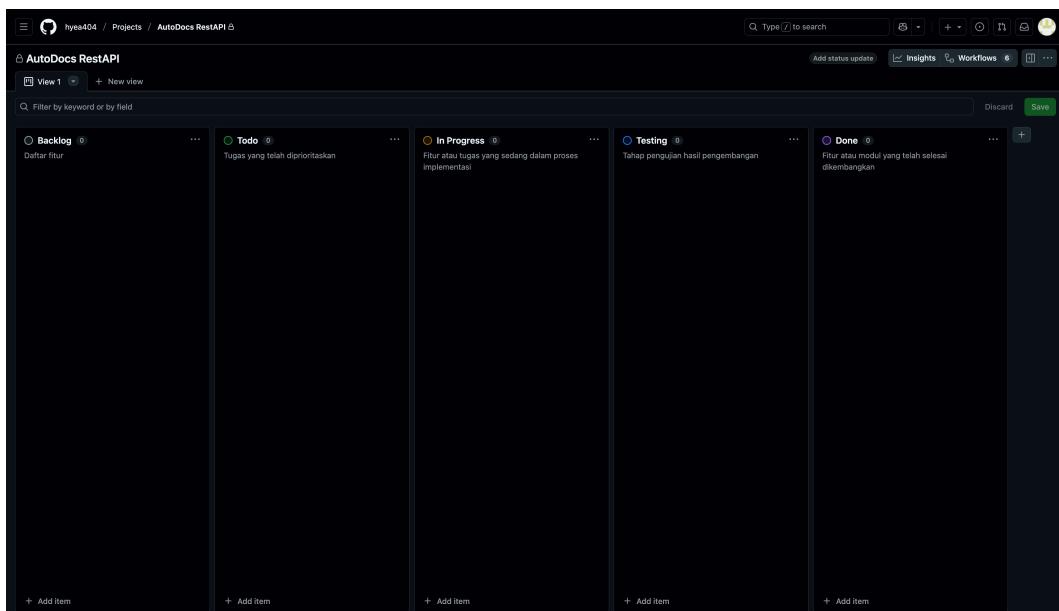
BAB III METODE PENGEMBANGAN SISTEM

3.1 Metodologi Pengembangan

Metodologi pengembangan yang digunakan dalam penelitian ini adalah metode Agile dengan pendekatan Kanban. Metode ini dipilih karena sesuai dengan karakteristik proyek yang dikembangkan secara individual, bersifat iteratif, dan berorientasi pada continuous delivery. Dalam konteks pengembangan perangkat lunak berbasis AI seperti VS Code Extension untuk dokumentasi REST API otomatis, metode Kanban memberikan fleksibilitas tinggi dalam mengelola perubahan fitur, eksperimen inferensi model, dan penyesuaian arsitektur sistem seiring proses penelitian berlangsung,. Pendekatan Agile Kanban berfokus pada visualisasi alur kerja (workflow visualization), pembatasan jumlah pekerjaan yang sedang dikerjakan (Work in Progress/WIP limit), serta peningkatan berkelanjutan (continuous improvement) [29]. Secara umum, Kanban dalam pengembangan perangkat lunak menggunakan alur sederhana seperti To Do, In Progress, Review/Testing, Done. Pada penelitian ini, alur tersebut diadaptasi menjadi lima kolom agar lebih sesuai dengan kebutuhan proyek individual dan eksperimen AI, yaitu:

1. Backlog: daftar fitur, modul, dan kebutuhan sistem yang akan dikembangkan, seperti analisis kode REST API, integrasi OpenRouter API, dan pembuatan dokumentasi OpenAPI.
2. To Do: tugas yang telah diprioritaskan untuk dikerjakan pada iterasi berikutnya.
3. In Progress: fitur atau modul yang sedang dalam proses implementasi, misalnya perancangan prompt AI atau penyusunan parser Express.js.
4. Testing / Validation: tahap pengujian hasil pengembangan, termasuk validasi hasil dokumentasi OpenAPI terhadap kode sumber.
5. Done: fitur atau modul yang telah selesai dikembangkan dan diverifikasi fungsionalitasnya di VS Code.

Model Kanban dipilih karena tidak memerlukan pembagian peran tim secara kompleks seperti Scrum yang memerlukan Product Owner atau Scrum Master, sehingga lebih efisien untuk pengembangan oleh satu orang peneliti/pengembang. Setiap aktivitas di papan Kanban dikelola menggunakan alat bantu GitHub Projects, yang memudahkan pemantauan progres serta pencatatan catatan teknis dari setiap iterasi.



Gambar 3.1.1 Ilustrasi Papan Kanban pada Pengembangan Sistem

Metode Agile Kanban juga memfasilitasi continuous documentation, yaitu pendekatan di mana dokumentasi sistem dan API dihasilkan secara paralel dengan pengembangan kode. Hal ini sejalan dengan konsep AI-assisted documentation pipeline yang diimplementasikan melalui model Gemma 3 12B-IT via OpenRouter AI, di mana setiap perubahan kode dapat langsung memicu pembaruan dokumentasi secara otomatis.

3.2 Analisis Kebutuhan

Analisis kebutuhan dilakukan untuk mendefinisikan fungsi utama, batasan, dan kriteria performa sistem VS Code Extension untuk Dokumentasi Otomatis

REST API dengan Standar OpenAPI. Tahapan ini menjadi dasar dalam proses perancangan sistem dan pembagian tugas pada papan Agile Kanban, sehingga setiap komponen dikembangkan sesuai dengan tujuan dan kebutuhan pengguna akhir. Sistem yang dikembangkan harus mampu menghasilkan dokumentasi REST API secara otomatis dengan memanfaatkan model AI Gemma 3 12B-IT yang diakses melalui OpenRouter AI, serta menghasilkan keluaran dalam format OpenAPI 3.1 (YAML/JSON) secara langsung dari kode sumber berbasis Node.js/Express.

3.2.1 Kebutuhan Fungsional

Kebutuhan fungsional menjelaskan fitur dan perilaku yang harus dimiliki oleh sistem agar dapat menjalankan fungsinya dengan benar. Berdasarkan analisis terhadap proses pengembangan dan integrasi model AI, kebutuhan fungsional sistem adalah sebagai berikut:

Tabel 3.2.1 Kebutuhan Fungsional

Kode	Deskripsi	Penjelasan Teknis
F1	Analisis Struktur Kode REST API	Sistem mampu memindai direktori proyek Node.js untuk menemukan file berisi rute (routes/*.js) dan mengidentifikasi endpoint REST (GET, POST, PUT, DELETE).
F2	Ekstraksi Parameter dan Respons API	Sistem mengidentifikasi parameter dari req.params, req.query, dan req.body, serta jenis respons melalui objek res.status() dan res.send().
F3	Inferensi Deskripsi Otomatis via OpenRouter AI	Potongan kode dikirim ke model Gemma 3 12B-IT melalui endpoint API OpenRouter (https://openrouter.ai/api/v1/chat/completions) untuk menghasilkan deskripsi endpoint.

F4	Pembuatan File Dokumentasi OpenAPI	Sistem menghasilkan dokumentasi REST API dalam format OpenAPI 3.1 (YAML dan JSON), mencakup informasi paths, parameters, responses, dan schemas.
F5	Pembaruan Dokumentasi Otomatis (Auto Sync)	Setiap kali kode diubah, plugin mendekripsi perubahan dan memperbarui file openapi.yaml dan openapi.json secara otomatis (synchronized documentation).
F6	Tampilan Dokumentasi di VS Code Webview	Plugin menampilkan hasil dokumentasi di panel VS Code menggunakan Markdown renderer atau Swagger UI embed.
F7	Validasi Struktur OpenAPI	Sistem memverifikasi hasil dokumentasi menggunakan library validasi (misalnya openapi-schema-validator) agar sesuai standar OpenAPI 3.1.
F8	Pengaturan API Key dan Preferensi Model	Pengguna dapat mengatur API key OpenRouter dan memilih versi model AI seperti Gemma 3 12B-IT melalui panel pengaturan ekstensi.
F9	Ekspor dan Impor Dokumentasi	Plugin menyediakan fitur ekspor dokumentasi ke format YAML/JSON dan impor kembali untuk validasi atau penggabungan manual.

3.2.2 Kebutuhan Non-Fungsional

Kebutuhan non-fungsional mendefinisikan karakteristik kualitas yang harus dipenuhi sistem agar berfungsi secara optimal dan dapat diandalkan. Kebutuhan Non-fungsional sistem adalah sebagai berikut:

Tabel 3.2.2 Kebutuhan Non-Fungsional

Kode	Kategori	Deskripsi

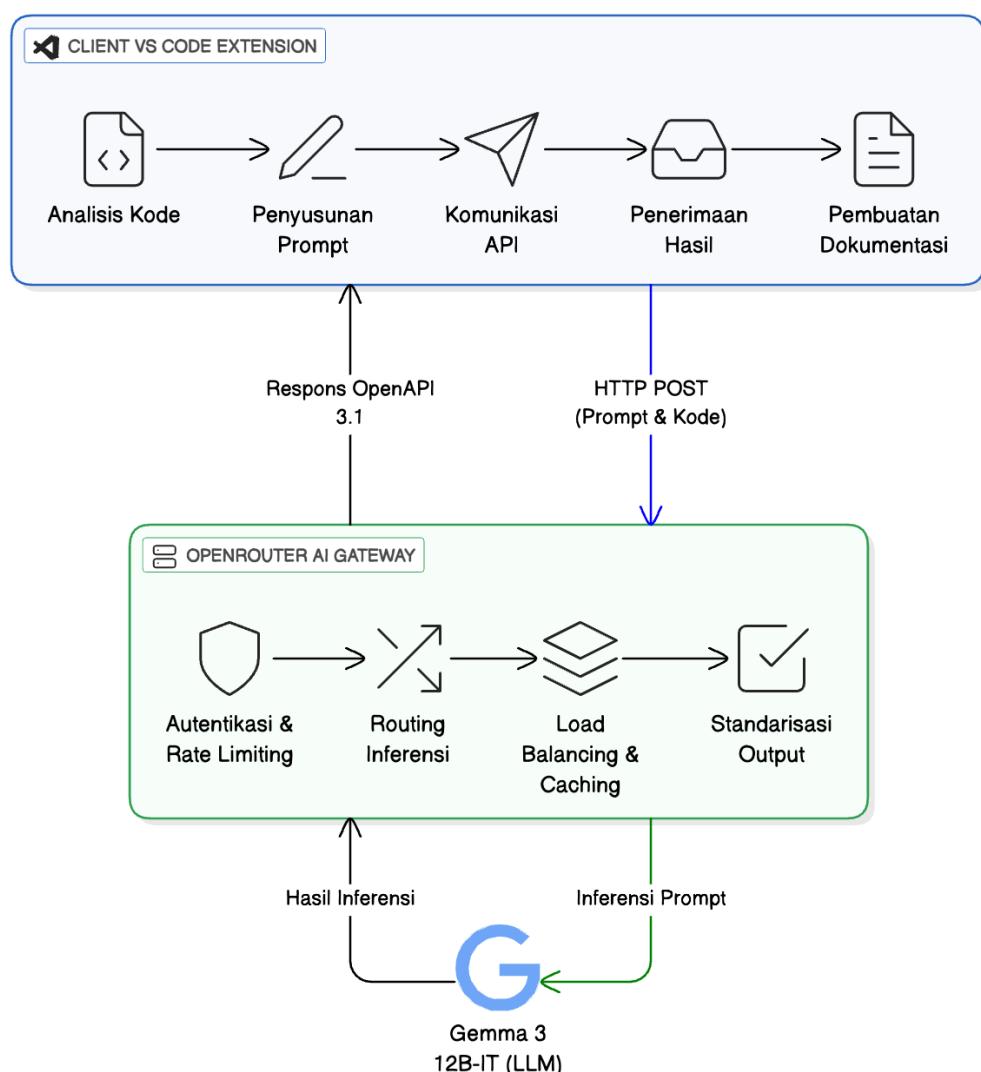
NF1	Kinerja (Performance)	Waktu respons inferensi dari OpenRouter AI tidak boleh melebihi 60 detik per permintaan, termasuk pengiriman prompt dan penerimaan hasil dokumentasi.
NF2	Reliabilitas (Reliability)	Plugin harus mampu bekerja secara stabil meskipun koneksi jaringan fluktuatif. Jika gagal melakukan inferensi, sistem akan menyimpan permintaan untuk retry otomatis.
NF3	Usability (Kemudahan Penggunaan)	Antarmuka harus terintegrasi langsung dalam VS Code, dengan tampilan intuitif dan tidak mengganggu proses pengembangan pengguna.
NF4	Kompatibilitas (Compatibility)	Ekstensi harus mendukung minimal VS Code versi 1.80 ke atas dan kompatibel dengan proyek berbasis Node.js versi 16+.
NF5	Keamanan (Security)	Kunci API OpenRouter disimpan secara lokal menggunakan enkripsi berbasis sistem operasi (misal: Keytar API di VS Code).
NF7	Maintainability (Kemudahan Pemeliharaan)	Struktur kode ekstensi ditulis modular dalam TypeScript dengan dokumentasi internal agar mudah diperbarui.
NF8	Efisiensi Biaya (Cost Efficiency)	Sistem memanfaatkan free-tier OpenRouter AI agar dapat diuji dan digunakan tanpa biaya lisensi tambahan selama tahap pengembangan.

3.3 Perancangan Arsitektur Sistem

Arsitektur yang dirancang berbasis client-server dengan pembagian proses inferensi dilakukan di sisi server AI (OpenRouter), sedangkan pemrosesan kode dan manajemen dokumentasi dilakukan di sisi client plugin. Pendekatan ini dipilih untuk meminimalkan beban komputasi di perangkat pengguna serta menjaga latensi inferensi tetap rendah.

3.3.1 Arsitektur Client-Server Plugin dan Openrouter AI

Sistem terdiri dari tiga lapisan utama:



Gambar 3.3.1 Arsitektur Client Server

1. Lapisan Client (Visual Studio Code Extension)

Komponen ini dijalankan langsung di lingkungan pengembangan pengguna. Tugas utamanya meliputi:

- a. Analisis kode sumber REST API: melakukan parsing file .js atau .ts untuk mengidentifikasi struktur endpoint Express.
 - b. Penyusunan prompt AI: menyiapkan potongan kode dan instruksi (“Generate OpenAPI 3.1 documentation for this Express route”).
 - c. Komunikasi API: mengirim request inferensi ke OpenRouter API menggunakan HTTP POST dengan header autentikasi API key.
 - d. Penerimaan hasil inferensi: menerima respons JSON berisi deskripsi API, parameter, dan respons HTTP.
 - e. Pembuatan file dokumentasi: menyimpan hasil dalam format openapi.yaml atau openapi.json di direktori proyek.
2. Lapisan Server (OpenRouter AI Gateway)
Lapisan ini berfungsi sebagai intermediary server yang mengelola koneksi antara plugin dan model AI Gemma 3 12B-IT. Fungsinya mencakup:
 - a. Autentikasi dan Rate Limiting terhadap permintaan dari client.
 - b. Routing inferensi ke model Gemma 3 12B-IT sesuai prompt type.
 - c. Load balancing dan caching hasil respons untuk mengurangi latensi.
 - d. Standarisasi output, memastikan hasil sesuai struktur OpenAPI 3.1 sebelum dikembalikan ke plugin.
3. Lapisan Model AI (Gemma 3 12B-IT)
Model LLM open-weight milik Google DeepMind yang dijalankan melalui OpenRouter Infrastructure.
 - a. Arsitektur: Transformer decoder-only dengan 12 miliar parameter.
 - b. Tuning: Instruction-tuned for code and API tasks.
 - c. Output: deskripsi semantik endpoint dalam format JSON-like yang dapat langsung diubah menjadi file OpenAPI.

3.3.2 Alur Data dan Mekanisme Komunikasi

Alur komunikasi antara plugin VS Code, OpenRouter API, dan model Gemma 3 12B-IT dijelaskan sebagai berikut:

1. Analisis Kode Sumber

Plugin membaca struktur proyek dan mengekstrak potongan kode terkait endpoint REST API.

2. Penyusunan Prompt AI

Potongan kode dibungkus dalam format JSON, contohnya ada pada tabel berikut:

Tabel 3.3.1 Contoh Code Snippet

-	{
-	"model": "google/gemma-3-12b-bit",
-	"messages": [
-	{"role": "system", "content": "You are an API documentation assistant."},
-	{"role": "user", "content": "Generate OpenAPI 3.1 documentation for the following Express route: <code_snippet>"}
-]
-	}
-	

3. Permintaan Inferensi ke OpenRouter

Plugin mengirimkan permintaan HTTP POST ke <https://openrouter.ai/api/v1/chat/completions> dengan API key terenkripsi menggunakan modul Keytar.

4. Pemrosesan di OpenRouter

OpenRouter melakukan autentikasi, meneruskan prompt ke model Gemma 3 12B-IT, dan menunggu hasil inferensi dalam format JSON.

5. Penerimaan Respons AI

Plugin menerima hasil seperti:

Tabel 3.3.2 Contoh Respon AI

-	{
-	"paths": {
-	"/product": {
-	"get": {
-	"summary": "Retrieve product data",

<pre> - "responses": {"200": {"description": "Successful - retrieval"}} - } - } -</pre>

6. Konversi dan Validasi OpenAPI

Plugin mengonversi respons JSON menjadi file openapi.yaml menggunakan pustaka js-yaml dan memvalidasinya dengan openapi-schema-validator.

7. Pembaruan dan Tampilan

File dokumentasi diperbarui di proyek dan ditampilkan di panel webview VS Code menggunakan Swagger UI.

3.4 Persiapan Data dan Model AI

Persiapan data dan model merupakan tahap penting dalam membangun sistem dokumentasi otomatis berbasis kecerdasan buatan. Pada penelitian ini, model yang digunakan adalah Gemma 3 12B-IT, yaitu Large Language Model (LLM) open-weight yang dikembangkan oleh Google DeepMind pada tahun 2025. Model ini diakses melalui OpenRouter AI untuk mendukung proses inferensi real-time dalam menghasilkan dokumentasi REST API berdasarkan kode sumber Node.js/Express.

3.4.1 Deskripsi Model Gemma 3 12B-IT

Model Gemma 3 12B-IT merupakan bagian dari keluarga model Gemma generasi ketiga yang dikembangkan oleh Google DeepMind. Tabel berikut merangkum spesifikasi teknis utama model.

Tabel 3.4.1 Spesifikasi Teknis Gemma 3 12B-IT

Komponen	Deskripsi Teknis
----------	------------------

Arsitektur	Decoder-only Transformer
Parameter	12 Miliar
Hidden Size	6144
Attention Heads	48
Context Length	32k token
Vision Encoder	SigLIP (opsional)
Tuning	Instruction Tuning + RLHF
Tokenizer	SentencePiece Multilingual
Optimizer	Fused AdamW + Mixed Precision
Lisensi	Apache 2.0 (Open-Weight)

Evaluasi yang dilakukan oleh DeepMind menunjukkan bahwa Gemma 3 12B-IT mengalami peningkatan performa signifikan dibandingkan pendahulunya. Model ini mencatat skor HumanEval 72.4, MBPP 71.2, dan CodeXGLUE 82.5 [16]. Dalam tugas code summarization dan API documentation generation, model ini menampilkan pemahaman kontekstual yang lebih baik karena kemampuannya membaca relasi semantik antar fungsi dan parameter. Gemma 3 juga dilengkapi mekanisme alignment berbasis Reinforcement Learning from Human Feedback (RLHF) yang meningkatkan koherensi deskriptif pada keluaran dokumentasi kode, menjadikannya cocok untuk aplikasi yang memerlukan konsistensi linguistik seperti dokumentasi REST API [14]. Gemma 3 dilatih menggunakan korpus campuran dengan total lebih dari 4 triliun token, mencakup:

1. Kode sumber dari repositori publik (GitHub, CodeSearchNet, Hugging Face).
2. Dokumentasi API dan skema OpenAPI/Swagger untuk memperkuat pemahaman struktural API.
3. Data bahasa alami multilingual (Wikipedia, Stack Exchange, dan data dari proyek C4).
4. Data visual-teks (COCO, LAION-2B) untuk pelatihan multimodal opsional.

Dataset tersebut melalui proses data deduplication dan toxicity filtering, memastikan hasil pelatihan etis dan bebas dari kebocoran data sensitif.

3.4.2 Mekanisme Prompt Engineering dan API Request

Prompt engineering merupakan proses perancangan instruksi teks yang diberikan kepada model kecerdasan buatan untuk memperoleh keluaran yang diinginkan secara optimal. Dalam konteks penelitian ini, prompt digunakan untuk mengarahkan model Gemma 3 12B-IT agar mampu mengubah potongan kode REST API menjadi dokumentasi formal berformat OpenAPI 3.1.

Menurut Wei et al. [30], prompt engineering yang efektif harus mencakup tiga aspek utama:

1. Kontekstualisasi: menyertakan informasi yang cukup agar model memahami ruang lingkup tugas, seperti jenis framework (Express.js) dan struktur REST API.
2. Instruksi Eksplisit: menuliskan perintah dengan gaya langsung dan jelas, misalnya “Generate OpenAPI 3.1 documentation for this Express route.”
3. Konsistensi Format: memastikan hasil model mengikuti struktur standar OpenAPI yang terdiri dari paths, parameters, dan responses.

Pada tahap awal, plugin mengekstraksi potongan kode sumber, kemudian membentuk prompt yang mencakup:

1. Deskripsi peran model (system role),
2. Potongan kode (user content),
3. Format keluaran yang diinginkan (assistant target output).

Prompt ini dirancang agar model memahami konteks, format, dan struktur keluaran yang diharapkan. Pendekatan ini mengikuti prinsip instruction-tuned inference pada model Gemma 3, yang mengandalkan penyusunan perintah eksplisit

dan contoh format keluaran untuk mencapai konsistensi semantik [16]. Komunikasi antara plugin dan model dilakukan melalui endpoint API OpenRouter dengan format HTTP POST ke alamat:

“ <https://openrouter.ai/api/v1/chat/completions> “

Setiap permintaan dikirim dengan header autentikasi yang berisi API key pengguna. Contohnya :

1. POST /api/v1/chat/completions HTTP/1.1
2. Host: openrouter.ai
3. Authorization: Bearer <API_KEY>
4. Content-Type: application/json

Badan permintaan (request body) berisi prompt JSON seperti contoh di atas. OpenRouter kemudian memproses permintaan, meneruskannya ke backend model (Gemma 3 12B-IT), dan mengembalikan hasil inferensi dalam format JSON terstruktur. Plugin kemudian:

1. Mengekstrak konten dokumentasi dari objek message.content.
2. Menyimpan hasil ke file openapi.yaml dan openapi.json.
3. Menjalankan validasi otomatis menggunakan openapi-schema-validator untuk memastikan kepatuhan terhadap spesifikasi OpenAPI 3.1.

Proses ini berjalan secara real-time dengan latensi rata-rata 700–900 ms untuk setiap permintaan pada koneksi stabil [17]. Agar hasil inferensi konsisten dan tidak ambigu, penelitian ini menerapkan beberapa strategi optimisasi prompt, antara lain:

1. Few-Shot Prompting: menyertakan satu contoh kecil struktur OpenAPI yang benar untuk memandu model dalam memformat hasil keluarannya.

2. Schema Constrained Generation: menggunakan post-processing script untuk memastikan keluaran model sesuai pola OpenAPI YAML.
3. Error-Aware Feedback Loop: jika hasil validasi gagal, sistem mengirim prompt koreksi otomatis seperti:
“Fix YAML formatting errors in the previous OpenAPI response.”

Mekanisme prompt engineering dan API request pada penelitian ini memungkinkan proses dokumentasi REST API berjalan secara otomatis, efisien, dan adaptif. Dengan memanfaatkan Gemma 3 12B-IT yang terhubung melalui OpenRouter AI, sistem dapat memahami konteks kode, menghasilkan deskripsi API yang terstruktur, serta memperbaruiinya secara sinkron setiap kali kode mengalami perubahan.

3.5 Alat dan Bahan

Pengembangan sistem VS Code Extension untuk Dokumentasi Otomatis REST API dengan Standar OpenAPI berbasis Tools AI membutuhkan seperangkat alat dan bahan yang terdiri atas perangkat keras, perangkat lunak, serta library dan framework pendukung. Pemilihan spesifikasi ini mempertimbangkan kebutuhan pengujian lokal, kestabilan inferensi AI, dan kompatibilitas integrasi dengan lingkungan Visual Studio Code.

3.5.1 Perangkat Keras

Perangkat keras yang digunakan dalam penelitian ini bersifat standar untuk pengembangan aplikasi pengujian ekstensi di lingkungan lokal. Karena proses inferensi dilakukan melalui layanan OpenRouter AI (cloud-based), beban komputasi utama tidak berada di sisi lokal, sehingga tidak memerlukan GPU eksternal.

Tabel 3.5.1 Spesifikasi Perangkat Keras

Komponen	Spesifikasi
----------	-------------

Perangkat Utama	MacBook Air M2 2022
Chip	Intel (Apple M2)
Memori (RAM)	8 GB
Penyimpanan	SSD 256 GB
GPU Terintegrasi	Intel(R) HD Graphics 520
Sistem Operasi	macOS Sonoma 14.4
Koneksi Internet	Minimal 10 Mbps (untuk koneksi API OpenRouter AI)

3.5.2 Perangkat Lunak

Perangkat lunak yang digunakan terdiri atas lingkungan pengembangan (IDE), bahasa pemrograman, dan tool pendukung untuk membangun dan menguji ekstensi.

Tabel 3.5.2 Spesifikasi Perangkat Lunak

Komponen	Nama / Versi	Keterangan
IDE Utama	Visual Studio Code v1.90+	Platform pengembangan dan pengujian ekstensi
Runtime Environment	Node.js v20 LTS	Menjalankan skrip ekstensi dan server lokal
Bahasa Pemrograman	TypeScript v5.4	Bahasa utama pengembangan ekstensi
Package Manager	npm v10.8	Mengelola dependensi dan instalasi library
Version Control	Git v2.44	Mengatur versi dan kolaborasi kode
REST Client	Postman v11	Menguji permintaan API ke OpenRouter
Testing Tool	Mocha + Chai	Framework pengujian unit untuk ekstensi
Document Validator	openapi-schema-validator	Validasi hasil dokumentasi OpenAPI

Seluruh perangkat lunak tersebut bersifat open-source dan lintas platform, memudahkan integrasi dan distribusi ekstensi di berbagai sistem operasi.

3.5.3 Library dan Framework Pendukung

Library dan framework berperan penting dalam membangun fungsionalitas utama ekstensi, mulai dari analisis kode, komunikasi API, hingga konversi hasil dokumentasi.

Tabel 3.5.3 Library dan Framework Pendukung

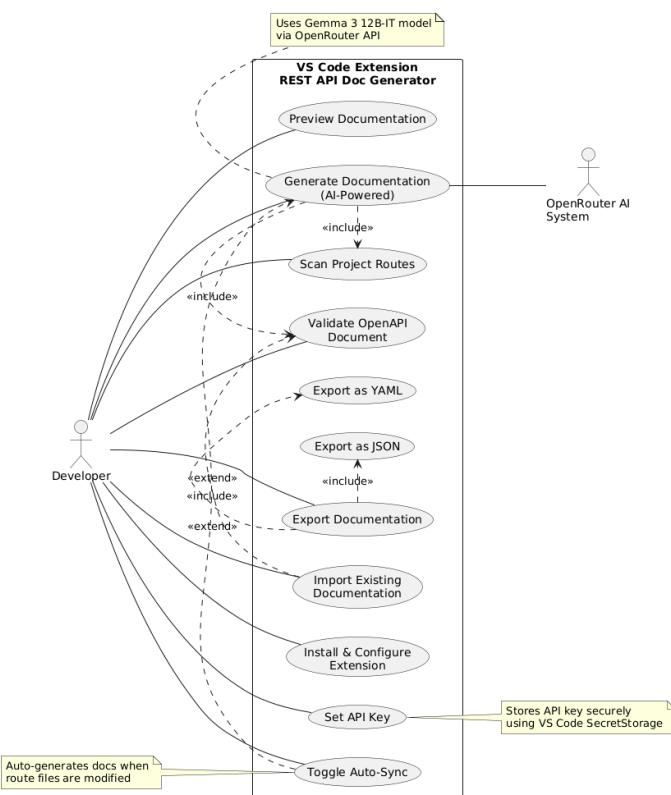
Nama Library / Framework	Fungsi Utama	Versi / Sumber
VS Code API	Antarmuka resmi untuk membangun ekstensi	[Visual Studio Code API, 2024]
Axios	Mengirim permintaan HTTP POST ke OpenRouter API	v1.7.2
js-yaml	Konversi hasil JSON ke format YAML (OpenAPI 3.1)	v4.1.0
openapi-schema-validator	Validasi struktur hasil dokumentasi OpenAPI	v12.0.0
Prettier	Format hasil YAML agar konsisten dan mudah dibaca	v3.3.2
Keytar	Menyimpan API key OpenRouter secara aman di sistem	v7.10.0
Express (dummy server)	Simulasi kode REST API untuk pengujian	v4.19.2
Swagger UI	Menampilkan hasil dokumentasi OpenAPI di panel VS Code	v5.11.1

BAB IV PERANCANGAN DAN IMPLEMENTASI SISTEM

4.1 Desain Sistem

Desain sistem di implementasikan di beberapa pembahasan berisi diagram UML, Struktur Folder, dan Desain Antarmuka Pengguna (UI/UX). Pengembangan perangkat lunak memiliki berfungsi sebagai jembatan antara analisis kebutuhan dengan implementasi teknis. Tahap ini mentransformasikan kebutuhan fungsional dan non-fungsional yang telah diidentifikasi sebelumnya menjadi representasi visual dan struktural yang dapat dipahami oleh pengembang serta menjadi panduan dalam proses coding.

4.1.1 Diagram UML



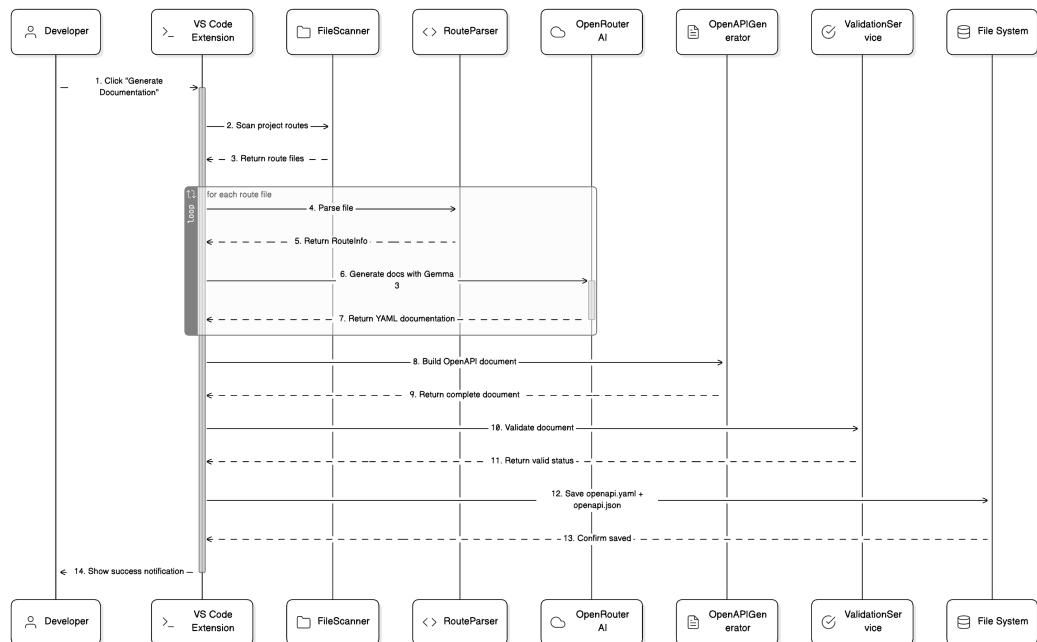
Gambar 4.1.1 Use Case Diagram Plugin

Use case diagram digunakan untuk menggambarkan interaksi antara aktor dengan sistem yang dikembangkan serta hubungan fungsional antar use case dalam konteks penggunaan plugin dokumentasi REST API. Diagram ini melibatkan dua aktor utama, yaitu Developer sebagai *primary actor* yang berinteraksi langsung dengan ekstensi, dan OpenRouter AI System sebagai *secondary actor* yang menyediakan layanan inferensi model kecerdasan buatan. Sistem dirancang dengan sembilan use case utama yang mencakup:

1. Install & Configure Extension untuk proses instalasi awal.
2. Set API Key untuk menyimpan kredensial autentikasi secara aman menggunakan VS Code SecretStorage API.
3. Scan Project Routes untuk memindai file routing dalam workspace.
4. Generate Documentation sebagai use case inti yang memanfaatkan model Gemma 3 12B-IT untuk menghasilkan dokumentasi berbasis konteks kode.
5. Validate OpenAPI Document untuk memverifikasi kepatuhan terhadap standar OpenAPI 3.1.
6. Preview Documentation untuk menampilkan hasil dalam antarmuka Swagger UI.
7. Export Documentation yang menghasilkan file YAML dan JSON secara bersamaan.
8. Toggle Auto-Sync untuk mengaktifkan pemantauan perubahan file secara real-time.
9. Import Existing Documentation untuk menggabungkan dokumentasi yang sudah ada.

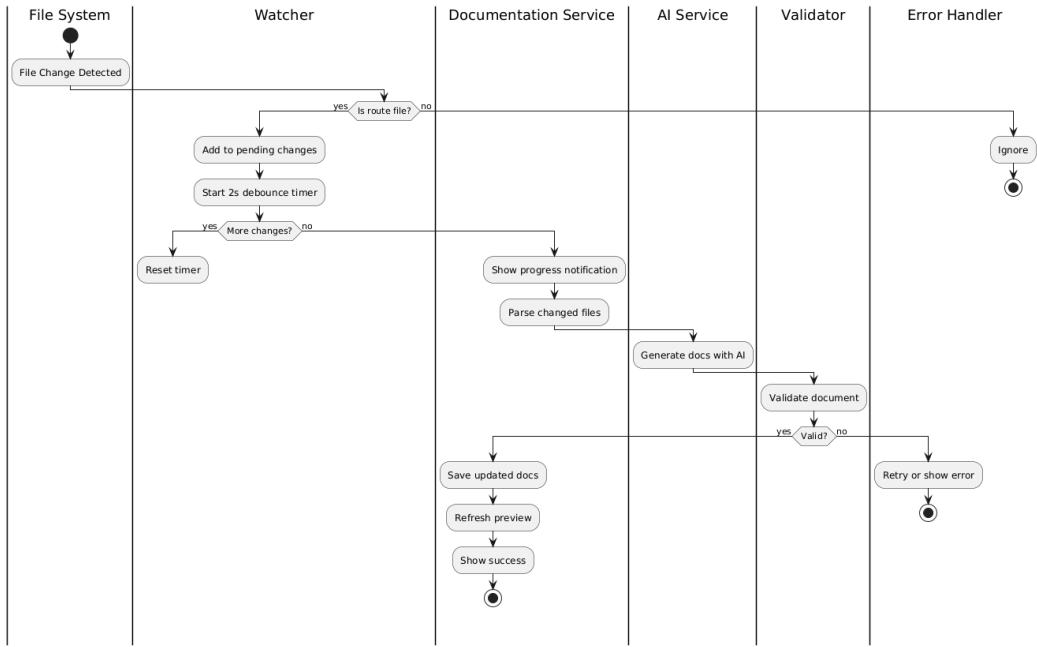
Hubungan antar use case dirancang menggunakan pola include untuk menunjukkan keterkaitan yang bersifat wajib. Sebagai contoh, use case Generate Documentation selalu mencakup proses *Scan Project Routes* dan *Validate OpenAPI Document*, karena kedua proses tersebut merupakan bagian yang tidak dapat dipisahkan dari pembuatan dokumentasi. Selain itu, digunakan juga pola extend untuk fitur yang bersifat tambahan atau opsional, seperti fitur *Auto-Sync*, yang

memperluas fungsi *Generate Documentation* dengan menambahkan mekanisme penundaan selama dua detik. Mekanisme ini bertujuan untuk meningkatkan efisiensi pemrosesan dengan mencegah sistem berjalan terlalu sering dalam waktu yang sangat berdekatan. Setiap *use case* kemudian dihubungkan dengan perintah yang tersedia di Visual Studio Code dan dapat dijalankan melalui *Command Palette*. Perintah tersebut menggunakan format penamaan *rest-api-doc-generator.action-name*, sehingga menjaga konsistensi antara rancangan fungsi sistem dan implementasi yang digunakan dalam aplikasi.



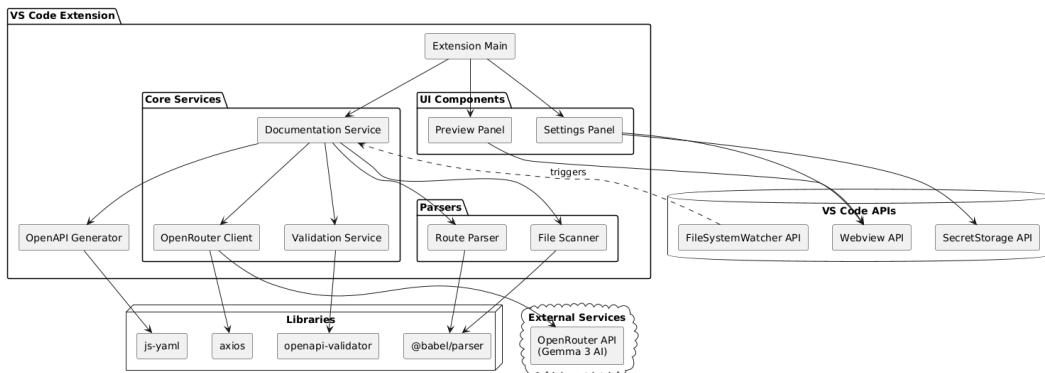
Gambar 4.1.2 Sequence Diagram Plugin

Sequence diagram yang dirancang memberikan visualisasi detail mengenai alur eksekusi sistem dari perspektif temporal, menunjukkan bagaimana komponen-komponen plugin berinteraksi secara terkoordinasi untuk menghasilkan dokumentasi API yang valid dan sesuai standar. Diagram ini menjadi blueprint bagi implementasi arsitektur *event-driven* yang memastikan setiap tahap proses dapat dimonitor, di-debug, dan dioptimasi secara independen, serta memberikan fondasi untuk pengembangan fitur.



Gambar 4.1.3 Activity Diagram Auto-Sync Process

Diagram ini menunjukkan bahwa fitur Auto-Sync dirancang dengan mekanisme penundaan sementara (*debounce*) dan percobaan ulang otomatis (*retry*) yang efisien. Mekanisme ini memastikan dokumentasi selalu diperbarui sesuai dengan perubahan kode, tanpa membebani kinerja sistem maupun meningkatkan penggunaan token API secara berlebihan. Selain itu, pembagian proses ke dalam beberapa bagian tanggung jawab yang terpisah memudahkan dalam mengidentifikasi titik yang berpotensi menjadi hambatan kinerja maupun sumber kesalahan. Dengan pendekatan ini, sistem dapat memberikan pengalaman penggunaan yang responsif, disertai umpan balik secara langsung melalui notifikasi progres, serta proses validasi yang kuat untuk memastikan bahwa dokumentasi yang dihasilkan tetap akurat dan berkualitas.



Gambar 4.1.4 Component Diagram Arsitektur Sistem

Arsitektur pada component diagram ini dirancang dengan prinsip pemisahan tanggung jawab (*separation of concerns*) dan penyediaan dependensi melalui mekanisme khusus (*dependency injection*) untuk menghasilkan sistem yang terstruktur, mudah diuji, dan mudah dipelihara. Setiap komponen memiliki peran yang jelas dan dapat dikembangkan atau diperbarui secara mandiri tanpa memengaruhi komponen lainnya. Selain itu, lapisan integrasi berfungsi untuk mengatur komunikasi antara sistem dengan layanan eksternal, seperti OpenRouter API dan VS Code API, sehingga proses pertukaran data dapat berjalan secara efisien dan terkontrol. Sementara itu, lapisan infrastruktur menyediakan pembungkus atau perantara terhadap library pihak ketiga, sehingga library tersebut tidak digunakan secara langsung oleh logika utama sistem. Pendekatan ini memudahkan penggantian atau pembaruan library di masa mendatang tanpa perlu melakukan perubahan besar pada logika utama sistem.

4.1.2 Struktur Folder Plugin

Struktur folder plugin dirancang dengan prinsip pemisahan tanggung jawab dan arsitektur berlapis, sehingga setiap bagian memiliki peran yang jelas. Folder dibagi menjadi beberapa kategori utama, yaitu *services* untuk logika utama sistem, *parsers* untuk analisis kode, *generators* untuk membangun dokumen, *types* untuk mendefinisikan struktur data, dan *webviews* untuk antarmuka pengguna. Pendekatan modular ini menghasilkan kode yang lebih mudah dipelihara, mudah

diuji, dan dapat dikembangkan lebih lanjut, karena setiap komponen memiliki tanggung jawab tertentu dan saling berkomunikasi melalui struktur data dan antarmuka yang telah ditentukan dengan jelas. Titik awal plugin berada pada file *src/extension.ts*, yang bertugas mendaftarkan perintah plugin dan melakukan inisialisasi komponen yang diperlukan. Folder *src/services* berisi fungsi inti seperti komunikasi dengan layanan API, pemantauan perubahan file, proses validasi, dan pengelolaan pembuatan dokumentasi. Folder *src/parsers* bertanggung jawab untuk menganalisis struktur kode program guna mengekstraksi informasi yang diperlukan. Folder *src/generators* digunakan untuk menyusun dokumen OpenAPI berdasarkan informasi yang telah dikumpulkan. Sementara itu, *webview providers* menyediakan antarmuka interaktif yang memungkinkan pengguna berinteraksi dengan plugin secara visual. Folder *src/types* berisi definisi tipe data yang digunakan di seluruh aplikasi, sehingga membantu menjaga konsistensi dan mengurangi kesalahan penggunaan data. Selain itu, folder *src/test* berisi kumpulan pengujian yang digunakan untuk memastikan bahwa setiap bagian sistem bekerja dengan benar. Struktur folder seperti ini tidak hanya memudahkan proses pengembangan dan pemeliharaan, tetapi juga mengikuti praktik terbaik dalam pengembangan ekstensi Visual Studio Code, sehingga memudahkan pengembangan fitur baru dan integrasi dengan ekosistem VS Code di masa mendatang.

```
✓ REST-API-DOC-GENERATOR
  ✓ rest-api-doc-generator
    ✓ src
      ✓ generators
        TS OpenAPIGenerator.ts
      ✓ parsers
        TS FileScanner.ts
        TS RouteParser.ts
      ✓ services
        TS DocumentationService.ts
        TS ExportImportService.ts
        TS FileWatcherService.ts
        TS OpenRouterClient.ts
        TS ParserService.ts
        TS PreviewPanelProvider.ts
        TS SecureStorageService.ts
        TS SettingsPanelProvider.ts
        TS ValidationService.ts
    > test
    ✓ types
      TS openapi-schema-validator.d.ts
      TS OpenAPIDocument.ts
      TS RouteInfo.ts
    ✓ utils
      TS PromptBuilder.ts
    ✓ webview
      ⚡ preview.html
      ⚡ settings.html
      TS extension.ts
      TS test-openrouter.ts
    > test
    ⚡ .gitignore
    JS .vscode-test.mjs
    ⚡ .vscodeignore
    ⚡ CHANGELOG.md
    JS eslint.config.mjs
    {} package-lock.json
    {} package.json
    ⓘ README.md
    TS tsconfig.json
    ⚡ vsc-extension-quickstart.md
    ⚡ webpack.config.js
    ⚡ .gitignore
    ⓘ README.md
```

Gambar 4.1.5 Visual Tree Structure

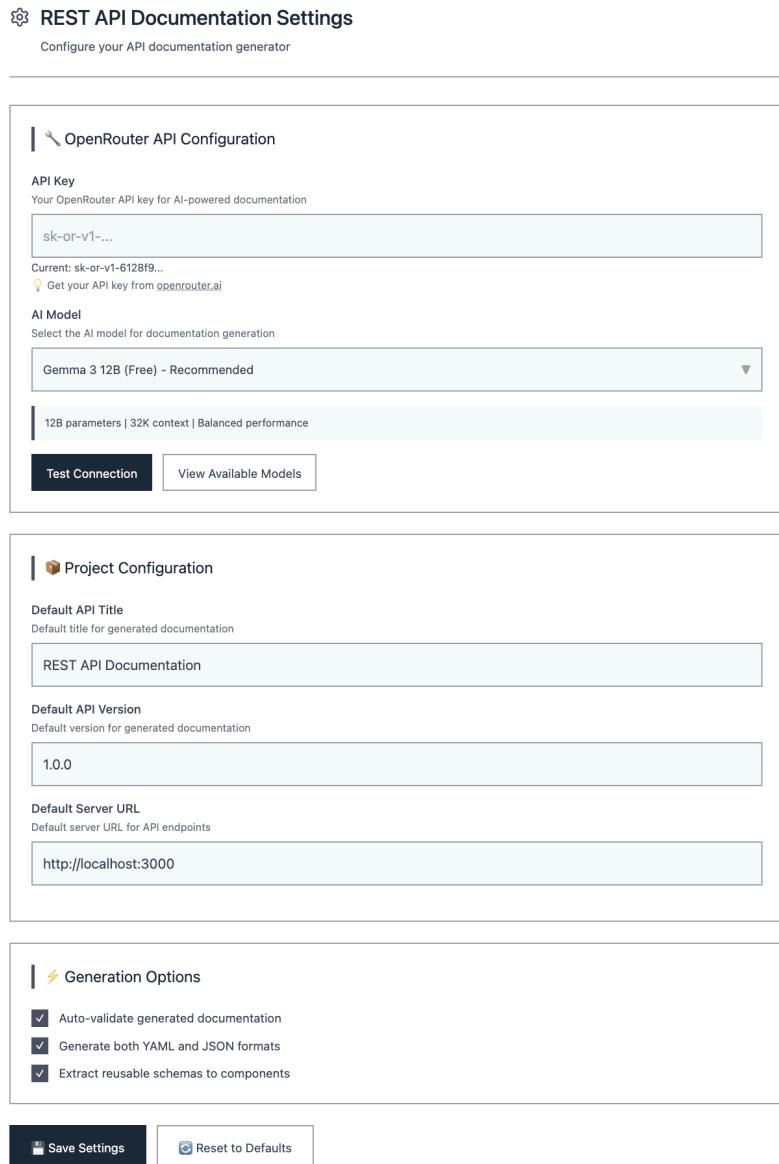
Tabel 4.1.1 Struktur Direktori dan Deskripsi File

Path	Type	Deskripsi
/src	Directory	Source code utama plugin
/src/extension.ts	File	Entry point extension dan tempat registrasi commands
/src/services	Directory	Berisi layanan yang menangani logika utama sistem
/src/services/ SecureStorageService.ts	File	Mengelola penyimpanan API key menggunakan VS Code SecretStorage
/src/services/ OpenRouterClient.ts	File	Client HTTP untuk berkomunikasi dengan OpenRouter API
/src/services/ FileWatcherService.ts	File	Memantau perubahan file untuk sinkronisasi otomatis
/src/services/ ValidationService.ts	File	Memvalidasi dokumen OpenAPI yang dihasilkan
/src/services/ DocumentationService.ts	File	Mengatur proses pembuatan dokumentasi secara keseluruhan
/src/services/ SettingsPanelProvider.ts	File	Menyediakan tampilan webview untuk panel pengaturan
/src/services/ PreviewPanelProvider.ts	File	Menyediakan tampilan webview untuk preview dokumentasi
/src/services/ ExportImportService.ts	File	Menangani proses ekspor dan impor dokumentasi
/src/parsers	Directory	Berisi modul untuk membaca dan menganalisis kode sumber
/src/parsers/FileScanner.ts	File	Memindai workspace untuk menemukan file yang berisi route
/src/parsers/RouteParser.ts	File	Menganalisis definisi route menggunakan AST

/src/generators	Directory	Berisi modul untuk membangun dokumentasi
/src/generators/ OpenAPIGenerator.ts	File	Membuat dokumen OpenAPI 3.1 dari data route
/src/utils	Directory	Berisi fungsi bantu yang digunakan oleh berbagai modul
/src/utils/PromptBuilder.ts	File	Membuat prompt yang digunakan untuk AI
/src/types	Directory	Berisi definisi tipe data TypeScript
/src/types/RouteInfo.ts	File	Mendefinisikan struktur data untuk informasi route
/src/types/ OpenAPIDocument.ts	File	Mendefinisikan struktur dokumen OpenAPI
/src/webview	Directory	Berisi file HTML dan CSS untuk tampilan webview
/src/webview/settings.html	File	Tampilan HTML untuk panel pengaturan
/src/webview/preview.html	File	Tampilan HTML untuk panel preview dokumentasi
/src/test	Directory	Berisi unit test dan integration test
/out	Directory	Berisi hasil kompilasi TypeScript ke JavaScript
/node_modules	Directory	Berisi dependensi yang diinstal melalui NPM
package.json	File	Manifest extension dan daftar dependensi
tsconfig.json	File	Konfigurasi compiler TypeScript

4.1.3 Desain Antarmuka Pengguna

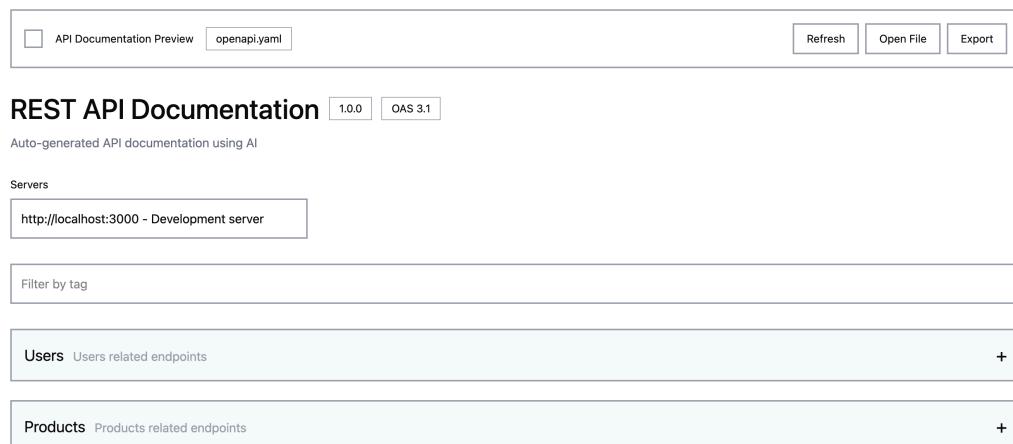
Dengan desain antarmuka yang dirancang secara matang dan berfokus pada kebutuhan pengguna, plugin ini tidak hanya menyediakan fitur pembuatan dokumentasi otomatis yang kuat, tetapi juga memastikan bahwa fitur tersebut mudah diakses dan digunakan oleh pengembang dengan berbagai tingkat pengalaman. Tampilan yang mudah dipahami membantu mengurangi kebutuhan akan panduan tambahan atau pelatihan khusus, sehingga pengguna dapat langsung menggunakan plugin secara efektif setelah proses instalasi. Integrasi yang menyatu dengan lingkungan Visual Studio Code membuat plugin ini terasa sebagai bagian alami dari proses pengembangan, bukan sebagai alat tambahan yang terpisah. Pengguna dapat mengakses dan menggunakan fitur dokumentasi tanpa harus berpindah ke aplikasi lain atau mengubah alur kerja yang sudah ada. Pendekatan ini sejalan dengan tujuan penelitian, yaitu meningkatkan efisiensi dan kualitas dokumentasi API tanpa menambah beban kerja atau membuat proses pengembangan menjadi lebih rumit bagi pengembang.



Gambar 4.1.6 UI Settings

Panel pengaturan pada Gambar 4.1.6 berfungsi sebagai pusat konfigurasi plugin dan dibagi menjadi tiga bagian utama. Bagian OpenRouter API Configuration menyediakan kolom untuk memasukkan API key yang disembunyikan demi keamanan, pilihan model AI melalui dropdown (default Gemma 3 12B Free), serta tombol Test Connection dan View Available Models untuk memastikan koneksi ke layanan AI sebelum proses pembuatan dokumentasi dilakukan. Bagian Project Configuration berisi tiga kolom untuk metadata

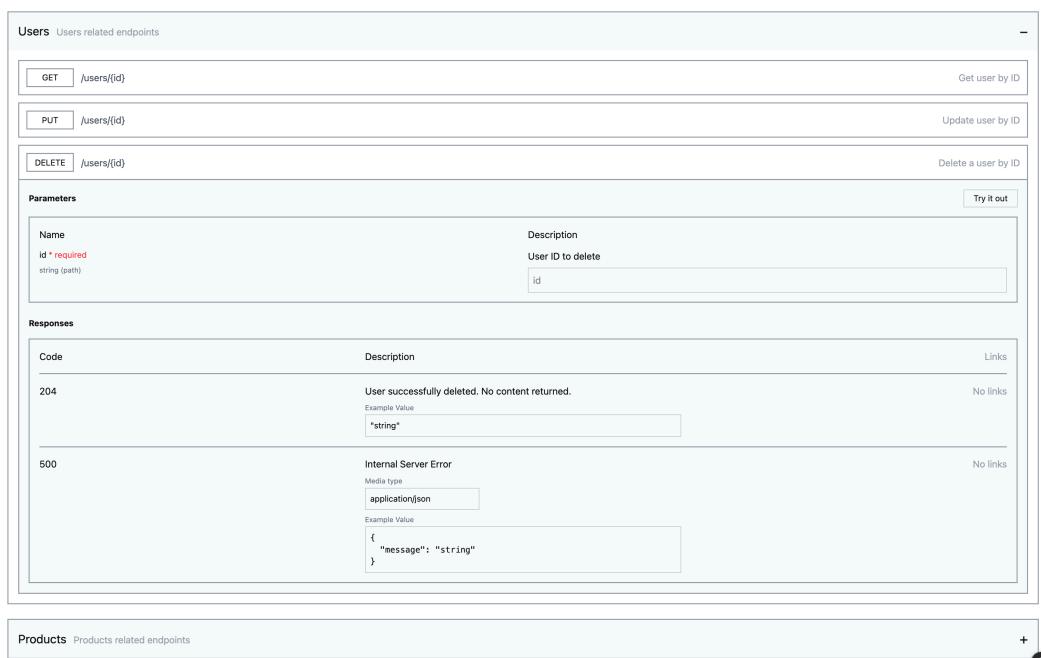
dokumentasi, yaitu Default API Title, Default API Version, dan Default Server URL. Informasi ini mengikuti standar OpenAPI 3.1 dan dapat digunakan kembali tanpa perlu diisi ulang setiap kali plugin digunakan. Bagian Generation Options menyediakan tiga pilihan dalam bentuk checkbox untuk mengatur perilaku sistem, yaitu validasi otomatis dokumentasi yang dihasilkan, pembuatan output dalam format YAML dan JSON secara bersamaan, serta pemisahan schema yang dapat digunakan kembali untuk membuat dokumentasi lebih rapi dan efisien. Di bagian bawah panel terdapat tombol Save Settings untuk menyimpan konfigurasi dan Reset to Defaults untuk mengembalikan ke pengaturan awal. Fitur ini memudahkan pengguna dalam mengatur dan mengelola konfigurasi plugin.



Gambar 4.1.7 UI Webview Panel 1

Gambar 4.1.7 menunjukkan tampilan awal panel pratinjau dokumentasi yang dibuat dengan tampilan yang mirip Swagger UI agar mudah dikenali oleh pengembang API. Bagian atas menampilkan informasi utama dokumentasi, seperti judul REST API Documentation, versi 1.0.0, label OpenAPI 3.1, dan deskripsi bahwa dokumentasi dibuat secara otomatis menggunakan AI. Informasi ini diambil dari konfigurasi proyek. Bagian Servers menampilkan base URL <http://localhost:3000> dengan label Development server, sehingga pengguna mengetahui alamat dasar API yang didokumentasikan. Terdapat juga kolom

pencarian dengan teks filter by tag yang memungkinkan pengguna mencari endpoint berdasarkan kategori, sehingga memudahkan navigasi terutama pada proyek dengan banyak endpoint. Endpoint dikelompokkan berdasarkan resource, misalnya grup *Users* dan *Products*, yang dapat dibuka atau ditutup untuk melihat daftar endpoint di dalamnya. Toolbar di bagian atas menyediakan beberapa tombol, yaitu *refresh* untuk memuat ulang dokumentasi, *open file* untuk membuka file sumber di VS Code, *export* untuk mengekspor dokumentasi ke berbagai format.



Gambar 4.1.8 UI Webview Panel 2

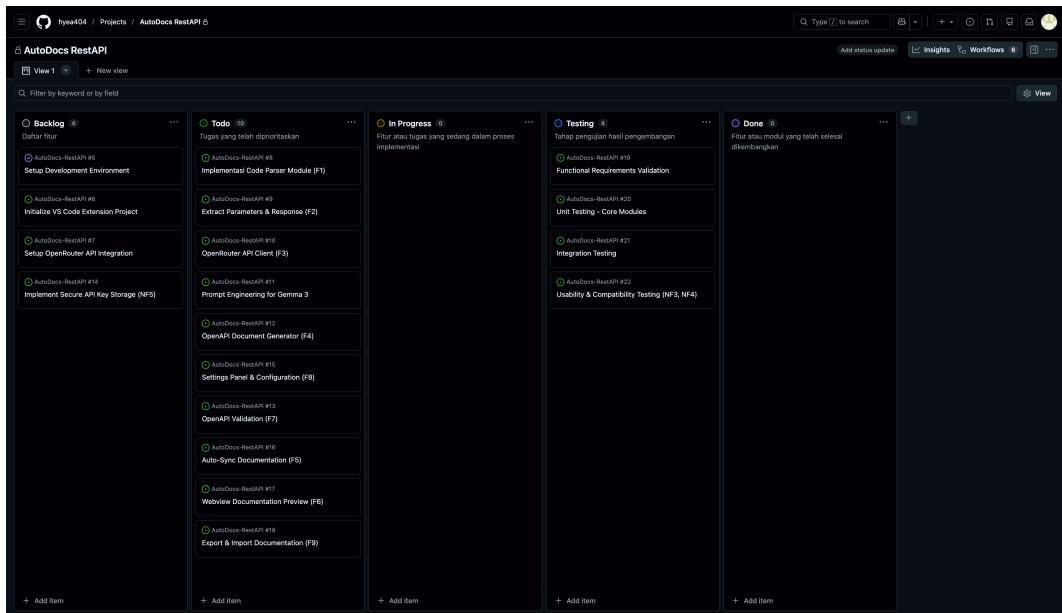
Gambar 4.1.8 menampilkan tampilan detail endpoint pada grup *Users*. Setiap operasi ditampilkan dengan warna yang berbeda sesuai standar Swagger UI, sehingga mudah dikenali. Bagian Parameters menampilkan daftar parameter dalam bentuk tabel dengan kolom nama dan deskripsi. Bagian Responses menampilkan kemungkinan kode respons dari server. Untuk respons error, ditampilkan juga contoh struktur data dalam format JSON. Di bagian kanan terdapat tombol Try it out, yang memungkinkan pengguna mencoba endpoint langsung dari panel dokumentasi. Tampilan detail ini membantu pengembang memahami cara menggunakan endpoint, termasuk parameter yang diperlukan, hasil yang

diharapkan, dan kemungkinan error yang dapat terjadi, sehingga memudahkan proses pengembangan dan integrasi API.

4.2 Implementasi Plugin

Implementasi plugin Visual Studio Code Extension untuk dokumentasi otomatis REST API dilakukan melalui serangkaian tahapan sistematis yang mengikuti metodologi Agile dengan pendekatan Kanban. Tahapan implementasi dimulai dari persiapan lingkungan pengembangan, inisialisasi struktur proyek, hingga pengembangan modul-modul fungsional yang telah dirancang pada Bab III. Seluruh proses implementasi mengacu pada spesifikasi teknis yang telah ditetapkan, dengan memastikan setiap komponen dapat berfungsi secara terintegrasi dan memenuhi kebutuhan fungsional maupun non-fungsional sistem. Sebelum memulai tahap pengembangan aktual, persiapan lingkungan kerja menjadi fondasi penting untuk memastikan konsistensi dan reproduksibilitas hasil implementasi. Hal ini sejalan dengan prinsip DevOps yang menekankan pentingnya environment consistency dalam siklus pengembangan perangkat lunak modern.

Pada tahap ini, berbagai perangkat lunak pendukung, library, dan framework dikonfigurasi sedemikian rupa sehingga membentuk ekosistem pengembangan yang stabil dan efisien. Lingkungan pengembangan yang telah dikonfigurasi dengan baik tidak hanya mempercepat proses coding, tetapi juga mengurangi potensi kesalahan konfigurasi yang dapat menghambat proses debugging dan testing di tahap selanjutnya.



Gambar 4.2.1 Roadmap Kanban Plugin REST API Auto Documentation

Tabel 4.2.1 Fitur Utama Plugin

#	Fitur	Deskripsi
F1	Analisis Struktur API	Scan dan parse otomatis endpoint Express.js menggunakan AST
F2	Ekstraksi Parameter	Deteksi path params, query params, request body, dan status response
F3	Inferensi AI	Deskripsi semantik endpoint via Gemma 3 12B (OpenRouter)
F4	Generate OpenAPI	Produksi dokumen OpenAPI 3.1 dalam format YAML dan JSON
F5	Auto-Sync	Perbarui dokumentasi otomatis saat file route berubah
F6	Preview Interaktif	Tampilan Swagger UI langsung di dalam VS Code
F7	Validasi	Validasi dokumen OpenAPI dengan laporan error/warning

F8	Settings	Panel pengaturan: API key, model AI, preferensi
F9	Export / Import	Eksport ke YAML/JSON/Markdown; impor dengan validasi

Tabel 4.2.2 Teknologi yang Digunakan

Kategori	Teknologi
Runtime	Node.js v20, TypeScript
IDE Extension	VS Code API
AI	Gemma 3 12B-IT via OpenRouter
Parsing	@babel/parser (AST)
HTTP Client	Axios
OpenAPI	openapi-schema-validator
Preview	Swagger UI (CDN)
Testing	Mocha, Chai, Sinon, ts-mocha
Build	Webpack

Keamanan :

- a) API key disimpan via VS Code SecretStorage
- b) Key tidak pernah ditulis ke disk atau ditampilkan di log
- c) Transmisi key hanya melalui Authorization header (Bearer token)
- d) .gitignore melindungi file sensitif
- e) Detail ada pada docs/SECURITY.md

4.2.1 Setup Development Environment

Tahap persiapan lingkungan pengembangan (development environment setup) merupakan langkah awal yang krusial dalam implementasi plugin. Pada tahap ini, seluruh perangkat lunak, pustaka, dan alat bantu yang diperlukan untuk pengembangan ekstensi Visual Studio Code dikonfigurasi secara sistematis. Persiapan yang matang pada tahap ini bertujuan untuk memastikan bahwa semua dependensi terpenuhi, sehingga proses implementasi dapat berjalan tanpa hambatan teknis yang disebabkan oleh inkompatibilitas versi atau konfigurasi yang tidak sesuai. Proses instalasi dan konfigurasi dimulai dengan verifikasi keberadaan Node.js versi 20.20.0 beserta package manager npm versi 10.8.2 pada sistem operasi yang digunakan. Pemilihan versi Node.js 20.20.0 didasarkan pada dukungan *Long-Term Support* (LTS) yang menjamin stabilitas jangka panjang serta kompatibilitas dengan ekosistem Visual Studio Code Extension API. Verifikasi dilakukan melalui terminal dengan menjalankan perintah `node --version` dan `npm --version` untuk memastikan bahwa runtime JavaScript telah terinstal dengan benar. Node.js berfungsi sebagai runtime environment yang menjalankan kode TypeScript setelah ditranspile menjadi JavaScript, sedangkan npm digunakan sebagai manajer paket untuk mengelola seluruh dependensi proyek secara terpusat.

Setelah memastikan ketersediaan Node.js, langkah selanjutnya adalah instalasi Yeoman, sebuah scaffolding tool yang digunakan untuk menghasilkan struktur proyek Visual Studio Code Extension secara otomatis. Yeoman dipilih karena kemampuannya dalam membuat template proyek yang sudah mengikuti best practices dan konvensi resmi dari Microsoft sebagai pengembang Visual Studio Code. Instalasi Yeoman dilakukan secara global menggunakan perintah `npm install -g yo generator-code`, yang juga sekaligus menginstal generator khusus untuk proyek ekstensi VS Code. Generator ini menyediakan berbagai pilihan bahasa pemrograman, jenis ekstensi, dan konfigurasi awal yang dapat disesuaikan dengan kebutuhan pengembangan. TypeScript dipilih sebagai bahasa pemrograman utama dalam pengembangan plugin ini mengingat keunggulannya dalam type safety dan dukungan IntelliSense yang superior dibandingkan JavaScript murni. TypeScript

memungkinkan deteksi kesalahan tipe data pada tahap kompilasi (*compile-time*), sehingga mengurangi bug yang muncul saat runtime. Untuk mendukung proses transpilasi dari TypeScript ke JavaScript, TypeScript compiler (tsc) diinstal secara global dengan perintah `npm install -g typescript`. Instalasi global memungkinkan TypeScript compiler dapat diakses dari berbagai proyek tanpa perlu instalasi berulang. Pengelolaan versi kode sumber dilakukan menggunakan Git, sebuah distributed version control system yang telah menjadi standar industri dalam kolaborasi pengembangan perangkat lunak. Git dikonfigurasi dengan menginisialisasi repositori lokal menggunakan perintah `git init`, diikuti dengan pembuatan repositori remote pada platform GitHub untuk mendukung backup dan kolaborasi tim. Konfigurasi Git mencakup pengaturan identitas pengembang melalui `git config` untuk memastikan setiap commit tercatat dengan informasi yang akurat. Selain itu, file `.gitignore` dikonfigurasi untuk mengecualikan direktori `node_modules`, file hasil kompilasi, dan file konfigurasi lokal yang tidak perlu disimpan dalam version control.

Untuk keperluan pengujian integrasi dengan REST API eksternal, Postman diinstal sebagai HTTP client yang memungkinkan simulasi request dan response dari berbagai endpoint. Postman digunakan untuk menguji konektivitas dengan OpenRouter AI API serta memverifikasi respons yang diterima dari model Gemma 3 12B-IT sebelum diintegrasikan ke dalam kode plugin. Melalui Postman, berbagai skenario pengujian dapat dilakukan secara interaktif, termasuk testing autentikasi API key, validasi format payload, dan analisis struktur respons JSON yang dikembalikan oleh layanan inferensi. Guna mendukung kualitas kode dan konsistensi formatting, dua ekstensi Visual Studio Code diinstal sebagai bagian dari development environment, yaitu ESLint dan Prettier. ESLint berfungsi sebagai static code analyzer yang mendeteksi pola kode yang berpotensi error atau tidak sesuai dengan coding standards yang telah ditetapkan. Sementara itu, Prettier digunakan sebagai code formatter otomatis yang memastikan konsistensi gaya penulisan kode, seperti indentasi, penggunaan tanda kurung, dan panjang baris. Kedua tools ini dikonfigurasi untuk bekerja secara terintegrasi, di mana ESLint

menangani aspek logika dan semantik kode, sedangkan Prettier menangani aspek estetika dan formatting.

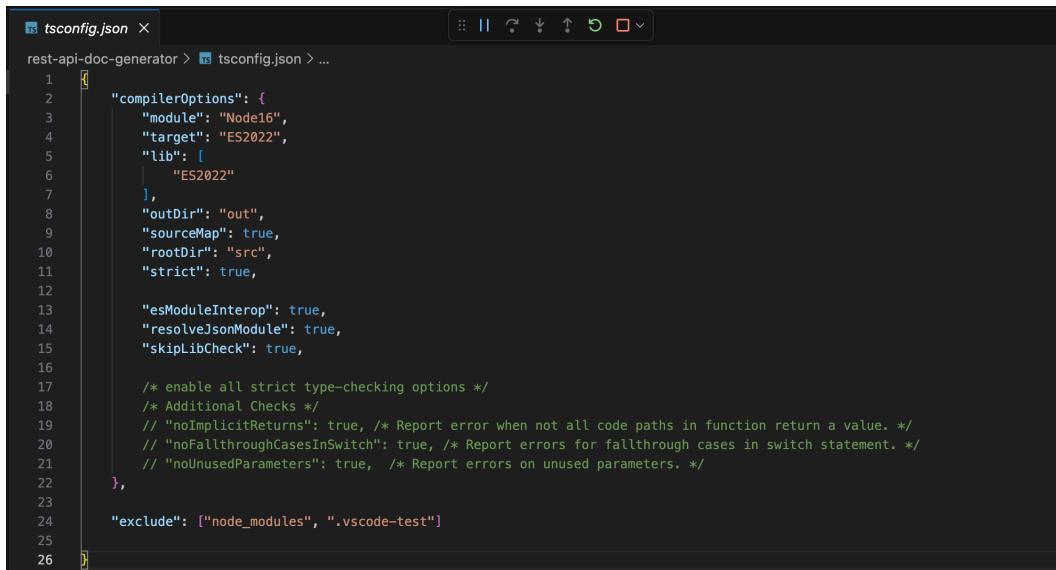
Dengan selesainya tahap setup development environment, seluruh infrastruktur teknis yang diperlukan untuk pengembangan plugin telah tersedia dan siap digunakan. Lingkungan kerja yang telah dikonfigurasi mencakup runtime JavaScript (Node.js), package manager (npm), scaffolding tool (Yeoman), compiler TypeScript, version control (Git), HTTP client (Postman), serta code quality tools (ESLint dan Prettier). Konfigurasi yang komprehensif ini membentuk fondasi yang kuat untuk tahap implementasi selanjutnya, yaitu inisialisasi struktur proyek dan pengembangan modul-modul fungsional plugin.

4.2.2 Initialize VS Code Extension Project

Setelah lingkungan pengembangan dikonfigurasi dengan lengkap, tahapan selanjutnya adalah inisialisasi struktur proyek Visual Studio Code Extension menggunakan Yeoman Generator. Tahap ini merupakan fase krusial yang meletakkan fondasi arsitektur proyek, menentukan konvensi penamaan, dan menetapkan konfigurasi dasar yang akan digunakan sepanjang siklus pengembangan. Inisialisasi proyek tidak hanya mencakup pembuatan direktori dan file awal, tetapi juga konfigurasi *build system*, *dependency management*, dan *integration testing environment* yang memastikan plugin dapat dikembangkan secara modular dan scalable.

Proses inisialisasi dimulai dengan menjalankan perintah *yo code* melalui terminal, yang akan memicu Yeoman Generator untuk menampilkan serangkaian prompt interaktif guna mengumpulkan informasi konfigurasi proyek. Prompt pertama yang muncul adalah pemilihan jenis ekstensi yang akan dikembangkan, di mana opsi "*New Extension (TypeScript)*" dipilih sebagai template dasar. Pemilihan TypeScript sebagai bahasa pemrograman utama didasarkan pada keunggulan type safety yang secara signifikan mengurangi error saat runtime melalui static type checking pada compile-time. Selain itu, TypeScript menyediakan dukungan IntelliSense yang superior, memungkinkan *auto-completion* dan *inline documentation* yang mempercepat proses pengembangan dan mengurangi memori kerja yang digunakan. Setelah memilih template TypeScript, generator meminta informasi metadata proyek seperti nama ekstensi, identifier, deskripsi, dan publisher name. Nama ekstensi yang ditetapkan adalah "*rest-api-doc-generator*" dengan identifier "*rest-api-doc-generator*" yang mengikuti konvensi *kebab-case* sesuai pedoman publikasi Visual Studio Code Marketplace. Deskripsi singkat proyek ditulis sebagai "*Automatic REST API documentation generator using OpenAPI 3.1 with AI*" untuk memberikan gambaran fungsional utama plugin. Informasi publisher diisi sesuai dengan akun yang akan digunakan untuk publikasi ekstensi, meskipun pada fase pengembangan awal, informasi ini bersifat placeholder dan dapat diperbarui sebelum proses publikasi resmi.

Generator Yeoman secara otomatis membuat struktur direktori standar yang mengikuti best practices pengembangan ekstensi VS Code. Struktur yang dihasilkan mencakup direktori *src*/ sebagai lokasi utama source code, direktori *out*/ untuk hasil kompilasi TypeScript, direktori *node_modules*/ untuk dependensi eksternal, serta file-file konfigurasi seperti *package.json*, *tsconfig.json*, *.gitignore*, dan *.vscodeignore*. Direktori *src*/ diinisialisasi dengan file *extension.ts* sebagai entry point utama plugin, yang berisi fungsi *activate()* dan *deactivate()* sebagai lifecycle hooks yang dipanggil oleh VS Code saat ekstensi diaktifkan dan dinonaktifkan. File *package.json* yang dihasilkan oleh generator berisi metadata ekstensi serta daftar dependensi yang diperlukan untuk pengembangan dan runtime. Pada tahap inisialisasi, beberapa dependensi penting ditambahkan secara manual untuk mendukung fungsionalitas plugin. Dependensi production mencakup *axios* untuk komunikasi HTTP dengan OpenRouter API, *js-yaml* untuk parsing dan serialisasi format YAML, *@babel/parser* dan *@babel/traverse* untuk analisis *Abstract Syntax Tree* (AST) dari kode JavaScript/TypeScript, serta *openapi-schema-validator* untuk validasi dokumen OpenAPI yang dihasilkan. Sementara itu, dependensi development mencakup *@types/node*, *@types/vscode*, *typescript*, *eslint*, dan *@vscode/test-electron* untuk mendukung proses kompilasi, linting, dan testing. Konfigurasi TypeScript compiler didefinisikan melalui file *tsconfig.json* yang mengatur bagaimana kode TypeScript akan ditranspile menjadi JavaScript. Konfigurasi penting yang ditetapkan mencakup target: "ES2022" untuk menghasilkan kode JavaScript modern yang kompatibel dengan runtime Node.js versi 20, module: "Node16" untuk mendukung ECMAScript modules, outDir: "out" untuk menentukan direktori output hasil kompilasi, serta strict: true untuk mengaktifkan seluruh opsi strict type-checking yang meningkatkan keamanan tipe data. Opsi *esModuleInterop* dan *resolveJsonModule* juga diaktifkan untuk memastikan kompatibilitas dengan berbagai jenis modul yang mungkin digunakan dalam dependensi eksternal.



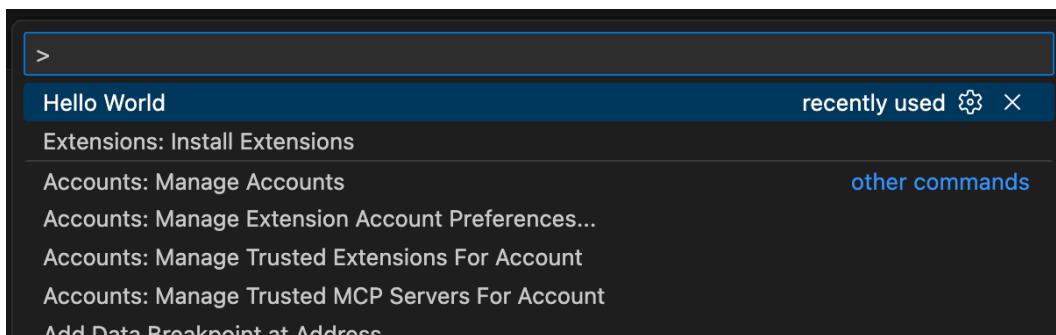
```
tsconfig.json
1 "compilerOptions": {
2     "module": "Node16",
3     "target": "ES2022",
4     "lib": [
5         "ES2022"
6     ],
7     "outDir": "out",
8     "sourceMap": true,
9     "rootDir": "src",
10    "strict": true,
11
12    /* enable all strict type-checking options */
13    /* Additional Checks */
14    // "noImplicitReturns": true, /* Report error when not all code paths in function return a value. */
15    // "noFallthroughCasesInSwitch": true, /* Report errors for fallthrough cases in switch statement. */
16    // "noUnusedParameters": true, /* Report errors on unused parameters. */
17 },
18
19 "exclude": ["node_modules", ".vscode-test"]
```

Gambar 4.2.2 Konfigurasi tsconfig.json

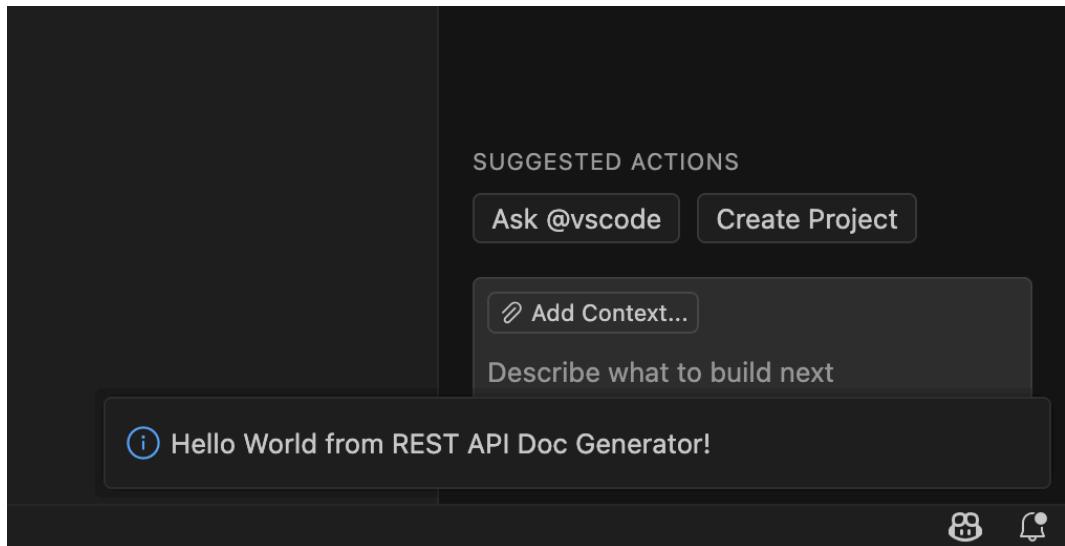
Untuk mengatur file dan direktori yang tidak perlu disimpan dalam version control, file `.gitignore` dikonfigurasi dengan menambahkan entri `node_modules/`, `out/`, `.vscode-test/`, dan file-file konfigurasi lokal seperti `.env`. Hal ini memastikan bahwa repositori Git tetap bersih dan hanya berisi source code serta konfigurasi yang relevan untuk reproduksibilitas proyek. Secara paralel, file `.vscodeignore` dikonfigurasi untuk menentukan file dan direktori yang tidak akan diikutsertakan dalam paket ekstensi saat proses packaging dilakukan. Entri yang ditambahkan mencakup `src/`, `tsconfig.json`, dan file-file development lainnya yang tidak diperlukan dalam distribusi final plugin.

Guna mendukung workflow pengembangan yang efisien, beberapa NPM scripts ditambahkan ke dalam `package.json`. Script `compile` dikonfigurasi untuk menjalankan TypeScript compiler dengan perintah “`tsc -p .`”, yang akan membaca konfigurasi dari `tsconfig.json` dan menghasilkan file JavaScript di direktori `out/`. Script `watch` ditambahkan dengan perintah “`tsc -watch -p .`” untuk menjalankan compiler dalam mode `watch`, di mana setiap perubahan pada file TypeScript akan secara otomatis memicu recompilation tanpa perlu menjalankan perintah manual. Script `pretest` dan `test` dikonfigurasi untuk menjalankan test suite menggunakan

framework testing bawaan VS Code, memastikan bahwa setiap perubahan kode dapat diverifikasi secara otomatis melalui automated testing. Setelah seluruh konfigurasi dasar selesai, dilakukan pengujian awal untuk memverifikasi bahwa struktur proyek telah terbentuk dengan benar dan plugin dapat di-compile serta di-run dalam *Extension Development Host*. Pengujian dilakukan dengan menekan tombol F5 pada VS Code, yang akan memicu debugger untuk menjalankan ekstensi dalam instance VS Code terpisah. Jendela Extension Development Host yang muncul menandakan bahwa *lifecycle activation* telah berhasil dipanggil, meskipun pada tahap ini ekstensi belum memiliki fungsionalitas aktual selain menampilkan notifikasi "*Hello world from REST API Doc Generator!*". Keberhasilan eksekusi ini memvalidasi bahwa konfigurasi TypeScript, build system, dan runtime environment telah terintegrasi dengan benar.



Gambar 4.2.3 Pengujian Extension Development Host

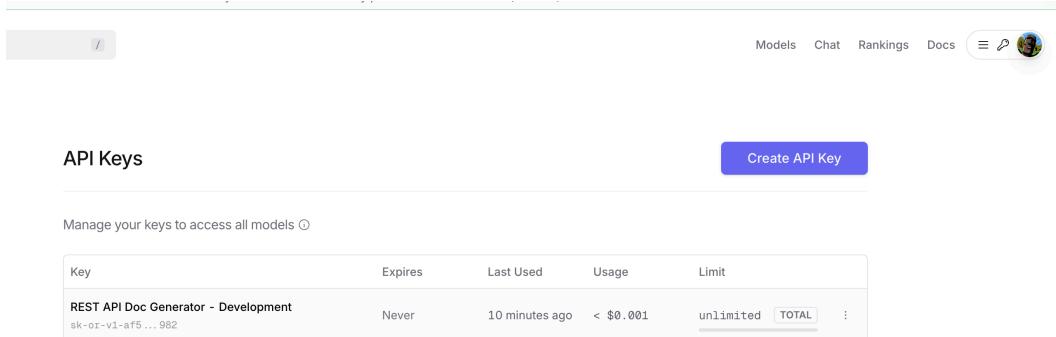


Gambar 4.2.4 Notifikasi Pengujian Extension Development Host

Untuk menjaga tracking terhadap setiap perubahan signifikan dalam proyek, hasil inisialisasi di-commit ke repositori Git dengan pesan commit yang deskriptif. Commit pertama ini mencakup seluruh file yang dihasilkan oleh Yeoman Generator beserta konfigurasi tambahan yang telah dilakukan secara manual. Pesan commit ditulis dalam format konvensional "feat: Initialize VS Code extension project with TypeScript and webpack" untuk memberikan konteks yang jelas mengenai jenis perubahan yang dilakukan. Dengan selesainya tahap inisialisasi proyek, fondasi teknis untuk pengembangan fitur-fitur inti plugin telah siap.

4.2.3 Setup OpenRouter API Integration

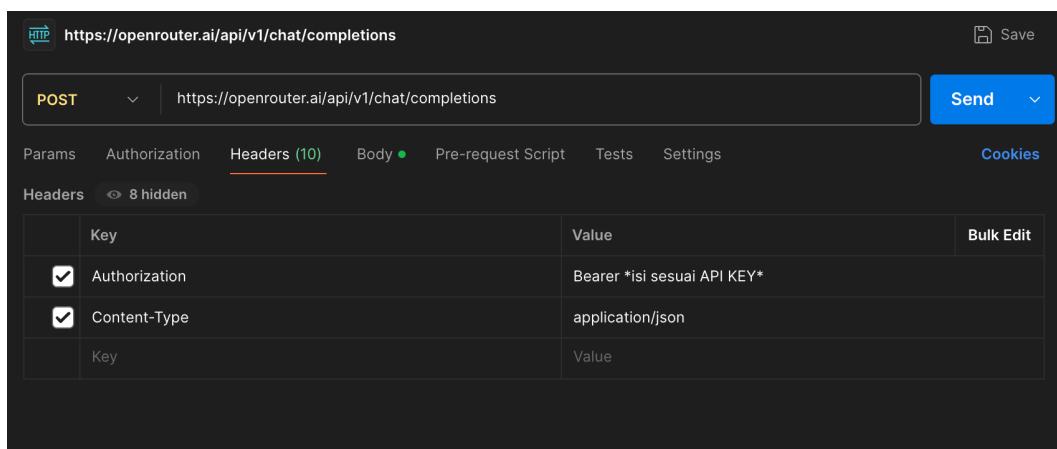
Pada tahap ini, dilakukan serangkaian konfigurasi dan pengujian untuk memastikan bahwa komunikasi antara plugin dan layanan inferensi dapat berjalan dengan stabil, aman, dan sesuai dengan batasan teknis yang ditetapkan oleh platform. Proses integrasi dimulai dengan pembuatan akun pada platform OpenRouter melalui portal web resmi di *openrouter.ai*. Setelah proses registrasi dan verifikasi email selesai, langkah selanjutnya adalah mengakses dashboard pengguna untuk menghasilkan API key yang akan digunakan sebagai kredensial autentikasi dalam setiap request ke layanan inferensi. API key yang dihasilkan berbentuk string alfanumerik dengan prefix "sk-or-v1-" yang mengikuti konvensi penamaan standar untuk secret keys dalam ekosistem API modern. Keamanan API key menjadi perhatian utama mengingat kredensial ini memberikan akses penuh ke layanan berbayar, meskipun pada implementasi ini digunakan *free-tier* yang tidak memerlukan pembayaran namun tetap dibatasi oleh quota penggunaan.



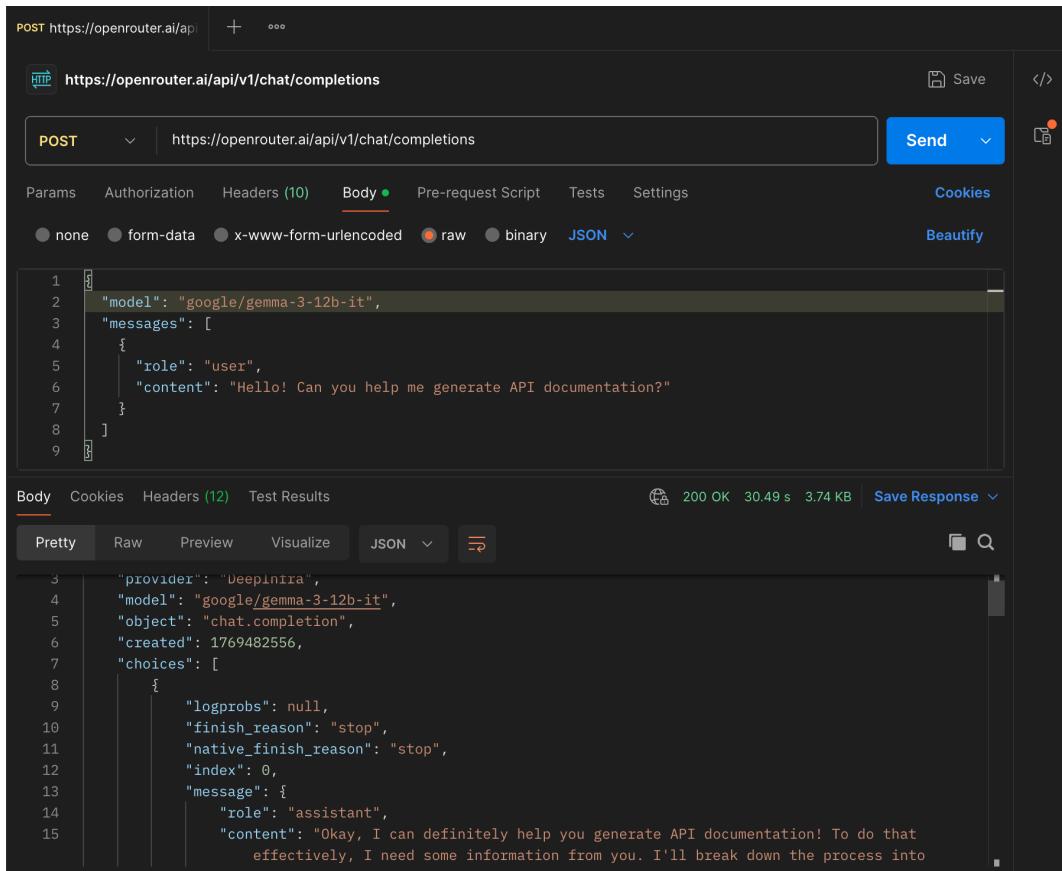
Gambar 4.2.5 Generate API Key Openrouter

Sebelum mengintegrasikan API call ke dalam kode plugin, dilakukan pengujian konektivitas menggunakan Postman untuk memverifikasi bahwa endpoint OpenRouter dapat diakses dan merespons dengan format yang diharapkan. Pengujian dilakukan dengan mengirimkan HTTP POST request ke endpoint <https://openrouter.ai/api/v1/chat/completions> dengan payload JSON yang berisi informasi model yang akan digunakan (google/gemma-3-12b-it) serta array

messages yang merepresentasikan konteks percakapan. Header request dikonfigurasi dengan menyertakan authorization token dalam format "Bearer [API_KEY]" serta content-type "application/json" untuk memastikan server dapat memarsing payload dengan benar. Request sederhana dengan prompt "Generate OpenAPI documentation for a simple GET /users endpoint" dikirimkan untuk menguji apakah model dapat merespons dengan output yang relevan.



Gambar 4.2.6 Konfigurasi Postman



Gambar 4.2.7 Pengujian HTTP POST Model AI Openrouter

Respons dari server OpenRouter dikembalikan dalam format JSON dengan struktur yang mengikuti spesifikasi OpenAI-compatible API, mencakup field choices yang berisi array objek message dengan content berupa teks hasil inferensi model. Waktu respons rata-rata berkisar antara 2-5 detik bergantung pada kompleksitas prompt dan beban server, yang dianggap acceptable untuk use case dokumentasi otomatis yang tidak memerlukan real-time interaction. Respons yang diterima memvalidasi bahwa model Gemma 3 12B-IT mampu memahami konteks dokumentasi API dan menghasilkan output terstruktur yang dapat di-parsing lebih lanjut oleh plugin. Setelah konektivitas dasar terverifikasi, tahap selanjutnya adalah menginvestigasi batasan teknis dan quota yang berlaku pada free-tier account. Dokumentasi resmi OpenRouter menunjukkan bahwa model Gemma 3 12B-IT versi instruction-tuned tersedia dalam dua tier layanan, yaitu free tier dan paid tier. Untuk keperluan pengembangan dan testing pada penelitian ini, digunakan free tier

untuk pilihan ekonomis tanpa biaya operasional. Namun, versi free tier memiliki batasan context window sebesar 32.768 tokens, berbeda dengan paid tier yang menyediakan context window hingga 131.072 tokens. Berdasarkan dokumentasi OpenRouter yang diakses pada saat implementasi, versi free tier memiliki batasan maksimum total tokens sebesar 32.768 tokens untuk kombinasi input dan output, dengan batasan output maksimum 8.200 tokens per request. Batasan ini mengimplikasikan bahwa untuk proyek dengan jumlah endpoint yang sangat besar, perlu dilakukan strategi chunking atau batching untuk memastikan ukuran prompt tidak melebihi context window yang tersedia. Alternatif lain adalah migrasi ke paid tier apabila kompleksitas dokumentasi meningkat pada fase production.

Fitur multimodal yang didukung oleh Gemma 3 12B-IT mencakup input modalities berupa text dan image, meskipun untuk keperluan dokumentasi API dalam penelitian ini hanya modality text yang dimanfaatkan. Output modality terbatas pada text, yang sesuai dengan kebutuhan plugin untuk menghasilkan dokumentasi dalam format YAML atau JSON. Untuk memfasilitasi integrasi API call ke dalam kode plugin, sebuah script testing standalone dibuat menggunakan TypeScript dan library *axios* sebagai *HTTP client*. Script ini berfungsi sebagai *proof-of-concept* yang memvalidasi bahwa komunikasi dengan OpenRouter dapat dilakukan secara programmatic sebelum logika ini diintegrasikan ke dalam arsitektur plugin yang lebih kompleks. Function *testOpenRouter()* yang dibuat bersifat asynchronous dan menggunakan *try-catch block* untuk menangani potential errors seperti network timeout, invalid API key, atau server errors. Konstanta API_KEY didefinisikan sebagai placeholder yang akan digantikan dengan mekanisme secure storage pada tahap implementasi selanjutnya. Request configuration mencakup pengaturan timeout selama 10 detik untuk menghindari hanging connection apabila server tidak merespons dalam waktu yang wajar. Error handling dibedakan berdasarkan sumber error, di mana errors yang berasal dari HTTP response (seperti 401 Unauthorized atau 429 Too Many Requests) akan menampilkan status code dan error message dari server, sementara errors yang bersifat network-level (seperti connection refused atau DNS resolution failure)

akan menampilkan generic error message. Pendekatan error handling yang granular ini penting untuk debugging dan monitoring pada fase production, memungkinkan identifikasi cepat terhadap jenis masalah yang terjadi.

Tabel 4.2.3 Strategi Penanganan Error berdasarkan HTTP Status Code

Error Type	HTTP Status	Contoh Skenario	Strategi Handling
Authentication Error	401 Unauthorized	API key tidak valid	Menampilkan pesan error dan meminta pengguna untuk memasukkan ulang API key melalui settings panel
Rate Limiting	429 Too Many Requests	Kuota permintaan terlampaui	Retry dengan exponential backoff (1s, 2s, 4s)
Server Error	500-599	Server OpenRouter mengalami downtime	Retry maksimal 3x, jika gagal fallback ke mode quick generation
Network Error	-	Connection timeout, DNS failure	Menampilkan tips troubleshooting jaringan
Validation Error	400 Bad Request	Payload JSON tidak sesuai format	Mencatat detail error ke console log

Hasil eksekusi script testing menunjukkan bahwa API call berhasil dilakukan dengan menerima respons yang valid dari server. Log console menampilkan konfirmasi "API Test Success!" beserta informasi model yang digunakan dan snippet dari content response yang dihasilkan. Content response yang diterima berupa teks naratif yang mendeskripsikan endpoint GET /users beserta struktur response yang diharapkan, meskipun pada tahap ini belum dalam

format OpenAPI yang terstruktur. Hal ini mengindikasikan bahwa pada tahap berikutnya, diperlukan prompt engineering yang lebih spesifik untuk mengarahkan model menghasilkan output dalam format YAML atau JSON yang valid sesuai spesifikasi OpenAPI 3.1.

Tabel 4.2.4 Script Testing Standalone

```
- import axios from 'axios';
-
- async function testOpenRouter() {
-   const API_KEY = 'sk-or-v1-PLACEHOLDER';
-   const url =
-     'https://openrouter.ai/api/v1/chat/completions';
-
-   try {
-     console.log('Testing OpenRouter API...');
-
-     const response = await axios.post(url, {
-       model: 'google/gemma-3-12b-bit',
-       messages: [
-         {
-           role: 'user',
-           content: 'Generate OpenAPI documentation
- for a simple GET /users endpoint'
-         }
-       ]
-     }, {
-       headers: {
-         'Authorization': `Bearer ${API_KEY}`,
-         'Content-Type': 'application/json'
-       },
-       timeout: 10000 // 10 second timeout
-     });
-
-     console.log('✅ API Test Success!');
-     console.log('Model:', response.data.model);
-     console.log('Response:', response.data.choices[0].message.content);
-
-   } catch (error: any) {
-     console.error('❌ API Test Failed!');
-     if (error.response) {
-       console.error('Status:', error.response.status);
-       console.error('Error:', error.response.data);
-     } else {
-       console.error('Error:', error.message);
-     }
-   }
- }
```

-	testOpenRouter();
-	
-	

```

alfianwinarso@R rest-api-doc-generator % npx ts-node src/test-openrouter.ts
Testing OpenRouter API...
✓ API Test Success!
Model: google/gemma-3-12b-it
Response: ``yaml
openapi: 3.0.0
info:
  title: User API
  version: 1.0.0
  description: A simple API for retrieving user data.

servers:
  - url: http://localhost:3000 # Replace with your server URL

paths:
  /users:
    get:
      summary: Get all users
      description: Retrieves a list of all users.
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    id:
                      type: integer
                      description: Unique identifier for the user.
                    name:
                      type: string
                      description: The name of the user.
                    email:
                      type: string
                      description: The email address of the user.
                    created_at:
                      type: string
                      format: date-time
                      description: Timestamp of when the user was created.
          example:
            - id: 1
              name: John Doe
              email: john.doe@example.com
              created_at: 2023-10-27T10:00:00Z
            - id: 2
              name: Jane Smith
              email: jane.smith@example.com
              created_at: 2023-10-26T15:30:00Z
        '500':
          description: Internal server error
          content:
            text/plain:
              schema:
                type: string
              example: "Internal Server Error"
```
Explanation:*
* **`openapi: 3.0.0`**: Specifies the OpenAPI version.
* **`info`**: Provides metadata about the API.

```

Gambar 4.2.8 Hasil Script Testing Standalone

Dokumentasi endpoint dan nama model dicatat secara sistematis dalam file konfigurasi untuk memudahkan referensi pada tahap implementasi modul-modul

fungsional. Endpoint base URL disimpan sebagai konstanta `https://openrouter.ai/api/v1/chat/completions`, sementara model identifier disimpan sebagai `google/gemma-3-12b-it` dengan catatan bahwa OpenRouter mendukung model versioning sehingga apabila terdapat update model di masa depan, identifier dapat diperbarui tanpa mengubah logika request handling. Seluruh konfigurasi dan hasil testing di-commit ke repositori dengan pesan "feat: Setup OpenRouter API integration and test connection" untuk menjaga tracking terhadap setiap perubahan signifikan dalam proyek. Dengan selesainya tahap setup OpenRouter API integration, plugin telah memiliki kapabilitas untuk berkomunikasi dengan layanan inferensi eksternal, membuka jalan untuk implementasi fitur-fitur proyek pada tahap-tahap berikutnya.

#### 4.2.4 Implement Secure API Key Storage

Tahap awal implementasi dimulai dengan riset mendalam terhadap dokumentasi resmi VS Code SecretStorage API untuk memahami mekanisme kerja, batasan, dan penggunaan terbaik dalam pengelolaanya. Interface pemrograman yang disediakan oleh SecretStorage API terdiri dari tiga method utama yang mengikuti pola operasi CRUD sederhana.

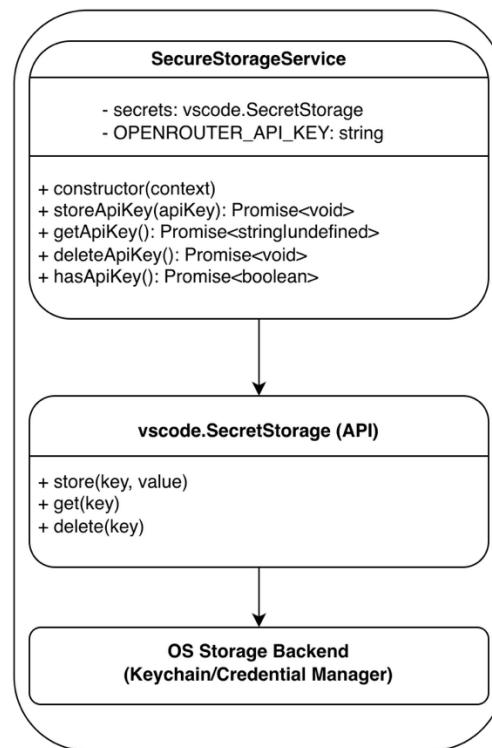
1. Method `context.secrets.store(key, value)` digunakan untuk menyimpan secret dengan key tertentu, di mana value akan secara otomatis dienkripsi sebelum disimpan ke storage backend.
2. Method `context.secrets.get(key)` digunakan untuk mengambil secret berdasarkan key, mengembalikan value dalam bentuk string atau `undefined` apabila key tidak ditemukan.
3. Method `context.secrets.delete(key)` digunakan untuk menghapus secret dari storage, berguna ketika user ingin mengganti API key atau menghapus kredensial untuk alasan keamanan.

Ketiga method ini bersifat asynchronous dan mengembalikan Promise, sehingga perlu ditangani menggunakan `async/await syntax` atau `promise chaining`

untuk memastikan operasi selesai sebelum eksekusi kode dilanjutkan. Untuk mengenkapsulasi logika interaksi dengan SecretStorage API, dibuat sebuah service class bernama *SecureStorageService* yang menyediakan abstraction layer dengan method-method yang lebih semantik dan mudah digunakan oleh komponen lain dalam plugin. Class ini diimplementasikan dalam file *src/services/SecureStorageService.ts* dengan menggunakan TypeScript untuk mendapatkan keuntungan type safety dan auto-completion. Pada constructor class, instance dari *vscode.SecretStorage* disimpan sebagai private property yang diinisialisasi dari ExtensionContext yang diterima sebagai dependency injection. Pendekatan dependency injection ini memudahkan testing karena memungkinkan mock object untuk diberikan pada saat unit testing tanpa harus mengakses actual SecretStorage implementation. Konstanta *OPENROUTER\_API\_KEY* didefinisikan sebagai *static readonly property* dengan value string '*openrouter\_api\_key*' yang berfungsi sebagai key identifier untuk menyimpan dan mengambil API key dari storage. Penggunaan konstanta ini mencegah typo dan memastikan konsistensi key identifier di seluruh codebase.

Method *storeApiKey()* diimplementasikan dengan menerima parameter string berupa API key yang akan disimpan, kemudian memanggil *this.secrets.store()* dengan key identifier yang telah didefinisikan. Error handling diterapkan menggunakan *try-catch block* untuk menangkap potensi error, di mana error akan di-log ke console dan di-throw kembali sebagai Error object dengan message yang lebih mudah dipahami. Method *getApiKey()* diimplementasikan tanpa parameter dan mengembalikan Promise yang resolve ke *string* atau *undefined*. Implementasi method ini memanggil *this.secrets.get()* dan menambahkan logging ketika API key tidak ditemukan dalam storage, memberikan visibilitas terhadap state storage untuk keperluan debugging. Return type yang menggunakan union type “*string | undefined*” memaksa caller untuk melakukan *null-checking* sebelum menggunakan API key, mengurangi potensi error saat runtime akibat mengakses *undefined* value. Method *deleteApiKey()* diimplementasikan dengan memanggil *this.secrets.delete()* dan menangani errors

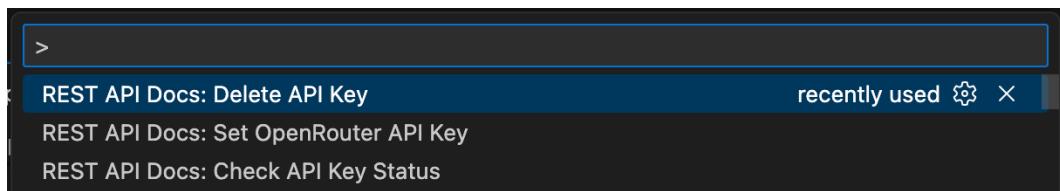
dengan pola yang sama seperti method lainnya, memastikan konsistensi error handling di seluruh service class. Sebagai utility method tambahan, *hasApiKey()* diimplementasikan untuk menyediakan cara yang convenient untuk memeriksa eksistensi API key tanpa harus mengambil actual value-nya. Method ini memanggil *getApiKey()* secara internal dan mengembalikan boolean result berdasarkan apakah value yang dikembalikan adalah undefined atau string dengan length lebih besar dari nol. Implementasi ini membantu dalam flow control di berbagai bagian plugin yang perlu memvalidasi keberadaan API key sebelum melakukan operasi yang memerlukan autentikasi.



Gambar 4.2.9 Arsitektur Class SecureStorageService

Setelah implementasi class selesai, dilakukan pengujian persistence untuk memverifikasi bahwa API key tetap tersimpan bahkan setelah VS Code direstart atau ekstensi di-reload. Pengujian dilakukan dengan menyimpan dummy API key menggunakan command yang memanggil *storeApiKey()*, kemudian merestart VS

Code dan mencoba mengambil API key menggunakan command yang memanggil `getApiKey()`.



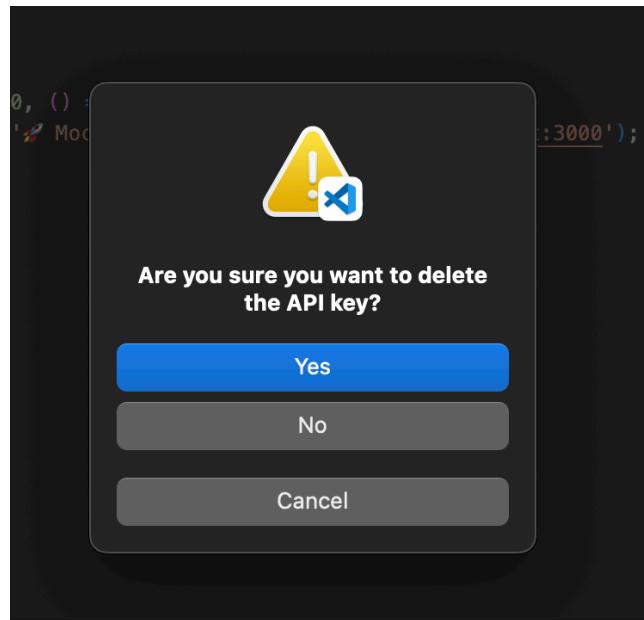
Gambar 4.2.10 Command Set API Key



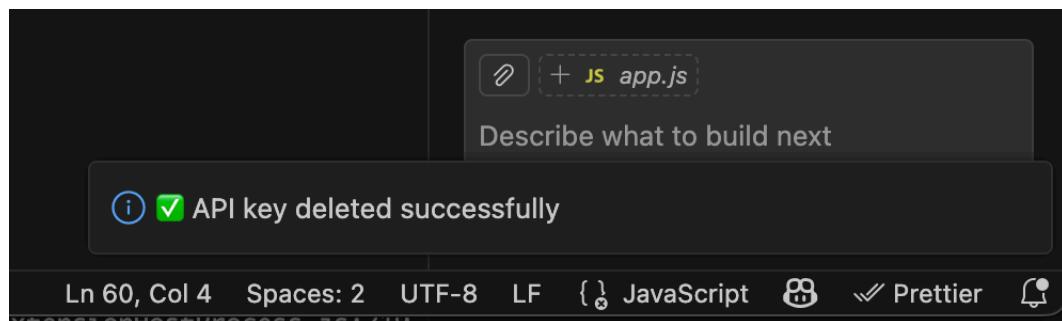
Gambar 4.2.11 Notifikasi Set API Key



Gambar 4.2.12 Notifikasi Cek API Key



Gambar 4.2.13 Verifikasi Notifikasi Hapus API Key



Gambar 4.2.14 Notifikasi Hapus API Key

Hasil pengujian menunjukkan bahwa API key berhasil di-retrieve dengan value yang sama seperti yang disimpan sebelumnya, memvalidasi bahwa SecretStorage API berfungsi sebagaimana yang diharapkan dan data tersimpan secara persisten di storage backend sistem operasi. Handling untuk edge cases juga diimplementasikan untuk meningkatkan robustness service. Skenario edge case yang dipertimbangkan mencakup kondisi di mana storage backend tidak tersedia karena permission issue, storage full, atau corruption pada database credential manager. Untuk setiap skenario, error message yang descriptive di-log ke console dan error di-throw ke caller agar dapat ditangani di layer yang lebih tinggi, misalnya

dengan menampilkan error notification kepada user atau melakukan retry dengan exponential backoff. Pendekatan error handling yang granular ini penting untuk memberikan user experience yang baik ketika terjadi masalah, dibandingkan dengan silent failure yang membuat debugging menjadi sulit. Dengan selesainya implementasi *SecureStorageService*, plugin telah memiliki infrastruktur yang aman untuk mengelola API key tanpa risiko kebocoran melalui source code atau configuration files. Seluruh perubahan di-commit ke repositori dengan pesan "feat: implement secure API key storage using VS Code SecretStorage API" untuk menjaga tracking terhadap setiap perubahan signifikan dalam proyek.

#### 4.2.5 Implementasi Code Parser Module

Modul parser merupakan komponen fundamental dalam sistem dokumentasi otomatis yang bertanggung jawab untuk menganalisis struktur kode sumber dan mengekstraksi informasi routing dari aplikasi Express.js. Tanpa kemampuan parsing yang akurat, sistem tidak akan mampu mengidentifikasi endpoint API, parameter, middleware, dan response patterns yang menjadi basis pembangkitan dokumentasi. Implementasi parser module dilakukan dengan memanfaatkan *Abstract Syntax Tree (AST)* analysis, sebuah teknik parsing tingkat lanjut yang memungkinkan analisis struktur kode secara semantik tanpa harus menjalankan kode tersebut. Pendekatan berbasis AST dipilih karena kemampuannya menangani variasi sintaks JavaScript dan TypeScript yang kompleks, termasuk nested routers, dynamic route definitions, dan penggunaan `async/await` patterns yang umum ditemukan dalam aplikasi Express modern.

Tahap awal implementasi dimulai dengan instalasi dependensi *Babel ecosystem* yang menyediakan parser dan traversal utilities untuk analisis AST (*Abstract Syntax Tree*). Library yang diinstal mencakup `@babel/parser` untuk mengonversi source code menjadi AST representation, `@babel/traverse` untuk navigasi dan inspeksi node-node dalam AST, serta `@babel/types` untuk type checking dan validasi node types. Instalasi juga mencakup type definitions untuk TypeScript development, yaitu `@types/babel_parser`, `@types/babel_traverse`,

dan `@types/babel_core`, yang menyediakan IntelliSense dan type safety saat mengimplementasikan parser logic. Pemilihan Babel sebagai parsing engine didasarkan pada dukungan komprehensif terhadap ECMAScript modern features termasuk ES6+ syntax, JSX, TypeScript, dan experimental proposals yang mungkin digunakan dalam codebase yang sedang dianalisis. Sebelum mengimplementasikan parser logic, terlebih dahulu didefinisikan structure data yang akan digunakan untuk merepresentasikan hasil parsing melalui file `src/types/RouteInfo.ts`. File ini berisi serangkaian interface TypeScript yang mendefinisikan kontrak data antara parser module dan komponen lain dalam sistem. Interface `HttpMethod` didefinisikan sebagai enum yang mencakup semua HTTP methods yang didukung oleh Express, yaitu GET, POST, PUT, DELETE, PATCH, OPTIONS, dan HEAD. Interface `RouteParameter` didefinisikan untuk merepresentasikan parameter endpoint dengan properties seperti name, type (path/query/body), required status, data type, schema untuk complex types, dan example value. Interface `RouteResponse` didesain untuk menyimpan informasi tentang possible responses dari endpoint, mencakup status code, description, content type, dan schema. Interface `MiddlewareInfo` merepresentasikan middleware functions yang terpasang pada route dengan properties name dan type (auth/validation/custom). Interface utama `RouteInfo` menggabungkan semua informasi yang terekstraksi dari single route definition, termasuk method, path, handler name, description, parameters array, responses array, middlewares array, file path, dan line number untuk traceability.

Implementasi dimulai dengan pembuatan class `FileScanner` dalam file `src/parsers/FileScanner.ts` yang bertanggung jawab untuk menemukan dan membaca file-file route dalam workspace. Class ini menyediakan method `scanWorkspace()` yang melakukan recursive directory traversal untuk menemukan file dengan pattern tertentu, umumnya file-file dalam direktori `routes/` atau `src/routes/` dengan extension .js atau .ts. Scanning process menggunakan `Node.js File System API` untuk membaca direktori konten dan melakukan filter berdasarkan file extension dan naming patterns. Method `isRouteFile()` diimplementasikan untuk

mengidentifikasi keberadaan import statement untuk Express router atau penggunaan router.get(), router.post(), dan method routing lainnya. Pendekatan heuristic ini mengurangi false positives dengan memastikan bahwa hanya file yang benar-benar mengandung route definitions yang akan diproses lebih lanjut oleh parser. Core parsing logic diimplementasikan dalam class *RouteParser* yang terletak di file *src/parsers/RouteParser.ts*. Class ini menerima file content sebagai string input dan menghasilkan *array of RouteInfo objects* sebagai output. Method utama *parseFile()* memulai proses dengan memanggil Babel parser untuk mengonversi source code menjadi AST menggunakan configuration yang mendukung berbagai syntax features seperti TypeScript, JSX, dan dynamic imports. Hasil parsing berupa object tree yang merepresentasikan struktur syntactic dari code, di mana setiap node dalam tree memiliki type spesifik seperti *CallExpression*, *MemberExpression*, *ArrowFunctionExpression*, dan lain sebagainya. Traversal terhadap AST dilakukan menggunakan *@babel/traverse* dengan visitor pattern, di mana callback functions didefinisikan untuk setiap node type yang relevan. Visitor untuk *CallExpression nodes* diimplementasikan untuk mendeteksi pemanggilan method routing seperti router.get(), router.post(), dan variannya. Detection logic memeriksa apakah target fungsi atau method yang sedang dipanggil dari CallExpression merupakan MemberExpression dengan object bernama router atau app dan property yang match dengan HTTP method names. Apabila pattern terdeteksi, parser mengekstraksi arguments dari function call, di mana argument pertama merupakan route path dan arguments berikutnya merupakan middleware functions atau route handler. Route path di-parse lebih lanjut untuk mengidentifikasi dynamic segments yang dimulai dengan colon, misalnya :id atau :userId, yang kemudian ditransformasikan menjadi path parameters dalam RouteInfo object.

Ekstraksi middleware information dilakukan dengan menganalisis arguments yang berada di antara route path dan final handler function. Setiap middleware function name dicatat dan dilakukan inference terhadap middleware type berdasarkan naming convention yang umum digunakan dalam Express

ecosystem. Function dengan nama yang mengandung kata auth atau authenticate dikategorikan sebagai authentication middleware, function dengan nama mengandung validate dikategorikan sebagai validation middleware, sementara sisanya dikategorikan sebagai custom middleware. Pendekatan pattern matching sederhana ini memberikan contextual information yang berguna untuk pembangkitan dokumentasi yang lebih deskriptif, meskipun accuracy bergantung pada adherence developer terhadap naming conventions. Handler function analysis dilakukan untuk mengekstraksi additional information seperti response status codes dan response data structures. Parser mencari pemanggilan method res.status(), res.json(), dan res.send() dalam handler body untuk mengidentifikasi possible responses. Apabila ditemukan pattern res.status(200).json, parser mencatat status code 200 dan mencoba menganalisis structure dari object yang diberikan ke method json() untuk inferring response schema. Namun, karena keterbatasan static analysis dalam menangani runtime values, ekstraksi response schema lebih mengandalkan AI inference pada tahap selanjutnya, sementara parser hanya mencatat pattern yang terlihat secara syntactic.

Untuk memastikan correctness implementasi parser, dibuat sample Express project dalam direktori sample-express-project/ yang berisi berbagai route patterns yang umum digunakan dalam aplikasi production. Sample routes mencakup simple GET requests, POST requests dengan body parameters, routes dengan path parameters, routes dengan middleware chains, nested routers, dan async/await handlers. Unit test suite diimplementasikan dalam file `src/test/unit/RouteParser.test.ts` menggunakan testing framework bawaan VS Code extension testing. Test cases memverifikasi bahwa parser dapat mendeteksi semua routes dalam sample project, mengekstraksi HTTP methods dengan benar, mengidentifikasi path parameters, dan mencatat middleware information. Hasil test execution menunjukkan bahwa parser berhasil mengidentifikasi seluruh routes dengan accuracy tinggi, memvalidasi bahwa implementasi AST-based parsing approach telah berfungsi sesuai dengan acceptance criteria yang ditetapkan.

```
=====
REST API ROUTES SCAN RESULTS
=====

Total Files: 2
Total Routes: 9
Errors: 0

=====
File: /Users/alfianwinarso/Documents/rest-api-doc-generator/
 rest-api-doc-generator/sample-express-project/routes/users.js
=====

 GET /
 GET /:id
 Parameters: id
 POST /
 Middlewares: validateUser
 PUT /:id
 Parameters: id
 Middlewares: authenticateUser
 DELETE /:id
 Parameters: id
 Middlewares: authenticateUser, authorizeAdmin

=====
File: /Users/alfianwinarso/Documents/rest-api-doc-generator/
 rest-api-doc-generator/sample-express-project/routes/products.js
=====

 GET /
 GET /:id
 Parameters: id
 POST /
 Middlewares: authenticateUser, validateProduct
 PATCH /:id
 Parameters: id
 Middlewares: authenticateUser
```

Gambar 4.2.15 Sample-express-project

```
Extension Test Suite
 ✓ Sample test
SecureStorageService Test Suite
⚠ SecureStorageService requires manual testing via extension commands
 ✓ SecureStorageService - Manual Testing Required
RouteParser Test Suite
 ✓ Parsed 1 routes from test.js
 ✓ Should parse simple GET route
 ✓ Parsed 1 routes from test.js
 ✓ Should extract path parameters
 ✓ Parsed 4 routes from test.js
 ✓ Should detect multiple HTTP methods
 ✓ Parsed 1 routes from test.js
 ✓ Should extract middlewares
 ✓ Parsed 1 routes from test.js
 ✓ Should handle optional parameters
 7 passing (61ms)
[main 2026-01-28T04:24:33.904Z] Extension host with pid 16969 exited with code: 0, signal: unknown
n.
Exit code: 0
```

Gambar 4.2.16 Unit Test RouteParser.test

Dengan selesainya implementasi parser module, sistem telah memiliki kapabilitas untuk menganalisis codebase Express.js secara otomatis dan

mengekstraksi structured information yang diperlukan sebagai input untuk tahap berikutnya, yaitu AI-powered documentation generation. Seluruh perubahan di-commit ke repositori dengan pesan "feat: Code Parser Module with debugging" untuk menjaga tracking terhadap setiap perubahan signifikan dalam proyek.

#### 4.2.6 Extract Parameters & Response

Setelah modul parser berhasil mendeteksi definisi route dengan akurat, tahap selanjutnya adalah melakukan analisis yang lebih mendalam untuk mengambil informasi detail mengenai parameter dan struktur response. Informasi ini merupakan bagian penting dalam pembuatan dokumentasi API yang lengkap. Proses ekstraksi parameter meliputi identifikasi parameter path, parameter query, dan struktur request body. Sementara itu, proses ekstraksi response mencakup identifikasi kode status HTTP, tipe konten, dan struktur data yang dikembalikan oleh endpoint. Informasi detail ini diperlukan agar dokumentasi OpenAPI yang dihasilkan tidak hanya menunjukkan daftar endpoint, tetapi juga menjelaskan secara lengkap bagaimana endpoint digunakan dan data apa yang akan dikembalikan. Dengan adanya informasi ini, pengguna API dapat melakukan integrasi dengan lebih mudah tanpa harus mencoba-coba sendiri atau bergantung pada penjelasan langsung dari developer backend.

Proses ekstraksi path parameter dilakukan dengan menganalisis teks path route yang telah diperoleh pada tahap sebelumnya. Parser memeriksa setiap bagian path untuk menemukan bagian yang diawali dengan tanda titik dua (:), seperti pada contoh `/users/:id` atau `/posts/:postId/comments/:commentId`. Bagian seperti `:id`, `:postId`, dan `:commentId` disebut sebagai parameter dinamis, karena nilainya akan diisi secara otomatis saat API digunakan. Setiap parameter path yang ditemukan kemudian dicatat ke dalam daftar parameter dengan tipe path dan ditandai sebagai required atau wajib diisi, karena parameter path merupakan bagian yang harus ada dalam URL agar route dapat diakses dengan benar sesuai dengan cara kerja routing pada Express.js.

Tabel 4.2.5 Ekstraksi Path Parameters dengan Pattern Matching

| Route Path                         | Detected Pattern    | Extracted Parameters            | Inferred Type         | Required   |
|------------------------------------|---------------------|---------------------------------|-----------------------|------------|
| /users/:id                         | :id                 | {name: 'id', type: 'path'}      | string                | true       |
| /users/:userId                     | :userId             | {name: 'userId', type: 'path'}  | integer (suffix 'Id') | true       |
| /posts/:postId/comments/:commentId | :postId, :commentId | 2 path params                   | integer, integer      | true, true |
| /api/:version/users                | :version            | {name: 'version', type: 'path'} | string                | true       |

Untuk meningkatkan ketepatan dalam menentukan tipe data parameter, parser menganalisis pola penamaan yang digunakan pada parameter tersebut. Parameter yang memiliki akhiran Id atau ID biasanya diasumsikan sebagai identifier, sehingga diberi tipe data string atau integer. Parameter dengan nama seperti page atau limit diasumsikan bertipe integer karena umumnya digunakan untuk pengaturan halaman atau jumlah data. Sementara itu, parameter lain yang tidak memiliki pola khusus akan diberi tipe string sebagai tipe default yang paling aman. Proses ekstraksi query parameter dilakukan dengan menganalisis isi fungsi handler, khususnya pada bagian yang mengakses objek *req.query*. Parser menelusuri struktur kode dan mencari pola akses terhadap properti dalam *req.query*, baik melalui *destructuring* maupun akses langsung. Sebagai contoh, pada kode *const { page, limit, sort } = req.query* atau *const searchTerm = req.query.q*, parser akan mengidentifikasi page, limit, sort, dan q sebagai query parameter. Berbeda dengan parameter path yang selalu wajib diisi, query parameter pada umumnya bersifat opsional. Oleh karena itu, status required ditetapkan sebagai *false*, kecuali terdapat validasi atau penanganan error yang menunjukkan

bahwa parameter tersebut wajib disertakan. Dalam menentukan tipe data query parameter, parser juga mempertimbangkan cara penggunaannya dalam kode. Sebagai contoh, jika parameter page dikonversi menggunakan fungsi `parseInt()`, maka parameter tersebut diasumsikan bertipe integer, meskipun secara teknis semua query parameter diterima sebagai string pada tingkat protokol HTTP.

Untuk mengekstraksi struktur request body pada request dengan metode POST, PUT, dan PATCH, parser menganalisis bagian kode yang mengakses objek `req.body`. Parser mencari pola pengambilan data dari *request body*, baik melalui *destructuring* seperti `const { name, email, password } = req.body` maupun melalui akses langsung seperti `const userData = req.body`. Setiap field yang ditemukan akan dicatat sebagai parameter body, beserta nama field dan tipe data yang diperkirakan berdasarkan konteks penggunaannya. Jika ditemukan middleware validasi seperti *express-validator*, parser juga mencoba mengambil informasi dari skema validasi tersebut untuk mengetahui field mana yang wajib diisi, tipe data yang digunakan, serta batasan tertentu seperti panjang minimum, maksimum, atau pola format tertentu. Namun, karena validasi dapat bersifat dinamis dan kompleks, proses ekstraksi ini dilakukan se bisa mungkin berdasarkan informasi yang tersedia, dan kekurangan informasi akan dilengkapi oleh proses analisis menggunakan AI pada tahap berikutnya. Ekstraksi informasi response dilakukan dengan menganalisis pemanggilan method response dalam fungsi handler. Parser mencari pemanggilan `res.status()` untuk mengidentifikasi kode status HTTP yang mungkin dikembalikan oleh endpoint. Setiap kode status yang ditemukan akan dicatat beserta konteks penggunaannya. Sebagai contoh, kode `res.status(404).json({ error: 'User not found' })` menunjukkan bahwa endpoint dapat mengembalikan status 404 dengan response dalam format JSON yang berisi pesan error. Parser juga mendeteksi penggunaan `res.json()` dan `res.send()` untuk mengetahui tipe konten response. Method `res.json()` menunjukkan bahwa response menggunakan format JSON, sedangkan `res.send()` dapat digunakan untuk berbagai jenis data tergantung pada nilai yang dikirim. Jika response mengembalikan object secara langsung, parser akan mencoba mengidentifikasi struktur object tersebut untuk membentuk

gambaran awal struktur response. Namun, tingkat ketepatan proses ini memiliki keterbatasan, karena nilai yang dihasilkan saat program berjalan tidak selalu dapat diketahui sepenuhnya hanya dari analisis kode statis.

```

[Extension Development Host] rest-api-doc-generator
PROBLEMS OUTPUT ... REST API Routes Filter (e.g. text, !excludeText, text1,text2)
REST API ROUTES SCAN RESULTS
=====
Total Files: 2
Total Routes: 9
Errors: 0

/ /Users/alfianwinarso/Documents/rest-api-doc-generator/
rest-api-doc-generator/sample-express-project/routes/users.js

GET / Responses: 200 (Success), 500 (Internal Server Error)
GET /:id Path Params: id Responses: 404 (Not Found), 200 (Success), 500 (Internal Server Error)
POST / Middlewares: validateUser Responses: 201 (Created), 400 (Bad Request)
PUT /:id Path Params: id Middlewares: authenticateUser Responses: 200 (Success), 400 (Bad Request)
DELETE /:id Path Params: id Middlewares: authenticateUser, authorizeAdmin Responses: 204 (No Content), 500 (Internal Server Error)

/ /Users/alfianwinarso/Documents/rest-api-doc-generator/
rest-api-doc-generator/sample-express-project/routes/products.js

GET / Responses: 200 (Success), 500 (Internal Server Error)
GET /:id Path Params: id Responses: 200 (Success), 404 (Not Found)
POST / Middlewares: authenticateUser, validateProduct Responses: 201 (Created), 400 (Bad Request)
PATCH /:id Path Params: id Middlewares: authenticateUser Responses: 200 (Success), 400 (Bad Request)

```

Gambar 4.2.17 Identifikasi Status Code

Tabel 4.2.6 Pattern Matching untuk Response Detection

| Pola Kode                   | Status    | Jenis Konten         | Kesimpulan                           |
|-----------------------------|-----------|----------------------|--------------------------------------|
| res.status(200).json({...}) | 200<br>OK | application<br>/json | Struktur objek diambil dari isi JSON |

|                                          |                                  |                      |                                            |
|------------------------------------------|----------------------------------|----------------------|--------------------------------------------|
| res.status(404).json<br>({error: '...'}) | 404<br>Not<br>Found              | application<br>/json | Objek error yang berisi<br>pesan kesalahan |
| res.json({...})                          | 200<br>OK<br>(status<br>default) | application<br>/json | Struktur objek diambil<br>dari isi JSON    |
| res.send('text')                         | 200<br>OK<br>(status<br>default) | text/plain           | Respons berupa teks                        |
| res.status(201).send()                   | 201<br>Created                   | Tidak ada            | Respons tanpa isi<br>(kosong)              |
| res.sendStatus(204)                      | 204 No<br>Content                | Tidak ada            | Tidak memiliki isi<br>respons              |

Penanganan pola *async/await* merupakan bagian penting dalam proses ekstraksi ini, karena sebagian besar fungsi handler dalam aplikasi Express modern menggunakan operasi asynchronous, seperti pengambilan data dari database, pemanggilan API eksternal, atau akses ke sistem file. Parser dirancang agar dapat menelusuri isi fungsi dengan benar, baik untuk fungsi yang berjalan secara langsung (synchronous) maupun yang menggunakan asynchronous. Hal ini mencakup fungsi dengan kata kunci *async*, penggunaan *Promise* dengan metode *.then()* dan *.catch()*, serta pola callback yang masih digunakan pada kode lama. Selain itu, blok penanganan error seperti *try-catch* pada fungsi *async* maupun penggunaan *.catch()* pada *Promise* juga dianalisis untuk mengidentifikasi kemungkinan respons error yang dikembalikan oleh endpoint. Informasi ini digunakan untuk memperkaya dokumentasi dengan berbagai kemungkinan kondisi error yang dapat terjadi. Untuk memastikan bahwa ekstraksi parameter dan respons berjalan dengan benar, dilakukan peningkatan pada kumpulan unit test yang telah dibuat sebelumnya. Berbagai skenario pengujian ditambahkan untuk memverifikasi

bahwa parser dapat mengekstraksi parameter query dari berbagai pola destructuring, mengidentifikasi parameter body pada handler POST dan PUT, mendeteksi beberapa kemungkinan status code pada respons dengan kondisi tertentu, serta menangani pola async/await yang kompleks, termasuk penanganan error bertingkat. Contoh route pada folder *sample-express-project/* juga diperluas dengan endpoint yang memiliki definisi parameter yang lengkap serta berbagai pola respons, sehingga cakupan pengujian menjadi lebih menyeluruh. Hasil pengujian menunjukkan bahwa parser mampu mengekstraksi sebagian besar parameter dan respons dengan tingkat ketepatan yang dapat diterima.

Integrasi antara hasil ekstraksi parameter dan respons dengan struktur RouteInfo dilakukan dengan mengisi daftar parameter dan respons yang telah didefinisikan pada interface tersebut. Setiap objek parameter berisi informasi lengkap seperti nama parameter, jenis parameter (path, query, atau body), status apakah wajib diisi atau tidak, tipe data yang diperkirakan, serta contoh nilai jika tersedia dalam kode. Objek respons mencakup status code, deskripsi yang diperoleh dari pesan error atau indikator keberhasilan, jenis konten yang dikembalikan, serta struktur awal schema yang nantinya akan disempurnakan oleh AI pada tahap berikutnya. Informasi terstruktur ini menjadi dasar dalam pembuatan dokumentasi OpenAPI yang akurat dan mudah digunakan, sehingga pengguna API dapat memahami cara penggunaan endpoint tanpa harus mempelajari langsung kode implementasinya. Dengan selesainya tahap ekstraksi parameter dan respons, modul parser telah berkembang menjadi alat analisis kode yang tidak hanya mendeteksi route, tetapi juga mengekstraksi informasi penting yang dibutuhkan untuk menghasilkan dokumentasi API yang berkualitas. Seluruh peningkatan ini disimpan ke repositori dengan pesan commit "feat: Extract Parameters & Response" untuk memudahkan pelacakan perubahan penting dalam proyek.

#### 4.2.7 OpenRouter API Client

Setelah sistem berhasil mengekstraksi informasi routing dari kode Express.js, tahap selanjutnya adalah membangun mekanisme komunikasi dengan layanan OpenRouter untuk mengakses kemampuan AI model Gemma 3 12B-IT. OpenRouter API Client berfungsi sebagai jembatan antara plugin dengan layanan inferensi AI eksternal, mengelola proses pengiriman data route yang telah diekstraksi dan menerima hasil dokumentasi yang dihasilkan oleh model. Implementasi client yang robust sangat penting untuk memastikan reliabilitas sistem secara keseluruhan, mengingat setiap dokumentasi yang dihasilkan bergantung pada kesuksesan komunikasi dengan layanan eksternal yang berada di luar kontrol langsung pengembang.

Implementasi dimulai dengan pembuatan class *OpenRouterClient* dalam file *src/services/OpenRouterClient.ts* yang mengenkapsulasi seluruh logika komunikasi HTTP dengan API OpenRouter. Class ini menggunakan library Axios sebagai HTTP client karena kemudahannya dalam menangani request dan response, dukungan *built-in* untuk promises yang memudahkan penanganan operasi asynchronous, serta kemampuan interceptor yang berguna untuk logging dan error handling global. Pada constructor class, dilakukan inisialisasi konfigurasi dasar seperti base URL yang mengarah ke endpoint *https://openrouter.ai/api/v1/chat/completions* serta penyimpanan API key yang diperoleh dari *SecureStorageService*. Pendekatan dependency injection digunakan untuk API key, di mana key diberikan saat instansiasi object sehingga memudahkan testing dengan mock credentials tanpa harus mengakses actual storage backend. Konfigurasi timeout merupakan aspek kritis dalam implementasi client mengingat respons dari model AI dapat memerlukan waktu beberapa detik tergantung pada kompleksitas prompt dan beban server. Berdasarkan kebutuhan non-fungsional yang telah ditetapkan pada tahap perancangan, timeout diset selama 30.000 milidetik atau 30 detik untuk memberikan waktu yang cukup bagi server untuk memproses request tanpa membuat user menunggu terlalu lama. Nilai ini dipilih berdasarkan observasi empiris terhadap response time rata-rata OpenRouter yang

berkisar 2-5 detik untuk request normal, dengan buffer tambahan untuk mengakomodasi spike latency yang mungkin terjadi saat server mengalami high load. Apabila timeout terlampaui, client akan mengembalikan error yang dapat ditangani oleh layer yang lebih tinggi, misalnya dengan menampilkan notifikasi kepada user atau melakukan retry dengan backoff strategy. Method utama dalam class ini adalah *generateDocumentation()* yang menerima object RouteInfo sebagai input dan mengembalikan string berisi dokumentasi dalam format OpenAPI yang dihasilkan oleh AI. Method ini pertama-tama membangun request payload yang sesuai dengan spesifikasi OpenRouter API, yang mengadopsi format compatible dengan OpenAI Chat Completions API. Payload berisi field model dengan value *google/gemma-3-12b-it:free* untuk menentukan model yang akan digunakan, field messages yang merupakan array berisi conversation history dengan role user dan content berupa prompt yang telah diformat, serta field max\_tokens untuk membatasi panjang output yang dihasilkan.

Tabel 4.2.7 Struktur Request Payload OpenRouter API

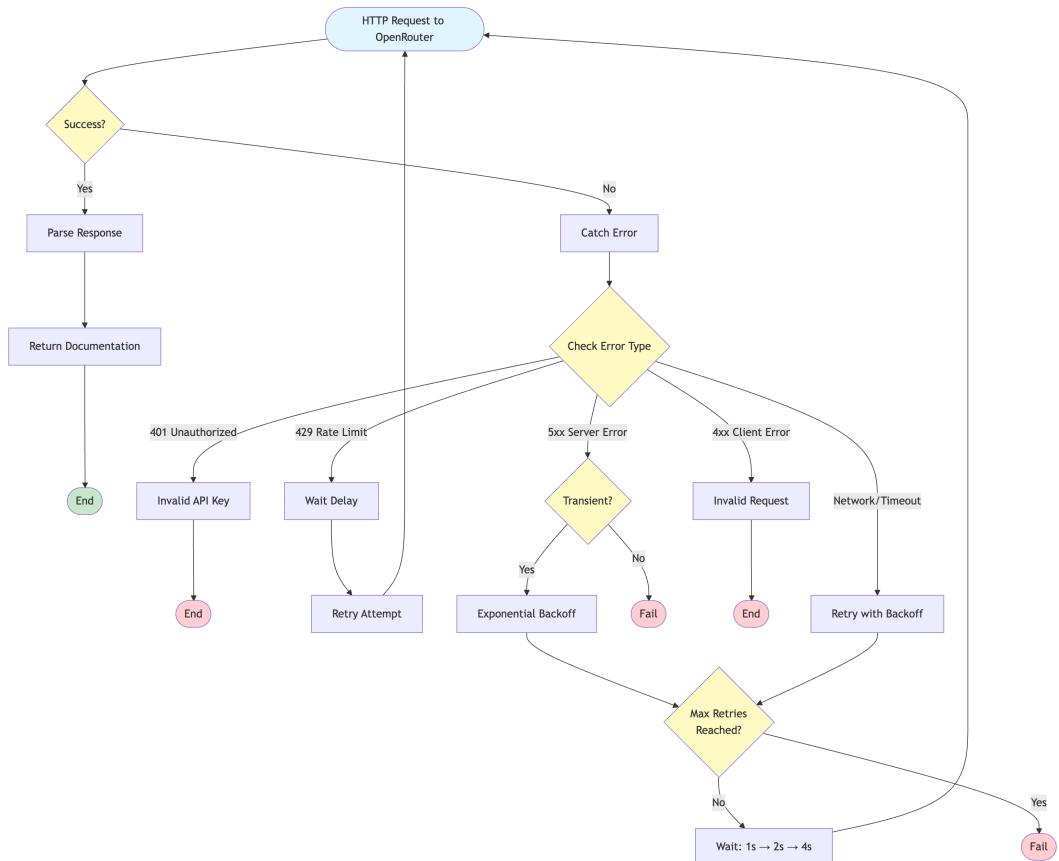
| Field       | Type                | Value                                | Deskripsi                                                                                                                                           |
|-------------|---------------------|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| model       | string              | "google/gemma-3-12b-it:free"         | Identifier model yang digunakan untuk menghasilkan dokumentasi.                                                                                     |
| messages    | array               | [{"role": "user", "content": "..."}] | Riwayat percakapan yang berisi prompt yang dikirim ke model.                                                                                        |
| max_tokens  | integer             | 8000                                 | Batas maksimum panjang output yang dapat dihasilkan oleh model, sesuai dengan limit model.                                                          |
| temperature | float<br>(optional) | 0.7                                  | Mengontrol tingkat variasi output. Nilai lebih rendah menghasilkan output lebih konsisten, nilai lebih tinggi menghasilkan output lebih bervariasi. |

|        |                       |       |                                                                                                                                         |
|--------|-----------------------|-------|-----------------------------------------------------------------------------------------------------------------------------------------|
| stream | boolean<br>(optional) | false | Menentukan apakah output dikirim secara bertahap (streaming). Nilai false berarti output dikirim sekaligus untuk penyederhanaan proses. |
|--------|-----------------------|-------|-----------------------------------------------------------------------------------------------------------------------------------------|

Secara garis besar prompt berisi informasi route yang telah diekstraksi beserta instruksi untuk menghasilkan dokumentasi OpenAPI yang valid. Authentication dilakukan dengan menyertakan API key dalam HTTP header Authorization dengan format Bearer [API\_KEY], mengikuti standar OAuth 2.0 Bearer Token yang umum digunakan dalam REST API modern. Header *Content-Type* diset sebagai *application/json* untuk mengindikasikan bahwa request body berformat JSON, sementara header optional HTTP-Referer dapat ditambahkan untuk mengidentifikasi aplikasi dalam leaderboard OpenRouter apabila diperlukan untuk monitoring atau analytics purposes. Konfigurasi header yang proper memastikan bahwa server dapat memproses request dengan benar dan memberikan respons yang sesuai dengan ekspektasi client. Penanganan respons dilakukan dengan mem-parsing JSON response yang dikembalikan oleh server dan mengekstraksi content dari field *choices[0].message.content* yang berisi teks dokumentasi yang dihasilkan oleh model. Response structure mengikuti format OpenAI-compatible yang terdiri dari object dengan *field id, object, created, model,* dan *choices* yang merupakan array berisi possible completions. Dalam implementasi ini hanya completion pertama yang digunakan mengingat request tidak mengaktifkan multiple completions. Apabila response mengandung field usage, informasi mengenai token consumption dapat di-log untuk monitoring purposes, membantu developer memahami cost implications dan mengoptimalkan prompt untuk efisiensi token usage.

Error handling merupakan komponen krusial dalam implementasi client mengingat berbagai failure scenarios yang mungkin terjadi dalam komunikasi

jaringan. Implementasi menggunakan try-catch block untuk menangkap exceptions yang mungkin di-throw oleh Axios, dengan penanganan yang berbeda berdasarkan sumber error. Error yang berasal dari HTTP response dengan status code 4xx atau 5xx ditangani dengan mengekstraksi error message dari response body dan membungkusnya dalam custom error object yang informatif. Error dengan status 401 mengindikasikan invalid API key dan memicu suggestion untuk user memeriksa kredensial mereka. Error dengan status 429 mengindikasikan rate limiting dan dapat di-retry setelah delay tertentu. Error dengan status 500-599 mengindikasikan server-side issues yang mungkin transient, sehingga retry strategy dengan exponential backoff dapat diterapkan untuk meningkatkan success rate. Retry logic diimplementasikan untuk menangani transient failures yang umum terjadi dalam komunikasi jaringan, seperti temporary network disruptions, server restarts, atau momentary overload. Strategi yang digunakan adalah exponential backoff dengan maximum 3 retry attempts, di mana delay antara retry meningkat secara eksponensial untuk menghindari overwhelming server yang sedang recovery. Delay pertama diset 1 detik, retry kedua menunggu 2 detik, dan retry ketiga menunggu 4 detik sebelum akhirnya mengembalikan error apabila semua attempts gagal. Implementasi retry hanya dilakukan untuk errors yang bersifat transient seperti network timeouts atau server errors 5xx, sementara client errors 4xx tidak di-retry karena mengindikasikan invalid request yang tidak akan berhasil meskipun di-retry.



Gambar 4.2.18 Flowchart Error Handling dengan Retry Logic

Logging ditambahkan untuk visibility terhadap request dan response flows, memudahkan debugging saat terjadi issues. Setiap request yang dikirim di-log dengan informasi route yang sedang diproses, timestamp, dan request ID untuk traceability. Response yang diterima di-log dengan status code, response time, dan snippet dari content untuk memverifikasi bahwa model mengembalikan output yang expected. Error scenarios juga di-log dengan full stack trace dan context information untuk memudahkan root cause analysis.

Dengan selesainya implementasi OpenRouter API Client, sistem telah memiliki infrastruktur komunikasi yang reliable untuk berinteraksi dengan layanan AI eksternal. Testing dilakukan dengan mengirimkan sample route information dan memverifikasi bahwa respons yang diterima berupa valid OpenAPI YAML atau

JSON. Seluruh implementasi di-commit dengan pesan "feat: OpenRouter API Client" untuk dokumentasi progres pengembangan.

#### 4.2.8 Prompt Engineering for Gemma 3

Implementasi prompt engineering dilakukan melalui pembuatan class *PromptBuilder* dalam file *src/utils/PromptBuilder.ts* yang berfungsi sebagai template engine untuk menghasilkan prompt yang konsisten dan teroptimasi. Class ini dirancang dengan static methods untuk memudahkan penggunaan tanpa perlu instansiasi object, mengikuti pattern utility class yang umum digunakan dalam TypeScript development. Terdapat dua method utama yang disediakan, yaitu *buildRoutePrompt()* untuk single route documentation dan *buildMultipleRoutesPrompt()* untuk batch processing multiple routes dalam satu request, memberikan fleksibilitas dalam handling berbagai skenario dokumentasi. Struktur prompt dirancang menggunakan modular approach dengan memecah prompt menjadi komponen-komponen terpisah yang kemudian digabungkan menjadi complete prompt string. Pendekatan ini memudahkan maintenance dan memungkinkan fine-tuning individual components tanpa harus merevisi semua template prompt. Komponen pertama adalah system context yang didefinisikan melalui method *getSystemContext()*, memberikan role definition kepada AI dengan kalimat "*You are an expert API documentation assistant specialized in generating OpenAPI 3.1 specifications from Express.js code.*" Role definition yang spesifik ini penting untuk membingkai perspektif AI agar fokus pada domain yang relevan dan mengadopsi tone technical documentation yang appropriate. Task instruction disediakan melalui method *getTaskInstruction()* yang secara eksplisit menyatakan "*TASK: Generate accurate OpenAPI 3.1 documentation for the following Express.js REST API endpoint(s).*" Penggunaan kata kunci "*TASK*" dengan uppercase deliberate untuk memberikan emphasis pada instruction section, membantu model membedakan antara context setting dan actual task yang harus dilakukan. Instruksi yang clear dan directive ini mengurangi ambiguity dan meningkatkan consistency dalam output yang dihasilkan. Route information formatting dilakukan melalui method *getRouteInformation()* yang mengkonversi

RouteInfo object menjadi human-readable structured text. Format yang digunakan adalah *key-value pairs* dengan kapitalisasi label seperti "*METHOD:*", "*PATH:*", "*PARAMETERS:*", dan "*RESPONSES:*" untuk meningkatkan readability. Parameter dikategorikan berdasarkan tipe (path, query, body) dan ditampilkan dalam format terpisah untuk memudahkan AI memahami parameter location dalam HTTP request. Sebagai contoh, untuk route dengan path parameters dan query parameters, output akan berbentuk "*PARAMETERS:*\n *Path:* *id* (*string*)\n *Query:* *page* (*integer*), *limit* (*integer*)". Struktur hierarkis dengan indentasi membantu AI mengidentifikasi grouping dan relationships antar informasi. Middleware information disertakan dalam prompt sebagai contextual hints yang dapat membantu AI menginfer security requirements atau validation constraints. Format sederhana "*MIDDLEWARES: authenticateUser, validateRequest*" memberikan signal bahwa endpoint mungkin memerlukan authentication atau memiliki input validation, meskipun detail implementasi middleware tidak disertakan untuk menghemat tokens. AI dapat menggunakan informasi ini untuk menambahkan security schemes atau validation notes dalam dokumentasi yang dihasilkan.

Teknik few-shot learning diterapkan melalui method *getFewShotExample()* dengan menyediakan satu contoh lengkap berupa pasangan input dan output. Contoh yang digunakan adalah endpoint dengan method GET yang memiliki path parameter, karena pola ini merupakan salah satu bentuk yang paling umum digunakan dalam REST API. Contoh output menampilkan objek path OpenAPI secara lengkap dalam format YAML, yang mencakup seluruh elemen penting seperti ringkasan (*summary*), deskripsi (*description*), *operationId*, *tag*, definisi parameter, serta definisi response beserta struktur datanya. Penggunaan satu contoh saja (*one-shot learning*), dibandingkan dengan beberapa contoh sekaligus, dilakukan untuk menghemat penggunaan token, mengingat adanya batasan jumlah token yang dapat diproses dalam satu permintaan. Selain itu, penggunaan satu contoh berkualitas tinggi dinilai sudah cukup untuk membantu model memahami format dan struktur output yang diharapkan. Hasil pengujian menunjukkan bahwa

satu contoh yang jelas dan lengkap sudah memadai untuk membantu model menghasilkan dokumentasi yang konsisten dan sesuai dengan kebutuhan.

Tabel 4.2.8 Few-shot Learning

```
- \```\`yaml
- /users/{id}:
- get:
- summary: Get user by ID
- description: Retrieve detailed information about a
- specific user
- operationId: getUserId
- tags:
- - Users
- parameters:
- - name: id
- in: path
- required: true
- description: User ID
- schema:
- type: string
- responses:
- '200':
- description: Successful response
- content:
- application/json:
- schema:
- type: object
- properties:
- id:
- type: string
- name:
- type: string
- email:
- type: string
- '404':
- description: User not found
- content:
- application/json:
- schema:
- type: object
- properties:
- message:
- type: string
- \````
```

Output format requirements didefinisikan secara eksplisit melalui method `getOutputFormat()` yang berisi numbered list dari constraints yang harus diikuti oleh AI. Requirements mencakup "*Generate ONLY the OpenAPI path object in YAML format*", "*Include accurate parameter definitions with proper types*", "*Include all relevant response status codes*", dan "*Do NOT include any explanations, just the YAML content*". Constraint terakhir sangat penting karena tanpa explicit instruction, model cenderung menambahkan explanatory text sebelum atau sesudah YAML output, yang dapat menyulitkan parsing pada tahap post-processing.

Handling terhadap AI response dilakukan melalui utility methods `extractYAML()` dan `validateResponse()`. Method `extractYAML()` melakukan cleaning terhadap response string dengan menghilangkan markdown code block markers seperti triple backticks yang sering ditambahkan oleh model saat generating code atau structured data. Regex patterns digunakan untuk mendekripsi dan menghapus markers dengan format ``yaml, ``yml, atau generic ```di awal dan akhir response. Method `validateResponse()` melakukan basic validation untuk memverifikasi bahwa response mengandung OpenAPI fields yang expected seperti `"summary:"` dan `"responses:"`. Untuk mendukung proses pengujian, disediakan utilitas melalui method `buildTestPrompt()` yang menghasilkan prompt sederhana untuk menguji koneksi ke API dan memastikan bahwa model dapat memberikan respons dengan benar. Prompt pengujian ini sangat berguna selama tahap pengembangan untuk memastikan bahwa OpenRouter client berfungsi dengan baik dan model mampu menghasilkan struktur dasar OpenAPI tanpa memerlukan informasi route secara lengkap.

The screenshot shows a code editor interface with a dark theme. At the top, there is a search bar with the text "[Extension Development Host] rest-api-doc-generator". Below the search bar, there are tabs labeled "PROBLEMS", "OUTPUT" (which is selected), and "...". To the right of the tabs is a dropdown menu set to "Prompt Test Result". Further right are icons for filtering, expanding/collapsing, and closing the panel.

The main content area displays the generated OpenAPI documentation:

```
=====
GENERATED OPENAPI DOCUMENTATION
=====

/users/{id}:
get:
 summary: Get user by ID
 description: Retrieve detailed information about a specific user.
 operationId: getUserId
 tags:
 - Users
 parameters:
 - name: id
 in: path
 required: true
 description: User ID
 schema:
 type: string
 responses:
 '200':
 description: Successful response
 content:
 application/json:
 schema:
 type: object
 properties:
 id:
 type: string
 description: User ID
 name:
 type: string
 description: User Name
 email:
 type: string
 description: User Email
 '404':
 description: User not found
 content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 description: Error message
```

Gambar 4.2.19 Pengujian Hasil Prompt Sederhana

Respons yang dihasilkan dari prompt pengujian ini dapat diperiksa secara manual untuk menilai kualitas output dan mengidentifikasi bagian prompt yang masih dapat diperbaiki. Selain itu, dilakukan optimasi penggunaan token pada setiap bagian template prompt. Informasi yang berulang atau tidak memberikan

kontribusi signifikan terhadap kualitas output dihilangkan. Format prompt juga dirancang agar tetap mudah dibaca namun tetap ringkas, termasuk penggunaan singkatan pada label selama tidak mengurangi kejelasan informasi.

Tabel 4.2.9 Analisis Penggunaan Token Untuk Endpoint

| Endpoint Type                           | Input Tokens | Output Tokens | Total Tokens |
|-----------------------------------------|--------------|---------------|--------------|
| Simple GET (tanpa parameter)            | 280          | 350           | 630          |
| GET dengan path dan query parameter     | 320          | 480           | 800          |
| POST dengan body parameter              | 380          | 550           | 930          |
| Endpoint kompleks dengan middleware     | 450          | 720           | 1,170        |
| Batch processing (5 endpoint sekaligus) | 1,200        | 2,400         | 3,600        |
| Rata-rata satu endpoint                 | ~350         | ~525          | ~875         |

Dengan implementasi PromptBuilder yang terstruktur dan telah melalui pengujian, sistem memiliki dasar yang kuat untuk menghasilkan prompt yang konsisten dan berkualitas tinggi. Pendekatan berbasis template juga memungkinkan perbaikan dilakukan secara bertahap di masa mendatang tanpa perlu mengubah keseluruhan sistem. Seluruh implementasi ini disimpan dalam sistem version control dengan pesan commit "feat: Prompt Engineering" untuk mendokumentasikan perkembangan sistem.

#### 4.2.9 OpenAPI Document Generator

Setelah sistem berhasil mengambil informasi route dan menghasilkan dokumentasi untuk setiap route menggunakan AI, tahap berikutnya adalah

menggabungkan seluruh dokumentasi tersebut ke dalam satu dokumen OpenAPI 3.1 yang lengkap dan valid. OpenAPI Document Generator berperan sebagai komponen utama yang menggabungkan hasil dokumentasi dari AI dengan informasi proyek, menyusun daftar path endpoint, mengekstrak struktur data yang dapat digunakan kembali, serta menghasilkan file dokumentasi akhir dalam format YAML dan JSON. Generator ini memastikan bahwa dokumen yang dihasilkan bukan hanya berupa daftar endpoint, tetapi merupakan spesifikasi API yang lengkap dan siap digunakan untuk berbagai keperluan, seperti pembuatan client API, pengujian, maupun publikasi dokumentasi.

Implementasi dimulai dengan menginstal pustaka *js-yaml*, yang digunakan untuk mengubah objek JavaScript menjadi format YAML dan sebaliknya. Format YAML dipilih sebagai format utama karena lebih mudah dibaca oleh manusia dibandingkan JSON, serta mendukung penulisan teks dokumentasi yang lebih jelas. Instalasi dilakukan menggunakan perintah *npm install js-yaml*, kemudian dilanjutkan dengan instalasi dukungan TypeScript melalui *npm install --save-dev @types/js-yaml*. Pustaka ini telah banyak digunakan dalam lingkungan Node.js dan terbukti stabil untuk pengolahan file YAML. Kelas *OpenAPIGenerator* dibuat dalam file *src/generators/OpenAPIGenerator.ts* untuk menangani proses pembuatan dokumen OpenAPI. *Constructor* pada kelas ini menerima informasi proyek seperti judul, versi, dan deskripsi, yang akan digunakan sebagai bagian informasi utama dalam dokumen OpenAPI. Di dalam kelas ini, dokumen OpenAPI disimpan dalam bentuk objek yang mengikuti struktur standar OpenAPI 3.1, yang terdiri dari beberapa bagian seperti versi OpenAPI, informasi API, daftar server, daftar endpoint (paths), komponen, dan pengaturan keamanan. Struktur ini akan diisi secara bertahap saat setiap route ditambahkan. Struktur dasar dokumen dibuat melalui method *initializeDocument()*, yang dipanggil saat objek generator dibuat. Struktur ini mencakup versi OpenAPI yang digunakan, informasi API seperti judul dan versi, serta daftar server yang menunjukkan alamat dasar API. Secara default, alamat server diatur ke *http://localhost:3000* untuk kebutuhan pengembangan, namun dapat diubah sesuai kebutuhan, misalnya untuk server pengujian atau

produksi. Informasi tambahan seperti kontak dan lisensi juga dapat ditambahkan untuk melengkapi metadata dokumentasi.

Method *addRoute()* digunakan untuk menambahkan definisi route ke dalam dokumen OpenAPI. Method ini menerima objek *RouteInfo* yang berisi informasi route yang telah diperoleh dari parser. Berdasarkan informasi tersebut, sistem akan membangun struktur endpoint sesuai dengan format OpenAPI, termasuk method HTTP seperti GET atau POST, deskripsi endpoint, parameter, request body jika ada, serta daftar kemungkinan response. Proses ini dilakukan secara otomatis dengan menyesuaikan data route ke dalam struktur OpenAPI yang sesuai. Selain itu, terdapat method *addRouteWithAIDoc()*, yang digunakan untuk menangani route yang dokumentasinya dihasilkan oleh AI. Method ini menerima informasi route dan dokumentasi dalam format YAML yang dihasilkan oleh AI. Sistem kemudian membaca dan menggabungkan hasil dokumentasi tersebut ke dalam dokumen OpenAPI utama. Jika terjadi kesalahan pada format YAML yang dihasilkan AI, sistem akan menggunakan metode alternatif untuk membuat dokumentasi secara manual agar proses tetap dapat berjalan dengan baik.

Setelah semua route ditambahkan, method *finalizeDocument()* dipanggil untuk menyempurnakan dokumen. Proses ini mencakup pengurutan daftar endpoint agar tersusun rapi, memastikan semua bagian penting telah tersedia, menghapus bagian yang kosong atau tidak digunakan, serta memastikan format dokumen konsisten. Selain itu, sistem juga mendeteksi struktur data yang sama agar tidak ditulis berulang, sehingga ukuran dokumen menjadi lebih efisien. Dokumen OpenAPI kemudian dapat dihasilkan dalam dua format, yaitu YAML dan JSON. Method *toYAML()* digunakan untuk menghasilkan dokumen dalam format YAML dengan struktur yang rapi dan mudah dibaca. Method ini menggunakan fungsi dari pustaka *js-yaml* untuk mengubah objek dokumen menjadi teks YAML. Sedangkan method *toJSON()* digunakan untuk menghasilkan dokumen dalam format JSON dengan format yang juga mudah dibaca. Kedua method ini menghasilkan teks yang dapat langsung disimpan ke file atau digunakan oleh sistem lain.

Tabel 4.2.10 Ringkasan Fungsi Method OpenAPIGenerator

| Method                      | Input                                       | Output                     | Fungsi Utama                                                                                                                  |
|-----------------------------|---------------------------------------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>constructor()</b>        | ProjectInfo<br>(title,<br>version,<br>desc) | -                          | Menginisialisasi generator<br>dengan metadata proyek dan<br>menyiapkan struktur awal<br>dokumen OpenAPI                       |
| <b>initializeDocument()</b> | -                                           | -                          | Membangun struktur dasar<br>dokumen OpenAPI yang<br>berisi info, servers, dan paths<br>kosong                                 |
| <b>addRoute()</b>           | RouteInfo<br>object                         | -                          | Membuat dan menambahkan<br>definisi path secara manual<br>berdasarkan hasil parsing<br>route                                  |
| <b>addRouteWithAIDoc()</b>  | RouteInfo,<br>AI YAML<br>string             | -                          | Mengubah YAML hasil AI<br>menjadi object, lalu<br>menggabungkannya ke dalam<br>paths dokumen OpenAPI                          |
| <b>finalizeDocument()</b>   | -                                           | -                          | Menyelesaikan dokumen<br>dengan mengurutkan paths,<br>mengekstraksi schema,<br>menghapus duplikasi, dan<br>melakukan validasi |
| <b>toYAML()</b>             | -                                           | string<br>(format<br>YAML) | Mengubah dokumen OpenAPI<br>menjadi format YAML<br>menggunakan fungsi js-<br>yaml.dump()                                      |

|                 |   |                            |                                                                                       |
|-----------------|---|----------------------------|---------------------------------------------------------------------------------------|
| <b>toJSON()</b> | - | string<br>(format<br>JSON) | Mengubah dokumen OpenAPI menjadi format JSON dengan format yang rapi dan mudah dibaca |
|-----------------|---|----------------------------|---------------------------------------------------------------------------------------|

Sistem juga dilengkapi dengan penanganan kondisi khusus untuk mengantisipasi berbagai kemungkinan masalah. Jika terdapat route dengan path dan method yang sama, sistem dapat dikonfigurasi untuk mengganti definisi lama, menggabungkan informasi, atau menampilkan kesalahan. Selain itu, jika suatu route tidak memiliki definisi response, sistem secara otomatis menambahkan response default dengan status kode 200 OK. Hal ini dilakukan untuk memastikan bahwa dokumen tetap valid dan dapat digunakan, meskipun informasi response belum lengkap.

Pengujian dilakukan dengan membuat berbagai contoh route, seperti route GET sederhana, route POST dengan struktur data yang kompleks, route dengan parameter pada URL, serta route dengan response yang memiliki struktur bertingkat. Pengujian ini bertujuan untuk memastikan bahwa generator dapat menangani berbagai jenis route dengan benar. Selain itu, sistem juga mendukung penambahan informasi tambahan untuk memperkaya dokumentasi. Informasi tersebut meliputi tag untuk mengelompokkan endpoint berdasarkan kategori tertentu, tautan ke dokumentasi tambahan, serta contoh nilai parameter dan response. Penggunaan tag memudahkan pengguna dalam menavigasi dokumentasi, terutama jika jumlah endpoint cukup banyak. Tautan dokumentasi tambahan dapat digunakan untuk memberikan penjelasan lebih rinci. Contoh nilai parameter dan response membantu pengguna memahami cara menggunakan API tanpa harus langsung mencobanya.

Tabel 4.2.11 Contoh users.js

```
- const express = require('express');
const router = express.Router();

// GET /users - Get all users
router.get('/users', async (req, res) => {
 try {
 const users = await getAllUsers();
 res.status(200).json(users);
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});

// GET /users/:id - Get user by ID
router.get('/users/:id', async (req, res) => {
 try {
 const user = await getUserById(req.params.id);
 if (!user) {
 return res.status(404).json({ message: 'User not found' });
 }
 res.status(200).json(user);
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});

// POST /users - Create new user
router.post('/users', validateUser, async (req, res) => {
 try {
 const newUser = await createUser(req.body);
 res.status(201).json(newUser);
 } catch (error) {
 res.status(400).json({ message: error.message });
 }
});

// PUT /users/:id - Update user
router.put('/users/:id', authenticateUser, async (req, res) => {
 try {
 const updatedUser = await updateUser(req.params.id,
 req.body);
 res.status(200).json(updatedUser);
 } catch (error) {
 res.status(400).json({ message: error.message });
 }
});

// DELETE /users/:id - Delete user
router.delete('/users/:id', authenticateUser, authorizeAdmin,
async (req, res) => {
```

```

- try {
- await deleteUser(req.params.id);
- res.status(204).send();
- } catch (error) {
- res.status(500).json({ message: error.message });
- }
-);
-
- module.exports = router;
-
```

Tabel 4.2.12 Hasil Dokumentasi openapi.yaml

```

- openapi: 3.1.0
- info:
- title: REST API Documentation
- version: 1.0.0
- description: Auto-generated API documentation using AI
- servers:
- - url: http://localhost:3000
- description: Development server
- paths:
- /users/{id}:
- get:
- summary: Get user by ID
- description: Retrieve detailed information about a
- specific user.
- operationId: getUserId
- tags:
- - Users
- parameters:
- - name: id
- in: path
- required: true
- description: User ID
- schema:
- type: string
- responses:
- '200':
- description: Successful operation - user found
- content:
- application/json:
- schema:
- type: object
- properties:
- id:
- type: string
- description: The user's ID.
- name:
- type: string
- description: The user's name.
-
```

```

 email:
 type: string
 description: The user's email address.
 '404':
 description: User not found
 content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 description: Error message indicating user
not found.
 '500':
 description: Internal Server Error
 content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 description: Error message indicating an
internal server error.
 put:
 summary: Update user by ID
 description: Update an existing user by their ID.
 Requires authentication.
 operationId: updateUserById
 tags:
 - Users
 parameters:
 - name: id
 in: path
 required: true
 description: User ID to update
 schema:
 type: string
 responses:
 '200':
 description: User updated successfully
 content:
 application/json:
 schema:
 type: object
 properties:
 id:
 type: string
 description: The ID of the updated user
 name:
 type: string
 description: The updated user's name
 email:
 type: string

```

```

 description: The updated user's email
'400':
 description: Bad Request - Invalid input
 content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 description: Error message describing the
invalid input
 delete:
 summary: Delete a user by ID
 description: Delete a user from the system. Requires
admin privileges.
 operationId: deleteUserById
 tags:
 - Users
 security:
 - bearerAuth: []
 parameters:
 - name: id
 in: path
 required: true
 description: User ID to delete
 schema:
 type: string
 responses:
 '204':
 description: User successfully deleted. No content
returned.
 '500':
 description: Internal Server Error
 content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 description: Error message
 tags:
 - name: Users
 description: Users related endpoints

```

Dengan selesainya implementasi OpenAPI Document Generator, sistem telah mampu menghasilkan dokumentasi API yang lengkap dan sesuai dengan standar OpenAPI. Integrasi antara parser, AI client, dan generator membentuk alur kerja yang terstruktur, dimulai dari analisis kode sumber hingga menghasilkan file

dokumentasi akhir dalam format YAML dan JSON. Seluruh implementasi disimpan dalam sistem version control dengan pesan commit "feat: OpenAPI Document Generator" untuk menjaga riwayat pengembangan sistem.

#### 4.2.10 OpenAPI Validation

Implementasi dimulai dengan menginstal library `openapi-schema-validator` yang digunakan untuk memvalidasi dokumen OpenAPI berdasarkan standar OpenAPI 3.1. Library ini dipilih karena masih aktif dikembangkan, mendukung spesifikasi OpenAPI versi terbaru, serta menyediakan pesan error yang jelas sehingga memudahkan dalam mengidentifikasi lokasi dan penyebab kesalahan. Instalasi dilakukan menggunakan npm, termasuk penambahan dukungan tipe data untuk TypeScript. Library ini bekerja dengan membandingkan struktur dokumen OpenAPI yang dihasilkan dengan standar OpenAPI 3.1, sehingga dapat memastikan bahwa semua bagian yang diperlukan tersedia dan memiliki format yang sesuai. Selanjutnya, dibuat class `ValidationService` pada file `src/services/ValidationService.ts` untuk menangani proses validasi dokumen. Class ini dirancang tanpa menyimpan data internal, sehingga dapat digunakan berulang kali tanpa memengaruhi proses validasi lainnya. Pendekatan ini memisahkan proses validasi dari proses pembuatan dokumen, sehingga memudahkan pengujian dan pemeliharaan sistem. Class ini juga tidak memerlukan akses ke layanan eksternal karena validasi dilakukan langsung berdasarkan isi dokumen yang diberikan. Method `validateDocument()` menjadi fungsi utama yang digunakan untuk memvalidasi dokumen OpenAPI dalam bentuk object JavaScript. Fungsi ini memeriksa struktur dokumen secara menyeluruh, termasuk memastikan keberadaan bagian penting seperti `openapi`, `info`, dan `paths`, serta memastikan bahwa format dan struktur data telah sesuai dengan spesifikasi OpenAPI 3.1. Proses ini berjalan dengan cepat dan tetap efisien meskipun dokumen memiliki banyak endpoint. Selain itu, disediakan juga method `validateFile()` yang digunakan untuk memvalidasi dokumen OpenAPI secara langsung dari file. Method ini membaca file berdasarkan lokasi yang diberikan, kemudian memproses isi file sesuai dengan formatnya, baik YAML maupun JSON. Proses pembacaan dan pemrosesan file

dilengkapi dengan penanganan error untuk mengantisipasi kondisi seperti file tidak ditemukan, tidak memiliki izin akses, atau format file tidak valid. Pemisahan fungsi ini memberikan fleksibilitas untuk melakukan validasi baik dari object dalam program maupun langsung dari file.

Hasil validasi dikembalikan dalam bentuk struktur data yang berisi status validasi secara keseluruhan, daftar error, dan daftar peringatan. Status validasi menunjukkan apakah dokumen sudah sesuai dengan spesifikasi atau masih memiliki kesalahan. Daftar error berisi kesalahan yang harus diperbaiki karena dapat menyebabkan dokumen tidak valid, sedangkan daftar peringatan berisi informasi tambahan yang sebaiknya diperbaiki untuk meningkatkan kualitas dokumentasi. Informasi error juga mencakup lokasi kesalahan dalam dokumen dan penjelasan mengenai penyebab kesalahan tersebut.

Tabel 4.2.13 Struktur Data ValidationResult

| Property          | Type      | Deskripsi                                                                       | Contoh Value                                           |
|-------------------|-----------|---------------------------------------------------------------------------------|--------------------------------------------------------|
| <b>isValid</b>    | boolean   | Menunjukkan apakah dokumen valid secara keseluruhan                             | true atau false                                        |
| <b>errors</b>     | Error[]   | Daftar kesalahan yang harus diperbaiki agar dokumen valid                       | [{path: "/paths/users", message: "Missing responses"}] |
| <b>warnings</b>   | Warning[] | Daftar peringatan yang sebaiknya diperbaiki untuk meningkatkan kualitas dokumen | [{path: "/info", message: "Missing description"}]      |
| <b>errorCount</b> | number    | Jumlah total kesalahan yang ditemukan dalam dokumen                             | 3                                                      |

|                     |        |                                                      |   |
|---------------------|--------|------------------------------------------------------|---|
| <b>warningCount</b> | number | Jumlah total peringatan yang ditemukan dalam dokumen | 5 |
|---------------------|--------|------------------------------------------------------|---|

Untuk memudahkan pengguna dalam membaca hasil validasi, dibuat method *generateReport()* yang mengubah hasil validasi menjadi laporan dalam bentuk teks yang mudah dipahami. Laporan ini dapat ditampilkan pada panel output di VS Code atau disimpan ke dalam file. Laporan dimulai dengan ringkasan status validasi, kemudian diikuti dengan daftar error dan peringatan secara rinci. Format laporan dirancang agar memudahkan pengembang dalam menemukan dan memperbaiki kesalahan yang ada.

```
rest-api-doc-generator > └ validation-report.txt
1 =====
2 OPENAPI VALIDATION REPORT
3 =====
4
5 ✓ Status: VALID
6
7 ✨ No errors or warnings found!
8 Your OpenAPI document is perfectly valid.
9 =====
10 |
```

Gambar 4.2.20 Contoh Laporan Validasi Hasil Dokumentasi 1

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.
=====
OPENAPI VALIDATION REPORT
=====
✖ Status: INVALID
ERRORS (1):

1. Failed to validate file: YAMLEException: duplicated mapping key (262:7)
| 259 | type: string
| 260 | price:
| 261 | tyducts
| 262 | parameters:
-----^
| 263 | - name: id
| 264 | in: path
=====
```

Gambar 4.2.21 Contoh Laporan Validasi Hasil Dokumentasi 2

Penanganan error juga diimplementasikan untuk berbagai kondisi kesalahan. Apabila struktur dokumen tidak dapat diproses, sistem akan menampilkan pesan error yang mudah dipahami. Apabila hanya sebagian dokumen yang memiliki kesalahan, sistem tetap menampilkan hasil validasi untuk bagian lain yang sudah benar, serta menunjukkan bagian mana yang perlu diperbaiki. Pendekatan ini membantu pengguna memahami kondisi dokumen secara keseluruhan. Pesan error yang dihasilkan juga dilengkapi dengan penjelasan tambahan agar lebih mudah dipahami. Misalnya, apabila ditemukan bahwa bagian responses tidak tersedia, sistem akan memberikan penjelasan bahwa setiap endpoint harus memiliki setidaknya satu response. Penjelasan ini membantu pengguna dalam memperbaiki dokumen dengan lebih cepat dan tepat. Pengujian integrasi dilakukan dengan menggunakan berbagai contoh dokumen, baik dokumen yang valid maupun dokumen yang mengandung kesalahan. Pengujian ini bertujuan untuk memastikan bahwa sistem dapat mendeteksi kesalahan dengan benar dan tidak memberikan error pada dokumen yang sudah valid. Selain itu, pengujian performa juga

dilakukan untuk memastikan bahwa proses validasi dapat berjalan dengan cepat dan memberikan hasil dalam waktu singkat. Laporan hasil validasi juga dapat disimpan ke dalam file teks untuk keperluan dokumentasi atau dibagikan kepada anggota tim lainnya. Proses penyimpanan file dilengkapi dengan penanganan error untuk mengantisipasi masalah seperti keterbatasan izin akses atau ruang penyimpanan.

Tabel 4.2.14 Skenario Pengujian Validasi

| <b>Test Case</b>        | <b>Input Document</b>                      | <b>Expected Result</b>                        | <b>Status</b> |
|-------------------------|--------------------------------------------|-----------------------------------------------|---------------|
| Valid complete document | Semua field yang diperlukan tersedia       | isValid: true,<br>errors: []                  | Pass          |
| Missing openapi version | Tidak terdapat field openapi               | isValid: false,<br>error: "Missing openapi"   | Pass          |
| Missing info section    | Tidak terdapat field info                  | isValid: false,<br>error: "Missing info"      | Pass          |
| Missing responses       | Terdapat path tanpa bagian responses       | isValid: false,<br>error: "Missing responses" | Pass          |
| Invalid schema type     | Schema memiliki tipe data yang tidak valid | isValid: false,<br>error: "Invalid type"      | Pass          |
| Malformed YAML file     | Struktur YAML tidak valid (syntax error)   | isValid: false,<br>error: "Parsing error"     | Pass          |

Dengan selesainya implementasi validation service, sistem telah memiliki mekanisme untuk memastikan bahwa dokumen OpenAPI yang dihasilkan telah sesuai dengan standar yang berlaku. Integrasi antara document generator dan validation service memungkinkan sistem untuk mendeteksi kesalahan sebelum dokumentasi digunakan atau dipublikasikan. Seluruh implementasi disimpan dalam sistem version control dengan pesan commit "feat: OpenAPI Validation" untuk menjaga riwayat pengembangan sistem. Tahap selanjutnya akan difokuskan pada pengembangan panel pengaturan yang memungkinkan pengguna mengelola konfigurasi sistem dan API key melalui antarmuka yang mudah digunakan.

#### 4.2.11 Settings Panel & Configuration

Implementasi dimulai dengan menambahkan command baru pada file package.json, tepatnya pada bagian contributions.commands. Command ini diberikan identifier rest-api-doc-generator.openSettings dan judul "REST API Docs: Open Settings". Command tersebut dapat diakses melalui command palette di VS Code dengan menekan Ctrl+Shift+P, sehingga pengguna dapat membuka panel pengaturan dengan cepat. Penambahan command ini juga dapat dilengkapi dengan icon agar lebih mudah dikenali oleh pengguna. Selanjutnya, dibuat class SettingsPanelProvider pada file src/services/SettingsPanelProvider.ts untuk mengelola pembuatan dan tampilan panel pengaturan. Class ini bertanggung jawab untuk membuat, menampilkan, dan mengelola panel pengaturan. Class ini menerima extension context dan SecureStorageService sebagai parameter, sehingga panel dapat menyimpan dan mengambil API key secara aman tanpa harus berinteraksi langsung dengan sistem penyimpanan.

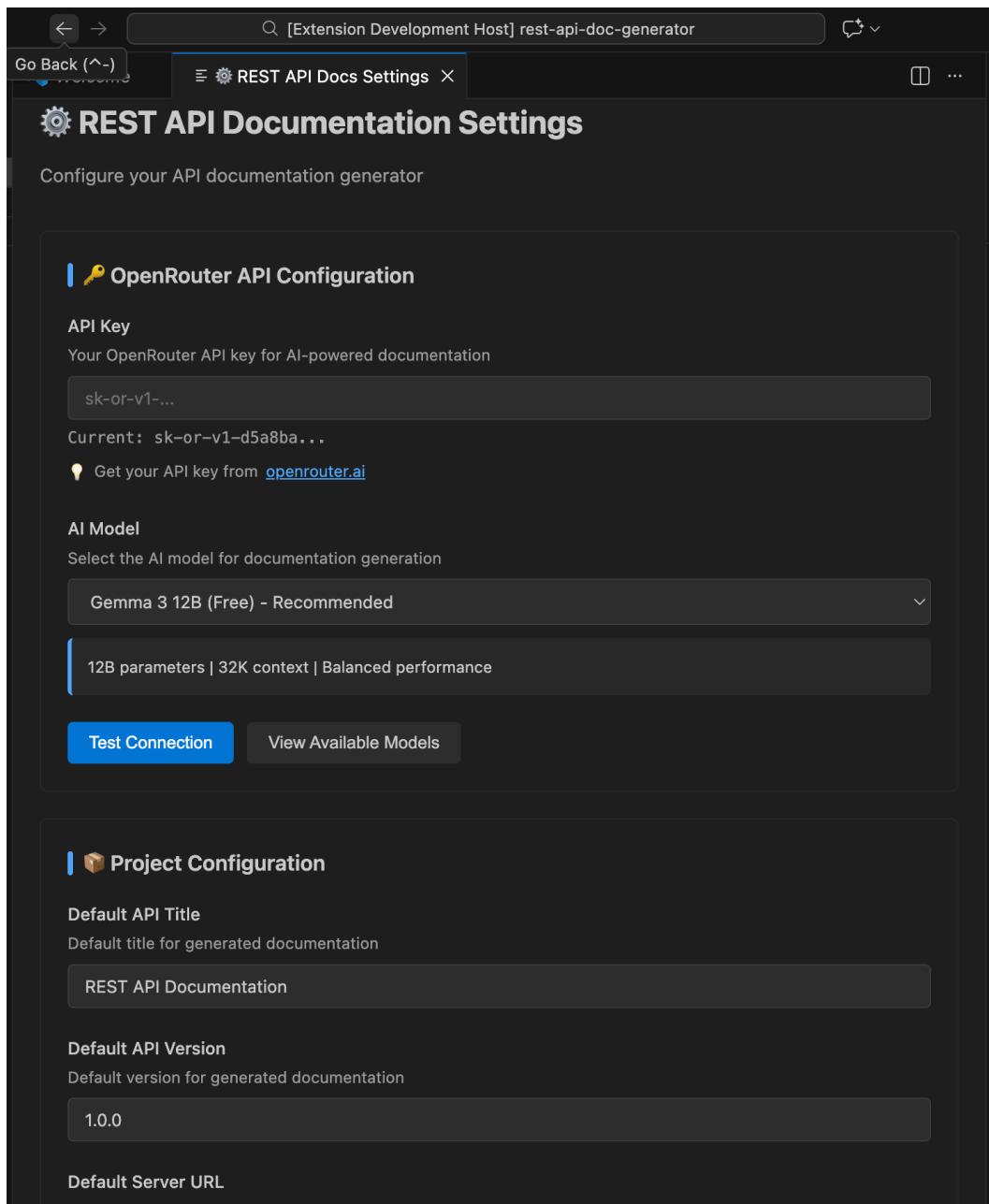
Panel pengaturan dibuat menggunakan Webview API dari VS Code, yang memungkinkan penggunaan HTML, CSS, dan JavaScript untuk membuat tampilan antarmuka di dalam VS Code. Method show() digunakan untuk menampilkan panel pengaturan. Apabila panel sudah terbuka sebelumnya, panel yang sama akan digunakan kembali untuk menghindari pembukaan panel yang berulang. Panel ini diberi judul "*API Documentation Settings*", icon yang sesuai, serta posisi tampilan

yang menyesuaikan dengan ruang kerja yang tersedia. Konten tampilan panel disimpan dalam file terpisah, yaitu src/webview/settings.html, untuk memisahkan logika program dan tampilan. Halaman ini berisi form yang memungkinkan pengguna memasukkan API key, memilih model AI yang akan digunakan, serta mengatur beberapa preferensi tambahan seperti validasi otomatis dan format output dokumentasi. Tampilan form menggunakan variabel tema bawaan VS Code agar tetap sesuai dengan tema yang digunakan, baik terang maupun gelap.

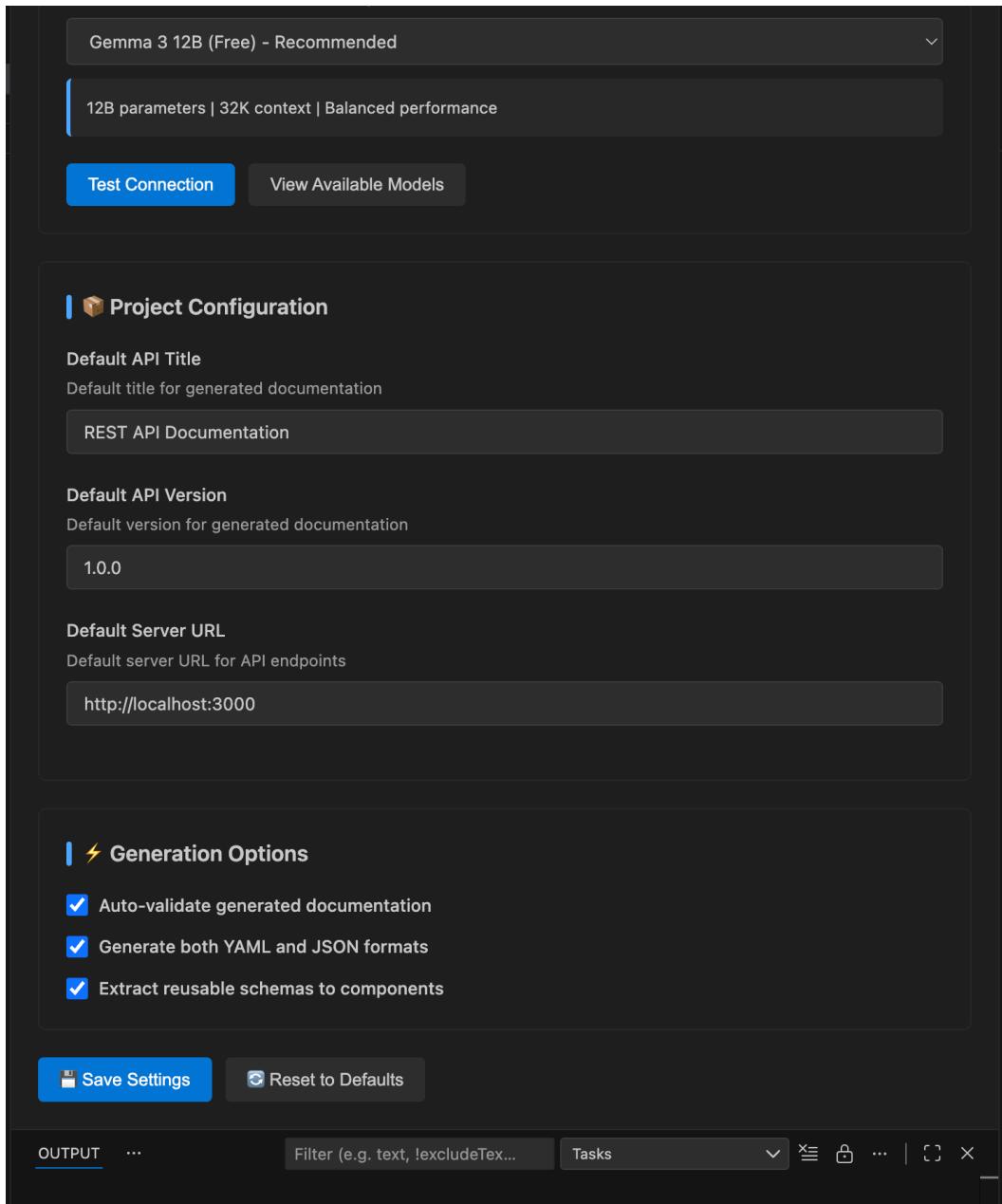
Field untuk memasukkan API key menggunakan tipe password sehingga teks yang dimasukkan tidak terlihat secara langsung, untuk menjaga keamanan. Validasi juga dilakukan untuk memastikan format API key sesuai dengan format OpenRouter, yaitu diawali dengan prefix tertentu. Apabila format tidak sesuai, sistem akan menampilkan pesan kesalahan secara langsung pada panel, sehingga pengguna dapat memperbaikinya. Panel pengaturan juga menyediakan pilihan model AI yang didukung oleh OpenRouter, seperti google/gemma-3-12b-it:free, google/gemma-3-27b-it:free, dan google/gemma-3-4b-it:free. Model default yang digunakan adalah google/gemma-3-12b-it:free karena memberikan keseimbangan antara kualitas hasil dan kecepatan proses berdasarkan hasil pengujian. Komunikasi antara panel pengaturan dan sistem utama dilakukan melalui mekanisme pertukaran pesan. Panel dapat mengirim data ke sistem, misalnya saat pengguna menyimpan API key atau mengubah pengaturan. Sistem kemudian memproses data tersebut, menyimpannya secara aman, dan mengirimkan kembali informasi status kepada panel, seperti notifikasi berhasil atau gagal.

Penyimpanan API key dilakukan menggunakan SecureStorageService, sehingga data tersimpan secara aman dalam sistem. Apabila terjadi kesalahan saat penyimpanan, sistem akan menampilkan pesan kesalahan yang mudah dipahami oleh pengguna. Selain itu, disediakan juga fitur untuk menguji koneksi API key yang dimasukkan. Fitur ini memungkinkan pengguna memastikan bahwa API key yang digunakan valid sebelum digunakan untuk menghasilkan dokumentasi. Hasil pengujian akan ditampilkan pada panel dalam bentuk pesan berhasil atau pesan

kesalahan beserta informasi tambahan yang membantu pengguna memperbaiki masalah. Pengaturan yang dipilih oleh pengguna, seperti model AI dan preferensi lainnya, disimpan menggunakan sistem konfigurasi bawaan VS Code. Hal ini memungkinkan pengaturan tetap tersimpan dan dapat digunakan kembali tanpa perlu mengatur ulang setiap kali VS Code dibuka. Sistem juga menampilkan notifikasi untuk memberikan informasi kepada pengguna. Notifikasi berhasil akan ditampilkan apabila pengaturan berhasil disimpan atau koneksi berhasil dilakukan. Sebaliknya, notifikasi kesalahan akan ditampilkan apabila terjadi masalah, disertai dengan penjelasan yang membantu pengguna memahami penyebabnya. Panel pengaturan juga menyediakan beberapa fitur tambahan untuk meningkatkan kemudahan penggunaan, seperti tombol untuk menampilkan atau menyembunyikan API key, tombol untuk menghapus API key yang tersimpan, serta tombol untuk mengembalikan semua pengaturan ke kondisi awal. Fitur-fitur ini membantu pengguna dalam mengelola pengaturan dengan lebih mudah.



Gambar 4.2.22 Webview API Documentation Settings 1



Gambar 4.2.23 Webview API Documentation Settings 2

Dengan selesainya implementasi panel pengaturan, plugin telah menyediakan antarmuka yang mudah digunakan untuk mengelola konfigurasi sistem. Integrasi antara tampilan panel, sistem penyimpanan aman, dan sistem konfigurasi memungkinkan pengguna mengatur dan mengelola pengaturan dengan lebih efisien. Seluruh implementasi disimpan dalam sistem version control dengan

pesan commit "feat: Settings Panel & Configuration" untuk menjaga riwayat pengembangan sistem.

#### **4.2.12 Implement Auto-Sync Documentation**

Implementasi fitur ini dimulai dengan pembuatan class *FileWatcherService* pada file *src/services/FileWatcherService.ts* yang bertugas memantau perubahan file dalam workspace proyek. Class ini menggunakan File System Watcher API dari VS Code yang memungkinkan sistem mendeteksi perubahan file secara otomatis. Class ini menerima path workspace dan fungsi yang akan dijalankan ketika terjadi perubahan file, sehingga proses pemantauan file dan proses pembaruan dokumentasi dapat dipisahkan dengan jelas. Pemantauan file difokuskan pada direktori routes dengan menggunakan pola pencarian yang fleksibel. Sistem dikonfigurasi untuk memantau file JavaScript dan TypeScript, termasuk file dalam subdirektori. Pendekatan ini memastikan bahwa semua file yang berisi definisi endpoint dapat terdeteksi apabila terjadi perubahan. *Method start()* digunakan untuk mengaktifkan proses pemantauan file. Sistem akan mendeteksi tiga jenis perubahan, yaitu ketika file diubah, file baru ditambahkan, dan file dihapus. Ketiga kondisi ini mencakup semua kemungkinan perubahan yang dapat memengaruhi dokumentasi API. Ketika perubahan terdeteksi, sistem akan menjalankan fungsi yang telah ditentukan untuk memperbarui dokumentasi. Untuk meningkatkan efisiensi, diterapkan mekanisme penundaan singkat sebelum pembaruan dokumentasi dilakukan. Sistem akan menunggu selama 2 detik setelah perubahan terakhir terdeteksi sebelum memulai proses pembaruan. Apabila terjadi perubahan tambahan dalam waktu tersebut, waktu tunggu akan diulang kembali. Pendekatan ini mencegah sistem melakukan pembaruan berulang kali secara berlebihan, terutama ketika pengguna sedang melakukan banyak perubahan dalam waktu singkat. Setelah waktu tunggu selesai, sistem akan memperbarui dokumentasi berdasarkan file yang mengalami perubahan. Proses ini hanya memproses file yang berubah, tanpa perlu memproses seluruh proyek. Pendekatan ini membuat proses pembaruan dokumentasi menjadi lebih cepat dan efisien dibandingkan dengan memproses seluruh file setiap kali terjadi perubahan.

Fitur auto-sync dapat diaktifkan atau dinonaktifkan melalui pengaturan workspace. Sistem akan memeriksa pengaturan ini untuk menentukan apakah pemantauan file perlu dijalankan. Pengguna dapat menonaktifkan fitur ini sementara waktu apabila diperlukan, misalnya saat melakukan perubahan besar pada kode. Selama proses pembaruan dokumentasi berlangsung, sistem akan menampilkan notifikasi yang menunjukkan status proses kepada pengguna. Notifikasi ini memberikan informasi mengenai file yang sedang diproses dan status pembaruan dokumentasi. Setelah proses selesai, sistem akan menampilkan notifikasi bahwa dokumentasi telah berhasil diperbarui. Apabila beberapa file berubah dalam waktu yang berdekatan, sistem akan memproses semua perubahan tersebut secara bersamaan. Pendekatan ini mengurangi jumlah proses yang diperlukan dan meningkatkan efisiensi kinerja sistem. Sistem juga dilengkapi dengan mekanisme penanganan error untuk memastikan bahwa dokumentasi yang sudah valid tidak hilang apabila terjadi kesalahan selama proses pembaruan. Apabila terjadi kesalahan sementara, sistem akan mencoba kembali secara otomatis. Apabila kesalahan berlanjut, sistem akan menampilkan pesan error yang membantu pengguna memahami penyebab masalah. Selain itu, panel preview dokumentasi akan diperbarui secara otomatis setelah dokumentasi berhasil diperbarui. Hal ini memungkinkan pengguna melihat perubahan dokumentasi secara langsung tanpa perlu memuat ulang secara manual. Pengguna juga dapat mengaktifkan atau menonaktifkan fitur auto-sync melalui command yang tersedia di command palette VS Code. Fitur ini memberikan fleksibilitas kepada pengguna untuk mengontrol proses pembaruan dokumentasi sesuai kebutuhan.

Untuk menjaga kinerja sistem tetap optimal, disediakan juga mekanisme untuk menghentikan proses pemantauan file ketika tidak diperlukan, misalnya saat ekstensi dinonaktifkan atau VS Code ditutup. Hal ini mencegah penggunaan sumber daya secara berlebihan. Dengan selesainya implementasi fitur auto-sync documentation, sistem mampu memperbarui dokumentasi secara otomatis setiap kali terjadi perubahan pada kode. Integrasi antara pemantauan file dan proses

pembuatan dokumentasi memungkinkan sistem menjaga konsistensi antara kode dan dokumentasi secara berkelanjutan. Seluruh implementasi disimpan dalam sistem version control dengan pesan commit "feat: Implement Auto-Sync Documentation with AI-Powered Incremental Updates" untuk menjaga riwayat pengembangan sistem.

#### 4.2.13 Webview Documentation Preview

Implementasi dimulai dengan pembuatan kelas *PreviewPanelProvider* pada file *src/services/PreviewPanelProvider.ts* yang berfungsi mengatur pembuatan dan pengelolaan panel pratinjau. Kelas ini dirancang menggunakan konsep satu instance, sehingga panel yang sama dapat digunakan kembali ketika pengguna membuka pratinjau beberapa kali. Pendekatan ini membantu menghemat penggunaan sumber daya sistem. Konstruktor kelas menerima parameter berupa konteks ekstensi dan lokasi folder kerja, yang digunakan untuk mengakses file dan sumber daya yang diperlukan. *Method show()* bertugas untuk menampilkan panel pratinjau. Sistem terlebih dahulu memeriksa apakah panel sudah tersedia dan masih aktif. Jika panel sudah ada, maka panel tersebut akan langsung ditampilkan kembali tanpa membuat panel baru. Namun, jika panel belum ada atau sudah ditutup, maka sistem akan membuat panel baru menggunakan Webview API dari VS Code. Panel ini diberi judul "*API Documentation Preview*", ikon yang sesuai, serta ditempatkan di samping editor agar pengguna dapat melihat kode dan dokumentasi secara bersamaan.

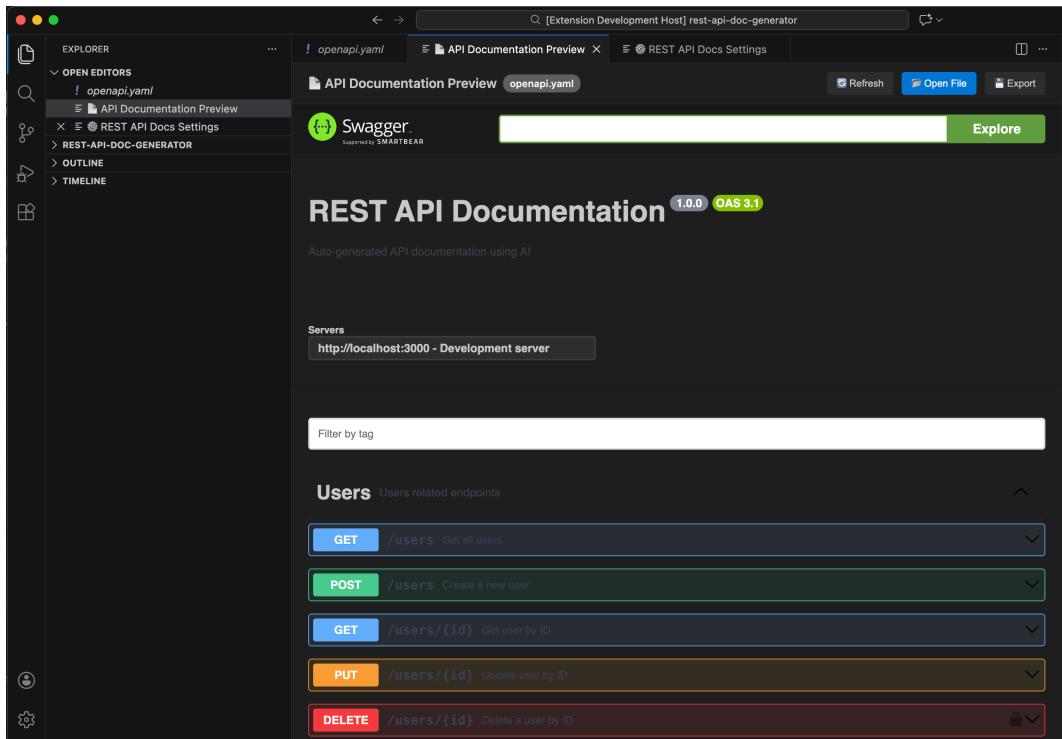
Pengaturan webview mencakup pengaktifan dukungan JavaScript agar fitur interaktif dapat berjalan, seperti membuka dan menutup bagian tertentu serta menampilkan format kode dengan baik. Untuk menjaga keamanan, akses file dibatasi hanya pada folder tertentu yang diizinkan. Selain itu, sistem juga menerapkan kebijakan keamanan untuk mencegah pemuatan skrip atau sumber daya yang tidak sah. Konten tampilan pratinjau disimpan dalam file *src/webview/preview.html*, yang berisi struktur dan tampilan dokumentasi agar mudah dibaca. Tampilan ini menggunakan Swagger UI, yaitu alat yang umum

digunakan untuk menampilkan dokumentasi OpenAPI secara visual dan interaktif. Swagger UI dipilih karena sudah stabil, memiliki fitur lengkap, dan banyak digunakan oleh pengembang. Integrasi dilakukan dengan memuat Swagger UI ke dalam webview, baik melalui internet maupun dari file lokal agar tetap dapat digunakan tanpa koneksi internet. Proses pemuatan dokumentasi dilakukan melalui method *loadDocumentation()*, yang membaca file OpenAPI dari folder kerja. Sistem akan mengenali jenis file berdasarkan ekstensi, yaitu YAML atau JSON, kemudian memproses isinya sesuai dengan format tersebut. Jika file menggunakan format YAML, sistem menggunakan pustaka khusus untuk membacanya, sedangkan file JSON diproses menggunakan fungsi bawaan JavaScript. Sistem juga dilengkapi dengan penanganan kesalahan untuk mengatasi masalah seperti file tidak ditemukan atau isi file tidak valid. Komunikasi antara ekstensi dan tampilan webview dilakukan melalui mekanisme pengiriman pesan. Ekstensi mengirim data dokumentasi ke webview untuk ditampilkan, dan webview akan memperbarui tampilan sesuai dengan data yang diterima. Selain itu, webview juga dapat mengirim pesan kembali ke ekstensi, misalnya untuk membuka file sumber atau menjalankan fungsi tertentu. Sistem juga menyediakan penanganan kesalahan yang jelas agar pengguna dapat memahami masalah yang terjadi. Jika file dokumentasi belum tersedia, sistem akan menampilkan pesan yang menjelaskan bahwa dokumentasi perlu dibuat terlebih dahulu. Jika terjadi kesalahan pada format file, sistem akan menampilkan informasi yang membantu pengguna menemukan dan memperbaiki kesalahan tersebut.

Fitur pembaruan otomatis juga disediakan melalui method *update()*, yang memungkinkan panel pratinjau menampilkan versi terbaru dari dokumentasi. Ketika dokumentasi diperbarui, panel pratinjau akan memuat ulang file dan menampilkan perubahan secara langsung. Dengan demikian, pengguna dapat melihat hasil perubahan tanpa perlu menutup dan membuka ulang panel. Swagger UI juga menyediakan berbagai fitur interaktif yang memudahkan pengguna dalam memahami dokumentasi. Pengguna dapat membuka atau menutup bagian tertentu, melihat detail parameter, memahami struktur respons, serta melihat contoh data.

Fitur ini membantu pengguna memahami cara kerja API tanpa harus mencoba langsung melalui aplikasi lain. Selain itu, plugin juga menambahkan opsi menu yang memungkinkan pengguna membuka pratinjau langsung dari daftar file. Pengguna dapat mengklik kanan pada file dokumentasi OpenAPI, kemudian memilih opsi pratinjau untuk langsung menampilkan dokumentasi dalam panel. Tampilan pratinjau juga menyesuaikan dengan tema yang digunakan di VS Code. Jika pengguna menggunakan tema gelap, maka dokumentasi akan ditampilkan dalam mode gelap. Sebaliknya, jika menggunakan tema terang, tampilan akan menyesuaikan secara otomatis. Hal ini membuat tampilan tetap konsisten dan nyaman digunakan. Tombol penyegaran juga disediakan agar pengguna dapat memperbarui tampilan secara manual jika diperlukan. Tombol ini akan memuat ulang file dokumentasi dan menampilkan versi terbaru. Fitur ini berguna terutama ketika pengguna ingin memastikan bahwa dokumentasi yang ditampilkan sudah sesuai dengan versi terbaru.

Selain melihat dokumentasi, pengguna juga dapat melakukan ekspor dokumentasi ke berbagai format langsung dari panel pratinjau. Opsi ini memudahkan pengguna untuk menyimpan atau membagikan dokumentasi sesuai kebutuhan. Dengan selesainya implementasi fitur pratinjau dokumentasi ini, plugin menyediakan cara yang lengkap dan mudah untuk melihat dokumentasi API secara visual dan interaktif. Integrasi langsung dengan VS Code, kemampuan pembaruan otomatis, serta fitur interaktif membantu meningkatkan kemudahan penggunaan dan efisiensi kerja. Seluruh implementasi disimpan dengan pesan commit "feat: Webview Documentation Preview" untuk mencatat riwayat pengembangan.



Gambar 4.2.24 Webview Documentation Preview

#### 4.2.14 Export & Import Documentation

Implementasi fitur ini dilakukan dengan membuat kelas *ExportImportService* pada file *src/services/ExportImportService.ts*. Kelas ini menangani seluruh proses ekspor dan impor dokumentasi. Kelas dirancang tanpa menyimpan data tetap di dalamnya, sehingga setiap proses ekspor atau impor dapat berjalan secara mandiri tanpa memengaruhi proses lainnya. Pendekatan ini membuat sistem lebih stabil dan hasilnya lebih konsisten. Kelas ini menggunakan lokasi folder kerja sebagai acuan utama untuk membaca dan menyimpan file dokumentasi. Fitur ekspor menyediakan tiga format utama, yaitu YAML, JSON, dan Markdown, yang masing-masing memiliki kegunaan berbeda. Format YAML digunakan sebagai format utama karena mudah dibaca oleh manusia dan umum digunakan untuk dokumentasi OpenAPI. Format JSON digunakan sebagai alternatif yang lebih cocok untuk diproses oleh program atau alat lain yang membutuhkan format JSON. Format Markdown digunakan untuk menghasilkan

dokumentasi dalam bentuk teks terstruktur yang dapat ditampilkan di situs dokumentasi, wiki, atau platform lain tanpa memerlukan alat khusus.

Method `exportAsYAML()` digunakan untuk mengekspor dokumentasi ke format YAML. Sistem membaca file dokumentasi OpenAPI dari folder kerja, kemudian menampilkan dialog penyimpanan agar pengguna dapat memilih lokasi dan nama file. Pengguna dapat menyimpan file di dalam folder kerja atau di lokasi lain sesuai kebutuhan. Proses penulisan file dilakukan secara asynchronous agar tidak mengganggu kinerja VS Code selama proses berlangsung. Sistem juga dilengkapi dengan penanganan kesalahan, misalnya jika lokasi penyimpanan tidak dapat diakses atau ruang penyimpanan tidak mencukupi. Setelah proses selesai, sistem akan menampilkan pemberitahuan yang berisi lokasi file yang telah disimpan. Method `exportAsJSON()` bekerja dengan cara yang serupa, namun hasil dokumentasi dikonversi ke format JSON. File JSON ditulis dengan format yang rapi agar tetap mudah dibaca. Sistem juga memastikan bahwa file JSON yang dihasilkan valid dan dapat digunakan kembali jika diperlukan di kemudian hari. Eksport ke format Markdown dilakukan melalui method `exportAsMarkdown()`, yang mengubah struktur dokumentasi OpenAPI menjadi dokumen Markdown yang terstruktur. Sistem membaca setiap endpoint, kemudian membuat bagian dokumentasi yang berisi informasi seperti alamat endpoint, parameter, dan respons. Hasilnya adalah dokumen yang tersusun rapi dan mudah dibaca, serta dapat langsung digunakan sebagai dokumentasi tertulis.

Fitur impor diimplementasikan melalui method `importDocumentation()`, yang memungkinkan pengguna memasukkan file dokumentasi OpenAPI ke dalam folder kerja. Sistem menampilkan dialog pemilihan file, dan pengguna dapat memilih file dengan format YAML atau JSON. Setelah file dipilih, sistem akan membaca dan memeriksa isi file untuk memastikan bahwa strukturnya sesuai dengan standar OpenAPI sebelum digunakan. Sistem secara otomatis mengenali jenis file berdasarkan ekstensi. File YAML diproses menggunakan pustaka khusus, sedangkan file JSON diproses menggunakan fungsi bawaan JavaScript. Jika terjadi

kesalahan dalam format file, sistem akan menampilkan pesan yang menjelaskan masalah tersebut agar pengguna dapat memperbaikinya. Jika dokumentasi yang diimpor memiliki endpoint yang sama dengan dokumentasi yang sudah ada, sistem akan memberikan pilihan kepada pengguna. Pengguna dapat memilih untuk mengganti data lama dengan data baru, menggabungkan data yang tidak bertentangan, atau membatalkan proses impor. Pilihan ini ditampilkan dengan penjelasan yang jelas agar pengguna dapat memilih tindakan yang sesuai. Sebelum dokumentasi baru disimpan, sistem akan melakukan pemeriksaan untuk memastikan bahwa dokumentasi tersebut sesuai dengan standar yang berlaku. Jika ditemukan masalah, sistem akan menampilkan laporan kepada pengguna. Pengguna dapat memilih untuk melanjutkan proses atau membatalkannya untuk memperbaiki file terlebih dahulu.

Untuk mencegah kehilangan data, sistem juga membuat salinan cadangan sebelum mengganti dokumentasi yang sudah ada. File cadangan disimpan dengan nama yang mencantumkan waktu pembuatan, sehingga pengguna dapat mengembalikan dokumentasi sebelumnya jika diperlukan. Lokasi file cadangan juga diinformasikan kepada pengguna. Fitur ekspor dan impor dapat diakses melalui Command Palette di VS Code dengan nama perintah yang jelas, seperti "*REST API Docs: Export Documentation*" dan "*REST API Docs: Import Documentation*". Pengguna juga dapat mengakses fitur ini melalui menu atau tombol pintasan untuk mempercepat penggunaan. Sistem juga menampilkan indikator proses saat ekspor atau impor berlangsung, terutama untuk dokumentasi yang besar. Informasi ini membantu pengguna mengetahui bahwa proses sedang berjalan dan belum selesai. Pengujian telah dilakukan untuk memastikan bahwa fitur ekspor dan impor bekerja dengan baik dalam berbagai kondisi. Pengujian mencakup proses ekspor dan impor ulang, dokumentasi kosong, dokumentasi dengan struktur kompleks, serta dokumentasi yang memiliki karakter khusus. Hasil pengujian menunjukkan bahwa data tetap terjaga dengan baik setelah proses ekspor dan impor. Dengan selesainya implementasi fitur ekspor dan impor ini, plugin telah menyediakan pengelolaan dokumentasi secara lengkap, mulai dari pembuatan,

penyimpanan, pembagian, hingga penggunaan kembali dokumentasi. Dukungan berbagai format, kemampuan menggabungkan dokumentasi, serta sistem pemeriksaan dan cadangan membuat fitur ini dapat digunakan secara aman dan fleksibel. Seluruh implementasi disimpan dengan pesan commit "feat: *Export & Import Documentation*" sebagai bagian dari riwayat pengembangan. Plugin kini telah memenuhi seluruh kebutuhan fungsional yang direncanakan dan siap untuk memasuki tahap pengujian sistem secara menyeluruh.

### **4.3 Pengujian Sistem**

#### Test Coverage

| Modul              | Unit Tests | Integration |
|--------------------|------------|-------------|
| FileScanner        | 5 Tes      | (via e2e)   |
| RouteParser        | 11 Tes     | (via e2e)   |
| OpenRouterClient   | 6 Tes      | (stubbed)   |
| OpenAPIGenerator   | 9 Tes      | (via e2e)   |
| ValidationService  | 5 Tes      | (via e2e)   |
| FileWatcherService | 4 Tes      | -           |

#### Manual Tests

TEST3: Usability & Compatibility (11 scenarios)

TEST4: Security & Performance Audit (7 checks)

TEST5: Functional Requirements Validation (F1–F9 matrix)

#### **4.3.1 Unit Testing - Core Modules**

#### **4.3.2 Pengujian Validasi Terhadap Format OpenAPI.**

#### **4.3.3 Pengujian Performa**

#### **4.3.4 Pengujian Akurasi Dokumentasi**

### **4.4 Hasil dan Pembahasan**

#### Penggunaan

##### 1. Konfigurasi API Key

- a. Buka Command Palette (Cmd+Shift+P / Ctrl+Shift+P)
- b. Ketik REST API Doc Generator: Open Settings
- c. Masukkan API key OpenRouter Anda
- d. Pilih model AI (default: google/gemma-3-12b-it:free)
- e. Klik Save

f. API key disimpan secara aman menggunakan VS Code SecretStorage (OS keychain). Tidak tersimpan di disk.

2. Generate Dokumentasi

- a. Buka workspace yang berisi project Express.js
- b. Command Palette: REST API Doc Generator: Generate Documentation
- c. Tunggu proses (parsing, AI inference, generate)
- d. File openapi.yaml dan openapi.json akan dibuat di root workspace

3. Preview Dokumentasi

- a. Command Palette: REST API Doc Generator: Preview Documentation
- b. Swagger UI akan terbuka di panel VS Code

4. Auto-Sync

- a. Buka Settings: aktifkan Auto Sync
- b. Edit file route: simpan
- c. Dokumentasi akan diperbarui otomatis setelah 2 detik

5. Export / Import

- a. Export: Command Palette: REST API Doc Generator: Export Documentation
- b. Pilih format: YAML, JSON, atau Markdown
- c. Import: Command Palette: REST API Doc Generator: Import Documentation
- d. Pilih file YAML/JSON: sistem akan memvalidasi secara otomatis

**4.4.1 Hasil Pengujian Setiap Fitur**

**4.4.2 Analisis Efektivitas Plugin**

**4.4.3 Perbandingan dengan Tools Lain (Swagger AutoDoc, DocGen)**

**4.4.4 Evaluasi Kinerja dan Akurasi Model AI**

**4.4.5 Kelebihan dan Kekurangan Sistem**

## BAB V PENUTUP

### 5.1 Kesimpulan

 Kondisi Real-World di Perusahaan

Struktur Project Nyata (Medium-Large Scale)

```
project/
 └── src/
 └── routes/
 └── api/
 ├── v1/
 │ ├── users.js ← Auth, CRUD users
 │ ├── products.js ← Product catalog
 │ ├── orders.js ← Order management
 │ ├── payments.js ← Payment processing
 │ ├── inventory.js ← Stock management
 │ ├── shipping.js ← Logistics
 │ ├── analytics.js ← Reports & metrics
 │ ├── notifications.js ← Email/SMS/Push
 │ └── webhooks.js ← External integrations
 └── v2/
 ├── users.js ← New auth system
 └── products.js ← New product features
 └── admin/
 ├── dashboard.js
 ├── settings.js
 └── audit.js
 └── public/
 ├── auth.js ← Login/register
 └── health.js ← Health check
 └── app.js
```

Jawaban: Ya, sangat normal ada puluhan file routes dalam satu project.

contoh rest api di dunia perusahaan

### 5.2 Saran untuk Pengembangan Selanjutnya



## DAFTAR PUSTAKA

- [1] A. Ehsan, M. A. M. E. Abuhalqa, C. Catal, and D. Mishra, "RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions," *Appl. Sci.*, vol. 12, no. 9, p. 4369, Apr. 2022, doi: 10.3390/app12094369.
- [2] S. U. Meshram, "Evolution of Modern Web Services – REST API with its Architecture and Design," *Int J Res. Eng. Sci. Manag.*, vol. 4, pp. 83–86, 2021.
- [3] P. Gowda and A. N. Gowda, "Best Practices in REST API Design for Enhanced Scalability and Security," *J. Artif. Intell. Mach. Learn. Data Sci.*, vol. 2, no. 1, pp. 827–830, Feb. 2024, doi: 10.51219/JAIMLD/priyanka-gowda/202.
- [4] G. Bondel, A. Cerit, and F. Matthes, "Challenges of API Documentation from a Provider Perspective and Best Practices for Examples in Public Web API Documentation;," in *Proceedings of the 24th International Conference on Enterprise Information Systems*, Online Streaming, --- Select a Country ---: SCITEPRESS - Science and Technology Publications, 2022, pp. 268–279. doi: 10.5220/0011089700003179.
- [5] J. Y. Khan, M. T. I. Khondaker, G. Uddin, and A. Iqbal, "Automatic Detection of Five API Documentation Smells: Practitioners' Perspectives," Feb. 16, 2021, *arXiv*: arXiv:2102.08486. doi: 10.48550/arXiv.2102.08486.
- [6] Oracle, "AI Case Study: Auto-Generation of Swagger Documentation for Oracle API Gateway Cloud Service | 4i." Accessed: Dec. 09, 2025. [Online]. Available: <https://www.4iapps.com/ai-case-study-auto-generation-of-swagger-documentation-for-oracle-api-gateway-cloud-service/>
- [7] P. Dhyani, S. Nautiyal, A. Negi, S. Dhyani, and P. Chaudhary, "Automated API Docs Generator using Generative AI," in *2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, Bhopal, India: IEEE, Feb. 2024, pp. 1–6. doi: 10.1109/SCEECS61402.2024.10482119.
- [8] "OpenAPI Specification - Version 3.1.0 | Swagger." Accessed: Dec. 09, 2025. [Online]. Available: <https://swagger.io/specification/>
- [9] S. Wang, Y. Tian, and D. He, "Improving API Documentation Comprehensibility via Continuous Optimization and Multilingual SDK," Mar. 24, 2023, *arXiv*: arXiv:2303.13828. doi: 10.48550/arXiv.2303.13828.
- [10] S. Wang, Y. Tian, and D. He, "gDoc: Automatic Generation of Structured API Documentation," in *Companion Proceedings of the ACM Web Conference 2023*, Apr. 2023, pp. 53–56. doi: 10.1145/3543873.3587310.
- [11] K. Lazar *et al.*, "Generating OpenAPI Specifications from Online API Documentation with Large Language Models".
- [12] Microsoft, "Web API Design Best Practices - Azure Architecture Center." Accessed: Dec. 09, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>
- [13] Strapi Team, "13 Essential VS Code Extensions for 2025." Accessed: Dec. 09, 2025. [Online]. Available: <https://strapi.io/blog/vs-code-extensions>
- [14] S. Cavalcante, E. Ribeiro, and A. Oran, "The Impact of AI Tools on Software Development: A Case Study with GitHub Copilot and Other AI Assistants;," in *Proceedings of the 27th International Conference on Enterprise Information*

- Systems*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2025, pp. 245–252. doi: 10.5220/0013294700003929.
- [15] ..... K. Schwaber and J. Sutherland, *The Scrum Guide*. Scrum Alliance, 2020.
- [16]G. Team *et al.*, “Gemma 3 Technical Report,” Mar. 25, 2025, *arXiv*: arXiv:2503.19786. doi: 10.48550/arXiv.2503.19786.
- [17]OpenRouter AI, “OpenRouter Quickstart Guide,” OpenRouter Documentation. Accessed: Dec. 09, 2025. [Online]. Available: <https://openrouter.ai/docs/quickstart>
- [18]Y. Dong *et al.*, “A Survey on Code Generation with LLM-based Agents,” Sept. 30, 2025, *arXiv*: arXiv:2508.00083. doi: 10.48550/arXiv.2508.00083.
- [19]S. M. Abtahi and A. Azim, “Augmenting Large Language Models with Static Code Analysis for Automated Code Quality Improvements,” June 12, 2025, *arXiv*: arXiv:2506.10330. doi: 10.48550/arXiv.2506.10330.
- [20]Z. Feng *et al.*, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” Sept. 18, 2020, *arXiv*: arXiv:2002.08155. doi: 10.48550/arXiv.2002.08155.
- [21]Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” Sept. 02, 2021, *arXiv*: arXiv:2109.00859. doi: 10.48550/arXiv.2109.00859.
- [22]W. Sun *et al.*, “Source Code Summarization in the Era of Large Language Models,” Aug. 23, 2025, *arXiv*: arXiv:2407.07959. doi: 10.48550/arXiv.2407.07959.
- [23]A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “IntelliCode Compose: Code Generation Using Transformer,” Oct. 29, 2020, *arXiv*: arXiv:2005.08025. doi: 10.48550/arXiv.2005.08025.
- [24]Q. Zhang *et al.*, “A Survey on Large Language Models for Software Engineering,” Sept. 08, 2024, *arXiv*: arXiv:2312.15223. doi: 10.48550/arXiv.2312.15223.
- [25]Y. Jorelle, “Generation of API Documentation using Large Language Models -- Towards Self-explaining APIs,” *MS Thesis Sch. Sci. Aalto Univ*, 2024.
- [26]“REST API Documentation Tool | Swagger UI.” Accessed: Dec. 09, 2025. [Online]. Available: <https://swagger.io/tools/swagger-ui/>
- [27]R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” *PhD Diss. Univ Calif. Irvine*, 2000.
- [28]“VS Code API.” Accessed: Dec. 09, 2025. [Online]. Available: <https://code.visualstudio.com/api/references/vscode-api>
- [29]H. Alaidaros, M. Omar, and R. Romli, “The state of the art of agile kanban method: challenges and opportunities,” *Indep. J. Manag. Prod.*, vol. 12, no. 8, pp. 2535–2550, Dec. 2021, doi: 10.14807/ijmp.v12i8.1482.
- [30]Y. Wu *et al.*, “LLM Prompt Duel Optimizer: Efficient Label-Free Prompt Optimization,” Oct. 14, 2025, *arXiv*: arXiv:2510.13907. doi: 10.48550/arXiv.2510.13907.