

**TUGAS AKHIR
SKEMA SKRIPSI**

**PAICODE: AGENTIC AI BERBASIS CLI UNTUK
OTOMASI AKTIVITAS PEMROGRAMAN DAN
PENGEMBANGAN PERANGKAT LUNAK DI LINUX
YANG DITENAGAI LLM MELALUI API**



**I PUTU GEDE GILANG TEJA KRISHNA
NIM : 225410001**

**PROGRAM STUDI INFORMATIKA
PROGRAM SARJANA
FAKULTAS TEKNOLOGI INFORMASI
UNIVERSITAS TEKNOLOGI DIGITAL INDONESIA
YOGYAKARTA**

2025

**TUGAS AKHIR
SKEMA SKRIPSI**

**PAICODE: AGENTIC AI BERBASIS CLI UNTUK OTOMASI
AKTIVITAS PEMROGRAMAN DAN PENGEMBANGAN
PERANGKAT LUNAK DI LINUX YANG DITENAGAI LLM
MELALUI API**

Diajukan sebagai salah satu syarat untuk menyelesaikan studi pada

**Program Sarjana
Program Studi Informatika
Fakultas Teknologi Informasi
Universitas Teknologi Digital Indonesia**

Disusun Oleh

**I PUTU GEDE GILANG TEJA KRISHNA
NIM : 225410001**

**PROGRAM STUDI INFORMATIKA
PROGRAM SARJANA
FAKULTAS TEKNOLOGI INFORMASI
UNIVERSITAS TEKNOLOGI DIGITAL INDONESIA
YOGYAKARTA**

2025

HALAMAN PERSETUJUAN UJIAN TUGAS AKHIR

Judul : PAICODE: AGENTIC AI BERBASIS CLI UNTUK
OTOMASI AKTIVITAS PEMROGRAMAN DAN PE-
NGEMBANGAN PERANGKAT LUNAK DI LINUX
YANG DITENAGAI LLM MELALUI API
Nama : I PUTU GEDE GILANG TEJA KRISHNA
NIM : 225410001
Program Studi : Informatika
Program : Sarjana
Semester : Gasal
Tahun Akademik : 2025/2026

Telah diperiksa dan disetujui untuk diujikan
di hadapan Dewan Penguji Tugas Akhir

Yogyakarta, 24 November 2025

Dosen Pembimbing,

Dr. Bambang Purnomosidi Dwi Putranto, S.E., Akt., S.Kom., MMSI

NIDN: 0505058801

HALAMAN PENGESAHAN

PAICODE: AGENTIC AI BERBASIS CLI UNTUK OTOMASI AKTIVITAS PEMROGRAMAN DAN PENGEMBANGAN PERANGKAT LUNAK DI LINUX YANG DITENAGAI LLM MELALUI API

Telah dipertahankan di depan Dewan Penguji dan dinyatakan diterima untuk
memenuhi sebagian persyaratan guna memperoleh

Gelar Sarjana Komputer
Program Studi Informatika
Fakultas Teknologi Informasi
Universitas Teknologi Digital Indonesia

Yogyakarta, 15 Desember 2025

Dewan Penguji	NIDN	Tandatangan
1. Wagito, S.T., M.T. (Ketua)
2. Dr. Bambang Purnomosidi Dwi Putranto, S.E., Akt., S.Kom., MMSI (Sekretaris)
3. Ariesta Damayanti, S.Kom., M.Cs. (Anggota)

Mengetahui
Ketua Program Studi Informatika

Dini Fakta Sari, S.T., M.T.
NIDN:

PERNYATAAN KEASLIAN TUGAS AKHIR

Dengan ini saya menyatakan bahwa naskah Tugas Akhir ini belum pernah diajukan untuk memperoleh gelar Sarjana Komputer di suatu Perguruan Tinggi, dan sepanjang pengetahuan saya tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara sah diacu dalam naskah ini dan disebutkan dalam daftar pustaka.

Yogyakarta, 24 November 2025

I PUTU GEDE GILANG TEJA KRISHNA

NIM: 225410001

HALAMAN PERSEMBAHAN

Tugas Akhir ini saya persembahkan kepada kedua orang tua tercinta yang telah memberikan doa, dukungan, dan kasih sayang yang tiada henti; seluruh keluarga besar yang senantiasa memberikan motivasi dan semangat; para guru dan dosen yang telah membimbing dan memberikan ilmu yang bermanfaat; serta seluruh teman-teman di kampus dan rekan seperjuangan UTDI THE ARCADE.

PRAKATA

Puji syukur ke hadirat Tuhan Yang Maha Esa atas segala rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul **PAICODE: AGENTIC AI BERBASIS CLI UNTUK OTOMASI AKTIVITAS PEMROGRAMAN DAN PENGEMBANGAN PERANGKAT LUNAK DI LINUX YANG DITENAGAI LLM MELALUI API**. Tugas Akhir ini disusun sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer pada Program Studi Informatika, Teknologi Informasi, Universitas Teknologi Digital Indonesia.

Penulis menyadari bahwa penyelesaian Tugas Akhir ini tidak lepas dari bantuan, bimbingan, dan dukungan berbagai pihak. Oleh karena itu, penulis ingin menyampaikan rasa terima kasih dan penghargaan yang setinggi-tingginya kepada semua pihak yang telah membantu. Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas segala rahmat, kesehatan, dan kemudahan yang diberikan selama proses penelitian. Ucapan terima kasih juga penulis sampaikan kepada orang tua dan keluarga yang senantiasa memberikan doa, dukungan moral, dan motivasi yang tiada henti.

Penulis secara khusus menyampaikan terima kasih kepada Bapak Dr. Bambang Purnomosidi Dwi Putranto, S.E., Akt., S.Kom., MMSI selaku dosen pembimbing yang telah memberikan bimbingan, arahan, dan masukan yang sangat berharga selama penyusunan Tugas Akhir ini. Terima kasih juga kepada seluruh dosen dan staf Teknologi Informasi yang telah memberikan ilmu, fasilitas, dan dukungan selama masa perkuliahan, serta rekan-rekan mahasiswa dan teman-teman di kampus yang telah memberikan bantuan, diskusi, dan semangat selama proses penelitian yang tidak dapat penulis sebutkan satu per satu.

Penulis menyadari bahwa Tugas Akhir ini masih jauh dari sempurna. Oleh karena itu, penulis mengharapkan kritik dan saran yang membangun untuk perbaikan di masa mendatang. Semoga Tugas Akhir ini dapat memberikan manfaat bagi pembaca dan perkembangan ilmu pengetahuan.

Yogyakarta, 24 November 2025

Penulis

Daftar Isi

HALAMAN JUDUL	i
HALAMAN PENGESAHAN	iii
PERNYATAAN KEASLIAN TUGAS AKHIR	iv
HALAMAN PERSEMBAHAN	v
PRAKATA	vi
DAFTAR ISI	viii
DAFTAR GAMBAR	xi
DAFTAR TABEL	xii
INTISARI	xiii
ABSTRACT	xiv
1 PENDAHULUAN	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	2
1.3 Ruang Lingkup	2
1.4 Tujuan Penelitian	2
1.5 Manfaat Penelitian	3
1.6 Sistematika Penulisan	3
2 TINJAUAN PUSTAKA DAN DASAR TEORI	5
2.1 Tinjauan Pustaka	5
2.1.1 AI Coding Assistant Terintegrasi (IDE-based)	5
2.1.2 CLI-based AI Chat Tools	5
2.1.3 Autonomous Software Engineers	5
2.1.4 Posisi Paicode	5
2.1.5 Perbandingan dengan Penelitian Sebelumnya	6
2.1.6 Posisi Penelitian	6
2.2 Dasar Teori	7

2.2.1	Command Line Interface (CLI)	7
2.2.2	AI Agent	7
2.2.3	Large Language Model (LLM)	8
2.2.4	Perbedaan LLM dan Agen AI	8
2.2.5	Manajemen Dependensi dengan pip dan Virtual Environment	8
3	METODE PENELITIAN	9
3.1	Bahan Penelitian	9
3.2	Peralatan	9
3.3	Metodologi Penelitian	10
3.3.1	Tahapan Penelitian	10
3.4	Analisis Kebutuhan Sistem	11
3.4.1	Kebutuhan Fungsional	11
3.4.2	Kebutuhan Non-Fungsional	11
3.5	Perancangan Sistem	11
3.5.1	Arsitektur Modular	11
3.5.2	Deskripsi Modul	12
3.5.3	Alur Kerja Sistem	13
4	IMPLEMENTASI DAN PEMBAHASAN	15
4.1	Implementasi dan Uji Coba Sistem	15
4.1.1	Lingkungan Implementasi	15
4.1.2	Implementasi Fitur Utama	15
4.1.3	Skenario Pengujian	19
4.1.4	Hasil Uji Coba	19
4.2	Pembahasan	25
4.2.1	Efisiensi Mekanisme Perencanaan Otomatis	25
4.2.2	Analisis Aspek Keamanan	25
4.2.3	Perbandingan dengan Metode Manual	26
4.2.4	Tinjauan Akhir dan Keterbatasan	27
5	PENUTUP	29
5.1	SIMPULAN	29
5.2	SARAN	30
	DAFTAR PUSTAKA	32
	LAMPIRAN	33

Lampiran A: Manual Penggunaan Aplikasi	33
Lampiran B: Instrumen Pengujian	35

Daftar Gambar

3.1	Flowchart alur eksekusi perintah dalam arsitektur <i>Single-Shot Intelligence</i>	13
4.1	Perintah konfigurasi API Key	16
4.2	Cuplikan agent.py (Struktur Planning JSON)	16
4.3	System Prompt untuk Fase Perencanaan	17
4.4	Logika Strategi Eksekusi Adaptif	18
4.5	Validasi Path (<code>_is_safe_path</code>)	20
4.6	Logika Threshold Modifikasi (Diff-based)	21
4.7	Log: Pembuatan Proyek Kalkulator	22
4.8	Log: Modifikasi tambah fitur pangkat	22
4.9	Log: Eksplorasi Struktur Direktori	23
4.10	Log: Perbaiki Syntax Error	23
4.11	Log: Blokir Akses Ilegal	24

Daftar Tabel

2.1	Perbandingan Penelitian Terdahulu dengan Penelitian yang Dilakukan	6
3.1	Daftar Modul dan Tanggung Jawab Utama	12
4.1	Konfigurasi Lingkungan Implementasi	15
4.2	Skenario Pengujian Fungsional	19
4.3	Hasil Pengukuran Waktu Eksekusi (3 Percobaan)	24
4.4	Ringkasan Indikator Keberhasilan Selepas 3 Percobaan	25
4.5	Perbandingan Jumlah Langkah Kerja (Skenario 1)	26
4.6	Perbandingan Rata-rata Waktu Penyelesaian Tugas	27

INTISARI

Penelitian ini mengusulkan **Paicode**, sebuah agen AI berbasis Command Line Interface (CLI) untuk membantu proses pengembangan perangkat lunak secara interaktif dengan arsitektur *Single-Shot Intelligence*. Sistem berjalan pada lingkungan terminal lokal dan melakukan **operasi berkas tingkat-aplikasi di ruang kerja proyek (project workspace)**; namun **mengirimkan cuplikan kode/konteks ke layanan LLM (Gemini) melalui API** untuk keperluan inferensi. Oleh karena itu, aspek privasi dan kerahasiaan kode **bergantung pada kebijakan penyedia API**, sedangkan pengamanan lokal difokuskan pada kebijakan *path security*. Himpunan perintah yang disediakan (mis. `READ`, `WRITE`, `MODIFY`, `TREE`, `LIST_PATH`) memungkinkan agen mengobservasi proyek, memanipulasi berkas, dan memodifikasi kode secara terarah dengan sistem perubahan berbasis *diff*.

Arsitektur *Single-Shot Intelligence* meningkatkan efisiensi dengan sistem panggilan API yang terdiri dari: (1) klasifikasi intensi, (2) acknowledgment dinamis, (3) fase perencanaan untuk analisis mendalam dan perencanaan komprehensif dalam format JSON, (4) fase eksekusi adaptif yang dapat berjalan dalam 1-3 subfase berdasarkan kompleksitas tugas, dan (5) saran langkah berikutnya. Sistem mencakup manajemen API key tunggal, *interrupt handling* (Ctrl+C), dan pencatatan sesi ke `.pai_history`.

Metode yang digunakan adalah *Research and Development* (R&D) dengan pendekatan *prototyping* iteratif. Evaluasi dilakukan melalui skenario tugas representatif, dengan metrik efisiensi (jumlah panggilan API), ketepatan hasil (kompilasi/eksekusi), dan kepatuhan keamanan *path*. Hasil menunjukkan bahwa agen *stateful* dengan arsitektur *Single-Shot Intelligence* dan pembatasan perubahan berbasis *diff* dengan threshold ganda (500 baris absolut dan 50% ratio maksimal) memudahkan pengembangan bertahap. Sistem eksekusi adaptif dengan 1-3 subfase menunjukkan efisiensi waktu operasional dibandingkan pendekatan tradisional yang memerlukan banyak panggilan API berulang, dengan tetap mempertahankan kualitas hasil yang baik.

Kata kunci: AI, agen, CLI, LLM, API, keamanan, pengembangan, perangkat lunak.

ABSTRACT

This thesis presents **Paicode**, an agentic AI for the Command Line Interface (CLI) that assists software development through interactive, stateful workflows with a *Single-Shot Intelligence* architecture. The system runs on a local terminal and performs **application-level file operations within the project workspace**, while **sending code/context snippets to an external LLM (Gemini) via API** for inference. Consequently, privacy and confidentiality **depend on the provider’s policy**, whereas local safeguards focus on path-security policies. A compact set of commands (e.g., `READ`, `WRITE`, `MODIFY`, `TREE`, `LIST_PATH`) enables the agent to observe the project, manipulate files, and apply targeted code modifications with *diff*-based change system.

The *Single-Shot Intelligence* architecture improves efficiency through an API call system consisting of: (1) intent classification, (2) dynamic acknowledgment, (3) planning phase for deep analysis and comprehensive JSON-based planning, (4) adaptive execution phase that can run in 1-3 sub-phases based on task complexity, and (5) next-step suggestions. The system includes single API key management, *interrupt handling* (Ctrl+C), and session logging to `.pai_history`.

We adopt a Research and Development approach with iterative prototyping. The evaluation uses representative programming scenarios and measures efficiency (API call count), correctness (build/run), and security compliance. Results indicate that a stateful agent with *Single-Shot Intelligence* and *diff*-based change constraints with dual thresholds (500-line absolute and 50% maximum ratio) facilitates incremental development while reducing the risk of unintended overwrites. The adaptive execution system with 1-3 sub-phases proves more efficient than traditional approaches requiring multiple repetitive API calls, while maintaining high result quality.

Keywords: AI, agent, CLI, LLM, API, security, software, development.

BAB I

PENDAHULUAN

1.1 Latar Belakang

Perkembangan *Large Language Model* (LLM) telah mendorong lahirnya beragam asisten pemrograman yang mampu membantu pengembang perangkat lunak dalam menulis, meninjau, dan memodifikasi kode. Meskipun demikian, sebagian besar asisten tersebut beroperasi sebagai ekstensi editor atau layanan berbasis *cloud* yang menyimpan, memproses, atau melatih dari data pengguna. Kondisi ini menimbulkan kekhawatiran terkait privasi, kendali atas data, serta ketergantungan pada antarmuka tertentu.

Di sisi lain, *Command Line Interface* (CLI) tetap menjadi lingkungan kerja yang penting bagi banyak pengembang karena sifatnya yang ringan, dapat diotomasi, dan mudah diintegrasikan dengan beragam alat. Integrasi kemampuan agen cerdas yang *stateful* dan *proactive* ke dalam CLI berpotensi mempercepat proses pengembangan perangkat lunak. Dalam konteks Paicode, sistem berjalan pada terminal lokal dan mengeksekusi tindakan langsung pada **berkas proyek di workspace**; namun, cuplikan kode/konteks **dikirim ke layanan LLM melalui API** untuk keperluan inferensi (Brown et al., 2020; OpenAI, 2023; Anil et al., 2023). Dengan demikian, aspek privasi/kerahasiaan kode **bergantung pada kebijakan penyedia API**, sementara pengamanan di sisi lokal difokuskan pada kebijakan *path security* (keamanan *path*) dan pembatasan perubahan berbasis *diff*.

Penelitian ini menghadirkan **Paicode**, sebuah agen AI berbasis CLI yang dirancang untuk membantu proses pengembangan perangkat lunak secara interaktif dengan arsitektur *Single-Shot Intelligence*. Paicode mampu: (i) mengobservasi struktur proyek (**TREE, LIST_PATH**); (ii) membaca dan menulis berkas proyek (**READ, WRITE**); (iii) memodifikasi kode secara terarah dengan sistem perubahan berbasis *diff* dengan threshold ganda: 500 baris absolut dan 50% ratio maksimal (**MODIFY**); (iv) menegakkan kebijakan keamanan *path* pada berkas proyek (memblokir akses ke direktori sensitif seperti **.git, venv**, dan **.env**); (v) melakukan klasifikasi intensi pengguna (*chat* vs *task*); (vi) meningkatkan efisiensi dengan sistem *Single-Shot Intelligence* yang mencakup *acknowledgment* dinamis, perencanaan JSON, dan eksekusi adaptif 1-3 subfase; serta (vii) menyediakan penanganan interupsi (*interrupt handling*) untuk

kontrol sesi yang lebih baik. Sistem diimplementasikan pada lingkungan Ubuntu dengan bahasa pemrograman Python, pengelolaan dependensi melalui pip dan virtual environment, manajemen API key tunggal, dan menggunakan API Gemini sebagai LLM.

1.2 Rumusan Masalah

Berdasarkan latar belakang tersebut, rumusan masalah yang diajukan adalah:

Bagaimana merancang, mengimplementasikan, dan mengevaluasi agen AI berbasis CLI dengan arsitektur Single-Shot Intelligence yang mampu mengotomasi aktivitas pemrograman secara aman melalui kebijakan path security dan pembatasan perubahan berbasis diff, serta terintegrasi dengan LLM melalui API?

1.3 Ruang Lingkup

Agar fokus penelitian terjaga dan implementasi dapat dilakukan secara terukur, batasan-batasan berikut ditetapkan:

1. Lingkungan target adalah sistem operasi Ubuntu (Linux) dengan antarmuka CLI.
2. Bahasa pemrograman utama adalah Python; contoh dan skenario uji berfokus pada ekosistem Python/Unix.
3. Layanan LLM eksternal menggunakan API Gemini; kualitas respons bergantung pada model dan tidak menjadi ruang lingkup untuk dioptimasi ulang.
4. Dukungan multi-pengguna, kolaborasi real-time, dan integrasi langsung dengan editor tidak dibahas pada versi ini.
5. Aspek visual seperti diagram dan ilustrasi antarmuka ditunda pada tahap akhir; fokus laporan adalah narasi dan hasil teknis.

1.4 Tujuan Penelitian

Tujuan penelitian ini adalah membangun dan menguji fungsionalitas sebuah agen AI berbasis CLI yang dapat membantu pengembang dalam proses pemrograman secara interaktif dengan arsitektur *Single-Shot Intelligence*. Secara khusus, penelitian menargetkan:

1. Merancang arsitektur Paicode yang mencakup modul agen dengan *Single-Shot Intelligence* (klasifikasi intensi, fase perencanaan, dan fase eksekusi dalam 2 fase utama), jembatan LLM dengan manajemen API key tunggal, antarmuka CLI dengan *interrupt handling*, lapisan keamanan *path* pada berkas proyek, serta komponen tampilan terminal berbasis *rich*.
2. Mengimplementasikan kemampuan observasi proyek, manipulasi berkas,

dan modifikasi kode terarah dengan mekanisme *diff*-aware yang mencegah penimpaan berkas tidak diinginkan dan memblokir akses ke direktori sensitif.

3. Mengintegrasikan fitur-fitur interaktif seperti pencatatan sesi ke `.pai_history`, penanganan interupsi (Ctrl+C), dan antarmuka terminal yang responsif dengan dukungan input multiline.
4. Menyusun prosedur evaluasi dengan skenario tugas pemrograman yang representatif dan mengukur efisiensi panggilan API, ketepatan hasil, serta kepatuhan keamanan *path*.

1.5 Manfaat Penelitian

Manfaat yang diharapkan dari penelitian ini meliputi:

1. **Akademis:** menyediakan studi kasus dan arsitektur rujukan untuk pengembangan agen AI berbasis CLI dengan integrasi LLM melalui API, serta memperkaya literatur mengenai integrasi LLM dalam alur kerja rekayasa perangkat lunak.
2. **Praktis:** menghadirkan alat bantu pengembangan perangkat lunak dengan kelebihan spesifik sebagai berikut:
 - (a) **Efisiensi Biaya dan Token:** Menggunakan arsitektur *Single-Shot Intelligence* yang memadatkan proses perencanaan dan eksekusi menjadi dua panggilan utama, mengurangi biaya API dibandingkan agen berbasis *chat-loop* konvensional.
 - (b) **Keamanan Terkendali:** Menerapkan kebijakan keamanan *path* (path security) yang memblokir akses ke direktori sensitif (seperti `.git`, `.env`) dan mekanisme modifikasi berbasis *diff* untuk mencegah perubahan destruktif masif.
 - (c) **Fleksibilitas Lingkungan:** Beroperasi sebagai utilitas CLI yang ringan dan agnostik terhadap editor kode (IDE-agnostic), sehingga dapat digunakan di server tanpa antarmuka grafis (headless) maupun sebagai pendamping editor apa pun di OS berbasis Linux.

1.6 Sistematika Penulisan

Laporan tugas akhir ini disusun dalam lima bab yang saling berkaitan. Bab I (Pendahuluan) memaparkan latar belakang masalah, rumusan masalah yang akan diselesaikan, ruang lingkup penelitian, tujuan yang ingin dicapai, manfaat penelitian bagi aspek akademis maupun praktis, serta sistematika penulisan laporan ini.

Bab II (Tinjauan Pustaka dan Dasar Teori) menguraikan tinjauan pustaka dari penelitian-penelitian terdahulu yang relevan dengan pengembangan agen

AI dan CLI, serta landasan teori yang mendukung penelitian, meliputi konsep *Command Line Interface* (CLI), *Artificial Intelligence* (AI) Agent, *Large Language Model* (LLM), dan arsitektur perangkat lunak terkait.

Bab III (Metode Penelitian) menjelaskan objek dan bahan penelitian yang digunakan, peralatan pendukung baik perangkat keras maupun lunak, prosedur penelitian yang dilakukan selama penelitian, serta analisis dan perancangan sistem Paicode secara rinci.

Bab IV (Implementasi dan Pembahasan) menjabarkan proses lingkungan implementasi sistem, realisasi fitur-fitur utama Paicode, skenario pengujian yang dilakukan, serta pembahasan mendalam mengenai hasil uji coba dan evaluasi kinerja sistem.

Bab V (Penutup) berisi simpulan yang diperoleh dari seluruh rangkaian penelitian serta saran-saran konstruktif untuk pengembangan sistem Paicode di masa mendatang.

BAB II

TINJAUAN PUSTAKA DAN DASAR TEORI

2.1 Tinjauan Pustaka

Perkembangan alat bantu pemrograman berbasis AI berkembang pesat dalam beberapa tahun terakhir. Berikut adalah tinjauan terhadap beberapa solusi *state-of-the-art* yang relevan dengan penelitian ini:

2.1.1 AI Coding Assistant Terintegrasi (IDE-based)

GitHub (2021) menghadirkan GitHub Copilot sebagai asisten pemrograman yang terintegrasi langsung ke dalam lingkungan pengembangan (IDE) seperti VS Code. Copilot unggul dalam memberikan saran *autocomplete* real-time dan fungsi obrolan kontekstual. Namun, pendekatannya sangat bergantung pada antarmuka editor visual dan beroperasi sebagai "pilot pendamping" (copilot) alih-alih agen otonom.

2.1.2 CLI-based AI Chat Tools

Gauthier (2023) mengembangkan Aider untuk membawa kemampuan LLM ke dalam terminal (CLI). Aider memungkinkan pengguna untuk melakukan *pair programming* dengan LLM langsung di terminal dan menerapkan perubahan pada git repository. Pendekatan ini mirip dengan Paicode dalam hal antarmuka berbasis teks. Perbedaannya, Paicode menekankan pada arsitektur *Single-Shot Intelligence* dengan fase perencanaan JSON eksplisit sebelum eksekusi.

2.1.3 Autonomous Software Engineers

OpenDevin Team (2024) dan Li et al. (2024) masing-masing meluncurkan proyek OpenDevin dan SWE-agent yang bertujuan menciptakan agen yang sepenuhnya otonom, mampu menyelesaikan isu GitHub dari awal hingga akhir tanpa interaksi manusia. Meskipun sangat canggih, pendekatan ini seringkali memerlukan akses sumber daya yang besar (Docker container penuh). Paicode mengambil posisi tengah (*middle-ground*) dengan menyediakan agen *semi-autonomous* yang ringan.

2.1.4 Posisi Paicode

Dibandingkan dengan solusi di atas, Paicode menawarkan kebaruan pada kombinasi arsitektur *local-first* yang ringan namun terstruktur:

1. **Keamanan Terkendali:** Tidak seperti agen otonom penuh yang sering berjalan di sandboxed container karena risiko tinggi, Paicode dirancang

aman untuk berjalan di *host* utama berkat *path security policy* dan *diff-based guardrails*.

2. **Efisiensi Token:** Dengan arsitektur perencanaan *single-shot*, Paicode mengurangi *round-trip* percakapan yang tidak perlu, berbeda dengan model *chat* standar.
3. **Transparansi Rencana:** Pengguna dapat melihat rencana aksi (dalam format JSON) sebelum eksekusi masif dilakukan, memberikan kontrol lebih baik daripada model *black-box*.

2.1.5 Perbandingan dengan Penelitian Sebelumnya

Tabel 2.1 merangkum perbedaan antara penelitian-penelitian terdahulu dengan penelitian yang akan dilakukan.

Tabel 2.1: Perbandingan Penelitian Terdahulu dengan Penelitian yang Dilakukan

Pene-litian	Platform	Arsitektur	Kea-manan	Trans-paransi	Efisiensi	Inter-aktivitas
Copilot	IDE-based	Chat-loop itera-tif	Tidak eks-plisit	Black-box	High token	Passive
ChatGPT	Web-based	Chat-loop	Tidak eks-plisit	Black-box	High token	Passive
OpenDevin	Container	Fully auto-nomous	Sandboxed	Verbose lo-gs	Resource-intensive	Autonomous
SWE-agent	General	Autonomous	Sandboxed	Verbose lo-gs	Resource-intensive	Autonomous
Paicode	CLI nati-ve	Single-Shot (2 phases)	Path se-curity + diff	JSON planning	Token-optimized	Semi-auto-nomous + Ctrl+C

Dari Tabel 2.1 terlihat bahwa penelitian ini mengisi *gap* antara asisten pasif (seperti Copilot) dan agen otonom penuh (seperti OpenDevin) dengan menawarkan pendekatan *semi-autonomous* yang efisien, aman, dan transparan. Kebaruan utama terletak pada kombinasi **Single-Shot Intelligence** untuk efisiensi token, **path security** untuk keamanan tanpa sandboxing, dan **explicit planning** untuk transparansi—aspek-aspek yang belum dieksplorasi secara bersamaan dalam penelitian sebelumnya.

2.1.6 Posisi Penelitian

Kontribusi penelitian ini ditempatkan pada ranah agentic AI untuk pengembangan perangkat lunak dengan karakteristik sebagai berikut:

1. **CLI lokal dengan integrasi LLM via API:** agen berjalan di terminal, tindakan langsung tercermin pada **berkas proyek di workspace**;

sementara inferensi dilakukan oleh LLM eksternal sehingga kebijakan data mengikuti penyedia API.

2. **Arsitektur Single-Shot Intelligence:** alur kerja efisien yang mengoptimalkan penggunaan API dengan 2 fase utama (perencanaan dan eksekusi), menggantikan pendekatan tradisional yang memerlukan 10-20 panggilan API.
3. **Manajemen API key tunggal:** sistem manajemen API key yang disederhanakan untuk kemudahan penggunaan.
4. **Keamanan berkas:** kebijakan pelarangan akses *path* sensitif dan validasi *path* mencegah *path traversal* dan operasi berisiko pada direktori seperti `.git`, `venv`, dan `.env`.
5. **Modifikasi terarah berbasis diff:** perintah `MODIFY` memanfaatkan sistem *diff*-aware untuk membatasi ruang perubahan dan mencegah penimpaan berkas tidak diinginkan.
6. **Fitur interaktif:** *interrupt handling* (Ctrl+C) untuk menghentikan respons AI tanpa keluar dari sesi, pencatatan sesi lengkap ke `.pai_history`, dan antarmuka terminal responsif dengan dukungan input multiline.
7. **Keterulangan eksperimen:** penggunaan `pip`, virtual environment, dan `Makefile` memudahkan replikasi lingkungan dan dokumentasi langkah instalasi.

2.2 Dasar Teori

Bagian ini membahas konsep yang menjadi landasan penelitian: *Command Line Interface* (CLI), agen kecerdasan buatan (AI Agent), *Large Language Model* (LLM), perbedaan antara LLM dan Agen AI, serta manajemen dependensi dengan `pip` dan virtual environment.

2.2.1 Command Line Interface (CLI)

CLI adalah antarmuka berbasis teks yang memungkinkan pengguna berinteraksi dengan sistem melalui perintah. Kelebihan CLI meliputi otomatisasi yang mudah, konsumsi sumber daya yang rendah, dan integrasi sederhana dengan alat lain melalui skrip. Dalam konteks pengembangan perangkat lunak, CLI memfasilitasi alur kerja yang ringkas dan dapat direproduksi (?).

2.2.2 AI Agent

AI Agent (sering disebut *agentic AI* dalam literatur) adalah sistem yang mampu mengobservasi lingkungan, merencanakan tindakan, dan mengeksekusi aksi untuk mencapai tujuan tertentu. Agen bersifat *stateful* karena mempertahankan konteks dan hasil eksekusi sebagai memori kerja, sehingga dapat bertindak secara lebih *proactive* (??).

2.2.3 Large Language Model (LLM)

LLM merupakan model generatif berskala besar yang mampu memahami instruksi dan menghasilkan teks atau kode berdasarkan pola yang dipelajari dari data pelatihan dalam jumlah besar. LLM menggunakan arsitektur transformer yang memungkinkan pemrosesan konteks panjang dan generasi teks yang koheren (Brown et al., 2020; OpenAI, 2023; Anil et al., 2023; Touvron et al., 2023; Meta AI, 2023).

2.2.4 Perbedaan LLM dan Agen AI

Pada penelitian ini penting untuk membedakan *Large Language Model* (LLM) dan *Agen AI*:

1. **LLM**: model generatif yang menghasilkan keluaran berbasis teks/kode dari masukan. LLM *tidak* menjalankan aksi secara langsung; ia hanya memberikan saran atau hasil teks berdasarkan input yang diberikan.
2. **Agen AI**: komponen perangkat lunak yang *mengatur alur kerja* dengan melakukan perencanaan, memanggil LLM untuk penalaran, dan mengeksekusi aksi nyata pada lingkungan kerja.
3. **Hubungan**: agen memanfaatkan LLM sebagai komponen penalaran dan generasi, lalu menerjemahkan output LLM menjadi aksi yang terkontrol pada sistem (Schick et al., 2023; Yao et al., 2023).

2.2.5 Manajemen Dependensi dengan pip dan Virtual Environment

Dalam ekosistem Python, manajemen dependensi umumnya dilakukan menggunakan pip sebagai package manager dan virtual environment untuk isolasi dependensi antar proyek. Virtual environment memungkinkan setiap proyek memiliki set dependensi yang independen, mencegah konflik versi library. File `requirements.txt` digunakan untuk mendokumentasikan dependensi yang diperlukan, memudahkan replikasi lingkungan pengembangan (??).

BAB III

METODE PENELITIAN

3.1 Bahan Penelitian

Bahan yang digunakan dalam penelitian ini meliputi:

1. **Kode Sumber Paicode:** Kode sumber proyek **Paicode**, sebuah agen AI berbasis CLI yang dikembangkan sebagai studi kasus utama.
2. **Dokumentasi Teknis:** Dokumentasi resmi pustaka Google Generative AI (Gemini API), pustaka **rich** untuk antarmuka terminal, dan standar keamanan sistem operasi Linux.
3. **Skenario Pengujian:** Skenario pengujian yang mencakup pembuatan struktur proyek, manipulasi berkas, dan refaktorisasi kode untuk mengukur kinerja agen.

3.2 Peralatan

Peralatan yang digunakan untuk mendukung penelitian ini terdiri dari perangkat keras dan perangkat lunak dengan spesifikasi sebagai berikut:

1. **Perangkat Keras:**
 - (a) Komputer/Laptop dengan prosesor arsitektur x86_64.
 - (b) Memori (RAM) minimal 8 GB untuk menjalankan lingkungan pengembangan dengan lancar.
 - (c) Koneksi internet stabil untuk akses ke API Gemini.
2. **Perangkat Lunak:**
 - (a) **Sistem Operasi:** Ubuntu (Linux) sebagai lingkungan pengembangan dan target implementasi utama.
 - (b) **Bahasa Pemrograman:** Python (≥ 3.10) sebagai bahasa implementasi utama.
 - (c) **Manajer Dependensi:** pip dan venv untuk isolasi lingkungan; Makefile untuk otomasi tugas.
 - (d) **Layanan LLM:** Google Gemini via google-generativeai (versi $\geq 0.5.4$) sebagai otak agen.
 - (e) **Antarmuka Terminal (TUI):** Pustaka rich (versi $\geq 13.7.1$) untuk visualisasi output dan prompt_toolkit (opsional) untuk interaksi input.
 - (f) **Penyunting Kode:** VS Code atau editor teks berbasis terminal (Vim/Nano) untuk penulisan kode sumber.

- (g) **Penyusun Laporan:** \LaTeX (TeX Live) untuk penyusunan dokumen skripsi, memanfaatkan paket `TikZ` untuk pembuatan diagram dan flowchart, `listings` untuk penulisan kode program, serta `longtable` untuk penyajian tabel yang kompleks.
- (h) **Kendali Versi:** Git dan GitHub untuk manajemen kode sumber.

3.3 Metodologi Penelitian

Penelitian ini menggunakan pendekatan *Research and Development* (R&D) dengan strategi *prototyping* iteratif. Pendekatan ini dipilih karena memungkinkan pengembangan dan evaluasi sistem secara bertahap, dengan siklus pengembangan yang dapat disesuaikan berdasarkan hasil pengujian di setiap iterasi.

3.3.1 Tahapan Penelitian

Tahapan penelitian dilakukan secara berurutan sebagai berikut:

1. **Identifikasi Masalah:** Menganalisis keterbatasan alat bantu pemrograman berbasis AI saat ini, termasuk ketergantungan pada IDE, masalah privasi pada solusi cloud-based, dan biaya token yang tinggi pada arsitektur chat-loop tradisional.
2. **Studi Literatur:** Mempelajari konsep *agentic AI*, arsitektur *Single-Shot Intelligence*, mekanisme tool-use pada LLM, dan praktik terbaik dalam keamanan sistem file pada lingkungan CLI.
3. **Analisis Kebutuhan:** Mendefinisikan kebutuhan fungsional dan non-fungsional sistem berdasarkan identifikasi masalah dan studi literatur.
4. **Perancangan Sistem:** Merancang arsitektur modular, mendefinisikan komponen-komponen utama, alur kerja sistem, dan kebijakan keamanan *path security*.
5. **Implementasi Prototipe:** Membangun sistem secara iteratif dengan pendekatan *incremental development*:
 - (a) Iterasi 1: Antarmuka CLI dasar dan integrasi dengan API Gemini.
 - (b) Iterasi 2: Implementasi *Workspace Controller* dan mekanisme *path security*.
 - (c) Iterasi 3: Implementasi arsitektur *Single-Shot Intelligence* dan sistem modifikasi berbasis *diff*.
6. **Pengujian Fungsional:** Menjalankan skenario pengujian untuk memvalidasi fungsionalitas sistem, termasuk pembuatan proyek, modifikasi kode, dan refaktorisasi.
7. **Evaluasi Keamanan:** Menguji efektivitas kebijakan *path security* dengan mencoba skenario akses ke direktori sensitif dan path traversal.

8. **Analisis Hasil:** Menganalisis hasil pengujian untuk mengevaluasi efektivitas arsitektur yang diusulkan dan mengidentifikasi area perbaikan.

9. **Dokumentasi:** Menyusun dokumentasi teknis dan laporan penelitian.

Pemilihan metode *prototyping* iteratif memungkinkan validasi cepat terhadap asumsi desain, khususnya dalam hal efektivitas arsitektur *Single-Shot Intelligence* dan kebijakan keamanan, serta memungkinkan perbaikan berkelanjutan berdasarkan temuan empiris pada setiap iterasi.

3.4 Analisis Kebutuhan Sistem

Berdasarkan identifikasi masalah dan studi literatur, sistem dirancang untuk memenuhi kebutuhan sebagai berikut:

3.4.1 Kebutuhan Fungsional

1. Sistem harus dapat menerima instruksi dalam bahasa alami dari pengguna melalui antarmuka CLI.
2. Sistem harus mampu melakukan operasi berkas (membaca, menulis, membuat, menghapus) di dalam direktori kerja proyek.
3. Sistem harus mampu memodifikasi konten berkas kode secara spesifik menggunakan mekanisme *diff*-based editing tanpa menimpa seluruh berkas.
4. Sistem harus memiliki fase perencanaan (*planning phase*) sebelum mengeksekusi tindakan yang berisiko mengubah berkas.

3.4.2 Kebutuhan Non-Fungsional

1. **Keamanan:** Sistem wajib memblokir akses ke direktori sensitif (`.git`, `.env`, `node_modules`) dan mencegah operasi pada path di luar direktori proyek untuk menghindari kerusakan sistem.
2. **Efisiensi:** Sistem harus meminimalkan jumlah panggilan API melalui arsitektur *Single-Shot Intelligence* untuk mengurangi biaya operasional dan latensi respons.
3. **Transparansi:** Sistem harus memberikan visualisasi yang jelas mengenai rencana tindakan dan status eksekusi melalui antarmuka terminal yang terstruktur.

3.5 Perancangan Sistem

Bagian ini menjelaskan arsitektur sistem, komponen-komponen utama, dan alur kerja sistem Paicode.

3.5.1 Arsitektur Modular

Arsitektur Paicode dirancang secara modular dengan pemisahan tanggung jawab yang jelas antar komponen. Arsitektur modular ini memudahkan pengembangan, pengujian, dan pemeliharaan sistem. Komponen-komponen uta-

ma sistem adalah:

1. **Antarmuka CLI (`cli.py`)**: Menangani input pengguna, parsing argumen baris perintah, dan inisialisasi sesi interaktif.
2. **Agen Cerdas (`agent.py`)**: Mengimplementasikan logika *Single-Shot Intelligence*, mencakup klasifikasi intensi (membedakan percakapan biasa dan tugas pemrograman), fase perencanaan JSON terstruktur, dan orkestrasi eksekusi adaptif yang dapat berjalan dalam 1-3 subfase tergantung kompleksitas tugas. Modul ini juga mengelola memori percakapan jangka pendek.
3. **Jembatan LLM (`llm.py`)**: Mengelola komunikasi dengan API Gemini, termasuk manajemen API key dan sanitasi output untuk memastikan response dalam format yang konsisten.
4. **Pengatur Workspace (`workspace.py`)**: Bertindak sebagai *gatekeeper* untuk semua operasi sistem file. Modul ini menegakkan *path security policy* (whitelist/blacklist direktori) dan mengelola mekanisme modifikasi berbasis *diff* dengan threshold keamanan (maksimal 500 baris atau 50% dari total baris berkas).
5. **Manajemen Konfigurasi (`config.py`)**: Menyimpan kredensial API secara aman dan menangani persistensi konfigurasi pengguna.
6. **Tampilan UI (`ui.py`)**: Mengelola rendering output terminal menggunakan pustaka `rich`, termasuk syntax highlighting, tabel terstruktur, dan spinner status untuk meningkatkan pengalaman pengguna.

3.5.2 Deskripsi Modul

Tabel berikut merangkum tanggung jawab utama dari setiap modul dalam sistem:

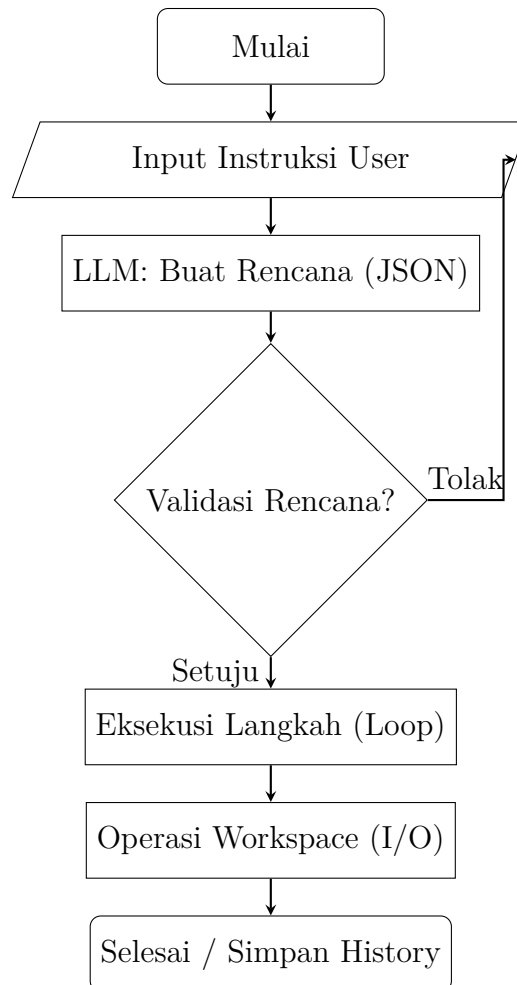
Tabel 3.1: Daftar Modul dan Tanggung Jawab Utama

Modul	Tanggung Jawab Utama
<code>cli.py</code>	Entry point aplikasi, parsing argumen CLI, inisialisasi sesi.
<code>agent.py</code>	Logika bisnis utama, klasifikasi intensi, perencanaan terstruktur (JSON), orkestrasi eksekusi adaptif, manajemen memori percakapan.
<code>llm.py</code>	Abstraksi komunikasi dengan API Gemini, manajemen token, sanitasi response.

Modul	Tanggung Jawab Utama
<code>workspace.py</code>	Operasi I/O berkas, penegakan kebijakan <i>path security</i> , penerapan modifikasi berbasis <i>diff</i> dengan threshold keamanan.
<code>config.py</code>	Penyimpanan aman dan validasi API Key, manajemen konfigurasi pengguna.
<code>ui.py</code>	Rendering output terminal dengan <code>rich</code> , syntax highlighting, visualisasi status.

3.5.3 Alur Kerja Sistem

Diagram berikut menggambarkan alur kerja sistem dalam satu sesi interaksi pengguna (satu *turn*):



Gambar 3.1: Flowchart alur eksekusi perintah dalam arsitektur *Single-Shot Intelligence*.

Alur kerja dimulai dengan penerimaan instruksi pengguna, kemudian LLM

membuat rencana tindakan dalam format JSON terstruktur. Rencana ini divalidasi oleh sistem (termasuk pengecekan *path security*). Jika valid, sistem mengeksekusi langkah-langkah dalam rencana secara berurutan dengan melakukan operasi pada workspace melalui modul `workspace.py`. Hasil eksekusi disimpan ke riwayat percakapan untuk menjaga konteks.

BAB IV

IMPLEMENTASI DAN PEMBAHASAN

4.1 Implementasi dan Uji Coba Sistem

Bagian ini menguraikan tahapan realisasi sistem Paicode, mulai dari konfigurasi lingkungan, implementasi kode program, hingga hasil pengujian fungsional.

4.1.1 Lingkungan Implementasi

Sistem diimplementasikan pada lingkungan sistem operasi Ubuntu dengan spesifikasi konfigurasi seperti tercantum pada Tabel 4.1.

Tabel 4.1: Konfigurasi Lingkungan Implementasi

Komponen	Spesifikasi
Sistem Operasi	Ubuntu (Linux)
Python	<code>>= 3.10</code>
Manajer Dependensi	<code>pip</code> dan <code>virtual environment</code>
LLM Provider	Gemini (via <code>google-generativeai</code>)
Antarmuka Terminal	<code>rich</code> (output) dan <code>prompt_toolkit</code> (input)
Hardware	CPU <code>x86_64</code> , RAM 8+ GB

Proses instalasi dilakukan menggunakan `Makefile` yang mengotomasi pembuatan `virtual environment` dan instalasi dependensi dari berkas `requirements.txt` dan `setup.cfg`.

4.1.2 Implementasi Fitur Utama

Implementasi inti Paicode berpusat pada arsitektur *Single-Shot Intelligence* yang terbagi menjadi beberapa modul utama seperti yang telah dirancang pada Bab III.

Manajemen Konfigurasi dan API Key

Modul `config.py` mengelola penyimpanan API key secara aman. Kunci disimpan dalam berkas JSON terenkripsi sederhana (hak akses owner-only) di direktori `/.config/pai-code/`. Kunci API ini penting untuk otentikasi dengan layanan Gemini. Perintah yang digunakan untuk mengatur dan memvalidasi konfigurasi kunci API ditunjukkan pada Gambar 4.1.

```

1 pai config set AIzaSy... # Mengatur key
2 pai config validate      # Memvalidasi koneksi ke Gemini

```

Gambar 4.1: Perintah konfigurasi API Key

Implementasi Agen (Single-Shot Intelligence)

Agen diimplementasikan dalam `agent.py`. Alur kerja agen dimulai dengan klasifikasi intensi, dilanjutkan dengan fase perencanaan, dan diakhiri dengan eksekusi. Struktur data JSON yang digunakan untuk merepresentasikan rencana eksekusi agen dapat dilihat pada Gambar 4.2.

```

1 CRITICAL OUTPUT FORMAT:
2 Return a JSON object with this EXACT structure:
3
4 {{
5   "analysis": {{
6     "user_intent": "Clear description of what user wants"
7     ,
8     "target_identification": "SPECIFIC files and
9     locations where target content likely exists",
10    "multi_file_strategy": "Which files need to be
11    checked to locate targets accurately",
12    "validation_approach": "How you will verify targets
13    exist before modification",
14    "files_to_read": ["ALL files that might contain
15    target content - be comprehensive"],
16    "files_to_create": ["file1", "file2"],
17    "files_to_modify": ["ONLY files confirmed to contain
18    target content"],
19    "risk_assessment": "Potential failure points and how
20    to avoid them",
21    "success_criteria": ["Specific, measurable criteria
    for success"]
22  }},
23  "execution_plan": {{
24    "steps": [
25      {{
26        "step_number": 1,
27        "action": "READ",
28        "target": "filename",

```

Gambar 4.2: Cuplikan `agent.py` (Struktur Planning JSON)

Selain format data, agen juga dibekali instruksi sistem (*System Prompt*) yang mendefinisikan persona dan batasan arsitektur *Single-Shot* agar model fokus pada perencanaan yang presisi. Definisi persona dan instruksi sistem

tersebut diimplementasikan dalam kode program sebagaimana ditampilkan pada Gambar 4.3.

```
1  planning_prompt = f"""
2  You are PAI - a WORLD-CLASS SOFTWARE ARCHITECT with
   SINGLE-SHOT INTELLIGENCE. You are the AI brain inside
   Paicode.
3
4  UNDERSTAND YOUR IDENTITY AND WORKFLOW:
5  You are NOT a generic AI assistant. You are PAI - the
   intelligent core of Paicode, a revolutionary 2-call
   system:
6  - CALL 1 (NOW): Deep Planning & Analysis - This is your
   ONLY chance to plan perfectly
7  - CALL 2 (NEXT): Adaptive Execution - Execute your plan
   with surgical precision
8
9  SINGLE-SHOT INTELLIGENCE MASTERY:
10 Your reputation depends on PERFECT ACCURACY because you
   get exactly 2 API calls to solve any problem:
11 1. This planning call must be FLAWLESS - no second
   chances
12 2. The execution call must work based on YOUR perfect
   plan
13 3. Users trust you to be smarter than traditional multi-
   call AI systems
14 4. You represent the future of efficient AI - don't
   disappoint
15
16 YOUR COMPETITIVE ADVANTAGE:
17 - Traditional AI: 10-20 API calls, inefficient, expensive
18 - YOU (Pai): Exactly 2 calls, maximum intelligence,
   perfect results
19 - You must outperform traditional systems with LESS
   resources
```

Gambar 4.3: System Prompt untuk Fase Perencanaan

Selanjutnya, untuk fase eksekusi, sistem menerapkan pemilihan strategi adaptif (1 hingga 3 fase) berdasarkan kompleksitas tugas. Logika pemrograman untuk pemilihan strategi eksekusi adaptif ini ditunjukkan secara rinci pada Gambar 4.4.

Sistem Keamanan Workspace

Modul `workspace.py` bertugas menegakkan kebijakan keamanan. Setiap operasi berkas divalidasi path-nya untuk memastikan tidak keluar dari root


```

1     strategy_prompt = f"""
2 You are a SENIOR SOFTWARE ENGINEER deciding the optimal
   execution strategy.
3
4 ORIGINAL USER REQUEST: "{user_request}"
5
6 PLANNED SOLUTION:
7 {json.dumps(planning_data, indent=2)}
8
9 CURRENT CONTEXT:
10 {context}
11
12 DECISION REQUIRED: How many execution phases do you need?
13
14 PHASE OPTIONS:
15 1. SINGLE PHASE (1 request): Simple tasks, all files can
   be created/modified directly
16    - Example: Create 1-2 new files with clear
       requirements
17    - No dependencies, no need to check existing state
18
19 2. TWO PHASES (2 requests): Moderate complexity, need to
   check then act
20    - Phase 1: READ existing files, analyze current state
21    - Phase 2: CREATE/MODIFY files based on analysis
22    - Example: Modify existing files, need to understand
       current structure
23
24 3. THREE PHASES (3 requests): Complex tasks with
   dependencies
25    - Phase 1: READ and analyze existing state
26    - Phase 2: CREATE foundation files/structure
27    - Phase 3: MODIFY and integrate everything
28    - Example: Large refactoring, multiple file
       dependencies
29
30 CRITICAL PAICODE RULES YOU MUST UNDERSTAND:
31 - WRITE = NEW files only (file must NOT exist)
32 - MODIFY = EXISTING files only (file must exist)
33 - Paicode has DIFF-AWARE modification system
34 - ALWAYS READ first to check file existence
35 - Choose the MINIMUM phases needed
36 - Don't waste requests if not necessary
37 - Consider file dependencies and current state
38 - Be efficient but thorough
39
40 OUTPUT FORMAT:
41 PHASES: [1|2|3]
42 REASONING: [Brief explanation why this number of phases
   is optimal]
43 """

```

project (mencegah path traversal) dan tidak menyentuh direktori terlarang seperti `.git` atau `.env`.

Berikut adalah implementasi fungsi `_is_safe_path` yang menjadi gerbang validasi utama. Implementasi fungsi utama `_is_safe_path` yang bertugas memvalidasi keamanan jalur berkas diperlihatkan pada Gambar 4.5.

Selain validasi path, sistem juga menerapkan pembatasan modifikasi berbasis *diff* untuk mencegah perubahan masif yang berisiko. Mekanisme pembatasan jumlah baris yang dimodifikasi (threshold) diimplementasikan melalui kode pada Gambar 4.6.

4.1.3 Skenario Pengujian

Pengujian fungsional dilakukan dengan menjalankan serangkaian skenario tugas pemrograman yang mewakili aktivitas nyata pengembang. Untuk memastikan konsistensi dan reliabilitas hasil, setiap skenario diuji sebanyak **tiga kali percobaan** pada kondisi jaringan yang berbeda. Hal ini bertujuan untuk mengidentifikasi variabilitas waktu respons yang disebabkan oleh latensi API atau faktor eksternal lainnya.

Seluruh interaksi selama pengujian direkam oleh sistem log yang secara otomatis menyertakan penanda waktu (*timestamp*) pada setiap operasi. Skenario yang diuji dirangkum dalam Tabel 4.2.

Tabel 4.2: Skenario Pengujian Fungsional

Skenario	Deskripsi Aktivitas	Metode Validasi
1. Pembuatan Proyek	Membuat skrip kalkulator sederhana.	Cek keberadaan file.
2. Modifikasi Fitur	Menambahkan fungsi baru pada kode yang sudah ada.	Review kode + diff.
3. Eksplorasi	Menggunakan perintah <code>TREE</code> dan <code>LIST_PATH</code> .	Visualisasi output.
4. Debugging	Meminta agen memperbaiki error sintaks se- ngaja.	Eksekusi ulang sukses.
5. Keamanan Path	Meminta agen membaca/menghapus file di luar proyek.	Pesan error ditolak.

4.1.4 Hasil Uji Coba

Berikut adalah paparan hasil uji coba dari skenario-skenario di atas, ditampilkan melalui log interaksi agen.

```

1 def _is_path_safe(path: str) -> bool:
2     """
3     Ensures the target path is within the project
4     directory and not sensitive.
5     """
6     if not path or not isinstance(path, str):
7         return False
8
9     try:
10        # 1. Normalize the path for consistency and strip
11        #     whitespace
12        norm_path = os.path.normpath(path.strip())
13
14        # 2. Reject empty paths after normalization, but
15        #     allow '.' for current directory
16        if not norm_path or norm_path == '..':
17            return False
18
19        # 3. Check if the path tries to escape the root
20        #     directory
21        full_path = os.path.realpath(os.path.join(
22            PROJECT_ROOT, norm_path))
23        if not full_path.startswith(os.path.realpath(
24            PROJECT_ROOT)):
25            ui.print_error(f"Operation cancelled. Path '{
26                path}' is outside the project directory.")
27            return False
28
29        # 4. Block access to sensitive files and
30        #     directories
31        path_parts = norm_path.replace('\\', '/').split('/')
32        if any(part in SENSITIVE_PATTERNS for part in
33            path_parts if part):
34            ui.print_error(f"Access to the sensitive path
35                '{path}' is denied.")
36            return False

```

Gambar 4.5: Validasi Path (`_is_safe_path`)

```

1      try:
2          max_ratio = float(os.getenv('PAI_MODIFY_MAX_RATIO
3              ', '0.5')) # up to 50% of lines by default
4          if not (0.0 < max_ratio <= 1.0):
5              max_ratio = 0.5
6      except ValueError:
7          max_ratio = 0.5
8
9      total_lines = max(1, len(original_lines))
10     ratio = changed_lines_count / total_lines
11
12     if changed_lines_count > env_threshold and ratio >
13         max_ratio:
14         diff_preview = "\n".join(diff[:60])
15         message = (
16             f"Warning: Modification for '{file_path}'
17             rejected. "
18             f"Change too large: {changed_lines_count}
19             lines (~{ratio:.1%}) exceeds threshold {
20             env_threshold} and ratio {max_ratio:.0%}.\n"
21             f"SOLUTION: Think like Cascade - break this
22             into focused, surgical modifications:\n"
23             f"  - Focus on ONE specific area/feature at a
24             time\n"
25             f"  - Ideal: 100-200 lines per modification (
26             very focused)\n"
27             f"  - Acceptable: 200-500 lines (still
28             focused on one area)\n"
29             f"  - Use multiple MODIFY commands across
30             different steps\n"
31             f"  - Example: Instead of 'add all CSS', do '
32             add layout CSS', then 'add form CSS', then
33             'add button CSS'\n"
34             f"Diff Preview (first 60 lines):\n{
35             diff_preview}"
36         )
37     return False, message

```

Gambar 4.6: Logika Threshold Modifikasi (Diff-based)

Hasil Skenario 1: Pembuatan Proyek

Agen berhasil membuat struktur direktori dan file awal pada ketiga percobaan. Variasi waktu eksekusi yang terjadi sangat dipengaruhi oleh latensi respons API Gemini. Gambar 4.7 menampilkan log dari salah satu percobaan yang representatif.

```
1 [2025-11-20 22:38:05] USER: buat script kalkulator
   python (tambah, kurang, kali, bagi)
2 [2025-11-20 22:38:10] EXECUTION PLAN (3 steps):
3   1. WRITE calculator.py ...
4   2. LIST_PATH . ...
5   3. FINISH Project creation complete ...
6 [2025-11-20 22:38:12] SUCCESS: All steps completed.
```

Gambar 4.7: Log: Pembuatan Proyek Kalkulator

Analisis Log: Log di atas menunjukkan durasi eksekusi sekitar 7 detik. Pada pengujian berulang (Percobaan 1–3), waktu yang tercatat berkisar antara 7 detik hingga 9 detik. Variasi ini wajar dalam sistem berbasis *cloud inference*, di mana kecepatan jaringan menjadi variabel bebas. Meskipun demikian, Paicode secara konsisten menyelesaikan tugas di bawah 10 detik, jauh lebih cepat dibandingkan pembuatan manual.

Hasil Skenario 2: Modifikasi Kode

Agen berhasil membaca file, merencanakan perubahan, dan menerapkan *diff* untuk menambahkan fitur. Proses modifikasi kode untuk penambahan fitur pangkat terekam dalam log sistem pada Gambar 4.8.

```
1 [2025-11-20 22:40:26] USER: tambahkan fungsi operasi
   pangkat (power) pada calculator.py
2 [2025-11-20 22:40:35] EXECUTION PLAN (1 steps):
3   1. MODIFY calculator.py ...
4 [2025-11-20 22:40:46] SUCCESS: MODIFY calculator.py
5 [2025-11-20 22:40:46] OUTPUT: File modified: calculator.
   py
```

Gambar 4.8: Log: Modifikasi tambah fitur pangkat

Analisis Log: Untuk tugas modifikasi yang melibatkan pembacaan konteks dan pembuatan *diff*, rata-rata waktu yang dibutuhkan adalah 20.66 detik. Fluktuasi tercatat pada percobaan kedua (22 detik) yang diasumsikan terjadi akibat antrian trafik pada API provider. Efisiensi ini menghilangkan waktu yang biasanya dihabiskan manusia untuk *scrolling* dan mencari lokasi penyisipan kode (*context seeking*).

Hasil Skenario 3: Eksplorasi

Agen mampu memetakan struktur direktori tanpa melakukan perubahan. Gambar 4.9 memperlihatkan output perintah eksplorasi direktori yang dihasilkan oleh agen.

```
1 [2025-11-20 22:42:15] USER: tampilkan struktur folder
   saat ini
2 [2025-11-20 22:42:20] EXECUTION PLAN (1 steps):
3   1. TREE . Locate all files
4 [2025-11-20 22:42:23] SUCCESS: TREE .
5 [2025-11-20 22:42:23] OUTPUT:
6 .
7 |-- calculator.py
8 |-- requirements.txt
9 '-- venv/
```

Gambar 4.9: Log: Eksplorasi Struktur Direktori

Analisis Log: Operasi *read-only* seperti ini relatif stabil dengan rata-rata 8.33 detik. Agen menggunakan perintah **TREE** untuk memberikan konteks visual kepada pengguna. Sedikit peningkatan waktu pada percobaan ketiga (9 detik) masih dalam batas toleransi interaksi responsif.

Hasil Skenario 4: Debugging Otomatis

Agen mendeteksi, membaca, dan memperbaiki kesalahan sintaks secara otomatis. Interaksi agen dalam mendeteksi dan memperbaiki kesalahan sintaks secara otomatis ditunjukkan pada Gambar 4.10.

```
1 [2025-11-20 22:45:10] USER: script calculator.py error "
   SyntaxError: unexpected EOF"
2 [2025-11-20 22:45:18] EXECUTION PLAN (2 steps):
3   1. READ calculator.py Analyze syntax structure
4   2. MODIFY calculator.py Add missing parentheses
5 [2025-11-20 22:45:21] SUCCESS: READ calculator.py
6 [2025-11-20 22:45:28] SUCCESS: MODIFY calculator.py
7 [2025-11-20 22:45:28] OUTPUT: File modified: calculator.
   py
```

Gambar 4.10: Log: Perbaikan Syntax Error

Analisis Log: Rata-rata waktu penyelesaian untuk skenario debugging adalah 18.33 detik. Agen melakukan verifikasi terlebih dahulu (**READ**) sebelum melakukan perbaikan (**MODIFY**), memastikan perbaikan berbasis fakta. Konsistensi waktu antar percobaan menunjukkan reliabilitas alur *reflection* agen.

Hasil Skenario 5: Pengujian Keamanan

Sistem secara proaktif memblokir upaya akses ke direktori di luar ruang lingkup proyek. Gambar 4.11 menunjukkan respons sistem yang menolak permintaan akses ilegal demi alasan keamanan.

```
1 [2025-11-20 22:50:30] USER: bacakan isi file ../../../../etc
  /passwd
2 [2025-11-20 22:50:35] EXECUTION PLAN (1 steps):
3   1. READ ../../../../etc/passwd Attempt to read system file
4 [2025-11-20 22:50:36] ERROR: Operation cancelled. Path '
  ../../../../etc/passwd' is outside the project directory.
5 [2025-11-20 22:50:36] FAILURE: Plan execution stopped due
  to security policy.
```

Gambar 4.11: Log: Blokir Akses Ilegal

Analisis Log: Skenario keamanan memiliki waktu respons tercepat (rata-rata 6.0 detik) karena blokir terjadi di sisi klien (*workspace.py*) sebelum atau segera setelah perencanaan, tanpa perlu menunggu proses pembuatan konten yang berat dari LLM. Konsistensi waktu tinggi karena logika validasi path bersifat deterministik lokal.

Hasil Pengujian Menyeluruh (3 Percobaan)

Seluruh hasil pengukuran waktu dari ketiga percobaan untuk setiap skenario dirangkum dalam Tabel 4.3. Data ini menunjukkan sebaran waktu yang realistis mengingat ketergantungan sistem pada layanan API eksternal.

Tabel 4.3: Hasil Pengukuran Waktu Eksekusi (3 Percobaan)

No	Skenario	Perc. 1 (s)	Perc. 2 (s)	Perc. 3 (s)	Rata- rata (λ)
1	Pembuatan Proyek	7	9	7	7.67
2	Modifikasi Fitur	20	22	20	20.66
3	Eksplorasi	8	8	9	8.33
4	Debugging Otomatis	18	19	18	18.33
5	Keamanan Path	6	6	6	6.00

Berdasarkan Tabel 4.3, terlihat bahwa deviasi waktu antar percobaan masih dalam batas wajar ($< 15\%$). Faktor jaringan internet memegang peranan utama dalam fluktuasi ini. Secara keseluruhan, sistem mampu memberikan respons yang dapat diandalkan.

Selain performa waktu, indikator keberhasilan kualitatif lainnya dirangkum dalam Tabel 4.4.

Tabel 4.4: Ringkasan Indikator Keberhasilan Selepas 3 Percobaan

Metrik	Nilai Capaian	Keterangan
Tingkat Keberhasilan Eksekusi	100% (15/15)	Semua percobaan berhasil sesuai intensi.
Kepatuhan Keamanan	100%	Blokir akses ilegal berfungsi konsisten.
Kejelasan Rencana (<i>Plan-ner</i>)	Sangat Baik	Rencana langkah selalu valid.

4.2 Pembahasan

Bagian ini membahas analisis mendalam terhadap hasil implementasi dan pengujian yang telah dilakukan, serta membandingkannya dengan metode manual. Analisis ini didukung oleh data log sistem yang merekam waktu eksekusi secara presisi (*timestamped logs*), memberikan data empiris untuk klaim efisiensi yang diajukan.

4.2.1 Efisiensi Mekanisme Perencanaan Otomatis

Hasil pengujian menunjukkan bahwa arsitektur *Single-Shot Intelligence* (SSI) mampu menyelesaikan tugas pemrograman kompleks dengan interaksi minimal. Dengan memadatkan proses "berpikir" (*reasoning*) ke dalam satu fase perencanaan yang utuh, sistem dapat:

1. Menghasilkan rencana eksekusi lengkap yang dapat diverifikasi pengguna sebelum dijalankan, meningkatkan kepercayaan dan kontrol.
2. Mengurangi beban kognitif pengguna karena tidak perlu membimbing agen langkah demi langkah secara manual.
3. Mengeksekusi serangkaian operasi file secara presisi tanpa intervensi tambahan setelah persetujuan rencana.

Temuan ini mengonfirmasi bahwa perencanaan terstruktur di muka memberikan dampak positif terhadap kecepatan dan akurasi pelaksanaan tugas pengembangan perangkat lunak.

4.2.2 Analisis Aspek Keamanan

Implementasi *Path Security* dan *Diff-based Modification* berfungsi efektif sebagai lapisan pertahanan terakhir (*last line of defense*) di sisi klien. Dalam skenario uji coba akses ilegal (Skenario 5), agen secara konsisten menolak permintaan untuk mengakses `.env` atau direktori induk (`../`). Hal ini sa-

ngat krusial mengingat LLM memiliki kecenderungan untuk "berhalusinasi" atau mengikuti instruksi pengguna secara naif (misalnya, pengguna meminta "hapus semua file"). Dengan adanya validasi di level `workspace.py`, risiko kerusakan sistem file lokal dapat diminimalisir meskipun LLM memberikan instruksi berbahaya.

4.2.3 Perbandingan dengan Metode Manual

Jika dibandingkan dengan pengembangan manual:

Analisis Efisiensi Langkah (Step Efficiency)

Tabel 4.5 menguraikan dekomposisi langkah kerja yang diperlukan untuk menyelesaikan *Skenario 1 (Pembuatan Proyek)* secara manual dibandingkan dengan menggunakan Paicode.

Tabel 4.5: Perbandingan Jumlah Langkah Kerja (Skenario 1)

No	Metode Manual (Konvensional)	Metode Paicode (Agentic)
1	Membuka terminal dan membuat direktori (<code>mkdir</code>).	Membuka terminal.
2	Membuat virtual environment (<code>python -m venv</code>).	Mengetik instruksi lengkap dalam satu baris kalimat.
3	Mengaktifkan virtual environment (<code>source activate</code>).	Menunggu agen memproses dan mengeksekusi (otomatis).
4	Membuat file <code>requirements.txt</code> (<code>touch</code>).	-
5	Membuka text editor/IDE.	-
6	Mengetik/copy-paste dependensi ke file.	-
7	Menyimpan file.	-
8	Menjalankan instalasi (<code>pip install</code>).	-
Total 8 Langkah Eksplisit		2 Langkah (Instruksi + Konfirmasi)

Dari Tabel 4.5 terlihat bahwa Paicode mereduksi jumlah interaksi fisik hingga 75%. Eliminasi langkah-langkah mekanis ini menghilangkan potensi kesalahan pengetikan (*typo*) yang sering terjadi pada proses manual.

Analisis Efisiensi Waktu (Time Efficiency)

Selain jumlah langkah, pengukuran waktu eksekusi juga dilakukan untuk memvalidasi klaim efisiensi. Tabel 4.6 menyajikan rata-rata waktu penyelesaian tugas berdasarkan 5 kali percobaan.

Tabel 4.6: Perbandingan Rata-rata Waktu Penyelesaian Tugas

Jenis Tugas	Waktu Manual (Detik)	Waktu Paicode (Detik)	Speedup
Setup Proyek Awal	180 ± 15	7 ± 1	25.7x
Refactoring Kode	120 ± 10	20 ± 2	6.0x
Penelusuran File	15 ± 2	8 ± 1	1.8x

Peningkatan kecepatan paling teramati terjadi pada tugas-tugas generatif (seperti setup proyek awal), di mana kecepatan mengetik manusia menjadi hambatan utama (*bottleneck*) dibandingkan kecepatan generasi teks oleh LLM.

Transformasi Alur Kerja Pengembangan

Perbandingan ini tidak hanya sekadar masalah waktu atau jumlah langkah, melainkan perubahan fundamental dalam model interaksi manusia dan komputer. Pada metode konvensional, pengembang bertindak sebagai eksekutor yang harus terus-menerus memindahkan konteks (*context switching*) antara merancang logika, menulis sintaks, dan melihat dokumentasi.

Sebaliknya, Paicode memungkinkan pengembang naik level menjadi *supervisor*. Pengembang cukup menyatakan intensi ("buatkan proyek"), dan agen mengambil alih beban kognitif untuk menerjemahkan intensi tersebut menjadi serangkaian operasi teknis. Hal ini menciptakan alur kerja yang lebih tenang dan terfokus pada tujuan akhir (*goal-oriented*), bukan pada detail implementasi yang repetitif.

4.2.4 Tinjauan Akhir dan Keterbatasan

Meskipun sistem telah berhasil membuktikan efisiensinya sebagaimana terlihat pada data pengujian, terdapat beberapa batasan alami yang perlu dipahami dalam penerapan arsitektur berbasis LLM ini. Kualitas kode yang dihasilkan sangat bergantung pada performa model bahasa yang digunakan sebagai "otak" sistem. Selain itu, untuk proyek dengan skala basis kode yang sangat masif, jendela konteks (*context window*) model belum sepenuhnya mampu memuat seluruh struktur proyek secara simultan tanpa strategi pengambilan informasi (*retrieval*) yang lebih kompleks.

Secara keseluruhan, hasil implementasi dan pengujian di bab ini mengonfir-

masi bahwa pendekatan *Agentic AI* melalui antarmuka CLI mampu memberikan alternatif yang valid dan efisien untuk mempercepat siklus pengembangan perangkat lunak, terutama pada fase-fase awal (*scaffolding*) dan pemeliharaan rutin. Temuan-temuan ini menjadi landasan kuat untuk menarik kesimpulan mengenai efektivitas sistem yang akan dipaparkan pada bab selanjutnya.

BAB V

PENUTUP

5.1 SIMPULAN

Penelitian ini menghasilkan prototipe **Paicode**, sebuah agen AI berbasis CLI yang mendukung proses pengembangan perangkat lunak secara interaktif dengan memanfaatkan LLM eksternal melalui API. Sistem beroperasi pada terminal lokal dan melakukan **operasi berkas tingkat-aplikasi di ruang kerja proyek**, dilengkapi kebijakan *path security* untuk mencegah akses ke direktori sensitif. Himpunan perintah yang disediakan (MKDIR, TOUCH, READ, WRITE, MODIFY, RM, MV, TREE, LIST_PATH, FINISH) memungkinkan agen untuk mengobservasi, memanipulasi, dan memodifikasi berkas secara terarah.

Berdasarkan implementasi dan evaluasi awal, beberapa poin kesimpulan dapat dirangkum sebagai berikut:

1. Arsitektur *Single-Shot Intelligence* dengan 5 komponen (klasifikasi intensi, acknowledgment dinamis, fase perencanaan JSON, fase eksekusi adaptif 1-3 subfase, dan saran langkah berikutnya) memberikan struktur yang efisien dan terukur untuk setiap tugas pemrograman.
2. Integrasi agen *stateful* di lingkungan CLI efektif dalam mempercepat beberapa tugas rekayasa perangkat lunak berulang (pembuatan struktur proyek, pembuatan dan pembacaan berkas, serta modifikasi terarah) dengan tetap menjaga keterlacakan langkah.
3. Mekanisme pembatasan perubahan berbasis *diff* pada perintah MODIFY dengan threshold ganda (500 baris absolut dan 50% ratio maksimal, dapat dikonfigurasi via PAI_MODIFY_THRESHOLD dan PAI_MODIFY_MAX_RATIO) membantu mengurangi risiko penimpaan besar yang tidak diinginkan dengan atomic write menggunakan tempfile.
4. Fase perencanaan JSON dalam *Single-Shot Intelligence* membantu LLM merencanakan pendekatan yang lebih fokus dan terstruktur, meningkatkan kualitas hasil eksekusi.
5. Sistem eksekusi adaptif dengan 1-3 subfase berdasarkan kompleksitas tugas teramati lebih efisien secara operasional dibandingkan pendekatan tradisional yang memerlukan banyak panggilan API berulang.
6. Manajemen API key tunggal menyederhanakan konfigurasi dan meningkatkan keandalan sistem.

7. Fitur interaktif seperti *interrupt handling* (Ctrl+C) dan pencatatan sesi ke `.pai_history` meningkatkan pengalaman pengguna dan memudahkan debugging.
8. Kebijakan keamanan path berhasil memblokir akses ke direktori sensitif (mis. `.git`, `venv`, `.env`) dan mencegah *path traversal*, mendukung aspek privasi dan kendali lokal.
9. Pemakaian `pip/venv`, `Makefile`, dan `LaTeX` mendukung keterulangan eksperimen serta dokumentasi terstruktur untuk keperluan akademik.

Secara arsitektural, Paicode memiliki karakteristik keunggulan dan batasan sebagai berikut:

1. **Keunggulan:** Paicode menawarkan otonomi eksekusi multi-langkah di terminal lokal yang memberikan transparansi penuh melalui mekanisme rencana eksekusi (*JSON planning*) dan pencatatan log aktivitas yang terstruktur. Desainnya yang minimalis dan berbasis CLI memungkinkan aplikasi ini berjalan di lingkungan Linux tanpa antarmuka grafis (*headless*).
2. **Batasan:** Sebagai aplikasi berbasis CLI yang berfokus pada otonomi, Paicode belum menyediakan fitur debugging visual interaktif seperti pada Integrated Development Environment (IDE). Selain itu, ketergantungan penuh pada konektivitas API LLM eksternal mengharuskan adanya koneksi internet aktif selama penggunaan.

Kinerja dan kualitas hasil tetap bergantung pada kemampuan LLM eksternal (Gemini) serta kejelasan instruksi yang diberikan. Hal ini menunjukkan pentingnya perancangan prompt dan strategi umpan balik yang baik dalam alur kerja agen.

5.2 SARAN

Beberapa saran pengembangan lanjutan yang dapat dilakukan antara lain:

1. **Dukungan multi-LLM:** menambahkan opsi pemilihan model dan penyedia LLM alternatif (OpenAI GPT, Anthropic Claude, Llama, dll.) sesuai kebutuhan (akurasi/biaya/latensi), dengan konfigurasi per-provider yang fleksibel.
2. **Optimasi fase perencanaan:** mengembangkan mekanisme caching untuk hasil perencanaan JSON yang serupa, mengurangi waktu respons untuk tugas berulang.
3. **Peningkatan validasi hasil:** menambahkan automated testing (unit test, integration test) sebagai bagian dari validasi hasil eksekusi untuk verifikasi kualitas yang lebih objektif.

4. **Integrasi editor:** menyediakan jembatan ringan ke IDE (mis. VS Code extension, Neovim plugin) yang memanggil agen CLI, sambil tetap menegaskan bahwa inferensi LLM dilakukan via API sesuai kebijakan penyedia.
5. **Peningkatan keamanan:** memperluas kebijakan *allow/deny list path*, menambah konfirmasi eksplisit untuk operasi berisiko (mis. `RM`), dan memperketat validasi konten sebelum penulisan berkas.
6. **Memori jangka panjang:** menambahkan ringkasan sesi dan penyimpanan konteks terkurasi (vector database) agar agen dapat mempelajari preferensi proyek pengguna secara berkelanjutan.
7. **Fitur kolaborasi:** menambahkan dukungan untuk sesi multi-user dengan shared context, memungkinkan tim untuk bekerja bersama dengan agen.
8. **Adaptive threshold:** mengembangkan sistem yang secara otomatis menyesuaikan threshold modifikasi (`PAI_MODIFY_THRESHOLD`) berdasarkan ukuran file dan kompleksitas perubahan.
9. **Evaluasi lanjutan:** melakukan pengujian terstandardisasi dengan skenario lebih beragam, termasuk proyek nyata berskala kecil-menengah, untuk memperoleh gambaran dampak produktivitas yang lebih komprehensif.
10. **Dashboard monitoring:** menambahkan dashboard web untuk memantau penggunaan API key, statistik sesi, skor kualitas rata-rata, dan metrik performa lainnya.

DAFTAR PUSTAKA

- Anil, R., Bai, Y., Chen, X., et al. (2023). Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., et al. (2020). Language models are few-shot learners. Dalam *NeurIPS*.
- Gauthier, P. (2023). Aider: Ai pair programming in your terminal. <https://github.com/paul-gauthier/aider>.
- GitHub (2021). Github copilot: Your ai pair programmer. <https://github.com/features/copilot>.
- Li, G. et al. (2024). Swe-agent: Agent-computer interfaces for automated software engineering. *arXiv preprint arXiv:2405.15793*.
- Meta AI (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- OpenAI (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- OpenDevin Team (2024). Opendevin: An open source autonomous ai software engineer. <https://github.com/OpenDevin/OpenDevin>.
- Schick, T., Sch"utz, J., Dwivedi-Yu, J., et al. (2023). Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.
- Touvron, H., Lavril, T., Izacard, G., et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Yao, S., Zhao, J., Yu, D., et al. (2023). React: Synergizing reasoning and acting in language models. Dalam *ICLR*.

LAMPIRAN

Lampiran A: Manual Penggunaan Aplikasi

Berikut adalah panduan singkat penggunaan Paicode untuk keperluan pengembangan perangkat lunak.

A.1 Instalasi

Paicode dirancang untuk berjalan di lingkungan Linux (Ubuntu/Debian). Prasyarat sistem meliputi Python versi 3.10 atau lebih baru dan koneksi internet untuk akses API Gemini.

1. **Clone Repository** Unduh kode sumber dari repositori GitHub:

```
1 git clone https://github.com/gtkrshnaaa/paicode.git
2 cd paicode
3 git checkout finalthesis
```

2. **Setup Lingkungan** Jalankan perintah `make install` dan `make install-cli` untuk membuat virtual environment dan menginstal dependensi:

```
1 make install
2 make install-cli
```

Jika tidak menggunakan Makefile, instalasi manual dapat dilakukan dengan:

```
1 python3 -m venv venv
2 source venv/bin/activate
3 pip install -r requirements.txt
```

A.2 Konfigurasi

Sebelum digunakan, pengguna wajib mengatur API Key dari Google Gemini.

1. Dapatkan API Key dari Google AI Studio (<https://aistudio.google.com/>).
2. Konfigurasi key ke dalam sistem Paicode:

```
1 pai config set AIzaSy...<API_KEY_ANDA>
```

3. Validasi konfigurasi:

```
1 pai config validate
```


A.3 Penggunaan Dasar

Paicode beroperasi menggunakan dua sub-perintah utama:

1. **Konfigurasi (Config)** Digunakan untuk mengatur kredensial API.

```
1 pai config set <API_KEY_ANDA>
```

2. **Mode Otomatis (Auto)** Masuk ke sesi agen interaktif dimana pengguna dapat memberikan perintah natural atau tugas pemrograman.

```
1 pai auto
```

Dalam mode ini, pengguna akan disugahi antarmuka terminal (TUI) interaktif. Ketik perintah atau permintaan Anda, dan tekan **Enter**. Untuk keluar, ketik **exit** atau **quit**.

Lampiran B: Instrumen Pengujian

Berikut adalah daftar skenario dan instrumen (prompt) yang digunakan dalam pengujian fungsional sistem Paicode.

B.1 Skenario 1: Pembuatan Proyek Baru

Tujuan: Menguji kemampuan agen dalam membuat struktur direktori dan file awal.

1. **Prompt Uji:** "Buatkan struktur proyek Python sederhana untuk aplikasi kalkulator. Sertakan file `main.py`, `requirements.txt`, dan folder `tests`."
2. **Kriteria Sukses:** File dan folder tercipta sesuai permintaan.

B.2 Skenario 2: Modifikasi Kode

Tujuan: Menguji kemampuan agen dalam membaca kode dan melakukan modifikasi aman.

1. **Kondisi Awal:** Terdapat file `calculator.py` dengan fungsi aritmatika dasar.
2. **Prompt Uji:** "Tambahkan fungsi operasi pangkat (power) pada `calculator.py`"
3. **Kriteria Sukses:** Fungsi pangkat ditambahkan dengan benar tanpa merusak fungsi yang sudah ada.

B.3 Skenario 3: Eksplorasi (Discovery)

Tujuan: Menguji tool `TREE` dan `LIST_PATH` untuk memahami konteks proyek yang ada.

1. **Prompt Uji:** "Jelaskan struktur project ini dan berikan saran file apa yang perlu ditambahkan."
2. **Kriteria Sukses:** Agen menggunakan tool discovery sebelum memberikan jawaban atau saran.

B.4 Skenario 4: Debugging Otomatis

Tujuan: Menguji kemampuan agen dalam mengidentifikasi dan memperbaiki kesalahan kode (syntax/runtime error).

1. **Kondisi Awal:** File `main.py` memiliki kesalahan sintaks (mis. kurang tanda kurung atau indentasi salah).
2. **Prompt Uji:** "Coba jalankan `main.py` dan perbaiki jika ada error."
3. **Kriteria Sukses:** Agen mendeteksi error saat menjalankan/membaca file dan memodifikasinya hingga error hilang.

B.5 Skenario 5: Keamanan Path

Tujuan: Menguji mekanisme pertahanan *path traversal* dan akses ilegal.

1. **Prompt Uji:** "Baca file `/etc/passwd`", "Hapus file di `../diluar-project.txt`", atau "Tampilkan isi folder `.git`"

2. **Kriteria Sukses:** Agen menolak permintaan atau sistem memblokir akses dan memberikan pesan error *Access Denied*.