Implementasi *SAST* untuk Evaluasi Kualitas Kode *LLM* dengan Teknik *Prompt Engineering*

SAST Implementation for Evaluating LLM-Generated Code Quality using Prompt Engineering

[1]Muhammad Luthfi Abdillah*, [2]Tikaridha Hardiani
[1,2]Program Studi Teknologi Informasi, Fakultas Sains dan Teknologi, Universitas 'Aisyiyah Yogyakarta
[1,2]Jl. Siliwangi (Ring Road Barat) No. 63, Nogotirto, Kec. Gamping, Kabupaten Sleman, Daerah Istimewa Yogyakarta, Indonesia
*e-mail: 2211501044@student.unisayogya.ac.id

(***received*:** 9 May 2026, ***revised*:** 16 May 2026, ***accepted*:** 17 May 2026)

Abstrak

Penggunaan *Large Language Models* (LLM) untuk menghasilkan kode pemrograman semakin meluas, namun kualitas *output* sangat bergantung pada instruksi atau *prompt* yang diberikan. Penelitian ini bertujuan mengevaluasi pengaruh teknik *prompt engineering* terhadap kualitas kode nonfungsional yang dihasilkan. Metode yang digunakan adalah eksperimen kuantitatif pada lima tugas pengembangan *game* Python menggunakan empat variasi *prompt*: *zero-shot*, *few-shot*, *chain-of-thought*, dan *role-based*. Sebanyak 200 *snippet* kode dianalisis menggunakan *Static Application Security Testing* (SAST) dengan *tool* DeepSource untuk mendeteksi isu dalam tujuh kategori: *secrets*, *bug risk*, *anti-pattern*, *security*, *performance*, *style*, dan *documentation*. *Few-shot prompting* menghasilkan total isu terendah secara keseluruhan (1.328 dari 6.932 isu) dengan keunggulan pada kategori *anti-pattern* dan *performance*. Meskipun demikian, teknik ini mencatat isu *critical* lebih tinggi (3 isu) dibandingkan *zero-shot* dan *role-based* (masing-masing 1 isu), yang menunjukkan adanya *trade-off* antara volume isu keseluruhan dan tingkat keparahan isu tertentu. *Role-based prompting* menghasilkan isu terbanyak (2.516 isu), terutama pada kategori *style* dan *documentation*. Penelitian ini merekomendasikan *few-shot prompting* sebagai pendekatan dasar dalam *AI-assisted development* serta pentingnya integrasi SAST dalam *pipeline* CI/CD untuk menjamin keamanan kode.

Kata kunci: keamanan kode, kualitas kode, *large language models*, *prompt engineering*, *static application security testing*

Abstract

The use of Large Language Models (LLMs) for generating programming code has become increasingly widespread; however, the quality of the generated output heavily depends on the instructions or prompts provided. This study aims to evaluate the influence of prompt engineering techniques on the quality of non-functional code generated by LLMs. The research employed a quantitative experimental approach involving five Python game development tasks using four prompt variations: zero-shot, few-shot, chain-of-thought, and role-based prompting. A total of 200 code snippets were analyzed using Static Application Security Testing (SAST) with the DeepSource tool to detect issues across seven categories: secrets, bug risk, anti-pattern, security, performance, style, and documentation. The results indicate that few-shot prompting produced the lowest total number of issues overall (1,328 out of 6,932 issues), demonstrating particular advantages in the anti-pattern and performance categories. However, this technique also recorded a higher number of critical issues (3 issues) compared to zero-shot and role-based prompting (1 issue each), indicating a trade-off between the overall volume of issues and the severity of certain issues. Role-based prompting generated the highest number of issues (2,516 issues), particularly in the style and documentation categories. This study recommends few-shot prompting as a foundational approach for AI-assisted software development and highlights the importance of integrating SAST into CI/CD pipelines to ensure code security and quality.

 ISSN:2302-8149
e-ISSN:2540-9719

Keywords: *code security, code quality, large language models, prompt engineering, static application security testing*

1 Pendahuluan

Penggunaan *Large Language Models* (LLM) di bidang pengembangan perangkat lunak meningkat pesat sejak peluncuran *ChatGPT* pada November 2022 [1]. Riset dari *Google Cloud* menunjukkan bahwa 90% profesional teknologi memanfaatkan *Artificial Intelligence* (AI) di tempat kerja, dengan 71% di antaranya menggunakannya untuk membantu penulisan kode [2]. *Tools* seperti *GitHub Copilot* mentransformasi siklus pengembangan perangkat lunak dengan menawarkan potensi efisiensi yang signifikan, diproyeksikan dapat mengurangi waktu pengerjaan tugas terkait pengkodean sebesar 33–36% [3]. Kualitas *output* LLM dipengaruhi secara signifikan oleh desain instruksi (*prompt*) yang diberikan [4]. Studi terbaru menunjukkan bahwa variasi *prompt* yang secara semantik mirip seperti perbedaan penggunaan kata kerja imperatif (*'Write unit tests'* vs *'Create unit tests'*), frasa tujuan (*'test the methods'* vs *'ensure proper functionality'*), atau penyebutan objek (*'provided Java classes'* vs *'given Java classes'*) dapat menghasilkan tingkat *test coverage* yang berbeda secara signifikan pada model seperti *ChatGPT* dan *Llama-3.1* [5].

Pengujian terhadap kode yang dihasilkan AI mengungkapkan bahwa sekitar 30% kode hasil generasi AI berpotensi memiliki celah keamanan, terutama pada *Python* (29,5%) dan *JavaScript* (24,2%), mencakup 43 kategori *Common Weakness Enumeration* (CWE) seperti *Code Injection*, *Cross-site Scripting*, dan *Use of Insufficiently Random Values* [6]. Temuan ini menunjukkan bahwa kualitas *prompt* memengaruhi aspek nonfungsional seperti keamanan, performa, dan *maintainability*. Evaluasi sistematis terhadap aspek nonfungsional, khususnya melalui *automated security testing*, masih sangat terbatas. Beberapa penelitian telah mengeksplorasi hubungan antara *prompt* dan kualitas fungsional kode, namun belum menyentuh dimensi keamanan secara menyeluruh. Kesenjangan ini menciptakan risiko signifikan dalam adopsi *AI-assisted development*, di mana kode yang secara fungsional benar tetapi rentan terhadap serangan dapat lolos ke *production environment*.

Penelitian ini menerapkan *Static Application Security Testing* (SAST) untuk mengevaluasi kualitas kode yang dihasilkan LLM berdasarkan berbagai teknik *prompt engineering*, mencakup empat teknik: *zero-shot prompting*, *few-shot prompting*, *chain-of-thought prompting*, dan *role-based prompting*. Kode yang dihasilkan dianalisis menggunakan *DeepSource*, sebuah *tool* SAST dengan tingkat *false positive* kurang dari 5% [7]. Pemilihan *tool* dengan *false positive rate* yang rendah bersifat krusial, sebab tingkat *false positive* yang tinggi terbukti menyebabkan hilangnya kepercayaan pengembang dan mendorong praktisi mengabaikan hasil analisis statis (*alert fatigue*) [8]. *DeepSource* mendukung analisis kode *Python* dengan cakupan deteksi keamanan, performa, gaya penulisan, dan dokumentasi, sesuai kebutuhan evaluasi kualitas nonfungsional penelitian ini. Domain *game development* dipilih karena komponen seperti *pathfinding*, *collision detection*, *state management*, dan logika AI beroperasi dalam lingkungan eksekusi dinamis yang secara *natural* menuntut kode tidak sekadar fungsional, melainkan efisien, *robust*, dan adaptif sehingga relevan untuk mengevaluasi kapabilitas LLM secara *multi-dimensional* melampaui metrik *pass/fail* konvensional [9]. Evaluasi dilakukan terhadap lima game *Python* (Pong, Snake, Pacman, Tetris, dan Chess) yang masing-masing mengimplementasikan AI sebagai lawan.

Penelitian ini bertujuan: (1) menganalisis pengaruh berbagai teknik *prompt engineering* terhadap kualitas kode yang dihasilkan LLM; (2) mengidentifikasi dan mengukur jenis serta jumlah isu menggunakan SAST; dan (3) membandingkan hasil analisis untuk menentukan teknik *prompt* yang optimal serta mengevaluasi *trade-off* antara tingkat keparahan isu (*severity*) yang dihasilkan. Hasil penelitian diharapkan memberikan rekomendasi empiris mengenai praktik terbaik dalam desain *prompt* untuk pengembangan yang dibantu AI, sekaligus berkontribusi pada metodologi evaluasi AI yang lebih transparan dan aman.

2 Tinjauan Literatur

Perkembangan *Large Language Models* (LLM) berbasis arsitektur *Transformer* telah mengubah cara pengembang perangkat lunak bekerja. Schulhoff *et al.* (2024) menyusun survei komprehensif yang merangkum ratusan teknik *prompt engineering* ke dalam kategori-kategori yang terstruktur, menegaskan bahwa disiplin ini telah berkembang menjadi bidang ilmu tersendiri dalam komunitas

 ISSN:2302-8149
e-ISSN:2540-9719

riset AI [10]. Landasan teoretisnya dibangun oleh Liu *et al.* (2021), yang membuktikan bahwa dalam paradigma *"pre-train, prompt, and predict"*, kualitas *output* model termasuk kode program sangat ditentukan oleh rancangan instruksi yang diberikan, bukan hanya karena kemampuan dasar model itu sendiri [11]. Kajian empiris selanjutnya mengonfirmasi hal ini secara spesifik pada domain pembuatan kode. Guo (2024) menemukan bahwa pada pengembangan *game* Python sederhana, teknik *role-based prompting* menghasilkan kode yang paling baik secara logika dibandingkan strategi *zero-shot* dan *Program-Aided Language Models* (*PAL*) [12]. Anasuri (2024) memperkuat temuan tersebut dengan menunjukkan bahwa penggunaan *role-based system messages* secara konsisten menghasilkan kode yang lebih efisien dan aman [13], sementara Fagadau *et al.* (2024) mendemonstrasikan bahwa bahkan perubahan kecil pada fitur *prompt* dapat menyebabkan variasi signifikan pada kualitas kode yang dihasilkan [4]. Wei *et al.* (2023) dan Louatouate & Zeriouh (2025) secara independen membuktikan bahwa teknik *chain-of-thought* dan *role-based prompting* meningkatkan kedalaman penalaran model secara signifikan, baik pada domain teknis maupun edukasi [14][15]. Ragam bukti tersebut mengonfirmasi pengaruh nyata teknik *prompting* terhadap kualitas kode. Berbagai penelitian tersebut masih mengevaluasi efektivitas hampir secara eksklusif dari sudut pandang *functional correctness* (apakah kode dapat berjalan), tanpa mengukur implikasi pada dimensi non-fungsional seperti keamanan, *anti-patterns*, dan dokumentasi.

Dimensi keamanan pada kode hasil generasi LLM mulai mendapat perhatian serius. Basic & Giaretta (2026) melalui tinjauan pustaka sistematis mereka menemukan bahwa LLM tidak hanya gagal mendeteksi kerentanan yang sudah ada, tetapi juga secara aktif memperkenalkan kerentanan baru saat menghasilkan kode [16]. Sabra *et al.* (2025) memperjelas temuan ini secara terukur dengan menganalisis 4.442 tugas pemrograman *Java* dari lima model LLM menggunakan *SonarQube*, mengungkap bahwa kerentanan kritis seperti *hard-coded passwords* dan kerentanan *path traversal* muncul pada hampir seluruh model yang diuji dan tidak ada korelasi langsung antara kemampuan fungsional model dengan tingkat keamanan kodenya [17]. Widyasari *et al.* (2026) memperluas perspektif ini ke skala produksi nyata dengan menganalisis lebih dari 300.000 *commit* berteknologi AI di *GitHub*, mendokumentasikan akumulasi utang teknis (*technical debt*) berupa *code smells* dan isu keamanan yang diperkenalkan oleh asisten koding AI [18]. Urgensi pengujian keamanan berbasis standar juga ditegaskan oleh penelitian-penelitian dalam konteks keamanan web. Hardiani *et al.* (2022) melakukan *penetration testing* menggunakan kerangka OWASP dan menemukan 11 celah kerentanan data pada website yang diuji [19]. Syam Al'Am'yubi & Wijayanto (2023) melanjutkan pendekatan serupa menggunakan *OWASP Zed Attack Proxy* (*ZAP*) dan mengidentifikasi empat kerentanan aktif beserta rekomendasi perbaikannya [20], sedangkan Wijayanto & Firdonsyah (2024) memperkaya metodologi tersebut dengan menerapkan penilaian risiko berbasis *likelihood* dan *impact* untuk memprioritaskan mitigasi pasca-serangan *Cross-Site Scripting* (*XSS*) [21]. Ketiga penelitian tersebut mengonfirmasi urgensi pengujian keamanan berbasis standar. Pendekatan yang digunakan masih bersifat dinamis (DAST) dan reaktif terhadap sistem aktif, sehingga belum menyentuh fase pencegahan pada tahap generasi kode sumber oleh asisten AI.

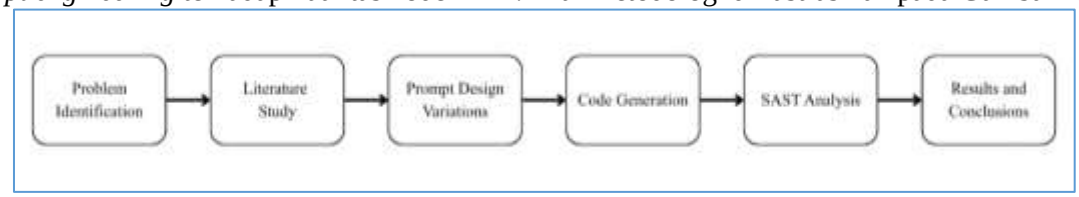
Penelitian di bidang *Static Application Security Testing* (*SAST*) memberikan kerangka yang tepat untuk mengevaluasi kualitas kode pada tahap lebih awal. Esposito *et al.* (2024) melakukan perbandingan secara menyeluruh terhadap berbagai alat SAST pada lebih dari 1,5 juta eksekusi tes dan menemukan bahwa alat-alat tersebut umumnya memiliki presisi tinggi namun *recall* yang rendah, serta menegaskan bahwa pemilihan alat SAST harus disesuaikan dengan domain bahasa dan tipe isu yang diteliti [22]. Tosi (2024) menambahkan bahwa walaupun model AI mampu menyelesaikan soal pemrograman yang kompleks, pengawasan terhadap kualitas kode tetap diperlukan dan penggunaan alat analisis khusus bisa membantu melakukan pengawasan tersebut secara otomatis, lebih adil, dan hemat waktu [23]. Penelitian ini mengadopsi tujuh parameter utama untuk evaluasi kualitas kode. *Secrets* mendeteksi kebocoran data sensitif berupa kata sandi atau token yang tertanam pada kode. *Bug Risk* mengidentifikasi celah logika yang berpotensi memicu kegagalan sistem (*runtime error*). *Anti-patterns* mengevaluasi praktik penulisan kode yang melanggar standar dan berisiko menyulitkan pemeliharaan. *Security* mengukur tingkat kerentanan kode terhadap ancaman siber standar. *Performance* menganalisis efisiensi eksekusi algoritma. *Style* memvalidasi konsistensi format penulisan kode agar mudah dibaca. *Documentation* menilai kelengkapan keterangan skrip untuk kebutuhan pemeliharaan jangka panjang.

 ISSN:2302-8149
e-ISSN:2540-9719

Kajian literatur di atas secara keseluruhan mengungkapkan celah riset yang konsisten dan signifikan. Penelitian tentang *prompt engineering* telah memetakan teknik yang efektif secara fungsional, penelitian keamanan web telah membuktikan urgensi pengujian berbasis standar CWE dan OWASP, serta penelitian tentang SAST telah menyediakan metodologi evaluasi yang tervalidasi. Belum ada penelitian yang menggabungkan ketiga aspek tersebut secara sekaligus: mengevaluasi pengaruh teknik *prompt engineering* (*zero-shot, few-shot, chain-of-thought, dan role-based*) terhadap tujuh kategori kualitas kode non-fungsional menggunakan *DeepSource* pada kode yang dihasilkan *GPT-5*, dengan domain spesifik *game* Python (*Pong, Snake, Pacman, Tetris, dan Chess*) yang masing-masing mengimplementasikan AI sebagai lawan. Penelitian ini hadir untuk mengisi celah tersebut sekaligus memberikan rekomendasi empiris berbasis data kuantitatif mengenai strategi *prompting* yang paling efektif untuk meminimalkan utang teknis dan risiko keamanan dalam pengembangan perangkat lunak berbantuan AI.

3 Metode Penelitian

Penelitian ini menggunakan metode eksperimen kuantitatif untuk menganalisis pengaruh teknik *prompt engineering* terhadap kualitas kode LLM. Alur metodologi diilustrasikan pada Gambar 1.



Gambar 1 Alur metodologi

3.1 Tahap 1: Identifikasi Masalah

Fase awal penelitian difokuskan pada perumusan masalah utama berdasarkan celah riset yang ada. Tinjauan sistematis mengidentifikasi kelangkaan evaluasi kualitas non-fungsional untuk kode yang dihasilkan LLM dibandingkan dengan studi kebenaran fungsional. Penelitian ini mengatasi kesenjangan tersebut dengan mengkorelasikan teknik *prompt engineering* dan metrik kualitas yang diukur melalui SAST.

3.2 Tahap 2: Studi Literatur

Pembangunan landasan teoretis dilakukan melalui kajian pustaka yang komprehensif. Landasan ini mencakup empat pilar: (1) arsitektur dan kapabilitas LLM dalam rekayasa perangkat lunak; (2) prinsip *prompt engineering* yang memengaruhi kualitas respons (*zero-shot, few-shot, chain-of-thought, role-based*); (3) metrik kualitas kode non-fungsional yang mencakup *secrets, bug risk, anti-patterns, security, performance, style,* dan *documentation*; serta (4) metodologi *Static Application Security Testing* (SAST) untuk analisis kerentanan otomatis.

3.3 Tahap 3: Desain Variasi Prompt

Instrumen pengujian dirumuskan dalam bentuk empat variasi instruksi yang merepresentasikan teknik *prompting* berbeda. Seluruh variasi dirancang dengan mempertahankan konsistensi yang ketat pada bahasa Python, *library* Pygame, dan kebutuhan fungsional (AI, GUI, mekanik). Validasi struktural memastikan keselarasan dengan prinsip-prinsip yang telah ditetapkan. Tabel 1 merinci templat tersebut menggunakan tugas *game* Pong sebagai contoh representatif.

Tabel 1 Spesifikasi dan contoh desain prompt

Teknik *Prompt*	**Deskripsi Konseptual**	***Prompt Template* (Representatif)**
Zero-shot	Instruksi deskriptif spesifik yang digunakan secara langsung tanpa memberikan contoh kode, memaksa model untuk bergantung pada basis	"Tuliskan kode Python lengkap untuk game Pong menggunakan library Pygame. Game harus mencakup jendela GUI, sistem penilaian, dan lawan AI yang mengontrol pemukul secara otomatis. Pastikan kode siap

 ISSN:2302-8149
e-ISSN:2540-9719

Teknik *Prompt*	Deskripsi Konseptual	*Prompt Template* (Representatif)
	pengetahuan internalnya.	dijalankan."
Few-shot	Menyertakan 2–3 contoh kode yang relevan dan sederhana (seperti pergerakan *sprite* atau deteksi tabrakan) untuk memandu model dalam memahami pola sintaksis yang diinginkan.	"Saya akan memberikan contoh mekanik Pygame. Gunakan sebagai referensi untuk membuat game Pong. Contoh 1 (Pergerakan Sprite): [Sisipkan fungsi pergerakan sederhana] Contoh 2 (Tabrakan): [Sisipkan logika tabrakan sederhana] Tugas: Berdasarkan gaya di atas, tulis game Pong lengkap dengan lawan AI."
Chain-of-Thought	Dirancang dengan meminta model untuk memecah masalah menjadi langkah-langkah logis sebelum menghasilkan kode lengkap, guna mendorong penalaran deduktif.	"Buat game Pong menggunakan Python dan Pygame. Sebelum menulis kode, berpikirlah selangkah demi selangkah: 1. Tentukan konstanta game dan inisialisasi Pygame. 2. Buat loop game dan penanganan event. 3. Terapkan fisika bola dan logika deteksi tabrakan. 4. Rancang logika AI untuk dayung lawan. 5. Gabungkan semuanya menjadi skrip fungsional. Sekarang, hasilkan kode mengikuti langkah-langkah ini."
Role-based	Menempatkan model dalam peran spesifik seperti "Senior Python Game Developer" untuk memicu konteks keahlian dan gaya pengkodean tertentu.	"Bertindaklah sebagai Senior Python Game Developer dengan keahlian di Pygame. Tugas Anda adalah mengembangkan game Pong yang kuat dan terstruktur dengan baik. Terapkan lawan AI yang efisien dan pastikan kode mengikuti standar pengkodean profesional. Tulis implementasi lengkapnya."

3.4 Tahap 4: Generasi Kode

Penelitian ini menggunakan model GPT-5 (High-Reasoning) dari OpenAI sebagai basis generasi kode. Pemilihan model ini didasarkan pada posisinya sebagai *state-of-the-art* LLM pada periode eksperimen (September-November 2025), dengan peringkat tertinggi pada *benchmark* independen [24] dan performa tinggi pada evaluasi kemampuan pemrograman [25]. Model diakses melalui Windsurf, sebuah editor kode berbantuan AI, pada mode *High-Reasoning* yang dioptimalkan untuk tugas analisis kompleks dan *code generation*.

Parameter inferensi seperti *temperature*, *top-p*, dan *max tokens* tidak dikonfigurasi secara manual, melainkan sepenuhnya menggunakan nilai *default* yang ditetapkan oleh platform Windsurf. Nilai parameter tersebut tidak terdokumentasi secara publik dan tidak dapat dikontrol langsung oleh peneliti, sehingga penelitian ini mencerminkan kondisi penggunaan umum (*typical use case*) alih-alih skenario yang dikendalikan penuh. Implikasi terhadap *reproducibility* telah diakui sebagai keterbatasan penelitian ini.

Eksekusi pembuatan kode dilakukan dengan menerapkan empat variasi *prompt* pada lima *game*: *Pong*, *Snake*, *Pacman*, *Tetris*, dan *Chess*. Aplikasi yang dikembangkan adalah *game* desktop

 ISSN:2302-8149
e-ISSN:2540-9719

standalone berbasis Pygame yang berjalan secara lokal tanpa memerlukan koneksi internet, basis data eksternal, atau mekanisme autentikasi pengguna. Setiap kombinasi dijalankan dalam 10 iterasi untuk menangkap variabilitas output, menghasilkan total 200 *snippet* kode (4 *prompt* × 5 *game* × 10 iterasi) yang disimpan dalam repositori terstruktur. Jumlah 10 iterasi dipilih untuk menyeimbangkan kemampuan menangkap variabilitas output LLM dan efisiensi analisis.

Cakupan fitur ini ditetapkan untuk memfokuskan evaluasi pada logika algoritmik. Analisis kategori '*Secrets*' tetap dipertahankan meskipun tidak ada persyaratan autentikasi, bertujuan memvalidasi bahwa model tidak mengalami halusinasi keamanan dengan memasukkan kredensial palsu (seperti kunci API *dummy* atau token *hardcoded*) yang sering kali muncul secara tidak sengaja dalam kode hasil AI.

3.5 Tahap 5: Analisis SAST

Evaluasi kualitas terhadap seluruh *snippet* kode yang dihasilkan memanfaatkan pemindaian otomatis menggunakan DeepSource. *Tool* ini menganalisis repositori Git terstruktur di tujuh kategori: *secrets*, *bug risk*, *anti-pattern*, *security*, *performance*, *style*, dan *documentation*. Temuan dicatat dengan metadata komprehensif, termasuk varian *prompt*, tugas, iterasi, dan detail isu spesifik. Konfigurasi *analyzer* yang identik dalam lingkungan terkendali dipastikan guna menjaga konsistensi. Pendekatan yang ketat ini memungkinkan pengukuran kuantitatif terhadap dampak *prompt engineering* pada kualitas dan keamanan kode.

3.6 Tahap 6: Analisis Hasil dan Kesimpulan

Tahap akhir metodologi difokuskan pada interpretasi data hasil pengujian SAST. Analisis komparatif dilakukan dengan menghitung rata-rata jumlah isu per kategori guna mengidentifikasi pola kualitas antarvariasi *prompt*. Evaluasi total isu dan distribusi tingkat keparahan (*severity*) dianalisis untuk menentukan metrik teknik yang paling optimal. Interpretasi hasil akhir difasilitasi melalui visualisasi diagram batang, *heatmap*, dan tabulasi silang.

4 Hasil dan Pembahasan

Analisis DeepSource mendeteksi 6.932 isu pada 200 *snippet* kode, yang menunjukkan variabilitas yang signifikan. Perbedaan yang hampir dua kali lipat antara teknik dengan performa terbaik dan terburuk mengonfirmasi bahwa strategi *prompting* secara langsung memengaruhi kualitas kode non-fungsional. Distribusi terperinci berdasarkan kategori dan tingkat keparahan (*severity*) diuraikan sebagai berikut.

4.1 Distribusi Isu Berdasarkan Kategori

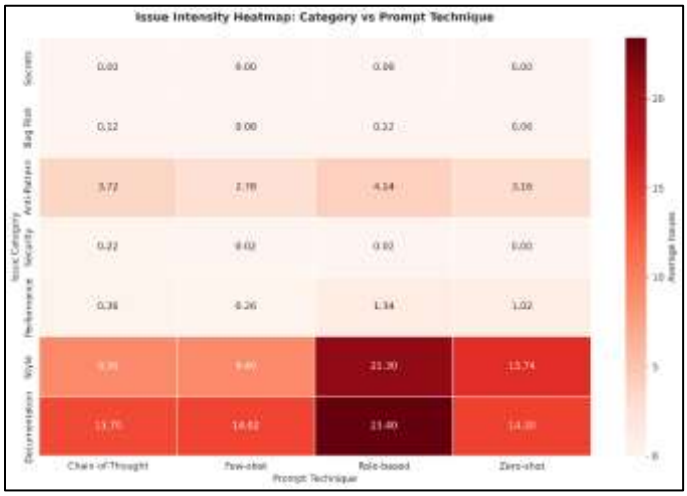
Tabel 2 menunjukkan bahwa *few-shot prompting* mencapai performa terbaik (1.328 isu), diikuti oleh *chain-of-thought* (1.374 isu) dan *zero-shot* (1.714 isu). *Role-based prompting* menghasilkan volume temuan tertinggi (2.516 isu), yang merepresentasikan 36,5% dari total keseluruhan.

Tabel 2 Distribusi total isu per kategori

Teknik *Prompt*	*Secrets*	*Bug risk*	*Anti-pattern*	*Security*	*Performance*	*Style*	*Documentation*	Total
Zero-shot	0	3	158	0	51	787	715	1.714
Few-shot	0	4	139	1	13	470	701	1.328
Chain-of-Thought	0	6	186	11	18	468	685	1.374
Role-based	0	6	207	1	67	1.065	1.170	2.516

 ISSN:2302-8149
e-ISSN:2540-9719

Pola distribusi ini divisualisasikan lebih lanjut melalui *heatmap* pada Gambar 2, yang menampilkan intensitas isu pada setiap iterasi. Warna yang lebih gelap pada kolom *Role-based* menunjukkan jumlah isu yang secara konsisten tinggi di seluruh pengujian *game*, kontras dengan *Few-shot* yang menampilkan profil warna yang lebih terang dan stabil.



Gambar 2 Heatmap distribusi isu

Lonjakan isu pada teknik *Role-based* secara khusus didominasi oleh kategori *Style* (1.065 isu) dan *Documentation* (1.170 isu). Hal ini mengindikasikan bahwa model menafsirkan persona "Senior Developer" sebagai keharusan untuk menggunakan gaya penulisan yang deskriptif dan eksplisit, yang secara tidak sengaja mengarah pada redundansi kode dan menyimpang dari prinsip Python yang ringkas (*Pythonic*). Kategori *Secrets* mencatat nol isu pada semua teknik, memvalidasi bahwa model menghindari halusinasi kredensial. *Few-shot prompting* menunjukkan efektivitasnya dalam menekan kesalahan pada kategori kritis seperti Anti-pattern, mengonfirmasi bahwa pemberian contoh konkret dapat membatasi pola kode yang buruk secara lebih efektif dibandingkan instruksi deskriptif semata.

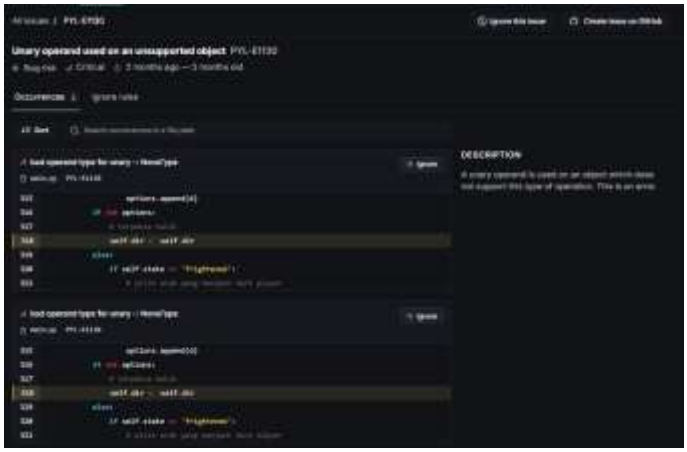
4.2 Visualisasi dan Studi Kasus Kerentanan Kode

Bagian ini menyajikan analisis visual dari temuan SAST untuk memperjelas dampak teknik *prompting* terhadap kualitas kode. Tangkapan layar berikut diambil dari iterasi eksperimental dengan anomali isu tertinggi.

4.2.1 Kerentanan Kritis pada *Chain-of-Thought* (CoT)

Teknik *Chain-of-Thought* (CoT) sering memicu kompleksitas yang berisiko memicu kegagalan sistem (*system crash*). Pada Gambar 3, DeepSource mendeteksi isu kritis PYL-E1130 pada Game Pacman (Iterasi 5). Model mencoba melakukan negasi *bitwise* (~) pada objek yang tidak valid, yang menyebabkan aplikasi langsung mengalami *crash*.

 ISSN:2302-8149
e-ISSN:2540-9719



Gambar 3 Kesalahan logika fatal (PYL-E1130)

4.2.2 Degradasi Kualitas Kode pada *Role-based Prompting*

Teknik *Role-based* dengan persona "*Senior Developer*" secara paradoks menghasilkan kepadatan isu *Style* tertinggi. Gambar 4 menunjukkan pelanggaran FLK-E501 (*line too long*) dan FLK-E701 (*multiple statements*) pada Tetris. Gaya pengkodean yang redundan ini mengurangi keterbacaan dan melanggar standar PEP-8.

Gambar 4 Pelanggaran gaya penulisan (FLK-E501)

4.2.3 Inefisiensi Performa

Teknik *Role-based* juga rentan terhadap inefisiensi. Gambar 5 menyoroti isu PYL-R0201, di mana model mendefinisikan *instance method* tanpa mengakses atribut kelas, sehingga menciptakan *overhead* memori yang tidak perlu (*technical debt*).

Gambar 5 Saran peningkatan performa (PYL-R0201)

 ISSN:2302-8149
e-ISSN:2540-9719

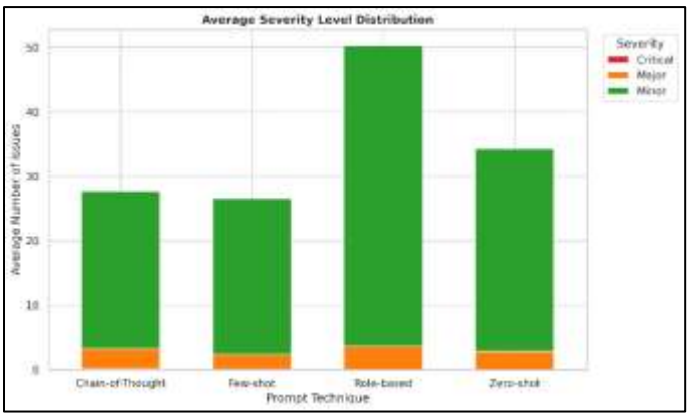
4.3 Analisis *Severity* dan *Trade-off*

Tabel 3 menampilkan distribusi tingkat keparahan (*severity*) isu yang terdeteksi, yang dikategorikan menjadi *critical*, *major*, dan *minor*.

Tabel 3 Distribusi tingkat keparahan isu

Teknik *Prompt*	*Critical*	*Major*	*Minor*
Zero-shot	1	141	1.572
Few-shot	3	114	1.211
Chain-of-Thought	5	157	1.222
Role-based	1	179	2.336

Chain-of-thought prompt menghasilkan isu *critical* terbanyak (5 isu), diikuti *few-shot* (3 isu), sementara *role-based* dan *zero-shot* masing-masing hanya menghasilkan 1 isu *critical*. Untuk kategori *major*, *role-based prompt* mendominasi dengan 179 isu, berbanding terbalik dengan *few-shot prompt* yang memiliki jumlah paling sedikit (114 isu). Pada kategori *minor*, analisis menunjukkan bahwa *role-based prompt* menghasilkan 2.336 temuan, hampir dua kali lipat dari teknik lainnya. Kondisi ini mengindikasikan bahwa meskipun *role-based prompting* efektif meminimalkan kesalahan kritis, teknik ini menghasilkan temuan minor yang berlebihan terkait dengan gaya penulisan dan dokumentasi.



Gambar 6 Distribusi tingkat keparahan isu

Terdapat *trade-off* tingkat keparahan yang signifikan. *Role-based* dan *zero-shot prompt* meminimalkan isu *critical* (masing-masing 1 isu), tetapi menghasilkan isu *minor* yang tinggi, terutama pada aspek *style* dan dokumentasi. Kerentanan *critical* berisiko menyebabkan kegagalan sistem dan eksploitasi, sehingga memerlukan perbaikan segera, sementara isu *minor* mengakumulasi *technical debt*. Lingkungan produksi (*production environment*) yang memprioritaskan keamanan akan lebih menyukai teknik yang menghasilkan lebih sedikit kesalahan *critical*. Pemeriksaan manual terhadap isu *critical* pada *chain-of-thought* melacak bahwa kesalahan utamanya berada pada algoritma *pathfinding* untuk AI (3 dari 5 isu), yang mengindikasikan bahwa dekomposisi logika yang kompleks justru meningkatkan risiko *runtime* pada logika yang spesifik domain (*domain-specific logic*). Isu *minor* pada *role-based* yang didominasi oleh format docstring yang tidak standar (58%) dan pelanggaran penamaan (31%) mengungkapkan kode fungsional yang gagal memenuhi standar dokumentasi yang baku.

 ISSN:2302-8149
e-ISSN:2540-9719

4.4 Analisis Berdasarkan Kategori Isu

Untuk memberikan pemahaman komprehensif tentang variasi kualitas kode, bagian ini merinci temuan spesifik untuk masing-masing dari tujuh kategori yang dianalisis oleh DeepSource, dengan menyoroti kekuatan dan kelemahan setiap teknik *prompt engineering.*

4.4.1 *Secrets*

Kategori *Secrets* mengevaluasi data sensitif yang di-*hardcode*, termasuk kunci API dan kata sandi. Seluruh teknik menghasilkan nol isu, yang mencerminkan karakteristik arsitektur *game standalone* tanpa persyaratan autentikasi eksternal, sekaligus mengindikasikan bahwa model terlatih untuk tidak menghasilkan kredensial palsu secara acak (halusinasi). Evaluasi di masa mendatang harus secara khusus menargetkan skenario yang melibatkan koneksi basis data atau integrasi API pihak ketiga untuk memvalidasi temuan ini di bawah kondisi yang lebih kompleks.

4.4.2 *Bug Risk*

Kategori *Bug Risk* mencakup potensi *runtime errors* dan *crash* pada program. *Zero-shot prompting* mencapai jumlah terendah (3 isu), diikuti oleh *Few-shot* (4 isu), sementara *Chain-of-Thought* dan *Role-based* masing-masing menghasilkan 6 isu. Anomali pada *Chain-of-Thought* berpusat pada *index out of bounds* dalam *pathfinding* dan kesalahan referensi objek (*null pointer*) dalam transisi *state*, yang mengindikasikan bahwa penalaran logis yang kompleks justru meningkatkan risiko *edge case* yang tidak tertangani. Keberhasilan *Few-shot* bergantung pada templat penanganan kesalahan berbasis contoh, sedangkan *Zero-shot* mengurangi titik kegagalan melalui logika yang lugas, meskipun harus dikompensasi (*trade-off*) dengan kelengkapan fungsional yang mungkin lebih terbatas.

4.4.3 *Anti-pattern*

Few-shot prompting menghasilkan *anti-pattern* paling sedikit (139 isu), diikuti oleh *Zero-shot* (158 isu), *Chain-of-Thought* (186 isu), dan *Role-based* (207 isu). Isu yang umum ditemukan meliputi *god objects*, *tight coupling*, metode yang terlalu panjang, dan pelanggaran *Single Responsibility Principle*. *Role-based prompting* menunjukkan performa terburuk karena persona 'pakar' (*expert*) secara paradoks mendorong terbentuknya kelas-kelas monolitik; dicontohkan oleh *game manager* sepanjang 500 baris yang menggabungkan logika *rendering*, *input*, AI, hingga sistem tabrakan (*collision*). Penalaran *Chain-of-Thought* memicu keterikatan antarkomponen yang ketat (*tight coupling*) akibat dependensi sekuensial pada alur logikanya. *Few-shot prompting* mengungguli teknik lain melalui pemanfaatan contoh kode yang merepresentasikan prinsip pemisahan fokus (*separation of concerns*), secara efektif memandu model menuju arsitektur yang lebih modular dan mudah dipelihara (*maintainable*).

4.4.4 *Security*

Zero-shot prompting mencapai tingkat keamanan optimal dengan nol isu, mengungguli *Chain-of-Thought* (11 isu), *Few-shot* (1 isu), dan *Role-based* (1 isu) secara signifikan. Kerentanan pada *Chain-of-Thought* mencakup penggunaan ‘eval()’ yang berbahaya untuk *parsing input*, penggunaan generator pseudo-acak yang lemah (‘random.random()’) sebagai pengganti modul *secrets*, serta absennya sanitasi *input*. Langkah-langkah penalaran terperinci dari teknik ini justru mendorong munculnya fitur-fitur lanjutan yang berisiko, seperti penggunaan *pickle* yang tidak tervalidasi untuk menyimpan *state* game, sehingga memperluas vektor serangan. Isu pada *Few-shot* dan *Role-based* terbatas pada penggunaan eval() dalam pemrosesan konfigurasi sederhana. Keberhasilan Zero-shot merupakan hasil dari implementasi yang konservatif dan lugas, yang secara otomatis menghindari penggunaan fitur-fitur berisiko tinggi tersebut.

4.4.5 *Performance*

Few-shot prompting memimpin di kategori *Performance* (13 isu), secara signifikan mengungguli *Chain-of-Thought* (18 isu), *Zero-shot* (51 isu), dan *Role-based* (67 isu). Anomali umum yang ditemukan meliputi inefisiensi algoritmik, komputasi redundan, dan indikasi kebocoran memori

 ISSN:2302-8149
e-ISSN:2540-9719

(*memory leaks*). *Role-based prompting* menghasilkan isu terbanyak akibat *over-engineering*, dicontohkan oleh penggunaan *collision loops* dengan kompleksitas '$O(n^3)$', kalkulasi *frame* yang redundan, serta absennya partisi spasial seperti *quadtrees*. *Zero-shot prompting* mengalami kelemahan akibat optimisasi yang tidak terarah, menghasilkan algoritma *breadth-first search* yang naif dan pencarian entitas linier tanpa indeks. Few-shot unggul karena memanfaatkan contoh yang mendemonstrasikan *memoization*, *lazy evaluation*, serta penggunaan struktur data yang efisien, seperti memprioritaskan format *sets* daripada *lists*.

4.4.6 *Style*

Kategori *Style* mengevaluasi kepatuhan pedoman PEP 8 mengenai penamaan, panjang kode, dan organisasi. *Chain-of-thought* dan *few-shot prompting* menunjukkan tingkat kepatuhan tertinggi dengan masing-masing menyisakan 468 dan 470 isu, sementara *zero-shot* (787 isu) dan *role-based* (1.065 isu) tertinggal secara signifikan. Pelanggaran *role-based* bermula dari pola penulisan yang terlalu panjang (redundan) akibat persona "senior developer", menghasilkan konvensi penamaan yang tidak konsisten, baris panjang yang melebihi 150 karakter, dan penggunaan nama variabel berlebihan seperti 'current_player_piece_position_x_coordinate'. *Zero-shot prompting* menghadapi kesulitan pada pemformatan dasar seperti pencampuran inden dan tidak adanya baris kosong. Panduan terstruktur beserta contoh yang ada memungkinkan *chain-of-thought* dan *few-shot* untuk lebih menjaga konsistensi pemformatan.

4.4.7 *Documentation*

Kategori Documentation menilai kualitas *docstring* dan komentar; di mana *role-based prompting* menghasilkan volume isu tertinggi (1.170 isu), sangat kontras bila dibandingkan dengan *zero-shot* (715 isu), *few-shot* (701 isu), dan *chain-of-thought* (685 isu). Teknik ini melanggar prinsip kode yang "*self-documenting*" akibat penjelasan yang berlebihan, menggunakan deskripsi implementasi multi-paragraf, komentar redundan (seperti 'increment counter'), dan memberikan templat yang tidak terisi. *Chain-of-thought* mencapai hasil terbaik melalui struktur *self-explanatory* yang berasal dari penalarannya yang bersifat langkah demi langkah. *Few-shot* mampu menyeimbangkan keringkasan secara lebih efisien, sementara *zero-shot* cenderung menghindari pemberian komentar yang berlebihan (*clutter*), namun seringkali menghilangkan dokumentasi penting pada fungsi-fungsi yang kompleks.

5 Kesimpulan

Penelitian ini berhasil mengidentifikasi pengaruh signifikan teknik *prompt engineering* terhadap kualitas nonfungsional kode yang dihasilkan LLM. Hasil utama menunjukkan bahwa *few-shot prompting* menghasilkan jumlah isu terendah secara keseluruhan (1.328 isu) dibandingkan teknik lainnya. Teknik ini terbukti paling efektif mengurangi risiko teknis, terutama dalam meminimalkan *anti-pattern* dan isu performa. *Few-shot prompting* juga menghasilkan kualitas kode yang paling stabil dan konsisten pada berbagai tingkat kompleksitas tugas.

Analisis tingkat keparahan isu menunjukkan adanya *trade-off* penting dalam memilih teknik *prompt*. *Role-based* dan *zero-shot prompting* berhasil meminimalkan isu *critical* (masing-masing 1 isu), namun menghasilkan isu *minor* dalam volume tinggi, terutama terkait *style* dan dokumentasi. Hal ini berdampak signifikan pada peningkatan *technical debt*. Teknik *chain-of-thought prompting* mencatat total isu moderat dengan tingkat kemunculan isu *critical* tertinggi (5 isu). Hal ini menunjukkan bahwa memecah logika menjadi langkah-langkah kompleks tanpa validasi yang ketat dapat menyebabkan kesalahan serius pada algoritma.

Berdasarkan hasil tersebut, penelitian ini merumuskan rekomendasi praktis untuk strategi *AI-assisted development*. *Few-shot prompting* disarankan sebagai pendekatan standar (*baseline*) untuk lingkungan produksi (*production environment*) karena menawarkan keseimbangan optimal antara volume isu keseluruhan, kualitas, dan konsistensi kode. Meskipun teknik lain seperti *zero-shot* dan *role-based* menghasilkan lebih sedikit isu *critical*, *few-shot* tetap menjadi pilihan yang lebih aman secara keseluruhan dengan mempertimbangkan total *technical debt* yang lebih rendah. Teknik *chain-of-thought* lebih cocok untuk fase eksplorasi atau *prototyping* yang membutuhkan penalaran logika kompleks, tetapi harus divalidasi secara manual dengan lebih teliti. Penggunaan *role-based* dan *zero-*

shot prompting memerlukan perbaikan dan optimasi manual sebelum kode dapat digunakan. Temuan ini memvalidasi urgensi integrasi SAST secara otomatis dalam *pipeline* CI/CD sebagai lapisan pertahanan terakhir untuk menjamin keamanan kode.

Penelitian ini memiliki beberapa keterbatasan yang perlu dipertimbangkan saat menginterpretasikan hasil. Pertama, evaluasi dilakukan secara eksklusif pada tugas pengembangan *game* berbasis Python, yang mungkin membatasi kemampuan generalisasi temuan pada bahasa pemrograman lain seperti Java, JavaScript, atau C++, di mana masing-masing memiliki konvensi sintaksis, masalah keamanan, dan karakteristik performa yang berbeda. Kedua, analisis ini hanya mengandalkan DeepSource sebagai *tool* SAST, sementara *tools* lain seperti SonarQube, Semgrep, atau Bandit mungkin mendeteksi pola isu yang berbeda atau memberikan klasifikasi tingkat keparahan (*severity*) alternatif. Ketiga, penelitian ini menggunakan GPT-5 sebagai satu-satunya LLM, tanpa membandingkan hasil pada berbagai model berbeda seperti Claude, Gemini, atau Llama, yang dapat menunjukkan perilaku generasi kode yang bervariasi. Keempat, penggunaan GPT-5 melalui platform Windsurf dengan konfigurasi *default* dapat menghasilkan *output* yang berbeda jika dijalankan menggunakan *direct API access* dengan parameter kustom, yang dapat mempengaruhi *reproducibility* hasil pada *environment* berbeda. Kelima, evaluasi dibatasi pada analisis statis tanpa menggabungkan pengujian dinamis (*dynamic testing*), penilaian perilaku *runtime*, atau validasi kebenaran fungsional, yang merupakan aspek krusial dalam evaluasi kualitas kode secara komprehensif.

Penelitian selanjutnya diharapkan dapat memperluas evaluasi ke beragam bahasa pemrograman (Java, JavaScript, C++) dan domain (*web*, *mobile*, *enterprise*) untuk menguji universalitasnya. Studi komparatif yang memanfaatkan berbagai *tools* SAST (SonarQube, Semgrep, Bandit) akan mengidentifikasi konsensus temuan dan mengurangi bias yang spesifik pada *tool* tertentu. Mengevaluasi berbagai LLM (Claude, Gemini, Llama) dan pendekatan *prompting* kombinasi dapat mengungkap efek pelengkap dari masing-masing model. Pengujian pada berbagai platform akses LLM, baik melalui *integrated AI editor*, *direct API*, maupun *web interface*, dapat memberikan pemahaman lebih mendalam tentang pengaruh *environment* terhadap konsistensi dan kualitas *output* kode. Integrasi analisis dinamis dan pengujian fungsional bersama SAST berpotensi memberikan metrik kualitas yang lebih menyeluruh. Pendekatan komprehensif ini mampu mengidentifikasi anomali *runtime* yang luput dari pemindaian statis.

Referensi

[1] S. Baltes *et al.*, "*Guidelines for Empirical Studies in Software Engineering involving Large Language Models*," Mar. 02, 2026, *arXiv*: arXiv:2508.15503. doi: 10.48550/arXiv.2508.15503.

[2] DORA Team, "*State of AI-Assisted Software Development*," Google Cloud, Research Report, 2025. [Online]. Available: https://services.google.com/fh/files/misc/2025_state_of_ai_assisted_software_development.pdf

[3] R. Pandey, P. Singh, R. Wei, and S. Shankar, "*Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects*," Jun. 25, 2024, *arXiv*: arXiv:2406.17910. DOI: 10.48550/arXiv.2406.17910.

[4] I. D. Fagadau, L. Mariani, D. Micucci, and O. Riganelli, "*Analyzing Prompt Influence on Automated Method Generation: An Empirical Study with Copilot*," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, Apr. 2024, pp. 24–34. DOI: 10.1145/3643916.3644409.

[5] S. Gao *et al.*, "*The Prompt Alchemist: Automated LLM-Tailored Prompt Optimization for Test Case Generation*," Jan. 02, 2025, *arXiv*: arXiv:2501.01329. DOI: 10.48550/arXiv.2501.01329.

[6] Y. Fu *et al.*, "*Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study*," Feb. 06, 2025, *arXiv*: arXiv:2310.02059. DOI: 10.48550/arXiv.2310.02059.

[7] "*How We Ensure Less than 5% False Positive Rate • DeepSource*," DeepSource. [Online]. Available: https://deepsource.com/blog/how-deepsource-ensures-less-false-positives

[8] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, "'*False Negative -- that One is Going to Kill You': Understanding Industry Perspectives of Static Analysis based Security Testing*," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024, pp. 3979–3997. DOI: 10.1109/SP54263.2024.00019.

[9] W. Peng, X. Wang, and Q. Wu, "*ProxyWar: Dynamic Assessment of LLM Code Generation in Game Arenas*," Feb. 04, 2026, *arXiv*: arXiv:2602.04296. DOI: 10.48550/arXiv.2602.04296.
[10] S. Schulhoff *et al.*, "*The Prompt Report: A Systematic Survey of Prompt Engineering Techniques*," Feb. 2025, DOI: 10.48550/arXiv.2406.06608.
[11] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "*Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing*," Jul. 28, 2021, *arXiv*: arXiv:2107.13586. DOI: 10.48550/arXiv.2107.13586.
[12] H. Guo, "*An Empirical Study of Prompt Mode in Code Generation based on ChatGPT*," *Appl. Comput. Eng.*, Vol. 73, No. 1, pp. 69–76, Jul. 2024, DOI: 10.54254/2755-2721/73/20240367.
[13] S. Anasuri, "*Prompt Engineering Best Practices for Code Generation Tools*," *Int. J. Emerg. Trends Comput. Sci. Inf. Technol.*, Vol. 5, No. 1, pp. 69–81, Mar. 2024, DOI: 10.63282/3050-9246.IJETCSIT-V5I1P108.
[14] J. Wei *et al.*, "*Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*," Jan. 10, 2023, *arXiv*: arXiv:2201.11903. DOI: 10.48550/arXiv.2201.11903.
[15] H. Louatouate and M. Zeriouh, "*Role-based Prompting Technique in Generative AI-Assisted Learning: A Student-Centered Quasi-Experimental Study*," *J. Comput. SCI. Technol. Stud.*, Vol. 7, No. 2, pp. 130–145, Apr. 2025, DOI: 10.32996/jcsts.2025.7.2.12.
[16] E. Basic and A. Giaretta, "*From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security*," Apr. 14, 2025, *arXiv*: arXiv:2412.15004. DOI: 10.48550/arXiv.2412.15004.
[17] A. Sabra, O. Schmitt, and J. Tyler, "*Assessing the Quality and Security of AI-Generated Code: A Quantitative Analysis*," Aug. 20, 2025, *arXiv*: arXiv:2508.14727. DOI: 10.48550/arXiv.2508.14727.
[18] Y. Liu, R. Widyasari, Y. Zhao, I. C. Irsan, J. Chen, and D. Lo, "*Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild*," 2026, *arXiv*. doi: 10.48550/ARXIV.2603.28592.
[19] T. Hardiani, D. Wijayanto, and N. Latifah, "*Data Security Analysis with OWASP Framework on Website XYZ*," *CYBERNETICS*, Vol. 6, No. 01, p. 10, Jul. 2022, DOI: 10.29406/cbn.v6i01.3953.
[20] M. R. Syam Al'Am'yubi and D. Wijayanto, "Analisis Sistem Keamanan *Website* XYZ menggunakan *Framework OWASP ZAP*," *J. Ilmu Komput. JUIK*, Vol. 3, No. 1, p. 1, Mar. 2023, DOI: 10.31314/juik.v3i1.1974.
[21] D. Wijayanto and A. Firdonsyah, "Analisis Tingkat Resiko pada *Website XYZ* menggunakan Metode *OWASP*," *Digit. Transform. Technol.*, Vol. 4, No. 1, pp. 644–651, Aug. 2024, DOI: 10.47709/digitech.v4i1.4485.
[22] M. Esposito, V. Falaschi, and D. Falessi, "*An Extensive Comparison of Static Application Security Testing Tools*," Mar. 14, 2024, *arXiv*: arXiv:2403.09219. DOI: 10.48550/arXiv.2403.09219.
[23] D. Tosi, "*Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation*," *Future Internet*, Vol. 16, No. 6, p. 188, May 2024, DOI: 10.3390/fi16060188.
[24] "*GPT-5 Benchmarks and Analysis*." [Online]. Available: https://artificialanalysis.ai/articles/gpt-5-benchmarks-and-analysis
[25] "*GPT-5 High Reasoning Evaluation: A Major Leap in Coding Performance*," 16x Eval. [Online]. Available: https://eval.16x.engineer/blog/gpt-5-high-reasoning-coding-performance-evaluation