VERIFIKASI KUALITAS KODE BERBASIS LARGE LANGUAGE MODELS: TINJAUAN LITERATUR SISTEMATIS TERHADAP TANTANGAN DAN PENDEKATAN STRATEGIS

Sandi Zikrullah, Evans Fuad
Teknik Informatika, Universitas Muhammdiyah Riau
Jl. Tuanku Tambusai, Delima, Kec. Tampan, Kota Pekanbaru, Riau
220401086@student.umri.ac.id

ABSTRAK

Pemanfaatan *Large Language Models* (LLM) dalam rekayasa perangkat lunak menawarkan lonjakan produktivitas yang signifikan. Namun, hal ini menghadirkan paradoks kualitas berupa fenomena "ilusi kebenaran" di mana kode tampak valid namun cacat logika, risiko replikasi kerentanan keamanan dari data pelatihan, degradasi keterpeliharaan akibat tingginya *code churn*, serta ketidakjelasan aspek kepatuhan lisensi. Penelitian ini bertujuan untuk memetakan tantangan kualitas tersebut serta mengevaluasi strategi verifikasi yang efektif. Metode yang digunakan adalah *Systematic Literature Review* (SLR) dengan kerangka PRISMA terhadap 40 studi primer yang diterbitkan antara tahun 2018 hingga 2025. Hasil penelitian menunjukkan bahwa strategi verifikasi telah berevolusi dari pengujian manual menuju pendekatan otonom hibrida. Solusi mutakhir yang teridentifikasi mencakup penerapan *Shift-Left Security* melalui *secure prompting*, penggunaan *AI-Driven Test Generation*, serta pemanfaatan agen otonom (*Self-Healing Agents*) untuk perbaikan mandiri. Penelitian ini mengusulkan kerangka kerja "Siklus Verifikasi Otonom Hibrida" sebagai standar baru untuk memastikan integritas sistem di era pemrograman berbasis AI.

Kata kunci : *Large Language Models, Verifikasi Kode, Keamanan Perangkat Lunak, Tinjauan Literatur Sistematis, Self-Healing Code.*

1. PENDAHULUAN

Perkembangan pesat kecerdasan buatan, khususnya *Large Language Models* (LLM) seperti ChatGPT dan GitHub Copilot, telah merevolusi paradigma rekayasa perangkat lunak modern. Kemampuan LLM untuk memahami konteks dan menghasilkan kode secara otomatis menawarkan efisiensi yang belum pernah terjadi sebelumnya dalam siklus pengembangan perangkat lunak . Studi empiris mengonfirmasi bahwa penggunaan asisten pengkodean berbasis AI dapat mempercepat penyelesaian tugas pemrograman secara signifikan, yang berdampak langsung pada peningkatan produktivitas pengembang [1]. Konsep *AI-Assisted Software Engineering* kini telah bertransformasi dari eksperimen akademis menjadi standar praktik industri.

Namun, adopsi masif teknologi ini menghadirkan paradoks kualitas yang serius. Meskipun produktivitas meningkat, terdapat kekhawatiran mendasar mengenai integritas, keamanan, dan keterpeliharaan kode yang dihasilkan. Literatur terkini mengidentifikasi fenomena yang disebut sebagai "ilusi kebenaran" (*illusion of correctness*), di mana kode yang dihasilkan oleh AI tampak valid secara sintaksis dan meyakinkan, namun mengandung cacat logika yang halus atau kerentanan keamanan yang tersembunyi [2], [3]. Berbeda dengan kesalahan manusia yang sering kali mudah diprediksi, kesalahan AI berakar pada probabilitas statistik yang dapat memicu kegagalan sistem yang tidak terduga pada skenario batas (*edge cases*).

Masalah keamanan menjadi perhatian utama, mengingat LLM dilatih menggunakan data publik yang sering kali mengandung pola pengkodean yang tidak aman. Sebuah Penelitian menemukan bahwa tanpa pengawasan yang ketat, asisten AI berpotensi mereplikasi kerentanan keamanan siber lama, seperti *SQL Injection*, ke dalam basis kode modern [4]. Selain itu, dari sisi pemeliharaan jangka panjang, studi longitudinal menunjukkan adanya korelasi antara penggunaan AI dengan peningkatan *code churn* dan penurunan kualitas modularitas kode, yang berisiko memperburuk utang teknis (*technical debt*) perusahaan [5]. Selain itu, ketidakjelasan aspek lisensi dan hak cipta turut menjadi risiko hukum baru, di mana kode generatif berpotensi melanggar aturan atribusi *open-source* atau mengandung plagiarisme yang tidak disengaja [5].

Meskipun tantangan ini nyata, literatur mengenai strategi verifikasi yang sistematis untuk kode generatif masih terfragmentasi. Sebagian besar penelitian berfokus pada evaluasi akurasi model semata, tanpa menawarkan kerangka kerja verifikasi yang komprehensif untuk lingkungan produksi [6]. Oleh karena itu, penelitian ini bertujuan untuk mengisi kesenjangan tersebut. Sebagai rencana penyelesaian masalah, penelitian ini menerapkan metode *Systematic Literature Review* (SLR) untuk memetakan 40 studi primer terpilih. Melalui pendekatan ini, artikel akan menganalisis tantangan multidimensional kualitas kode AI dan mensintesis evolusi strategi verifikasi terkini, mulai dari *secure prompting* hingga *self-healing agents*, guna merumuskan standar baru dalam penjaminan kualitas perangkat lunak berbasis AI.

2. TINJAUAN PUSTAKA

2.1. Large Language Models dalam AI-Assisted Programming

Integrasi *Large Language Models* (LLM) ke dalam lingkungan pengembangan terintegrasi (*Integrated Development Environment* / IDE) telah membawa perubahan mendasar dalam praktik rekayasa perangkat lunak. Proses pengembangan yang sebelumnya didominasi oleh penulisan kode secara manual kini bergeser menuju pendekatan orkestrasi berbasis prompt. Dengan kemampuan memahami konteks semantik dalam skala besar, LLM memungkinkan otomatisasi berbagai tugas kompleks, termasuk pembuatan kode, refaktorisasi, serta penyusunan dokumentasi teknis secara efisien [7]. Transformasi ini berkontribusi signifikan terhadap peningkatan produktivitas pengembang dan percepatan siklus pengembangan perangkat lunak.

Meskipun demikian, secara konseptual LLM tidak dirancang sebagai mesin penalaran logika yang deterministik. Model ini beroperasi sebagai sistem probabilistik yang dioptimalkan untuk meminimalkan *loss function* dalam proses prediksi token berikutnya. Karakteristik tersebut menimbulkan keterbatasan fundamental, khususnya dalam menjamin kebenaran logika dan konsistensi semantik dari kode yang dihasilkan. Salah satu implikasi paling menonjol adalah munculnya fenomena *hallucination*, di mana model menghasilkan kode yang valid secara sintaksis, namun mengandung referensi terhadap pustaka yang tidak tersedia atau logika yang keliru.

Berbeda dengan *static analysis tools* konvensional yang bekerja secara deterministik berdasarkan aturan formal, luaran LLM bersifat non-deterministik dan dapat bervariasi untuk masukan yang serupa. Kondisi ini menjadikan proses verifikasi kualitas kode sebagai tantangan baru yang tidak sepenuhnya dapat ditangani melalui pendekatan pengujian tradisional. Oleh karena itu, penggunaan LLM dalam *AI-assisted programming* menuntut mekanisme verifikasi yang lebih adaptif dan kontekstual untuk menjamin kualitas perangkat lunak yang dihasilkan.

2.2. Standar Kualitas Perangkat Lunak (ISO/IEC 25010)

Dalam konteks evaluasi kode generatif, literatur umumnya merujuk pada standar kualitas perangkat lunak ISO/IEC 25010 sebagai kerangka acuan utama. Standar ini mendefinisikan sejumlah karakteristik kualitas, antara lain fungsionalitas, keandalan, keamanan, dan keterpeliharaan. Penerapan standar tersebut memungkinkan penilaian kualitas kode dilakukan secara sistematis dan terstruktur, termasuk pada artefak perangkat lunak yang dihasilkan oleh sistem berbasis AI.

Sejumlah studi menyoroti bahwa di antara berbagai dimensi kualitas tersebut, aspek keamanan merupakan karakteristik yang paling rentan mengalami degradasi dalam konteks kode generatif. Kerentanan ini berkaitan erat dengan mekanisme pelatihan LLM yang menyerap pola dari korpus data open-source berskala besar, yang sering kali tidak melalui proses sanitasi keamanan yang memadai. Akibatnya, praktik pengkodean yang tidak aman dapat terinternalisasi ke dalam perilaku model.

Penelitian terdahulu menunjukkan bahwa LLM cenderung mereproduksi pola pengkodean usang yang lazim ditemukan dalam data historis, seperti penggunaan perintah SQL tanpa parameterisasi atau pemilihan algoritma kriptografi yang sudah tidak direkomendasikan [4]. Tanpa adanya intervensi atau pembatasan eksplisit, model berfungsi sebagai refleksi dari praktik masa lalu dan secara tidak langsung memperbanyak *security debt* ke dalam basis kode modern [8]. Temuan ini menegaskan pentingnya pendekatan verifikasi yang secara khusus menargetkan dimensi keamanan dalam pengembangan kode berbasis LLM.

2.3. Evolusi Paradigma Verifikasi

Meningkatnya kompleksitas kode yang dihasilkan oleh sistem generatif telah mendorong perubahan signifikan dalam paradigma verifikasi perangkat lunak. Pendekatan tradisional yang sangat bergantung pada *manual code review* semakin dipandang tidak efisien, terutama ketika berhadapan dengan kode yang dihasilkan oleh mesin. Literatur menggambarkan fenomena ini sebagai *paradoks beban kognitif*, di mana upaya mental yang dibutuhkan untuk memahami dan memverifikasi kode hasil generasi AI justru melebihi upaya yang diperlukan untuk menulis kode secara manual [9].

Sebagai respons terhadap keterbatasan tersebut, paradigma verifikasi mulai bergeser ke arah otomatisasi hibrida yang memadukan peran manusia dan sistem otonom. Pendekatan *Shift-Left Security* menekankan pentingnya pencegahan kerentanan sejak tahap awal pengembangan melalui rekayasa instruksi (*prompt engineering*) yang ketat dan terstruktur [10]. Dengan membatasi ruang solusi model sejak awal, potensi munculnya kerentanan dapat ditekan sebelum kode dihasilkan.

Di sisi lain, konsep *Self-Healing Code* mulai mendapatkan perhatian sebagai strategi untuk meningkatkan kualitas dan keterpeliharaan kode generatif. Pendekatan ini memanfaatkan agen otonom yang mampu melakukan introspeksi terhadap kesalahan eksekusi, menganalisis *stack trace*, serta melakukan perbaikan dan refaktorisasi secara iteratif tanpa intervensi manusia langsung [11]. Evolusi paradigma ini mencerminkan pergeseran peran manusia dari pelaksana teknis menjadi pengawas strategis dalam proses verifikasi perangkat lunak berbasis AI.

2.4. Penelitian Terdahulu

Sejumlah penelitian terdahulu telah mengkaji berbagai aspek spesifik dari kode yang dihasilkan oleh kecerdasan buatan. Pearce memfokuskan kajiannya

pada evaluasi keamanan GitHub Copilot dan menemukan bahwa sekitar 40% kode yang dihasilkan mengandung kerentanan keamanan siber, terutama pada skenario yang berkaitan dengan basis data dan penanganan input pengguna [4]. Di sisi lain, Nguyen dan Nadi melakukan studi empiris terhadap aspek kebenaran logika (*correctness*) dan mengungkapkan adanya variabilitas akurasi yang signifikan antar bahasa pemrograman, di mana performa pada bahasa Java jauh mengungguli JavaScript [2].

Selain aspek teknis, Harding menyoroti dampak jangka panjang penggunaan AI terhadap kualitas struktural kode, mencatat adanya peningkatan *code churn* dan duplikasi yang berisiko memperburuk utang teknis (*technical debt*) [5]. Meskipun studi-studi tersebut memberikan wawasan krusial, mayoritas penelitian yang ada masih bersifat parsial dengan hanya berfokus pada satu dimensi kualitas (misalnya hanya keamanan atau hanya akurasi). Penelitian ini hadir untuk melengkapi literatur tersebut dengan menyajikan tinjauan sistematis yang mengintegrasikan empat dimensi sekaligus—logika, keamanan, keterpeliharaan, dan lisensi—serta merumuskan kerangka kerja verifikasi otonom hibrida yang belum banyak dibahas secara holistik dalam studi sebelumnya.

3. METODE PENELITIAN

Penelitian ini menggunakan pendekatan kualitatif dengan metode Systematic Literature Review (SLR) untuk mengkaji tantangan serta strategi verifikasi kualitas kode yang dihasilkan oleh *Large Language Models* (LLM). Proses kajian ini mengadopsi kerangka kerja PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guna memastikan transparansi, replikasi, dan ketelitian ilmiah dalam proses penelusuran serta seleksi literatur. Prosedur SLR dilaksanakan melalui lima tahap utama, yaitu: (1) penentuan kriteria kelayakan artikel, (2) identifikasi sumber informasi, (3) pemilihan artikel, (4) pengumpulan data, dan (5) analisis data.

3.1. Tahap 1: Kriteria Kelayakan Artikel

Kriteria inklusi (*Inclusion Criteria/IC*) ditetapkan untuk memastikan bahwa artikel yang dianalisis relevan dengan tujuan dan pertanyaan penelitian. Adapun kriteria inklusi yang digunakan adalah sebagai berikut:

a. IC1: Artikel membahas topik spesifik mengenai kualitas kode hasil generasi AI, verifikasi kode, atau evaluasi keamanan LLM (misalnya GitHub Copilot atau ChatGPT).
b. IC2: Artikel diterbitkan dalam rentang tahun 2018–2025 guna menjamin kebaruan dan relevansi teknologi yang dikaji.
c. IC3: Artikel ditulis dalam Bahasa Inggris atau Bahasa Indonesia.
d. IC4: Artikel merupakan publikasi ilmiah berupa jurnal atau prosiding konferensi yang telah melalui proses *peer-review*, serta preprint bereputasi tinggi.

Mengingat pesatnya perkembangan teknologi LLM, artikel preprint dari repositori terpercaya seperti arXiv disertakan untuk menangkap temuan *state-of-the-art* yang belum dipublikasikan secara formal namun memiliki kontribusi ilmiah yang signifikan.

3.2. Tahap 2: Identifikasi Sumber Artikel

Proses pencarian literatur dilakukan pada basis data akademik bereputasi yang relevan dengan bidang teknik informatika dan rekayasa perangkat lunak, yaitu IEEE Xplore, ACM Digital Library, Scopus, dan repositori arXiv. Penelusuran dilakukan menggunakan kombinasi kata kunci (keywords) dengan operator Boolean sebagai berikut: "AI-generated code" OR "LLM code generation" AND "software quality" OR "code verification" OR "security vulnerability". Strategi pencarian ini dirancang untuk menjaring studi yang membahas tantangan kualitas kode serta pendekatan verifikasi dan mitigasi risiko dalam pengembangan perangkat lunak berbasis AI.

3.3. Tahap 3: Pemilihan Artikel

Seleksi artikel dilakukan melalui proses penyaringan bertingkat sesuai dengan alur PRISMA, yang meliputi:

a. Identifikasi: Pencarian awal artikel berdasarkan kata kunci pada seluruh basis data yang ditentukan.
b. Penyaringan (Screening): Peninjauan judul dan abstrak untuk mengeliminasi artikel yang tidak relevan dengan topik verifikasi kualitas kode.
c. Kelayakan (Eligibility): Pemeriksaan teks lengkap (full-text) artikel kandidat untuk memastikan kesesuaiannya dengan pertanyaan penelitian.
d. Inklusi Akhir: Artikel yang memenuhi seluruh kriteria ditetapkan sebagai sumber data utama (primary studies).

3.4. Tahap 4: Pengumpulan Data

Berdasarkan hasil pencarian awal, diperoleh total 553 artikel dari seluruh basis data. Setelah melalui proses penyaringan dan seleksi yang ketat, sebanyak 40 studi primer (N = 40) dinyatakan memenuhi kriteria inklusi dan digunakan dalam analisis akhir.
Rincian distribusi temuan awal dan jumlah studi terpilih dari setiap basis data disajikan pada Tabel 1.

Tabel 1. Distribusi Sumber Data Literatur

Basis Data (Sumber)	Jumlah Temuan Awal	Jumlah Studi Terpilih (Inklusi)
IEEE Xplore	112	9
ACM Digital Library	145	10
Scopus	86	6
arXiv (Preprint Bereputasi)	210	16
Total	553	40

3.5. Tahap 5: Analisis Data

Data yang dikumpulkan dari 40 artikel terpilih dianalisis menggunakan analisis tematik. Informasi yang diekstraksi mencakup jenis tantangan kualitas kode (logika, keamanan, dan keterpeliharaan) serta strategi verifikasi yang diusulkan dalam masing-masing studi.

Analisis dilakukan dengan mengidentifikasi pola masalah yang dominan serta mengelompokkan pendekatan solusi yang muncul dalam literatur, guna menjawab pertanyaan penelitian mengenai standar kualitas dan mekanisme verifikasi kode pada era AI-Assisted Programming.

4. HASIL DAN PEMBAHASAN

Berdasarkan analisis terhadap 40 studi primer yang terbit pada periode 2018–2025, penelitian ini menggambarkan secara komprehensif perkembangan praktik verifikasi pada kode yang dihasilkan secara generatif. Hasil analisis menunjukkan bahwa pemanfaatan *Large Language Models* (LLM) dalam rekayasa perangkat lunak menghadirkan dilema mendasar, yakni peningkatan produktivitas pengembangan yang beriringan dengan munculnya potensi penurunan kualitas dan integritas struktural kode. Temuan-temuan tersebut kemudian dikelompokkan ke dalam dua fokus utama pembahasan, yaitu identifikasi tantangan kualitas kode generatif sebagai jawaban atas RQ1 serta perkembangan pendekatan verifikasi yang digunakan untuk merespons tantangan tersebut sebagai jawaban atas RQ2. Secara keseluruhan, analisis ini menegaskan bahwa adopsi LLM tidak hanya membawa manfaat efisiensi, tetapi juga memunculkan risiko baru yang menuntut strategi verifikasi yang lebih adaptif dan sistematis.

4.1. Analisis Tantangan Kualitas Kode

Hasil sintesis literatur menunjukkan adanya empat kategori tantangan utama yang bersifat sistemik dalam penggunaan kode generatif berbasis LLM, yaitu ketidakpastian logika, peningkatan risiko keamanan, penurunan keterpeliharaan dalam jangka panjang, serta ketidakjelasan aspek lisensi. Keempat area ini muncul secara konsisten dalam berbagai studi dan merepresentasikan persoalan fundamental yang memengaruhi kualitas perangkat lunak secara keseluruhan.

4.2. Ketidakpastian Logika dan Ilusi Kebenaran

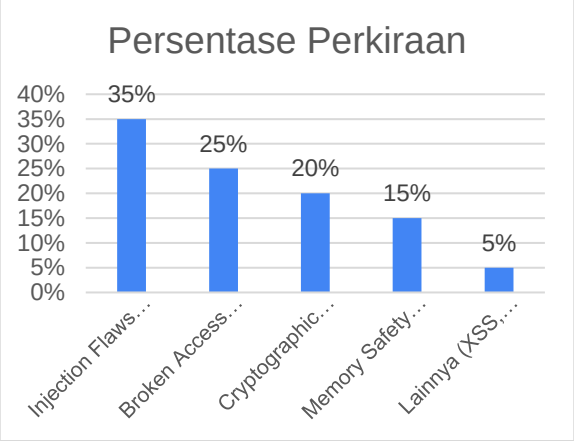
Tantangan paling mendasar yang teridentifikasi dalam literatur adalah munculnya fenomena *illusion of correctness* atau ilusi kebenaran. Kode yang dihasilkan oleh LLM umumnya terlihat benar secara sintaksis, rapi secara struktur, dan meyakinkan dari sisi visual, namun kerap menyimpan kesalahan logika yang bersifat kritis. Beberapa studi empiris yang mengevaluasi penggunaan GitHub Copilot menunjukkan adanya variasi tingkat akurasi yang signifikan antar bahasa pemrograman. Temuan empiris mengindikasikan bahwa solusi yang dihasilkan dalam bahasa Java memiliki tingkat kebenaran sekitar 57%, sementara pada bahasa JavaScript tingkat kebenarannya turun drastis hingga kisaran 27%[2].

Analisis lebih lanjut mengungkap bahwa sekitar 51,2% dari kode yang dihasilkan masuk dalam kategori "sebagian benar" (*partially correct*)[3]. Kode jenis ini mampu beroperasi pada skenario standar (*happy path*), namun gagal saat menangani kondisi batas (*edge cases*). Kondisi ini memicu apa yang dikenal sebagai "Paradoks Beban Kognitif", di mana pengembang justru menghabiskan waktu lebih lama untuk menelaah dan memperbaiki kode mesin yang asing dibandingkan menulis kode dari awal[9].

4.3. Amplifikasi Kerentanan Keamanan

Dalam domain keamanan siber, sejumlah studi mengidentifikasi bahwa LLM berpotensi berfungsi sebagai mekanisme amplifikasi kerentanan perangkat lunak. Hal ini berkaitan erat dengan karakteristik data pelatihan model yang banyak bersumber dari repositori open-source, yang tidak selalu melalui proses penyaringan atau sanitasi keamanan yang memadai. Akibatnya, pola kode yang tidak aman dapat direplikasi dan disebarkan kembali melalui hasil generatif model.

Sebuah evaluasi keamanan terhadap kode yang dihasilkan GitHub Copilot melaporkan temuan yang cukup mengkhawatirkan, di mana sekitar 40% kode yang dihasilkan mengandung kerentanan dengan tingkat risiko tinggi berdasarkan klasifikasi *Common Weakness Enumeration* (CWE)[4]. Sebagaimana ditunjukkan pada Gambar 1, jenis kerentanan yang paling sering muncul mencakup kesalahan injeksi serta kelemahan pada mekanisme kontrol akses.



Gambar 1. Distribusi Kategori Kerentanan Keamanan pada Kode Generatif.

Dominasi kerentanan *Injection Flaws* (35%) dan *Broken Access Control* (25%) pada Gambar 1 menunjukkan kegagalan model dalam memisahkan instruksi dan data yang tidak tepercaya[4]. Risiko ini diperburuk oleh temuan bahwa pengembang dengan akses AI cenderung menulis kode yang kurang aman

karena bias kepercayaan berlebihan (*automation bias*) terhadap luaran sistem[8].

4.4. Degradasi Efisiensi dan Keterpeliharaan

Di luar aspek kebenaran fungsional dan keamanan, penggunaan AI dalam pengembangan perangkat lunak juga berdampak signifikan terhadap kualitas kode dalam jangka panjang, khususnya dari sisi efisiensi dan keterpeliharaan (*maintainability*). Studi berskala besar melaporkan peningkatan tajam pada nilai *code churn*, yaitu proporsi kode yang mengalami revisi atau penghapusan dalam kurun waktu kurang dari dua minggu setelah ditulis[5]. Peningkatan churn ini diproyeksikan dapat mencapai dua kali lipat pada proyek yang secara intensif memanfaatkan AI, yang mengindikasikan rapuhnya struktur kode serta rendahnya pemahaman terhadap kebutuhan sistem sejak tahap awal pengembangan.

Selain itu, aspek efisiensi algoritmik turut menjadi perhatian utama. LLM ditemukan cenderung merekomendasikan pendekatan penyelesaian masalah yang bersifat naif atau *brute-force* dengan kompleksitas waktu sebesar $O(n^2)$, alih-alih solusi yang lebih optimal dengan kompleksitas $O(n \log n)$. Pola ini berpotensi meningkatkan konsumsi sumber daya komputasi, khususnya pada lingkungan berbasis komputasi awan, sehingga dapat berdampak langsung pada performa sistem dan biaya operasional dalam skala besar[1].

4.5. Risiko Kepatuhan Lisensi

Tantangan terakhir yang teridentifikasi dalam literatur berkaitan dengan isu kepatuhan lisensi serta *provenance* atau asal-usul kode yang dihasilkan oleh LLM. Sejumlah studi menyoroti potensi terjadinya pelanggaran hak cipta secara tidak disengaja, di mana model dapat mereproduksi fragmen kode secara nyaris identik dari repositori sumber terbuka yang berada di bawah lisensi *copyleft*, seperti GNU General Public License (GPL), tanpa menyertakan atribusi yang semestinya[5].

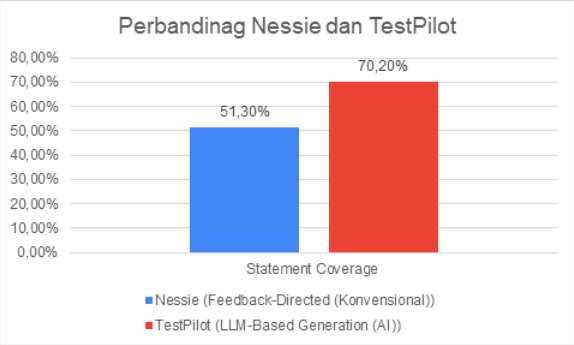
Meskipun tingkat kemunculan penjiplakan langsung relatif rendah pada fungsi-fungsi sederhana atau pola kode yang bersifat umum, risiko tersebut meningkat secara signifikan pada kasus algoritma kompleks dan implementasi yang bersifat unik. Dalam konteks pengembangan perangkat lunak proprietari, kondisi ini menciptakan ketidakpastian hukum yang serius dan berpotensi berkembang menjadi "bom waktu" litigasi apabila pelanggaran lisensi baru teridentifikasi pada tahap distribusi atau komersialisasi produk[12].

4.6. Strategi dan Pendekatan Verifikasi (RQ2)

Sebagai respons terhadap meningkatnya kompleksitas dan skala tantangan yang ditimbulkan oleh penggunaan kode generatif, komunitas riset mulai mengembangkan pendekatan verifikasi yang bergerak melampaui praktik pengujian manual tradisional. Evolusi ini ditandai dengan pergeseran menuju mekanisme verifikasi yang lebih otonom dan bersifat hibrida, dengan memadukan kemampuan analitis manusia dan kecerdasan buatan. Dalam literatur, sedikitnya empat pendekatan utama diidentifikasi sebagai strategi kunci untuk mengatasi berbagai dimensi risiko yang telah dipetakan pada RQ1.

4.7. Pendekatan Logika melalui AI-Driven Test Generation

Untuk mereduksi ketidakpastian logika pada kode yang dihasilkan LLM, sejumlah penelitian mengadopsi strategi yang dikenal sebagai *inversi peran*, yaitu memanfaatkan AI sebagai alat untuk memverifikasi keluaran AI itu sendiri. Dalam konteks ini, berbagai alat berbasis LLM, seperti *TestPilot*, dikembangkan untuk menghasilkan unit test secara otomatis dengan memanfaatkan pemahaman semantik dan konteks program[13]. Pendekatan ini menunjukkan keunggulan dibandingkan metode pengujian konvensional yang bergantung pada heuristik statis.



Gambar 2. Perbandingan Statement Coverage

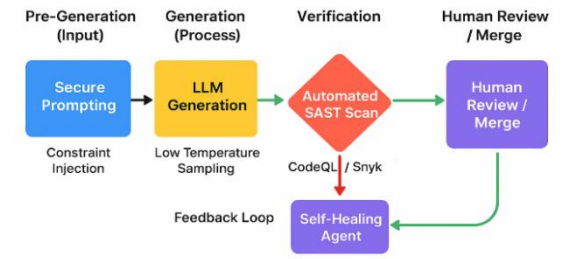
Perbandingan efektivitas antara pendekatan generatif dan teknik tradisional ditunjukkan melalui metrik *statement coverage*, sebagaimana disajikan pada Gambar 2. Metode berbasis LLM dilaporkan mampu mencapai nilai median *coverage* sebesar 70,2%, jauh lebih tinggi dibandingkan pendekatan *feedback-directed* konvensional seperti Nessie yang hanya mencapai 51,3%[13]. Temuan ini menegaskan bahwa kemampuan LLM dalam memahami struktur dan konteks kode berperan penting dalam meningkatkan kualitas pengujian, khususnya pada logika program yang kompleks.

Meskipun demikian, bebeapa literatur juga mencatat adanya tantangan konseptual yang belum sepenuhnya teratasi, salah satunya adalah *Oracle Problem*. Dalam kondisi ini, kasus uji yang dihasilkan dinilai valid secara sintaksis dan berhasil dieksekusi, namun memiliki kelemahan pada aspek asersi sehingga tidak selalu mampu mendeteksi kesalahan logika secara efektif[14]. Oleh karena itu, pengembangan mekanisme mitigasi terhadap permasalahan ini masih menjadi agenda penelitian yang relevan dalam verifikasi kode generatif.

4.8. Verifikasi Keamanan melalui Shift-Left Prompt Engineering

Dalam upaya meningkatkan keamanan kode generatif, praktik verifikasi kini semakin diarahkan ke tahap paling awal dari siklus pengembangan melalui pendekatan *shift-left*. Alih-alih mengandalkan deteksi kerentanan setelah kode dihasilkan, sejumlah penelitian mengusulkan penerapan *vulnerability-aware prompting*, yaitu teknik perancangan prompt yang secara eksplisit memuat batasan keamanan, prinsip pengkodean aman, serta contoh keluaran yang tidak diinginkan (*negative examples*)[10]. Pendekatan preventif ini bertujuan untuk membentuk ruang solusi model sejak awal agar selaras dengan praktik keamanan yang baik.

Hasil eksperimen menunjukkan bahwa penyisipan instruksi keamanan secara eksplisit dalam prompt mampu menurunkan secara signifikan probabilitas LLM menghasilkan kode yang mengandung kerentanan[15]. Strategi ini umumnya dipadukan dengan penggunaan *Static Application Security Testing* (SAST) kemudian diintegrasikan ke dalam alur kerja komprehensif sebagaimana diilustrasikan pada Gambar 3.



Gambar 3. Alur Kerja Pipeline Verifikasi Keamanan (Synthesis Model)

Seperti terlihat pada diagram di atas, model sintesis ini menggabungkan pencegahan di hulu (*Secure Prompting*) dengan validasi otomatis di hilir menggunakan *Static Application Security Testing* (SAST). Kode yang dihasilkan tidak langsung diserahkan ke manusia, melainkan dipindai terlebih dahulu oleh alat seperti CodeQL atau Snyk. Jika kerentanan terdeteksi, sistem memicu mekanisme *feedback loop* ke agen *Self-Healing* untuk melakukan perbaikan mandiri, memastikan hanya kode yang bersih yang diteruskan ke tahap tinjauan akhir (*Human Review*).

4.9. Peningkatan Keterpeliharaan melalui Self-Healing Agents

Untuk mengatasi tingginya tingkat *code churn* dan menurunnya keterpeliharaan kode, literatur terkini mulai mengeksplorasi pemanfaatan agen otonom yang dikenal sebagai *self-healing agents*. Pendekatan ini memanfaatkan kerangka kerja penalaran reflektif, seperti *Reflexion*, yang memungkinkan agen beroperasi dalam suatu siklus tertutup. Dalam siklus tersebut, agen secara berurutan menghasilkan kode, mengevaluasi kesalahan melalui analisis *stack trace*, serta melakukan perbaikan dan *refactoring* secara mandiri hingga memenuhi kriteria kualitas yang ditetapkan[11].

Temuan empiris menunjukkan bahwa agen semacam ini tidak hanya efektif dalam memperbaiki kesalahan fungsional, tetapi juga dapat diarahkan untuk melakukan optimasi struktural, termasuk pengurangan duplikasi kode dan peningkatan modularitas[16]. Dengan demikian, *self-healing agents* berpotensi menjadi mekanisme pendukung yang signifikan dalam menjaga kualitas dan keterpeliharaan kode generatif dalam jangka panjang.

4.10. Mitigasi Risiko Lisensi melalui Embeddings dan Watermarking

Untuk menjawab permasalahan kepatuhan lisensi dan penelusuran asal-usul kode (*provenance*), literatur terkini mengusulkan pemanfaatan teknik deteksi berbasis *embeddings*. Pendekatan ini bekerja dengan merepresentasikan potongan kode ke dalam ruang vektor semantik dan membandingkannya dengan basis data repositori open-source. Keunggulan utama metode ini terletak pada ketahanannya terhadap perubahan superfisial, seperti penggantian nama variabel atau penyesuaian format, sehingga tetap efektif dalam mengidentifikasi kemiripan struktural yang berpotensi melanggar lisensi[17].

Sebagai pendekatan komplementer, teknik *watermarking* pada model generatif mulai dikembangkan untuk menanamkan penanda identitas yang bersifat implisit ke dalam struktur kode yang dihasilkan. Watermark tersebut dirancang agar tidak memengaruhi fungsionalitas maupun performa program, namun tetap memungkinkan proses atribusi dan akuntabilitas ketika diperlukan[18]. Kombinasi antara pelacakan berbasis *embeddings* dan *watermarking* dipandang sebagai langkah strategis dalam mengurangi risiko hukum sekaligus meningkatkan transparansi dalam penggunaan kode generatif berbasis LLM.

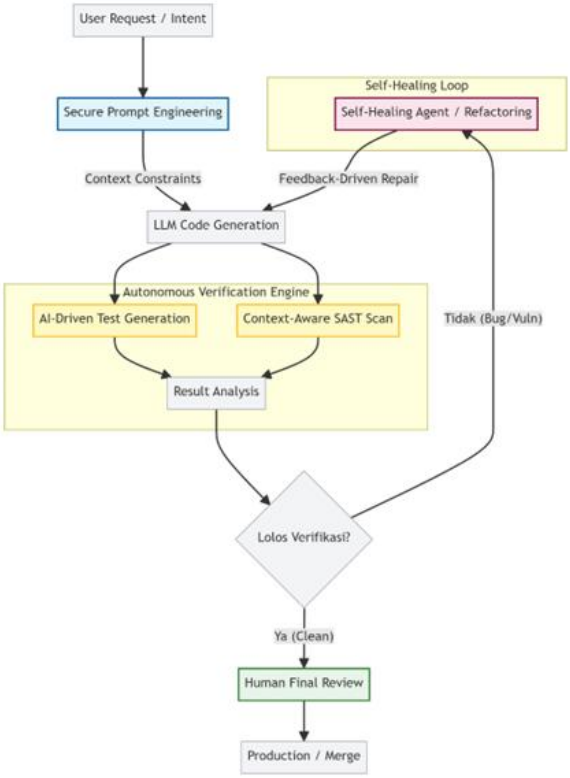
4.11. Sintesis: Siklus Verifikasi Otonom Hibrida

Berdasarkan pemetaan sistematis terhadap tantangan kualitas (RQ1) dan strategi mitigasi yang diusulkan (RQ2), penelitian ini menyusun suatu sintesis konseptual yang mengaitkan setiap kategori permasalahan dengan pendekatan verifikasi yang relevan. Hubungan kausal antara risiko dan mekanisme mitigasinya dirangkum secara terstruktur dalam Tabel 1, yang berfungsi sebagai matriks analitis untuk mengevaluasi efektivitas relatif masing-masing strategi verifikasi.

Tabel 2. Matriks Sintesis Tantangan Kualitas dan Strategi Verifikasi

Tantangan Utama (RQ1)	Strategi Mitigasi (RQ2)	Analisis Efektivitas	
		(+)	(-)
Logika & Fungsional (Halusinasi, Edge Cases)	AI-Driven Unit Testing (Inversi Peran)	Meningkatkan code coverage otomatis.	Risiko Bias Collusin antara pembuat & penguji.
Keamanan (Injeksi, Kredensial Lemah)	Shift-Left Prompting & Context-Aware SAST	Efisien mencegah kerentanan umum di hulu.	SAST sering gagal mendeteksi cacat logika bisnis.
Maintainability (Code Bloat, Duplikasi)	Self-Healing Agents (Refactoring)	Mengurangi beban perbaikan rutin.	Biaya komputasi tinggi; risiko regresi.
Lisensi (Plagiarisme)	Embedding-Based Detection	Tahan terhadap teknik obfusikasi.	Latensi tinggi untuk pemindaian real-time.

Sintesis tersebut selanjutnya diformalkan ke dalam sebuah kerangka kerja terpadu yang disebut "Siklus Verifikasi Otonom Hibrida". Kerangka ini dirancang untuk mereduksi *paradoks beban kognitif* yang muncul akibat penggunaan kode generatif, dengan mengalihkan aktivitas verifikasi rutin dan berulang ke sistem otonom, tanpa menghilangkan peran pengawasan manusia secara keseluruhan.



Gambar 4. Kerangka Kerja Siklus Verifikasi Otonom Hibrida.

Sebagaimana ditunjukkan pada Gambar 4, alur kerja dalam model ini diawali dengan mekanisme pencegahan di tahap hulu melalui *secure prompt engineering*, yang bertujuan membatasi ruang solusi model agar selaras dengan prinsip keamanan dan kualitas. Tahap ini diikuti oleh proses generasi kode oleh LLM yang kemudian langsung diproses oleh mesin verifikasi otonom secara paralel, meliputi *AI-driven test generation* dan pemindaian keamanan berbasis *context-aware SAST*. Hasil verifikasi dianalisis secara terintegrasi untuk menentukan kelayakan kode.

Apabila teridentifikasi cacat fungsional atau kerentanan keamanan, kode secara otomatis dialihkan ke dalam *self-healing loop*, di mana agen otonom melakukan perbaikan dan *refactoring* secara iteratif hingga memenuhi kriteria kualitas yang ditetapkan. Hanya artefak kode yang berhasil melewati seluruh tahapan verifikasi ini yang kemudian diteruskan ke tahap *human final review*. Dalam konfigurasi hibrida ini, peran manusia direposisikan sebagai auditor tingkat tinggi yang berfokus pada evaluasi arsitektur, konteks bisnis, dan keputusan desain strategis, sementara AI menangani kompleksitas teknis pada level sintaksis, fungsional, dan keamanan.

Dengan demikian, Siklus Verifikasi Otonom Hibrida tidak hanya menawarkan solusi teknis terhadap tantangan kualitas kode generatif, tetapi juga merepresentasikan redistribusi peran yang lebih efisien antara manusia dan mesin dalam proses rekayasa perangkat lunak modern.

5. KESIMPULAN DAN SARAN

Penelitian ini menyimpulkan bahwa integrasi *Large Language Models* (LLM) dalam rekayasa perangkat lunak menghadirkan paradoks fundamental antara lonjakan produktivitas dan degradasi integritas struktural, di mana kode generatif sering kali terjebak dalam fenomena "ilusi kebenaran" serta bertindak sebagai mekanisme replikasi kerentanan keamanan siber dari data pelatihan. Merespons risiko tersebut, paradigma verifikasi terbukti telah berevolusi dari tinjauan manual yang rentan terhadap beban kognitif menuju "Siklus Verifikasi Otonom Hibrida", sebuah kerangka kerja yang menyinergikan *secure prompting* di hulu, pengujian berbasis AI (*AI-Driven Testing*) dan SAST di hilir, serta agen *self-healing* untuk perbaikan mandiri. Untuk pengembangan penelitian selanjutnya, disarankan agar fokus dialihkan pada pengukuran dampak efisiensi energi dari kode generatif (*Green AI Gap*), pelaksanaan studi longitudinal untuk memvalidasi resistensi kode AI terhadap pembusukan perangkat lunak (*software rot*) dalam jangka panjang, serta inovasi pada teknik *watermarking* semantik dan analisis *provenance* guna menjawab tantangan kepatuhan lisensi dan atribusi yang belum terselesaikan sepenuhnya.

DAFTAR PUSTAKA

[1] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot," Feb. 2023, [Online]. Available: http://arxiv.org/abs/2302.06590

[2] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," in *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*, Institute of Electrical and Electronics Engineers Inc., Oct. 2022, pp. 1–5. doi: 10.1145/3524842.3528470.

[3] A. M. Dakhel, Y. Noller, and A. Noy, "GitHub Copilot AI: A Mixed-Methods Study on Efficacy and Perceived Usability," *ACM Transactions on Software Engineering and Methodology*, pp. 1–28, 2023.

[4] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," Dec. 2021, [Online]. Available: http://arxiv.org/abs/2108.09293

[5] W. Harding, M. Kloster, and C. Gitclear, "Coding on Copilot 2023 Data Shows Downward Pressure on Code Quality 150m lines of analyzed code + projections for 2024," 2024.

[6] J. Wang *et al.*, "Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval," 2024. doi: XXXXXXX.XXXXXXX.

[7] A. Nguyen-Duc *et al.*, "Generative Artificial Intelligence for Software Engineering -- A Research Agenda," Oct. 2023, [Online]. Available: http://arxiv.org/abs/2310.18648

[8] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 2785–2799. doi: 10.1145/3576915.3623157.

[9] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models," in *Conference on Human Factors in Computing Systems - Proceedings*, Association for Computing Machinery, Apr. 2022. doi: 10.1145/3491101.3519665.

[10] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024, doi: 10.1109/TSE.2024.3368208.

[11] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," Oct. 2023, [Online]. Available: http://arxiv.org/abs/2303.11366

[12] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How Secure is Code Generated by ChatGPT?," Apr. 2023, [Online]. Available: http://arxiv.org/abs/2304.09655

[13] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," Dec. 2023, [Online]. Available: http://arxiv.org/abs/2302.06527

[14] S. B. Hossain and M. B. Dwyer, "A Brief Survey on Oracle-based Test Adequacy Metrics," Feb. 2023, [Online]. Available: http://arxiv.org/abs/2212.06118

[15] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?," *Empir Softw Eng*, vol. 28, no. 6, p. 129, 2023, doi: 10.1007/s10664-023-10380-1.

[16] J. Yang *et al.*, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," Nov. 2024, [Online]. Available: http://arxiv.org/abs/2405.15793

[17] M. Saberi, V. S. Sadasivan, A. Zarei, H. Mahdavifar, and S. Feizi, "DREW : Towards Robust Data Provenance by Leveraging Error-Controlled Watermarking," Jun. 2024, [Online]. Available: http://arxiv.org/abs/2406.02836

[18] B. Li, Z. Fu, M. Zhang, P. Zhang, J. Sun, and X. Wang, "Efficient and Universal Watermarking for LLM-Generated Code Detection," Aug. 2025, [Online]. Available: http://arxiv.org/abs/2402.07518