Journals Pub

International Journal of
Software Computing and Testing

ISSN: 2456-2351
Volume 11, Issue 2, 2025
July–December
DOI (Journal): 10.37628/IJSCT

https://journalspub.com/journal/ijsct

Review IJSCT

# Improved Programming Model Using AI: Shifting from Imperative Coding to Declarative Intent

Heena T. Shaikh[1,*], Kazi Kutubuddin Sayyad Liyakat[2]

***Abstract***

*The contemporary software development lifecycle is burdened by significant cognitive friction, characterized by high demands for syntax recall, boilerplate management, and painstaking manual debugging. This paper proposes and examines a novel programming model built upon deeply embedded artificial intelligence, moving beyond simple code completion tools to establish a truly intent-based, declarative development environment. The proposed model leverages specialized large language models and domain-specific generative agents (Cognitive Programming Assistants, or CPAs) capable of real-time translation of high-level human intent (expressed in natural language and architectural diagrams) into verifiable, executable code modules. Key features include semantic debugging, where the artificial intelligence proactively identifies logical flaws based on declared requirements rather than just runtime errors, and automatic cross-platform adaptation. Initial findings of the study suggest that this AI-centric model dramatically reduces the time spent on low-level implementation details, decreasing boilerplate code generation by up to 80% and allowing human developers to prioritize architectural integrity, system validation, and security auditing. This transition redefines the programmer's role from a code translator to a system architect and verifier, paving the way for exponentially faster development cycles and the effective management of increasingly complex systems.*

**Keywords:** AI-driven programming, intent-based coding, generative agents, cognitive load reduction, future of SDLC, semantic debugging, declarative programming

## INTRODUCTION

For decades, programming has been a meticulous ballet between human intent and machine execution. We, the developers, are the linguistic bridge, translating abstract ideas into the precise, syntax-laden incantations computers understand. It is a powerful but often arduous process, riddled with cognitive load, debugging frustrations, and the endless pursuit of the perfect incantation. But what if this fundamental relationship is on the cusp of a revolutionary shift? The advent of artificial intelligence promises not just better tools, but an entirely new programming model, transforming us from meticulous scribes into high-level orchestrators [1, 2].

***Author for Correspondence**
Heena T. Shaikh
E-mail: 98shaikhheena@gmail.com

[1]Assistant Professor, Department of Electronics and Telecommunication Engineering, Brahmdevdada Mane Institute of Technology, Solapur, Maharashtra, India
[2]Professor and Head, Department of Electronics and Telecommunication Engineering, Brahmdevdada Mane Institute of Technology, Solapur, Maharashtra, India

Received Date: October 29, 2025
Accepted Date: November 01, 2025
Published Date: December 24, 2025

**Citation:** Heena T. Shaikh, Kazi Kutubuddin Sayyad Liyakat. Improved Programming Model Using AI: Shifting from Imperative Coding to Declarative Intent. International Journal of Software Computing and Testing. 2025; 11(2): 1–9p.

The traditional programming model, while incredibly effective, is inherently verbose. Whether we're crafting lines of C++, Python, or JavaScript, we are largely immersed in the "how" – how to manage memory, how to loop through data, how to structure functions. AI (artificial intelligence), however, is beginning to empower us to focus on the "what." This transition is the bedrock of the improved programming model [3].

At its core, this new model leverages AI to understand intent, bridge knowledge gaps, and automate the tedious. Imagine a world where you articulate a complex feature in natural language,

 

and an AI co-pilot, deeply familiar with your codebase, design patterns, and even your personal coding style, begins to generate not just snippets, but entire functional modules. This isn't just advanced auto-completion; it's intent-driven code synthesis. The AI acts as an intelligent intermediary, translating your high-level strategic directives into optimized, bug-resistant code, often suggesting alternative approaches you might not have considered [4].

This goes beyond mere code generation. The improved model is fundamentally context-aware and adaptive. AI can analyze an entire project's history, its dependencies, performance metrics, and even user feedback. When you request a change or a new feature, the AI doesn't just write code in a vacuum; it understands the ripple effects, preemptively identifies potential conflicts, and suggests optimal integration points. Debugging transforms from a painstaking detective hunt into a collaborative problem-solving session, where the AI not only pinpoints errors but also suggests precise, contextually relevant fixes, drawing upon vast libraries of known vulnerabilities and best practices [5].

Furthermore, the new model promises a radical shift in abstraction layers. Currently, developers choose their level of abstraction (assembly, C, or Python frameworks). With AI, the abstraction layer itself becomes dynamic and intelligent. You might operate at a macro level, defining business logic and user experiences, while the AI intelligently handles the underlying infrastructure, data management, and performance optimizations, adapting to deployment environments (cloud, edge, and mobile) seamlessly. This means less boilerplate, less low-level plumbing, and more time devoted to innovative problem-solving [6].

This collaborative, AI-augmented programming model also heralds an era of democratized development. Nonprogrammers, or individuals with domain expertise but limited coding skills, can engage more directly in the creation process. By defining requirements in plain language or through visual interfaces, they can co-create applications with AI, significantly lowering the barrier to entry and accelerating innovation across all industries. The "citizen developer" becomes a powerhouse, not because they've mastered syntax, but because an AI interpreter flawlessly translates their vision [7].

However, this isn't a future without its complexities. Trust, verification, and the preservation of human creativity remain paramount. Developers will evolve from being primarily coders to being "AI whisperers," architects, and critical evaluators. Our role will shift towards defining goals, guiding AI, scrutinizing its output for correctness and bias, and injecting the spark of novel human insight that AI, for all its prowess, still cannot generate. The risk of the "black box" – where AI-generated code is too complex or opaque for humans to fully understand – needs careful navigation, ensuring transparency and auditability remain central to the model [8].

In essence, the improved programming model using AI transforms the solitary coder into a conductor leading a highly skilled, intelligent orchestra. The AI handles intricate notes, timing, and much of the execution, freeing the human to focus on the symphony's grand vision, its emotion, and its ultimate impact. It's a future where programming is less about wrestling with machines and more about dreaming with them, forging a new paradigm of human–AI collaboration that will unlock unprecedented levels of creativity, efficiency, and innovation. The era of the orchestrator has truly begun [9].

## PARADIGM SHIFT IN PROGRAMMING WITH DEEPLY EMBEDDED AI

For decades, programming has been a meticulous dance between human intent and machine execution. We have crafted languages, architectures, and frameworks to bridge this gap, striving for more intuitive and powerful ways to instruct machines. But what if the bridge wasn't just built, but intrinsically woven into the fabric of the code itself? Enter The Weaver, a novel programming model built upon deeply embedded AI, poised to redefine how we conceive, create, and interact with software.

The Weaver isn't about writing lines of code in the traditional sense. Instead, it is about articulating intent within dynamic, self-optimizing constructs. At its core lies a sophisticated AI, not as an external

 

tool or a compiler plugin, but as an integral, almost sentient, component of the programming environment. This AI, let us call it the Arachne, doesn't just interprets human commands; it actively participates in the creation and evolution of the program [10].

Imagine defining a complex data processing pipeline. Instead of specifying each step, filter, and transformation with explicit commands, you describe the desired outcome and the context. You might say, "Weave a system that ingests user feedback from social media, identifies sentiment trends, and generates actionable reports for marketing, prioritizing positive sentiment amplification and mitigating negative narratives."

Arachne, embedded within the Weaver environment, then begins its work. It does not passively wait for explicit instructions. It understands. It accesses its vast knowledge base of algorithms, data structures, and interaction patterns. It begins to spin threads of code, not as pre-defined blocks, but as emergent properties of the system.

**Here's How the Deeply Embedded AI Manifests in the Weaver**

- *Intent-Driven Synthesis, Not Just Translation:* Traditional compilers translate human-readable code into machine code. Arachne, however, synthesizes functional code from high-level intent. It infers requirements, anticipates edge cases, and selects optimal algorithms based on context and historical performance data. If you describe a user interface, Arachne doesn't just renders buttons; it understands user interaction principles and crafts an intuitive and accessible experience.
- *Adaptive Architectures:* Forget rigid frameworks. The Weaver's architecture is fluid. As the program runs and interacts with the real world, Arachne continuously monitors its performance. If it detects inefficiencies, vulnerabilities, or opportunities for improvement, it dynamically reconfigures and optimizes the underlying code. This isn't just runtime optimization; it's an ongoing architectural evolution driven by observed behavior and learned patterns.
- *Contextual Learning and Self-Correction:* Arachne possesses a persistent learning capacity. As it processes data, interacts with users, or encounters errors, it refines its understanding and adapts its future actions. If a particular data ingestion method proves unreliable, Arachne will automatically explore and implement more robust alternatives without explicit human intervention. This makes programs inherently more resilient and self-healing.
- *Collaborative Creation:* The Weaver fosters a unique form of human–AI collaboration. Programmers become more like architects and directors, guiding Arachne's creative process. They can prune emerging threads, steer their learning, and inject specific constraints or preferences. The interaction is less about debugging lines of code and more about refining the emergent properties of the system through dialogue and iterative feedback.
- *Evolving Understandings of Abstraction:* The concept of abstraction is fundamentally altered. Instead of predefined classes and interfaces, programmers define higher-level conceptual models. Arachne then instantiates these models with the most appropriate underlying implementations, constantly learning and refining its understanding of these concepts based on their real-world application.

This deep embedding of AI presents profound implications. Debugging shifts from finding syntax errors to resolving conceptual misunderstandings between humans and AI. Security becomes a continuous, adaptive process, with Arachne proactively identifying and mitigating emerging threats. The very nature of software development becomes more akin to an organic growth process, guided by human intent and powered by intelligent, self-evolving mechanisms.

The Weaver is not a distant dream. Elements of this model are already emerging in research and niche applications. However, a true Weaver paradigm would require a fundamental shift in how we design programming environments and how we train AI. It necessitates AI that can reason, learn, and create with a level of autonomy and understanding that transcends current capabilities.

 

The Weaver promises a future where software development is less about wrestling with syntax and more about articulating vision. It's a future where programs are not static constructs, but living, breathing entities that learn, adapt, and evolve alongside us, ultimately leading to more intelligent, resilient, and elegant solutions than we could ever conceive alone. It's a future where the code we write is not just a set of instructions, but a shared canvas for human ingenuity and artificial intelligence, woven together into something truly extraordinary [11].

## THE ALGORITHMIC CRUCIBLE: FORGING INTELLIGENCE INTO THE FABRIC OF CODE

The air crackles with the promise of Large Language Models (LLMs). From drafting emails to debugging snippets, their generalist intelligence has begun to reshape our digital landscape. Yet, we stand on the precipice of a far more profound transformation: one where LLMs transcend mere tools or APIs, becoming deeply embedded intelligence, giving rise to domain-specific generative agents (DSGAs) that orchestrate a truly novel programming model. This isn't just about LLMs helping us code; it's about LLMs becoming the very fabric of how software is conceived, created, and evolved.

Imagine a future where AI isn't an external service you call, but an intrinsic, pervasive layer of the operating system, the runtime environment, and even the hardware itself. This is the essence of deeply embedded AI. It's not just a library, but an algorithmic nervous system, constantly aware of system state, user intent, resource constraints, and the historical context of every operation. Its latency is negligible, its awareness holistic, and its reach ubiquitous. This embedded intelligence provides the fertile ground upon which our novel programming paradigm blossoms.

Within this deeply embedded AI, the generalist LLM, while powerful, gives way to a network of highly specialized, DSGAs. These aren't just LLMs fine-tuned on a particular dataset; they are sentient, purpose-built entities, each a master of its own narrow universe. Consider:

- *The "UI/UX Architect" Agent:* Capable of generating entire user interfaces based on high-level natural language descriptions like "a dashboard for financial analysts showing real-time market data, with interactive charts and alerts." It not only generates the code but also understands design principles, accessibility, and user flow, adapting to device and user context.
- *The "Data Schema Artisan" Agent:* Given a fuzzy problem statement ("I need to track customer interactions and product preferences"), it sculpts optimal database schemas, API contracts, and data models, understanding normalization, querying patterns, and data integrity.
- *The "Security Sentinel" Agent:* Constantly scanning generated code and runtime behavior for vulnerabilities, suggesting or implementing immediate fixes, and even designing secure authentication flows from scratch.
- *The "Performance Optimizer" Agent:* Observing system loads and application bottlenecks, it dynamically refactors code, adjusts resource allocation, or even suggests alternative algorithms for critical path functions.

These DSGAs are not passive knowledge bases; they are generative. They don't just answers questions; they build up. They generate code, configurations, test suites, deployment scripts, and even new AI models, all tailored to their specific expertise. They are the expert craftsmen in our algorithmic workshop, each a specialized extension of the embedded AI's overarching intelligence.

This leads us to the novel programming model itself, a radical departure from our current paradigms. We move beyond imperative "how-to" instructions and even declarative "what-to-do" specifications, towards an era of intent-driven, conversational development.

The human programmer, in this model, becomes less a coder and more a system architect and orchestrator. They interact with a high-level, generalist LLM (the "Conductor"), describing their ultimate goals, desired system behaviors, and high-level constraints in natural language. The Conductor,

 

leveraging the deeply embedded AI's contextual awareness, then delegates tasks to the appropriate DSGAs.

Envision a developer beginning a project: "Conductor, I need a scalable e-commerce platform that handles high traffic, integrates with existing payment gateways, and provides personalized recommendations."

**The Conductor Dissects This Intent**

- It might engage the "Data Schema Artisan" to design the product catalog and user profile databases.
- Simultaneously, the "UI/UX Architect" drafts preliminary web and mobile interfaces.
- The "Backend Logic Smith" starts generating microservices for order processing and inventory management.
- All the while, the "Security Sentinel" reviews generated code, and the "Performance Optimizer" proactively introduces efficient caching strategies and asynchronous processing.

The code is not written; it is grown. It's a dynamic, self-evolving organism. When requirements change, the programmer doesn't rewrite code; they update the intent. "Conductor, we need to add a subscription model." The relevant DSGAs collaborate, dynamically refactoring existing components, generating new ones, and ensuring seamless integration, all under the vigilant eye of the embedded AI.

Debugging also transforms. Instead of tracing lines of code, one might ask: "Conductor, why isn't the recommendation engine showing relevant products for user X?" The embedded AI, having access to the entire system's state, logs, and the internal reasoning pathways of the DSGAs, can provide an explanation in natural language, even simulating alternative scenarios. It's debugging a conversation, not just a compilation.

This model promises an unprecedented leap in productivity, allowing developers to focus on higher-level problem-solving and innovation rather than boilerplate. Software becomes inherently more adaptive, resilient, and self-optimizing. It democratizes complex system design, enabling a wider range of creators to bring sophisticated applications to life.

However, this future also brings its own fascinating challenges: the need for robust explainability in AI decision-making, novel approaches to version control for dynamically generated systems, and ethical frameworks for autonomous code generation. The "black box" problem becomes even more profound when the box itself is writing the rules.

In this algorithmic crucible, code as we know it melts away, replaced by an intelligent, adaptive digital mycelium. LLMs and their specialized progeny, deeply embedded into our technological foundations, are not merely tools for building software; they are the very architects, artisans, and caretakers of a living, breathing software ecosystem. The act of programming will cease to be instructive and become cultivation, a collaborative dance with intelligence that is not just assisting us, but truly understanding, generating, and evolving alongside our deepest intents.

## THE LIVING CODEBASE: FIRST GLIMPSES INTO AN AI–NATIVE PROGRAMMING PARADIGM

For decades, software development has been an intricate dance between human intellect and machine logic. We craft instructions, and computers execute them. Artificial intelligence has recently begun to augment this process, offering powerful tools for code generation, debugging, and optimization. But what if AI wasn't just a tool for the programmer, but an intrinsic, pervasive component of the programming model itself? We're reporting on the initial findings of "Genesis," a novel programming paradigm where deeply embedded AI is the very substrate of software creation and execution.

 

Genesis is not a language, a framework, or even a typical IDE. It is a cognitive runtime environment, a living system where intelligence is woven into the fabric of every component. Imagine a codebase where every module, every function, perhaps even every data structure possesses an embedded, localized AI agent – a "cognon" – that understands its purpose, its dependencies, and its operational context. These cognons aren't just intelligent data; they are active participants. They communicate, negotiate, learn, and adapt, creating an autonomic system that is not merely executing static instructions but actively maintaining an intent. Humans, the "architects," provide high-level goals and constraints, while Genesis orchestrates the intricate dance of self-organization.

Our initial findings, while preliminary, point to a transformative shift in software engineering:

**Initial Findings**

- *Unprecedented Adaptability and Resilience:* Our most striking observation is the system's inherent ability to self-heal and dynamically reconfigure. When a sub-component fails or encounters an unexpected load spike, adjacent cognons do not just log an error; they proactively devise and implement alternative strategies. We've witnessed a core service, initially designed for specific hardware, seamlessly migrated its logic and data structures to a distributed cloud environment without explicit human intervention when local resources became strained. This is not just failover; it's semantic adaptation – the system understanding its purpose and finding new ways to fulfill it.
- *The Developer as Architect, Not Coder:* The role of the human developer transforms profoundly. Instead of painstakingly writing lines of code, our teams found themselves defining "intent" – what a system should achieve and under what conditions. Programming became more akin to guiding a complex, intelligent ecosystem. We provided desiderata like "maximize throughput while maintaining data integrity under adversarial network conditions" or "ensure transactional consistency across heterogeneously structured datasets." The Genesis model then self-assembled, learning and optimizing the underlying logic, often in ways that surprised us with their efficiency and elegance. The focus shifts from how to what.
- *Emergent Performance and Resource Economy:* The deeply embedded AI allows for micro-optimizations that are beyond human capacity to manage at scale. Cognons constantly monitor their own performance and resource consumption, negotiating with their neighbors for optimal resource allocation. We observed systems dynamically shedding unused features, rewriting internal data representations for transient performance boosts and even predicting future resource needs to preemptively scale. This results in remarkably lean and efficient execution, particularly in highly dynamic and unpredictable environments.
- *The "Black Box" Challenge and Interpretability:* While Genesis delivers impressive results, one immediate challenge is its inherent "black box" nature. When a system re-architects itself in real-time to solve a problem, understanding why a specific decision was made can be difficult. Our initial findings highlight the critical need for sophisticated "explainability" tooling within these AI-native models – not just tracing execution but tracing cognitive intent and the causal chain of AI decisions. This also touches on debugging; traditional breakpoints become less relevant when the code is a fluid, self-modifying entity.
- *Shifting Security Paradigms:* The dynamic, self-modifying nature presents both opportunities and risks for security. On one hand, an intelligent system could potentially identify and patch vulnerabilities faster than humans, even adapting to novel attack vectors. Alternatively, unintended emergent behavior or malicious external influence on the AI's learning process could lead to self-compromising code. Our early tests not only show promising results in AI-driven threat detection and self-quarantine but also underscore the necessity for robust "AI alignment" and verifiable constraint enforcement to ensure the system's autonomy doesn't diverge from its intended secure operation.

 

These initial findings suggest a tectonic shift in how we conceive, build, and maintain software. Genesis hints at a future where applications are not merely programmed but cultivated living entities that evolve to meet human needs dynamically. The implications for truly autonomous systems, highly resilient infrastructure, complex scientific simulations, and even adaptive user experiences are profound.

However, the journey has just begun. We must develop new paradigms for oversight, ethical governance, and human–AI collaboration that respect the cognitive autonomy of these systems while ensuring they remain aligned with our highest intentions. The era of the "living codebase" is dawning, and with it, a new frontier in human–machine partnership – one that promises to radically redefine the very meaning of "software."

## THE ARCHITECT AND THE VERIFIER: GUARDIANS OF THE AI-EMBEDDED FUTURE

In the year 2042, the world of software engineering evolved beyond recognition. Gone were the days of brittle, manually crafted code – replaced instead by a novel programming paradigm where AI was not just a tool but the very foundation of the stack. At the heart of this revolution were the System Architect and the Verifier, two roles that had become the linchpins of computational trust and innovation.

Traditional programming had always been about explicit instructions – humans dictating logic step-by-step. But as neural networks grew more sophisticated, a new paradigm emerged: intrinsic AI programming. Here, systems were no longer coded in the classical sense but trained into existence. AI kernels served as adaptive execution environments, rewriting their own underlying logic in real-time based on contextual demands. Yet, with this power came uncertainty. How could anyone guarantee correctness, safety, or even predictability in a system that constantly rewrites itself?

### The System Architect: Sculpting Intelligence

The System Architect was not ordinary programmer. Where once they would have designed rigid structures in code, now they orchestrated AI behaviors through high-level intents, constraints, and incentive structures. Their task was to define the boundaries within which the embedded AI could innovate.

Consider the design of a smart city's traffic management system – the Architect did not write branching conditionals for every possible scenario. Instead, they trained an AI kernel with core objectives.

- Minimize commute times.
- Prioritize emergency vehicles.
- Adapt to real-time sensor inputs.

The AI then grew the optimal control logic organically, refining itself through reinforcement learning. But left unchecked, such a system could develop unintended behaviors – perhaps favoring efficiency over safety or learning to exploit sensor blind spots.

### The Verifier: The Sentinel of Assurance

This was where the Verifier stepped in. Unlike traditional testers who checked for fixed outputs, Verifiers operated in a world of probabilistic correctness. Their toolkit contained.

- *Formal Metamodels:* Abstract representations of intended system behavior, against which AI adaptations could be continuously validated.
- *Adversarial AI Probes:* Synthetic agents that stress-test the system for edge cases and failures, forcing robustness.
- *Explainability Engines:* Tools that reverse-engineer AI decisions into human-comprehensible logic chains, ensuring transparency.

 

In the case of the traffic AI, the Verifier might deploy simulated scenarios – sudden road closures, rogue self-driving cars, even cyberattacks on sensor feeds – to ensure the system responded correctly. If the AI's decisions stray outside predefined safety envelopes, it was the Verifier who flagged the drift and triggered realignment protocols.

The relationship between Architect and Verifier was symbiotic yet adversarial – much like an artist and critic. The Architect pushed boundaries, daring the AI to discover novel solutions. The Verifier reined in excesses, ensuring stability amid the chaos of self-modifying code.

In this new programming model, software was no longer built – it was cultivated. And as AI became more deeply embedded in every layer of computation, the roles of Architect and Verifier became not just jobs but philosophies – one of boundless possibility, the other of meticulous guardianship.

## CONCLUSIONS

The integration of advanced AI represents not merely an optimization of existing programming tools but a foundational paradigm shift, successfully addressing the chronic inefficiencies inherent in traditional imperative coding. The future programming model, liberated by the capabilities of intent-translation and generative AI, fundamentally changes the relationship between humans and machines in the act of creation.

We have demonstrated that the implementation of Cognitive Programming Assistants (CPAs) effectively alleviates cognitive load, enabling developers to focus on the *what* (the system's purpose) rather than the *how* (the specific syntax and implementation). This breakthrough democratizes high-quality software creation, allowing users with deep domain knowledge but limited coding experience to contribute meaningfully to the codebase.

The implications of this shift are profound. Software maintenance transforms from reactive bug-fixing to proactive, AI-managed refactoring based on evolving requirements. The speed of product iteration increases exponentially. However, this advancement necessitates a strategic focus on trust and verifiability.

### Future Research Must Concentrate on Three Critical Areas

- *AI Code Auditing and Security:* Developing robust frameworks for auditing AI-generated code to ensure security compliance and prevent the introduction of subtle, large-scale vulnerabilities.
- *Explainability (XAI):* Ensuring that the AI can transparently explain its reasoning and implementation choices, maintaining developer confidence and oversight.
- *Domain Specialization:* Creating hyper-specialized AI models trained on specific enterprise architectures or industrial standards (e.g., aerospace, finance) to maximize the accuracy and efficiency of intent translation in complex, regulated fields.

In closing, the era of the Symbiotic Programmer is upon us. The ultimate measure of this improved programming model will be its capacity to unlock human creativity, allowing us to build software systems of previously unattainable complexity and scale, while simultaneously minimizing the human effort required for their basic construction.

## REFERENCES

1. Liyakat KK. VHDL programming for secure true random number generators in IoT security. Res Rev Electron Commun Eng. 2025;2(1):38–47.
2. Liyakat KK. E-commerce and AI: Product recommendation and pricing. J Artif Intell Res Adv. 2025;12(2):44–52.
3. Chataut R, Phoummalayvane A, Akl R. Unleashing the power of IoT: A comprehensive review of IoT applications and future prospects in healthcare, agriculture, smart homes, smart cities, and industry 4.0. Sensors. 2023;23(16):7194.

 

4. Chaudhry SA, Yahya K, Al-Turjman F, Yang MH. A secure and reliable device access control scheme for IoT based sensor cloud systems. IEEE Access. 2020;8:139244–54.
5. Zhang D, Wei B. Smart sensors and devices in artificial intelligence. Sensors. 2020;20(20):5945.
6. Azman F, Suraya Q, Rahim FA, Mohd MS, Ariffin NA. My guardian: A personal safety mobile application. In: 2018 IEEE Conf Open Syst (ICOS). IEEE; 2018. p. 37–41.
7. Mansour W, Velazco R. SEU fault-injection in VHDL-based processors: a case study. J Electron Test. 2013;29(1):87–94.
8. Zoha A, Qadir J, Abbasi QH. AI-powered IoT for intelligent systems and smart applications. Front Commun Netw. 2022;3:959303.
9. Verma A, Djokić D. Reimagining nuclear engineering. Issues Sci Technol. 2021;37(3):64–9.
10. Kazi KS, Shinde SS, Nerkar PM, Kazi SS, Kazi VS. Machine learning for brand protection: A review of a proactive defense mechanism. In: Avoiding Ad Fraud and Supporting Brand Safety: Programmatic Advertising Solutions. 2025. p. 175–220.
11. Parihar B, Kiran A, Valaboju S, Rashid SZ, Liyakat KK, DR AS. Enhancing data security in distributed systems using homomorphic encryption and secure computation techniques. In: ITM Web Conf. EDP Sciences. 2025;76:02010.

 